



HAL
open science

Design of a Simulation Model of Multi-Agent Systems, and its Parallel Algorithmic and Implementation on Shared-Memory MIMD Computers: ParSSAP Model

Eugen Dedu

► **To cite this version:**

Eugen Dedu. Design of a Simulation Model of Multi-Agent Systems, and its Parallel Algorithmic and Implementation on Shared-Memory MIMD Computers: ParSSAP Model. Networking and Internet Architecture [cs.NI]. Université de Versailles-Saint Quentin en Yvelines, 2002. English. NNT: . tel-00071184

HAL Id: tel-00071184

<https://theses.hal.science/tel-00071184>

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PRiSM

Université de Versailles
Saint-Quentin-en-Yvelines

SUPÉLEC

**Conception d'un modèle
de simulation de systèmes multi-agent,
et de son algorithmique et implantation parallèle
sur architectures MIMD à mémoire partagée :
modèle ParSSAP**

**Design of a Simulation Model of Multi-Agent Systems,
and its Parallel Algorithmic and Implementation
on Shared-Memory MIMD Computers:
ParSSAP Model**

(In English, with in-depth abstract in French)

Thèse présentée pour obtenir le titre de docteur
de l'Université de Versailles Saint-Quentin-en-Yvelines,
spécialité informatique, par

Eugen Dedu

Composition du jury :

Michel Tréhel	rapporteur	LIB UFC, Besançon
Jean-Paul Sansonnet	rapporteur	LIMSI CNRS, Orsay
Claude Timsit	directeur de thèse	PRiSM UVSQ, Versailles
Stéphane Vialle	co-encadrant de thèse	SUPÉLEC, Metz
Traian Muntean	examineur	LIM U. Aix-Marseille II
Jens Gustedt	examineur	INRIA Lorraine LORIA, Nancy

Soutenue le 8 mars 2002

Remerciements

Je voudrais tout d'abord remercier mes deux encadrants :

M. Stéphane Vialle, enseignant-chercheur à Supélec, pour sa rigueur, la liberté qu'il m'a laissée dans mes choix, les nombreuses relectures et remarques de qualité pour améliorer ce mémoire de thèse et, plus généralement, pour son aide scientifique.

Professeur Claude Timsit, de l'Université de Versailles Saint-Quentin-en-Yvelines, pour ses précieux conseils, son aide et sa gentillesse.

Mes sincères remerciements vont à mes rapporteurs, Professeur Michel Tréhel et Jean-Paul Sansonnet Directeur de Recherches, pour l'honneur qu'ils m'ont fait en rapportant ma thèse.

Je remercie également Professeur Traian Muntean et Jens Gustedt Directeur de Recherches pour avoir examiné ma thèse.

Je ne peux pas oublier les discussions scientifiques fructueuses avec mes collègues et amis Hocine Bekkouche, Malik Hocine et Michel Narozny, de Supélec, que je remercie vivement.

Je remercie particulièrement Hervé Frezza, enseignant-chercheur à Supélec, pour ses discussions scientifiques et son amitié, et Patrick Mercier, administrateur système de Supélec, pour m'avoir fourni un cadre de travail de grande qualité.

Je remercie l'ensemble du personnel de Supélec, ainsi que celui de PRiSM de l'Université de Versailles, trop nombreux pour les citer ici.

Je remercie le Centre Charles Hermite, Nancy, pour l'accès et le support parfaits de son supercalculateur Origin 2000, et la Région Lorraine pour son support financier.

Enfin, je remercie de tout mon cœur ma petite famille, loin de chez moi et proche en même temps. . .

Contexte scientifique et institutionnel

Le travail de cette thèse a été fait à Supélec, campus de Metz, dans l'équipe ERSIDP¹ (Equipe de Recherche sur les Systèmes Intelligents Distribués et Parallèles). Il a été fait sous la coordination de M. Stéphane Vialle, enseignant-chercheur à Supélec, et de Professeur Claude Timsit du laboratoire PRiSM, Université de Versailles Saint-Quentin-en-Yvelines.

Cette thèse s'inscrit dans la direction de recherche de l'équipe ERSIDP, qui s'intéresse à la création d'outils permettant de faciliter l'écriture des applications distribués sur les architectures parallèles modernes (voir la section 2.2).

Cette thèse a permis l'écriture de plusieurs articles et présentations, présentés dans l'annexe C.

¹<http://www.es-metz.fr/~ersidp>.

Table des matières

I	Introduction et résumé étendu	13
1	Introduction et résumé étendu	15
1.1	Motivations et introduction	15
1.2	Principes des systèmes multi-agent	17
1.3	Principes du calcul parallèle	19
1.4	Exemples d'implantations de systèmes multi-agent	23
1.5	Définition d'un modèle de simulation de systèmes multi-agent	25
1.6	Conception d'un algorithme de vision	28
1.7	Conception d'algorithmes de propagation de potentiel	33
1.8	Caractéristiques et implantation de ParSSAP	39
1.9	Domaines d'application et performances globales	43
1.10	Bilan et travaux futurs	47
II	Mémoire de thèse (en anglais)	49
	Table des matières	51
2	Motivations et introduction	57
2.1	Motivations	57
2.2	Famille des langages ParCeL	59
2.3	Caractéristiques de notre modèle de programmation	59
2.4	Caractéristiques de notre outil de développement	60
2.5	Hypothèses prises dans notre modèle et outil de développement	61
2.6	Plan de la thèse	62
3	Principes des systèmes multi-agent	63
3.1	Principaux intérêts du paradigme multi-agent	63
3.2	Apperçu général des systèmes multi-agent	65
3.2.1	Large définition du concept d'agent	65
3.2.2	Types d'agents	66
3.2.3	Coopération <i>vs.</i> antagonisme	67
3.3	Applications	68
3.4	Modélisation de sociétés d'agents situés	69

3.4.1	Modélisation de l'espace et du temps	69
3.4.2	Modélisation des agents	70
3.4.3	Modélisation des percepts des agents	71
4	Principes du calcul parallèle	75
4.1	Introduction	75
4.1.1	Apports du parallélisme	75
4.1.2	Mesure des performances des programmes parallèles	76
4.1.3	Possibles sources de pertes de performances en parallélisme	78
4.1.4	Sources séquentielles de pertes de performances en parallélisme	78
4.1.5	Sources parallèles de pertes de performances en parallélisme	80
4.1.6	Complexité de la programmation parallèle	83
4.1.7	Brève prospective sur le futur du parallélisme	85
4.2	Architectures des ordinateurs parallèles	86
4.2.1	Taxonomie de Flynn	86
4.2.2	Machines MIMD	87
4.2.3	Influence de la mémoire cache sur les performances du parallélisme	90
4.2.4	Règles de développement pour l'optimisation des accès au cache	92
4.3	Stratégies de programmation parallèle	93
4.3.1	Sources de parallélisme	93
4.3.2	Niveaux d'abstraction de la programmation parallèle	94
4.3.3	Paradigmes de programmation parallèle	95
4.4	Paradigmes de programmation parallèle à mémoire partagée	96
4.4.1	Liste des paradigmes existants	96
4.4.2	Techniques de synchronisation	97
4.5	Génération parallèle et déterministe de nombres aléatoires	97
4.6	Entrées/sorties parallèles sur disque dans les systèmes multi-agent	99
5	Implantations parallèles de systèmes multi-agent	101
5.1	Simulateurs de systèmes multi-agent séquentiels	101
5.1.1	Pengi , de Agre et Chapman	101
5.1.2	Pengi amélioré, de Drogoul, Ferber et Jacopin	102
5.2	Simulateurs de systèmes multi-agent parallèles	103
5.2.1	Simulateur PIOMAS considérant les incertitudes et les erreurs	103
5.2.2	Simulateur massif BioLand	104
5.3	Environnements de développement des SMA parallèles	106
5.4	Conclusions	107

6	Le modèle ParSSAP de simulation de systèmes multi-agent	109
6.1	Composants du modèle	109
6.2	Comportement des agents	111
6.3	Simulation de la simultanéité des actions	112
6.4	Moteur d'exécution	112
6.5	Informations sur l'état et l'évolution du système	114
6.6	Transparence du parallélisme dans le modèle	115
6.7	Conclusions	115
7	Algorithmique parallèle du percept de vision	117
7.1	Propriétés du percept de vision	117
7.2	Définition de la topologie du champ de visibilité	118
7.3	Algorithme de vision	120
7.4	Difficultés introduites par les obstacles	121
7.5	Algorithmes de traçage de lignes et de lignes épaisses	122
7.5.1	Algorithmes de traçage de lignes	122
7.5.2	Algorithmes de traçage de lignes épaisses	124
7.6	Notre algorithme de traçage de lignes épaisses	126
7.7	Optimisations de l'algorithme de vision	128
7.8	Problèmes soulevés par la parallélisation de l'algorithme de vision	131
7.9	Modèles d'espace mémoire et de temps d'exécution	133
7.9.1	Modèle d'espace mémoire	133
7.9.2	Modèle de temps d'exécution	134
7.10	Performances de l'algorithme de vision	135
7.10.1	Conditions expérimentales	135
7.10.2	Implantation de l'algorithme de vision	136
7.10.3	Influence des allocations mémoire parallèles	137
7.10.4	Résultats des mesures de performances	141
7.10.5	Influence des caches	145
7.10.6	Influence des paramètres de vision	146
7.10.7	Performances globales de l'algorithme de vision	153
7.11	Conclusions	154
8	Algorithmique parallèle de la propagation et de la perception des champs de potentiel	157
8.1	Propriétés du gradient de potentiel	157
8.2	Définition de la topologie du champ de potentiel	159
8.3	Difficultés introduites par les obstacles	159
8.4	Modèle de propagation par vagues	159
8.4.1	Propagation par vagues d'une ressource	160
8.4.2	Superposition de champs de potentiel	162
8.5	Vue globale des implantations de la propagation par vagues	162
8.6	Méthodes séquentielles	164

8.6.1	Méthodes récursives	164
8.6.2	Méthodes itératives	167
8.6.3	Méthodes utilisant des distances aux ressources	170
8.6.4	Autres méthodes	175
8.6.5	Comparaison théorique des méthodes séquentielles	176
8.7	Méthodes de parallélisation	176
8.7.1	Décomposition fixe du domaine	177
8.7.2	Décomposition variable du domaine	184
8.7.3	Décomposition de données réalisée par exclusions mutuelles	187
8.7.4	Décomposition de données avec environnement privé sur chaque processeur	188
8.7.5	Comparaison théorique des méthodes parallèles	192
8.8	Performances des algorithmes de propagation par vagues	192
8.8.1	Conditions expérimentales	192
8.8.2	Implantation des algorithmes de propagation par vagues	192
8.8.3	Performances de chacune des méthodes	194
8.8.4	Comparaison détaillée des meilleures méthodes	201
8.8.5	Meilleure méthode finale	210
8.9	Propagation inexacte du potentiel	211
8.10	Conclusions	213
9	Caractéristiques et implantation de ParSSAP	215
9.1	Caractéristiques de notre bibliothèque	215
9.1.1	Caractéristiques générales	215
9.1.2	Informations sur l'état et l'évolution du système	216
9.1.3	Choix techniques de parallélisation	217
9.2	Gestion de la simultanéité et du parallélisme dans la bibliothèque	218
9.2.1	Structures internes	218
9.2.2	Flot d'exécution	219
9.2.3	Aspect aléatoire et reproductibilité des simulations	222
9.3	Informations sur la compilation	224
9.3.1	Compilation de la bibliothèque	224
9.3.2	Compilation d'une application	225
9.4	Exécution d'une simulation du point de vue de l'utilisateur	225
9.5	Exemple « Hello world »	225
9.5.1	Implantation et explications	226
9.5.2	« Hello world » amélioré	227
9.6	« Collectionneur de diamants », un exemple plus complexe	228
9.6.1	Spécifications	228
9.6.2	Implantation et explications	229
9.7	« Robots transporteurs », un exemple avec des champs de potentiel	231
9.7.1	Initialisation du système	231

9.7.2	Fonction utilisateur de fin de cycle	233
9.7.3	Fonctions relatives aux ressources	233
9.7.4	Fonction de comportement des agents	233
9.7.5	Simulation	234
9.8	Conception générique d'une application	236
9.9	Conclusions	236
10	Domaines d'application et performances globales	237
10.1	Jeu de la vie de Conway	237
10.2	Simulateur multi-consommateur	240
10.3	Conclusions	257
11	Bilan et travaux futurs	259
11.1	Bilan de notre travail	259
11.2	Travaux futurs	261
A	Manuel de référence de ParSSAP	263
A.1	Considérations générales	263
A.2	Structures et types de données	264
A.3	Fonctions de commencement et de fin de simulation	265
A.4	Fonctions de création de l'environnement	266
A.5	Fonctions de configuration de la simulation	268
A.6	Fonctions d'entrée/sortie	271
A.7	Fonctions de création et de destruction dynamique d'agents	273
A.8	Fonctions d'information du système	274
A.9	Exemple complet de fichiers système	276
A.10	Exemple complet de fichier statistiques	278
B	Implantation de l'algorithme de traçage de lignes épaisses	281
C	Publications écrites pendant cette thèse	283
C.1	Conférences internationales avec comité de lecture	283
C.2	Conférence nationale avec comité de lecture	284
	Bibliographie	285

Première partie

Introduction et résumé étendu

Chapitre 1

Introduction et résumé étendu

Ce chapitre est un résumé en français du mémoire de thèse. Pour chaque point, plus d'informations se trouvent dans le corps du mémoire (en anglais), qui constitue la deuxième partie de ce document.

1.1 Motivations et introduction

Motivations

De grands systèmes existent où une solution optimum doit être trouvée, par exemple le trafic routier. Une voie prometteuse pour résoudre ces problèmes est le calcul distribué, où la connaissance du système est éparpillée dans tout le système.

Les systèmes multi-agent (SMA) sont un modèle approprié pour résoudre ces problèmes. Un agent est une entité *autonome* qui évolue dans un environnement ; elle le perçoit et agit sur lui. Chaque agent a un comportement et un but propres. Dans certains cas, des comportements simples d'agents peuvent générer des comportements globaux très efficaces, concept désigné sous le nom d'*émergence*. Un tel exemple est l'organisation des fourmis.

Actuellement, il n'y a pas de théorie générale ou efficace pour ces systèmes d'auto-organisation. De plus, beaucoup de paramètres influencent l'efficacité d'un tel système. Une approche classique pour comprendre les règles qui régissent ces systèmes est de faire beaucoup d'expérimentations et d'en interpréter les résultats.

Cependant, les simulations prennent du temps pour être implantées et exécutées. Leur exécution sur des machines parallèles permet de réduire leur temps d'exécution, mais la programmation parallèle est plus difficile et rend plus longue encore leur implantation. Cette difficulté devient plus accrue dans les SMA, où chaque agent a son propre comportement et se déplace dans l'environnement ; les calculs sont donc irréguliers et non localisés.

Actuellement, il n'y a pas d'outils génériques et faciles pour simuler de tels systèmes. Dans ce contexte, la contribution de cette thèse s'articule autour

des trois axes suivants :

1. Fournir un modèle de simulation de SMA situés, aptes à la parallélisation ;
2. Faire un travail algorithmique dans les SMA, notamment dans l'implantation et l'analyse des algorithmes de vision et de propagation par vagues, ainsi que dans la totale reproductibilité de simulations aléatoires ;
3. Fournir une implantation qui valide le modèle et qui permet de développer *rapidement* des simulations *rapides à l'exécution*.

Famille des langages ParCeL

Notre contribution n'est pas singulière. Elle est une continuation des travaux menés dans notre équipe depuis 1989. L'intérêt des langages **ParCeL** (**Parallel Cellular Language**) est d'offrir des outils parallèles permettant d'écrire facilement des applications de calcul distribué sur des machines parallèles modernes. Le modèle (avec son implantation) que nous présentons dans cette thèse est le cinquième dans cette catégorie (**ParCeL-5**).

Notre modèle de programmation

Dans notre modèle discret, l'environnement est bi-dimensionnel rectangulaire. Chaque entité de simulation occupe une case. Il y a trois types d'entités : obstacles, ressources et agents. Les obstacles empêchent les agents d'entrer dans certaines cases. Les ressources ont comme rôle principal d'attirer les agents. Pour cela, elles peuvent propager des champs de potentiel, perçus par les agents. Ces champs de potentiels contournent les obstacles et peuvent évoluer pendant la simulation. Les ressources peuvent également contenir des objets qui peuvent être transportés par les agents.

Les agents sont les entités mobiles de la simulation. Les agents ont un comportement propre et une mémoire propre associée. Ils peuvent naître et disparaître dynamiquement, pendant la simulation. À part la perception des potentiels introduite plus haut, ils peuvent également être influencés par la vision, qui leur permet ou non d'obtenir des informations sur une case distante. Pour la flexibilité du modèle, les agents peuvent avoir aussi une perception globale de l'environnement.

La simulation est basée sur des cycles. Pendant chaque cycle, le comportement de chaque agent est exécuté et son action est accomplie (si elle est possible). Des exemples d'actions sont le mouvement et la prise et la dépose d'objets.

Plusieurs types de statistiques sur la simulation peuvent être sauvegardées dans des fichiers pour une analyse post-mortem.

Caractéristiques et hypothèses de notre outil

Nous fournissons également un outil, sous la forme d'une bibliothèque, basé sur ce modèle de simulation, qui implante la plupart des spécifications du modèle. Additionnellement, la bibliothèque est parallèle, mais cela est quasi-entièrement caché à l'utilisateur.

Pour notre outil nous avons dû prendre des hypothèses simplificatrices. Par exemple, le potentiel décroît linéairement avec la distance à la ressource et la vision est empêchée seulement par les obstacles. Ces hypothèses ne sont pas toujours réelles, mais elles sont suffisamment bonnes pour ces simulations et suffisamment simples pour avoir de bonnes performances en exécution.

Plan du mémoire de thèse

Les deux sections suivantes présentent les principes des SMA et du parallélisme. La partie bibliographique finit avec une section sur des simulateurs parallèles de SMA. Ensuite, le travail principal de notre thèse commence avec la définition du modèle de simulation de SMA. Deux autres sections algorithmiques présentent la conception des algorithmes de vision et de propagation du potentiel. Deux autres sections sont dédiées à la bibliothèque dans son ensemble : ses caractéristiques et ses domaines d'application. Les classiques bilan et perspectives finissent ce résumé. Le manuel de référence de la bibliothèque peut être trouvé dans la thèse.

1.2 Principes des systèmes multi-agent

Intérêts du paradigme multi-agent

Plusieurs chercheurs considèrent que le paradigme multi-agent est un pas en avant dans la programmation. Pour Gasser [40], par exemple, la puissance des agents consiste dans leur autonomie et dans leur comportement dirigé vers un but. Sipper [75] voit le futur dans la « programmation cellulaire » (caractéristique aux agents) guidée par trois principes : simplicité, parallélisme massif et localité. Jennings [54] considère ce paradigme pour sa robustesse, son extensibilité et sa généralisation, mais évoque deux points difficiles : l'interaction imprévisible des agents et la difficulté de prédiction du comportement global du système. Finalement, Ferber [33, 32] cite plusieurs domaines d'application appropriés aux SMA, comme l'intelligence artificielle distribuée et la simulation de populations réelles (populations de fourmis par exemple).

Introduction au concept d'agent

Le concept d'agent a actuellement un sens très large. Cette diversité vient du fait qu'un agent est défini comme une entité *autonome* qui se trouve

dans un environnement, qui communique avec lui par ses percepts et ses actions, qui a un comportement propre et qui peut communiquer avec d'autres agents [32]. Dans ce contexte, un agent peut être aussi un programme informatique qui voyage d'ordinateur en ordinateur, que l'équivalent d'un objet de la programmation orienté objet ou bien l'entité simulant un être vivant (un oiseau par exemple).

Un type particulier de SMA est le SMA situé, où l'environnement possède une métrique et des contraintes topologiques existent, comme l'interdiction pour un agent d'entrer dans un mur (cohérence du système) ou l'interdiction d'avoir deux agents dans une même place en même temps (simultanéité).

Applications des systèmes multi-agent

Ferber [32] fait l'état de l'art des architectures, des environnements et des applications des SMA. Dans l'industrie, les SMA sont utilisés dans des centres de production d'énergie [85], la médecine et la reconnaissance de formes [32]. En recherche, ils sont utilisés pour simuler des équipes sportives [78], particulièrement le championnat RoboCup¹.

Modélisation des systèmes multi-agent

Nous décrivons brièvement dans cette section la modélisation des composants intervenant dans les SMA.

L'environnement est généralement discret et bi-dimensionnel (2D). Il peut être borné, comme dans *Pengi* [4, 30], ou torique, comme dans *BioLand* [86]. Notre modèle permet la simulation des deux types.

Dans notre modèle le temps aussi est discret, basé sur des cycles de simulation. Pour prendre en compte la simultanéité des actions, un cycle est divisé en plusieurs sous-cycles. Par exemple, *PIOMAS* [15] utilise 2 sous-cycles et *BioLand* utilise 7 sous-cycles. Notre outil utilise 5 sous-cycles : (1) perception et décision, (2) résolution des conflits, (3) action, (4) exécution de la fonction utilisateur et (5) mise à jour des percepts et de l'environnement.

Les actions faites par les agents peuvent être exactes, comme dans notre modèle et dans *Pengi*, ou inexactes, comme dans *PIOMAS*. Certaines actions, comme le mouvement, peuvent engendrer des conflits spatiaux. Dans notre modèle, ces conflits sont automatiquement résolus par le simulateur. D'autres actions possibles sont de prendre des objets, de déposer des objets et de pousser des objets. L'implantation actuelle de notre modèle permet de prendre et de déposer des objets.

Dans notre modèle les agents peuvent être créés et détruits dynamiquement, mais la génétique des agents n'est pas prise en compte.

Plusieurs percepts d'agents existent dans la littérature [31]. Nous en présentons deux, qui sont aussi implantés dans notre bibliothèque : l'odorat et

¹RoboCup, site Web : www.robocup.org.

la vision.

L'odorat permet aux agents de sentir l'« odeur » (qu'on appellera potentiel) des cases voisines. Chaque source d'odeur propage dans l'environnement un champ de potentiel, qui permet aux agents de trouver leur chemin vers la ressource. Une caractéristique importante des champs de potentiel est qu'ils contournent les obstacles. Un modèle de propagation de potentiel est la *propagation par vagues* [31, 30, 59, 86]. Il consiste à propager le potentiel de la ressource de plus en plus loin en décroissant la valeur de potentiel, comme les vagues. Plusieurs fonctions peuvent être utilisées pour la décroissance du potentiel. **BioLand** utilise une fonction inversement proportionnelle avec le carré de la distance. Notre modèle utilise une fonction linéaire. Également, lorsqu'une case est influencée par plusieurs ressources, une fonction de combinaison est utilisée, par exemple **add**, comme dans **BioLand**, ou **maximum**, comme dans notre bibliothèque, chacune avec des avantages et des inconvénients.

La vision permet aux agents de voir des informations sur une case de l'environnement. Contrairement à l'odorat, le champ de vision peut être grand, contenant généralement toutes les cases à une distance inférieure à un certain nombre, appelé *rayon de vision*. Une case A est visible d'une case B si la ligne *continue* du centre de A au centre de B ne rencontre aucune case opaque. Dans notre modèle, les cases opaques sont les obstacles. Nous allons présenter dans le chapitre de vision plusieurs algorithmes de traçage de lignes utilisables dans la vision.

1.3 Principes du calcul parallèle

Nous allons présenter dans cette section seulement les concepts du parallélisme exploités dans cette thèse, et non une description exhaustive du parallélisme. Nous entendons par « parallélisme » l'utilisation simultanée de plusieurs processeurs afin de diminuer les temps d'exécution ou de traiter dans le même temps des problèmes plus importants.

Mesure des performances

La mesure des performances des programmes, surtout des programmes parallèles, est très complexe, parce que beaucoup de paramètres influencent les performances. Nous verrons dans les sections 7.10 et 8.8 les principales mesures de performances de notre outil.

L'indicateur le plus connu pour quantifier les performances d'un programme parallèle est l'*accélération*. L'accélération S est définie comme le rapport entre le temps d'exécution séquentiel et le temps d'exécution parallèle : $S(P) = T(1)/T(P)$, où P est le nombre de processeurs. Le temps d'exécution peut être le temps total passé (« temps horloge ») ou bien le temps passé par le processeur (« temps processeur »). Le temps séquentiel

peut être principalement le temps du même programme quand il est exécuté sur un seul processeur (thread) ou bien le temps du *meilleur* programme séquentiel.

Une accélération égale au nombre de processeurs est appelée *accélération idéale*. Il y a toutefois trois cas où un programme peut dépasser cette accélération : quand toutes les données finissent par tenir dans l'ensemble des caches de tous les processeurs utilisés, quand le problème est non déterministe (à cause des multiples points de départ, la solution est trouvée beaucoup plus rapidement) et quand la parallélisation a entraînée une modification de l'algorithme de départ.

Sources de pertes de performances en parallélisme

Il existe deux sources principales de perte de performances en parallélisme : la présence inévitable de fractions séquentielles et les surcoûts des opérations de parallélisation.

L'influence des parties séquentielles sur les performances est donnée par deux lois. La première, la loi d'Amdahl [6], limite l'accélération d'un programme parallèle à

$$S_{\max}(P) = \frac{1}{f_A + \frac{1-f_A}{P}}$$

où f_A est la fraction séquentielle — la fraction du temps d'exécution qui ne peut pas être parallélisée — et P est le nombre de processeurs.

La deuxième loi, la loi de Gustafson-Barsis [46], s'applique pour les programmes dont la partie séquentielle a une durée constante même quand la taille du problème augmente, ainsi que le temps d'exécution total. Dans ce cas, la fraction séquentielle $f_{GB} = T_{\text{seq}}/T_{\text{total}}$ est une fonction décroissante de la taille du problème W , cette loi étudiant alors l'évolution de l'accélération lorsqu'on traite des problèmes de plus en plus gros sur de plus en plus de processeurs, en gardant un temps d'exécution constant : $T(P, W) = T(1, W_0)$. On aboutit à une limite d'accélération bien meilleure que celle d'Amdahl :

$$S_{\max}(P) = f_{GB} + (1 - f_{GB})P$$

Il faut remarquer que ces deux lois ne sont *pas* incompatibles, mais s'appliquent à des comportements différents du même programme : la loi d'Amdahl suppose la fraction séquentielle constante et varie le nombre P de processeurs à taille de données W constante, ce qui donne des temps d'exécution T différents, tandis que la loi de Gustafson-Barsis suppose la fraction séquentielle décroissante en fonction de la taille du problème et varie P et W tel que T reste constant. Dans tous les cas, il existe une fraction séquentielle qui limite l'accélération.

L'utilisation du parallélisme ajoute des surcharges d'exécution qui réduisent les performances parallèles. Elles peuvent être dues aux limites du

matériel, comme la création de tâches, la contention de mémoire et les délais de communication et de synchronisation. Elles peuvent également être dues au logiciel, comme le déséquilibre de charge ou le coût de l'équilibrage, et le surtravail. Le surtravail dans un programme apparaît quand, pour être exécuté en parallèle, du travail additionnel est donné aux processeurs par rapport à la version séquentielle ; un tel exemple est notre méthode itérative-fixe de propagation par vagues (section 8.8.3).

Finalement, un autre facteur qui influence les performances parallèles est la granularité, définie comme la taille moyenne des sous-tâches des processeurs [5] ou le rapport entre le nombre de tâches et le nombre de processeurs [37].

Complexité de la programmation parallèle

La programmation parallèle est souvent plus difficile que la programmation séquentielle. Les conditions d'accès concurrent aux données, la décomposition des tâches, la portabilité et le débogage sont quelques-unes de ces difficultés. De plus, pour augmenter fortement ses performances en exécution, une conception plus appropriée et plus complexe d'un algorithme parallèle doit être faite, comme montré par nos expériences [83].

Cependant, la balance entre le temps d'exécution et le temps de développement peut être équilibrée. Et, comme Skillicorn [76, page 131] le remarquait, dans la recherche, beaucoup de logiciels sont destinés à une utilisation à court terme, générant des problèmes nouveaux à essayer. Pour ce type de logiciels, le temps de développement est aussi important que le temps d'exécution. Cette idée a influencé la conception de notre modèle, car la simplicité de programmation est son but premier. Par exemple, le parallélisme de la bibliothèque est transparent à l'utilisateur (chapitre 9).

Architectures parallèles

Lors de l'implantation d'une application parallèle, plusieurs architectures parallèles doivent être prises en compte. Flynn [35], par exemple, a classifié les ordinateurs en quatre catégories :

1. SISD (Single Instruction Single Data) machines, qui exécutent *une* instruction sur *une* donnée en même temps, par exemple une machine mono-processeur. Il est intéressant de citer un article plus récent (1996) de Flynn [36], où, face aux caractéristiques parallèles des processeurs modernes (pipelinés et superscalaires), il reformule sa définition en « *une opération est exécutée par transition d'état* ».
2. SIMD (Single Instruction Multiple Data) machines, qui exécutent *une* instruction sur *plusieurs* données en même temps, par exemple les machines vectorielles.

3. MISD (Multiple Instruction Single Data) machines, qui exécutent *plusieurs* instructions sur *une* donnée en même temps. Ces machines n'existent pas réellement en forme pure, mais une analogie peut être faite avec le pipeline des processeurs.
4. MIMD (Multiple Instruction Multiple Data), qui exécutent *plusieurs* instructions sur *plusieurs* données en même temps.

À leur tour, les machines MIMD peuvent être divisées en trois catégories : multicomputers (machines à mémoire distribuée), multi-processeurs (machines à mémoire partagée) et machines à mémoire partagée distribuée (partagée mais physiquement distribuée). Comme nous le présenterons plus loin, nous serons intéressés par les machines supportant le paradigme de programmation à mémoire partagée.

Un élément essentiel d'une architecture est sa mémoire cache, une mémoire rapide s'interposant entre les registres du processeur et la mémoire. Son rôle est d'avoir des copies de lignes de mémoire fréquemment utilisées. L'intérêt du cache vient des deux principes suivants, rencontrés dans la quasi-totalité des programmes [48, page 38] :

1. *Localité temporelle* : une même donnée est accédée plusieurs fois dans une courte période de temps, ce qui exploite la vitesse supérieure des caches.
2. *Localité spatiale* : des données proches d'une donnée déjà accédée sont à leur tour accédées, ce qui exploite le fait que les caches mémorisent des *lignes* entières de mémoire.

Hill et al. [49] remarquent un autre intérêt des caches dans les programmes parallèles : la *localité processeur*. Un programme a une bonne localité processeur si pour de courtes périodes de temps les accès à une donnée sont faites par un même processeur (plutôt que par plusieurs processeurs).

Connaître l'organisation du cache est essentiel pour obtenir de bonnes performances. Plusieurs techniques permettent de l'exploiter au mieux, comme l'ordre de rangement et d'accès aux données, le maintien de la cohérence des caches [77] et l'évitement du faux partage [48, pages 669–670], [49].

Stratégies de programmation parallèle

Germain-Renaud et Sansonnet [43, chapitre 1] présentent les trois sources de parallélisme dans les programmes :

1. Parallélisme de données, où un même traitement peut être appliqué concurremment sur plusieurs données.
2. Parallélisme de flux, où chaque donnée subit plusieurs traitements successifs et plusieurs données en des étapes différentes du programme peuvent être traitées concurremment.
3. Parallélisme de contrôle, où plusieurs tâches différentes sont exécutées concurremment.

Notre bibliothèque utilise la première et la troisième source.

Skillicorn [76] présente six modèles de programmation parallèle, en fonction de leur abstraction au parallélisme offerte à l'utilisateur. Elle commence par une programmation est totalement transparente et finit par une programmation où tout est explicite, comme la décomposition des données et la communication entre les processeurs. Notre bibliothèque se situe parmi les premiers niveaux, car le parallélisme de notre bibliothèque est quasiment transparent à l'utilisateur.

Plusieurs paradigmes de programmation existent, comme l'envoi de messages et la mémoire partagée. Dans notre modèle où les agents et les ressources peuvent influencer tout l'environnement, nous avons choisi la programmation à mémoire partagée. Nous avons également utilisé un style de programmation SPMD (Single Program Multiple Data) [5, pages 609–610] avec un modèle de programmation BSP (Bulk Synchronous Parallelism). Ce choix a été aussi influencé par nos expériences antérieures de développement de systèmes multi-agent [57].

Sur les machines à mémoire partagée plusieurs méthodes de programmation parallèle existent, comme le multi-processus, le multi-threading explicite et le multi-threading implicite. Nous avons choisi la parallélisation explicite par threads, car elle est de nos jours portable, performante et simple à mettre en œuvre.

Nombres aléatoires parallèles déterministes

Les nombre aléatoires sont nécessaires dans notre modèle pour fournir la diversité des simulations. Deux aspects de la suite de nombres utilisés sont importants : son caractère aléatoire et sa reproductibilité. Knuth [58], Anderson [7] et Foster [37, pages 329–335] fournissent une analyse détaillée sur la génération séquentielle et parallèle des nombres aléatoires et sur leur reproductibilité. En parallèle, la reproductibilité devient difficile. L'approche classique est d'utiliser une fonction mathématique qui, à partir d'un nombre donné, appelé *racine*, génère une séquence de nombres quasi-aléatoires.

Notre bibliothèque utilise une racine donnée par l'utilisateur et fournit une simulation aléatoire et totalement reproductible, en recalant le générateur à chaque appel.

1.4 Exemples d'implantations de systèmes multi-agent

Nous présentons dans cette section les simulateurs multi-agent qui se rapprochent le plus du but de notre modèle. Quelques idées sur l'environnement de développement de tels simulateurs seront aussi introduites.

Pengi, un simulateur séquentiel simple

Pengi, de Agre et Chapman [4, 3], consiste en un environnement rectangulaire où se trouvent des blocs de glace, des « blocs magiques », des abeilles et un pingouin. Le pingouin a un comportement donné par l'utilisateur, alors que les abeilles ont un comportement déjà écrit et tendent généralement à se rapprocher du pingouin. Le but des abeilles est de tuer le pingouin, soit en se rapprochant de lui et le piquant, soit en jetant sur lui un bloc de glace. Le but du pingouin est de collecter tous les « blocs magiques ». Il peut aussi jeter des blocs de glaces vers les abeilles, pour les tuer.

Les agents (pingouin ou abeilles) se déplacent d'une case. Ils ont un seul percept : la vision. Les conflits spatiaux entre les agents sont laissés à la charge de l'utilisateur.

Drogoul, Ferber et Jacopin [30] ont écrit une version améliorée de **Pengi**. Les auteurs ont ajouté de l'intelligence et de l'apprentissage au niveau des agents, ainsi qu'un autre percept : l'odeur. Pour la propagation des odeurs ils ont utilisé un algorithme comme le nôtre, de propagation par vagues.

Comparée à ces simulateurs, notre bibliothèque ne fournit pas à ce jour d'action consistant à pousser des objets. Cependant, elle offre d'autres fonctionnalités et plus de généralité que le simulateur **Pengi**.

PIOMAS, un simulateur parallèle utilisant des incertitudes

La spécificité du simulateur **PIOMAS** [14, 16, 15] est qu'il prend en compte les incertitudes et les erreurs, autant dans les perceptions que dans les actions des agents. Il utilise pour cela des matrices de probabilités obtenues suite à des expériences réelles avec un robot **NOMAD 200**.

L'environnement contient des objets statiques (murs) et des objets mobiles (portes, agents). Le système maintient la cohérence du système et gère la simultanéité des actions. La simulation est divisée en cycles, chacun contenant quatre sous-cycles, ce qui lui permet de résoudre les conflits spatiaux entre les agents.

Le simulateur est implanté en **ParCeL-3** [82], un langage cellulaire parallèle orienté vers les applications multi-agent, mais très générique et relativement complexe. **ParCeL-3** s'appuie sur le paradigme multi-threading et a été développé à Supélec.

La résolution des conflits spatiaux découlant de la simultanéité des actions a été résolue par une parallélisation basée sur les conflits et un équilibrage de charge dynamique utilisant deux work-pools [60] en cascade. L'accélération sur deux machines parallèles différentes, dont un SGI Origin 2000 d'architecture DSM, est de 2 sur 4 processeurs.

Ce simulateur est approprié pour des simulations fines, avec un petit nombre d'agents, contrairement à notre modèle et à notre bibliothèque.

BioLand, un simulateur massivement parallèle

Le simulateur **BioLand** fournit beaucoup de fonctionnalités, comme des réseaux de neurones et des informations génétiques utilisables dans le comportement des agents, et peut simuler un très grand nombre d'agents sur des machines parallèles.

Les agents ont deux percepts, l'odeur et l'ouïe, et plusieurs actions, principalement se déplacer, manger, se reproduire et émettre des sons. Tous les objets de simulation produisent un gradient d'odeur, perçu par les différents types d'agents.

Le simulateur a été écrit dans le langage **C*** sur une machine CM-2 avec 16k processeurs (une machine SIMD massivement parallèle). Des populations de dizaines de milliers d'agents ont pu être simulées.

BioLand offre donc un modèle de simulation très riche en fonctionnalités. Cependant, la vision et les conflits entre les agents ne sont pas pris en compte. Enfin, il ne fonctionne que sur des machines SIMD, qui ne sont plus couramment utilisées aujourd'hui.

Environnement de développement de systèmes multi-agent

Plusieurs approches sont possibles pour réaliser un environnement de développement de développement de SMA : langage spécifique, extension d'un langage existant par de nouveaux mots clés, bibliothèque etc. Ferber [32] présente par exemple plusieurs langages spécifiques aux SMA. Pour notre part, nous avons choisi de réaliser une bibliothèque en langage C, afin d'obtenir un outil rapide, portable et très générique, vu la grande souplesse du langage C.

1.5 Définition d'un modèle de simulation de systèmes multi-agent

Cette section présente le modèle que nous avons conçu pour la simulation de systèmes multi-agent situés.

Composants du modèle

Dans notre modèle quatre composants apparaissent (figure 6.1, page 110) :

1. L'environnement : le monde où les entités évoluent. Il est discrétisé et peut être torique ou borné. Les cases peuvent contenir des obstacles, qui empêchent les agents d'y entrer.
2. Les ressources : entités qui attirent les agents. Elles propagent un champ de potentiel perçu par les agents. Le potentiel d'une ressource et le champ qui en découle peuvent évoluer pendant une simulation. Les ressources peuvent également contenir des objets, qui peuvent être pris,

transportés et déposés dans d'autres endroits par les agents. Plusieurs types de ressources peuvent être définis.

3. Les agents : entités mobiles avec mémoire et comportement propres. Le comportement de chaque agent génère une action, basée sur les perceptions et la mémoire de l'agent. Pendant la simulation, la mémoire des agents peut évoluer en contenu et en taille. Un agent ne connaît pas les actions des autres agents, donc, en essayant d'entrer dans une même case, ils peuvent entrer en conflit. Ces conflits spatiaux sont automatiquement résolus par l'arbitre.
4. L'arbitre : entité virtuelle qui maintient la cohérence du système. Son rôle principal est de résoudre les conflits spatiaux entre les agents. Pour chaque conflit, il laisse un seul agent accomplir son action et bloque les autres. Plusieurs stratégies pour le choix du gagnant sont possibles. Dans notre bibliothèque le choix est fait aléatoirement sur une échelle à trois niveaux : d'abord parmi les agents nouvellement créés, ensuite parmi les agents à priorité supérieure et finalement parmi les autres agents.

Comportement des agents

Nous présentons le comportement des agents dans une section à part car il ne dépend pas des autres caractéristiques déjà mentionnées.

Dans un modèle purement multi-agent, les perceptions et les actions d'un agent sont locales à sa position. Cependant, pour une flexibilité accrue, nous laissons la possibilité d'utiliser aussi des caractéristiques globales, mais elles sont bien différenciées des caractéristiques locales lors de l'utilisation.

Deux exemples de perceptions sont la vision et la perception du potentiel, qui sont également implantées dans notre bibliothèque. La vision permet à un agent d'obtenir des informations sur une case distante. La perception du potentiel permet à l'agent de retrouver, en suivant un potentiel croissant, un chemin vers la ressource qui l'émet. Les agents n'ont pas tous les mêmes percepts.

Quelques exemples d'actions sont le mouvement, la prise et le dépôt d'objets. Les actions sont prédéterminées et déterministes, par exemple les agents iront dans une case si ce mouvement est possible et voulu, et ils ne pourront pas arriver accidentellement dans une case (en « glissant » par exemple).

Les agents peuvent être nés et détruits dynamiquement, tout au long de la simulation. Ils peuvent être créés dans une case spécifique ou une case au hasard, soit dans une certaine région de l'environnement, soit dans tout l'environnement. La création est contrôlée par l'arbitre.

Moteur d'exécution

Dans notre modèle les agents agissent de façon synchrone (pendant chaque cycle de simulation, tous les agents sont activés une seule fois) et simultanément (il n'y a pas de priorité d'exécution d'un agent sur un autre). Ces deux caractéristiques peuvent générer des conflits spatiaux.

À l'exception des étapes d'initialisation et de fin de la simulation, le moteur d'exécution est discretisé et basé sur des cycles de simulation. Chaque cycle contient 5 sous-cycles, qui permettent la synchronicité et la simultanéité des activations des agents et permettent également la résolution des conflits :

1. activation du comportement de chaque agent, où ils planifient une ou plusieurs actions, comme le mouvement et la création d'autres agents ;
2. résolution des conflits spatiaux par l'arbitre ;
3. exécution des actions des agents non conflictuels ou vainqueurs des conflits ;
4. exécution de la fonction utilisateur (en séquentiel), qui permet à l'utilisateur de reprendre la main à la fin de chaque cycle ; ici il peut arrêter la simulation ou sauvegarder diverses informations pour une analyse ultérieure ;
5. mise à jour de l'environnement, notamment des champs de potentiel des ressources, si ces potentiels ont évolué.

Seuls les sous-cycles 1 et 4 doivent être fournis par l'utilisateur ; les autres sont pris automatiquement en charge par le simulateur.

Informations dynamiques sur la simulation

Tout au long de la simulation, l'utilisateur a accès à divers paramètres de la simulation. Additionnellement, le modèle fournit trois types d'informations. Le premier type permet de mesurer l'efficacité des agents : il fournit des informations succinctes sur l'évolution du système, comme le nombre d'agents et le nombre de ressources non encore visitées par les agents. Le deuxième type permet une analyse fine du système : il génère l'évolution détaillée du système, comme la position des agents et la charge en objets de chaque ressource. Le troisième type permet de faire une sauvegarde complète du système et il peut également être utilisé pour initialiser un nouveau système.

Le modèle que nous proposons ne fait pas intervenir le parallélisme. Comme nous le verrons plus loin, le parallélisme de notre implantation du modèle est caché à l'utilisateur. Toutefois, notre modèle a été conçu pour supporter une parallélisation.

1.6 Conception d'un algorithme de vision

Introduction

La vision est le percept qui permet à un agent d'obtenir des informations sur une case distante. Pour cela, il utilise d'abord le champ de visibilité de la case où il se trouve, qui lui dit si la case cherchée est visible ou non. Ensuite, si la réponse est affirmative, il peut obtenir l'information cherchée en appelant la fonction appropriée. Cet algorithme traite le calcul des champs de visibilité.

Dans nos simulations, les seules entités opaques sont les obstacles, et ils sont fixes pendant la simulation. Le calcul des champs de visibilité peut alors se faire une seule fois, en début de simulation. Naturellement, la relation de visibilité est symétrique : A voit B si et seulement si B voit également A .

Le champ de visibilité d'une case contient toutes les cases à une distance inférieure à un nombre qu'on appellera *rayon de vision*. La forme du champ dépend de la connectivité des cases : un losange dans une connectivité de 4 et un carré deux fois plus grand dans une connectivité de 8. Pour la mesure des performances, nous serons amenés à connaître le nombre de cases du champ de vision. Pour une connectivité de 4 il est :

$$NS_4 = 4 \sum_{i=1}^{i=r_v} i + 1 = 2r_v^2 + 2r_v + 1$$

Pour une connectivité de 8 il est :

$$NS_8 = (2r_v + 1)^2 = 4r_v^2 + 4r_v + 1$$

L'algorithme de vision consiste à calculer le champ de visibilité de toutes les cases de l'environnement. Comme l'environnement est en dimension 2 et le champ de visibilité de chaque case est en dimension 2 aussi, la matrice stockant les informations de visibilité est de dimension 4.

En absence d'obstacles, le champ de visibilité serait entièrement « visible ». En présence de ceux-ci, la visibilité entre deux cases est donnée par les propriétés de la ligne droite continue entre les centres des deux cases. Les deux cases sont visibles entre elles si et seulement si la ligne ne rencontre aucune case opaque. Au cas où la ligne passe par le coin d'une case opaque, nous avons choisi de considérer les deux cases comme invisibles l'une de l'autre.

Les cases traversées par la ligne continue sont données par une ligne spéciale, qu'on appelle *ligne épaisse*². Plusieurs algorithmes de traçage de lignes normales et épaisses sont donnés dans la littérature [17, 13, 38, 11, 8, 89]. Nous sommes particulièrement intéressés par l'algorithme de Bresenham

²Cette ligne est appelée *ligne supercouverture* par Éric Andres, de l'Université de Poitiers, qui fait des recherches sur la modélisation analytique discrète d'objets géométriques.

de traçage de lignes, car il est utilisé dans l'algorithme de lignes épaisses que nous avons conçu.

Algorithme de Bresenham de traçage de lignes

L'algorithme de Bresenham [17] a deux particularités importantes : il n'utilise pas d'opérations avec des nombres flottants, ni de multiplication ou division de nombres. L'évitement de ces opérations coûteuses en temps d'exécution nous a amenés à le choisir comme base pour notre algorithme.

Pour présenter l'idée de l'algorithme de Bresenham, nous considérons le cas de traçage d'une ligne entre deux points, dans le premier octant ($\Delta x \geq \Delta y$). Le traçage de la ligne commence par le point avec la coordonnée x la plus petite. À chaque étape, la coordonnée x du point à dessiner est incrémentée et, en même temps, un terme de contrôle (l'*erreur*) est mis à jour. Plus précisément, la valeur de la pente de la droite est ajoutée au terme de contrôle et, s'il dépasse la dimension d'une case, la coordonnée y du point est aussi incrémentée et le terme de contrôle réajusté. Après ce calcul, le point est dessiné (voir également les cas a et b de la figure 1.1).

Notre algorithme original de traçage de lignes épaisses

Pour obtenir la visibilité entre deux cases, nous avons conçu un algorithme original de traçage de lignes épaisses basé sur l'algorithme de Bresenham. Dans notre algorithme, à chaque étape le terme de contrôle est sauvegardé avant d'être recalculé, et nous utilisons les deux termes pour trouver les cases traversées par la ligne continue. En effet, comme présenté dans la figure 1.1, nous savons les cases à dessiner en fonction de la somme entre ces deux termes. Trois cas apparaissent :

1. somme inférieure à la taille d'une case : le point à droite du point courant est alors dessiné.
2. somme égale à la taille d'une case : la droite continue passe par un coin, et nous dessinons alors les trois points voisins.
3. somme supérieure que la taille d'une case : le point au-dessus du point courant est alors dessiné.

Optimisations de l'algorithme de vision

L'algorithme de vision consiste à tracer des lignes épaisses entre *tous* les couples de points de l'environnement distants d'une longueur inférieure ou égale au rayon de vision. Cet algorithme a plusieurs caractéristiques qui permettent son optimisation :

- La symétrie : si une case A voit une case B , alors la case B voit aussi la case A . En conséquence, il est possible de calculer, pour chaque case, seulement une moitié de son champ de visibilité.

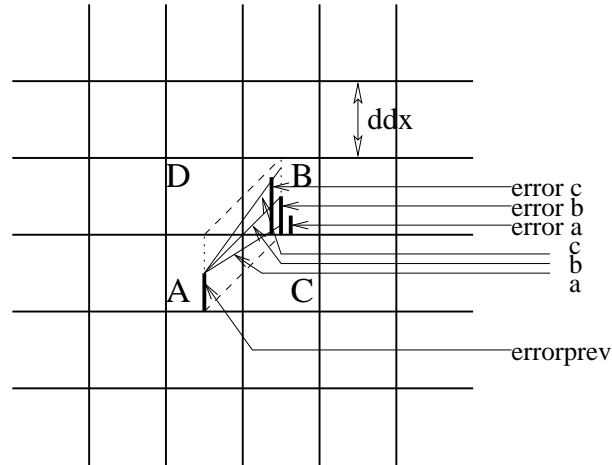


FIG. 1.1 – Illustration d’une étape de l’algorithme de traçage de lignes épaisses : la comparaison entre la somme des erreurs courante et précédente et la taille d’une case génère trois cas (*a*, *b* et *c*).

- Connectivité : pour une connectivité de 4, le champ de visibilité est un losange. Il est possible d’allouer de l’espace mémoire seulement pour le losange, avec une fonction bijective entre les cases du losange et la mémoire (qui est linéaire).
- Type de propagation : les champs de visibilité peuvent être calculés statiquement, en début de simulation, ou dynamiquement, seulement quand cela est nécessaire. Des approches mixtes peuvent également être imaginées.

Notre bibliothèque permet de calculer les champs de visibilité en exploitant la symétrie ou non, ainsi que statiquement ou dynamiquement.

Parallélisation de l’algorithme de vision

Nous nous intéressons à la parallélisation de l’algorithme dans le cas statique, quand les champs de visibilité sont calculés en début de simulation. Pendant la simulation, les champs sont seulement lus, quand les agents en ont besoin, donc aucun problème de parallélisation n’apparaît.

Nous considérons alors l’initialisation de ces champs. Calculer un champ revient à *lire* les structures d’environnement, pour savoir où sont les obstacles, et *écrire* dans la matrice des champs. Comme l’écriture concerne seulement la case dont le champ est calculé, aucun problème de conflit d’écriture n’apparaît. Le calcul de champ de chaque case étant identique, la parallélisation la plus naturelle est le partitionnement du domaine. Dans notre bibliothèque, à chaque processeur est assigné un nombre identique de lignes (partitionnement horizontale de l’environnement). En fait, le nombre

de lignes des domaines peut différer de 1 si le nombre de lignes de l'environnement n'est pas divisible par le nombre de processeurs.

Modèles de temps et de mémoire

Plusieurs paramètres influencent les performances de l'algorithme de vision : le nombre de cases de l'environnement N , sa topologie (environnement torique ou borné), la connectivité des cases (4 ou 8), le rayon de vision r_v , le nombre d'obstacles et le nombre de processeurs.

L'espace nécessaire à l'algorithme de vision est l'espace occupé par la matrice de visibilité. Comme décrit précédemment, cette matrice contient le champ de visibilité de chaque case. L'espace mémoire est alors :

$$\text{mem}(N, r_v) = N(2r_v + 1)^2$$

quelle que soit la connectivité, si on n'optimise pas l'occupation mémoire dans le cas de la connectivité de 4, en losange.

Le temps d'exécution de chaque processeur est donné par :

$$\text{time} = \frac{1}{P} N(2r_v + 1)^2 t_{elem}$$

où t_{elem} est le temps nécessaire pour calculer la visibilité entre deux cases.

Les optimisations présentées ci-dessus peuvent réduire d'un facteur constant autant l'espace mémoire que le temps d'exécution.

Performances de l'algorithme de vision

Pour nos expérimentations nous avons utilisé deux systèmes. L'un des systèmes est un serveur Sun Workgroup 450, une machine SMP, avec 4 processeurs, sous GNU/Linux. L'autre système est un supercalculateur SGI Origin 2000 [34], une machine DSM, avec 64 processeurs, sous Irix.

L'algorithme de vision a trois parties : une allocation séquentielle, une allocation parallèle et l'opération de calcul. Les paramètres qui influencent le temps d'exécution ont déjà été présentés ci-dessus.

Les nombreuses mesures de performance que nous avons effectuées nous ont amenés aux conclusions suivantes. L'allocation séquentielle a un temps d'exécution négligeable, même en considérant ses conséquences données par la loi d'Amdahl. En parallèle, les nombreuses allocations mémoire, même de petite taille, peuvent représenter un goulot d'étranglement. Après avoir réduit les nombreuses allocations à seulement deux allocations par processeur, les allocations parallèles ont très peu influencé le temps d'exécution total. Enfin, l'opération de calcul est équilibrée en charge. Dans un environnement 512x512, sans obstacles, avec une connectivité de 4 et un rayon de vision de 8, nous obtenons de très bonnes performances : accélération de 49 sur 64 processeurs. La courbe d'efficacité a une allure très intéressante, avec de

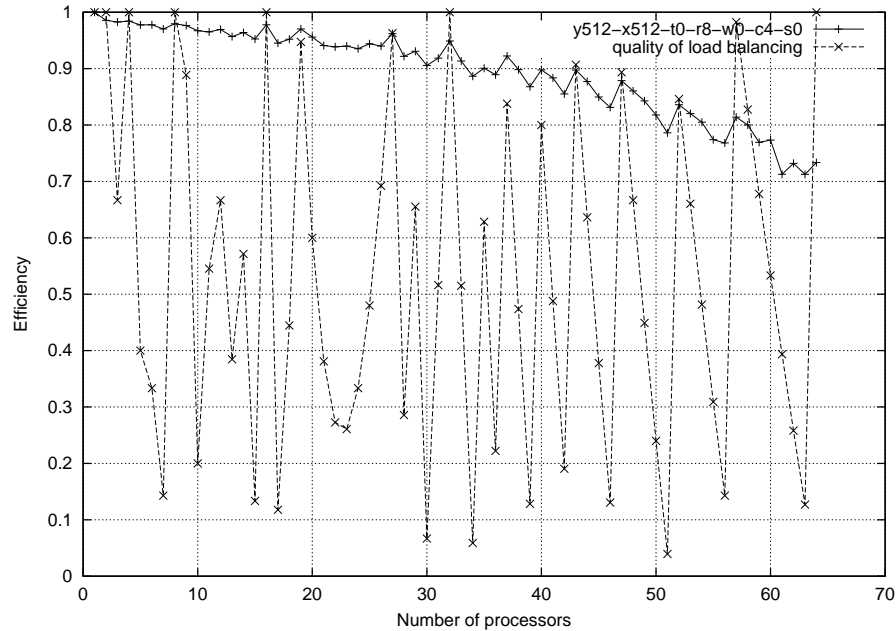


FIG. 1.2 – La fonction théorique d'équilibrage de charge concorde avec les mesures expérimentales faites : les extrema locaux apparaissent pour les mêmes nombres de processeurs (DSM Origin 2000).

fortes variations. Comme montré dans la figure 1.2, nous avons pu trouver la source de cette forme. Elle est due au déséquilibre de charge qui provient de la décomposition horizontale imparfaite de l'environnement. L'équilibrage théorique de charge entre les processeurs est donné par la fonction Q_{dbal} , que nous avons pu établir, et la figure 1.2 montre très bien que la fonction théorique et les mesures pratiques concordent.

En faisant varier les paramètres de vision lors des nombreuses mesures de performances, nous avons établi empiriquement que le temps d'exécution séquentiel de l'opération de calcul est donné par la relation :

$$\text{time}_{\text{filling}}^{\text{seq}} = \alpha N r_v^2$$

De plus, l'utilisation de la symétrie, ainsi qu'une connectivité de 4 à la place de 8, réduit le temps d'exécution exactement à moitié. Ces résultats expérimentaux concordent également avec nos calculs théoriques.

En revanche, aucune influence des caches n'a été observé lors de ces mesures.

Finalement, pour les paramètres donnés ci-dessus, les performances globales de l'algorithme de vision sont, pour 57 processeurs, un temps d'exécution de moins d'une seconde et une accélération de 43. Nous considérons que les performances de l'algorithme de vision sont suffisamment bonnes pour nos simulations.

1.7 Conception d'algorithmes de propagation de potentiel

Introduction

Toute ressource peut générer aux alentours d'elle un champ de potentiel qui décroît avec l'éloignement de la ressource. La perception de ces potentiels est un percept des agents qui leur permet donc de trouver le chemin vers les ressources.

Contrairement à la vision, les champs de potentiel contournent les obstacles. Mais les champs de potentiel évoluent pendant la simulation, et ils doivent être mis à jour plusieurs fois pendant la simulation.

Dans nos simulations nous avons fait deux hypothèses. La première est que le potentiel décroît linéairement avec la distance vers la ressource. La deuxième est que, lorsque plusieurs champs se superposent dans une case, le potentiel de la case est donné par le *maximum* des potentiels concernés. Ces hypothèses ont été suffisamment bonnes pour le but de nos simulations et suffisamment simples à implanter efficacement.

La propagation du champ de potentiel se base sur le principe de Huygens [51]. Elle commence d'abord par mettre le potentiel de la ressource dans la case de celle-ci et continue par mettre récursivement dans toutes les cases du nouveau front de potentiel un potentiel décrétementé de 1. La topologie du champ de potentiel est alors identique à la topologie du champ de vision (voir section précédente).

Méthodes séquentielles

Nous avons imaginé 4 types de méthodes de propagation séquentielle de potentiel.

Les méthodes récursives mettent le potentiel de la ressource dans sa propre case et, tout en évitant les obstacles, propagent récursivement ce potentiel en soustrayant 1 à chaque fois qu'on s'éloigne de la ressource. La propagation peut se faire en largeur ou en profondeur. La propagation en largeur utilise une queue pour la sauvegarde des cases en train d'être traitées, et met dans chaque case directement son potentiel final. La propagation en profondeur utilise la pile fournie automatiquement par le langage de programmation, mais a le désavantage que certaines cases reçoivent plusieurs potentiels avant de recevoir leur potentiel final. Dans les deux cas, l'espace mémoire nécessaire est $O(N)$ (chaque case stocke son potentiel), où N est le nombre de cases de l'environnement. Le temps d'exécution est $O(p^2)$, où p est le potentiel de la ressource, ce qui donne un temps d'exécution total de $O(p^2R)$, où R est le nombre de ressources dans l'environnement.

Les méthodes itératives balaient plusieurs fois tout l'environnement en appliquant une opération simple sur chaque case. Les deux méthodes que

nous présentons commencent par mettre dans les cases des ressources leur propre potentiel. Ensuite, la méthode de Bouton [13], qui a travaillé dans notre équipe, balaie l'environnement successivement avec chacun des potentiels en commençant par le plus grand potentiel des ressources et en finissant à 1. Lors de chaque balayage, toutes les cases avec un potentiel égal au potentiel de balayage mettent dans leurs cases voisines ce potentiel moins 1, au cas où elles ne contiennent pas déjà une valeur plus grande. À la fin des balayages, l'environnement contient tous les champs de potentiel complètement propagés.

Notre méthode itérative est de balayer l'environnement jusqu'à ce que le potentiel d'aucune case ne soit plus mis à jour. Lors de chaque balayage, toutes les cases voisines de la case courante reçoivent le potentiel de la case courante moins 1, au cas où elles ne contiennent pas déjà une valeur plus grande. Cette méthode a l'avantage d'être très rapide lorsque peu d'obstacles existent dans l'environnement.

L'espace mémoire des méthodes itératives est identique aux méthodes récursives : $O(N)$ (chaque case stocke son potentiel). Le temps d'exécution dans le pire cas est de $O(Np)$, nécessaire pour les p balayages des N cases.

Un autre type de méthodes consiste à stocker dans chaque case des distances vers des ressources. Il serait possible de stocker dans chaque case soit les distances vers toutes les ressources qui l'influencent, soit une seule distance, vers la ressource qui lui impose son potentiel. La première méthode a le désavantage d'utiliser beaucoup de mémoire dans le cas général, quand les cases sont influencées par plusieurs ressources. La deuxième méthode n'est pas très facile à mettre en œuvre, car elle doit prendre en compte les frontières dynamiques d'influence entre les ressources. Nous n'avons donc pas pris en considération ces méthodes.

Enfin, la méthode multi-grille est efficace dans le cas où il existe des régions de l'environnement qui sont équipotentielles. Elle divise l'environnement en plusieurs domaines rectangulaires et met des potentiels identiques dans un même domaine. En répétant le procédé et en changeant les domaines, en faisant l'hypothèse de la convergence de la méthode on obtient des potentiels de plus en plus exacts. Cette méthode a aussi l'avantage que la propagation peut s'arrêter plus tôt, obtenant une propagation faiblement inexacte des potentiels. Nous n'avons pas démontré la convergence de cette méthode et nous ne l'avons pas prise en considération.

Il est difficile de trouver la meilleure de ces méthodes en comparant seulement leurs temps d'exécution théoriques. Nous avons donc recouru à des mesures de performance en changeant tous les paramètres qui peuvent influencer le temps d'exécution.

Méthodes parallèles

Les méthodes séquentielles ne peuvent pas être utilisées en parallèle telles qu'elles sont à cause des accès concurrents aux potentiels des cases. Plusieurs méthodes de parallélisation des méthodes séquentielles présentées précédemment existent. Nous avons étudié les méthodes de parallélisations suivantes : par décomposition fixe du domaine, par décomposition variable du domaine, par répartition des données protégées par exclusions mutuelles et par décomposition du domaine avec environnement privé pour chaque processeur.

La première méthode est la méthode bien connue de parallélisation par décomposition fixe du domaine. Dans notre cas, cette méthode décompose horizontalement l'environnement en plusieurs domaines et chaque processeur reçoit un domaine propre. Les domaines ont un nombre de lignes identique ou différent de 1. La propagation complète comprend trois étapes.

La première étape correspond, pour chaque processeur, à la propagation des potentiels dans son domaine propre. Toute méthode séquentielle peut être utilisée ici. La deuxième étape consiste à sauvegarder les frontières dans des tampons. Enfin, pendant la troisième étape, chaque processeur fait une mise à jour de ses frontières en les comparant avec les tampons adjacents à son domaine et propage ensuite les potentiels de ses frontières (repropagation). Des méthodes séquentielles sont utilisées de nouveau.

Une seule propagation de frontières peut ne pas suffire. Les étapes 2 et 3 sont répétées jusqu'à ce qu'aucune case de la frontière ne change plus son potentiel. En présence des obstacles, dans le pire cas le nombre de repropagations est égal au potentiel de la ressource la plus forte.

Cette méthode a l'inconvénient d'avoir besoin de plusieurs repropagations pour compléter la propagation des potentiels. En revanche, son espace mémoire est faible : seuls les tampons utilisent de la mémoire.

La deuxième méthode utilise une décomposition variable du domaine et a pour but de réduire le nombre de repropagations. Pour cela, entre deux repropagations successives on change les domaines et les frontières affectées aux processeurs. Par exemple, les nouvelles frontières sont situées à mi-chemin entre deux anciennes frontières. La méthode précédente avait l'inconvénient de faire de nombreuses repropagations, nécessaires à cause des frontières qui changeaient souvent leur potentiel. Avec la nouvelle méthode, pendant la propagation des anciennes frontières, les nouvelles frontières sont loin des anciennes, donc elles sont bien plus rarement mises à jour et le nombre de repropagations est bien réduit. Mais les changements de domaines impliquent des conflits de cache pour un nombre important de cases, ce qui nous a conduit à rejeter cette méthode.

La troisième méthode essaie de résoudre simplement, sans modification de l'algorithme séquentiel, le problème posé par la parallélisation des algorithmes séquentiels, à savoir l'accès concurrent aux potentiels des cases. Chaque région du code où le potentiel d'une case est accédée est transformée

en une région d'exclusion mutuelle, et un verrou (*lock*) par case peut être utilisé. La gestion des régions d'exclusion mutuelle représente une surcharge importante en mémoire et en temps d'exécution, ce qui nous a amené à ne pas prendre en considération cette méthode.

La quatrième méthode utilise des environnements privés à chaque processeur. Dans une première étape, les ressources sont divisées en un nombre de groupes égal au nombre de processeurs et chaque processeur propage dans son propre environnement le potentiel des ressources qui lui sont assignées. Cette propagation se fait avec une des méthodes séquentielles déjà présentées. Dans la deuxième étape, l'environnement reçoit ses potentiels finaux par superposition des potentiels des environnements privés. Pour cela, l'environnement est décomposé en domaines et chaque processeur met à jour son domaine en faisant dans chaque case la superposition des potentiels de toutes les cases homologues des environnements privés.

En parallèle, les environnements privés sont accédés par plusieurs processeurs au cours des deux étapes. Cette méthode ajoute donc des conflits de cache qui peuvent fortement pénaliser les performances de cette méthode. En revanche, cette méthode n'utilise qu'une seule synchronisation entre les processeurs.

La comparaison théorique des méthodes parallèles n'est pas suffisante non plus pour trouver la meilleure méthode. Par conséquent, nous avons procédé à de nombreuses mesures de performances, autant séquentielles que parallèles.

Performances des algorithmes de propagation par vagues

Pour nos expérimentations nous avons utilisé les deux machines parallèles présentées dans la section précédente.

Les algorithmes de propagation par vagues ont une partie d'initialisation, exécutée en début de simulation, et une partie de propagation proprement dite, exécutée plusieurs fois pendant la simulation. Nous nous intéressons seulement à cette partie de propagation.

Les paramètres qui influencent les performances des algorithmes de propagation sont : le nombre de cases de l'environnement, sa topologie (environnement torique ou borné), la connectivité des cases (4 ou 8), le nombre d'obstacles, le nombre et le potentiel des ressources (nous avons utilisé un seul type de potentiel pour nos expérimentations) et le nombre de processeurs.

Notre démarche pour la mesure de performances est la suivante. Nous avons d'abord mesuré les performances séquentielles et parallèles de chaque méthode, ensuite nous avons fait une comparaison détaillée des méthodes qui ont donné les meilleures performances et finalement nous avons développé la meilleure méthode.

Performances de chaque méthode. Nous avons implanté plusieurs méthodes de propagation par vagues et nous avons mesuré les performances de chacune. Pour comparer toutes ces méthodes nous avons utilisé un environnement borné de 1024×1024 , sans obstacles, avec une connectivité de 4 et 1 % de ressources, chacune avec un potentiel de 16.

La méthode de récursion en largeur avec décomposition en domaines fixe donne un temps séquentiel de 1.35 secondes, et sa meilleure accélération est de 15, pour 32 processeurs. C'est une méthode qu'on retrouvera dans la comparaison des meilleures méthodes.

La méthode de propagation en profondeur avec décomposition en domaines fixe a un temps séquentiel de 70 secondes. Ce temps est bien plus grand que celui de la méthode précédente, à cause des multiples mises à jour des cases, comme expliqué dans la présentation de la méthode. Par conséquent, nous ne prenons plus en considération cette méthode.

Notre méthode itérative avec décomposition de domaines fixe a un temps séquentiel de 0.6 secondes (plus petit que celui de la première méthode), mais son accélération n'est pas satisfaisante, par exemple elle est de moins de 2 pour 4 processeurs. Cependant, elle est fortement influencée par deux facteurs, comme montré dans la figure 1.3. Le premier est le cache, qui correspond au redressement de l'accélération à partir de 3 processeurs. Le deuxième est le surtravail, dû à l'étape de repropagation qui apparaît seulement en parallèle, qui est plus visible à partir de 4-8 processeurs³. Le faible temps d'exécution séquentiel, ainsi que l'influence du cache nous ont amenés à prendre en considération cette méthode.

La méthode de récursion en largeur avec des environnements privés à chaque processeur diffère de la première méthode seulement par la méthode de parallélisation. Mais sa meilleure accélération est de 3 sur 4 processeurs, moins que celle de la première méthode. Nous ne la prenons donc plus en considération.

Enfin, notre méthode itérative avec des environnements privés à chaque processeur diffère de la méthode similaire avec domaines fixes seulement par la méthode de parallélisation. Mais sa courbe d'accélération n'est pas extensible et nous ne la prenons plus en considération.

Performances détaillées des meilleures méthodes. Suite aux mesures de performances précédentes, nous avons distingué deux méthodes : la propagation en largeur et notre méthode itérative, les deux parallélisées avec la décomposition de domaines fixe. Pour une meilleure comparaison du temps d'exécution séquentiel, nous allons faire varier *séparément* les paramètres qui influencent leurs performances.

³Le surtravail apparaît dans toutes les méthodes basées sur la décomposition de domaine, mais il est plus visible dans cette méthode à cause de la mauvaise accélération pour un nombre petit de processeurs.

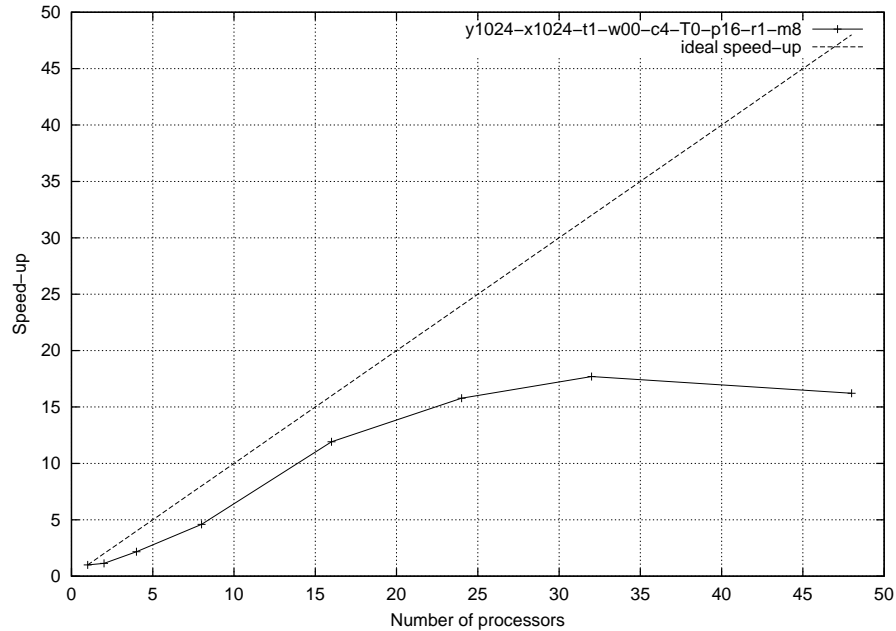


FIG. 1.3 – Accélération de notre méthode itérative avec décomposition de domaines fixe, où on observe un redressement à partir de 3 processeurs (DSM Origin 2000).

Nous avons ainsi constaté empiriquement les caractéristiques suivantes. La taille de l’environnement influence de la même façon les deux méthodes. La connectivité, ainsi que le nombre et le potentiel des ressources, influencent légèrement la méthode itérative et beaucoup la méthode de récursion en largeur. En contrepartie, le nombre d’obstacles influencent beaucoup la méthode itérative, mais pas la méthode de récursion en largeur.

Meilleure méthode. Basés sur les résultats précédents, nous avons pu établir un certain nombre de règles qui spécifient les cas où une méthode est meilleure en séquentiel que l’autre. Ces règles font intervenir plusieurs indicateurs, comme le nombre d’obstacles et le nombre moyen de ressources qui influencent une case.

Comme dans certains cas la méthode itérative est plus rapide en séquentiel que la méthode récursive en largeur, nous avons implanté une méthode mixte, qui utilise la méthode itérative pour la propagation de domaine et la méthode récursive pour la propagation des frontières. La figure 1.4 présente les performances de cette méthode mixte et de la méthode purement itérative. Nous remarquons la différence des performances parallèles et l’influence des caches, qui est bien visible dans la méthode itérative pure. Nous pouvons considérer cette accélération comme celle du point de vue de l’utilisateur,

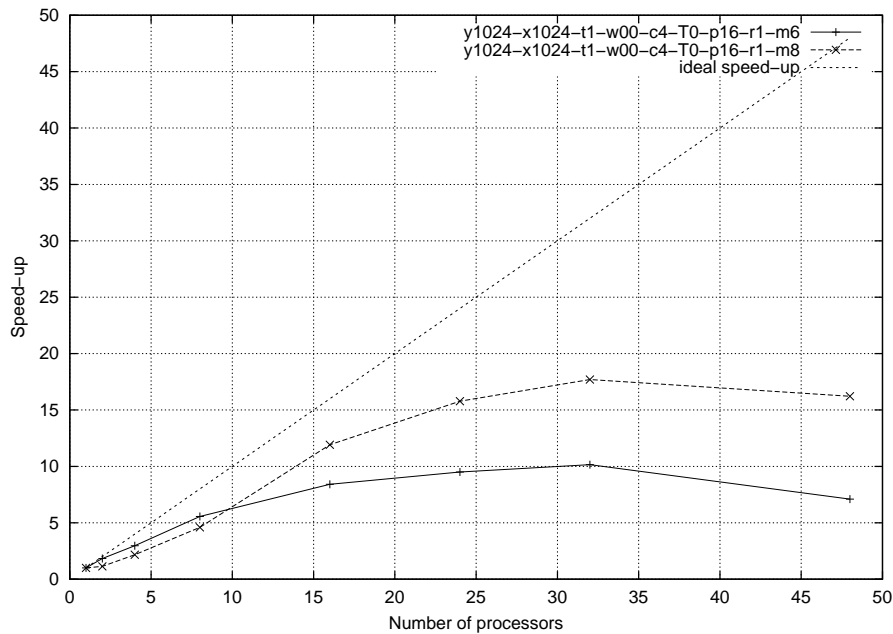


FIG. 1.4 – Accélération de la meilleure méthode, une combinaison entre la méthode mixte (la première dans la légende) et la méthode itérative pure (la deuxième dans la légende) (DSM Origin 2000).

car elle utilise le meilleur temps séquentiel, celui de la méthode itérative, comme référence. La meilleure méthode dépend alors du nombre de processeurs utilisés, comme montré dans la figure 1.4 : la méthode mixte pour peu de processeurs ou la méthode itérative pure pour plus de processeurs.

Propagation inexacte de potentiel

Les agents sont, par définition, autonomes et s’adaptent à l’environnement. En conséquent, pour obtenir des performances encore meilleures, on envisage aussi des propagations inexactes de potentiel. Ce type de propagation tolérerait des erreurs mineures dans les potentiels des cases, erreurs que les agents sont sensés pouvoir surmonter.

1.8 Caractéristiques et implantation de ParSSAP

Caractéristiques de notre bibliothèque

Nous avons implanté et validé notre modèle de simulation de systèmes multi-agent par une bibliothèque qui implante la plupart des fonctionnalités du modèle. Les champs de visibilité, la propagation par vagues et la résolution des conflits sont automatiquement prises en charge par la bibliothèque.

Les trois types d'informations sur le système multi-agent (statistiques, informations succinctes et informations complètes sur son évolution) sont également fournis. En revanche, la version actuelle de notre bibliothèque présente quelques limitations, comme de ne supporter qu'au maximum une ressource par case, de n'offrir que deux percepts pour les agents (la vision et la perception du potentiel) et de n'accepter comme actions des agents que de se déplacer (aléatoirement ou non) et de prendre ou déposer des objets dans des ressources.

La bibliothèque, écrite en langage C, est parallèle. Elle utilise des threads et supporte deux interfaces de programmation multi-threading : les threads POSIX et les threads Irix. Un de nos buts a été de cacher les difficultés du parallélisme à l'utilisateur. En contrepartie, à l'intérieur de la bibliothèque, le parallélisme est utilisé dans tous les sous-cycles de simulation, à l'exception de la fonction utilisateur (qui est exécutée en séquentiel). Le parallélisme a été exploité dans la bibliothèque de la même façon : par décomposition statique du domaine. Le nombre de processeurs, la seule information de parallélisme donnée par l'utilisateur, est spécifié par l'utilisateur et reste inchangé pendant toute la simulation.

Simultanéité et parallélisme dans notre bibliothèque

Si, du point de vue macroscopique, notre modèle de simulation est synchrone et organisé en cycles de simulation, pendant un cycle les agents doivent avoir une vue asynchrone de l'environnement et leurs comportements doivent être activés conceptuellement en simultané. Pour résoudre ces problèmes nous avons utilisé deux types de structures de données, des structures locales à chaque thread et des structures globales, et chaque cycle de simulation a été divisé en plusieurs étapes bien distinctes :

1. étape de planification : exécution du comportement de chaque agent et planification de la création et de la destruction des agents ;
2. barrière de synchronisation ;
3. transfert des demandes de création globales d'agents dans des structures locales aux threads concernés ;
4. transfert des demandes de mouvement inter-domaine d'agents dans des structures locales aux threads concernés ;
5. étape de gestion des conflits spatiaux : résolution de tous les conflits spatiaux, création des agents gagnants et refus de l'action des agents perdants ;
6. création des agents locaux ;
7. barrière de synchronisation ;
8. étape d'action : exécution de l'action des agents et destruction des agents ;

9. barrière de synchronisation ;
10. étape utilisateur : exécution séquentielle de la fonction utilisateur (qui permet par exemple de spécifier la fin de la simulation) et incrémentation du numéro du cycle ;
11. barrière de synchronisation ;
12. étape de sauvegarde : sauvegarde parallèle du système, si demandé ;
13. étape de mise à jour des percepts de l'environnement ;
14. barrière de synchronisation.

Aspect aléatoire et reproductibilité des simulations

Une grande attention a été accordée dans notre bibliothèque à l'aspect aléatoire et à la reproductibilité des simulations. L'aspect aléatoire signifie que toutes les fonctions à caractère aléatoire, comme le mouvement aléatoire des agents et le choix aléatoire des gagnants dans les conflits, utilisent des nombres aléatoires, avec des probabilités *égales* pour tous les choix⁴. D'un autre côté, la reproductibilité assure que les simulations sont complètement reproductibles, indifféremment du nombre de processeurs.

Notre méthode consiste à utiliser une seule racine globale, sans utiliser des racines locales aux processeurs (voir la section 1.3 pour la génération des nombres aléatoires). Additionnellement, des racines temporaires, qui ne dépendent pas du nombre de processeurs, ont été utilisées. L'utilisateur spécifie seulement la racine globale, et toutes les racines temporaires sont calculées à partir de cette unique racine et d'autres informations, comme le numéro du cycle et le numéro d'identification de l'agent.

Pendant la simulation, les nombres aléatoires apparaissent dans trois cas :

- Aucun agent n'est concerné. Ce cas apparaît seulement pendant l'initialisation, donc nous utilisons la racine globale.
- Un seul agent est concerné. Ce cas apparaît quand un agent fait un mouvement aléatoire, quand il crée un autre agent dans un endroit aléatoire de l'environnement ou tout simplement quand un agent a besoin d'un nombre aléatoire. Nous utilisons dans ce cas une racine créée juste avant l'activation de l'agent et initialisée par une fonction dédiée, présentée ci-dessous.
- Plusieurs agents sont concernés. Ce cas apparaît seulement dans les conflits. Nous ordonnons dans ce cas les agents suivant leur numéro interne d'identification, qui est unique. Ensuite, nous choisissons le gagnant en utilisant un nombre aléatoire généré à partir d'une racine créée par la même fonction dédiée (voir ci-dessous).

Cette fonction dédiée doit générer des racines différentes lors de ses différents appels. Elle doit être paramétrable, donc utiliser la racine globale

⁴Sous réserve de l'efficacité du générateur de nombres aléatoires fourni avec le système d'exploitation.

donnée par l'utilisateur. Elle doit aussi être spatiale et temporelle, pour générer des racines différentes dans des cycles de simulation et/ou des cases de l'environnement différents. Dans notre bibliothèque nous avons donc choisi la fonction suivante, qui remplit toutes ces caractéristiques :

```
seed=gseed+7·cycle+y·dimx+x,
```

où (y, x) est la position du conflit, $dimx$ est la dimension de l'environnement sur l'axe x et $cycle$ est le numéro du cycle en cours.

Utilisation de la bibliothèque

Pour utiliser la bibliothèque `ParSSAP` il faut inclure le fichier `agents.h` dans les fichiers de l'application et utiliser la bibliothèque `agents` pendant l'édition de liens.

Nous allons présenter les principales parties d'une application « collectionneurs de diamants ». Il s'agit d'un monde virtuel de 50x100 cases avec 50 cases inaccessibles (obstacles) et 15 diamants. Dans ce monde se trouvent 5 collectionneurs qui cherchent les diamants. Ils se déplacent aléatoirement et ils prennent tous les diamants qu'ils rencontrent. Nous nous intéressons à l'efficacité de ces collectionneurs pendant un certain nombre de cycles.

La création de l'environnement se fait avec les lignes suivantes :

```
44 pmCreateEnv (DIMY, DIMX, EDGES);
45 pmPutRandomObstacles (OBSTACLES);
46 pmPutRandomResources (DIAMONDS, 0, 1, 0, NULL);
47 pmPutRandomAgents (WORKERS, 1, NORM_PR, 100, NULL, 0, take_or_random);
```

Le comportement des collectionneurs est décrit par les lignes suivantes :

```
15 pmAction_t take_or_random (void *mem)
16 {
17     pmAction_t act;
18     // if found a resource with a diamond, take it
19     if ((pmlmGetSquareType() == RESSQ) && (pmlmGetSquareResLoad() == 1)){
20         act.type = TAKE;
21         act.param = 1;
22     }else // else do random movement
23         act.type = MOVERANDOM;
24     return act;
25 }
```

Enfin, la simulation finit quand un certain nombre de cycles est passé :

```
30 int runfunc (void)
31 {
32     if (pmgGetCycle() >= CYCLES) // end the simulation, if max count reached
33         return STOP;
34     else
35         return CONT;
36 }
```

Comme partiellement décrit dans cet exemple, la méthodologie de développement d'une application utilisant la bibliothèque `ParSSAP` est :

1. écrire le code ou le fichier de données pour l'initialisation de l'environnement ;
2. écrire la fonction utilisateur, qui permet, entre autres, de spécifier la fin de la simulation ;
3. écrire la fonction qui donne le potentiel de chaque ressource en fonction de sa charge ;
4. écrire le comportement de chaque agent.

Plus d'informations sur les fonctions fournies par la bibliothèque sont données dans son manuel de référence (annexe A).

1.9 Domaines d'application et performances globales

Notre bibliothèque permet de simuler une grande variété de systèmes multi-agent. Par exemple, nous avons implanté le jeu de la vie de Conway [39], basé sur des cellules statiques et la création et la destruction dynamique de ces cellules. Nous avons aussi implanté un simulateur plus complexe, présenté ci-dessous.

Simulateur multi-consommateur

Nous essayons de simuler ici l'application suivante. Plusieurs personnes habitent dans un village. Le village contient leurs maisons, des magasins où ils font leurs achats et d'autres bâtiments non intéressants (appelés simplement bâtiments). Le but des simulations est d'étudier l'efficacité de cette population à faire ses achats et à les ramener à domicile.

Le village. Pour le village, nous avons utilisé un environnement borné avec 25x25 cases (pour la mesure des performances nous utiliserons un environnement plus grand). Dans cet environnement nous avons créé deux régions : une première région, sans bâtiments, à la périphérie du village, contenant les maisons, et une deuxième région, avec de nombreux bâtiments, en centre du village, contenant les magasins. Ce village est illustré sur la figure 1.5. Chaque magasin contient 10 objets destinés à être achetés, simulé par une ressource avec une charge de 10. Leur potentiel est constant et couvre tout l'environnement.

Le comportement des personnes. Nous avons créé trois types de personnes :

1. Les non-consommateurs, qui se promènent dans le village sans but précis et sans rien acheter (les *enfants*).
2. Les consommateurs inexpérimentés et déterministes, qui vont dans les magasins *dans un ordre prédéfini*, font des achats et les ramènent à leurs maisons (les *hommes*).

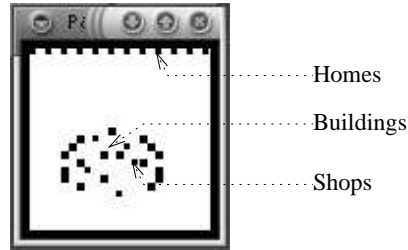


FIG. 1.5 – Le village utilisé dans le simulateur multi-consommateur, avec la région des maisons en haut, et la région des cinq magasins et des bâtiments (plus petits que les magasins dans cette figure) en centre.

3. Les consommateurs expérimentés, qui agissent comme les précédents à l'exception que, s'ils sont impliqués dans un embouteillage, ils changent l'ordre de visite des magasins et se dirigent vers d'autres magasins (les *femmes*).

Aucune personne n'a une vision globale du village. Elles n'utilisent que leurs perceptions locales et leur mémoire propre pour visiter tous les magasins et faire leurs achats. La perception locale qu'elles utilisent est la perception du gradient de potentiel, qui les dirigent vers les magasins.

Efficacité des personnes. Nous avons fait des tests dans lesquels nous avons mis 2 enfants, et 10 hommes et femmes. Nous avons varié le nombre d'hommes/femmes de 0 à 10 et nous avons constaté son influence sur l'efficacité du système. L'efficacité du système est donnée par le pourcentage du nombre d'objets achetés et ramenés jusqu'aux maisons.

Grâce aux informations que la bibliothèque fournit sur le mouvement des agents, nous avons pu aussi visualiser le trajet de chaque personne. Nous avons ainsi découvert que les personnes se bloquent souvent entre elles. La figure 1.6 présente quelques captures de fenêtre avec ces situations.

L'efficacité de ces personnes dans notre simulation, avec les achats ramenés aux maisons et le nombre de cycles de simulation nécessaire, est donnée dans le tableau 1.1. Une étude plus poussée de l'efficacité de ces comportements peut être faite avec **ParSSAP**.

Performances. Pour la mesure de performances nous avons utilisé un environnement de 256x256 cases, avec 655 personnes (1 % des cases), 655 maisons (1 % des cases), 163 magasins (0,25 % des cases) et 327 bâtiments (0,5 % des cases). Parmi les personnes, 20 % sont des hommes et 80 % sont des femmes. Le potentiel de chaque ressource couvre tout l'environnement. Chaque ressource a un potentiel constant de 512 et nous avons donc propagé leurs potentiels seulement en début de simulation.

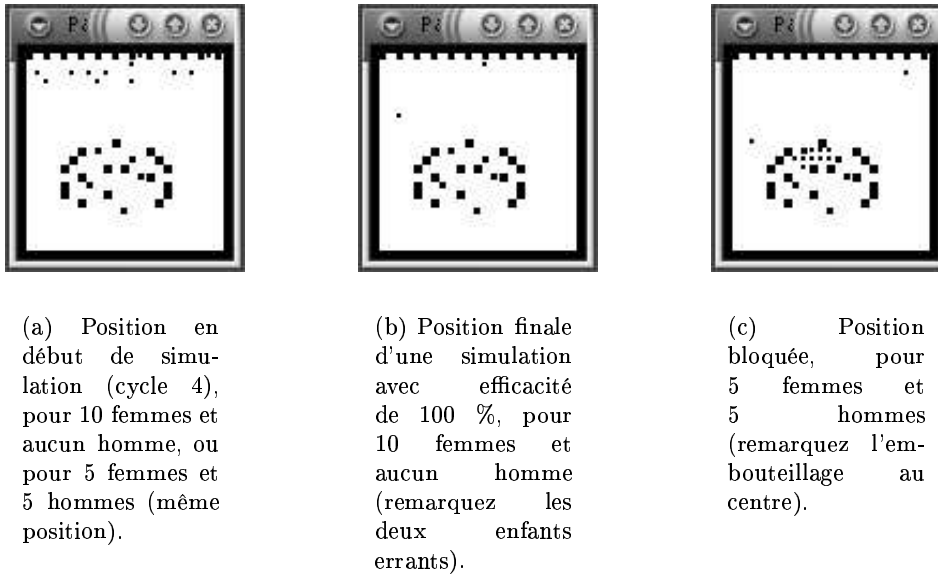


FIG. 1.6 – Captures d'écran de la simulation multi-consommateur.

<i>Hommes</i>	<i>Femmes</i>	<i>Achats</i>	<i>Cycles</i>
0	10	100 %	178
1	9	100 %	168
2	8	26 %	251
3	7	50 %	251
4	6	50 %	251
5	5	0 %	251
6	4	0 %	251
7	3	0 %	251
8	2	0 %	251
9	1	0 %	251
10	0	0 %	251

TAB. 1.1 – Efficacité des personnes dans la simulation multi-consommateur.

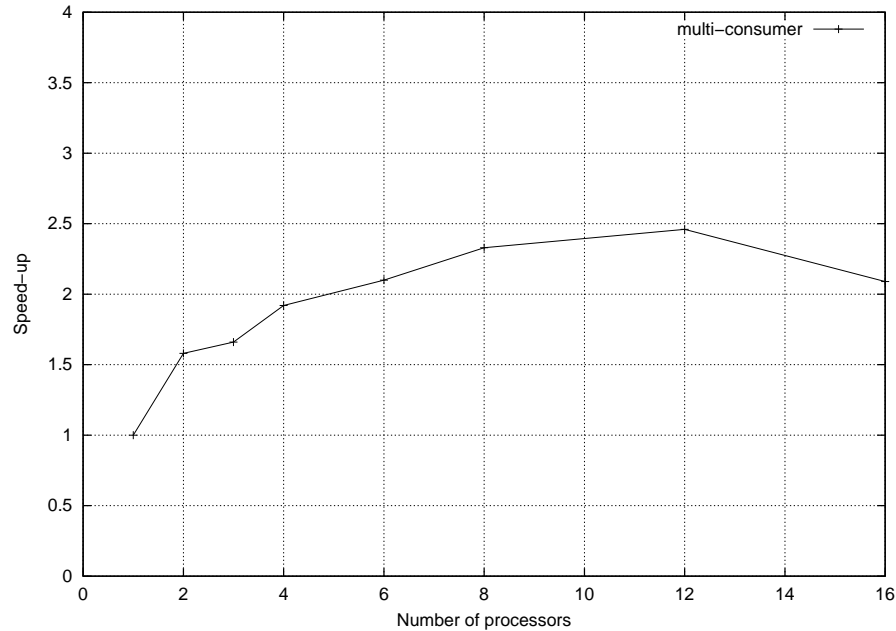


FIG. 1.7 – Accélération de la simulation multi-consommateur (peu favorable à notre parallélisation), qui augmente lentement et atteint presque 2 sur 4 processeurs (DSM Origin 2000).

Nous avons exécuté cette simulation pendant 400 cycles, ce qui s’est avéré insuffisant pour que toutes les personnes ramènent leurs achats à la maison. Nous avons supprimé les entrées/sorties et nous avons mesuré le temps total de la simulation (« wall-clock time »).

L’exécution sur la machine DSM Origin 2000 a donné un temps séquentiel de 55 secondes. L’accélération est présentée sur la figure 1.7. Elle augmente lentement jusqu’à 12 processeurs.

Sur la machine SMP avec 4 processeurs nous avons obtenu un temps séquentiel d’exécution de 260 secondes. L’accélération est ensuite comparable à celle de la machine DSM.

Cette application a montré que notre bibliothèque est suffisamment flexible pour implanter rapidement et simuler une telle application. En fait, toute l’application utilisée pour nos mesures de performances a 240 lignes de code source. En revanche, cette application n’est pas très adaptée à notre parallélisation : le potentiel de chaque ressource couvre tout l’environnement et plusieurs repropagations sont nécessaires, et l’environnement n’est pas homogène : les agents tendent à se concentrer vers le centre. Néanmoins, nous avons obtenu une accélération de presque 2 sur 4 processeurs (50 % d’efficacité) pour cette application originale.

1.10 Bilan et futurs travaux

Bilan de nos travaux

Les systèmes multi-agent (SMA) sont un domaine de recherche en croissance. Dans ce contexte, notre recherche traite la simulation de larges populations d'agents. Notre contribution s'est étendue dans trois directions.

Modèle de simulation. Nous avons tout d'abord conçu un modèle de simulation de SMA. En particulier, il prend en considération les mouvements simultanés des agents. Notre modèle contient quatre composants : (1) l'environnement, (2) les ressources, qui propagent des champs de potentiels, (3) les agents, qui se déplacent dans l'environnement suivant leur but et leurs perceptions, et (4) l'arbitre, qui a le rôle de maintenir la cohérence du système. Plusieurs fonctionnalités utiles dans la simulation de systèmes multi-agent sont offertes par la bibliothèque.

La simulation est discrétisée. Chaque cycle de simulation contient plusieurs sous-cycles, qui permettent de gérer la simultanéité des actions des agents, notamment les conflits spatiaux entre les agents.

Algorithmique parallèle. Nous avons également fait un intense travail algorithmique parallèle sur la simulation de SMA. Le calcul des champs de visibilité est basé sur le traçage de lignes épaisses. Un algorithme nouveau pour le traçage de telles lignes a été proposé. L'algorithme de vision est un algorithme régulier, et nous avons obtenu de grandes performances : accélération de 28 sur 32 processeurs (efficacité supérieure à 90 % jusqu'à 32 processeurs). Nous avons identifié des variations dans la courbe d'efficacité et nous avons trouvé leur cause dans un déséquilibre de charge entre les processeurs.

Le percept de détection des potentiels utilise les champs de potentiels propagés par les ressources. La propagation est basée sur le principe de Huygens [51] et constitue un algorithme irrégulier. Nous avons présenté plusieurs méthodes séquentielles et parallèles pour son implantation. Après de nombreuses mesures de performances, nous avons découvert les meilleures méthodes en temps d'exécution. Nous avons ensuite fait une comparaison détaillée de ces méthodes. Finalement, nous avons implanté une combinaison de ces méthodes et nous avons trouvé les cas où chaque méthode est la meilleure.

Une attention toute particulière a été accordée à l'aspect aléatoire et à la totale reproductibilité des simulations.

Implantation parallèle portable du modèle. Enfin, nous avons implanté et validé notre modèle par la création d'une bibliothèque, appelée **ParSSAP**. Nous y avons implanté la plupart des fonctionnalités du modèle. Elle est parallèle, mais la difficulté de la programmation parallèle est cachée

à l'utilisateur. Elle a été conçue pour une utilisation facile et pour de bonnes performances parallèles. Les hypothèses qu'elle fait sont suffisamment bonnes pour nos simulations multi-agent et suffisamment simples pour être efficaces en termes de temps d'exécution. Pour une application complexe, le simulateur « multi-consommateur », nous avons obtenu une modeste accélération de 2 sur 4 processeurs. Néanmoins, nous considérons que le temps de développement a été largement réduit, comme illustré par les seules 240 lignes de code de toute l'application.

Travaux futurs

Amélioration du modèle. Nous envisageons d'introduire la communication explicite entre les agents et le mélange des ressources et des agents, ce qui permettrait aux ressources d'être mobiles et aux agents de propager des potentiels.

Optimisations des algorithmes. Au niveau des algorithmes des percepts, nous envisageons, par ordre croissant de difficulté, d'optimiser les algorithmes de propagation des potentiels, d'implanter et d'évaluer d'autres algorithmes de propagation de potentiel, et de permettre une propagation inexacte, mais plus rapide, des potentiels.

Au niveau des agents, nous envisageons introduire d'autres percepts, tout en gardant la simplicité et les performances de la bibliothèque à un niveau acceptable.

Au niveau de la simulation, nous envisageons d'optimiser les performances parallèles de la bibliothèque. L'amélioration la plus importante ici est de fournir un équilibrage de charge dynamique, très utile pour des simulations avec des environnements hétérogènes.

Amélioration de l'environnement de programmation. Pour une facilité encore plus accrue, nous envisageons l'implantation d'outils graphiques pour la création des environnements de simulation et pour la visualisation des résultats générés par les simulations.

Exploitation. Après la création d'un produit, une étape importante est son utilisation. Nous envisageons donc d'utiliser cette bibliothèque dans un cadre d'expérimentation et de recherche en SMA à Supélec par l'intermédiaire de projets d'étudiants et/ou de thèses de doctorat.

En fournissant un outil pour la simulation de larges populations d'agents, facile à l'utilisation et performant à l'exécution, nous espérons avoir apporté notre pierre à l'édifice de la recherche multi-agent et à celui de l'algorithmique parallèle.

Deuxième partie
Thesis dissertation

Contents

I	Introduction and in-depth abstract (in French)	13
1	Introduction and in-depth abstract	15
1.1	Motivations and introduction	15
1.2	Principles of multi-agent systems	17
1.3	Principles of parallel computing	19
1.4	Parallel multi-agent system implementations	23
1.5	ParSSAP model of multi-agent system simulation	25
1.6	Parallel algorithmic of vision percept	28
1.7	Parallel algorithmic of potential field propagation and perception	33
1.8	ParSSAP features and implementation	39
1.9	Domains of application and global performance	43
1.10	Conclusions and future work	47
II	Thesis dissertation	49
2	Motivations and introduction	57
2.1	Motivations	57
2.2	ParCeL language family	59
2.3	Features of our programming model	59
2.4	Features of our development tool	60
2.5	Hypotheses taken in our model and development tool	61
2.6	Plan of the dissertation	62
3	Principles of multi-agent systems	63
3.1	Main interests on multi-agent paradigm	63
3.2	Multi-agent system overview	65
3.2.1	Broad definition of agent concept	65
3.2.2	Types of agents	66
3.2.3	Cooperation <i>vs.</i> antagonism	67
3.3	Applications	68
3.4	Modelling of societies of situated agents	69

3.4.1	Modelling of space and time	69
3.4.2	Modelling of agents	70
3.4.3	Modelling of agents' percepts	71
4	Principles of parallel computing	75
4.1	Introduction	75
4.1.1	Benefits of parallelism	75
4.1.2	Measurement of performance of parallel programs . . .	76
4.1.3	Overview of the impediments of parallelism	78
4.1.4	Sequential impediments to performance of parallelism	78
4.1.5	Parallel impediments to performance of parallelism . .	80
4.1.6	Parallel programming complexity	83
4.1.7	Brief prospect on the future of parallelism	85
4.2	Parallel computer architectures	86
4.2.1	Flynn's taxonomy	86
4.2.2	MIMD machines	87
4.2.3	Impacts of cache memory on performance	90
4.2.4	Software development guidelines for cache access opti- misation	92
4.3	Parallel programming strategies	93
4.3.1	Sources of parallelism	93
4.3.2	Abstraction levels of parallel programming	94
4.3.3	Parallel programming paradigms	95
4.4	Shared-memory parallel programming paradigms	96
4.4.1	List of existing paradigms	96
4.4.2	Synchronisation techniques	97
4.5	Parallel and deterministic generation of random numbers . . .	97
4.6	Parallel disk input/output in multi-agent systems	99
5	Parallel implementations of multi-agent systems	101
5.1	Sequential multi-agent system simulators	101
5.1.1	<i>Pengi</i> , by Agre and Chapman	101
5.1.2	Improved <i>Pengi</i> , by Drogoul, Ferber and Jacopin . . .	102
5.2	Parallel multi-agent system simulators	103
5.2.1	PIOMAS error-accounting simulator	103
5.2.2	BioLand massively simulator	104
5.3	Development environments of parallel MASs	106
5.4	Conclusions	107
6	The ParSSAP model of multi-agent system simulation	109
6.1	Components of the model	109
6.2	Agents' behaviour	111
6.3	Simulation of the simultaneity of actions	112
6.4	Execution engine	112

6.5	Information about the state of the system and its evolution	114
6.6	The transparency of parallelism of the model	115
6.7	Conclusions	115
7	Parallel algorithmic of vision perception	117
7.1	Properties of the vision percept	117
7.2	Definition of the visibility field topology	118
7.3	Vision algorithm	120
7.4	Difficulties introduced by obstacles	121
7.5	Line and supercover line tracing algorithms	122
7.5.1	Line tracing algorithms	122
7.5.2	Supercover line tracing algorithms	124
7.6	Our supercover line tracing algorithm	126
7.7	Optimisations of the vision algorithm	128
7.8	Parallel issues of the vision algorithm	131
7.9	Memory and timing models	133
7.9.1	Memory model	133
7.9.2	Timing model	134
7.10	Performance of the vision algorithm	135
7.10.1	Experimental condition overview	135
7.10.2	Vision algorithm implementation	136
7.10.3	Influence of parallel memory allocations	137
7.10.4	Performance measurement results	141
7.10.5	Influence of caches	145
7.10.6	Influence of the vision parameters	146
7.10.7	Overall performance of the vision algorithm	153
7.11	Conclusions	154
8	Parallel algorithmic of potential field propagation and perception	157
8.1	Properties of the potential gradient	157
8.2	Definition of the potential field topology	159
8.3	Difficulties introduced by obstacles	159
8.4	Wave propagation model	159
8.4.1	Wave propagation of one resource	160
8.4.2	Overlapping of potential fields	162
8.5	Wave propagation implementation overview	162
8.6	Sequential methods	164
8.6.1	Recursive methods	164
8.6.2	Iterative methods	167
8.6.3	Distance-storing methods	170
8.6.4	Other methods	175
8.6.5	Theoretical comparison of sequential methods	176
8.7	Parallelisation methods	176

8.7.1	Fixed domain decomposition	177
8.7.2	Changing domain decomposition	184
8.7.3	Data decomposition with mutexes	187
8.7.4	Data decomposition with processor-private environments	188
8.7.5	Theoretical comparison of parallel methods	192
8.8	Performance of the wave propagation algorithms	192
8.8.1	Experimental condition overview	192
8.8.2	Wave propagation algorithm implementation	192
8.8.3	Performance of each method	194
8.8.4	In-depth comparison between the best methods	201
8.8.5	Final best method	210
8.9	Inexact potential propagation	211
8.10	Conclusions	213
9	ParSSAP features and implementation	215
9.1	Features of our library	215
9.1.1	General features	215
9.1.2	Information about system's state and evolution	216
9.1.3	Parallel issues	217
9.2	Simultaneity processing and parallelism in the library	218
9.2.1	Internal structures	218
9.2.2	Execution flow	219
9.2.3	Randomness and reproducibility of simulations	222
9.3	Compiling information	224
9.3.1	Compiling the library	224
9.3.2	Compiling an application	225
9.4	Execution of a simulation from the user point of view	225
9.5	"Hello world" example	225
9.5.1	Implementation and explanations	226
9.5.2	Richer "Hello world"	227
9.6	"Diamond collector", a more complex example	228
9.6.1	Specifications	228
9.6.2	Implementation and explanations	229
9.7	"Carrier robots", an example with potential fields	231
9.7.1	System initialisation	231
9.7.2	User function of end of cycle	233
9.7.3	Functions related to resources	233
9.7.4	Agent behaviour function	233
9.7.5	Simulation	234
9.8	Generic design of an application	236
9.9	Conclusions	236

10 Domains of application and global performance	237
10.1 Conway's game of life	237
10.2 Multi-consumer simulator	240
10.3 Conclusions	257
11 Conclusions and future work	259
11.1 Main results of our research	259
11.2 Future work	261
A ParSSAP reference manual	263
A.1 General considerations	263
A.2 Data structures and types	264
A.3 Simulation start and end functions	265
A.4 Environment creation functions	266
A.5 Simulation configuration functions	268
A.6 Input/output functions	271
A.7 Agents dynamic creation and destruction functions	273
A.8 System information functions	274
A.9 Full-featured example of system files	276
A.10 Full-featured example of statistic file	278
B Implementation of the supercover line tracing algorithm	281
C Publications written during this thesis	283
C.1 International conferences with lecture committee	283
C.2 National conference with lecture committee	284
Bibliography	285

Chapter 2

Motivations and introduction

2.1 Motivations

There are large and distributed systems where an efficient or an optimum solution has to be found. Examples of such systems are traffic simulation and simulation of ant populations. Such systems are too complex for a total planning. A promising way to cope with such systems is the distributed computing, where the knowledge of the problem is not central, but spread in several points of the system.

Currently, an appropriate model to simulate such systems is the multi-agent model. An agent is an autonomous entity found in an environment, which perceives it and acts on it. Its knowledge and its perceptions and actions are local, hence simpler than a centralised system. Agents move in the environment and run concurrently in the system. They have goals and can communicate to reach their goals. Sometimes, the cooperation, which can be explicit, as well as indirect, among simple agents can lead to an efficient global system, concept known as *emergence*. It is interesting to look for self-organisation of populations of agents with basic behaviour and knowledge. We have chosen to focus on situated multi-agent systems, where agents are constrained by physical laws, such as obstacles which prevent agents to move in some places.

No general, nor efficient theory exists yet for this kind of systems. A classical approach in finding mechanisms for self-organisation consists in doing many experimentations of such systems. In fact, there are many parameters which influence the performance of the system, and we are first trying to understand their effects on the global behaviour. However, simulations can take a lot of time to execute and to be written. A methodology of writing such systems would allow their faster implementation, while their execution on parallel architecture would allow decreased execution times.

The simplicity of writing such programs is yet more necessary as researchers in artificial intelligence are not necessary specialists in parallelism.

Also, the optimisation and parallelisation of such systems allow to reduce their execution time, but the parallel programming is known to be more difficult than its sequential counterpart. This is especially true for multi-agent systems, which are irregular by nature, given that each agent can have its own behaviour and that they move in the environment. Even if the multi-agent model is naturally parallel, as there are many agents running concurrently, this parallelism is different than that provided by modern parallel architectures, where the overhead of parallelism (such as memory coherence) is often a real issue. Also, it is known that, unlike general programs, which are written once and executed a lot of times without modification, programs written by researchers are meant to be executed a few times and modified frequently, according to their results. Therefore, the balance between execution time and development time is no more unbalanced, and the development time becomes an important issue.

Currently, there is a lack of tools allowing the simulation of such systems. This is due in part to the generality of the agent concept. A few such simulators exist, but they are too simple or too specific. In this context, our aim is to write a tool which allow researchers in a given area of multi-agent systems to quickly experiment behaviours of agents. Our first focus is the development time of experiments. In fact, if the development time is not much reduced compared to a version written from scratch, the user would rather prefer to write the application from scratch. Also, the difficulty of writing such applications, and their performance in execution time do not encourage people to test such systems. We provide our tool as a library written in C language, and the simplicity of use of our tool is our first concern.

Our second focus is the execution time: experimentations have to execute fast. We have chosen to provide a tool with a parallel implementation. As an example, if an experiment takes three days to complete, a parallel implementation six times faster will take only one night, and the user can analyse the results the next day. It is worthwhile to precise that our tool deals with parallelism transparently for the user of the tool. In fact, the only information where the parallelism is involved is the number of processors to use during the simulation. On the other side, we, as tool developers, have been faced with several parallel paradigms. Even if at any given time agents have local information and they make local computations, through the simulation agents move and the computations are global. Therefore, for simplicity of our programming and performance in execution time, we have chosen the shared-memory paradigm for programming, and shared-memory architectures for execution.

During the development of the tool we have needed to do research on parallel algorithmic applied to multi-agent systems. As examples, two useful percepts have been studied (vision and wave propagation), and the simultaneity of agent's actions has been fully taken into account.

The contribution of this thesis is threefold. Firstly, we have created

a *programming model* to simulate situated multi-agent systems. Secondly, we have done an *algorithmic work* on multi-agent systems, especially for implementation and analysis of wave propagation and vision algorithms, and for providing *fully reproducibility* of random simulations. Thirdly, we provide a parallel implementation of our model allowing researchers and other users interested in this domain to *quickly* develop *execution-efficient* simulations. The tool validates also our model.

2.2 ParCeL language family

Our contribution is not singular. It is a continuation of the work started in our team 12 years ago. ParCeL (Parallel Cellular Language) family's long-term goal is to provide parallel tools to ease the writing of distributed computation, especially situated agents and neural networks. A second goal is to use modern parallel architectures, since they evolve rapidly.

Compared together, tools provided by ParCeL family (figure 2.1 on the following page) are however different. We have first created a language, ParCeL-1 [81, 84], influenced by the actor model [2], which allows to implement neural networks and symbolic calculus (semantic networks). It works on today unused transputer [52] machines. The next generation is a virtual machine, ParCeL-3 [82], destined to simulation of multi-agent systems. For such simulations it replaces ParCeL-1, which was not sufficiently flexible, for instance it did not take into account cells' movement. It works on modern MIMD machines. However, ParCeL-3's goal was to be sufficiently generic, hence its use is not as easy as we have wished to. With ParCeL-4 [12] and our ParCeL-5, we try to find the right balance between power of expressiveness and development easiness. They are much more targeted: ParCeL-4 for neural networks and our ParCeL-5 for situated multi-agent systems. Instead, they are simpler to use and provide a higher level than their predecessors.

2.3 Features of our programming model

The simulations are done in a discrete environment. Each entity contained in the environment occupies one square. Three kinds of entities exist: obstacles, resources and agents. Obstacles prevent agents to enter their square. Resources' main goal is to guide agents. They can propagate potential fields which are perceived by agents, allowing them to find the way to resources, even in the presence of obstacles. The potential field size can change during the simulation. Moreover, several types of potential can be used, in order to introduce several kinds of resources. Also, resources can contain objects which can be carried by agents.

Agents are the mobile entities of the simulation. Each agent has a specific behaviour and a memory associated. Agents can be created and destroyed

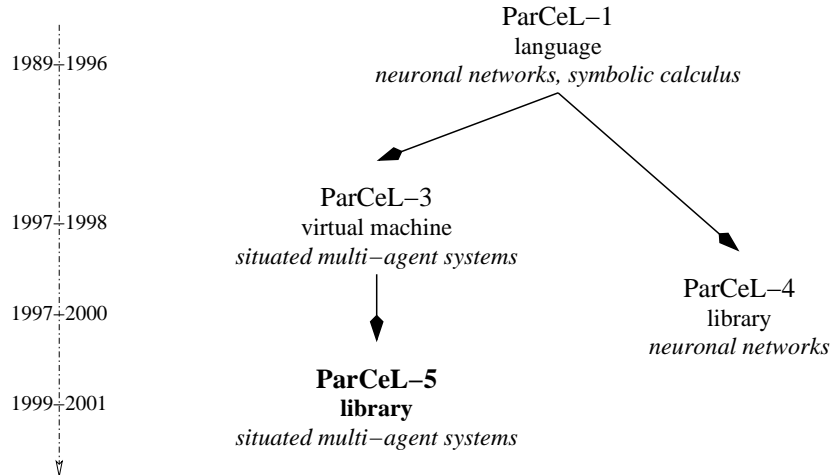


Figure 2.1: ParCeL language family.

dynamically. Additionally to the potential percept presented above, they can be also influenced by the vision percept, which allows them to see at a certain distance if no obstacle prevents them.

The time also is discretised and the simulation is based on cycles. During each cycle the potential fields can be recalculated. During each cycle each agent is activated. The behaviour of agents generates an action, based on their perception. Examples of actions are move and take/drop objects in resources.

The model takes into account the simultaneity of actions and solves the spatial conflicts among agents.

In a pure multi-agent system, agents have only a local perception. However, for the flexibility of the model global perception is also available, but a clear distinction between local and global perception is done.

Additionally, statistics about the state of the simulation for every cycle are taken into account.

2.4 Features of our development tool

We have implemented a library based on our model. It allows the simulation of agents on sequential machines and of large populations of agents on parallel machines. We summarise here its main features according to the model presented above.

In our implementation the environment is two-dimensional. Agents can move only one square, in its neighbourhood of 4 or 8 squares. The main actions an agent can do are move and take/drop an object. Agents have memory and two main percepts: the odour, allowing them to sense potential fields, and the vision. In order to bypass obstacles, a wave propagation

method is used to spread the potential.

Care has been taken to give to the user of the library a clear distinction between local and global perception of agents.

The library allows to save in files three kinds of information:

1. Statistics about the state of the system on each cycle, such as the number of agents and the number of empty resources.
2. Information about the agents, such as their move, and the resources, such as their load, useful to visualise the evolution of the system. A simple tool¹ to visualise this information is also provided.
3. The state of the whole system (checkpoint). The simulation may be continued afterwards using this information.

The library is parallel. However, the use of the library is eased by hiding almost entirely the parallelism to the user of the library. Also, regardless of the number of processors, the simulation takes into account the simultaneity of actions and generates a fully reproducible simulation.

Our library has surely limited features, nevertheless it allows the simulation of a broad type of applications, such as Conway's universe, based on local creation and destruction of cells, and city centre jams, characterised by people movements and spatial conflicts.

2.5 Hypotheses taken in our model and development tool

During the development of our tool, in order to keep a small development time, we have been forced to take some hypotheses. Some of them concern two important models of perception, namely the vision and the potential propagation (wave propagation), presented below.

The vision percept allows or prevent agents to see a given square. The only squares preventing the vision are the obstacles. Also, the vision radius for all the squares has the same value.

The wave propagation model used in odour perception of agents contain two important hypotheses. The first hypothesis is that the decreasing of potential is linear, equal to $p-d$, i.e. the potential of a square is the potential of the resource influencing it minus the distance to it. (Surely, we suppose that the potential influenced is positive, otherwise it is zero.) The second hypothesis is that when a square is influenced by several resources, it receives the maximum potential among them.

These assumptions are not always real. For example, for electrostatic fields the potential decreasing is in p/d^2 and the function used is sum. Nevertheless, our assumptions are simple to use and sufficiently flexible to allow

¹This tool has been written by other people in our team.

a broad domain of applications to be written, such as Conway's universe and traffic simulation. Moreover, they are simple and efficient in terms of execution time. Therefore, we have adopted these hypotheses, which are appropriate for the multi-agent system simulations we want to parallelise successfully.

2.6 Plan of the dissertation

The following two chapters present the principles of multi-agent systems and parallelism issues, describing the options which we have faced and presenting, with arguments, our choices. The following chapter describes some sequential and parallel implementations of multi-agent simulators, and we compare them with our model and its implementation. After a chapter describing our model of simulation of situated multi-agent systems, two other chapters deal with algorithmic issues: vision and wave propagation algorithms. Besides a chapter dedicated to the perspectives of our work, the remainder of this dissertation describes mainly the implementation and use of our tool: its features, examples of applications and the reference manual.

Chapter 3

Principles of multi-agent systems

This chapter introduces multi-agent system topics that are related to our thesis. It does not provide an exhaustive multi-agent system overview.

3.1 Main interests on multi-agent paradigm

If in the beginning the hardware, compared to the software, was the main part of computer science and engineering, nowadays the software is given a much greater attention. The applications are becoming more and more numerous and diversified, and the software engineering is becoming too complex for our minds [54]. New methods and models of programming, which deal with this complexity, are needed. The multi-agent approach promises to cope with the actual complexity of the programming.

For example, Gasser [40], from University of Illinois, USA, thinks that the agent-oriented programming is a step forward in the language programming. He states that programming technologies have always oscillated between two extremes:

1. flexibility of the programs, and
2. the need to control that openness and complexity (flexibility) with structure, with language design decisions.

The agent concept in programming deals with complexity in the same way as functions do (in the programming sense), but at a higher level. It directly supports autonomy, goal-directed behaviour, which Gasser calls “structured persistent action”: “You can now have a program that will persistently try to accomplish something” [40]. On the contrary, other key concepts in agent-based programming, like communication and different kinds of action, seem to him as not being fundamental.

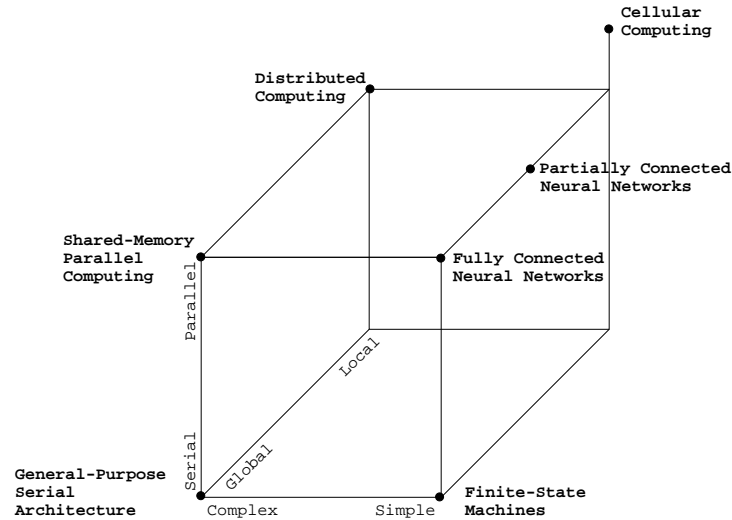


Figure 3.1: Compared to other paradigms, cellular computing allows simultaneously simplicity, *vast* parallelism and locality (image obtained through courtesy of M. Sipper [75]).

Sipper [75], from Swiss Federal Institute of Technology, Lausanne, thinks that the future of programming is the “cellular computing” (as he calls it), guided by 3 principles: simplicity, vast parallelism and locality. This leads him to place cellular computing at the other side of the *programming space*, at the opposite of the general-purpose serial systems, which are complex, serial and global, as shown in figure 3.1. He considers that cellular computing is a good candidate for image processing applications, finding fast solutions to a few NP-complete problems and fast calculating machines which involve arithmetic operations.

In [54], Jennings, from University of Southampton, UK, discusses in depth the engineering of agent-based systems. He identifies some of their key concepts, such as decentralisation/decomposition, abstraction and organisation, and he states that these systems are capable to “significantly improve the theory and the practice of modelling, designing, and implementing computer systems.” Moreover, the development of robust and scalable software systems *requires* cooperating autonomous agents which can deal with dynamic and uncertain environments. However, the multi-agent model has two major drawbacks, both concerning unpredictability:

- unpredictable interaction of the agents
- difficulty, even impossibility, of predictable behaviour of the whole system, due to the possibility of emergence of new and global behaviour.

Nowadays, agent-oriented paradigm continues to evolve. Ferber, from

Montpellier II University, France, sees four reasons which push on the development of this paradigm [32]:

1. The artificial intelligence (AI), in particular distributed AI (DAI), has reached a limit in organisation of knowledge [33].
2. It allows the simulation of real populations, such as societies of ants, in order to discover successful behaviours of individuals in groups.
3. It provides alternative methods in robotics: several small, elementary entities can sometimes be better than one big entity, while being also easier to develop.
4. The general tendency towards “distributed” systems, which means network, heterogeneity and parallelism, generates software cooperation, which is a key point of this paradigm.

The work of this thesis fits reasons 2 and 3. As we will see in chapter 9, it allows to simulate populations of autonomous entities, for finding emergence of global behaviour for example [1, pages 15–17].

3.2 Multi-agent system overview

Multi-agent systems are a relatively new domain of computer science, which nowadays becomes more and more popular. The first attempts date back to late 80s. In 1988, Ferber and Briot [33] were working on a concurrent language, Mering IV, for distributed AI. They wrote: “We believe that it is easier to study and model the activity of a social community than the intelligent activity of one man.” In 1993, Shoham [73] pushed on the definition of an agent and endowed it with mental components such as beliefs, capabilities, choices, and commitments. It is worthwhile to note that some researchers [40, 54] consider agents as programming entities which extend objects.

3.2.1 Broad definition of agent concept

A multi-agent system (MAS) is a system where multiple agents are simulated. Nowadays the definition of agent is still unclear. Generally, the definition of an agent involves an *autonomous* entity which is in an environment. An agent has a goal. In order to accomplish its goal, it generally communicates with the environment (by sensing it with its percepts and by modifying it through its actions) and with other agents (figure 3.2 on the following page).

Ferber [32] finds two definitions of the agent: weak and strong. The weak definition uses only a few of the concepts usually developed in MASs and is very close to software engineering. It considers an agent as an entity which

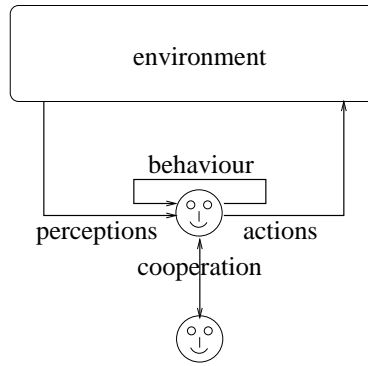


Figure 3.2: Illustration of the agent concept.

has a behaviour described by a “script” in the software sense, such as a procedure, and which can optionally travel on the computer networks. On the other hand, the strong definition takes into account all the usual key points of MASs: agents can communicate, an agent has its own goals, has resources and owns only a *local* knowledge about the environment. Local means they perceive and act only on their neighbourhood. The model of simulation presented in this thesis allows the implementation of weak agents. Additionally, in order to give a more realistic simulation of autonomous robots, a special attention has been paid to allow the use of only local information.

As defined, an agent is a possible model for many real societies of beings. Any being can be modelled by an agent, as they are autonomous, have a goal and have a local perception.

Many researchers have observed a similarity between the implementation of an agent and an object. However, as noticed by Gasser [40], Guessom [45] and others, an agent is also autonomous. Even an active object, which integrates an object along with its activity (namely a thread or process), cannot fully encapsulate the notion of agent, which is [40] “persistently trying to accomplish something”. We can consider that the design of an active entity has started by the objects, continued by active objects, processes and daemons and has finally arrived to agent. It has also been influenced by the *actor* concept [2], which is oriented mainly towards AI, and has a more specific model.

3.2.2 Types of agents

The actions of the agent are entirely given by its behaviour. This behaviour can use internal state of the agent and sensitive information, from environment or other agents. We distinguish two types of agents:

- *Reactive*¹ agents, which act instantly as a function of its inputs and

¹Ferber [31, page 207] defines a *tropic agent* as an agent which acts *entirely* by reflex.

which generally have no memory or state.

- *Cognitive* agents, which have memory and can include complex knowledge and inexact information, such as doing plans.

The first type is simple to implement and predict, while the second should give an artificial intelligence to the *agent* level. In practice, an agent often cannot be categorised in only one of these categories, because it inherits from both types of behaviour. For instance, as we will see in chapter 9, the model of simulation presented in this thesis is more suited to reactive agents, although it provides some useful functionalities from cognitive agents, such as memory.

A special kind of MASs are the *situated* MASs [31, page 16]. These systems try to mimic a world with topological constraints, such as our physical world. The modelling of the environment becomes important in such systems. The environment can be discrete or continuous. Generally, it is two-dimensional (2D) or three-dimensional (3D). In 2D, its shape is generally rectangular, and it can be toroidal (the left and right, up and bottom edges are respectively glued) or edged. The cells are generally square. The modelling of the environment is discussed in more detail in section 3.4.

The constraints in situated MASs are given by world laws [15], which deal with:

- System coherence, forbidding actions like entering an obstacle or moving to a long distance.
- Simultaneity, forbidding actions leading to a situation where several agents are simultaneously in the same place (we suppose only one agent can occupy a given place in the environment).

The model of simulation presented in this thesis is designed for simulation of situated MASs, and it uses a discrete, rectangular environment, toroidal or not. It pays attention to world laws by forbidding invalid actions such as those presented above.

3.2.3 Cooperation vs. antagonism

The communication is used by the agents to exchange information. In particular, when the agents have similar goals, they use it to avoid conflicts when they try to access a shared resource. Without this communication, the possibly resulting conflicts among agents can lead to natural antagonism. Table 3.1 on the next page presents the classification made by Ferber [31, page 74] about the interactions among agents by considering the compatibility of their goals, their skills and the number of resources.

<i>Goals</i>	<i>Resources</i>	<i>Skills</i>	<i>Types of situation</i>	<i>Category</i>
Compatible	Sufficient	Sufficient	Independence	Indifference
Compatible	Sufficient	Insufficient	Simple collaboration	Cooperation
Compatible	Insufficient	Sufficient	Obstruction	
Compatible	Insufficient	Insufficient	Coordinated collaboration	
Incompatible	Sufficient	Sufficient	Pure individual competition	Antagonism
Incompatible	Sufficient	Insufficient	Pure collective competition	
Incompatible	Insufficient	Sufficient	Individual conflicts over resources	
Incompatible	Insufficient	Insufficient	Collective conflicts over resources	

Table 3.1: Classification made by Ferber [31, page 74] about the interactions among agents, function of their goals, resources and skills.

However, this antagonism is not always a disadvantage. Dagaëff et al. [24] discuss antagonism *vs.* cooperation of agents. The combined actions of autonomous agents naturally induce an emergent antagonism, which is generally avoided by explicit and direct cooperation. The authors show that, even in the presence of this antagonism, cooperation can emerge, and it is sometimes possible to make advantage of this antagonism.

In our model the communication among agents cannot be done directly. Nevertheless, agents can communicate indirectly, through the environment for example.

3.3 Applications

A model cannot become viable unless the apparition of tools to construct applications and applications themselves. Ferber [32] does a survey of the multi-agent architectures, development environments and applications.

The applications of MASs are successfully used in industry and in research. Jennings and Wooldridge, in “Applications of Intelligent Agents” [53], “identify and distill the key conceptual foundations of agent-based computing and present them in the context of a variety of commercial and industrial application domains.” Some of the domains addressed are telecommunications, portfolio management and financial services. Wang [85] presents the benefits of agent technology in real-time process industry like power plant. “A scalable agent-based architecture enables it to enjoy a level of flexibility that cannot be found in traditional industry systems.” Ferber [32] shows several domains of multiple expertise, such as medical diagnostics, shapes recognition and natural language comprehension, which can be designed with agent technology.

The multi-agent model is often used in simulations. There are simulations of real and possibly difficult applications, such as the car traffic in a city, in order to avoid traffic jams, or simulation of distributed intelligence

applications, such as the sport teams [78], where each player is modelled by an autonomous and goal-oriented agent. There are also simulations of real multi-robot applications. To find appropriate behaviours of the robots, a lot of tests are needed. This cannot be done in reality, but only with simulations, each robot being modelled by an agent. In both cases, the execution time of the simulation can be an important parameter. Later, the automatic parallelisation provided by our model of simulation will be presented, as a method to decrease the execution time of the applications.

An example of such simulation is the Robot World Cup Initiative². This is an international research and education initiative which provides a framework and a competition for several types of artificial football game, from simulation to real robots. Its goal is to push on the research in MAS domain. Tabme et al. [78] deal with two challenges of MASs: multi-agent collaboration (or teamwork) and learning. As the model of teamwork, they use a hierarchy of agents. As learning, they use both off-line learning (coded by the programmer) and on-line learning (developed by the agent itself during training) in order to improve and specialise agents' individual skills in RoboCup competition. They notice that both off-line learning and on-line learning have been critical in improving agents' skill.

More generally, a category of applications suitable to multi-agent simulations is artificial life. They model societies of entities and study their evolution. In this case, the goal is generally to discover the emergence of interesting properties, such as collective intelligence, where simple individual behaviours lead to a powerful global behaviour [1, pages 15–17].

The model of simulation presented in this thesis can be appropriate for simulations such as those presented above.

3.4 Modelling of societies of situated agents

The topological constraints of the environment in SMASs lead to several specific features of these systems. Their modelling uses notions of space, time, movement, as described in the following.

3.4.1 Modelling of space and time

The environment of experiments is generally discrete, a two-dimensional (2D) grid of rectangular shape. The cells are generally square, but hexagon cells or other shapes can be imagined. *Pengi* [4] and its new implementation [30] (called in the following *Pengi-2*) use 2D rectangular environments formed by squares. *BioLand* [86] uses a rectangular toroidal 2D environment. Our model of simulation allows generally environments both toroidal and edged.

²RoboCup, Web site: www.robocup.org.

Several objects can also be considered as belonging to the environment, such as obstacles, resources, active (mobile) objects. The most simulators include such objects. However, **BioLand** environment does not provide obstacles. This influences the algorithms used.

The computer simulations are by definition time discrete. This can be divided in two categories: discrete time, such as cycle-based ([15] and our model) or discrete event time.

If the time is discrete (in cycles), the simultaneity of actions (which appears in reality) has to be taken into account. This can be done by dividing every cycle in several subcycles. The **PIOMAS** simulator [14] for instance uses *four* subcycles, which allow to deal with spatial conflicts. **BioLand** [86] uses *seven* subcycles, which allows to update the environment and to simulate the biots. Our model of simulation uses a virtual machine with *five* subcycles: (1) perception and decision, (2) conflicts avoiding, (3) action, (4) user function call, and (5) percept and environment update. The first and the third cycles mark the difference between planning of agents action and their effective action. The first and the fourth subcycles are provided by the user of the tool. The other subcycles are done automatically by the tool.

3.4.2 Modelling of agents

Modelling of agents' movements

It can be exact, such as in **Pengi** simulator [30], or inexact, such as in **PIOMAS** simulator [15]. **PIOMAS** takes into account uncertainties at both levels of effectors and sensors, based on a probability table.

Also, in some simulators [15] the agent has two actions concerning the move: rotate and go ahead, while in others (such as our tool) it can move directly in any direction, without needing to rotate.

The movement, compared to other agent actions, has generally the property to potentially engender spatial conflicts in situated agents simulations. However, **BioLand** [86] seems to not take into account such conflicts.

Our tool offers also a random movement in any direction, used for simplicity of programming. A special care has been payed to provide random but deterministic movements.

Modelling of other actions

Besides the movement, the agents can have other capabilities, such as taking objects from the environment or dropping objects they possess. It can be done simultaneously with a movement (such as our tool) or not. The side-effects of these actions can be taken into account entirely by the simulator or not. Our tool belongs to the former category, as it automatically increases/decreases the load of the agents doing such action and that of the resources concerned.

	<i>vision</i>	<i>odour</i>
<i>distance</i>	far	small
<i>obstacles</i>	highly disturbed	less disturbed

Table 3.2: Main user differences between odour and vision percepts.

Other actions, not present in our tool but sometimes simulable, are used in other simulators:

- Pushing blocks is a feature of **Pengi** [4, 30].
- Rotation of agents is a feature of the **PIOMAS** simulator [15]. The agents can move only in front of them.

Modelling of agents' reproduction and death

This can make use of genetics. Our tool provides a simple functionality for the reproduction and the death of agents, but does not provide a built-in functionality for genetics.

3.4.3 Modelling of agents' percepts

As seen in section 3.2, the agents generally use one or more percepts in order to fulfil their goals. The percepts are used to interact with the environment and/or to communicate with other agents.

It is worthwhile to note that the agents, by definition, have a local perception. For my part, I think this is the reality too: we do not see, for example, something at a given distance; it is the information which comes to us. A special attention has thus been paid to the library for locality: it provides special functions that allow the use of *only* local perceptions.

Several percepts are used in the literature [31]. We describe only two of them, which are used in our tool: odour and vision. The vision gives agents information not only about the adjacent cells, but on a larger domain. However, this percept is highly disturbed by obstacles. The odour, on the other hand, is not disturbed by obstacles, as shown below, but has generally a smaller radius of action, as shown in table 3.2. A combination of these two percepts seems useful to create efficient agents.

Odour

The odour (smell) percept tries to simulate the properties of the real odour, namely the fact that the nearer the resource, the stronger is the odour. An important consequence is that the potential field has to get around obstacles. This field can thus be used by agents for finding the way to resources, while totally avoiding obstacles. In the following we will call *potential* the value of the odour in one cell.

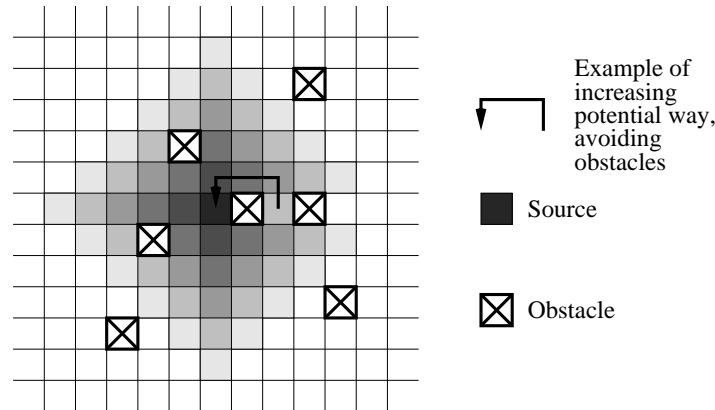


Figure 3.3: Illustration of the wave propagation model (4-connecting cells).

Most simulators [30, 59, 86], and our tool too, use the so-called *wave propagation* model for simulating the odour propagation through the environment. More information about potential propagation can be found in [31, pages 194–200 and 216–218]. For the exact propagation of one resource, this model starts by putting the right potential in the resource square and, like the waves, goes repeatedly further the resource while decreasing the value of the potential, as shown in figure 3.3.

While moving off the resources, the potential (as a function of the distance to the resource) is given by a strictly decreasing function. Let p be the potential of a resource, and d the distance between the resource and the square. Our tool uses a linear function $\text{pot}(d) = p - d$, while **BioLand** uses a square root function $\text{pot}(d) = p/d^2$ (for $d > 0$).

It is possible to have squares influenced by several resources. In this case, their potential is a function of all the potentials involved. **BioLand** uses the **add** function. Our tool uses the **maximum** function. This choice can have a high influence on the agent performance. The first function allows the creation of equipotential zones (their perception being unuseful, figure 3.4(a) on the facing page). The second function can absorb weak resources (figure 3.4(b)), nevertheless it provides always a path to a resource.

The cells of the environment can be 4 or 8-connected. This influences the spread of the potential. **BioLand**, **PIOMAS** simulators use 4-connected cells. Our model of simulation allows both connectivities.

The propagation of potential field, one part of the environment updating, is generally time consuming, as noticed by [86]. Two methods can be used to speed up this propagation. The first is to parallelise it. Several problems appear however, as a square can be influenced by several resources, and simultaneous accesses to variables appear.

The second is to create inexact gradient fields, which can potentially increase the speed of updates. This is not always a drawback, as agents are

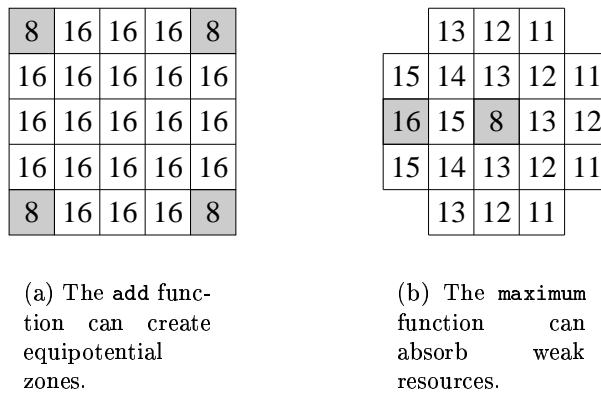


Figure 3.4: Problems of gradients models.

autonomous entities, so they have to adapt themselves to the environment. BioLand and our tool use always exact gradients. For more information see chapter 8.

Vision

The vision is a percept which allows agents to see what a certain square contains (agent, resource etc.) Contrary to the odour percept, this square can be far from the agent, and is highly affected by obstacles (which prevent its propagation). In particular, this percept is very useful for avoiding agent agglomeration, one point which can decrease the system global performance.

The vision can be global (all the environment is seen by the agent) or local to the agent (only a part of the environment, near to the agent, can be seen). If local, the region allowed to be seen traditionally contains all the squares situated at a distance (Euclidean or not) inferior to a certain number, called *vision radius*.

A square A can be seen from a square B if the *continuous* line from the centre of A to the centre of B does not intersect any opaque square. Traditionally, opaque squares are obstacles, but other objects, such as mobile objects or agents themselves, can be used. In order to compute the visibility domain from a square, modified lines drawing algorithms can thus be used. In general the visibility domains for squares are computed either statically, at the beginning of the simulation, or dynamically, during the simulation and when needed. A static computation can sometimes be preferable, in the case when this computation is time-consuming. However, it does not work when the visibility domains change run-time (opaque squares change run-time).

In chapter 7 several line tracing algorithms are presented together with the one we have used.

Chapter 4

Principles of parallel computing

This chapter introduces parallelism topics that are related to our thesis. It does not provide an exhaustive parallel computing overview.

4.1 Introduction

4.1.1 Benefits of parallelism

There are many applications whose execution time is either critical or too high to be useful. Some of them use numerical computations, and appear in image processing and forecast¹ fields for example. A significant decreasing of their execution time can sometimes be obtained by using several processors for their execution. Parallelism is the domain of computer science which involves the use of several processors for the execution of the program.

The beginnings of parallelism date back to sixties [5, page 31]. The “first supercomputer”, ILLIAC IV, was designed in 1967 and had 64 processors. It lived for several years and was eventually replaced by a faster mono-processor machine.

Nowadays, the parallel machines are used at large scale. Bal et al. [10] see three different reasons for the use of parallelism, with respect to the paradigm used:

1. Problems exhibiting inherent parallelism are better designed with a parallel programming paradigm. In this case, the parallel programming becomes the paradigm to be used. An example is a discrete event simulator, dealing with multiple entities: a convenient approach is to have one piece of code for every entity, and possibly run all these entities in parallel.

¹One can imagine that, if forecasting the next three days takes four days to the application to execute, its result is certainly unuseful.

2. Nature of some applications requires to run them on multiple machines connected to a network. The distributed programming paradigm is then appropriate, as these applications have not sequential counterpart. Examples are electronic mails, and banking or airline reservations.
3. Performance, more precisely execution time, can be critical for some applications. Here, the parallel paradigm does not always help to write the program, but it is useful to decrease its execution time or to run bigger problems. Examples of such applications are forecast and nuclear simulations.

The tool presented in this thesis uses the parallel paradigm for the first and the third reasons. An MAS is by definition a collection of autonomous entities, so a parallel programming paradigm is appropriate. In addition, if the native parallelism of an MAS is identical or at least compatible with the parallelism offered by parallel machines, the simulation can also be sped up by using such a machine.

As this latter point is a matter of choice, we are mostly interested in the performance analysis of parallelism when its sequential counterpart exists. As the benefits of parallelism have already been presented, the rest of this section treats the difficulties it engenders.

4.1.2 Measurement of performance of parallel programs

The measure and analysis of the performance of a program is generally very complex, especially for parallel programs. One reason is that it involves many factors, both internal to the program, such as input data, and external to the program, such as the speed of various devices². Furthermore, the performance may vary a lot if changing these factors. Thus, a lot of carefully-selected tests need to be done, in order to obtain a measure of the performance of a particular program as accurate as possible. Sections 7.10 and 8.8 deal with the main tests done for the tool presented in this thesis.

Knowing the performance of a parallel program is useful for comparing it with other similar parallel programs, but it is also useful for comparing it with a sequential implementation. On the one hand, this sequential implementation can be the best sequential implementation of the application. A customer or the final user is certainly interested by this comparison, as it gives the real benefit of the parallelisation of the application. Comparing the best sequential implementation with the sequential execution of the parallel program allows also to know its sequential slowdown (or the overhead of parallelism in sequential execution). On the other hand, the sequential

²Such examples are the overlapping between communications and computations, whose relative speed can influence the performance, the processors load, as they can be used by other applications in the same time, cache and memory size, hard disk bandwidth and so on.

implementation can be the parallel program itself when executed either by a single thread/process, or on a mono-processor. When comparing with its own execution on a mono-processor, we obtain an indicator not on the efficiency of the algorithm itself, but only on its degree of parallelisation (its processor scalability).

Several indicators exist to quantify these comparisons, but the most often used is the *speed-up*. The speed-up of a parallel program is defined as the ratio between the sequential execution time and the parallel execution time: $S(P) = T(1)/T(P)$. Two points need to be addressed here:

1. In general, *time* may have several meanings. It may be the *wall-clock time*, i.e. the real, physical time elapsed between the start and the end of the application, or between two arbitrary points of the program. It may also be the *processor time*, i.e. the time passed by the processor *only*, for example without I/O or time passed for other applications in a multi-tasking environment. The *parallel* execution time adds different meanings to this. In both types of times, it may be the average, the maximum or the minimum time among the times passed by the processors.
2. The sequential time $T(1)$ may have different meanings too. It may be considered as being the execution time of either the same parallel program on one processor, or its *best* sequential implementation.

A speed-up equal to the number of processors used is called *ideal speed-up*. This is generally the maximum speed-up which can be obtained by parallelisation. However, we see three cases when the speed-up of a program can be superior to the ideal one (supralinear speed-up):

1. *Cache influence*. The reason is that the total cache of several processors is greater than the cache of one processor. A supralinear speed-up appears for applications whose data which is regularly used does not fit in one cache (sequential execution case), but fit in the caches of the processors involved (parallel execution case).
2. *Non deterministic work*. For some applications which end when a solution of the problem has been found, a supralinear speed-up can be obtained when the parallel version, because of the multiple starting points, happens to start with variants nearer to the solution than the sequential version.
3. *Modification of the algorithm*. This can appear involuntarily when parallelising some algorithms.

Unless otherwise stated, in the following of this thesis we are interested by methods which seek to minimise the total execution time of a single application running on a parallel system.

As seen above, the measuring of the performance is a sensitive issue, as it can influence the judgement of external people on an application. Many results are thus presented in chapters 7 and 8, in order to provide greater accuracy on program performance.

4.1.3 Overview of the impediments of parallelism

As shown in the previous section, parallelism can be a a solution in reducing the execution time of an application. However, this comes with a price.

Firstly, the performance of parallel programs is limited by several factors, both in software, such as sequential bottlenecks, and in hardware, such as communication among processors. Due to the nature of parallelism, which involves *several* processors, we classify parallel performance impediments based on two extremes, which degrade obviously its performance:

1. Too little use of parallelism, i.e. existence of sequential parts, which limits *theoretically* the performance by not using all the processors available for it³.
2. Too high use of parallelism, more precisely overhead of parallelism, such as memory contention and communication among processors, which limits *practically* the performance by the time constraints (latency and bandwidth) of the network connecting the processors.

On the other hand, Bull [18] divides them in temporal and spatial, and presents a hierarchical classification of all the parallelism overheads. Lester [60, page 15] also cites a list of such factors, described in the following two sections.

Secondly, a parallel program is often harder to write than its sequential counterpart. Depending on the type of the problem, the difference in difficulty can be less or greater. The next two sections present these difficulties in more detail, comparing them to their sequential implementation. Finally, sections 4.2 and 4.3 describe problems which are architecture-specific and software-specific.

4.1.4 Sequential impediments to performance of parallelism

Sequential parts of program affect parallel performance. There are two laws which give the theoretical upper limit of the speed-up, given the fraction f of sequential part of the program. The best known is Amdahl's law [6], which answers the question: given a program executed on 1 processor, how much time is gained when executing it on P processors (figure 4.1)?

³It can be argued that these unused processors can be used temporarily by other applications, but we are interested here only in the performance of a given application.

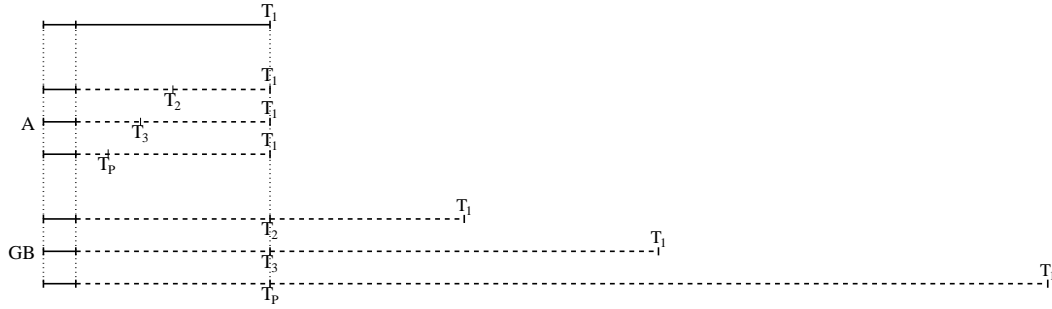


Figure 4.1: The sequential and parallel execution times used in the definition of the speed-up, as calculated by Amdahl’s law and by Gustafson-Barsis’ law.

Amdahl’s law [6]: Let T_1 be the execution time of a parallel program on a single processor, and f the fraction of T_1 which cannot be parallelised. Then the speed-up of the parallel program for P processors is upper limited by $S_{\max}(P) = \frac{1}{f + \frac{1-f}{P}}$ (hence $< \frac{1}{f}$).

As an example, if $T_1 = 100s$ and $f = 5\%$ ($T_{\text{seq}} = 5s$), the speed-up cannot exceed 20, regardless of the number of processors. If $P = 95$, then the limit of the speed-up is $S_{\max}(P) \approx 16$.

Some authors, like Gustafson [46], Lewis [61, page 13] and Roosta [69, pages 227–228], claim that Amdahl’s law is not appropriate for certain types of parallel programs, such as data-parallel ones. For these applications, the amount of potential parallelism increases with the size of vectors in the application, while the execution time of the sequential part (initialisation for example) remains constant, regardless of the data size. Gustafson-Barsis’ law⁴ [46] gives a measure of the data size scalability. It answers the question: given a program executed on 1 processor in time T_1 , how much time is gained when increasing data size and using P processors give the same execution time T_1 (figure 4.1)?

Gustafson-Barsis’ law [46]: Let T_1 be the execution time of a parallel program on a single processor, and f the fraction of T_1 which cannot be parallelised. If the natively-sequential time fT_1 is constant regardless of the data size of the program, then the speed-up of the parallel program for P processors, with increased data size and execution time of T_1 , is upper limited by $S_{\max}(P) = f + (1 - f)P$.

In the example above, if $T_P = 100s$ and $f = 5\%$ ($T_{\text{seq}} = 5s$), then the speed-up is limited by $S_{\max}(P) \approx 90$ when $P = 95$, and is no more limited

⁴We call it Gustafson-Barsis because Gustafson [46] writes: “... an alternative to Amdahl’s law suggested by E. Barsis...”

when $P \rightarrow \infty$. Compared to Amdahl's law, the limit of the speed-up is greater and is scalable.

It is worthwhile to notice that the two laws above are *not* incompatible. They only are applied to different parallel execution times and data sizes (figure 4.1 on the page before). Amdahl's law varies the number of processors P at constant data size W , which gives different execution times T , while Gustafson-Barsis' law varies P and W so that T remains constant. Otherwise said, there are two fundamental differences between the two laws, both appearing during the compute of the speed-up:

1. The *data size* of the program is unchanged for Amdahl's law, while for Gustafson-Barsis' law it increases.
2. The *execution time* T_1, T_2, \dots, T_P is decreasing for Amdahl's law, while for Gustafson-Barsis' law it is *maintained* constant. As a consequence, the fraction f of sequential time which cannot be parallelised is constant for Amdahl's law ($f_A = \text{constant}$) and is decreasing for Gustafson-Barsis' law ($f_{GB} = f(W)$).

The link between the two laws is illustrated in figure 4.2 on the facing page. It presents Amdahl's curve in four distinct cases, corresponding to sequential fractions of 0.01, 0.02, 0.04 and 0.08. Gustafson-Barsis' law takes the hypothesis that when the number of processors available increases, the data size of application is also increased. Considering a simple case, when the amount of computation of the parallel part has complexity $O(W)$, the four cases come from data sizes of $8W, 4W, 2W$ and W , respectively. As data size increases and execution time of sequential part decreases, Gustafson-Barsis' law speed-up simply "jumps" from worse to better Amdahl's law curves. As an example, for 28 processors the limit given by Amdahl's law is about 8, while the limit given by Gustafson-Barsis' law is much greater (e.g. about 22 for data size 8 times greater and constant execution time of sequential part).

The tool presented in this thesis contains a few small sequential parts. These will be discussed in chapter 9.

4.1.5 Parallel impediments to performance of parallelism

Natively parallel obstacles can also affect parallel performance. Some of them are low-level, namely memory contention, process creation time, communication delay and synchronisation delay. Others are higher level, such as load unbalancing and overwork. Finally, we take into account another important factor in parallel performance, the granularity.

Hardware impediments to parallelism

It is important to notice that:

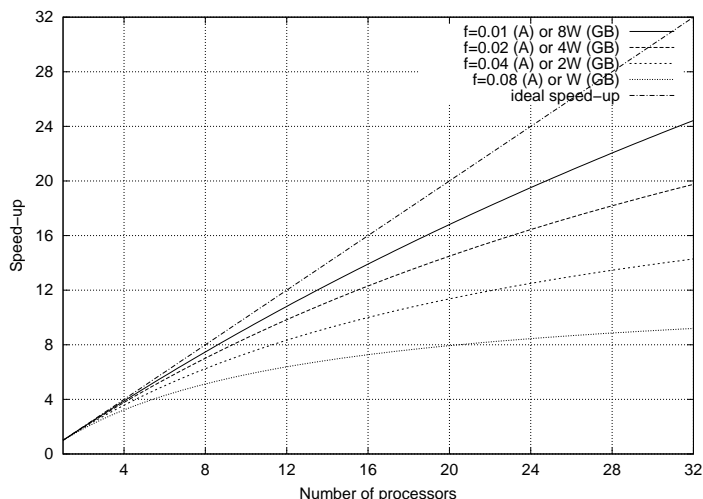


Figure 4.2: Link between Amdahl’s law and Gustafson-Barsis’ law: Gustafson-Barsis’ speed-up “jumps” from worse to better Amdahl’s law curves, increasing data size.

- all of them are not theoretical obstacles, but practical ones, given by the technology used for the architecture, and
- they are independent of the theoretical limits of the two laws given above and their influence is *added* to them.

These obstacles are presented below.

Unlike sequential applications, the parallelism involves *creation of tasks*, either implicitly or explicitly. If not done properly, for example when the time needed to create (and kill) a task is comparable to the duration of execution of the task, the overhead of task management becomes significant and reduces drastically the performance of the parallel program. This issue has influenced the tool presented in this thesis by constraining it to use the same number of tasks, to create them at the beginning of a simulation and to kill them only when the simulation ends.

Memory contention is specific to computers with shared memory and appears when several processors request data from memory at a higher speed than it can deliver data. Some processors are hence delayed. Memory contention acts at all the levels of memory hierarchy. For main memory, which is shared, the bandwidth may be insufficient to deliver the data in real time. As we will see later in section 4.2, this is the reason why the basic bus-based parallel architectures are not scalable. On the other hand, the caches are generally local to the processor, and, for cache coherence systems, it is the hardware which assures the coherence of caches. In this case, contention manifests when the caches are very frequently updated in order to guarantee

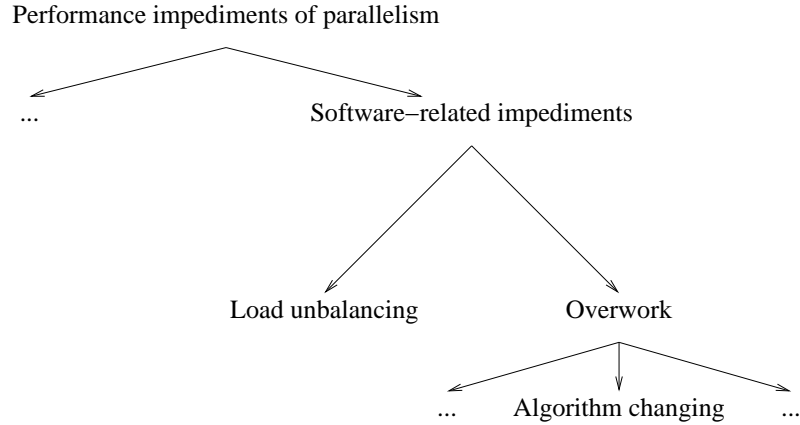


Figure 4.3: Software-related impediments to performance of parallelism.

their coherence [49], updates which do not exist in mono-processors. Several attempts have been done to cope with this difficulty, and more information can be found in section 4.2.4. The tool presented in this thesis is influenced by this issue mostly by the *false sharing* problem, as described later.

Communication delay is another issue of parallel programming, specific to multi-computers, where the processors communicate data through a network. This communication is necessary because the processors cooperate, more or less, to carry out the *same* task, in its entirety. Two parameters of network are of main interest: latency and bandwidth, defined in section 4.2.

Algorithms often cannot be divided in fully independent parts, processed by different processors, and a certain sequence of tasks, done with synchronisation, has to be done [5, page 13]. When the synchronisation is frequent, the *synchronisation delay* becomes important and can lead to memory contention also.

Software impediments to parallelism

The issues discussed above are low-level. They depend on the hardware performance and their absolute value decreases with the development of hardware. The next issues are high-level. They involve the software (figure 4.3).

Another source of parallel performance decreasing is the *overwork*. The parallelisation of some algorithms changes the algorithm and adds work to processors. This is the case for example of our iterative-fixed method of the wave propagation implementation (section 8.8.3, page 194). In sequential, the algorithm consists of a propagation. In parallel, an additional work of repropagation needs to be done to obtain correct results.

Finally, an important impediment is the *load unbalancing* of the processors. The sequential parts discussed before can be considered as a extreme case of load unbalancing, with only one processor working. In order to avoid

idle processors, the tasks given to the processors need to be as equilibrated as possible. A method is to statically divide the work to the processors. However, good load balancing can sometimes be difficult to obtain, especially when the application is irregular (the tasks durations are not the same), or dynamic and unpredictable (the tasks durations change unpredictably during the execution). In this case, a dynamic load balancing can be used [88].

Granularity

The factors presented above are responsible for parallel performance decrease of applications. It is also worth to notice an important parameter of applications which indirectly influences the performance: the *granularity*. Often, the granularity of an application is a measure of the frequency of processors synchronisation. Specifically, Almasi and Gottlieb [5, pages 13–18] call granularity the average size of the processor (individual) subtask, between two consecutive synchronisation or communication points. Thus, a *fine grain* parallelism means individual subtasks are relatively small in terms of code size and execution time; *coarse grain* is the opposite. The smaller the granularity, the greater the potential for parallelism (as given by load balancing) and hence speed-up. However, the smaller the granularity, the greater the overheads of synchronisation and communication.

Another definition of granularity concerns the number of tasks. Foster [37] calls a decomposition as being *fine-grained* if a large number of small tasks are defined. More generally, we can define the granularity as the ratio between the number of tasks and the number of processors. This definition is generally used when the first number is much greater than the second number. The reason of this definition is that in this case the parallel programming becomes more difficult, as one processor needs to process *several* tasks, and then problems of synchronisation and load balancing appear.

On the one hand, as the applications of the model and tool presented in this thesis often use a very high number of agents with simple behaviours, the tool has to offer a fine grain parallelism. On the other hand, as discussed below (section 4.2), we used MIMD machines, which provide medium and coarse grain parallelism. This means that, if from the software point of view the tool has to offer fine grain parallelism, from the hardware point of view the tool is bound to medium and coarse grain parallel architectures.

4.1.6 Parallel programming complexity

In addition to the performance impediments presented above, which decrease the execution time compared to the ideal one (sequential execution time divided by number of processors), often parallel programming is also more difficult than sequential programming, which increases the development time. This additional effort is less or greater, function of the nature of the algo-

rithm and of the parallel performance required. A good design of a parallel algorithm can sometimes increase drastically the parallel performance, as shown by our experiences [83].

However, it is worth to notice a distinction here. There are applications, in research for example, which are executed seldom. There are also applications, in forecast for example, which are executed often. Generally, higher is development time, lower is execution time. Sometimes, a tradeoff has to be done between these two times: Some algorithms can be optimised in order to minimise either the execution time, or the development time. We think that every simulation written with the tool realised in this thesis will be generally executed a few times, in order to discover if the current behaviour of the agents is productive. Skillicorn [76, page 131] notes:

“Long-term maintainability [of numerical or scientific programs] is emphasised less because of the research nature of such software. Many programs are intended for short-term use, generating results that make themselves obsolete by suggesting new problems to be attacked and new techniques to be used.”

In this case, the development time is as important as the execution time. This tradeoff has influenced the design of the tool presented in this thesis, as simplicity of programming (chapter 9) is its first goal, and execution performance the second one.

Another added difficulty of parallel programming is the decomposition of programs in sequential and concurrent tasks. Also, parallel programs contain parts which need to be executed in order, for example initialisation, user input waiting, computing and final result saving. *Synchronisations* among processors have to be made at appropriate places.

Additionally, in parallel programming objects shared by several processes need a special attention. If several processes modify simultaneously the same shared variable, different or incorrect results may appear. Such *race conditions* are avoided generally by synchronisations, such as `lock` and `unlock` functions⁵. Other methods of synchronisation are presented in section 4.4.2.

In MASs, the agents have to do *simultaneous* actions, which is not done the same in execution. For instance, it is possible that the simulator has already finished the execution of the behaviour of one agent when it starts the execution of another one, but no agent perceives it. In sequential execution, the simulation of the agents is done fully sequentially. In parallel execution, the behaviours of agents are executed sequentially for some, and parallelly for others. The reasons of not doing fully parallel execution is the fine granularity, as described in section 4.1.5. This simultaneity constraint has had consequences in the design (programming complexity) of the model

⁵This is also an example of synchronisation delay, as described in section 4.1.5.

and tool presented in this thesis and in its parallel performance, as described in chapters 6 and 9.

Another difficulty of parallel programming compared to sequential one is *portability*. Numerous different parallel architectures exist and, as shown by Skillicorn et al. [76], while the execution time of a sequential program changes by a constant factor when changing architectures, the execution time of parallel programs can change by an order of magnitude when changing parallel architectures. As a consequence, there is no single standard of parallel programming. Several attempts have been made, such as POSIX threads and `OpenMP`, however a lot of work is needed if the performance on several architectures is the most important parameter. This has direct implications to the maintenance also, because sometimes a great effort has to be done when the architecture changes. The tool implemented in this thesis uses the POSIX threads and Irix-native threads on SGI systems.

The *debugging* of parallel programs is also much more difficult than that of sequential ones. Firstly, there is the added complexity by the parallel implementation, as introduced above. Also, support for multi-task execution is needed by the debugger, which is not always fully given by the actual debuggers [23]. Furthermore, the overhead incurred by the debugger when executing the application, such as step-by-step execution, changes the timing of the processes of the applications, and can give different results in the presence of a race condition around a shared resource.

Others difficulties of parallel programming include deadlock, network security and network failure. Network failure or security are out of the scope covered by this thesis.

By providing a tool which hides almost completely the parallelism to the user, most of the parallel programming complexity described above is avoided by the user. On the other side, the implementation of our parallel tool has involved these complexities above.

4.1.7 Brief prospect on the future of parallelism

Performance and financial points of view

All the issues concerning parallelism presented above are targeted for performance (execution time costs) and ease of programming (human time costs too). However the real world imposes that the financial cost of the parallel machine needs to be taken into consideration too. Actually, the ratio between the performance of an efficient shared-memory parallel computer and its cost is at least one order of magnitude less than that of a sequential machine. The network of stations provides an alternative of parallel processing at lower performance, but at lower price too. For applications which do not need a very fast interconnection network or a global address space, the network of stations (clusters) can be a practical alternative.

Future parallel models

As shown above, generally the parallelism allows the increase of the performance of some programs in exchange of human time and material costs. A lot of research is done to increase the profitability of the parallelism. Skillicorn and Talia [76] think that the failure of parallel computation to become mainstream is its lack of a long-term growth path. By improving classical parallel methods, they hope that within a few years parallel models will become easy to program and will be portable, even if “it will take longer for software development methods to come into general use”.

Flynn [36] takes a longer-term view. Looking back to the beginning of parallelism, he recently noticed that the difficulty of parallelism was significantly underestimated. Contrary to the classical point of view above, he thinks that a *new* representation of problems, using a cellular design with element-private form, and not a global data space, is needed to exploit the real power of parallelism. The problems would be addressed by computers in which each element has its own memory. As in the past, when scientists of several domains worked together to the realisation of the first computers, he suggests that an interdisciplinary effort may be needed to take best advantage of “parallel processor’s great potential”.

4.2 Parallel computer architectures

Deciding whether an application is appropriate for execution on a parallel machine passes necessarily through the choice of an appropriate parallel architecture for the given application. This section describes the actual parallel architectures taken into account for the tool implemented in this thesis.

For us, the most important parameter of a parallel architecture is its performance for the target applications, the decision based on the cost of the parallel machine being out of the scope of this thesis. When speaking about performance, we take the nature of target applications into account, such as frequency of inter-processor communications and I/O issues, because it influences the performance.

4.2.1 Flynn’s taxonomy

In 1966, Flynn [35] made a classification of computers, based on the stream concept, which characterises the systems based on the concurrent sequences of instructions and data managed by the processor:

1. Single instruction single data (SISD) machines execute *one* instruction on *one* data at a time. The classical processor is such an example. However, modern processors are pipelined and superscalar, hence they may execute several instructions in parallel. Therefore, as Flynn has

recently noted [36], the more precise definition of SISD machines involves a processor in which *one operation is executed per state transition*. These are sequential machines, so we are not interested in them in this chapter.

2. Single instruction multiple data (SIMD) machines allow the execution of *one* instruction on *several* data at a time. For instance, this property appears in programs of weather forecasting. Examples are vector and array processors.
3. Multiple instruction single data (MISD) machines do not exist in reality in their pure form, because a data can be processed only sequentially. Thus, we are neither interested in this architecture. However, an analogy can be made with pipeline processing of modern processors, where each data (instruction) needs several stages to complete. Several instructions are processed in the same time, with a same instruction being at different stages in every processing unit at different times. The pipeline execution [43] is another form of software parallelism, as shown in section 4.3.1.
4. Multiple instruction multiple data (MIMD) machines allow the concurrent execution of *several* instructions on *different* data. We will discuss them below.

4.2.2 MIMD machines

The MIMD has been the computing paradigm of 1990s [36]. This type of machines can further be classified, as their performance depends highly of the interconnection network among processors. A common classification is to split them in two categories [60, page 153]: multicomputers, a network of computers, and multiprocessors, several processors bound together in a same machine.

Almost all programs obey, more or less, to the locality principle, both temporal and spatial, and both for instructions and for data. Also, the locality can be valid only on some parts of a program, not necessary all the program. This principle is virtually always exploited by the use of caches, a high-speed memory, bound to processors. They improve a lot the performance of the machine. We will present them later.

Multicomputers

A multicomputer (for example a cluster of workstations) is a distributed computer system: memory, processors and I/O are all distributed. Every processor has local memory and cache, and I/O accessible by itself only. However, a library could allow a shared-memory view of the whole memory

and disks transparently to the programmer, as described in section dedicated to parallel software (section 4.3).

For applications which need a lot of communication among processors (synchronisation or data) the type of the network is important. Very important parameters of network are [5, pages 381–382]:

1. Topology: star, token ring, bus-based etc.
2. Latency of the network, defined as the time needed for a bit to arrive from one processor to another. This influences directly the synchronisation delay and the data exchange speed among the processors.
3. Bandwidth of the network, defined as the maximum data size transferred through the network in the unit of time. This parameter becomes important when big data are exchanged among processors.

Compared to multiprocessors, the low cost of multicomputers and the fact that clusters can include machines already in use have influenced their large deployment.

Multiprocessors

A multiprocessor is a natively shared-resource computer system: memory and I/O are shared among all the processors. It provides natively the shared-memory parallel programming paradigm (as described in section 4.3) in exchange of a more complex hardware implementation of the machine [5, page 29].

Almost all shared-memory machines have caches. They are private to every processor. Unlike multicomputers, as they cache data from a global memory, the same data in memory can be found on several caches and a coherence protocol is often offered. Caches and their influence on the performance are described in section 4.2.3.

Multiprocessors are mainly characterised by a high speed interconnection network between processors and memory, much faster than a classical network of multicomputers. As in the case of multicomputers, the most important parameters are its latency and its bandwidth, presented above, and its topology. Several topologies exist for MIMD machines [5, chapter 8]:

- Classical bus-based networks consist of a single bus which connects all the processors to all the memory modules (figure 4.4 on the facing page). They are also called Symmetric MultiProcessors (SMPs). They are simple to build and allows a simple cache-coherence system, but are not scalable.
- Fully connected nodes, where connects directly every module. It is the fastest and the most complex topology type, and it is theoretically

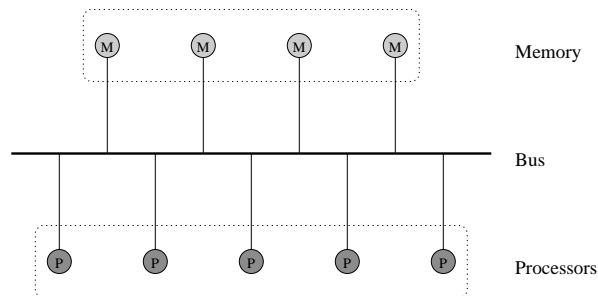


Figure 4.4: Architecture of an SMP machine, where processors are connected to memory banks through a common bus.

completely scalable. However, it is seldom used because of its hardware implementation complexity.

- Several other topologies exist between the centralised approach of bus-based systems and the fully connection of grid approach. Examples of such topologies are butterfly, fat-trees and hypercube.

The topology is important for parallel programs because of their different number of processors supported and their performance. In chapters 7 and 8, performance measurements of two machines used are given (an SMP bus-based system and a DSM system, presented below, with a hypercube topology).

Distributed shared-memory computers

The performance of shared-memory machines and the scalability of the clusters are claimed to be the successful goal of a recent type of parallel machines, the distributed shared-memory machines (DSM) [44, 67, 34]. Such a machine contains several cluster memories, which are shared by hardware, and a scalable interconnection network [44]. Origin 2000 is an example of such a machine (figure 4.5 on the next page), which is formed by several nodes, each node containing mainly one or more processors and their local memory, every node being interconnected with other nodes by a fast router. A common organisation of the nodes is the hypercube structure.

The distributed shared-memory machines have physical distributed memory, because every processor has its own local memory. But, unlike other machines (such as Cray-T3E) for the programmer all the memory is shared, as the hardware does *transparently* the routing of data. This is also the reason why we will put them in the category of shared memory-machines for the rest of this section.

However, in exchange to the physical distribution of the memory, these machines have lost one useful property of SMP architectures: the uniform

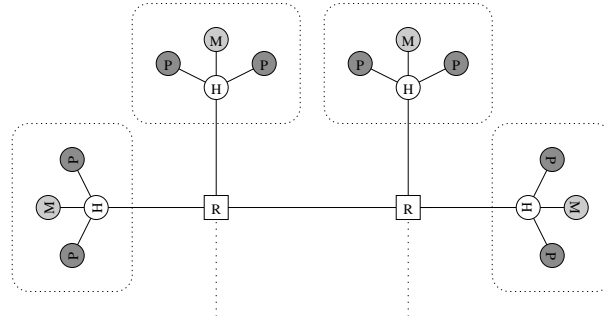


Figure 4.5: Origin 2000, a DSM machine with a hypercube organisation, contains nodes with one memory, two processors, and a connecting hub.

access to the memory. This NUMA (Non Uniform Memory Access) property comes from the fact that the access to data depends on its location. As a possible consequence, the access time may not be the same, for example a local memory may be much faster than a remote memory (local to another processor). Therefore, two factors need to be taken into account:

- On DSM machines with a large number of nodes, a cache miss is quite expensive if the data is found in the memory of a far node. Also, on cache-coherence NUMA (CC-NUMA) machines, special attention needs to be paid for cache conflicts, because a cache line invalidation can involve the whole system.
- In order to bring nearer data to the processors which use it, some DSM systems support data migration transparently by hardware. If it is used, frequent migrations can potentially add an expensive overhead.

Origin 2000 is a CC-NUMA DSM machine, supporting transparently data migration also [34].

4.2.3 Impacts of cache memory on performance

Nowadays, in terms of speed, the processor development is quite ahead of the memory. In order to reduce this divergence, the common technique is the use of caches, invented at IBM in the 1960s [72, page 182]. The cache is a small and fast memory interposed between the processor and the memory. It is managed by the hardware and acts transparently to the programmer. It improves significantly the performance of an application executed in sequential and, except in special cases, in parallel. Additionally, a programming development accordingly to the structure of the caches can improve significantly the execution time of most applications. The influence of the cache on the execution time of an application is particularly true for shared-memory parallel machines, where an added complexity is that several caches can contain the same data, as presented below.

<i>Storage</i>	<i>Size</i>	<i>Latency</i>	<i>Notes</i>
register	32 bytes	3 ns	register renaming file
L1 cache	32 KB	6 ns	on-chip, half Pentium-II clock rate
L2 cache	256 KB	57 ns	off-chip, on-package
memory	64000 KB	162 ns	100 MHz SDRAM, single bank

Table 4.1: Storage hierarchy sizes and latencies on 350MHz Deschutes Pentium II system (1998) [71].

The reason of using slow memory instead of fast caches is the cost of the hardware complexity of caches. For the same hardware complexity, as a general principle, faster the memory is, smaller it is (table 4.1 gives precise information for a Pentium II processor). To further optimise the role of caches, several caches exist in modern computers. Typically, there are two levels of cache: a small and very fast cache (level 1 cache), generally in the same chip as the processor, and a larger and less fast cache (level 2 cache) between the L1 (level 1) cache and the memory.

Actual cache architecture

Almost all programs obey to two principles of locality [48, page 38], [74, page 17]:

1. *Temporal locality*: Programs often access a same data several times in a relatively short time interval.
2. *Spatial locality*: Programs often tend to access data near a data already accessed.

The goal of actual caches is to exploit these localities. They do this by caching lines of memory data (see figure 4.6 on the next page). Caching data exploits temporal locality, while caching *lines* of data exploits spatial locality.

In addition to the two localities presented above, Hill et al. [49] consider another locality, specific to shared-memory parallel computing: *processor locality*. A program has good processor locality if contemporaneous accesses to a memory data come from a single processor (rather than many different ones). Its influences and guidelines for a better use are shown below.

For our purposes, the cache organisation has two important properties which influence the performance on parallel machines:

1. As the cache is organised by lines, a same cache line can contain several distinct data, such as variables in programming languages.
2. As the memory is global for shared-memory parallel architectures, the *same* memory line can be found on several caches.

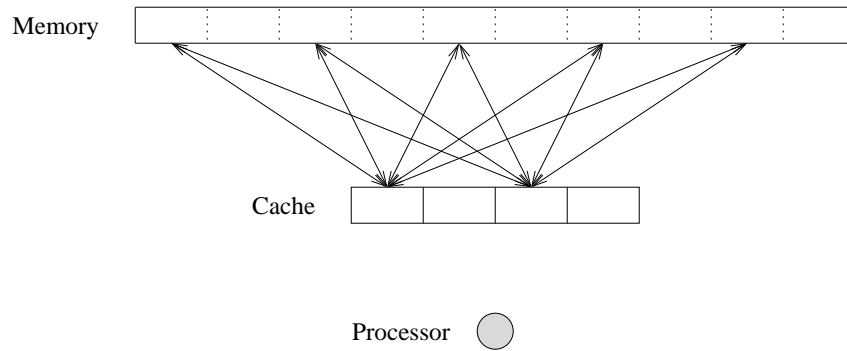


Figure 4.6: Caches store copies of data lines from memory.

A lot of research on cache architecture and on automatic compiler techniques has been made to avoid actual drawbacks. For example, such an architecture, presented in [70], splits the cache in several subcaches, based on the fact that several specialised subcaches better exploit the spatial and temporal localities. The high use of the processor locality remains however the programmer's task. Other works try to take benefit on the cooperation between the cache and the computer architecture [63]. It is worthwhile to notice that if such methods are adopted by hardware manufacturers, several optimisations presented below will no longer be needed.

4.2.4 Software development guidelines for cache access optimisation

In order to use efficiently the cache, several guidelines have to be taken into account by the programmer. Hill et al. [49] discuss some cache models (no caches, infinite word caches, infinite block caches and finite block caches) and analyses their properties in sequential and parallel programs. In the following we divide the optimisations in two categories, sequential and parallel, and we present both of them, as parallel programs involve the both.

Sequential optimisations

The memory access being expensive, the cache misses (data not found in cache) have to be avoided as much as possible. A method to do this is to access the memory continuously, because the caches are finite and useful data can be flushed from the cache. This exploits the lines organisation of the caches, because all the data found in the same cache line can be accessed with the price of only one cache miss. A typical programming example is the access of all the elements of a matrix. In `C` language, for example, matrices are stored by lines in memory. Therefore, in order to better exploit the cache,

matrices have to be accessed by lines.

During the processing of a problem, a data can be used several times. Another optimisation resulting from the finite size of the caches consists of trying to process data already in cache until its processing is finished. Thus, when the data will be flushed from the cache, it will no longer be necessary [49].

Parallel optimisations

The cache is yet more important for cache-coherent shared-memory architectures. In these systems the coherence of caches is assured by a continuous update of caches, and several cache coherence hardware protocols exist [77]. Therefore, if several processors use write operation on a same variable (a same memory location), this variable is found in several caches, and each time a processor changes it, all the caches are updated. This can even lead to speed-ups inferior to 1.

Another important aspect, less visible, is the *false sharing* [48, pages 669–670], [49]. The cause is the line organisation of caches. It appears when several processors modify distinct variables, which happen to belong to the *same* cache line. The cache of each processor contains this cache line, and each time a processor changes the value of its variable, the cache line in the other processor is updated. If these operations are frequent, the loss in performance can be very high. Methods to cope with this unwanted problem are the padding (adding unused memory space near a variable to ensure that another variable cannot be in the same line) and the data alignment (using data starting to a multiple of cache lines, which ensures that two data are not found in a same cache line) [49].

4.3 Parallel programming strategies

Our main goals concerning the parallelisation of our model implementation are to be portable and generic, i.e. to have a high power of expressiveness. We will analyse in this section the options we have had and the reasons for our choices.

4.3.1 Sources of parallelism

Germain-Renaud and Sansonnet [43, chapter 1] present three ways to exploit parallelism in applications:

1. Data parallelism, where the data of the application is divided in pieces, each piece being processed by one of the processors. This is the most used source of parallelism.

2. Pipeline parallelism, where the processing of the same data of the application can be decomposed in several consecutive sub-processings, each sub-processing being assigned to a different processor.
3. Control (functional) parallelism, where some parts of processing of the application can be done in parallel, each processor executing a part of processing.

The pipeline parallelism is suitable to specific applications, and we have not used it in our implementation of the simulation model. Instead, we have used the other two sources of parallelism. For instance, in our vision algorithm (section 7) the simulated environment is decomposed in several domains and each processor works on a different domain (different data). Also, each agent has a proper behaviour, and the behaviour of agents is executed in parallel by all the processors.

4.3.2 Abstraction levels of parallel programming

Skillicorn [76] considers six models for a parallel programming tool, based on their parallelism abstraction:

1. Completely abstract, where the parallelism use is not even known by the programmer.
2. Explicit parallelism only, where the programmer knows only that the program is executed in parallel.
3. Explicit parallelism and decomposition, where the programmer has to decompose himself the data.
4. Explicit parallelism, decomposition and mapping, where the programmer has to map himself each piece of data to processors.
5. Explicit parallelism, decomposition, mapping and communication, where the programmer takes into account also the communication among processors.
6. Everything explicit, where the synchronisation is also explicit. This is a difficult model of programming.

The final practical result of this thesis is a parallel programming tool of level 2. In fact, the user has only to specify the number of threads to use, and a few inherent constraints appear for his programs. Instead, in order to implement efficiently the tool, internally we have used low-level parallel programming of level 6.

4.3.3 Parallel programming paradigms

Writing parallel software is a difficult task, and several models of parallelism have been created. Kale [55] presents a state of the art of models and languages used in computational science and engineering. Based on the paradigm of communication among processors, we distinguish several parallel software paradigms:

- Message-passing paradigm, where the communication among processors is done through explicit messages. This is the natural method of parallelisation for multicomputers. Two standards have emerged: PVM [42] and MPI [65].
- Shared-memory paradigm, where all the processors can access the whole memory, and the communication is done through the memory. This is the natural method of parallelisation for multiprocessors.
- Mixed, combining both message-passing and shared-memory paradigms. An example is MPI-OpenMP, which may be efficient in execution time, but more complex (2 paradigms to deal with) [22].
- Natively parallel paradigms, such as actors [2], Linda model [66], CSP (Occam model [19]) and ParCeL model [84, 82, 12], which use message-passing or shared-memory paradigms.

In the message-passing paradigm each processor has a private memory, which cannot be accessed by other processors. Therefore, when processors need data from other processors, they have to communicate. The communication is done through explicit messages. This paradigm is appropriate for multicomputers, but it can also be simulated on multiprocessors (through memory copying for example). The advantage of this paradigm is that the programmer has greater control over the communication, since the communication is explicit. Therefore, he can optimise the communication for the program. However, the programmer is forced to program the communication, which increases the difficulty of writing parallel programs.

Based on the difficulty of message-passing programming used in our previous experimentations of situated multi-agent systems [57], we have considered the shared-memory paradigm easier for such systems, therefore we have focused on this paradigm for our implementation of the model.

Moreover, we have used a simple programming style, based on SPMD (Single Program Multiple Data) parallel programming strategy and BSP (Bulk Synchronous Parallelism) computation model. In BSP [80] all the processors have pure computation parts and communication parts, separated by barriers. In SPMD strategy [5, pages 609–610] all the processors run the same program but can execute different subroutines between synchronisation points (as a function of some local data for example). These computation model and strategy mix very well.

4.4 Shared-memory parallel programming paradigms

In this paradigm the memory is shared among all the processors. The communication among processors can be done through shared-memory techniques, explained below. However, if not carefully used (an example is the false sharing), the performance can decrease.

4.4.1 List of existing paradigms

The most used methods of parallel programming are based on sequential models. Based on the parallelism transparency, they can be divided in the following categories:

- Explicit multi-process. An example is the parallelisation by processes, as process creation in Unix with the `fork` system call.
- Explicit multi-threading, where threads operations are created explicitly [64]. Several multi-threading interfaces exist. In our tool we support the standard POSIX threads interface. We also support the native Irix multi-threading library, since our experience has shown that during the thesis the POSIX threads interface was not yet efficiently implemented on the Origin 2000 machine we used.
- Implicit multi-threading, where some thread-specific techniques, such as data decomposition, can be implicitly done by the compiler based on hints given by the programmer. For regular computations this is easier to use than explicit multi-threading, but can be inappropriate for complex problems with irregular computations. The parallelisation can be done in two ways:
 1. Directive-based, where the parallelisation hints are given by compiler directives. A recent standard has emerged, `OpenMP` [25, 20]. Its first implementation was in 1998, when this thesis has already been started. As we have written in our paper [27], we did not know if it has sufficient power of expressiveness to parallelise our irregular multi-agent systems, characterised by different behaviours for agents and different potential for resources. Consequently, we preferred to use another technique of shared-memory parallelisation.
 2. Sequential languages enriched with parallel constructions. Two examples are `Cilk` [68], based on `C` language, and Java threads [50].

We can add also some automatic parallelisation techniques, where the compiler reads the sequential program and adds transparently parallelisation constructs (for example thread creation and destruction).

- Original and natively parallel programming languages working also on shared-memory machines, such as Linda language [66] and ParCeL-3 [81].

In this context, we have preferred explicit multi-threading parallelisation, as they are nowadays portable, generic and efficient.

4.4.2 Synchronisation techniques

In shared-memory programming, the synchronisation among processors can be done by several techniques [5, 37, 60]. Some of them deal with critical regions (at most one processor at a time can be executing the given region). *Locks* allow to implement easily mutual exclusion with at most one processor at a time in the same region. They can be busy-wait, where the processor executes an empty loop verifying each time if it can continue, or suspending, where the processor is awoken by a signal. *Semaphores* allow to implement more complex synchronisation protocols, for example at most a given number of processors at a time in the same region, or producer-consumer protocol.

Other techniques allow to synchronise several processors, such as *barriers*, which are used for instance to implement SPMD parallelisations. Since processors are forced to wait the completion of the barrier, without actually executing the program, the more barriers, the greater the load unbalancing among processors.

As we have used the SPMD parallel programming strategy, we have used mainly synchronisation barriers for thread synchronisation.

4.5 Parallel and deterministic generation of random numbers

In order to allow diversity within the simulations written with our tool⁶, the use of random numbers is necessary.

When using computer-generated random numbers, the primary goal is the *randomness* of the sequence of numbers. The generator is generally a function which, applied to the current random number, gives the next random number. Knuth [58] and Anderson [7] provide a sound mathematical analysis of sequential random number generators. The quality of the randomness provided by the libraries of actual operating systems is often sufficient for applications. However, there are several issues to be aware of. Firstly, as the generation is done by a mathematical function, the numbers generated are *pseudo-random* numbers⁷. Secondly, as the next number is function

⁶The parameters involved in this diversity are given in section 9.2.3.

⁷However, mathematical functions may be avoided. For instance, the GNU/Linux operating system provides a random number generator based on the noise of physical devices: “The random number generator gathers environmental noise from device drivers and other sources into an entropy pool” (from `random` man page).

of the current one, and the number of integers is finite in a computer, the sequence generated is *periodic*. This can be an important issue if a lot of random numbers are needed.

Thirdly, the sequence of numbers is generated in a deterministic fashion. For a given generator, if the first number (called *seed*) used by a generator is identical, then the same sequence will be obtained. Thus, the choice of a unique seed between different executions of a program ensures the *reproducibility* of random numbers. The reproducibility can be useful for several reasons, such as validation and debugging of the programs, and comparison between the results of two executions.

For parallel applications, the generation of random numbers has several issues:

- It is *more difficult*, as the randomness of the sequence has to be preserved when concurrent access to the generator appears [37].
- The *scalability* (execution time efficiency) of the random number generation can be an issue, for example when several processes use a single generator which uses lock operations.
- If several identical generators are used, *correlation* problems can also appear, leading to the same sequence of random numbers on different generators [37].

Anderson [7] and Foster [37, pages 329–335] present some background on deterministic parallel random numbers. They present three approaches for general parallel random numbers generators:

1. The *centralised* approach uses one generator for all the tasks. Concurrent access and scalability must be dealt with. The reproducibility is hard to achieve, because the sub-sequence of random numbers given to each task depends on the request time.
2. The *replicated* approach gives a separate generator to each task. All the generators start with reproducible seeds, identical or not (given by the task identifier for example). A bad choice of the seed can lead to serious correlation problems.
3. The *distributed* approach uses several generators, as the previous approach. However, in this approach, any sequence of random numbers is given by several generators, which avoids the correlations problems from the previous approach.

As seen above, the independence of the number of processors is provided by binding the generators to tasks, not to processors. We will see in section 9.2.3 that our tool guarantees reproducibility by using simulation parameters, such as current square and cycle number, as “tasks”.

4.6 Parallel disk input/output in multi-agent systems

The analysis of a simulation can be done by using files written during the simulation. As these files need to contain sufficient information for analysis, they can have big sizes. As input/output (I/O) operations are slower than memory accesses, and because they can have high sizes, they can be a bottleneck in the simulation. Thus, their parallelisation becomes useful.

Choudhary [21] does a classification of the I/O used by different types of applications:

“Different types of applications present a variety of workloads to an I/O system. For example, transaction processing system can be characterised as performing a large number of I/O accesses of small sizes. Scientific applications require fewer accesses, each of larger size. Image processing applications require even fewer accesses, each of very large size. Finally, multimedia applications may involve accesses to large image and video data requiring tremendous bandwidth and storage capacity. The real-time requirements of multimedia systems add another dimension to the I/O system design problem because system’s response time must be deterministic.”

Our tool, depending on the information needed and on the parameters involved in the simulation, can write either one little-size file, or several high-size files. In this last case, their parallelisation can be very interesting.

The parallelisation of I/O is possible and provides good results on the parallel computer we have used [29].

Chapter 5

Parallel implementations of multi-agent systems

There are many SMASs used in literature [32]. Nevertheless, as the term “agent” is used in a very broad sense, we present here only some SMASs which are similar to our tool. We divide them in two categories: sequential simulators and parallel simulators.

5.1 Sequential multi-agent system simulators

5.1.1 Pengi, by Agre and Chapman

Pengi [4, 3] is an example of SMAS. It plays a video game known as *Pengo*.

The environment is a 2D rectangle of squares. Certain squares contain ice blocks, such a block having one square.

There is only one controllable agent in the simulation: a penguin. In the video game, it is navigated by the player through a joystick. In *Pengi*, the programmer can code himself its behaviour. The other agents, not controllable by the player, are the bees. The bees are not intelligent, but they generally tend to get closer to the penguin. From time to time, they suddenly change direction. The goal of the bees is to kill the penguin, the game being finished when this happens. The penguin is killed if a bee is close enough to the penguin, because it is stung. The goal of the penguin is to collect all the “magic blocks”. A bee and the penguin are killed when one ice block slides into it.

The agents (penguin or bee) can move only one square at a time. If an ice block is in front of an agent, either penguin or bee, and the agent moves in that direction, the block slides in that direction up to another ice block. It kills everything it meets.

The only percept of the agent is the vision. The only actions the penguin and the bees can do is the movement, potentially pushing an ice block.

Conflicts are taken into account in behaviour level. If the penguin sees a bee, it has to run away. If it simultaneously sees an ice block, it will go to it. The authors argue that several *levels* of arbitration can be used in this case, rather than planning.

The **Pengi** simulator is however a simple one. The goal has not been to write a MAS simulator, but to have a simulation support for behaviours other than planners. It is not flexible, as it is simply a program. It uses only two types of agents: penguin and bees. It is sequential. Only one percept, the vision, is implemented.

Compared to **Pengi**, the tool presented in this thesis does not allow movement of objects, such as ice cubes. Instead, it offers much more functionality, for example it allows several agents with personal behaviours. Several types of resources can exist in the environment, and the system can take care of propagating their potential, which can be perceived by the agents. It is also more flexible, being a library.

5.1.2 Improved Pengi, by Drogoul, Ferber and Jacopin

Drogoul et al. [30] wrote another version of **Pengi**, with the aim of showing that the simplicity of the model does not prevent the simulation game from being correct. They pointed out also the emergence of more complex behaviours.

However, compared to the original version of **Pengi**, several modifications were made. In original **Pengi**, the behaviour of the penguin was given by routines thought by the programmer. In the new version, the authors want to put intelligence and learning into the agent level. Their agents are based on Gul Agha's actor model [2]. Each agent follows the principle of locality and autonomy. They have simple behaviours consisting of only satisfaction and flight. The satisfaction of the penguin is to reach and eat the nearest diamond. The flight behaviours is generated when a bee is seen. It can choose to slip a cube, if one exists, or to get away as far as possible.

The percepts of the penguin are two: vision, as shown above, and odour. The odour is used to find the diamonds. The authors noted that for the odour the local perception is sufficient. The adjacent locations of the penguin are sufficient to find the way to the source. As the diamonds are fixed and they do not change their potential, the potential field can be rarely calculated, at the beginning of the simulation and when a diamond is collected (in this case it disappears). The propagation of the field is done by waves. Every square has a value equal to the minimal number of moves needed by an agent on this square to reach the diamond. Compared to the simple solution of having a value equal to the euclidian distance between the two points, this propagation allows to avoid obstacles.

In the future, the authors want to integrate the notion of learning, based on memorising past situation and on the use of genetic algorithms.

Compared to this simulator, our tool provides much more functionalities, such as several agents and flexibility of their behaviour.

5.2 Parallel multi-agent system simulators

An important feature of the simulators presented in this category is that they run efficiently on parallel machines. We present a simulator taking into account uncertainties and errors, and another one dedicated to massive simulations.

5.2.1 PIOMAS error-accounting simulator

The specificity of the PIOMAS simulation program designed by Bouzid [14, 16, 15] is that it takes into account uncertainties and errors in both effectors and sensors of agents. It is based on a Markovian model taken from Partially Observable Markov Decision Processes (POMDPs).

The environment of the simulation is discrete and contains inert objects (e.g. walls) and active objects (e.g. doors, robots). The evolution of the system is ruled by two types of constraints: laws to maintain the coherence of the system (e.g. a robot must never go through a wall) and laws to handle simultaneous influences.

As said above, the uncertainties and errors are taken into account at two levels:

1. Perception: for each agent, type and observation, a probability distribution of the set of observations that might be confused with the good one is calculated. A confusion matrix is thus obtained and used to express error occurrences.
2. Action: for each agent and action, a set of transitions and their occurrence probabilities are calculated. This transition matrix gives the next state of the agent.

The simulation model is divided in cycles of constant step. During each cycle, the simulator executes the following sub-cycles:

1. Perception/decision/action phase of each active object, where every agent plans its action.
2. Detection and solving by the simulator of all the conflicts.
3. Execution of the actions of all active objects.
4. Update of the environment observations and transition probability distributions.

Because the model is stochastic, the evolution of the system is not deterministic.

A simulator is provided based on some *real* information. This information has been acquired from many experimentations and measurements with a real robot. In the simulation, a robot occupies a contiguous set of boxes and moves just one box at a time in front of it. The slowness of the active objects is simulated by moving it every n cycles, thus all movements being multiple of 1 cycle. It has also basic rotating movements: it can rotate 90 degrees at left or at right. The sensor and effector models used are those of a NOMAD 200 robot (for which they have a training corpus). For each action, the corresponding transition set and its probability distribution have been calculated from the movement noticed on the real robot they have used.

From the parallelism point of view, the simulator is implemented in \mathbf{C} on a DSM machine (see section 4.2.2 for more information), Origin 2000, with up to 64 processors and 24GB of physical memory. The various behaviours of the agents and their dynamic position in the environment give an irregular nature to the simulator. The solution to cope with this irregularity has been the use of a parallel ultra-light cellular library, `ParCeL-3`¹ [82] (see section 2.2), built on a multi-threading library. One of the difficulties of the simulation, namely the simultaneity of actions and the solving of the spatial conflicts, is carefully taken into account by a two work-pool [60] model: the first work-pool contains all the tasks (any task represents an active object) in pre-planning state, while the second contains only super-tasks (a super-task contains a task along with all the tasks conflicting with this one, if they exist) which are planned but not already carried out. The work-pool model, by its dynamic load balancing, is also adapted to such irregular problems.

The parallel performance is satisfactory, with a speed-up greater than 2 on 5 processors.

This simulator has other goals as the ours. It is appropriate for fine simulation of uncertainties and errors in both effectors and sensors of agents, living in a population of *few* agents.

5.2.2 BioLand massively simulator

BioLand [86] is another parallel MAS simulator. It resembles our tool in several points. Its main features include *massively* parallelism (tens of thousands of agents) and a lot of provided functionalities, like gradient algorithm, ANN-based agents behaviours and use of genetic information. It is mainly useful in DAI (Distributed Artificial Intelligence).

Several distinct populations (“species”) of agents (called “biots”) evolve in a toroidal 2D environment. The environment is made by square cells. There is no obstacle in the world preventing the agents to enter.

¹This library has been developed at our laboratory (Supélec).

Any biot behaviour is entirely controlled by an ANN specified by its individual genome. Each ANN has input neurons connected to the sensors of the biot, and a few output neurons giving the action to be taken. The biots have two types of perceptions: smell and sound. These perceptions are propagated by any biot and by any *action* of the biots. The actions of biots are mainly moving a certain speed, eating, reproducing and producing sounds.

Any biot has a simulated metabolism, given by a number. The metabolism increases when the biot eats, and decreases generally over the time (e.g. when moving or emitting a sound). When its metabolism drops to zero, it dies.

Any biot has a gender. If two biots of the same type but of different gender meet, and their metabolism is higher than a certain value, they engender a biot. The genome of the new biot is created from recombination and mutation of parental genes.

The processing of several parameters involved in the biots (concerning metabolism and reproduction for example) is automatically taken into account and updated by the simulator. It is also responsible of the propagation of the gradients, which are perceived by the agents. Each object in the world and each action taken by an object produces a gradient. Although several types of gradient exist, it seems that only one type of propagation is used in the simulator. More specifically, the strength of a gradient in any square is inversely proportional to the square of the distance between itself and the source. When a square is influenced by several sources, the potential *adds*. The propagation of the potential is done entirely, i.e. the gradient is *exact* (see section 8.9 for a comparison exact/inexact gradient algorithms).

The simulator has been implemented in the C* language on a 16k-processor CM-2 machine (a massively parallel SIMD machine). It can be used for experiments with about 32000 entities in an environment of about 1000×1000 cells (1 biot occupies 1 cell). The total memory size of CM-2 was 128MB. Parallel performance is not shown, but it seems that the SIMD architecture is appropriate for such simulations and provides very good results.

BioLand authors present also some conclusions after such a simulation, whose main characteristics are presented in the following. It involved three species of biots: prairie dogs, hawks and snakes, and three non-animal species: plants, trees and holes. Prairie dogs eat plants, and hawks and snakes eat prairie dogs. Each type of object has a specific smell and sound. The biots can reproduce and can die. Plants are generated by the environment from time to time. The simulations done in such hypotheses showed an evolution of agents behaviours. They were capable to learn (during lifetime) and evaluate (over generations).

BioLand thus offers a very rich model of simulation of SMAS, richer than ours. However, the vision and the spatial conflicts among agents, provided in our tool, are not supported, its algorithmic is simpler (no obstacle in the world) and it runs on SIMD machines, not on classical modern MIMD ones.

5.3 Development environments of parallel MASs

From the practical point of view, the result of this thesis is a tool allowing the implementation of certain MASs, as shown in section 3.2. The two primary objectives of the tool are ease of programming and good performance. Also, the tool has to implement and to offer some parallel algorithms (some of them are presented in [26]).

When implementing a tool, one of the choices to be done is the programming development offered to the user, such as a programming language together with a compiler/interpreter, or a library. In this section, the term *user* refers to the user of our tool.

A first approach is to provide an application with multiple options or parameters, for example one of its parameters specifies the number of threads to use, another the dimensions of the environment. This approach is interesting in that it is easier to write, because it is specific and not flexible, and simple to use, because only the options or parameters have to be known. Nonetheless, this solution is generally inflexible (all its functionalities are fixed) and does not allow parametrisation by programming code². It is useful when an already known set of functionalities is needed.

The other approach is to allow and force the user to write code, as he does when he programs. This makes harder its use. However, it gives all the flexibility of a programming language. As such, it is adapted for tools whose set of functionalities is not known in advance. Three solutions can be offered to the user:

1. A domain-specific programming language and its compiler/interpreter eases the user programming by providing syntax specific to the tasks and functionalities necessary to implement them. It can also be optimised for such tasks. The compiler can generate machine code or another language, such as C, which will then be compiled with its appropriate compiler. The major drawback of this approach is that the user needs to learn this new language. This leads to the increase of learning and development time. Also, in the case when the compiler generates machine code, its output is machine-dependent, thus the programmer of the tool needs to write a compiler which supports the needed parallel architectures. Kale [55] does a survey of sequential and parallel programming languages, both general and domain-specific, used for computational science and engineering. Bagrodia [9] surveys parallel languages for discrete-event simulations. Ferber [32] surveys MAS-specific languages.
2. On the other extreme there is the library approach, which provides functionalities by means of a set of functions, commonly called API (Ap-

²Special mechanisms, such as plug-ins, can be used to cope with these difficulties, but supporting them starts to resemble to the next approach.

plication Programming Interface). It is possible to use the library simply by writing in the same programming language as the library. However, other languages can be supported too, sometimes in exchange of language transformation time loss and greater programming complexity. For instance, the CORBA [41] software architecture allows the execution of a program written in several languages. A library cannot provide, like a new programming language, a domain-specific syntax. If it does not provide a binding to other languages, the user can also be limited by the syntax of the corresponding language. On the other hand, a library does not create a new language that the user needs to learn. Its level of portability is much greater than the previous solution. Examples of parallel libraries are MPI [65], PVM [42] and POSIX threads.

3. A language extension is an intermediate approach which combines the two approaches presented above. Generally, several keywords are added to an existent language. Kale [55] presents some extension languages used in computational science and engineering.

Additionally, a graphic interface can be offered, as do `Visual Basic` and `Delphi`. Our tool does not provide such an interface for simulator programming, however such an interface is possible.

The programming language chosen for our tool is `C`. It provides performance, portability and flexibility.

5.4 Conclusions

There are few implementations allowing to simulate societies of situated MASs. We have presented two simple sequential simulators and two complex parallel simulators. The first parallel simulator is appropriate for simulations of a few number of agents, while the second one works on SIMD machines and is not as flexible as ours.

We have not found a general, simple and parallel simulator on MIMD machines. Our model and its implementation aim to fill this gap.

Chapter 6

The ParSSAP model of multi-agent system simulation

This chapter presents the model we have conceived for the simulation of situated multi-agent systems. It does *not* provide a model of multi-agent systems, nor even agents, which is the job of the user. Instead, it provides a simulation model which helps him to implement his multi-agent model. It aims to hide the algorithmic issues of simulation and to give to the user a programming view where agents simply evolve concurrently. For example, the user does not need to deal with linked lists of agents or with the dynamical state of the system. This is facilitated also by implementing it as a library, which allows user to use only functionalities he needs and to customise it for its model.

6.1 Components of the model

Four components appear in the model (figure 6.1 on the following page):

1. *Environment*: the world where the entities evolve. It is discretised in space. It is toroidal (left and right frontiers, and top and bottom frontiers are glued) or edged. Each square of the environment has a type: either free square, or obstacle, or resource, and a state: either unoccupied, or occupied by an agent. An obstacle prevents the agents to enter that square.
2. *Resources*: entities of one square size guiding agents. They can do so by spreading potential fields which are perceived by agents. The potential field decreases with the distance to resource, even in the presence of obstacles, thus allowing agents to find the way to resources. Each resource contains objects, which can be carried by agents. The potential field size can change during the simulation, according to its

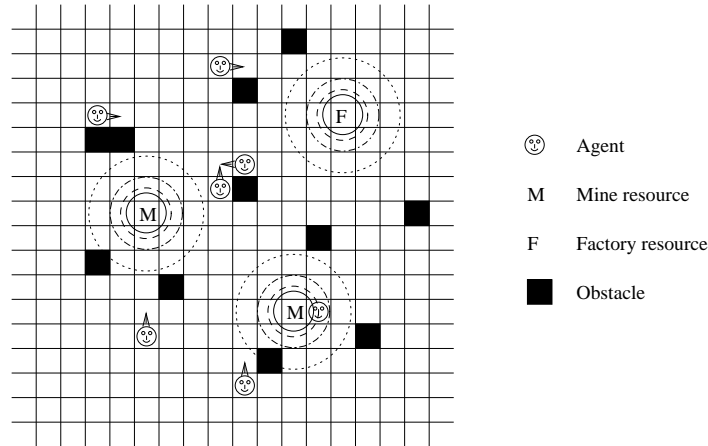


Figure 6.1: Example of situated multi-agent system provided by our implementation.

load in objects. Moreover, several types of potential can be used, in order to introduce several kinds of resources/objects.

3. *Agents*: mobile entities having a memory and a behaviour. The behaviours of agents generate actions, based on their perceptions and their memories. The perceptions allow agents to find information about the environment. The actions allow agents to change the state of the system. More information about agents' perceptions and actions is given in the next section.

The memory of each agent allows the agent to memorise facts during the simulation. Its size can dynamically grow or decrease, according to agent's needs.

Agents have also a speed, giving the frequency with which their behaviour is activated, and a priority. As agents act simultaneously and they do not know the other agents' action, several agents may try to enter the same square in the same time. The spatial conflicts use the priority of agents and are solved by the arbitrator, as described below.

4. *Arbitrator*: virtual entity maintaining the coherence of the system. During the simulation it verifies that the simulation laws are fulfilled. Such laws are avoiding agents to enter an obstacle or to move to a distance longer than possible, and avoiding spatial conflicts. Another example of simulation law is a given upper limit on the number of agents in the system.

A spatial conflict appears when two agents want to enter the same square of environment. In this case, the arbitrator lets only one agent, the winner, to carry out its action and blocks the others. Several

strategies may be used to find the winner. For instance, in our implementation the choice of the winner is done at three levels. If there are agents to be created, the winner is randomly chosen among them. Elsewhere, if there are agents with high priority, the winner is randomly chosen among them. Elsewhere, it is chosen randomly among the agents. The agents' priority, which *influences* the choice of the winner, is specified by the user during agent creation.

6.2 Agents' behaviour

This section presents important features used in the behaviour of agents, but which are general and not necessarily bound to other agents' features.

In a pure multi-agent system, agents' perceptions and actions are always local. However, for the flexibility of the model, global functionalities are also available, but a clear distinction between local and global functionalities is done at agent level. For example, an agent may create agents not in its neighbourhood, and we will use this feature to implement Conway's game of life (section 10.1).

Agents' perceptions

The perceptions allow agents to sense the environment. Examples of high-level percepts are: mark sensing, vision and odour detection. The mark sensing allows agents to know whether a square contains a mark (put by another agent) or not. The vision percept allows agents to know if they can see a given square, for example to know if it contains an agent or a resource. A square is visible from another square if the line between them does not intersect any obstacle. The odour percept allows agents to know the potential of squares near them, helping them to find the way to resources even if they have to go around obstacles. Agent communication is a complex problem, and it has not yet been taken into consideration.

Not all the agents have the same percepts. Some agents may have no percept, while others one or several percepts.

Agents' actions

The most frequent action of agents is the move. Examples of other actions are: take an object from the environment, drop an object it possesses in the environment and push an object in the environment. They can also create and destroy agents, as described below. Some actions are mutually exclusive, for example move in north and east direction in the same cycle.

For generality, an agent may do several actions during one activation, provided that they are compatible.

Except the spatial conflict processing, the effects of actions are predetermined and deterministic, e.g. agents will move in a square if such a move was asked and is possible, and they cannot arrive accidentally in another square.

Agents' dynamical creation and destruction

Agents may be created or destroyed in any cycle of simulation. They may be created in any free square of the environment, near or far from the creator agent. Their exact position may be specified by the user or may be randomly chosen by the system, either in all the environment, or at a specified distance from the creator agent. In all cases, if their creation leads to spatial conflicts, their creation obeys to the arbitrator's decision.

When an agent is destroyed it simply disappears from the system. Agents communicate through the environment, hence finding information about agents is always done through the environment. Therefore, the agent to be destroyed is specified by its position. An agent may destroy itself too.

6.3 Simulation of the simultaneity of actions

In our model the agents act synchronously and simultaneously. They act synchronously because during each cycle of simulation (see below) all the agents are activated only once. They act simultaneously because there is no predefined execution order of agents during activation, and no agent has precedence over other agents. Also, the state of the system remains identical during the activation of any agent.

The simultaneity of actions is source of many difficulties in the model design and its implementation. It leads to conflicts, managed by this model, as shown below.

6.4 Execution engine

The simulation starts by the initialisation of the system. During the initialisation, the environment is created and resources and initial agents are created and put in the environment. Afterwards, the percept data is initialised, such as potential fields and visibility fields.

The time is discretised, and the simulation is based on cycles. In order to achieve simultaneity of actions, the execution of agents is divided in three steps: behaviour execution (action plan), spatial conflict avoiding and action execution. The simulation consists of a synchronous virtual machine which repeatedly, every cycle, executes in order the following five steps (figure 6.2 on the next page):

1. Activate each agent by executing its behaviour (according to its speed, as shown above). Each behaviour generates the planned action of the

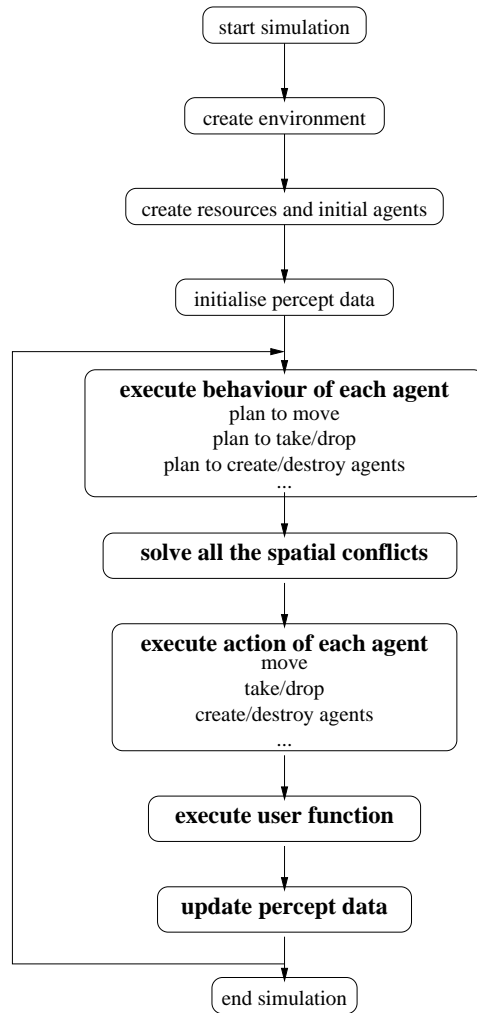


Figure 6.2: Execution engine of simulation.

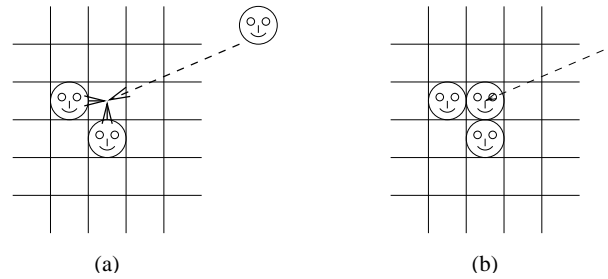


Figure 6.3: Example of conflict and its solution: (a) two agents want to move in the same square, where an agent is planned also to move, but (b) the arbitrator lets only one, at random, to execute its planned action.

agent. Additionally, it can plan to create or destroy agents. The creation/destruction of agents becomes effective in the third step, as explained below.

2. Avoid all the spatial conflicts. Conflicts appear among agents which either move in the given square, or are created in the given square (figure 6.3). For any such conflict, the arbitrator chooses a winner among all the agents involved, as explained above. Only the winner carries out its planned action. For all the other agents the arbitrator changes their action from `move` to `stay`, or refuses their creation.
3. Execute the action of each agent. During this step the actions of agents become effective (except for agents having lost a conflict).
4. Execute the user-function. This is a function given by the user of the model and thus allows to do per cycle actions or to interfere with other programs. It is the best place to specify the end of the simulation and to save information about the simulation.
5. Update the environment. For example, update the potential fields of the resources which have changed their potential.

Among these steps, the 1st and the 4th must be provided by the user, while the others may be done automatically by the model. Additionally, the user must specify the evolving law of resources by providing a function giving their potential based on their dynamic load.

6.5 Information about the state of the system and its evolution

During simulation, the system provides information about itself. Three types of information are provided:

1. Basic evolution of the system (statistic information), such as the number of agents, the number of resources not yet visited by agents and the number of objects in resources. This allows to measure the performance of agent population.
2. Detailed evolution of the system (evolution information), such as the movement of agents and the load of resources. This allows to look at agent's behaviour efficiency.
3. Complete state of the system (checkpoints), which describes the state of the whole system at a given point of simulation. A checkpoint may be used to resume a simulation as well as to initialise the system. For a better organisation, especially for system initialisation, the state of the system is divided in two parts: general and local information. The general information part consists of a file which contains information about simulation parameters, such as the dimension of the environment and links to the local information part. The local information part contains all the information about obstacles, agents and resources. So, in order to simplify its use, several files are involved in local information, one per domain of environment. In each local file, the coordinates are relative to the domain, so that these files can be assembled in several ways to form a complete environment.

6.6 The transparency of parallelism of the model

As noticed, the model does not take into account whether the execution is done in parallel or not. The model is not specific to sequential or to parallel execution. Therefore, if parallel machines are used, the parallelism will be transparent to the user.

6.7 Conclusions

This brief chapter has presented the simulation model we propose. It allows the simulation of situated multi-agent systems by offering to the user a programming view where agents simply evolve concurrently. The model has four components: environment, resources, agents and arbitrator. Agents may be dynamically created and destroyed. Agents' perceptions and actions are always local, but, for flexibility, global perceptions and actions are also provided and a clear distinction is made between them. The model takes into account the simultaneity of actions. The arbitrator solves the spatial conflicts among agents.

Agents act synchronously and simultaneously. Synchronism refers to that agents are activated only once during a cycle of simulation. Simultaneity

refers to that their activation is concurrent, and no agent has precedence over another.

The execution model is based on cycles, and each cycle has five steps. The first three steps deal with agents: plan step, conflict solve step and action step. The fourth step allows the user to take control over the simulation. The last step updates agents' percept data.

Additionally, several types of information about the system are provided.

The model is not specific to sequential or parallel features. Our implementation of this model (chapter 9) provides almost full transparency to the user.

We think that this model and its implementation are sufficiently complex to allow the simulation of a broad domain of systems, as partially shown in chapter 10.

Chapter 7

Parallel algorithmic of vision perception

The agent, by definition (section 3.2), does sense the environment by its percepts. Our tool provides built-in support for two percepts. One of them simulates the odour, and will be presented in the next chapter. The other one, presented in this chapter, simulates the vision (direct perception). Basic information has already been presented in section 3.4.3. More specifically, this chapter presents an algorithm for calculating visibility fields of the MASs simulated with our tool. When an agent needs to obtain information on a specific square, it uses firstly the visibility field of its square in order to know if the desired square is visible from it. If affirmative, it calls the function which gives it information about the occupy state of the desired square. The vision algorithm consists in finding the visibility field of each square of the environment (figure 7.1 on the next page).

7.1 Properties of the vision percept

In our simulations, the only opaque squares (which prevent the vision propagation) are the obstacles; the agents and the resources do not prevent the visibility. Moreover, the obstacles are fixed during the simulation. Thus, compared to the wave propagation algorithm (chapter 8), which need to be regenerated during the simulation, the visibility fields are fixed. It can be executed only once for all the squares of the environment. However, it uses large amounts of memory to store all the visibility fields. Methods to cope with these trade-offs are described later, in section 7.7. This algorithm is thus characterised by a once-execution and high memory size.

Another useful property of the vision percept is that it is generally symmetric: if A can see B , then B can also see A . The use of symmetry leads to the decrease of the execution time and of the memory used by a factor of 2.

Therefore, compared to the wave propagation model simulating the odour

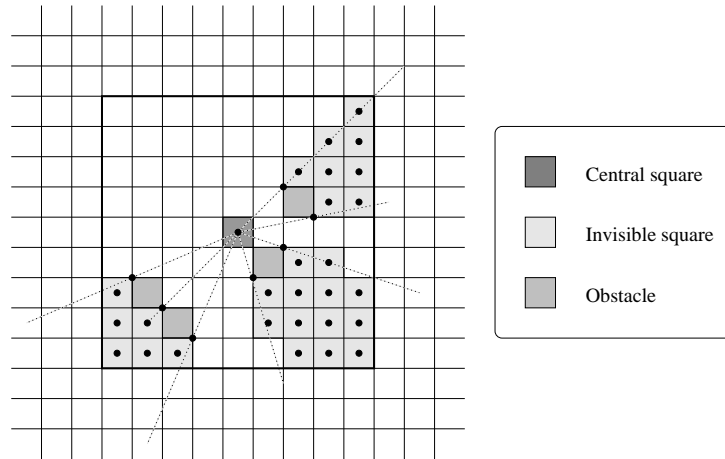


Figure 7.1: Example of visibility field of one square (with radius of 4 in an 8-connectivity environment).

percept, the vision percept has two specific properties:

1. It is independent of the distance between the squares, as soon as they are in the same visibility field.
2. It is highly influenced by obstacles.

7.2 Definition of the visibility field topology

Our first choice is to consider that the visibility field of a square is local and contains the squares in its proximity at a specified distance (called *radius*) from it. Obstacles can reduce the visibility field. Its own square is, by definition, always visible.

The second choice involves the distance used. In order to be coherent with the movements of the agents, we chose to consider as distance between two squares A and B , the minimum number of movements needed by an agent to go from A to B . Moreover, as the agent movements are function of the environment connectivity, the distance and thus the visibility field depend on the environment connectivity. In fact, the connectivity type is global to the system (see section 7.7) and acts upon the vision algorithm, the wave propagation algorithm, and on the agent movements. Figure 7.2 on the facing page gives an example of the distance in 4-connectivity and 8-connectivity cases, and compares them with the classical Euclidean distance. Our 4-connectivity case gives in fact the well-known von Neumann neighbourhood, and the 8-connectivity case the well-known Moore neighbourhood [79].

From the mathematical point of view, all these three distances are particular cases of norms. Some norms are defined in 2D environments [87,

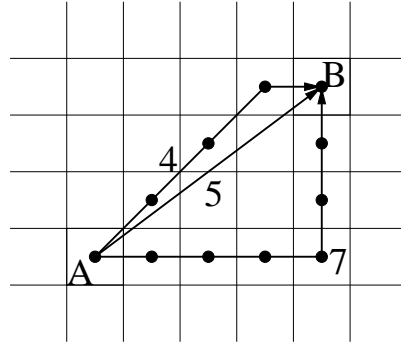


Figure 7.2: The distance between squares A and B is different: 5 for Euclidean distance, 7 in 4-connectivity case, and 4 in 8-connectivity case.

page 55] by:

$$\|v\|_n = \sqrt[n]{|dx|^n + |dy|^n}$$

In our case, $dx = x_A - x_B$, and $dy = y_A - y_B$. The three distances from figure 7.2 are obtained with:

- $n = 1$ for 4-connectivity (which corresponds to $\|v\| = |dx| + |dy|$),
- $n = 2$ for Euclidean distance, and
- $n = \infty$ (which corresponds to $\|v\| = \max(|dx|, |dy|)$) for 8-connectivity.

As shown above, the distance depends on the connectivity type. Figure 7.3 on the following page presents the maximum visibility field in both cases. By convention, the visibility field containing the squares adjacent to one square has a radius of 1. In 4-connectivity case, the shape of the field is a rhombus¹, shown in figure 7.3(a), and, if we note by r_v the vision radius, the number of squares it contains is given by:

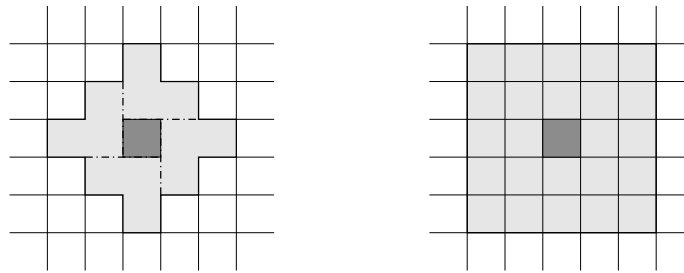
$$NS_4 = 4 \sum_{i=1}^{i=r_v} i + 1 = 2r_v^2 + 2r_v + 1 \quad (7.1)$$

In 8-connectivity case, the shape of the field is a square, shown in figure 7.3(b), and the number of squares is given by:

$$NS_8 = (2r_v + 1)^2 = 4r_v^2 + 4r_v + 1 \quad (7.2)$$

The two types of connectivity are taken into account for execution time and memory requirement optimisations of the vision algorithm, as described in section 7.7. For example, the 4-connectivity case needs twice fewer computes than the 8-connectivity case. Our tool allows the use of both connectivities, which has to be chosen at the beginning of the simulation.

¹In *continuous* environment, the rhombus becomes a square rotated by 90 degrees.



(a) The maximum visibility field for 4-connectivity is a rhombus.

(b) The maximum visibility field for 8-connectivity is a square.

Figure 7.3: Comparison of visibility fields between 4- and 8-connectivity (radius = 2).

7.3 Vision algorithm

As previously mentioned, in order to use the vision, the agents have to know if they can see a certain square or not. Two steps are involved: the initialisation of the visibility fields and their use. The latter is simply a function call giving the element value of the visibility field concerned. The former (which we call the vision algorithm) is the most complex and time consuming part, and is presented in the following.

The vision algorithm calculates the visibility field of each square of the environment (all the agents have the same vision radius). This gives a 4D matrix for all the environment (figure 7.4 on the next page). Calculating the field of one square consists of calculating what squares are visible from it. We thus use a line drawing algorithm, which will be presented later, for all the points at a distance inferior to the vision radius to the centre. More rigorously, the algorithm is the following:

```

for all square  $S_1$  of the environment do
  for all square  $S_2$  at distance inferior to radius do
    if line  $S_1S_2$  passes through an opaque square then
       $field[S_1][S_2] \leftarrow \text{false}$ 
    else
       $field[S_1][S_2] \leftarrow \text{true}$ 
    end if
  end for
end for

```

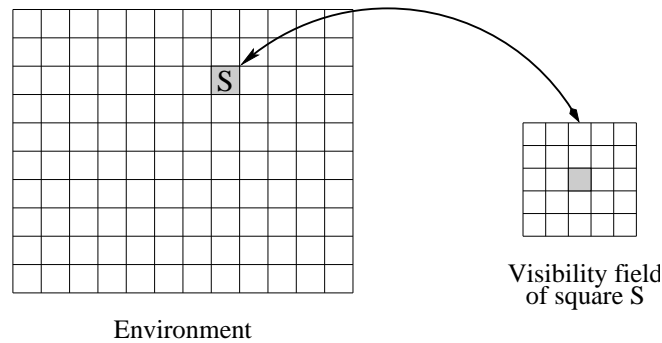


Figure 7.4: The visibility matrix of the environment is a 4D matrix.

7.4 Difficulties introduced by obstacles

In a world without obstacles, the visibility fields would not be prevented by anything. Additionally, the obstacles not only limit the visibility, but also add complexity, together with the space discretisation, to the computing of the vision fields. These difficulties are presented below.

The discretisation of the environment has led us to take two choices. Suppose we want to find out if the square B is visible from square A . The first choice is that the square A is condensed in one point: its centre. This comes from the fact that we consider any agent as being in the centre of the square². The second choice is that, from visibility point of view, the square B is also condensed in its centre. Thus, we consider a square B as visible from A if the real line between the centres of A and B does not intersect any opaque square. If either A or B is an obstacle, then they are considered invisible from each other (an obstacle cannot be seen).

Before explaining the implications of the second choice to the visibility field, we present a third choice, namely when the real line passes through the corner of an opaque square (figure 7.5 on the following page). We chose to consider that the obstacle does prevent the vision (points A are some examples). If we have chosen to not prevent the vision, an anomaly can appear in 4-connectivity case: a square is visible, but the way to it is blocked by obstacles (point B).

Finally, we need to check for obstacle all the squares the ideal line pierces. Therefore, classical lines, which have only one square per major coordinate, are not appropriate, hence we will use the so-called supercover lines.

²In fact, the dimension of an agent is less than a square, since a square can contain both an agent and a resource, for instance.

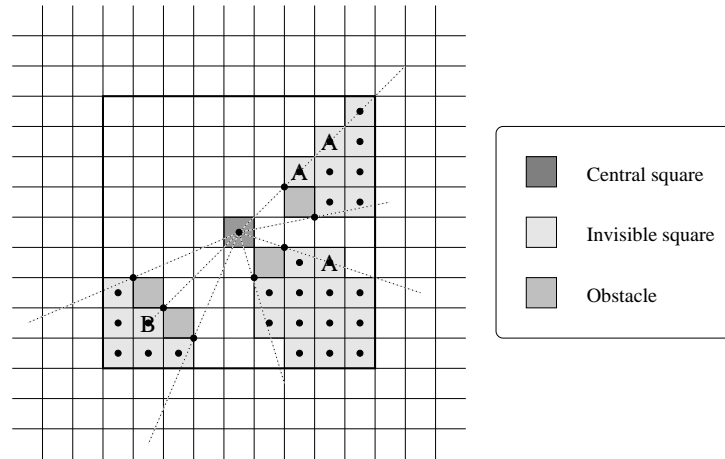


Figure 7.5: Example of visibility field of a square: if the real line intersects a corner of an opaque square, then it is considered as hidden (points *A*). If we had chosen to consider it as visible, some points would be visible, but unreachable (point *B*).

7.5 Line and supercover line tracing algorithms

In order to compute the visibility domain of a square, modified lines drawing algorithms can be used. This section deals with several algorithms which can be used in vision, algorithms found in ray-tracing. All these algorithms decompose uniformly the environment (cells of equal dimension). Algorithms which decompose non-uniformly the environment, for example taking into account the number of obstacles, can also be imagined. More information about ray-tracing algorithms can be found in [89].

7.5.1 Line tracing algorithms

Bresenham's original algorithm for drawing lines

Bresenham [17] presents an algorithm to draw lines, one point per coordinate. It has the important particularity that it uses no floating-point operations (which are time-consuming) and no multiplication or division operations. Only the principle of this algorithm is described here, for more information and rigorous demonstration, see the original paper [17].

We have to draw a line from (x_1, y_1) to (x_2, y_2) (figure 7.6 on the next page). Suppose we are in the first octant, so $\Delta x \geq 0$, $\Delta y \geq 0$, and $\Delta x \geq \Delta y$ ($0 \leq \text{slope} \leq 1$). This means we will draw one point per x-coordinate (Ox is the driving axis and Oy is the passive axis), i.e. we draw an 8-connected line. The other cases can be brought to this case by simply changing the signs of some parameters of the algorithm. Bresenham algorithm is clearly explained in [89]:

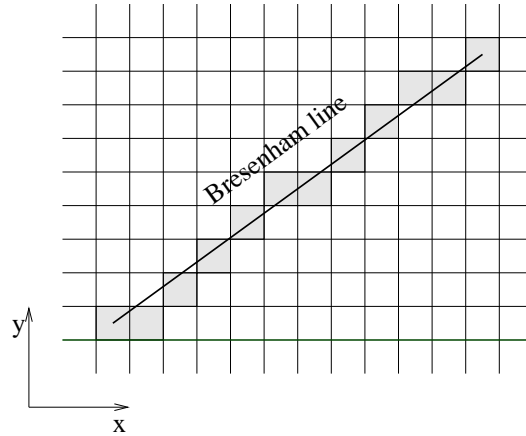


Figure 7.6: Illustration of the Bresenham line.

“During the process of generating each consecutive cell, the coordinate corresponding to the driving axis is unconditionally incremented by one unit. At the same time, a control term, the *error*, measured on the passive axis, is updated by subtracting from it the slope value and then checking whether it is still smaller than half the cell size. When this test fails, a unit increment of the passive axis is performed. The control term is corrected by adding it the value corresponding to one cell whenever underflow occurs.”

As the slope of the real line is $\Delta x/\Delta y$, the real line has to add $\Delta x/\Delta y$ every time x changes. However, when using integers, we use a variable, **error**, which accumulates the difference between the real line and the drawn line (see figure 7.7 on the following page). Every time we change x , we have to choose between the points B and C . If the next point is B , the **error** variable increments by Δy . Otherwise, the next point is C and the **error** variable increments by $\Delta y - \Delta x$:

```

1  ddy = 2 * (y2 - y1);
2  ddx = 2 * (x2 - x1);
3  for (i=0 ; i < dx ; i++){
4    x ++;
5    error += ddy;
6    if (error > ddx){ // if the point is B
7      y ++;
8      error -= ddx;
9    }
10  DRAW_POINT (y, x);
11 }

```

The algorithm implemented in our tool uses a slightly modified version of the Bresenham algorithm.

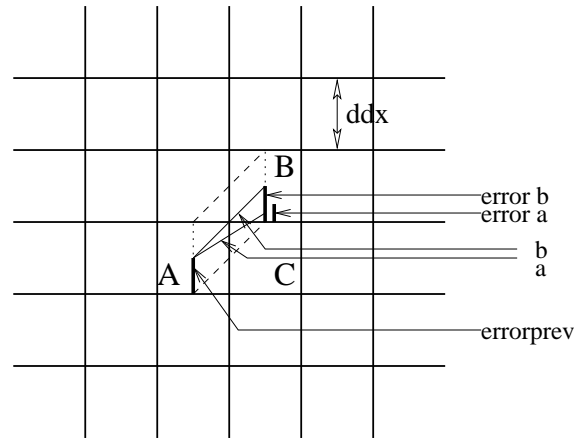


Figure 7.7: Illustration of the Bresenham algorithm.

Linear interpolation using fixed points

As previously said, the main goal of Bresenham algorithm is to use only integer operations. However, it has a serious drawback: for every point it draws, it does a decision either it does a jump or not (the `if` instruction above). The jumps are painful to actual processors, because their pipeline architecture is not efficiently used. A method to avoid the jumps is to use fixed point arithmetic for floating-point calculations. Thus, a simple interpolation on the major axis is sufficient. Fixed point representation is a simple method to store floating point numbers in integer ones³.

7.5.2 Supercover line tracing algorithms

The two previous methods allow to draw lines. The following methods (the first and the second use Bresenham original algorithm) draw lines which include *all* the cells pierced by the continuous line. These particular lines are usually called *supercover lines* [8]. Therefore they allow to calculate the visibility region.

Bouton's algorithm

Bouton [13] uses a variant of the Bresenham algorithm to calculate the visibility region. For every two points, he checks if the line drawn by Bresenham algorithm intersects an obstacle (figure 7.9 on page 127). If the coordinate of the major axis does not change, one square is drawn. If it changes, two squares are drawn. If the line intersects a corner, three squares are drawn.

³More information about fixed point representation can be found at http://www.nondot.org/sabre/graphpro/index_line.html, where line tracing implementations using fixed point representation are also explained.

It is close to our supercover algorithm. Here is the core part (suppose the first octant)⁴:

```

1  d = dy - dx;
2  di1 = 2*dy;
3  di2 = 2*dy - 2*dx;
4  for (i=0 ; i < dx ; i++)
5    if (d < 0){ // same coordinate => one square
6      d += di1;
7      x ++;
8      DRAW_POINT (x, y);
9    } else if (d != 0){ // coordinate changed => two squares
10     d += di2;
11     DRAW_POINT (x, y+1);
12     DRAW_POINT (x+1, y+1);
13     x ++;
14     y ++;
15   } else if (d > 0){ // corner => three squares
16     d += di2;
17     DRAW_POINT (x+1, y);
18     DRAW_POINT (x, y+1);
19     DRAW_POINT (x+1, y+1);
20     x ++;
21     y ++;
22   }

```

Fujimoto's generalised DDA algorithm

Maybe the first algorithm dealing with all the points a line *pierces* is presented by Fujimoto et al. [38]. It is based on the Bresenham's one. Several algorithms for decomposing a line into cells (pixels) exist under the generic name of DDA (Discrete Differential Analyser). Better algorithms derived from this one are presented in [89, 11].

Bandi [11] decomposes the processing of every point in two steps: one for testing the cell above, and one for testing the cell below. In this algorithm the line needs not terminate at cell centers. Also, the algorithm can be adapted easily to three-dimensional environments.

Andres' generalised supercover line

A method which does not use the Bresenham algorithm is presented by Andres [8]. The supercover line is the line which contains *all* the points the real line intersects. It is based on the following property:

Let's consider the two-dimensional line $D : ax + by + c = 0$. Then the supercover line corresponds to all the points in the plan verifying:

$$S(D) = \left\{ (x, y) \in Z^2 \mid -\frac{|a| + |b|}{2} \leq ax + by + c \leq \frac{|a| + |b|}{2} \right\}$$

⁴Taken directly from its code source.

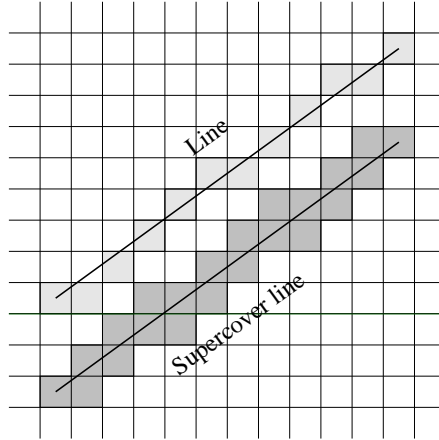


Figure 7.8: Comparison between line and supercover line.

The left and right terms of the relation provide the thickness of the line.

As an example, if the line D passes through points $(1, 3)$ and $(5, 9)$, then we obtain $D : 3x - 2y + 3 = 0$, and the points of the supercover line are those verifying the relation:

$$S(D) = \left\{ (x, y) \in \mathbb{Z}^2 \mid -\frac{5}{2} \leq 3x - 2y + 3 \leq \frac{5}{2} \right\}$$

Two methods of generation of such lines are given in [8]. This method works also for rational numbers coordinates. A similar formula can be used for three-dimensional case (and N -dimensional case).

7.6 Our supercover line tracing algorithm

In order to cope with the difficulties introduced by obstacles, the vision algorithm uses a line tracing algorithm which, compared to classical line drawing algorithms, has to draw *all* the points pierced by the ideal line, not only one per coordinate (see figure 7.8).

We propose a new algorithm to draw supercover lines, based on the Bresenham one. This section describes our supercover line algorithm.

The Bresenham algorithm (presented in section 7.5.1) has two particularities which give its interest: it uses no floating point operations or multiplication/division operations, which are time-consuming on actual processors. Square by square, it accumulates the error between the ideal line and the drawn line. When the error exceeds a certain value (corresponding to the dimension of a square), the drawn line is adjusted to the ideal one.

The Bresenham algorithm cannot be used in our case, because, unlike it, the vision has to take into account *all* the points the ideal line pierces. We needed another algorithm, and we have chosen to base it on Bresenham one

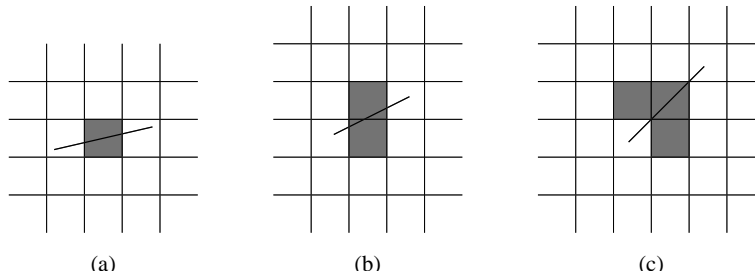


Figure 7.9: In a Bresenham-based supercover line tracing, during a step three cases appear: (a) one square drawn, (b) two squares drawn and (c) three squares drawn.

because of the two properties shown above. The key difference to Bresenham algorithm is that we use also the error of the previous step.

Because the Bresenham algorithm has already been presented, we present here only the difference between our algorithm and this one. In the following, we will use the following terms:

- Next square: the square which will become current in the next step. It will always be drawn.
- Drawn square: a square which belongs to the (supercover) line, so it will be drawn.
- To check a square: to verify if that square will be drawn or not.

As before, we take into account only the first octant. The other octants work similarly. Suppose a general step in the algorithm. We must differentiate three cases, presented in figure 7.9. In the following we will use figure 7.10 for our explanations. Suppose we are in square A (i.e. A is the current square). As in the Bresenham algorithm, the next square will be B or C. In fact, the next square in supercover algorithm is *always* identical to Bresenham one, and as a consequence it will always be drawn. The difference is that the supercover line algorithm additionally checks some squares, as shown below.

If the Bresenham algorithm does not change the y-coordinate (next square is C), this means that D will not be drawn, so we pass directly to C, and we go to the beginning again.

The difference appears when Bresenham algorithm changes the y-coordinate, i.e. when the next square is B. In this case, both C and D have also to be checked. As seen in figure 7.10, we can know if C and D are drawn or not by the following relation:

```

1 // three cases
2 if (error + errorprev < ddx) // case a: bottom square C will be drawn
3   DRAW_POINT (y-ystep, x);
4 else if (error + errorprev > ddx) // case c: left square D will be drawn

```

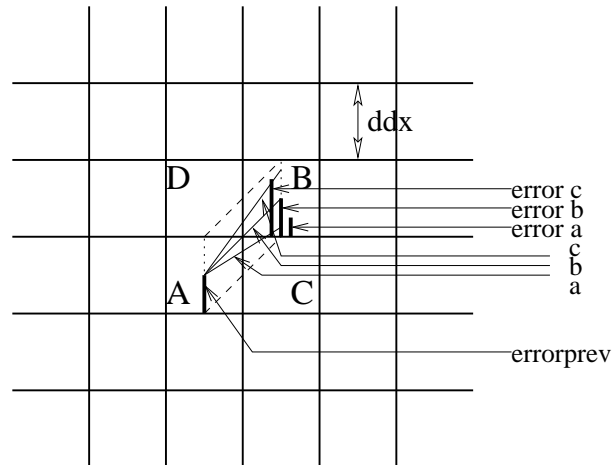



Figure 7.10: Parameters involved in the supercover line algorithm (the next point is B). In case *a* ($\text{error} + \text{errorprev} < \text{ddx}$), square C is drawn. In case *b* ($\text{error} + \text{errorprev} = \text{ddx}$, the ideal line passes through a corner), both squares C and D are drawn. In case *c* ($\text{error} + \text{errorprev} > \text{ddx}$), square D is drawn.

```

5  DRAW_POINT (y, x-xstep);
6  else{ // case b (corner): bottom C and left D squares will be drawn
7    DRAW_POINT (y-ystep, x);
8    DRAW_POINT (y, x-xstep);
9  }

```

error is the current error (in point B), while **errorprev** is the previous error (in point A). As described in the Bresenham algorithm, the error is the “distance” (non-normalised) from the ideal point to the grid line below the ideal point.

This algorithm draws a square if the ideal line passes through one of its corners. It is possible to remove this hypothesis by simply removing lines 6–9.

The full implementation of our algorithm for drawing supercover line can be found in appendix B.

7.7 Optimisations of the vision algorithm

Several optimisations can be done on the vision algorithm presented above. Their aim is to decrease execution time and memory size.

1. *Symmetry*: As noticed, a useful property of the vision relation is the symmetry: If the square *A* is visible from the square *B*, then *B* is also visible from *A*. When computing the visibility fields of all the squares, this can be used to decrease both execution time and memory used by a factor of 2. Our tool allows the use of the symmetry by computing

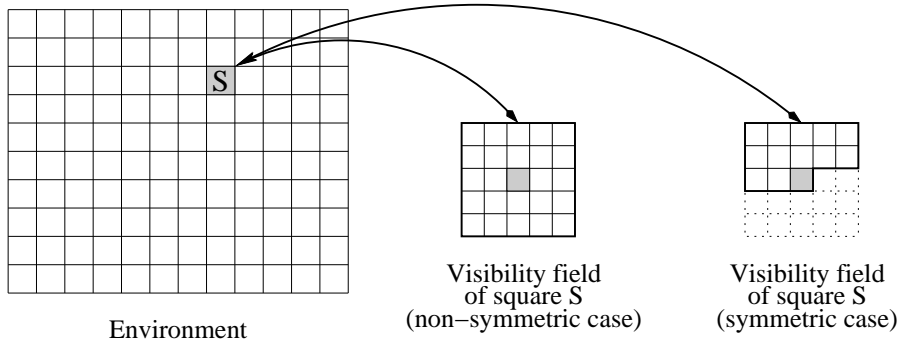


Figure 7.11: Memory space needed in non-symmetric and symmetric cases.

the visibility from (y_A, x_A) to (y_B, x_B) only if $(y_B < y_A) \vee ((y_B = y_A) \wedge (x_B < x_A))$, as shown in figure 7.11. It is worthwhile to notice that, while computing the fields only in half of the cases is always twice faster, using half of memory leads to small overheads during the reading of these fields. This comes from the fact that every time we need to know if a square A is visible from square B , we have to check which of A sees B and B sees A was computed. If the available memory is sufficiently large, it is possible to cope with this trade-off by computing only the first half and just copying it into the second half (however, synchronisation problems appear in parallel execution).

2. *Rhombus allocation on 4-connectivity*: We have previously seen that the field on 4-connectivity (a rhombus) has about twice less squares than for 8-connectivity. This allows to halve the execution time and the memory needed. For memory, however, as the computer memory is linear, a mapping (correspondence) should be made from the rhombus (logical coordinates) to the memory (physical coordinates), as shown in figure 7.12 on the next page. A method for doing this mapping is to *order* the squares of the field, and using this index to access linearly the memory. This ordering has to work for any vision radius.

It is worthwhile to notice that a bijection rhombus \rightarrow square cannot be obtained (easily) by a rotation operation of 45 degrees. Firstly, the rhombus is a square in continuous coordinates, but *not* on discrete ones (figure 7.12). Secondly, a rotation generates irrational coordinates, so they need to be converted to integer ones.

Both these methods of memory optimisation add an overhead both on initialisation and access phase. In 4-connectivity case, our tool is optimised for execution time, but not for memory requirements.

3. *Programming type of the vision matrix*: This is actually a coding issue. The vision relation having a binary result, a bit is sufficient to

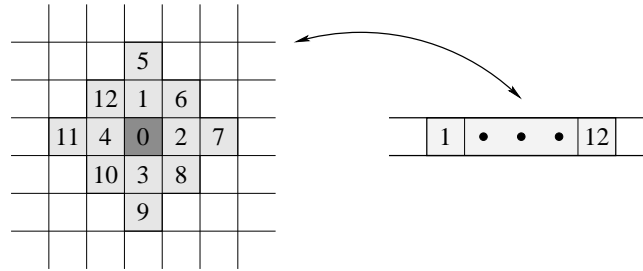


Figure 7.12: To optimise the memory requirements in 4-connectivity case, a bijection should be made between the visibility field (a rhombus) and the memory (linear), for example by ordering the squares of a rhombus.

store it. The use of bits instead of character type (8 bits) or integer type (generally 32 bits on actual machines) reduces thus the memory occupied by a factor of 8 (if character) or 32 (if integer). The trade-off is that every time we use the bit, it has to be isolated from the other bits (with an AND operation for example). Our tool uses the character type (8 bits).

4. *Type of propagation*: As said in the beginning of the chapter, in our tool, the vision is influenced only by obstacles, which are fixed during the simulation. Thus, the visibility fields are fixed also (the second method, dynamical, can be used also when the obstacles are not fixed). Several solutions have been explored for their computing:
 - (a) *Statical*: The visibility fields are entirely calculated at the beginning of the simulation. The characteristics of this method are: a lot of memory used, a lot of initialisation (all the fields) and a fast usability.
 - (b) *Dynamical*: The fields are calculated during the simulation, when they are needed. It can be used also when the obstacles are not fixed. No memory space is needed. This works also if the fields are not static. During a test, about 75% of squares were visited by agents. Thus, the field of some squares do not need to be calculated, while the field of others needs to be calculated several times.
 - (c) *Mixed (cache-like)*: The fields are calculated during the simulation and, based on the same principle as the cache, only some of them (the most frequently used for example) are stored. This is a trade-off between the memory used and the access time of the visibility fields.
 - (d) *Mixed (lazy)*: The fields are calculated dynamically when they are needed, and stored in the matrix once calculated. This method

uses the same amount of memory as the static one, but can decrease the execution time only when a little part of the visibility matrix is accessed during the simulation. However, during a test, about 75% of squares were visited by agents. Additionally, the load balancing is no more optimal and depends highly on the parameters of the system.

Our tool allows the use of the first two methods (static and dynamic).

5. *Matrix traversal method*: In the static case, when using frequently high quantities of data, the locality of the data becomes yet more important. The base principle is that, once in cache, the data be processed as much as possible. A typical situation appears in matrix traversal, especially when the squares near the current one are also accessed. Additionally to the simple raster scan (or scanline, from left to right and from top to bottom), several so-called *space filling curves* have been proposed, such as Hilbert curve, zig-zag scan and Regazzoni curve [62]. Their goal is to access all the elements of the matrix in an order that reduces the average distance between two consecutive processed elements, hence its interest in cases when for each element nearby elements are also used. Nevertheless, this does not appear in our algorithm, as will be shown also by performance measurement results: The vision matrix is not reused, but only the obstacles matrix, which has a size much smaller than the vision matrix, and hence a negligible influence on the performance. Therefore, the vision in our tool uses the simple raster scan.

7.8 Parallel issues of the vision algorithm

Independently of the optimisations presented above, our vision algorithm is highly parallel, both during the initialisation phase and during the access phase. In the dynamic case, the computing of the visibility fields is done during the simulation, by the processor associated to the square concerned. The load balancing depends on many factors, especially on the run-time behaviour of the agent, so it is difficult to express. Thus, in the following we consider only the static type of propagation.

As the visibility fields are static, during the access phase, the matrix storing them is read-only, so no data consistency problem occurs.

The initialisation phase involves the computation of the visibility fields of all the squares. Computing the field of a square involves *reading* the environment (to find if a square contains an obstacle or not) during the supercover line algorithm, and *writing* the result in the visibility matrix of the square. The obstacle information is stored in the environment matrix, shared read-only in this algorithm by all the processors. The writing operations

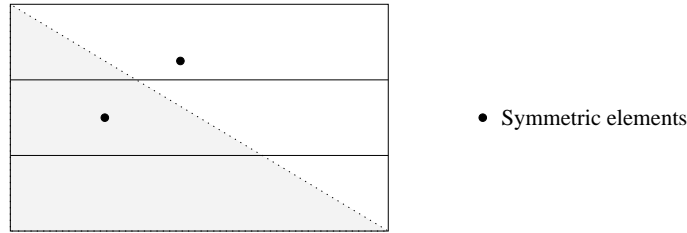


Figure 7.13: Load imbalance of a symmetric relation: on an equal-sized horizontal decomposition of a matrix (3 domains in this figure), load unbalancing appears if the symmetry (the computes are done on the grayed half of the matrix only) is taken into account.

apply only to the square involved, hence cache conflicts are very unlikely. The load-balancing is not a real issue either. In a toroidal world without obstacles, the computing of the field of a square takes the same amount of time⁵. The obvious parallelisation is thus the partitioning of the domain (presented in section 4.4), with domains of equal number of squares. If there are obstacles located at random coordinates, the load is balanced on expectation. Otherwise, domains based on their size and their number of obstacles may be imagined. Our library parallelises this algorithm by using an equal-sized decomposition of the domain: each domain contains the same number of lines. In fact, if the total number of lines in the environment is not divisible by the number of domains, the number of lines between any two domains may be differing by 1, and hence a small load unbalancing can appear.

It is worthwhile to mention a property of the vision model: the decomposition giving an ideal load-balancing is the same, whether the relation is symmetric or not. For 2D matrices, the linear decomposition of a symmetric relation (the matrix M is symmetric) is different than the linear decomposition of the same non symmetric relation (figure 7.13 presents such an example). Instead, in our case, the vision matrix (corresponding to the environment) is a 4D matrix, and, while the decomposition appears at matrix level (first and second dimensions), the symmetry is exploited at element level (third and fourth dimensions), as shown in figure 7.11 on page 129. Thus, the same number of elements is calculated, but the work on each element is halved.

⁵We take into account only the processor instructions here, not the cache considerations, often quasi-unpredictable, such as some data already in cache.

7.9 Memory and timing models

Before proceeding to theoretical models of the vision algorithm, we first present the parameters which influence it:

1. Environment size (N squares) and type: the type of the environment can be toroidal or with edged.
2. Connectivity type (c : 4 or 8): as previously seen, both the memory requirements and the execution time are influenced by the connectivity type.
3. Vision radius (r_v).
4. Walls percentage (w): the more obstacles exist in the environment, the faster the implementation works (the supercover algorithm stops at the first obstacle met).
5. Number of processors (P).

We want also to emphasise that this is an example of algorithm which needs not only a mathematical complexity analysis, but also a practical analysis. There are several optimisations which do not affect mathematical complexity, but practical measurement results, where the gain in execution time or memory is noticeable.

We consider here only the vision of statical type, which means the visibility fields are calculated at the beginning of the simulation. The 4D matrix of the vision stores elements of value `true` or `false`. The vision algorithm consists of computing and storing each of these values. The computing part gives its timing model, and the storing part gives its memory model.

Note: For the computing of complexities we will use an “extended” form of the classical order class O , as described in paragraph *Note* at page 163.

7.9.1 Memory model

We are interested in the application memory model mainly because the vision algorithm uses huge amounts of memory.

In the sequential execution, the vision matrix stores the visibility field of all the squares of environment (figure 7.4 on page 121). There are N total squares, and the visibility field of each square has $(2r_v + 1)^2$ elements (formula 7.2 on page 119). Therefore, the number of elements needed by the vision matrix of the *non optimised* algorithm is given by the following relation:

$$\text{mem}_{nonopt}(N, r_v) = N(2r_v + 1)^2 \quad (7.3)$$

As noticed, it does not depend on the walls percentage w , nor the connectivity type c . We obtain the theoretical memory complexity of this algorithm,

as a function of the number N of squares and the vision radius r_v :

$$\text{mem}(N, r_v) = O(Nr_v^2)$$

For a given r_v the complexity becomes:

$$\text{mem}(N) = O(N)$$

A special case appears when $r_v = \alpha N$, with α a constant. Notice that this means that *the characteristics of agents can be modified*: a larger environment leads to a large visibility field of agents. In this case the memory complexity becomes:

$$\text{mem}(N) = O(N^3)$$

However, the optimisations presented above allow the decreasing of this number. The symmetry decreases by 2 this number. In a connectivity of 4, it decreases by about 2, so we will use the term $c/8$, which gives 1 for 8-connectivity and $1/2$ for 4-connectivity. If we suppose that in the non optimised case each element has the type `char` (stored on 8 bits), then using 1 bit for each element decreases by 8 the memory requirements. Thus, we arrive at the following practical memory requirements of the *optimised* algorithm:

$$\text{mem}_{opt}(N, r_v, c) = N(2r_v^2 + 1)^2 \frac{1}{2} \frac{c}{8} 18 = \frac{N(2r_v + 1)^2 c}{16} \frac{c}{8} \text{bits}$$

This number does not depend on the walls percentage w .

As an example, in a 4-connected 1024×1024 environment with radius of 16, we obtain the following practical memory requirements: $M_{unopt} \approx 1GB$ and $M_{opt} \approx 34MB$. As noticed, the optimisations allow a noticeable decreasing of the memory needed of about 30 times, which can influence also the execution time due to latency of memory hierarchy.

For the parallel execution, since the vision matrix is equilibrated among the processors, the memory needed by each processor is equal to the total memory needed (in optimised or non optimised case) divided by the number of processors P .

7.9.2 Timing model

Let t_{elem} be the average time to compute the value of one element of the vision matrix, i.e. whether two squares are visible from each other or not. This time depends on w , r_v and c (it depends on the configuration of the obstacles in environment and the distance between the two squares). As previously shown, the decomposition of the matrix is equilibrated among the processors. Therefore, for the parallel execution of the vision algorithm, the time needed by one processor is approximately the total time divided by

the number of processors P . The total execution time is the multiplication between the number of elements $N(2r_v+1)^2$ of the vision matrix (formula 7.3 above), and the average time t_{elem} to compute the value of one element. The execution time of the *non optimised* algorithm is therefore given by the following relation (as shown above, the term $c/8$ takes into account also the connectivity):

$$\text{time}_{unopt} = \frac{1}{P}N(2r_v + 1)^2\frac{c}{8}t_{elem}$$

We obtain from this the theoretical time complexity of this algorithm:

$$\text{time}(N, r_v, P) = O\left(\frac{1}{P}Nr_v^2t_{elem}\right)$$

For P and r_v given, we obtain:

$$\text{time}(N) = O(N)$$

The symmetry optimisation presented above decreases the execution time by a factor of 2. We obtain thus the following *optimised* execution time:

$$\text{time}_{opt} = \frac{1}{2P}N(2r_v + 1)^2\frac{c}{8}t_{elem}$$

Consequently, the use of the symmetry property is more efficient than not using it, and we have implemented both (see appendix A.5).

7.10 Performance of the vision algorithm

7.10.1 Experimental condition overview

System information

For measurements, two systems have been used. The first system is an SMP machine, 450 Sun Enterprise⁶ with 4 Sparc processors at 400MHz, 1GB of memory and 4MB of cache per processor. It runs GNU/Linux RedHat 6.2, gcc (GNU Compiler Collection) version egcs-2.91.66, and the parallel library is the native one, linux threads⁷, an implementation of the Posix threads standard [64]. The compiler optimisation parameter has been `-O3`.

The second system is a DSM machine, SGI Origin 2000 [34] with 64 R10000 processors at 195MHz, 24GB of memory and 4MB of cache per processor. It runs Irix version 6.5, the compiler used is its native one, MIPSPro version 7.3.1.2m, with Irix native multi-threading library. The compiler optimisation parameter has been `-Ofast=ip27`.

For both systems, the memory allocation library used has been their default system library.

⁶<http://www.sun.com/servers/workgroup/450>

⁷<http://pauillac.inria.fr/~xleroy/linuxthreads>

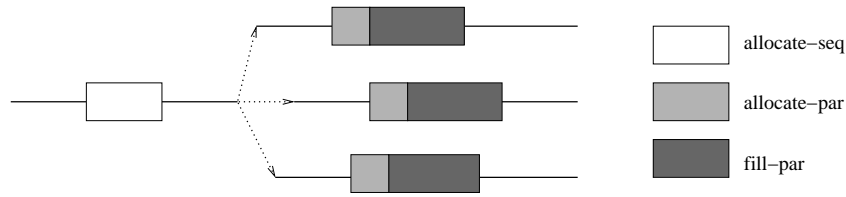


Figure 7.14: Execution of the three functions of the vision (case of 3 threads).

Measurement information

As described in section 4.1.2, several indicators need to be taken into account.

Firstly, each test has been executed once, but in mono-user mode, to reduce at maximum the influence of external programs. Also, we have experimented two times: wall-clock time (given by `times` function), and processor-time (given by `clock` function). Even if they have generated results which do not differ very much, for our plots we have used the results given by `times`, as it is the reference from the user point of view. Generally, these functions have an accuracy better or equal to 10ms⁸. The overhead of the parallel version is negligible. In fact, the only difference between a pure sequential version and the parallel version executed by each thread is only the dimension of the domain.

Two kinds of graphics will be presented. The first, the bar plot, plots for each number of processors the arithmetic average of execution time of all the processors during the *same* execution, along with the maximum and the minimum execution times. It allows to discover load unbalancing. The second plots the execution time, the speed-up and the efficiency for each processor, using always the *slowest* processor. It allows to discover mainly the scalability of the algorithm from the final user point of view. Also, we have used one thread per processor, therefore the terms processor and thread can often be used interchangeable in the analysis below.

7.10.2 Vision algorithm implementation

In static case (static computing of visibility fields), the vision uses a 4D matrix. In our library, the matrix is dynamically allocated, dimension by dimension, and afterwards filled with the results of the visibility fields. In order to measure separately the execution time of each step, this is implemented through three functions, shown in figure 7.14 (`vm` is the vision matrix):

1. `allocate-seq` function. The first two dimensions involve all the environment, hence they are allocated in sequential, before creating the

⁸For instance, on Irix “the resolution of the clock is 10 milliseconds on IRIS workstations.” (man page of function `clock`).

threads. From the programming point of view, this function ensures that all the elements `vm[i][j]` are allocated.

2. `allocate-par` function. The other two dimensions involve each square separately, and they are allocated in parallel, by its own thread. As previously shown, the memory size allocated depends whether the symmetry is taken into account or not. From the programming point of view, this function ensures that all the elements `vm[i][j][k][l]` are allocated.
3. `fill-par` function. Immediately after allocation, each element `vm[i][j][k][l]` of the vision matrix is filled with the appropriate values. This operation is done in parallel, by each thread. As previously shown, the number of computations depend whether the symmetry is taken into account or not.

We will present the performance of each of these three functions.

Two versions of the parallel allocation (allocate the third and the fourth dimension of the vision matrix) have been implemented. The first version allocates several small vectors for each square, while the second method allocates two large matrices for each domain (processor).

Several parameters influence the execution time of the vision algorithm. Also, for more precision, in each graphic the legend contains information about all these parameters. They are:

- N , the number of squares in the environment (y and x in legends).
- The type of the environment (t in legends, $t0$ for toroidal and $t1$ for edged).
- r_v , the vision radius (r in legends).
- w , the percentage of obstacles (walls).
- c , the connectivity (4 or 8).
- s , if we use the symmetry property of the vision ($s1$) or not ($s0$).
- P , the number of processors.

All the legends have the form $y.x.t.r.w.c.s.$, according to the parameters above.

7.10.3 Influence of parallel memory allocations

In this section we present the overhead of numerous parallel memory allocations, appearing in the basic version of the parallel allocation. For each

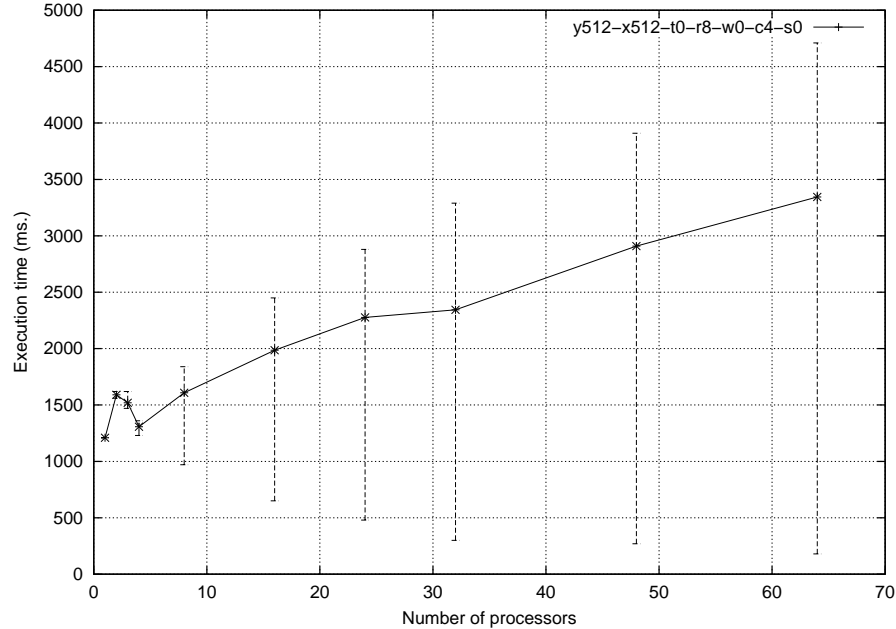


Figure 7.15: Bar plot of the execution time of the parallel allocation (basic version), showing increasing average time and important load unbalancing (DSM Origin 2000).

square of environment, this version allocates $r_v + 1$ vectors (of $2r_v + 1$ elements) in symmetric case ($2r_v + 1$ vectors in non symmetric case), see figure 7.11 on page 129. This gives in symmetric case $N(r_v + 1)$ small memory allocations ($N(2r_v + 1)$ for non symmetric case) for all the environment. Note that the total number of allocations is identical, no matter the number of threads used. As similar results appear for several values of the parameters, the following parameters have been used for illustration purposes: a toroidal environment of 512×512 squares, without obstacles, for a connectivity of 4 and a vision radius of 8. In non symmetric case, these parameters correspond to about 4 millions allocations ($512^2 * (2r_v + 1)$) of 17B each ($2r_v + 1$ elements with 1 byte/element).

For the DSM machine (with 64 processors) there are several memory allocators available. We have used the default allocator provided by `malloc` function. Figure 7.15 presents the bar plot giving the execution time of parallel allocation. Starting from 8 processors, an important load unbalancing can be noticed. Even if the number of allocations is identical, the execution with 1 thread is always faster than the execution with any other number of threads. Also, it is increasing starting from 4 processors, number probably corresponding to the trade-off between benefits and overhead of parallel memory allocation. This affects also the execution time of the whole vision

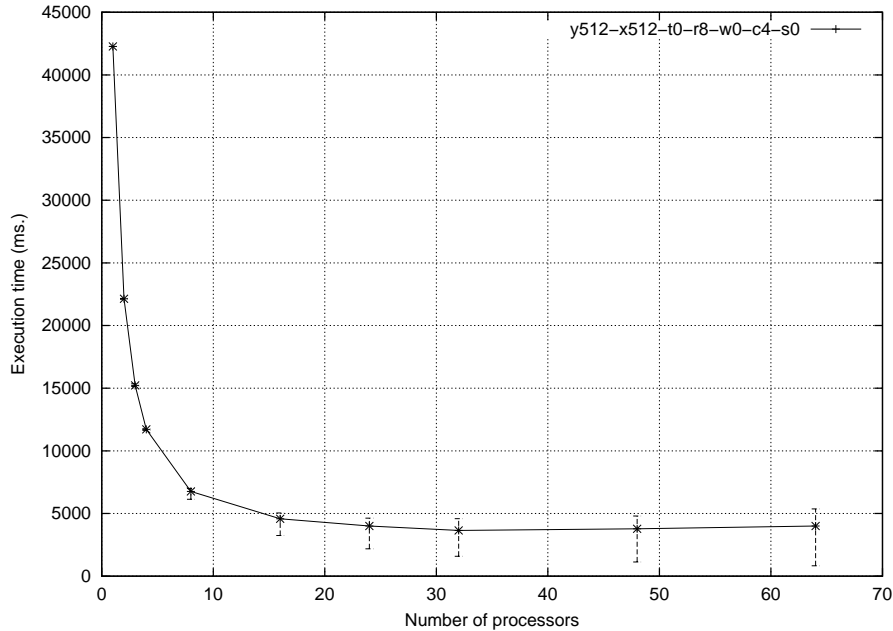


Figure 7.16: Bar plot of the execution time of the whole vision algorithm (basic version) showing slightly increasing execution time starting from 32 processors (DSM Origin 2000).

algorithm (allocation plus filling), as shown in figure 7.16, where the load unbalancing and the execution time of the slowest processor start slightly increasing from 32 processors.

For the SMP machine (with 4 processors), using its standard `malloc` function, the load unbalancing is much reduced, probably because of the SMP architecture. Figure 7.17 presents a case where it is visible. Unlike the DSM machine, for all the parameters, the execution time *decreases* with the number of processors (experimented with up to 4 processors).

These experiments confirm that memory allocation may be a sensitive issue in a parallel program. Consequently, we have implemented an enhanced version of the memory allocation in our implementation. The number of allocations appearing in parallel has been reduced to 2. Each thread allocates two larger memory blocks (corresponding to the third and the fourth dimensions of the vision matrix), depicted in figure 7.18.

In the following section we will see that the enhanced version has greatly improved the performance of parallel allocations. Therefore, no other parallel memory allocator systems have been tried.

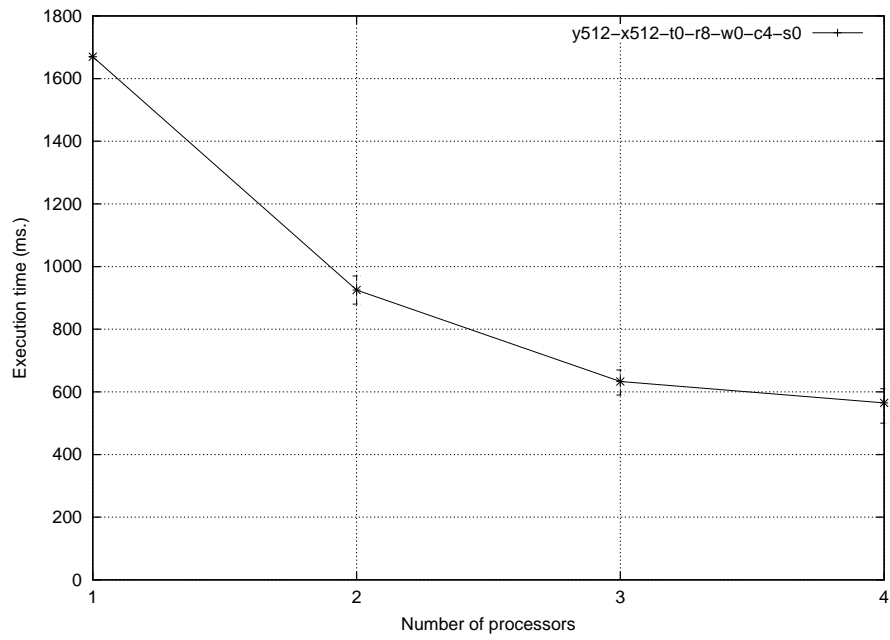


Figure 7.17: Bar plot of the execution time of the parallel allocation (basic version), showing decreasing average time (SMP Sun 450).

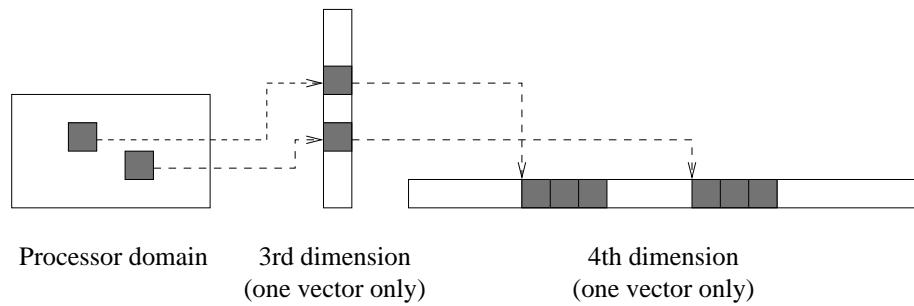


Figure 7.18: Enhanced memory allocation, where each processor allocates only two blocks.

7.10.4 Performance measurement results

As shown in the overview, the vision algorithm consists of three parts: sequential allocation, parallel allocation and parallel filling. In this section we present the performance of each of these functions.

Unless otherwise explicitly stated, the simulation parameters used are those of a typical simulation: a toroidal environment of 512×512 squares, without obstacles, with a connectivity of 4 and a vision radius of 8. In these tests we have chosen the non-symmetric case in order to obtain higher, hence more accurate, execution times for this simulation. As described in section 7.7 and as we will show later in figure 7.30 on page 151, the symmetry reduces *exactly* twice the execution time of the algorithm (with respect to the difference in data size). However, in applications the faster symmetric case may be used.

This is a representative case for the majority of tests for two reasons: any measured execution time is greater than 1 second (a minimum time to obtain a sufficiently accurate value from our point of view), and cache effects are still noticeable (if they exist). Also, the order of magnitude of these parameters is realistic for our planned simulations.

Sequential allocation performance

The sequential allocation consists of allocating the first and the second dimension of the vision matrix. This means allocating mainly a block of N pointers (N is the number of squares in environment). Even if executed sequentially, its execution time is very small compared to the other two functions. As an example, for one processor, we have obtained statistically an execution time of less than 1ms for this function, while the execution of the whole algorithm took about 10 seconds. Consequently, it counts for less than 10^{-4} of the execution time of the whole algorithm. As it is executed sequentially, Amdahl's law (section 4.1.4, page 78) limits the speed-up on P processors to:

$$S_{\max} = \frac{1}{10^{-4} + \frac{1-10^{-4}}{P}}$$

For 64 processors (the case for our DMS machine), this gives a limit of speed-up of about 63.6. For our requirements, this value is very close to the ideal speed-up of 64, therefore we have considered the influence of this sequential function as negligible, and we will not present it in our results.

Parallel allocation performance

Here we use the enhanced version of parallel allocation, where each thread allocates two larger memory blocks (see figure 7.18). The results obtained are given in figure 7.19. We notice that this version of parallel allocation is not

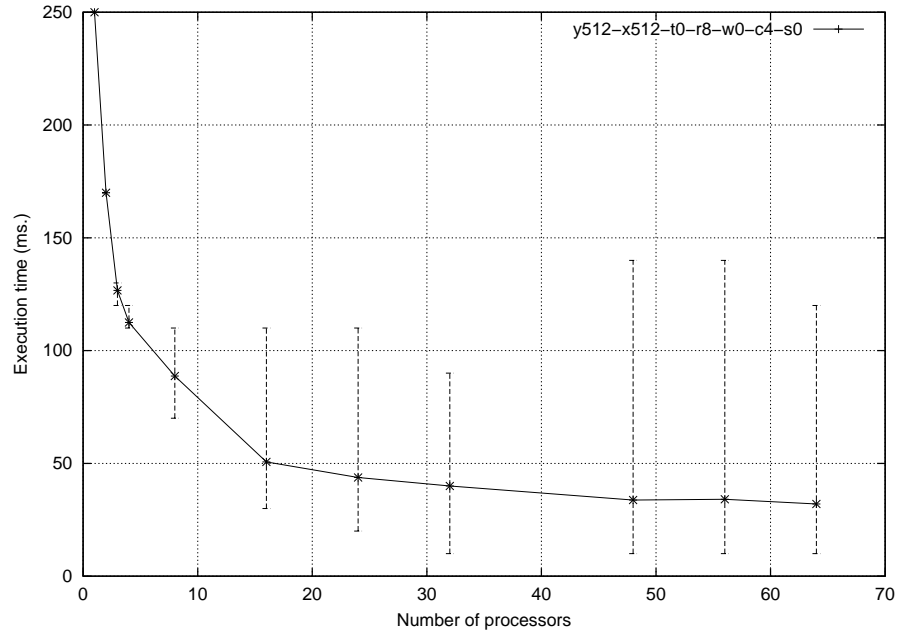


Figure 7.19: Bar plot of the execution time of the enhanced parallel allocation, showing low execution times (compared to those in figure 7.15) (DSM Origin 2000).

scalable either. Nevertheless, it is scalable for *average* execution time and, more important, it has low execution times for any number of processors.

Parallel filling performance

The filling is a highly parallelisable operation, since all the computations involved are independent. Therefore, for sufficient data to process, we expect to obtain good load balancing, and speed-ups close to the ideal speed-up. In figure 7.20, which presents the bar plot of the parallel filling, no important load unbalancing can be noticed. The speed-up, presented in figure 7.21, increases nearly linearly up to 48 processors, and is about 49 for 64 processors. We will see that on a higher environment, the speed-up on 64 processors becomes higher.

An interesting result is given by the efficiency curve. Figure 7.22 reveals that the efficiency curve is nonmonotonic. In order to have more information about the source of this non-monotonicity, we have done tests for any number of processors from 1 to 64. Figure 7.23 presents the result.

The reason of this non-monotonicity is the load unbalancing given by the horizontal decomposition of the environment. As described in section 7.8 (page 131), if the number of lines in environment is not divisible by the number of domains, some domains will obviously have a number of lines

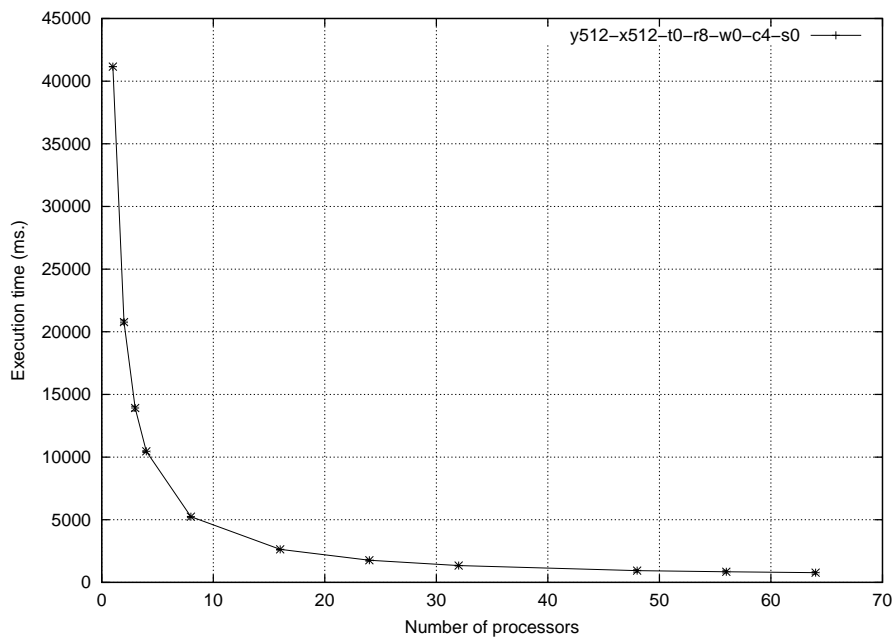


Figure 7.20: Bar plot of the execution time of the parallel filling, showing load balancing (DSM Origin 2000).

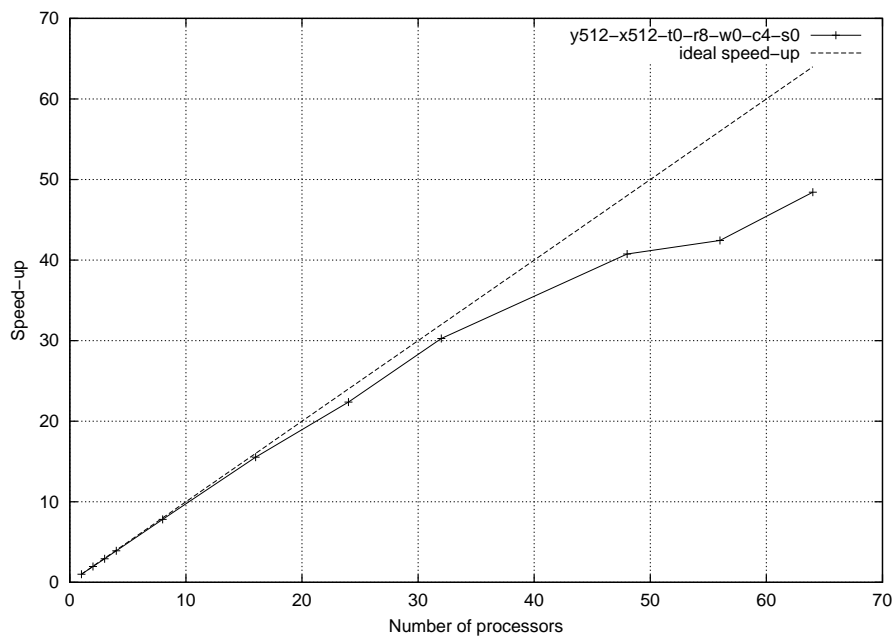


Figure 7.21: Speed-up of the parallel filling, showing good scalability (DSM Origin 2000).

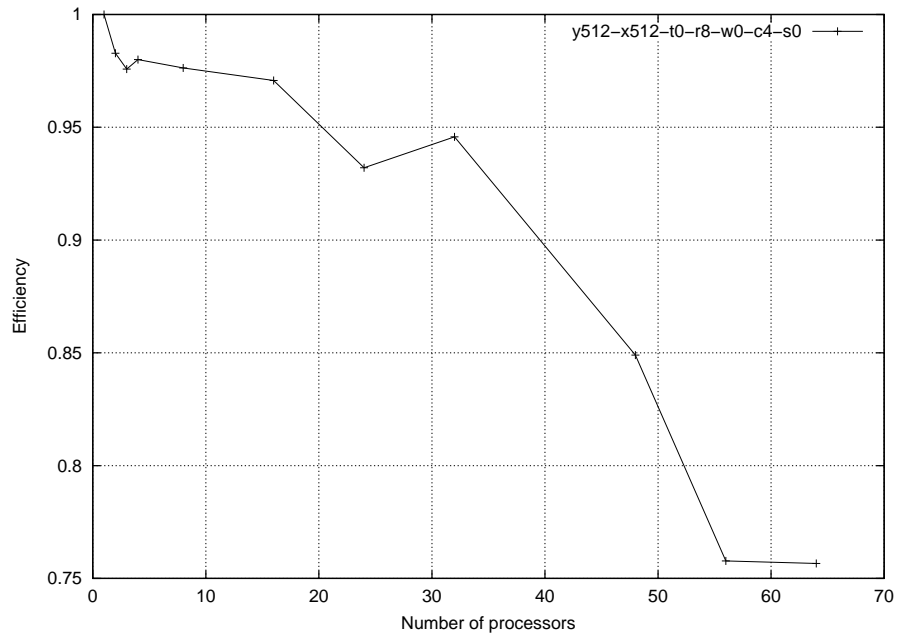


Figure 7.22: Efficiency of the parallel filling, showing a nonmonotonic curve (DSM Origin 2000).

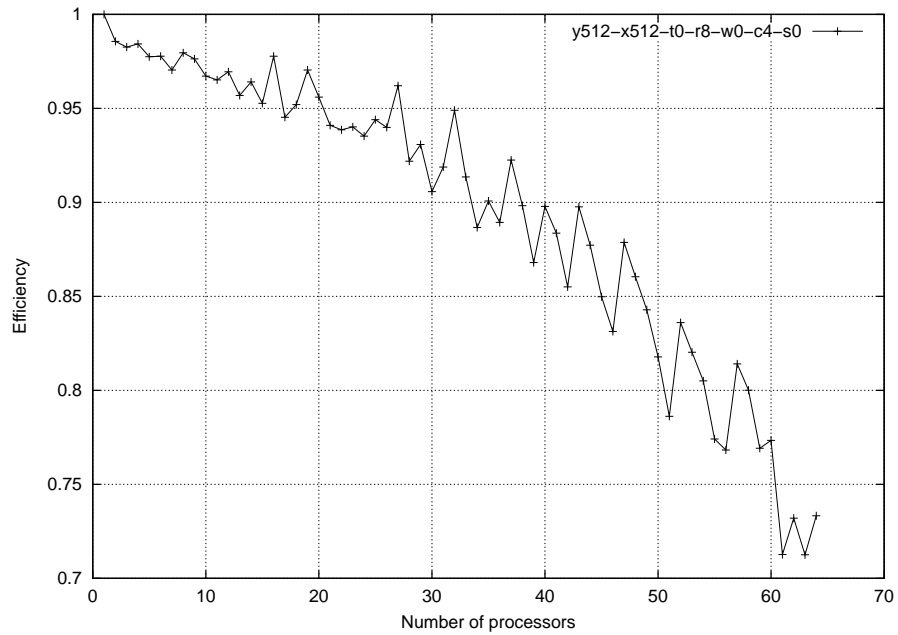


Figure 7.23: More accurate efficiency of the parallel filling, showing a zigzag curve (DSM Origin 2000).

greater by 1 than others. (For instance, if 33 processors are used to compute the visibility on an environment with 512 lines, the average number of lines given to each processor is $512/33 \approx 15.51$, therefore some processors will process 15 lines, while others will process 16 lines.)

Let f be the fractional part of the average number of lines ($f \in [0, 1)$). The ideal load balancing, when all the domains have the same number of lines, is obtained when $f = 0$. If $f \neq 0$, then some domains have 1 line more than others. As the efficiency curve is based on the execution time of the slowest thread, the load unbalancing is the most visible when there is only one domain with number of lines greater by 1 than the others, i.e. when f is close to 0 (but different than 0). The more domains have a number of lines greater by 1 than others (i.e. the greater is f), the less visible is the load unbalancing. Then, for our environment of 512 lines, we define the quality of load balancing as the discrete function $Q_{\text{ldbal}} : N^* \rightarrow (0, 1]$ given by ($\{x\}$ is the fractional part of x):

$$Q_{\text{ldbal}}(P) = \begin{cases} 1 & \text{if } \{512/P\} = 0 \\ \{512/P\} & \text{otherwise} \end{cases}$$

Given the discussion above, the greater the value of function Q_{ldbal} , the greater the load balancing. Now, if we plot the efficiency curve together with the discrete function $Q_{\text{ldbal}}(P)$ (figure 7.24), we notice that the two curves match very well, i.e. their local extremes appear at the same number of processors. We also notice that the load unbalancing is more visible as the number of processors increases, since the processing of 1 line has a greater influence when the total number of lines to process is smaller (i.e. the number of processors is greater). Therefore, the load unbalancing given by processing different number of lines gives indeed the efficiency's zigzag shape. This is an example of algorithm where some values of number of processors give better performance than other values.

7.10.5 Influence of caches

The vision algorithm computes only once the visibility field of all the squares. As shown in section 7.8 no cache conflicts appear. For cache performance purposes, we are then interested in the reusability of data. The only data reused is the two-dimensional matrix storing information about position of obstacles, which has a small size compared to the fourth-dimensional vision matrix. As cache conflicts do not appear and data is generally not reused, we expect that the caches have a little influence on the performance of the algorithm.

The vision matrix has $N(2r_v + 1)^2$ elements for non-symmetric case ($N(2r_v^2 + 2r_v + 1)$ elements for symmetric case). In our implementation, each such element is stored as `char`, so it occupies 1B (i.e. 1 byte). For our previous parameters, $N = 512 \times 512$ and $r_v = 8$, this gives 70MB (35MB).

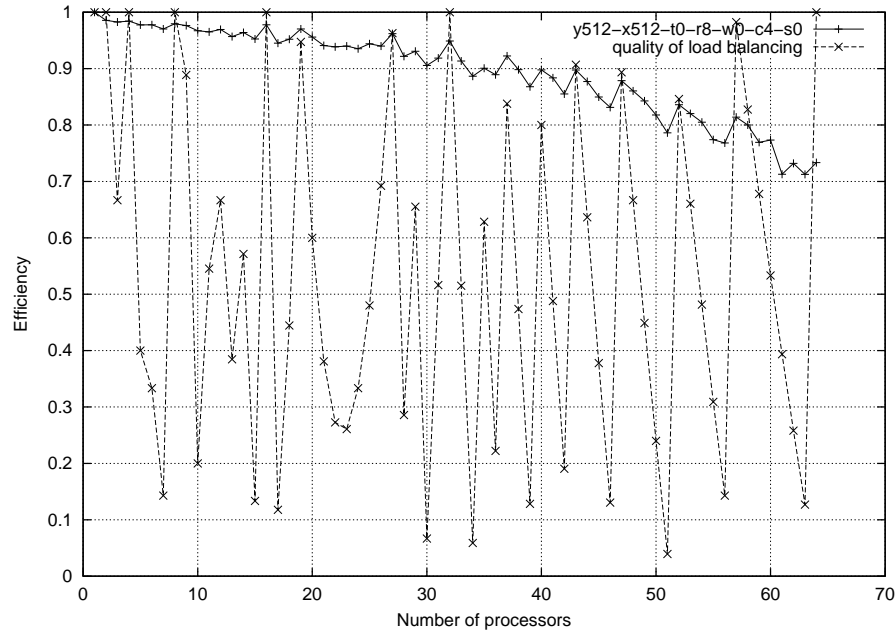


Figure 7.24: Comparison between the quality of the load balancing and the measured efficiency of parallel filling: their local extremes appear at the same number of processors (DSM Origin 2000).

Additionally, we have to add the space needed by the environment matrix, storing information about squares (whether they contain an obstacle or not): $5N$ elements. In our implementation, the integer type (`int`) has been used, occupying 4B each. This gives 5MB for the environment matrix. Summing up, we obtain the total memory processed by the vision algorithm: 75MB (40MB). As the cache of each processor on our DSM machine has 4MB, and the size of the data used by several processors is small for the parameters we have chosen, the data is contained completely in caches starting from 19 (10) processors. As our plots do not present any discrepancy starting from 19 (10) processors, we can conclude that the caches do not influence the performance of the algorithm. (However, high values for r_v may lead to cache misses during the reading of the environment matrix.)

We have not obtained supralinear speed-ups in any case. This result has been expected, since the caches do not influence the performance and the work is completely deterministic (as described in section 4.1.2, page 76).

7.10.6 Influence of the vision parameters

As previously shown, the performance of the vision algorithms depends on several parameters. We have varied these parameters, and obtained the following results:

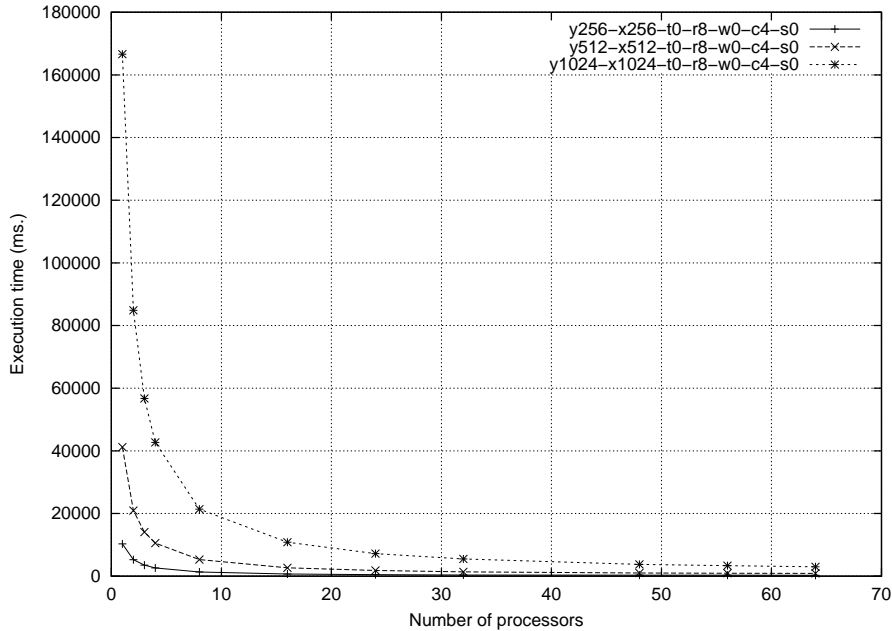


Figure 7.25: Comparative execution time of the parallel filling for environment sizes of 256×256 , 512×512 and 1024×1024 squares, showing that it increases linearly with the number of squares (DSM Origin 2000).

1. *Environment size.* We have executed the vision algorithm with several environment sizes: 256×256 , 512×512 and 1024×1024 . On the DSM machine, starting from 2048×2048 we have received memory allocation error from the system. As shown in the previous section, this corresponds to more than 1GB of total memory to allocate. This may come from the 2GB memory limitation and memory fragmentation of our 32-bit program. Nevertheless, for such high environment sizes the dynamic computing of the vision could be used, or a 64-bit compilation could be studied.

Figure 7.25 presents the comparative execution time of the parallel filling. We notice that in sequential the execution time grows linearly with the number of squares of the environment: about 1s for 256×256 squares, about 4s for 512×512 squares, and about 170s for 1024×1024 squares. Figure 7.26 presents the comparative speed-up of the parallel filling. Up to 32 processors the parallel filling scales very well. Starting from 32 processors the speed-up depends on the size of computation: the greater the environment size, the greater the speed-up.

2. *Type of environment.* As previously shown, the environment can be toroidal or edged. With the horizontal domain decomposition we have used, a load unbalancing can appear when the environment has edges.

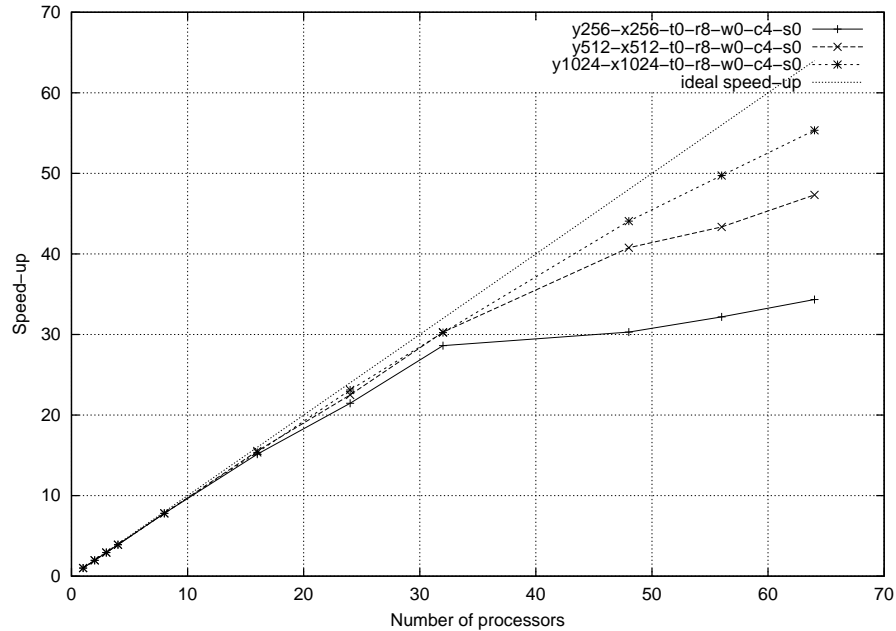


Figure 7.26: Comparative speed-up of the parallel filling for environment sizes of 256×256 , 512×512 and 1024×1024 squares, showing that the greater the environment size, the greater the speed-up (DSM Origin 2000).

In fact, if the environment has no obstacles, the computing of visibility fields of the squares near edges is much faster, because the number of squares to check is reduced by the edges of the environment. The domains which are most influenced are the first and the last domain, which touch the highest number of edge squares (this comes from the horizontal decomposition, section 7.8). The bar plot of execution time of the parallel filling in the torus case is presented in figure 7.27 (for better visibility, the number of processors is greater than 16). The average time is about at the middle of the maximum and minimum times. Figure 7.28 presents the execution times of the application with the same parameters, but on edged case. We notice that the execution times are smaller than in the torus case, since the number of computations of each thread is smaller. Also, the difference between the average time, and the maximum and minimum times is noticeable: The average time is nearer to the maximum time than to the minimum time. This result corresponds to our theoretical result, where the first and last processor have fewer computations to do than all the other processors.

3. *Connectivity and symmetry.* As previously shown, the number of computations is twice for a connectivity of 8 compared to a connectivity

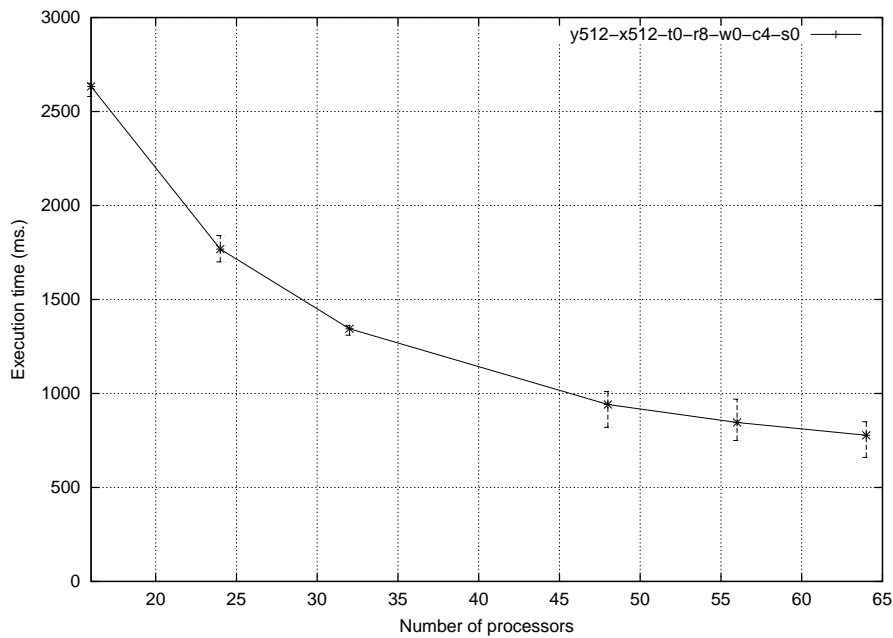


Figure 7.27: Bar plot of the execution time of the parallel filling in a toroidal environment (number of processors greater than 16) (DSM Origin 2000).

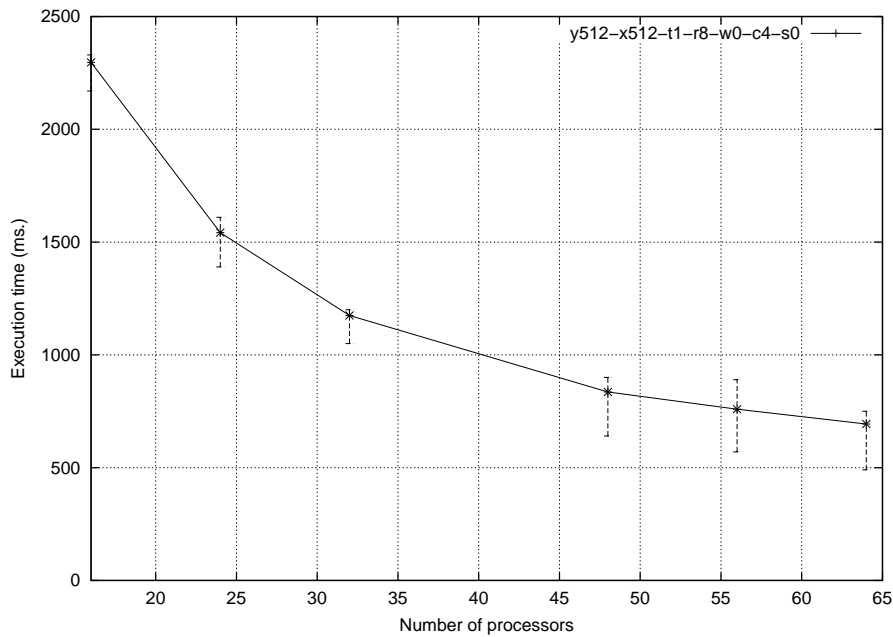


Figure 7.28: Bar plot of the execution time of the parallel filling in an edged environment, showing that a few processors have less execution time than all the others (number of processors greater than 16) (DSM Origin 2000).

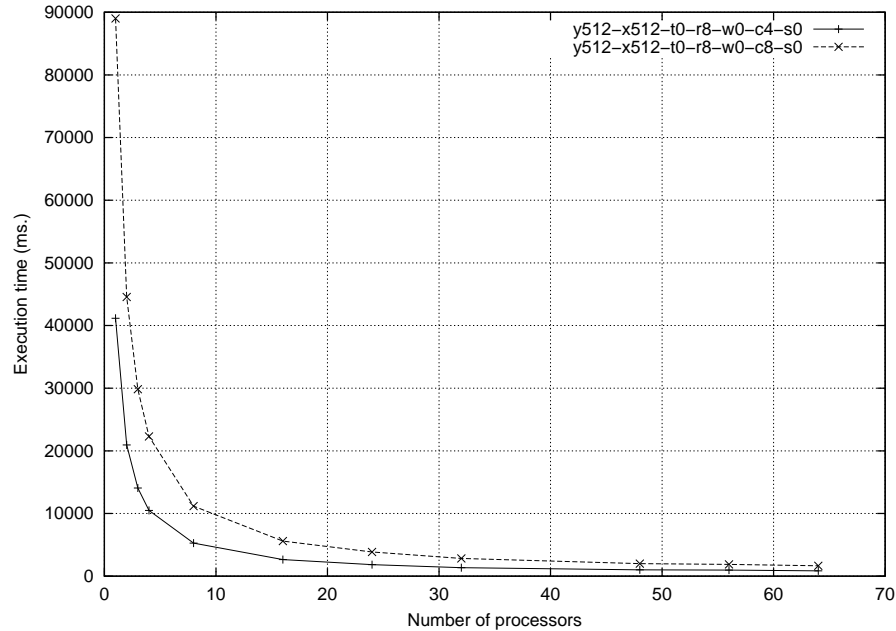


Figure 7.29: Comparative execution time of the parallel filling for connectivity of 4 and connectivity of 8, showing a twice execution time in 8-connectivity case compared to 4-connectivity case (DSM Origin 2000).

of 4. Also, the number of computations is half when the symmetry of the vision is taken into account. The experiments give the same results (figures 7.29 and 7.30).

4. *Vision radius.* In formulae 7.1 and 7.2 (page 119), the number of computations increases with the square of the vision radius. The experiments give approximately the same results: the sequential execution time (figure 7.31) grows with about the square of the vision radius, and the greater the vision radius, the greater the speed-up (figure 7.32). For $r_v = 16$, the execution time does not increase with *exactly* the square of the vision radius, which may probably be due to cache effects.
5. *Number of obstacles.* Since the supercover algorithm finishes when an obstacle is met, the execution time of the vision algorithm depends on the number of obstacles. A theoretical formula is difficult to be given. The experiments (figures 7.33 for execution time and 7.34 for speed-up) show that indeed the greater the number of obstacles, the smaller the execution time and, starting from about 40 processors the slightly smaller the performance. Nevertheless, the sequential execution time and the speed-up are not much influenced by this parameter. In these tests, the obstacles are *randomly* put in the environment.

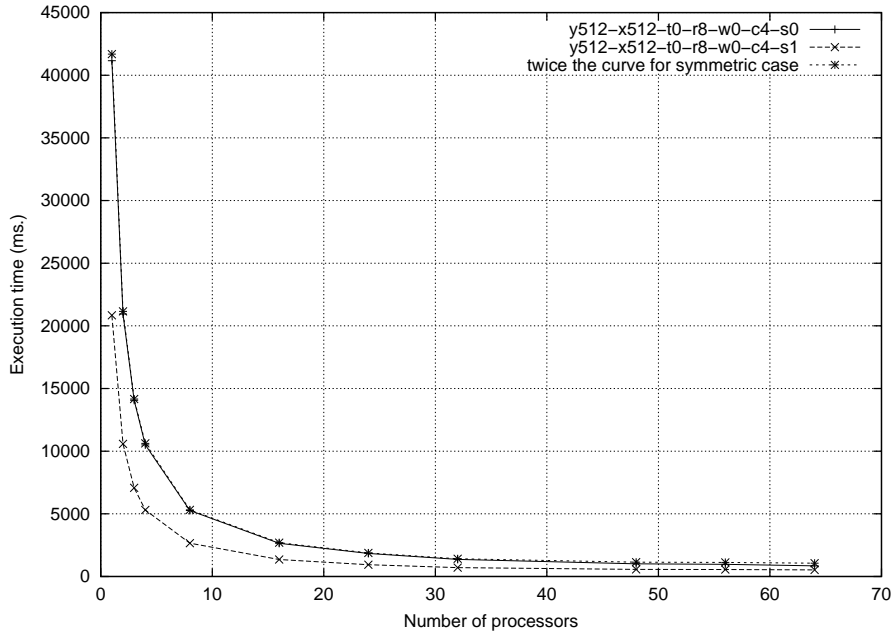


Figure 7.30: Comparative execution time of the parallel filling when taking the symmetry of the vision into account and not, showing that taking the symmetry into account reduces twice the execution time (DSM Origin 2000).

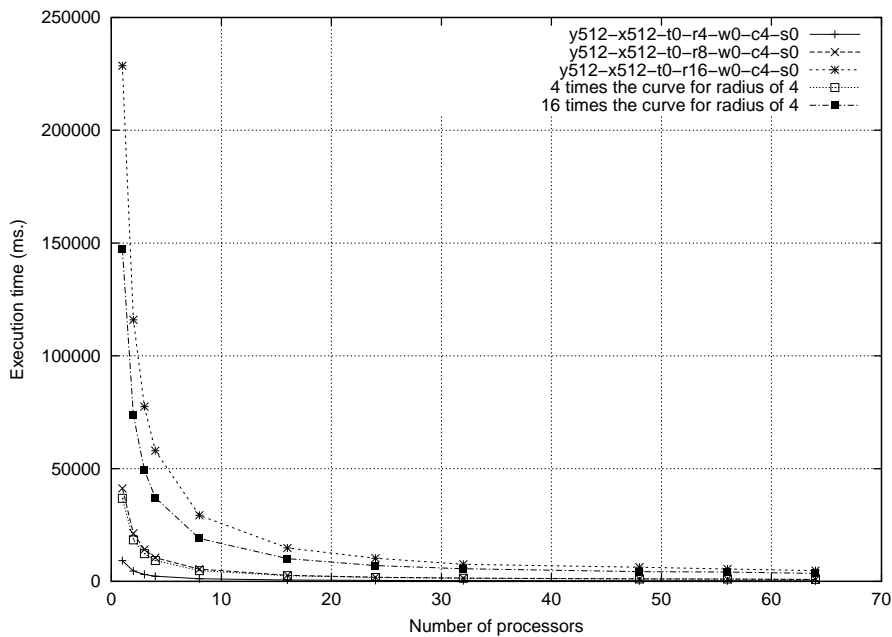


Figure 7.31: Comparative execution time of the parallel filling for several vision radiuses, showing that the execution time increases with about the square of the vision radius (DSM Origin 2000).

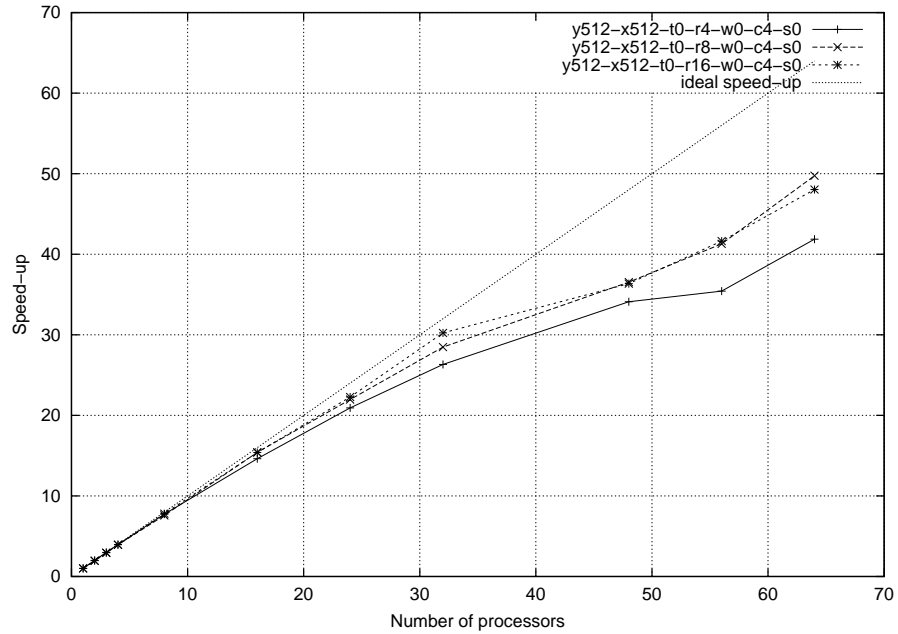


Figure 7.32: Comparative speed-up of the parallel filling for several vision radii, showing that the speed-up increases when the vision radius increases (DSM Origin 2000).

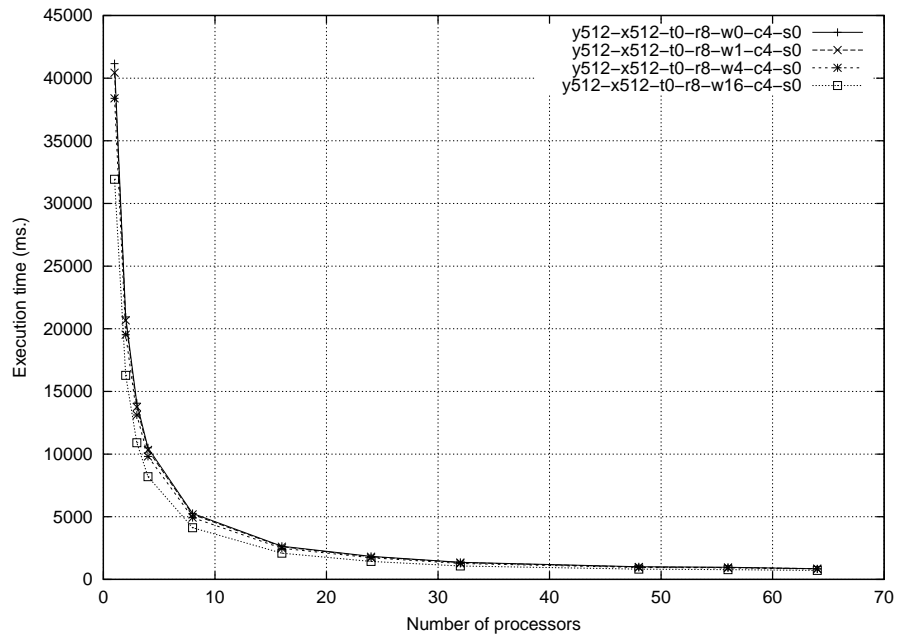


Figure 7.33: Comparative execution time of the parallel filling for several numbers of obstacles (0%, 1%, 4% and 16% of the number of squares), showing that the more obstacles, the less the execution time (DSM Origin 2000).

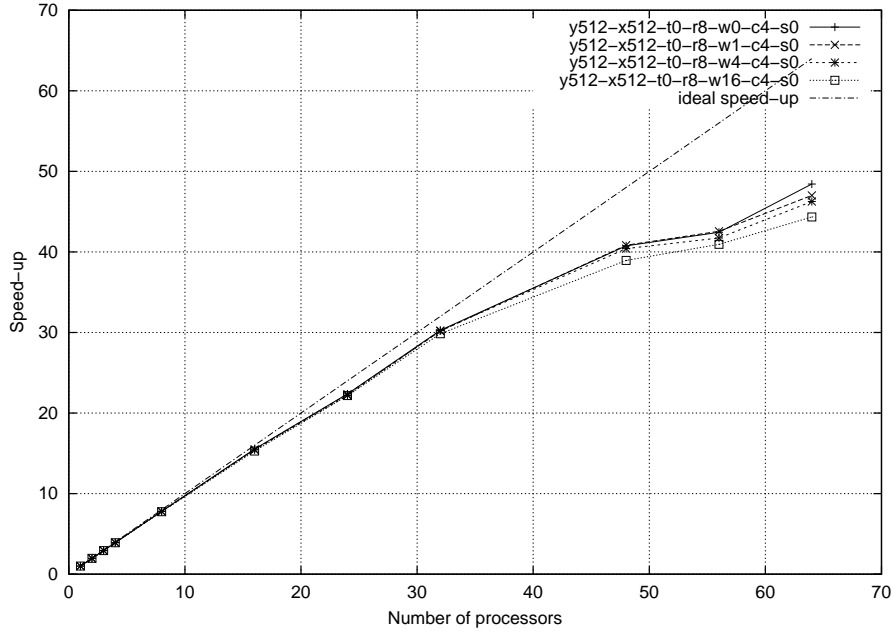


Figure 7.34: Comparative speed-up of the parallel filling with several values of number of obstacles, showing greater speed-ups for fewer obstacles (DSM Origin 2000).

Using the results of our experiments, we can deduce an approximative timing model of our vision algorithm. It increases linearly with the number N of squares of environment, it is twice on 8-connectivity compared to 4-connectivity, it is reduced by a factor of 2 when the symmetry is taken into account ($s = 1$ in symmetric case and $s = 2$ in non symmetric case) and it increases with about the square of the vision radius. Slightly, it is also affected by the type of environment and the number of obstacles. For the parallel filling part, the following formula gathers these results (α is a constant):

$$\text{time}_{\text{filling}}^{\text{seq}} = \alpha \frac{c}{8} \frac{s}{2} N r_v^2$$

This formula agrees with our expectations, since it is similar to the theoretical formula presented in section 7.9.2.

7.10.7 Overall performance of the vision algorithm

Summing up the results obtained, we can give the performance of the whole vision algorithm. The vision algorithm consists of a sequential memory allocation part, a parallel memory allocation part and a filling part. The sequential allocation part has a negligible time. The memory allocation part is difficult to parallelise, while the filling part has good parallel performance.

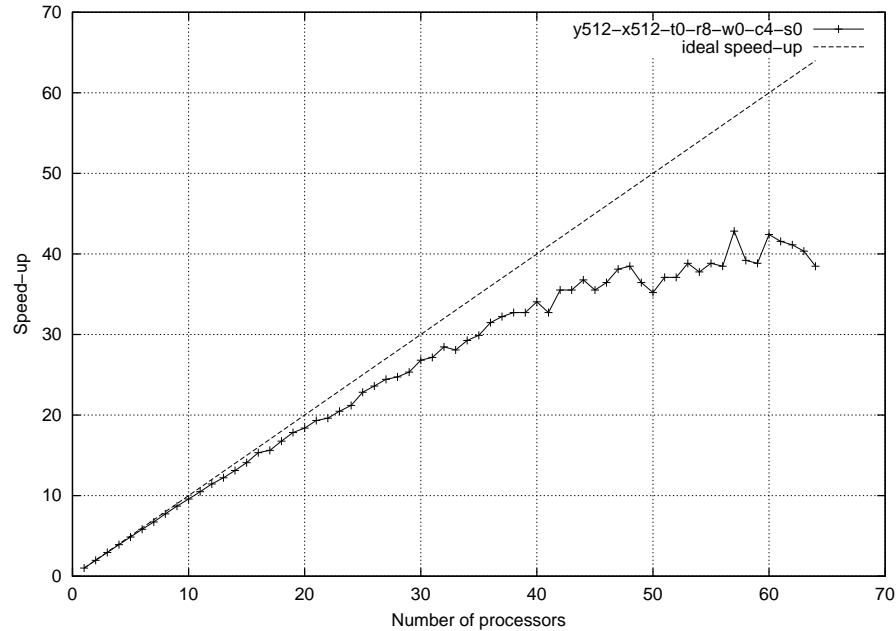


Figure 7.35: Speed-up of the whole vision algorithm, showing good parallel performance (DSM Origin 2000).

Figure 7.35 presents the overall speed-up of the vision algorithm (i.e. the sum of the three functions). Its performance is very good, though it seems that the speed-up stops growing after 60 processors, with a maximum value of about 43 for 57 processors. The speed-up is slightly smaller than for the parallel filling operation because on high number of processors, the execution time of the scalable parallel filling (about 1 second) becomes comparable to the execution time of the non scalable parallel allocation (about 100ms).

The efficiency curve of the whole algorithm is presented in figure 7.36. We notice the zigzag shape given by load unbalancing, but the efficiency remains close to 0.9 for 32 processors.

7.11 Conclusions

The vision algorithm is used in the percepts of agents in our multi-agent simulators. Compared to wave propagation algorithms, it needs a lot of memory and is executed only once during the simulation. The vision algorithm consists of computing for each square its visibility field, based on a constant vision radius. Because of obstacles, a special and innovative algorithm, a supercover algorithm, has been created to allow the computing of visibility fields. Some properties of the vision relation allow to enhance the execution performance of our vision algorithm. These optimisations appear not only

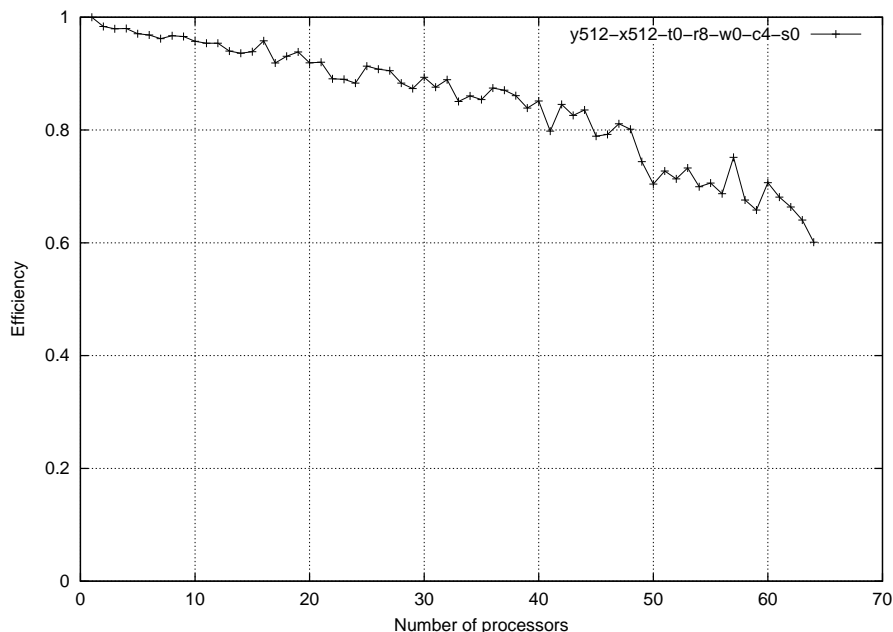


Figure 7.36: Efficiency of the whole vision algorithm, showing good parallel performance and the influence of load unbalancing (DSM Origin 2000).

in sequential, but also in parallel.

The vision algorithm has two parts: allocation and filling. The experiments give a small execution time for the allocation part. For the filling part, the computing of each square is independent of each other, and we have expected a very good parallel performance. Our measurements have agreed with our expectations. An interesting zigzag efficiency curve has been obtained, and we have proven that it is due to load unbalancing. Despite the good scalability of the filling part, for environments not very large and high number of processors the overall speed-up of the vision algorithm is made worse by the parallel allocation.

For yet better execution performance, inexact vision algorithms can be imagined. They would allow to build visibility fields with minor errors, without affecting agent's quality from SMA point of view. Nevertheless, in our practical simulations the vision algorithm is executed only once per simulation, and the parallel execution times are sufficiently small (e.g. 1 second).

In conclusion, the in-depth practical performance measurements we have done agree with our expected theoretical results. Summing up, as the speed-ups of the whole vision algorithm we have obtained for 32 processors are greater than 28 (with an efficiency of almost 0.9), and the parallel execution time can reach 1 second in many cases, we consider that the vision algorithm has a satisfactory parallel performance for our simulations.

Chapter 8

Parallel algorithmic of potential field propagation and perception

The first percept of agents, simulating the vision, has already been presented in the previous chapter. The other percept, presented in this chapter, simulates the odour by potential gradient (indirect perception). By following squares with increasing potential, the agents find the way to the resources found in the environment (figure 8.1 on the next page). A model simulating very well the spread of potential is the *wave propagation model*, which bypasses the obstacles. This chapter presents several wave propagation algorithms. (We have also written a brief paper [28] dedicated to these algorithms.)

8.1 Properties of the potential gradient

Compared to the vision percept, the odour (which will be called *potential* in the following) has two main characteristics:

1. The strength of the potential decreases with the distance from its source.
2. It copes with obstacles by getting around them, as shown below.

In our multi-agent simulation model, only resources can propagate potentials. Also, in our model, the position of resources is fixed during the simulation. But the value of the potential they propagate can change during the simulation, so their potential field changes. During a cycle of simulation, only some of the resources change their potential. In order to have always exact potential fields, update of the potential field of these resources needs to be done every time when the potential fields change. Compared to

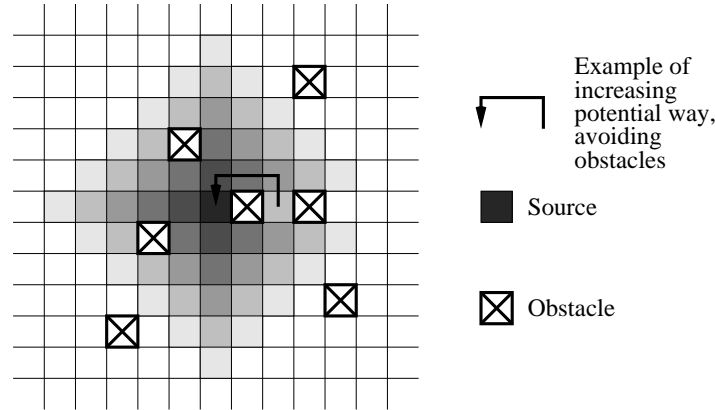


Figure 8.1: Illustration of the potential field of a resource.

the vision algorithm, which is executed only once during the simulation, the wave propagation is thus characterised by frequent executions, hence high execution times. This may become the most costly operation during the simulation [31].

Two functions are involved in resources:

1. The spreading of the potential in the environment is a function of the distance to the resource. This is a decreasing function. Several functions can be used, such as linearly decreasing or square root. Our tool uses a linearly decreasing function $\text{pot}(S) = \text{pot}(S_r) - d$, where S is the square where the potential is computed, S_r is the square containing the resource, and d is the distance between the resource in S_r and the square S . This function is used for squares at distance inferior to $\text{pot}(S_r)$.
2. In our model, the potential given by a resource in its own square is a function of its load. The load is simply the quantity of objects found in the resource. The function $\text{load} \rightarrow \text{potential}$ is private to each resource and is given by the user. As this function depends on the user and on the load, the potential evolution can be nonmonotonic in time. An example is a resource where some agents put objects inside, while others take them off.

Several resources can exist in an environment. It is thus possible that several potential fields overlap. The potential of such squares is described by another function, based on the potentials given by each field. We have used the \max function, as described later.

Each resource spreads potential of one type¹ (and only one). It is also possible to have different potential types coming from different resources.

¹More precisely, identifier.

This simulates then several types of potential, corresponding for example to several types of objects. The potentials of different type spread on a same square without mixing themselves.

Another interesting property of our multi-agent simulation model is that during a cycle of simulation, as the potential is perceived only by agents, the potential is needed only in certain squares, namely those that are perceived by agents. This can be used to create optimised algorithms, which calculate the potential of only the needed squares.

8.2 Definition of the potential field topology

The topology of the potential field involves the distance used and is identical to the vision percept. Section 7.2 details the distance used and the topology field.

8.3 Difficulties introduced by obstacles

In an environment without obstacles, the propagation of the potential would be very simple. In fact, the distance between any two squares, defined as the minimum number of movements needed by an agent to go from one square to the other, would be $|dx| + |dy|$. An iterative algorithm, efficient in terms of execution, would allow to sweep all the squares influenced by a resource.

In the presence of obstacles, the formula $|dx| + |dy|$ is no longer appropriate as distance between two squares (see figure 8.1 on the facing page). Because the obstacles prevent the movement of the agents, the number of movements needed by agents depends on the position of obstacles. Other models are thus necessary to correctly construct decreasing potential fields that avoid obstacles. One of these models is the wave propagation one, presented below.

8.4 Wave propagation model

In this section we present the two hypotheses we have taken in our model:

1. The potential decreases linearly from the resource.
2. If several potentials overlap on a same square, the square receives the maximum potential among them.

These hypotheses are not general, and therefore cannot be used in some real models such as electrostatic fields, where the potential decreases using a square root function. Nevertheless, they are *sufficiently good* for our multi-agent systems (since the agents can find the way to resources) and they are *sufficiently simple* to be fast in terms of execution speed (compared to square

root function for example). Some of the methods we present still work with other hypotheses too.

8.4.1 Wave propagation of one resource

The wave propagation model, based on the Huygens' principle [51], is a propagation model that allows the *exact* building of the potential field of a resource even in the presence of obstacles. This field is perceived locally by agents in order to arrive to a resource using a collision-free path (the collision concerns only obstacles, not agents). As we will see in section 8.9 the propagation can be done also inexactly, in order to optimise memory usage and/or execution time of the algorithm.

The key points of the model, allowing it to be similar to the gradient propagation of potential, are that the potential is spread *decreasingly* by moving off the resources, while *bypassing* obstacles. The algorithm below describes it more precisely, with p the potential function and S_r the resource square:

```

for  $d = 0$  to  $\text{pot}(S_r)$  do
  for all square  $S$  at distance  $d$  from the resource do
     $\text{pot}(S) \leftarrow \text{pot}(S_r) - d$ 
  end for
end for

```

In this algorithm we have used a linearly decreasing function, but other decreasing functions, such as square root, can be used as well.

Finding all the squares at distance d can be done recursively: they are all the *unprocessed* neighbours of the squares at distance $d - 1$ (figure 8.2 on the next page). Finding out if a square has been processed or not can be done by firstly initialising the potential of all squares to zero and, during the propagation, checking for each current square if it already contains a higher potential than the one being applied. We will use this technique in most of the sequential implementations described below. Thus the model is given by the following steps (figure 8.2 on the facing page):

1. Put p into the resource square (p is the potential given by the resource)
2. Put $p - 1$ into the neighbours of the resource
3. Put $p - 2$ into the unprocessed neighbours of the neighbours of the resource
4. Put $p - 3$ into the unprocessed neighbours of the neighbours of the neighbours of the resource
5. ... and so on...

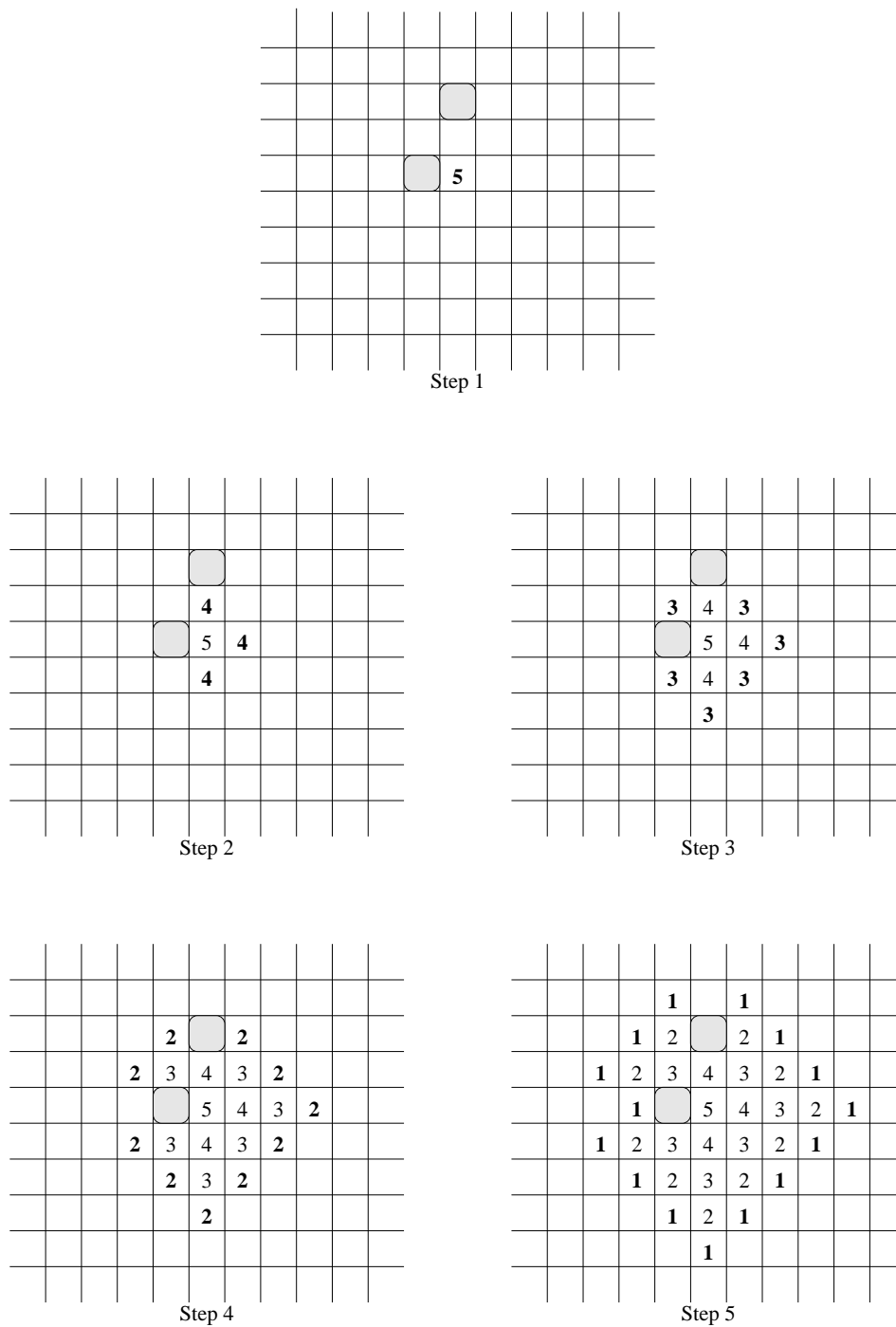


Figure 8.2: Steps used in the wave propagation model. Example with a resource of potential 5.

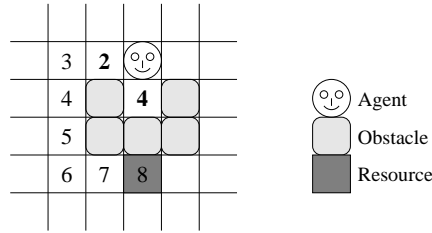


Figure 8.3: Example of hypothetical false local extreme: the agent is misled towards the neighbour bottom square (which does not contain a resource), since its potential (4) is greater than the potential of the left square (2).

A useful property of the model is that, for any square, the difference of potential between that square and its neighbours is either -1 (further from the resource), or 1 (nearer to the resource).

This model does not generate false local extremes (i.e. squares with local extreme potential where there is no resource). Such an example is presented in figure 8.3.

8.4.2 Overlapping of potential fields

A special case appears when a square is influenced by several resources, i.e. when it belongs to several potential fields. Several mixing functions can be used, as presented in section 3.4.3. We chose to use the **maximum** of the values imposed by fields. This function gives two properties to the final potential field:

1. The difference of potential between a square and any of its neighbours can be only -1 , 0 , or 1 .
2. As for one-resource propagation, it does not introduce false local extremes.

An example of overlapping potential fields is given in figure 8.7 on page 174.

8.5 Wave propagation implementation overview

The wave propagation model can be implemented in several ways. We will present in the following several possible implementations, though not all of them have been implemented. For each method, we present its specificities, particularly its strengths and its weaknesses, and compare it with the others. Firstly, we describe the sequential methods, which are sufficient, but not always time-efficient, to generate the potential field of all resources on the complete environment. We have divided them in the following categories:

1. Recursive methods, which propagate individually each resource.

2. Iterative methods, which sweep several times the whole environment.
3. Distance-based methods, which store in squares the distance to resources.
4. Other methods, consisting of a multi-grid method which repetitively decomposes the environment in regions and put the same potential in each of them.

Next, we will present the parallelisation methods, which work *only* in conjunction with sequential methods, and aim the decreasing of their execution time. We have divided them in two categories:

1. Based on domain decomposition, which decompose the environment.
2. Based on data decomposition, which decompose the resources in several groups. Some advantages of the domain decomposition can also be acquired if each group contains all the resources in a domain of environment and only them.

As written in the beginning of the chapter, there may be several types of resources in the environment, and each type needs to be completely propagated during each cycle of simulation. Based on this property, some methods can be optimised by decreasing the memory used, or better exploiting the cache. Also, the propagation is done repetitively, each cycle, and this property can be used to optimise some methods; an example of such optimisation is spreading the potential of only some of the resources.

During the simulation, an important optimisation applies to all these methods: The propagation of a type is done only if the potential fields have changed, i.e. there are resources of that type which have changed their potential since the last propagation of that type. This optimisation has not been implemented in our tool.

During the presentations of methods, the following notations will be used:

- N , the number of squares in the environment.
- p , the potential of a resource (i.e. the potential in its own square).
- R , the number of resources in the environment.
- P , the number of processors.

Note: For the computing of complexities we will use an “extended” form of the classical order class O . Its argument, even if it contains several variables, represents the *dominant* term of the expression (we assume that all the variables are greater than 1; in our case there will always be a dominant term). For example, for the expression $(2p^2 + 2p + 1)R/N$ we will also use $O(p^2R/N)$.

8.6 Sequential methods

8.6.1 Recursive methods

These methods create the potential fields by propagating successively each resource. The propagation of each resource is done recursively.

Recursive with breadth-first propagation

The obvious method to generate the potential fields, implemented also in our tool, is based on the recursivity mechanism. We use the breadth-first recursivity.

Explanation. As the programming language used by our library (C language) does not offer a built-in breadth-first search mechanism, it was simulated with a queue that stores the elements used in recursion. Thus the queue simulates also the recursion. The propagation is given by the following algorithm:

Require: the potential of all the squares is cleared to 0

Ensure: all the resources propagate completely their potential

```

for all square  $S_r$  containing a resource do
  propagate-square ( $S_r$ , pot( $R$ ))
end for

```

```

procedure propagate-square (square  $S$ , potential  $p$ )

```

```

  pot( $S$ )  $\leftarrow p$ 

```

```

  queue-add ( $S$ )

```

```

   $p \leftarrow p - 1$ 

```

```

  queue-add (DELIM)

```

```

loop

```

```

   $S \leftarrow$  queue-remove

```

```

  if  $S =$  DELIM then

```

```

     $p \leftarrow p - 1$ 

```

```

    if  $p = 0$  or queue is empty then

```

```

      end procedure

```

```

    end if

```

```

    queue-add (DELIM)

```

```

     $S_1 \leftarrow$  queue-remove

```

```

  end if

```

```

  for all  $S$  neighbour of  $S_1$  do

```

```

    if  $S$  not obstacle and pot( $S$ )  $< p$  then

```

```

      pot( $S$ )  $\leftarrow p$ 

```

```

      queue-add ( $S$ )

```

```

    end if

```

```

    end for
  end loop
end procedure

```

DELIM is a special element which delimits groups of consecutive potential.

Discussion. Because this method uses the breadth-first search, each square is modified only once. As such, the number of updates of squares is equal to the number of squares in the potential field. In an environment without obstacles, its complexity is then $O(p^2)$, where p is the potential in the resource square. In an environment with R resources, considering that no overlapping appears among the potential fields, the execution time is then $O(p^2R)$.

Because each square stores one potential value, the memory complexity is $O(N)$, where N is the number of squares in the environment. The length of the queue (the maximum number of elements stored simultaneously) does not need to appear in this formula, since it does not depend on N . Indeed, because the queue stores simultaneously only elements found at current distance d , and their neighbours (found at distance $d + 1$), the length of the queue is less than twice the maximum number of elements to a given distance. The maximum number of elements at a given distance appears at distance p (p is the maximum potential of a resource square), giving $4p$ elements. This means the maximum number of elements stored in the queue is $2 \times 4p = 8p$.

A useful optimisation exists, based on the fact that the propagation is done every cycle. It appears if, since the propagation during the previous cycle, all the modified resources have *grown* their potential. In this case, the potential field of any modified resource increases. As we have used the max function when fields overlap, the propagation of only the modified resources is sufficient to correctly propagate all the potentials in environment. In this case, the potential of the environment is not cleared to 0 anymore, and the old potentials will be used.

Finally, in case of several types of resources, it is more efficient, for cache reasons, to propagate entirely each type before propagating another type.

Recursive with depth-first propagation

We are interested to know the performance of depth-first propagation too. This method uses the depth-first recursivity.

Explanation. In this method, the propagation starts from each resource and is recursively spread on the environment while decreasing the value of the current potential. The algorithm below presents this method:

Require: the potential of all the squares is cleared to 0

Ensure: all the resources propagate completely their potential

```

for all square  $S$  containing a resource do
  propagate-square ( $S$ , pot( $S$ ))
end for

procedure propagate-square (square  $S$ , potential  $p$ )
if pot( $S$ ) <  $p$  then { $S$  has not the correct value, so it needs processing}
  pot( $S$ )  $\leftarrow p$ 
  if  $p > 1$  then
    for all neighbour  $S_1$  of  $S$  do {propagate to all its neighbours}
      if  $S$  is not an obstacle then
        propagate-square ( $S_1$ ,  $p - 1$ )
      end if
    end for
  end if
end if

```

Discussion. The key point of this method is that the call to the **propagate-square** recursive function gives a *depth-first* propagation. The depth-first recursion is simple to implement, since it is automatically provided by modern programming languages. However, this is an algorithm where the depth-first search is not efficient. The reason is that some squares are giving increasing potential, so their potential is successively updated (this appears only when the potential of resource is greater or equal to 4). Figure 8.4 on the next page presents an example in the case where the neighbours are taken in N , E , S , W order (potential of resource is 4). The potential of the square at right to the resource receives two updates: It is firstly set to 1, then to its correct value 3. Generally, greater the potential of the resource is, more squares are updated, and more updates are needed in average for each square.

It is worthwhile to notice that an update of the potential of a square generates also a recursive update of all its neighbours. This affects drastically the performance. The previous method has execution time complexity of $O(p^2)$. As the number of updates is greater in this method, its time complexity is at least $O(p^2)$, where p is the potential in the resource square.

Like the previous method, each square stores its own potential, so N memory elements are totally needed (N is the number of squares of the environment). The maximum level of recursivity is p . This means a memory requirement of αp , where α is a constant representing the number of elements involved in a recursive call. Then the total memory requirements are $N + \alpha p$. This gives $O(N)$.

Like the previous method, in the particular case when all the resources grow their potential between two consecutive cycles of simulation, the propagation can be done by propagating only the modified resources (see previous method for more information). Also, in case of several types of resources,

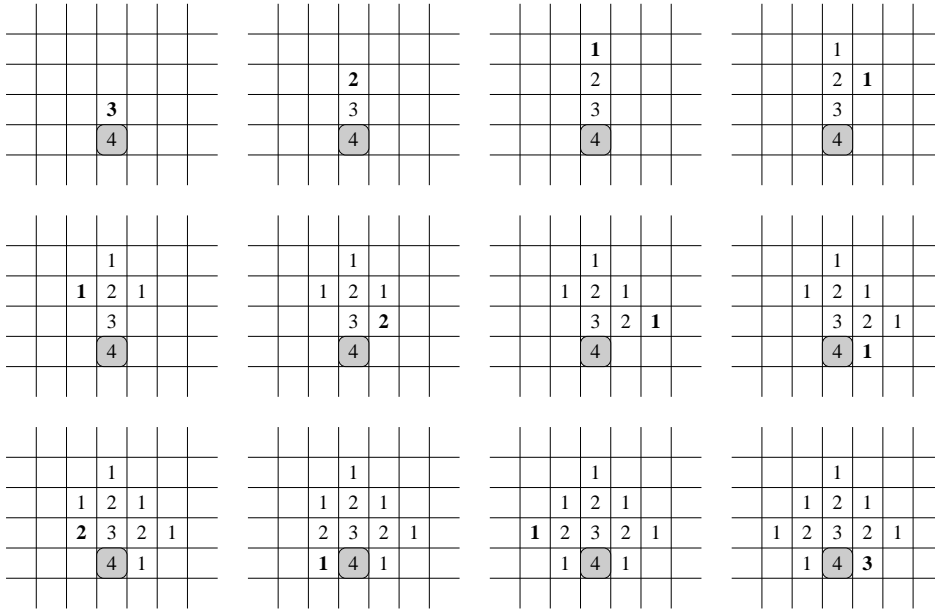


Figure 8.4: Depth-first search has the inconvenient to give some squares increasing potential (the square at the right of the resource in this example), which means its potential is updated several times during the propagation (twice in this example).

it is more efficient, for cache reasons, to propagate entirely each type before propagating another type.

8.6.2 Iterative methods

These methods sweep several times the whole environment, updating each square when necessary. The execution time complexity is therefore higher than recursive methods. However, as their processing is simple, they are interesting in some cases.

Iterative with fixed potential

This method is presented by Bouton [13], who has worked in our team.

Explanation. This method works by firstly putting the potential of every resource in its square. Then, during the first iteration, all the environment is swept in order to find all the squares containing the greatest potential p . Every time a square with potential p is found, all its neighbours having potential less than $p - 1$ are given a potential of $p - 1$. During the second iteration, all the squares with potential $p - 1$ are found and their neighbours

with less potential are given potential $p - 2$. The iterations continue until the potential 1 is reached. At this step, the propagation is completely finished on all the environment. It is worthwhile to notice that having no squares updated during a step is not a sufficient condition for the end of the propagation. It is also necessary to not have resources whose potential has not yet been propagated, specifically resources with potential lower than the last potential propagated.

The algorithm below describes this method more rigorously:

Require: the potential of all the squares is cleared to 0
Ensure: all the resources propagate completely their potential
for all square S_r containing a resource **do**
 $\text{pot}(S_r) = \text{potential given by resource } r$
end for

for $p = \max \text{pot}(S_r)$ downto 2 **do**
 for all square S in the environment **do**
 if $\text{pot}(S) = p$ **then**
 for all S_1 neighbour of S **do**
 if $\text{pot}(S_1) < p - 1$ **then**
 $\text{pot}(S_1) \leftarrow p - 1$
 end if
 end for
 end if
 end for
end for

Discussion. The drawback of this method is that during each step all the environment is swept, which leads to a lot of useless processed squares. In an environment without obstacles, this gives an execution time $T \approx t_{\text{square}} N p$, where t_{square} is the average time needed to process a square, N is the number of squares of the environment, and p is the maximum potential in a resource square. This gives an execution time complexity of $O(Np)$. Note that this number does not depend on the number of resources in the environment.

The interest of this method comes from the fact that the processing of each square is very simple: compare its potential with the maximum potential p of its neighbours, and update it with $p - 1$ if necessary. As its time complexity does not depend on the number of resources, this method achieves good performance in the worst cases of other methods (see section 8.8 dedicated to performance measurements).

Like the recursive methods, since each square stores its potential, the memory requirements are in $O(N)$.

Optimisations. This method, not implemented in our tool, can be optimised by reducing the part of environment swept to the useful part (a rectangle for simplicity). This optimisation is particularly useful when it is combined with the private environments parallelisation method (as shown in section 8.7). As an example, if there is only one resource in environment, the sweep can be done only on a rectangle containing entirely its potential field. In the general case, the rectangle will contain the potential fields of all the resources. It can be calculated during the first part of the algorithm, by adding and subtracting from each resource coordinates its potential:

```

 $y_{top} \leftarrow \infty, x_{top} \leftarrow \infty$  {rectangle initialisation}
 $y_{bottom} \leftarrow 0, x_{bottom} \leftarrow 0$  {suppose  $y$  increases from top to bottom}
for all square  $S_r$  containing a resource do
   $\text{pot}(S_r) \leftarrow$  potential given by resource in  $S_r$ 
  if  $y_{top} > y_S - \text{pot}(S_r)$  then
     $y_{top} \leftarrow y_S - \text{pot}(S_r)$ 
  end if
  {and so on for  $x_{top}, y_{bottom},$  and  $x_{bottom}$ }
end for

```

In the case where several types of resources exist, the performance can be influenced by the cache. If the cache is sufficiently large to contain all the data needed to process one type, but not all the types, then processing completely each type before processing another type allows to reuse the cache data, thus increasing performance. If the data size is greater than the cache size, then, depending on the data structures of the program, it can be more efficient to process for example each line of environment for all the types before processing the next line.

During the simulation, if between two consecutive cycles of simulation all the modified resources have grown their potential, then the squares with potential higher than the new potential of *modified* resources do not change. Therefore, the sweep needs not to start from the maximum potential of all resources; it can start on the maximum potential of only the modified resources. The potential of the environment needs not be cleared to 0 in this case, but the old potentials will be used.

Iterative with variable potential

A similar method have been implemented in our tool, with similar advantages and disadvantages.

Explanation. The difference between this method and the previous one is that, during each step, instead of processing only squares with a given potential p , it compares each square with its neighbours, updating it if necessary. The algorithm is the following:

Require: the potential of all the squares is cleared to 0
Ensure: all the resources propagate completely their potential
for all square S_r containing a resource **do**
 $\text{pot}(S_r) = \text{potential given by resource in } S_r$
end for

repeat
 modified \leftarrow false {has the environment been modified?}
 for all square S in the environment **do**
 for all S_1 neighbour of S **do**
 if $\text{pot}(S_1) - 1 > \text{pot}(S)$ **then**
 modified \leftarrow true
 $\text{pot}(S) \leftarrow \text{pot}(S_1) - 1$
 end if
 end for
 end for
until not modified

Discussion. This method presents the same theoretical complexities as the previous iterative method. The memory complexity is in $O(N)$, and the execution time complexity is in $O(Np)$. (Note that this number does not depend on the number of resources in the environment.)

Optimisations. All the optimisations presented in the previous iterative method apply to this method. More information is given in iterative fixed method.

An important optimisation specific to this method, implemented in our tool, is that in an environment without obstacles, a top-down sweep (each line processed from left to right) fills the south-east quarter of the potential field of a resource, plus nearby squares (figure 8.5(a) on the facing page). A following bottom-up sweep completes the filling of the potential field (figure 8.5(b)). During the third sweep no square changes its potential, so the algorithm ends. The number of sweeps is influenced by the number of obstacles and the overlapping quantity of potential fields.

8.6.3 Distance-storing methods

These methods are characterised by the fact that each square stores not potentials, but *distances* to resources, and the potential is computed based on this distance. This is very important, because the potential of *only* the squares perceived by agents needs to be calculated.

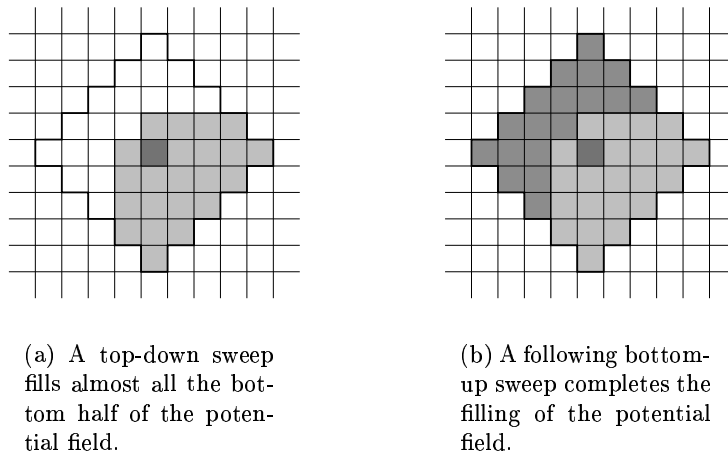


Figure 8.5: In an environment without obstacles, two sweeps completely propagate the potential of a resource (4-connectivity in this example).

Distance-storing of all influent resources

The interest of this method is that it does not need propagation, so it seems to work tremendously faster than all the previous methods. The trade-off is that it has high memory requirements, and it was not implemented in our library.

Explanation. Based on the fact that the place of resources is fixed during the simulation, a method which uses this fixed distance between squares and resources can be used. In order to identify resources, we associate a unique number (its identifier) to each resource. Each square can store the identifier of each of the resources which can influence its potential during the simulation, and the distance to it. When the potential of a square needs to be known, the influence of every resource on it can be simply calculated by using the distances it stores and the actual potential of the resource concerned.

This method has two phases. The initialisation phase fills each square with the identifier and the distance of the resources that can influence it:

Ensure: all squares store the identifier of all resources that can influence it

$p \leftarrow$ the maximum potential field radius

for all square S in the environment **do**

for all resource r at distance $\leq p$ from square S **do**

 store in square S the identifier of r and the distance to it

end for

end for

As noticed, the maximum potential field radius needs to be known in advance, in order to know the resources which can influence it.

The use phase consists of finding the potential of a given square, based on the resources identifiers it stores:

Ensure: know the value of the potential of square S
 $\text{pot}(S) \leftarrow 0$ {clear its potential}
for all resource identifier r stored in square S **do**
 $p \leftarrow$ the potential given by r in square S , based on the distance d to it
if $\text{pot}(S) < p$ **then**
 $\text{pot}(S) \leftarrow p$
end if
end for

Discussion. The potential can be calculated for all squares, as for the previous methods. Additionally, this method allows to calculate the potential “on demand” (only for the squares needed), which can be very fast. As an example, if agents are found on 1% of the squares, then, for 4-connectivity, the potential of maximum 5% (the square itself and its four neighbours) of the squares is calculated. Also, it is possible to calculate the potential only for squares that are influenced by resources having changed their potential.

In the following we calculate an approximate average of the number of resources influencing a square (N is the number of squares in environment, A is the number of agents in environment, R is the number of resources, and p is the average potential of a resource). Firstly, in average, in environment there are R/N resources per square. Secondly, on a 4-connectivity environment the potential field of each resource contains an average of $2p^2 + 2p + 1$ squares (formula 7.1 on page 119). Thus, the average number of resources influencing each square is given by:

$$R_s = (2p^2 + 2p + 1)R/N \quad (8.1)$$

As memory requirements, the whole environment contains N squares, thus it needs $R_s N = (2p^2 + 2p + 1)R$ identifiers, along with the distance to each of the resources. This gives a complexity of $O(p^2 R)$. As an example, for $N = 10^6$ squares, $R = 2\% \times N$ resources in environment, and an average potential $p = 10$, the number of identifiers (and the distance to each of them) that need to be stored is 4×10^6 .

We are interested also in the time complexity of the algorithm. Since the first stage is executed only once, in the beginning of simulation, we take into account only the second stage. The execution time depends on the number of agents A . Consider the 4-connectivity case (the 8-connectivity case is similar, with respect to a constant). Suppose each agent “smells” (read potential of) its own square and the 4 squares in its neighbourhood. Then the number of squares whose potential needs to be known is $5A$. Each square is influenced by an average of R_s squares (formula 8.1 above), so the time needed to calculate the potential of all the needed squares is $5AR_s t_s =$

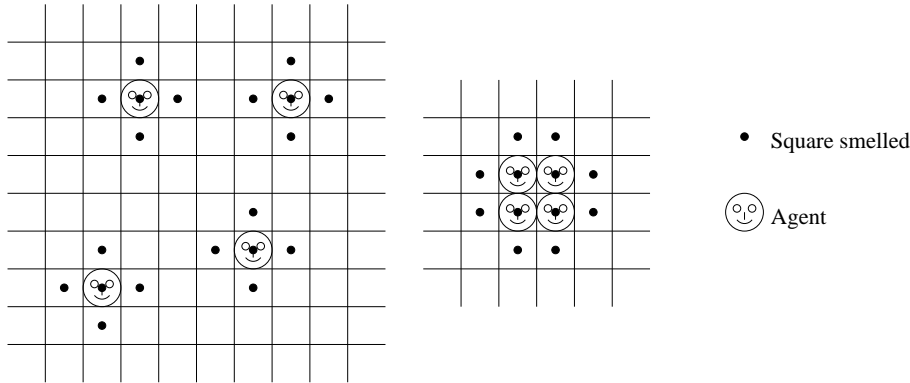


Figure 8.6: On a high density of agents, some squares can be perceived by several agents, which reduces the number of squares whose potential is needed: in this example, the number of squares “smelled” by 4 agents is 20 at left, and only 12 at right.

$5(2p^2 + 2p + 1)ARt_s/N$, where t_s is the unit time to calculate the potential influenced by a resource into a square.

In fact, the exact number of squares whose potential needs to be known is between about A and $5A$. The first value appears when agents are near to each other. In this case, a same square is smelt by several agents, so it needs to be counted only once. Figure 8.6 presents two densities of agents: one with agents far away from each other, and another with agents sufficiently near to each other that some squares be perceived by several agents. Nevertheless, the execution time complexity does not change, and it is in $O(p^2AR/N)$.

As we will present in the section 8.7, dedicated to parallelisation methods, this method has the advantage that the computes of each square are independent, so no special method of avoiding parallelisation conflicts is needed.

Distance-storing of the most influent resource

This method stores only one resource, the most influent one, per square. It is a fast method, and has less memory requirements than the previous distance-storing method. It was not implemented in our library, because of time constraints.

Explanation. This method is based on the previous method. However, instead of storing in each square the identifier of *all* the resources which influence it, it stores the identifier of only the most influent resource. The most influent resource is that which gives the potential of the square, i.e. it gives the maximum potential in the case of overlapping fields. By storing in each square information of only the most influent resource, this method thus

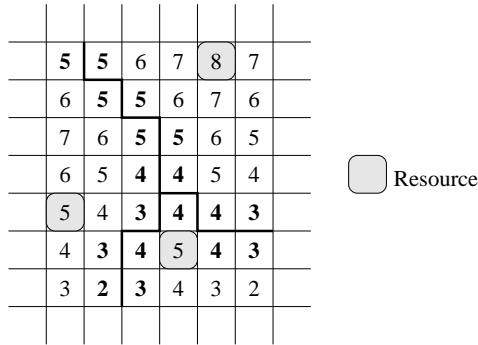


Figure 8.7: Each resource has its own influence zone in the environment. The frontiers of zones (in a 4-connectivity environment) contain bold digits.

divides the environment in zones of influence, as shown in figure 8.7.

This method is based on the fact that the propagation is done repetitively each cycle of simulation. In fact, a particularity of this method is that the potential field is not cleared, but updated since the last propagation. Each square contains the resource identifier R , the distance d to the resource that influences it, and a boolean value f allowing to know if it is on the frontier of a zone or not. As any square contains the distance to the resource, and not the potential, the field updates need to be done only on the neighbourhood of the resources which have changed their potential, and not on all the environment. The difficulty is then to recognise the frontiers, which need to be updated when its resource changes. Two variants can be imagined:

1. Store in each resource the squares which belong to it. This seems to be difficult, because the frontier between resources has variable length and changes during the simulation.
2. Store in each square the identifier of the resource it belongs to. This seems to be more appropriate, and will be used in the following.

Two phases can be noticed in this method. During the first phase, occurring at the beginning of the simulation, the parameters of each square (R , d , and f) are initialised. During the second phase, occurring during the simulation, the zones of influence and the parameters of squares are updated.

Discussion. This method needs three elements on each square (R , d , and f). The memory required is thus $3N$ elements, where N is the number of squares in environment, which gives a memory complexity of $O(N)$.

This method has a much more difficult implementation than the previous method. It has the same advantage that, during the second phase, the potential of only the squares needed by agents is computed. However, since

only one resource is stored in each square, it has less memory requirements than the previous method.

8.6.4 Other methods

Multi-grid

We present here the basic principle of another method we have imagined. It has not been implemented, because we have not proven its convergence, and as such we will not take into account in method comparison. This method decomposes the environment in domains and put a same potential value in each square of the domain. Reiterating this process for different domain decompositions, exact or approximate potential fields are hoped to be obtained.

Explanation. This method proposes that any square receives successive potential values, nearer and nearer to its exact value. The environment is decomposed in several domains, and each domain sets all its square at the same potential. This potential is calculated from the potential of the resources contained in the domain. By iterating this operation, and by changing the domain decomposition, the squares would receive potentials more and more exact. This algorithm would be described in the pseudo-code below:

```

set all potentials to 0
set all resources to their potential

repeat
  choose decomposition of environment
  for all domain do
    find all resources  $R_i$  in the domain
     $p_a \leftarrow f(R_i)$  {average potential, based on the resources in the domain}
    put  $p_a$  in each square of the domain
  end for
until no change appears

```

Discussion. The way in which the environment is decomposed is very important to the convergence of potentials. Thus, in order to correctly propagate the potential of any square, *any* two neighbour squares may need to belong to a same domain in one of the steps. Otherwise, the field does not propagate from one square to the other square.

A feature of this method is that the zones without resources would be eliminated from the beginning. Also, the iteration can be stopped before the exact value is reached, thus obtaining an inexact but faster propagation.

Method\Comparison element	Execution time	Memory	
	Complexity	Complexity	Number of elements
Recursive breadth-first	$O(p^2 R)$	$O(N)$	$N + 8p$
Recursive depth-first	$\geq O(p^2 R)$	$O(N)$	$N + \alpha p$
Iterative, fixed potential	$O(Np)$	$O(N)$	N
Iterative, variable potential	$O(Np)$	$O(N)$	N
Distance-storing, all resources	$O(p^2 AR/N)$	$O(p^2 R)$	$2(2p^2 + 2p + 1)R$
Distance-storing, most influent resource	?	$O(N)$	$3N$

Table 8.1: Comparison of wave propagation sequential methods (N is the total number of squares, R is the total number of resources, A is the total number of agents, and p is the strongest potential. Also, α is the memory needed by the system for a recursive function call with a few parameters).

8.6.5 Theoretical comparison of sequential methods

Table 8.1 shows the memory requirements and the execution time of all sequential methods presented above. Several conclusions can be taken:

- The recursive breadth-first method is better than recursive depth-first one.
- The two iterative methods give approximately the same results.
- The memory requirements of best algorithms are in $O(N)$.

It is also worthwhile to notice that for all methods, smaller the potential of resources is, generally smaller the execution time of update is. Therefore, if the potential of resources decreases during simulation, the execution time decreases too. The reasons of this are: for recursive methods, the potential fields are smaller, so there is less processing; for iterative methods, the maximum potential is smaller, so generally there are fewer iterations; and for the method based on all influent resources, squares will be influenced by fewer resources, so there are fewer resources to process.

8.7 Parallelisation methods

In the previous section several sequential methods were presented. They are not sufficient for execution on parallel machines. Indeed, the only impediment is the concurrent access to squares. The goal of the parallelisation methods presented in this section is to cope with this concurrent access. All these methods work in conjunction with sequential methods described above. Some of the combinations between the parallelisation methods and the sequential ones are possible without any modification, others are inefficient, while others are not possible. Their mixing is presented in table 8.2 on the next page and will be described below.

Parallelisation\Sequential method	Recursive	Iterative	Distance-all	Distance-most influent
Fixed domain partitioning	ok	ok	first stage only	?
Changing domain partitioning	ok	ok	first stage only	?
Thread-private environments	ok	ok	inefficient	ok
Mutex-based	ok	?	no	ok

Table 8.2: Mixing between the parallelisation methods and the sequential ones. The methods implemented are in **bold**.

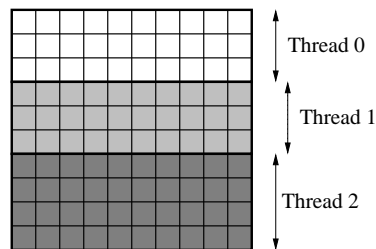


Figure 8.8: Domain partitioning example for 3 threads.

The parallelism can be exploited by three methods (section 4.3.1): data parallelism, flow parallelism, and control parallelism. The flow parallelism is used when there is a continuous flow of data (every square in this case) in input, and each data needs consecutive processing, which is not the case on the sequential propagation algorithms presented above. The control parallelism can be used if we do different processing on different data. As the potential propagation of each resource (the *data*, as shown below) is not fine grain, we prefer to consider that a mixing between data and control parallelism will be used.

The decomposition can be done in two ways: data (resources) decomposition, and domain (environment) decomposition. The first aims mainly a good load-balancing, while the second aims mainly to avoid parallelism conflicts (such as cache conflicts and concurrent access to data).

Four parallelisation methods have been explored, exploiting both data and domain decomposition, which are presented in the following. The first two methods exploit mainly domain decomposition, and the last two methods exploit mainly data decomposition.

8.7.1 Fixed domain decomposition

This is the classical domain decomposition parallelisation (section 4.4), and it was implemented in our library too. The basic principle is that each processor is affected to a different part of the domain. Thus, in our case, the environment is decomposed in several domains (figure 8.8). The number of domains is equal to the number of processors, and each processor is bound to a distinct domain.

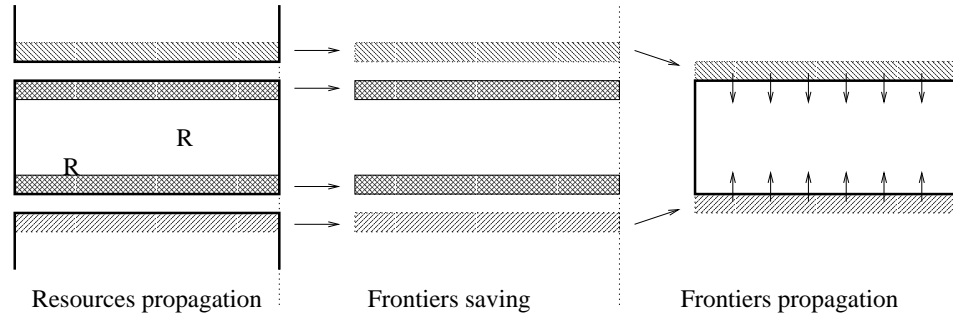


Figure 8.9: For each processor, the propagation is completely described by three stages: resources propagation, frontiers saving (only 2 frontiers in this figure), and frontiers propagation, the last two stages being repeated several times.

The simplest decompositions are rectangle decompositions: (1) horizontal, (2) vertical, and (3) horizontal and vertical decompositions. We took into account only these decompositions, because of their less overhead compared to other specific decompositions. We have used a two-dimensional matrix to store the squares of the environment, and, in the programming language we have used (C language), the matrices are stored in memory line by line [56, page 104]. Therefore, a vertical decomposition can generate false sharing (section 4.2.4), appearing for squares found on the same matrix line and the same cache line but on two distinct domains. Another very important parameter is the length of the frontiers. Generally, greater the length of the frontiers, greater the number of resources whose potential field intersects frontiers, hence greater the number of repropagations (as described below) and greater the execution time.

An ideal decomposition would then decompose the environment such that the load-balancing among the processors be as close as possible, the cache be best used, and the length of the frontiers be as small as possible. In our tool, we have used a horizontal decomposition which is firstly done such that the size of the domains be as close to each other as possible (figure 8.8 on the page before).

The complete propagation is described in three stages (figure 8.9). The first stage propagates the potential of all the resources in each domain *separately*. The second stage saves the frontiers to a memory accessed by neighbour processors. The third stage propagates in each domain separately the potentials of all the frontiers (we will use the term *repropagation* in this case). The second and the third stages are repeatedly executed until no change of potential is done during the third stage. At this moment, the propagation is entirely done in all the environment. These stages are detailed in the following sections.

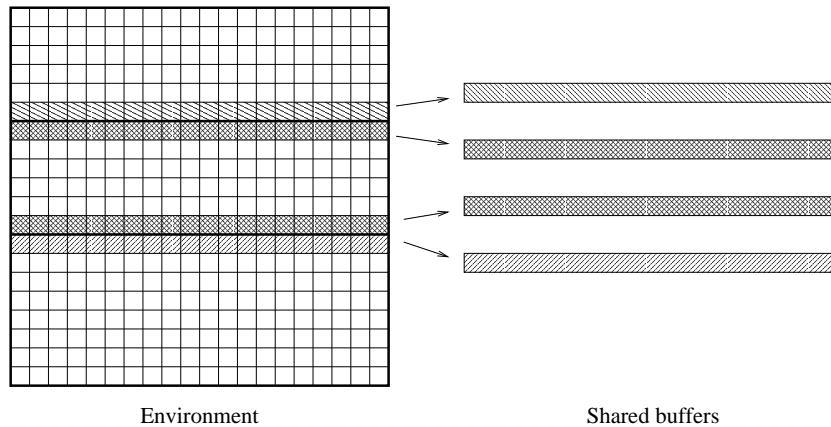


Figure 8.10: Illustration of the second stage, where the frontiers of the three domains are saved into buffers accessed by other processors.

First stage

During the first stage, each processor propagates individually and sequentially all the resources found in its domain, setting the correct potential on each square of its domain. This can be done with any of the sequential methods presented above, as shown in table 8.2 on page 177.

For the method storing distances to all influent resources (section 8.6.3), this stage is sufficient to have the correct potentials in every square. The reason is that, as described in its sequential implementation, any square processing is independent of the other.

Nevertheless, for the other methods the separate propagation on each domain does not assure the correct propagation on all the environment. In fact, squares in a domain can be influenced by resources in other domains, i.e. the potential field of a resource may spread on several domains.

Second stage

The second stage corresponds to frontiers “exchanging”. More precisely, the frontiers are saved into a memory accessed by other processors.

A frontier represents a line or column of environment (or a part of it, if the decomposition is both horizontal and vertical). Each processor contains a memory, called *buffer* in the following, of the same length as its frontiers (figure 8.10). Each processor, once it has finished its first stage (propagation of its domain), saves into its own buffers the frontiers of its domain. No synchronisation point is needed during this second stage, since there is no sharing conflict: each processor reads and writes its own data (domain and buffers).

Third stage

The third stage consists of a propagation of the potential of the frontiers of the neighbours (called repropagation in the following).

Firstly, we can notice that the repropagation needs the potentials of the neighbour frontiers, i.e. reading the buffers of the neighbour processors. Since these buffers have been written at the end of the previous stage, a synchronisation point is mandatory at this moment.

The repropagation is similar to the first stage (propagation), except that the potential is *not* cleared to zero, and the propagation starts from all the points of the frontiers, and not from resources.

Any of the sequential methods described, according to table 8.2 on page 177, can be used in this stage, with minor modifications:

1. For the recursive methods, as described, the propagation starts from resources. Instead, the repropagation variant has to start from all the squares found on frontiers. As the time complexity of the recursive methods is proportional to the number of potential sources (section 8.6.1), in the repropagation case this corresponds to frontier length. In a horizontal domain decomposition the frontier length is equal regardless of the number of processors. We will see later (section 8.8.3) how this property influences the parallel performance of the recursive methods.
2. For the iterative methods, as described, the propagation starts with the maximum potential of resources. The repropagation can be identical. Nevertheless, it is faster to start with the maximum potential of the squares found at frontiers, since only the frontiers have changed their potential.

Number of repropagations

A repropagation is generally not sufficient to complete the propagation of potential on the whole environment. Figure 8.11, left case, presents an example where two repropagations are necessary. During propagation, the obstacles prevent the square S to receive the potential from resource R . A first repropagation allow intermediate squares to receive correct potential values, and only the second repropagation can put the right potential into square S .

Even in an environment without obstacles, several repropagations may need to take place. In fact, it is possible for a square S to update its potential from a square found in another domain, which in turn gets its potential from a resource R found in yet another domain. Such a case can appear in horizontal *and* vertical decomposition in 4-connectivity (figure 8.11, middle case), or when a potential is greater than the length of a domain (figure 8.11, right case).

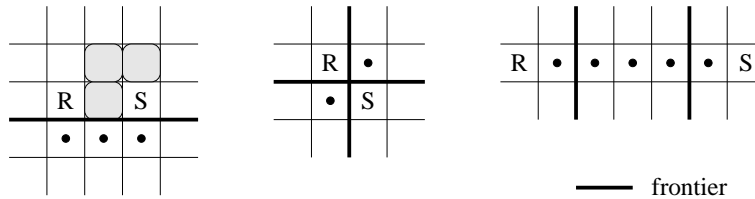


Figure 8.11: Examples where two repropagations are needed for square S to receive the correct potential value from resource R .

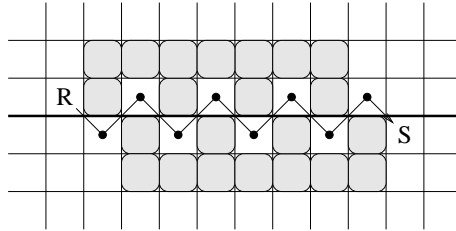


Figure 8.12: In the worst case, in 8-connectivity the number of repropagation stages needed is $p - 1$, where p is the potential of a resource (resource R has potential $p = 10$ in this figure).

The exact number of repropagations (*frontiers* propagations) needed cannot be known in advance (before the complete propagation). In fact, it depends on the potential of resources and on the configuration of the environment (position of obstacles and domain partitioning). Nevertheless, it exists a sufficient value, which, in the worst case, is also necessary. On the one hand, this number β is upper bounded by $p_{\max} - 1$, where p_{\max} is the greatest potential:

$$\beta \leq p_{\max} - 1 \tag{8.2}$$

To prove it, we notice that any square receives its final potential from one and only one resource, let w be the way of decreasing potential from the resource to the square. Its maximum length is $p - 1$, where p is the potential of the resource, and each frontier propagation advances in this way w at least by one square. This proves also that repropagation is a convergent procedure in *finite* time. On the other hand, the value $p - 1$ can also be reached in the worst case, as in figure 8.12, where the potential propagates by decreasing one value per propagation.

More precisely (as we implemented in our tool too), the repropagation is finished when and only when no *frontier* square changes its potential. No change in any frontier square is a sufficient condition: since the frontiers are identical to the previous repropagation, a new repropagation would not change any square.

Synchronisation issues

We are also interested in the number of synchronisations needed by this method. Up to now, the data accesses during all propagations can be described like this:

```

write own domain and frontiers {1st stage}
repeat
  synchronisation point {added to protect the access to the buffers}
  write own buffers {2nd stage}
  synchronisation point {3rd stage}
  read neighbours' buffers {3rd stage}
  write own domain and frontiers {3rd stage}
until no change on any frontier

```

The buffers are the only data accessed by several processors. Each buffer is written by its own processor, in the second stage, and read afterwards by a neighbour processor at the middle of the third stage. The first synchronisation point in the algorithm above protects this concurrent access to buffers (it could have been done also just after reading neighbours' frontiers). This method, implemented also in our tool, gives two synchronisation points for each repropagation.

A more efficient solution can be found, which avoids the use of a second synchronisation. This is provided by the double-buffering technique presented in section 4.4.2. The following procedure uses it:

```

write own frontiers
i = 0 {variable i is private to each processor}
repeat
  write own buffers[i]
  synchronisation point
  read neighbours' buffers[i]
  i = 1 - i {choose the other set of buffers}
  write own frontiers
until no change on any frontier

```

The use of this technique implies higher memory requirements. More precisely, this solution needs an additional memory equal to the memory needed by storing all the frontiers.

Discussion

In this method, the processors have their own domain, which does not change during the simulation. Therefore, if the cache is sufficiently large to contain all the data involved in a domain, then the cache misses will be avoided (except the buffers). Another advantage is that, since the only memory accessed by several processors is the buffers, it can be implemented on distributed memory machines also. The trade-off is that it has decreased performance

in the worst case, when a lot of repropagations are necessary.

When several types of resources exist, this method can be used in two ways:

1. Propagate each type *completely* before propagating another type:

```
for all type  $T_i$  of potential do
  propagate completely type  $T_i$ 
end for
```

Above, for each T_i , $N_{sync}(i)$ synchronisation points are needed. This has the advantage to use better the cache, in case the cache can contain all the data needed to one type, but not the data of several types. The disadvantage is that the total number of synchronisation points is the sum of synchronisations of each type: $N_{sync} = \sum_i N_{sync}(i)$.

2. Do the domain or frontier propagation of *all* the types (up to the synchronisation point) before passing to the following domain or frontier propagation:

```
for all type  $T_i$  of potential do
  do domain propagation of  $T_i$ 
end for
repeat
  synchronisation point
  for all type  $T_i$  of potential which need repropagation do
    do frontier propagation of  $T_i$  (no synchronisation point)
  end for
until no need of repropagation on any type
```

As memory requirements, this needs as many buffers as types. Nevertheless, the execution time is much lower: the number of synchronisation points is the maximum number of synchronisations of each type: $N_{sync} = \max_i N_{sync}(i)$. This leads also to better load balancing among processors.

The timing model depends on many factors and is difficult to model, so we present only the memory model.

Memory model

We do not include in these models the part corresponding to the sequential first stage (resources propagation), since, for any method, it has already been analysed in section 8.6.

We use in this discussion the following parameters: N_c is the number of columns of the environment, and P is the number of processors.

The second stage uses buffers, thus it needs a memory size corresponding to the size of all the frontiers. For a horizontal decomposition, and on a

toroidal environment, each frontier has the same length: N_c squares (see figure 8.10 on page 179). Each processor has two frontiers, hence its buffers need $2N_c$ squares. The third stage may use another set of identical buffers, in the case of double-buffering technique. Therefore, for all the P processors, this parallelisation method adds a memory overhead of

$$\text{mem} = O(PN_c)$$

8.7.2 Changing domain decomposition

The fixed domain decomposition method, as noticed in its description and taken into account by its timing model attempt, needs several repropagations to complete the potential spreading in the environment. The aim of the changing domain decomposition is to reduce the number of repropagations, generally to 1.

Principle

This method is similar to the previous one. It can be divided into two stages: resources propagation, and frontiers propagation. The decomposition analysis and the resources propagation are identical to those of the previous method. Therefore, we take into account in the following only the second stage, and its differences compared to the previous parallelisation method.

The distinctive feature of this method is that the domain decomposition changes when the frontiers are propagated (figure 8.13 on the facing page). The new frontiers are now located at some distance d from the old ones. As the repropagation starts on old frontiers, it is less frequently that new frontiers are modified, so, as detailed below, the number of repropagations is less than the one of the previous method. However, if during the repropagation the new frontiers were changed, during the next repropagation the set of frontiers changes: the new frontiers become the old ones, and the old frontiers become the new ones. This mechanism ensures that during each repropagation the new frontiers are located at some distance from the old ones.

It is worthwhile to notice that this method, based on moving off the frontiers so that the new ones are far from the old ones, works on horizontal decomposition and on vertical decomposition, but not on a horizontal and vertical decomposition. Figure 8.14 on the next page presents the latter case, with the continuous lines for the old frontiers. We can see that the new frontiers, the dotted ones, cannot be drawn without intersecting the old frontiers, so a minimal distance d between any square on old frontier and any square on new frontier cannot be guaranteed.

In figure 8.13, the number of processors used during each propagation is not the same, but differs by 1: P , $P-1$, P , $P-1$, and so on. Nevertheless, the same number of threads can still be used, by giving to the P th processor parts

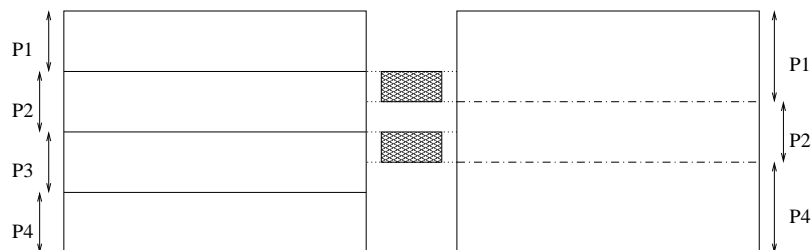


Figure 8.13: The repropagation changes the decomposition in order to move off the frontiers: the continuous lines mark the old frontiers, and the interrupted ones the new frontiers.

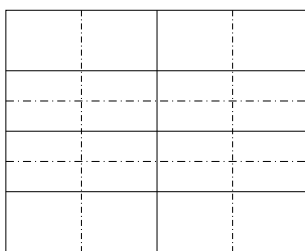


Figure 8.14: In a both horizontal and vertical domain decomposition, the new frontiers (the interrupted ones) cannot be drawn without intersecting the old frontiers (the continuous ones).

of the first and the last domains. In this case, in an edged environment the domain decomposition becomes more difficult (the last domain is composed by two rectangles).

Improvements on number of repropagations

In this section the differences from the previous method concerning the repropagation are described.

In the previous method, a synchronisation point was introduced because of the frontiers saving. This method does not need such a synchronisation.

After the domain propagation and a first frontiers propagation, another frontiers propagation can appear only if there is a resource whose potential is superior to the distance d between old and new frontiers (i.e. a new frontier has changed). When the potential of a square on the old frontier influences a square on the new frontier, their difference of potential is at least d , therefore the potential of a resource decreases at least by d . Therefore, the maximum number of frontier propagations β for this method is then limited according to the following formula (p is the maximum potential of a resource, and d is the minimum of all distances between old and new frontiers of any processor):

$$\beta \leq \left\lfloor \frac{p}{d} \right\rfloor \quad (8.3)$$

This formula insures that the number of repropagations in the worst case is smaller in this method than in the previous method (where it is equal to $p - 1$, see equation 8.2 on page 181). As an example, this method is not impacted by the case presented in figure 8.12 on page 181.

Drawbacks on square misses

In this method, the decreasing of the number of propagations is provided by a domain decomposition that changes. A first consequence of this technique is that it leads to cache misses. They correspond to the squares which change processor (called *square misses* in the following), and they appear during the execution of each stage (first or the second). We calculate in the following a minimum value for the number of square misses. Each processor different that the first and the last is involved in square misses. Therefore, the number of blocks with square misses (the filled squares in figure 8.13 on the preceding page) is at least $P - 2$, where P is the number of processors used. If the distance between any old and new analogous frontiers is d , then the number of square misses for these processors has $N_c d (P - 2)$ squares, where N_c is the number of columns. But all the distances d are equal (as presented in the principle of the method), hence d is half an old domain, i.e. $d = N_l / (2P)$, where N_l is the number of lines. It results that the number of square misses is at least:

$$N_s = N \frac{P - 2}{2P} = O(N)$$

where $N = N_l N_c$ is the number of squares in the environment. We notice that this number of square misses tends to $N/2$. A second consequence of this technique is that, since the number of squares misses is high, an implementation on purely distributed machines cannot be efficient.

This parallelisation method does not add any memory overhead. Nevertheless, when P increases, the number of square misses becomes higher and higher, therefore we preferred to choose other parallelisation methods.

8.7.3 Data decomposition with mutexes

The two previous methods have used mainly the domain (environment) decomposition. This method and the next one use mainly data (resources) decomposition.

The only problem posed by the wave propagation parallelisation is the concurrent access to squares. Therefore, the most obvious and simplest parallelisation method is to solve it directly by using mutexes, presented in section 4.4.2. It was not implemented in our tool, because of time constraints.

Explanation

Each processor is assigned a part of resources, and propagates their potential. However, as the potential fields may overlap, the access to squares, for both read and write, is protected by a mutex (section 4.4.2). Then the propagation of each resource is done completely, hence no repropagation is needed.

The mutexes can be fine-grained (each square is protected by a mutex), or coarse-grained (a mutex per group of squares). Examples of such groups are a whole line of environment, a whole column of environment, and groups of $n \times n$ squares. Larger the group, lower the overhead of mutexes, but higher the probability to wait for mutexes to become available. The efficiency of groups depends on the position and the potential of resources.

The combinations between this parallelisation method and sequential methods are presented below:

- The recursive methods work without modification: their propagation is completely done, using mutexes each time a square is accessed.
- The iterative methods themselves are based mainly on domain decomposition, hence the data decomposition based on mutexes is not appropriate. Nevertheless, a mixing is possible, which resembles to the fixed domain parallelisation described above. During a first stage, it puts the right potential in each resource square. During the second stage, each processor sweeps (and processes) once all its domain and the neighbour frontiers which touch its domain. Because of the domain decomposition, only frontiers are accessed by more than one processor.

Each time a square on frontier is accessed, mutexes are used. The second stage is repeatedly executed until no square changes its potential.

- The method distance-based with most influent resource works without any modification: the propagation of resources is completely done, using mutexes each time a square is accessed.
- The method distance-based with storing the distances to all influent resources is not appropriate to mutexes. Indeed, in this sequential method, computing the potential of a square involves only information on this square and the potential of resources involved.

Discussion

The advantage of this parallelisation method is that it does not have repropagations (except the combinations with iterative methods); repropagations increase execution time. Instead, the execution time is increased by the overheads of mutex operations. A timing model is difficult to give, since the overhead of mutexes (acquiring, waiting and releasing a mutex) depends on many parameters. While it seems that the overheads added by this method are higher than the improvements induced by the decreasing of the number of repropagations, only experiments can provide a qualitative measure of the performance of this method.

The memory overhead is given by the structures associated to mutexes. If we note by m_m the average memory overhead per square of a mutex, then the memory overhead of this parallelisation method is (N is the total number of squares in environment):

$$\text{mem} = Nm_m$$

As an example, if a mutex is associated to each square, and $m_m = 24$ bytes², we obtain a memory overhead of $\text{mem} = 24N$ bytes. This number is much higher than for fixed domain decomposition method, which needs a memory of $O(PN_c)$, with N_c the number of columns. It is also higher than the memory needed for recursive and iterative sequential methods (about N bytes, as shown in section 8.6).

8.7.4 Data decomposition with processor-private environments

The main issue of the previous method is the use of mutexes to avoid the concurrent access to squares. This concurrent access can be avoided also by duplicating the (main) environment on each processor, so that each processor access only its environment. This method based on private environments has been implemented in our tool.

²Case of a GNU/Linux machine with glibc library.

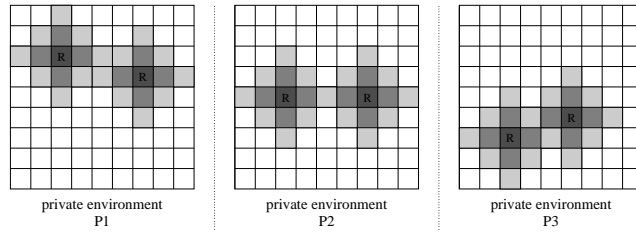


Figure 8.15: The first stage of the private-environment parallelisation method: each processor propagates a part of resources in its private environment.

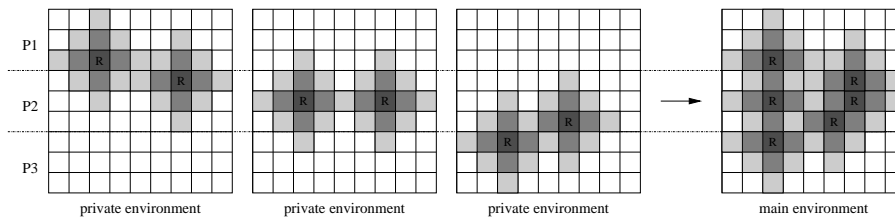


Figure 8.16: The second stage of the private-environment parallelisation method: each processor works on the same part of every private environment and updates each square of the main environment with the maximum potential of the corresponding squares.

Explanation

In this method, each processor has a private memory of the same size as the environment. The complete propagation is done in two stages. The first stage (private propagation) uses a data (resources) decomposition: each processor propagates a part of resources through all its private environment E_i (figure 8.15). During the second stage (global propagation), a domain decomposition is used: the environment is updated by all the threads in parallel; each square of the main environment receives the maximum of all its corresponding squares of the private environments E_i (figure 8.16). This stage can be skipped for squares whose potential is not needed. To avoid concurrent access to private environments, a synchronisation point is needed between the two stages. Thus each processor executes the following algorithm:

```

clear to 0 the potentials of its own private environment  $E_i$ 
for all own resource  $r$  do
    propagate  $r$  in its own private environment  $E_i$ 
end for
synchronisation point
for all square  $S_{main}$  in its domain of the main environment do

```

$S_{main} \leftarrow \max S_{private}$ {the corresponding squares in each private environment}

end for

Not all the sequential propagations described in the previous section are appropriate to the private propagation, as described below:

- Both recursive sequential methods of propagation work without modification. In our tool, the combination with recursive breadth-first method has been implemented.
- Both iterative sequential methods of propagation work without modification. In our tool, the combination with iterative variable potential method has also been implemented. It is worthwhile to note that, if the potential of resources is comparable to the environment size, the iterative sequential method is inefficient to such a parallelisation. Indeed, in this case reducing the number of resources to process (each processor is assigned only a part of resources) does not reduce the rectangle to be swept.
- The distance-based method with storing all the resources does not need private environments. Nevertheless, this combination can be imagined, by allowing each square of the private environments to store the distance to only *some* resources. It is very inefficient though, compared to its mixing with the fixed domain decomposition described above.
- The distance-based method with storing the most influent resource works without modification. The reason is that the most influent resource among all influent resources on a square is the same as the most influent resource among the most influent resource of each private environment. As we have not implemented this sequential method, we cannot give more information about this mixing.

Discussion

Compared to previous parallelisation methods, this method uses only one synchronisation point. The trade-off is that the processors have changed domain during the two stages, which leads to cache invalidations during the first stage and cache misses during the second stage. If the data is less than the cache size, then the performance of the algorithm depends on the number of square misses (squares processed by a different processor in the second stage). We are then interested by the number of square misses N_s . We suppose that each processor P_i writes *all* the squares of its own private environment, i.e. almost all the squares of environment have a potential greater than 0. Then, during the second stage, all the squares read by a processor in the private environment of other processors are square misses.

There are $P - 1$ other processors, and it reads N/P squares from each private environment, so the number of square misses for each of the processors is:

$$N_{sp} = (P - 1) \frac{N}{P}$$

Then the total number of square misses is given by:

$$N_s = PN_{sp} = N(P - 1)$$

This is a very high number, leading to a decrease of the practical performance of this parallelisation method (section 4.2.3).

The memory overhead of this method is given by the P private environments, each of them having N squares. Then the overhead in number of squares is:

$$\text{mem} = PN$$

If the potential radius of any resource is much less than the size of the environment, the memory overhead can be much reduced. In fact, the potential fields of resources in a domain spread on only a part E_i of the environment. Therefore, the private environments need not to have the same size as the main environment, but a less size, equal to E_i . This optimisation is independent of the sequential method chosen.

When several types of resources need to be propagated (let N_t be the total number of types), two variants can be used:

1. Propagate each type completely before propagating another type:

```

for all type  $T_i$  of potential do
  do private propagation
  synchronisation point
  do global propagation
end for

```

The memory overhead in this case is one private environment for each processor. The total number of synchronisation points is $N_{sync} = N_t$.

2. Do the private propagation for each type, and thereafter the global synchronisation:

```

for all type  $T_i$  of potential do
  do private propagation
end for
synchronisation point
for all type  $T_i$  of potential do
  do global propagation
end for

```

The memory overhead in this case is N_t private environments for each processor. The total number of synchronisation points is $N_{sync} = 1$. This leads also to better load balancing.

Parallelisation method	+	-
Fixed domain decomposition	very little memory overhead cache well exploited (works also on distributed memory)	high number of repropagations
Changing domain decomposition	no memory overhead few number of repropagations	cache not well exploited
Mutex-based	no repropagation	high overheads in execution time high overheads in memory
Private environments	2 steps only (works also on distributed memory)	cache not well exploited high memory overheads

Table 8.3: Main advantages and drawbacks of wave propagation parallelisation methods.

8.7.5 Theoretical comparison of parallel methods

Table 8.3 presents a comparison among the parallelisation methods presented in this section. Each of them has advantages and drawbacks. The theoretical measurement is not sufficient to find the best method, therefore we have done many practical measurements, described below.

8.8 Performance of the wave propagation algorithms

In this section we present the sequential and parallel execution times of the wave propagation methods implemented in our tool.

8.8.1 Experimental condition overview

The system and measurement information is the same as in the vision case, see section 7.10.1 on page 135. Again, we have used both functions `clock` and `times`, but the difference between the times they have reported has been very small. In fact, the implementations of this algorithm do not use I/O, but only computation. We will use only the results of function `times` in our discussion.

8.8.2 Wave propagation algorithm implementation

The wave propagation algorithms have an initialisation part and a propagation part. The initialisation part consists of various memory allocations and variable initialisations, and is done only once, at the beginning of simulation. It has a very low execution time. The propagation part is done generally each cycle of simulation. Therefore, we will focus only on the propagation part.

The propagation part needs the potential of resources. The potential of

each resource is computed from its current load³ using a function given by the user of the tool.

Several parameters influence the execution time of wave propagation algorithms. Also, for more precision, in each graphic the legend contains information about all these parameters. They are:

- N , the number of squares in environment (y and x in legends).
- The type of environment (t in legends, $t0$ for toroidal and $t1$ for edged). However, the torus case is not supported in our implementations of wave propagation, hence we will use only the edged case in our experiments.
- w , the percentage of obstacles (walls). In our benchmarks we will *randomly* put them in the environment.
- c , the connectivity (4 or 8).
- For each resource, its potential type and its potential value. In our benchmarks we will *randomly* put r percents of resources in environment, each of them with a potential value of p and a potential type t (we will use only one type, noted with T in legends). The load balancing in case of domain partitioning is then statistically assured. Also, this statistical balancing appears in many simulations done.
- The sequential and the parallelisation method (m in legends). Three sequential methods have been implemented: depth-first recursive, breadth-first recursive, and iterative with changing potential. The two parallelisation methods implemented are: fixed domain decomposition and private environments. These gives six combinations possible: depth-domain ($m4$), breadth-domain ($m0$), iterative-domain ($m8$), depth-private, breadth-private ($m9$) and iterative-private ($m10$). Additionally, we will use $m6$ in legends for the mixing iterative-breadth-domain.
- P , the number of processors.

All the legends have the form $y.x.t.w.c.T.p.r.m.$, according to the parameters above.

In our implementation a potential of 1 in a resource means its potential field is the resource square itself. This is different than in the vision algorithm, where a vision radius of 1 covers the square itself *and* its 4 (or 8) neighbours. Hence, in a world without obstacles, the potential field of a resource of potential n is the same as the visibility field of a square for a vision radius of $n + 1$. Notice that this does not affect theoretical complexities, but only practical results, which are presented in this section.

³Except for the very first propagation, where the initial potential given by the user is used (appendix A).

8.8.3 Performance of each method

In this section we present the performance of each method. As the theoretical results are not sufficient to find the most appropriate solution, in this section we will discover the slow methods we have implemented and, in the next section, we will discuss the fast methods.

Unless otherwise stated, we will use for each method the following parameters: an edged environment with 1024×1024 squares, without obstacles, with a connectivity of 4, and one potential type with 1% of resources each with a potential of 16. This is a realistic case, and we will present later its influence on the execution time of the best methods.

Note: The execution times presented in *all* these tests correspond to 10 runs of the propagation (hence 10 times greater execution times), in order to obtain execution times of minimum 0.5 seconds, for better accuracy.

The memory size needed by the wave propagation is given by the potential matrix and the environment matrix. The potential matrix has the size of environment (N elements), each element storing an integer value for each type of potential. For our tests we will use only one type of potential. This gives N integers. The environment matrix stores information about each square, such as its type and its dynamic state (occupied or not by an agent), and contains $5N$ integers. Summing up, we obtain $6N$ integers. In our experiments the size of an integer is 4B (4 bytes). Therefore, the total memory requirements of our implementation of wave propagation is $6N \times 4$ bytes, i.e. $24N$ bytes.

The cache of each processor on our DSM Origin 2000 64 processor machine has 4MB. Therefore, in an environment of 512×512 squares, 6MB are used, i.e. the size of 1.5 caches. For $N = 1024 \times 1024$, 24MB are used, i.e. the size of 6 caches. For $N = 4096 \times 4096$, 192MB are used, i.e. the size of 64 caches. Surely, as the cache contain code and other minor data, the real value may be a bit higher. We will use this information to discover the cache influence on our implementations.

Breadth-first with fixed domain decomposition

Figure 8.17 presents the bar plot of the execution time of this method. We can see that no important load unbalancing is visible, and that the execution time of 10 runs in 1-thread case is about 13.5 seconds. Figure 8.18 presents the speed-up of this method. Its maximum value (about 15) is reached somewhere near 32 processors. It is relatively scalable but is low, i.e. it does not have speed-ups close to the ideal speed-up. This result comes from the horizontal decomposition of the environment. Indeed, this method has two parts: domain propagation and frontier propagation. As shown in section 8.7.1, the execution time of the frontier propagation of the breadth-first method does not depend on the entire domain size, but just on the

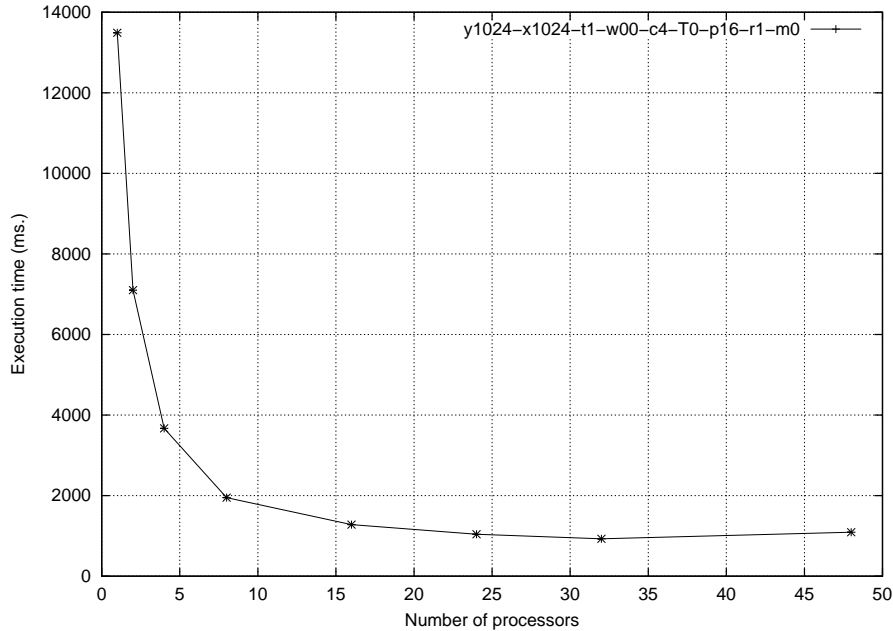


Figure 8.17: Bar plot of the execution time of 10 runs of breadth-first with fixed domain decomposition, showing good load balancing and a sequential time of about 13.5 seconds (DSM Origin 2000).

frontier length. In horizontal decomposition the frontier length is constant for any number of processors, hence the low curve of speed-up.

As explained above, the cache effects may appear after 6 processors. However, no change in performance can be seen in figure 8.18. In fact, this method propagates entirely and separately each resource, and the data size of the potential field of any resource is smaller than the cache size. Therefore, the cache size does not affect the performance of this method.

This method is relatively scalable so we will take it into account in the next section.

Depth-first with fixed domain propagation

Figure 8.19 presents the bar plot of the execution time of this method. No important load unbalancing appears. However, the execution time of 10 runs for 1-thread case is about 700 seconds. This time is much higher compared to previous breadth-first method (about 13.5 seconds). This result has been expected, since the number of squares visited by this method is much higher than for the breadth-first method, as shown in section 8.6.1. Therefore, we will eliminate this method of propagation in our following tests.

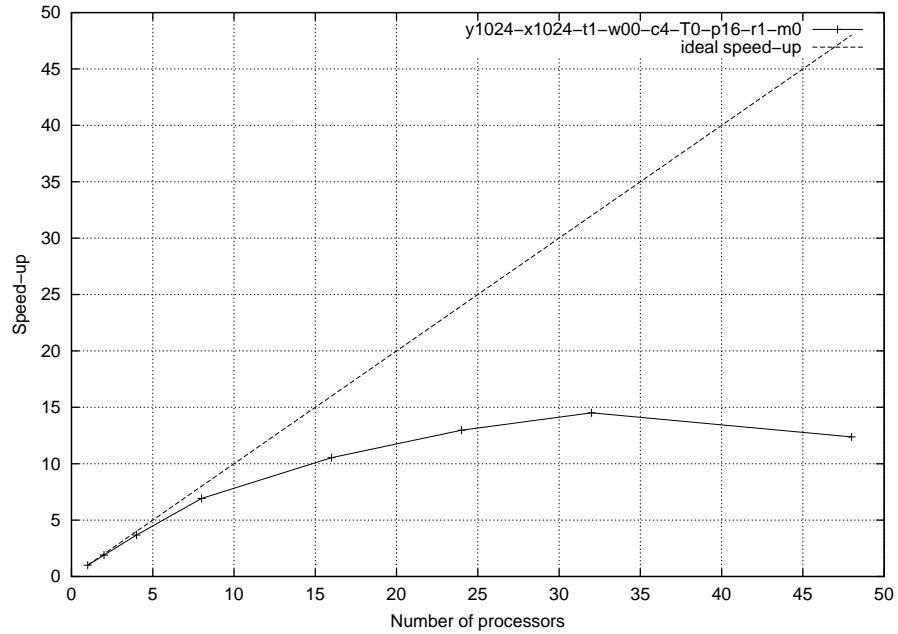


Figure 8.18: Speed-up of breadth-first with fixed domain decomposition, showing good speed-up, with a maximum somewhere near 32 processors (DSM Origin 2000).

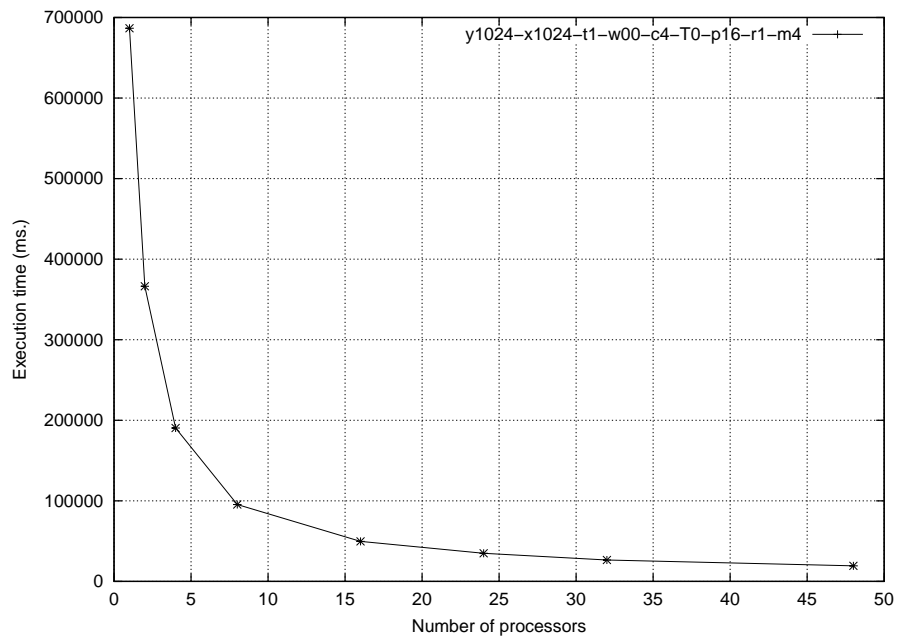


Figure 8.19: Bar plot of the execution time of 10 runs of depth-first with fixed domain decomposition, showing good load balancing but a very high sequential time of about 700 seconds (DSM Origin 2000).

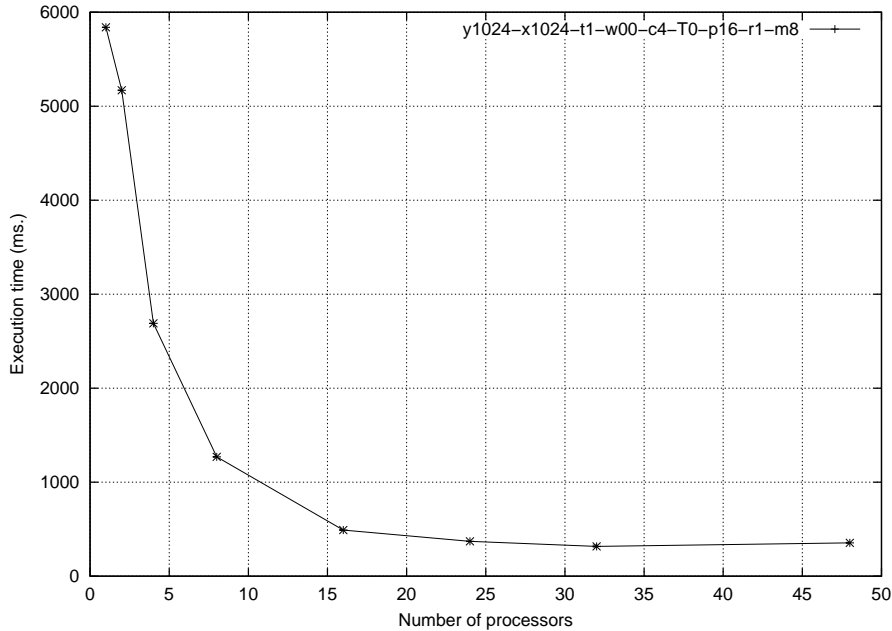


Figure 8.20: Bar plot of the execution time of 10 runs of iterative-variable with fixed domain decomposition, showing good load balancing and a sequential time of almost 6 seconds (DSM Origin 2000).

Iterative-variable with fixed domain propagation

Figure 8.20 presents the bar plot of the execution time of this method. We can see that no important load unbalancing is visible, and that the execution time of 10 runs in 1-thread case is almost 6 seconds (smaller than the breadth-first method).

Figure 8.21 presents the speed-up of this method. We can notice that for a very small number of processors (up to 4) the speed-up is very low. The reason is that the frontier propagation involves only the squares of the frontiers, but this method sweeps inefficiently all the environment.

It is very interesting to notice that starting for a few processors the speed-up curve of this method is in fact highly influenced by two factors, both positive. The first factor is the classical cache size, as described above: the speed-up of 8 processors (equal to 4.5) is more than twice higher than for 4 processors (equal to 2.1). Indeed, this method sweeps all the environment several times, and this operation is faster when all the data can be stored in caches.

The second factor is due to overwork of changing algorithm in parallel case (figure 4.3, page 82), and leads to the conclusion that the speed-up becomes better starting from a certain number of processors. Figure 8.22 presents three curves. The first one represents the execution time of the

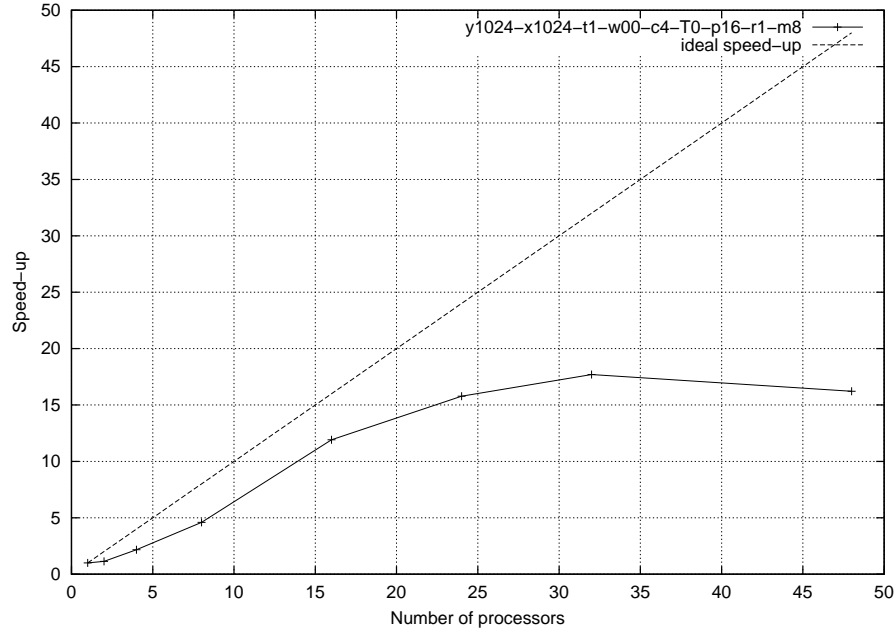


Figure 8.21: Speed-up of iterative-variable with fixed domain decomposition, showing mainly its non-scalability, the overwork effect from 3 processors, and the cache effects starting from 4–8 processors (DSM Origin 2000).

method without cache effects (environment 4096×4096). In parallel, this method has two parts: domain propagation and frontier propagation. Based on the execution time complexity of $O(Np)$ of the method (section 8.6.2), we consider that the execution time of domain propagation is proportional to the size of the domain, i.e. the more processors, the smaller domain, hence the smaller execution time. The second curve presents the theoretical execution time of the domain propagation part, with a parallel time equal to the sequential time divided by the number of domains (processors):

$$\text{time}_{\text{par}} = \frac{\text{time}_{\text{seq}}}{P}$$

The “frontier propagation” curve is simply the difference between the first and the second curves. It contains all the parallel overheads, the most important being the parallel overwork. This curve corresponds to theoretical expectations, since it decreases starting from 2 processors (and is 0 for 1 processor).

The smaller execution time for 1-thread compared to breadth-first method, the cache effects which can highly speed up its execution, and the better speed-ups for a higher number of processors have led us to take it into account. We will analyse it more thoroughly in the next section.

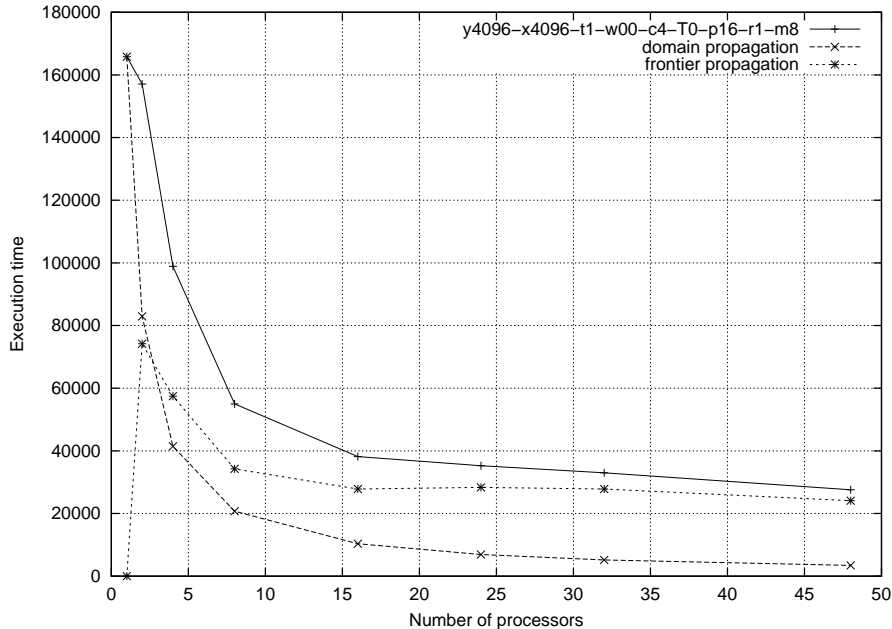


Figure 8.22: Execution time of 10 runs of iterative-variable with fixed domain decomposition on a bigger environment, along with theoretical execution times of its two parts, showing the overwork effects (DSM Origin 2000).

Breadth-first with private environments

This method differs from the breadth-first with domain decomposition method only in parallel, hence we are interested in its parallel performance only. Figure 8.23 presents the bar plot of this method. For the parameters chosen the best execution time appears for about 4 processors. After this number the cache misses, as shown in section 8.7.4, take over the benefits of parallelism. We are then interested on the speed-up on the scalable interval.

Figure 8.24 presents the speed-up of this method. For 4 processors the speed-up is about 3. This speed-up is not as high as for breadth-first with fixed domain decomposition, which has a speed-up of about 3.75 for 4 processors. Therefore, we will eliminate this method in our following tests.

Iterative-variable with private environments

Figure 8.25 presents the bar plot of the execution time of this method. We can see that the greater the number of processors, the greater the execution time: this method is not scalable. This result has been expected, since reducing the number of resources does not reduce the domain to be swept, always equal to the whole environment (section 8.6.2). Therefore, we will eliminate this method in our following tests.

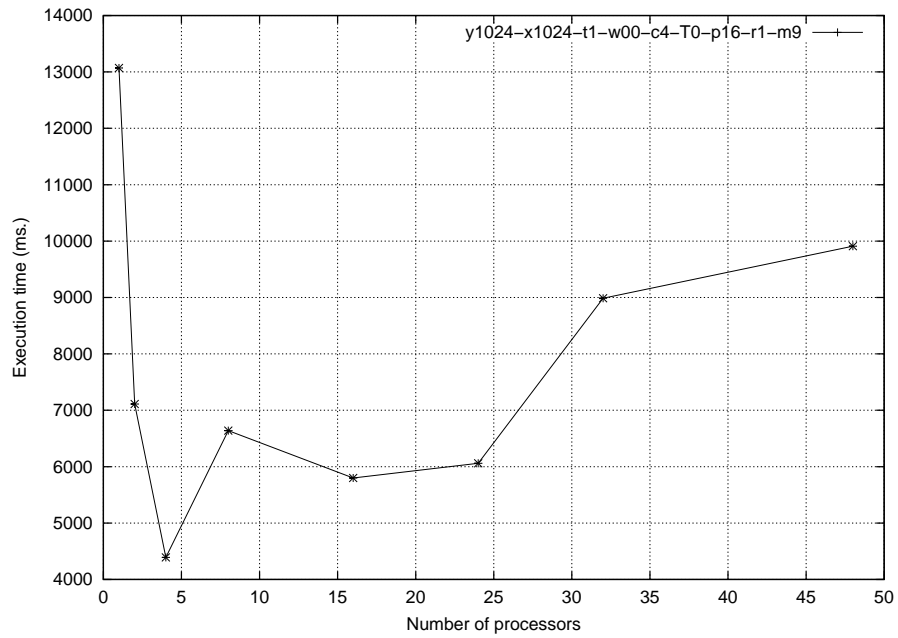


Figure 8.23: Bar plot of the execution time of 10 runs of breadth-first with private environments, showing good load balancing and best execution time for about 4 processors (DSM Origin 2000).

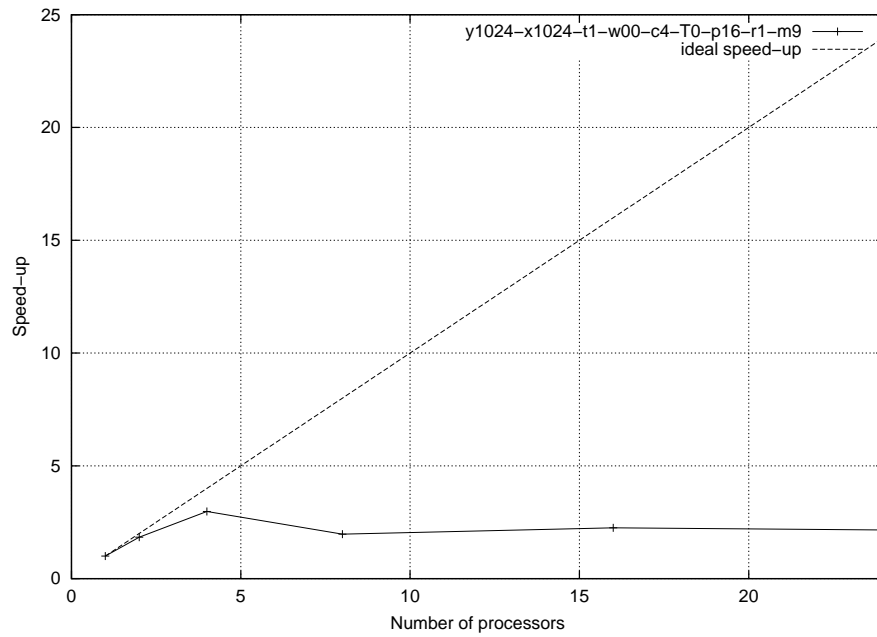


Figure 8.24: Speed-up of breadth-first with private environments, showing a speed-up of about 3 for 4 processors (DSM Origin 2000).

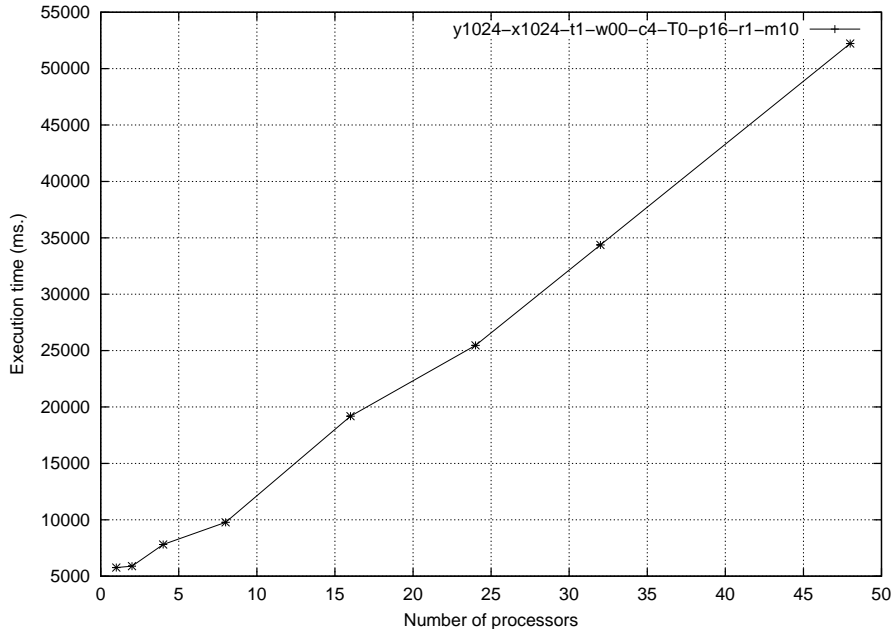


Figure 8.25: Bar plot of the execution time of 10 runs of iterative-variable with private environments, showing that it is not scalable at all (DSM Origin 2000).

8.8.4 In-depth comparison between the best methods

In the previous section we have eliminated all the methods but two, breadth-first and iterative-variable, both combined with fixed domain decomposition. In this section we will analyse in more detail the performance of the two methods.

Until now we have done tests on a realistic case: an edged environment with 1024×1024 squares, without obstacles, with a connectivity of 4, and one potential type with 1% of resources each with a potential of 16 (note that the number of resources is proportional to the environment size). We want to know if this realistic case is favourable to any of the methods.

In our 4-connectivity case we use formula 8.1 (page 172) with $p = 16$ and $R = 0.01N$, which gives the average number of resources influencing a square. We obtain that each square is influenced by an average of 5.5 resources. On the one hand, this is not the best case for breadth method. In fact, a better case for breadth method is when a square is influenced by only one resource. Also, an environment without obstacles is a case favourable to the iterative method, which is optimised in this case. On the other hand, this is not the best case for iterative method, since the benefit of cache effects in the iterative method are not visible until about 6 processors. Therefore, both methods have advantages and inconvenients in this case.

To obtain more information about the efficiency of both methods, we are interested in the performance of the methods when parameters change, both on the execution time and on the speed-up. We have done tests by varying *separately* each of these parameters, using the parameters above as basis: *An edged environment with 1024×1024 squares, without obstacles, with a connectivity of 4, and one potential type with 1% of resources each with a potential of 16.* We have obtained the following results:

1. *Environment size.* We have varied only the environment size. The percentage of the number of resources has been maintained constant (1% of the environment), hence their number has changed too. The execution times of 10 runs for 1-thread execution when varying the environment size are gathered in the following table:

<i>Size</i>	<i>Execution time (ms.)</i>	
	<i>Breadth</i>	<i>Iterative</i>
512×512	3230	1210
1024×1024	12540	5830
2048×2048	58630	26530
4096×4096	256620	165810

We notice that for both methods the execution time grows about linearly with the number of squares in environment. Also, the iterative method is faster than the breadth method in the 1-thread case.

The speed-ups of both methods are presented in figure 8.26 and 8.27. For both methods the speed-up increases with the environment size. However, the iterative curves show the positive influence of caches in the first two cases, and the scalability without the cache influence in the last case (4096), where the data size is always higher than the size of all the caches.

2. *Connectivity.* The execution time of 10 runs in 1-thread case when changing only the connectivity type is given in the following table:

<i>Connectivity</i>	<i>Execution time (ms.)</i>	
	<i>Breadth</i>	<i>Iterative</i>
4-connectivity	13520	5890
8-connectivity	30950	6770

We notice that the breadth method is highly influenced by the connectivity. The reason is that since the potential field in 8-connectivity is about twice larger than in 4-connectivity, each square is influenced by yet more resources, hence more square updates. However, the iterative method is not very much influenced by the connectivity. The reason is that the execution time depends mainly on the number of sweeps of environment, and in our case (0% obstacles) the number of sweeps is identical regardless of the connectivity.

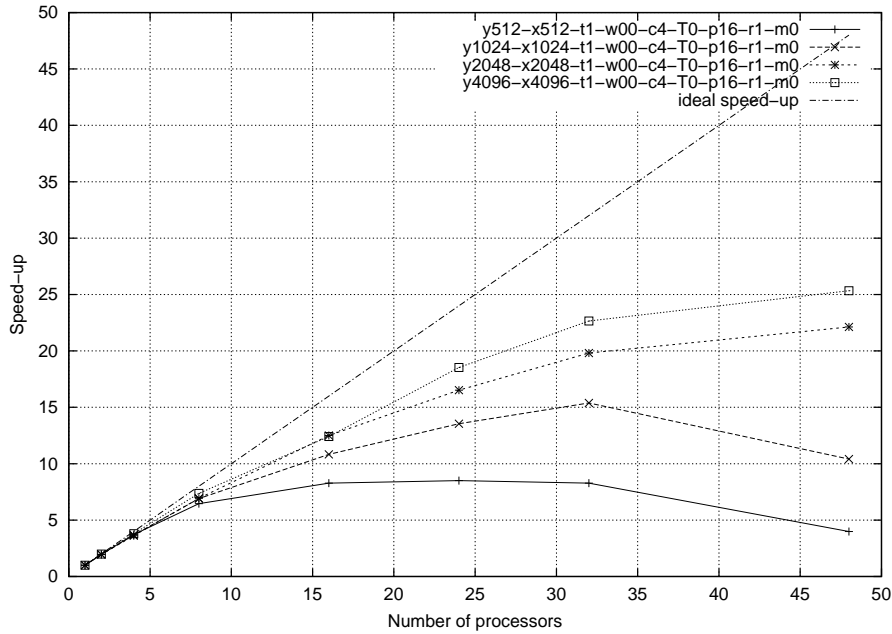


Figure 8.26: Speed-up of breadth-first with fixed domain decomposition, with various environment sizes, showing that the speed-up increases with the environment size (DSM Origin 2000).

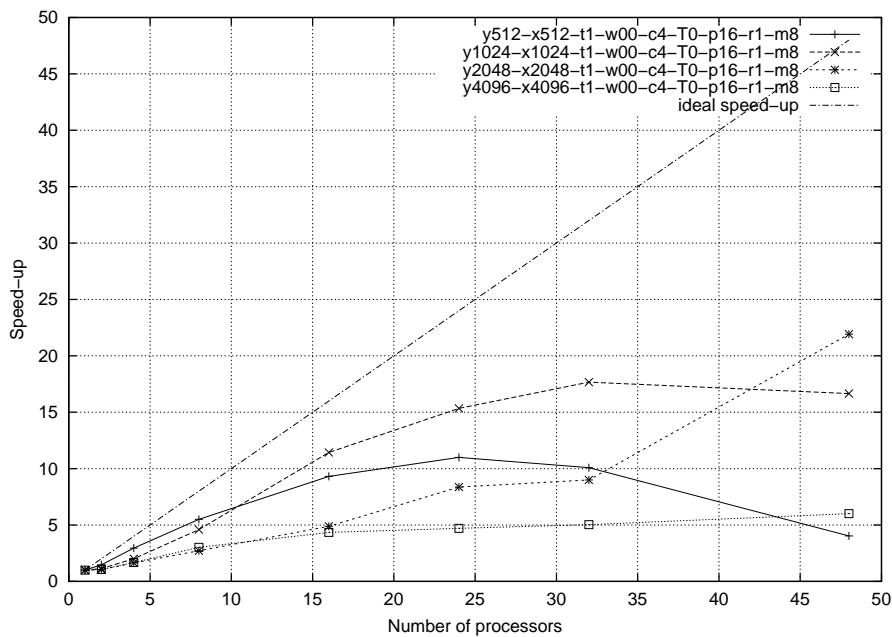


Figure 8.27: Speed-up of iterative-variable with fixed domain decomposition, with various environment sizes, showing the influence of caches and that the speed-up increases with the environment size (DSM Origin 2000).

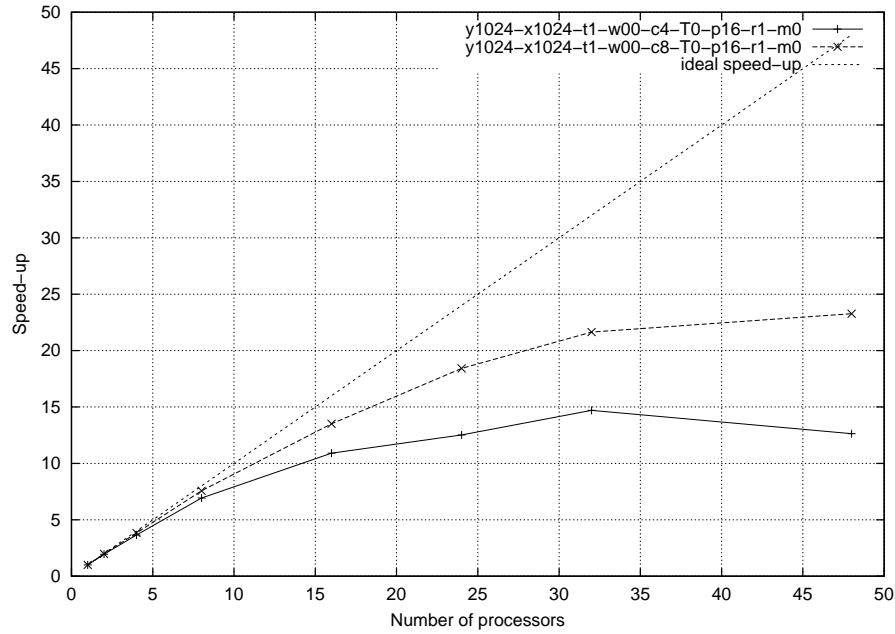


Figure 8.28: Speed-up of breadth-first with fixed domain decomposition, for 4-connectivity and 8-connectivity, showing a better speed-up for 8-connectivity (DSM Origin 2000).

The connectivity also influences the speed-up of the breadth method (greater for 8-connectivity, figure 8.28). The speed-up is better in 8-connectivity, but this is not useful in this case because the sequential execution time is much higher (30950ms compared to 13520ms, as shown above). The speed-up of the iterative method is not affected (figure 8.29).

3. *Number of resources.* The execution times of 10 runs in 1-thread execution when varying only the number of resources (in percentage of the squares of environment) are shown in the following table:

<i>Resources (%)</i>	<i>Execution time (ms.)</i>	
	<i>Breadth</i>	<i>Iterative</i>
0.1	3720	5560
0.2	5440	5620
0.3	6960	5720
1	13530	5890
10	45280	6740

For the breadth method, the greater the number of resources, the greater the execution time, because it propagates each resource consecutively and because squares are influenced by yet more resources.

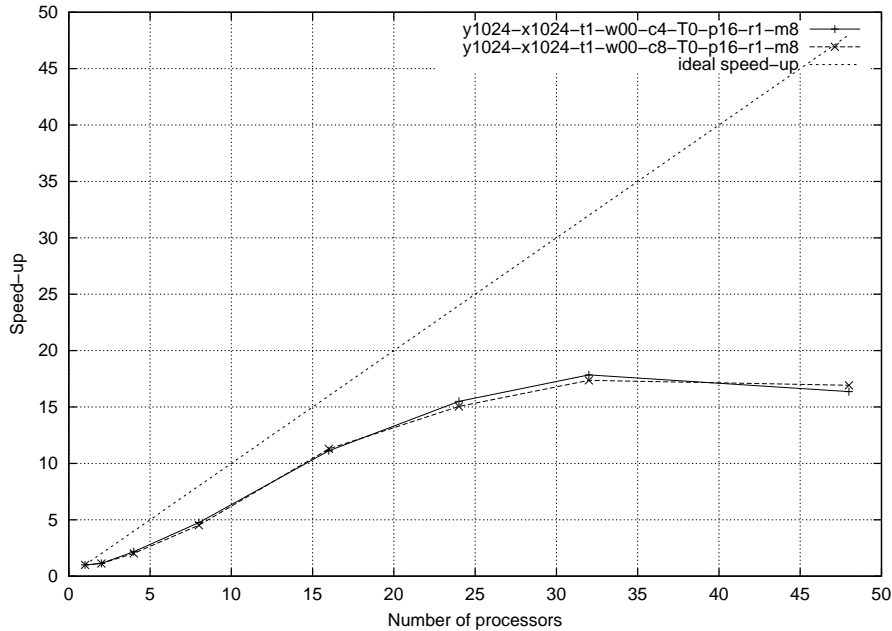


Figure 8.29: Speed-up of iterative-variable with fixed domain decomposition, for 4-connectivity and 8-connectivity, showing similar speed-ups (DSM Origin 2000).

The iterative method is not *very* much influenced by this parameter, the number of sweeps being the dominant parameter. For our parameters, the value where the execution times are similar for the methods is slightly greater than 0.2% resources, which corresponds to about 1.1 resources per square (formula 8.1, page 172, with a potential of 16). For about 0.185% of resources, where each square is influenced by an average of 1 resource (same formula), the breadth method is slightly faster than the iterative one.

For the breadth method, the greater the number of resources, the greater the number of computations, and the better the speed-up (figure 8.30). For the iterative method, the greater the number of resources, the smaller the speed-up (figure 8.31). Note that even if the speed-up is sometimes greater for breadth method than for iterative method (for instance for a potential of 10), the iterative method has a sequential time much smaller, hence its speed-up remains better than that of the breadth method.

4. *Resource potential.* The following table presents the execution time of 10 runs in 1-thread case when varying only the resource potential:

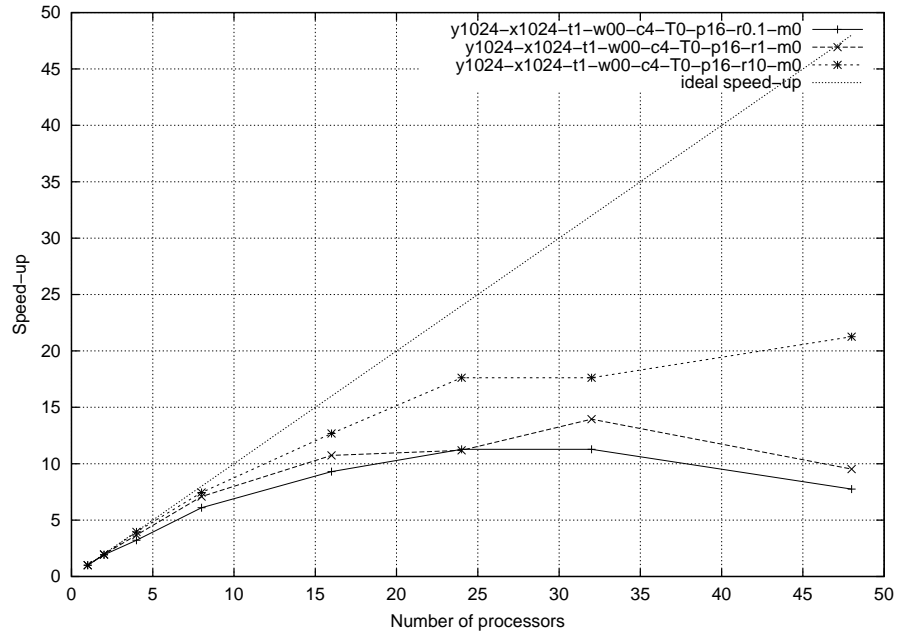


Figure 8.30: Speed-up of breadth-first with fixed domain decomposition, for various number of resources, showing that the speed-up grows with the number of resources (DSM Origin 2000).

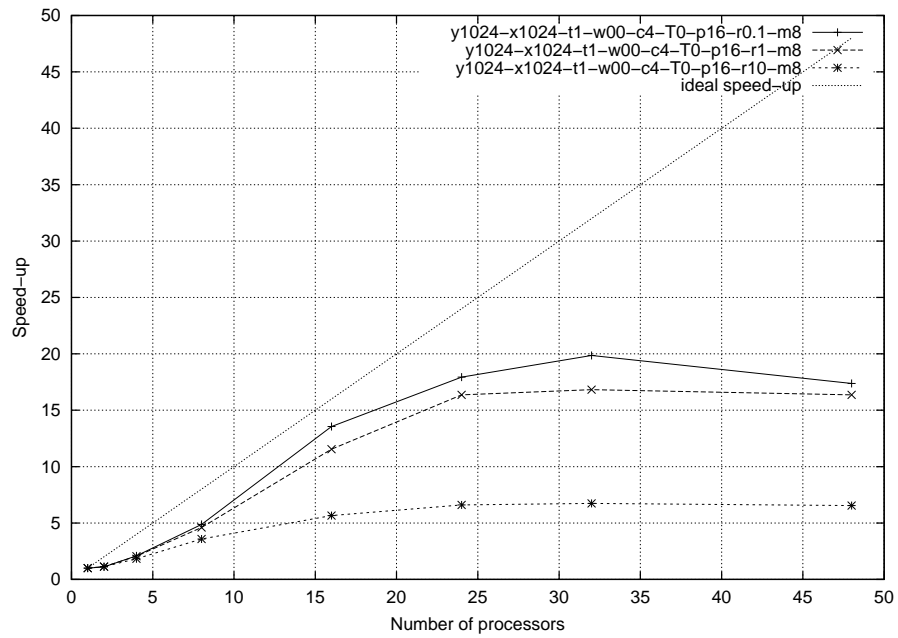


Figure 8.31: Speed-up of iterative-variable with fixed domain decomposition, for various number of resources, showing that the speed-up decreases with the number of resources (DSM Origin 2000).

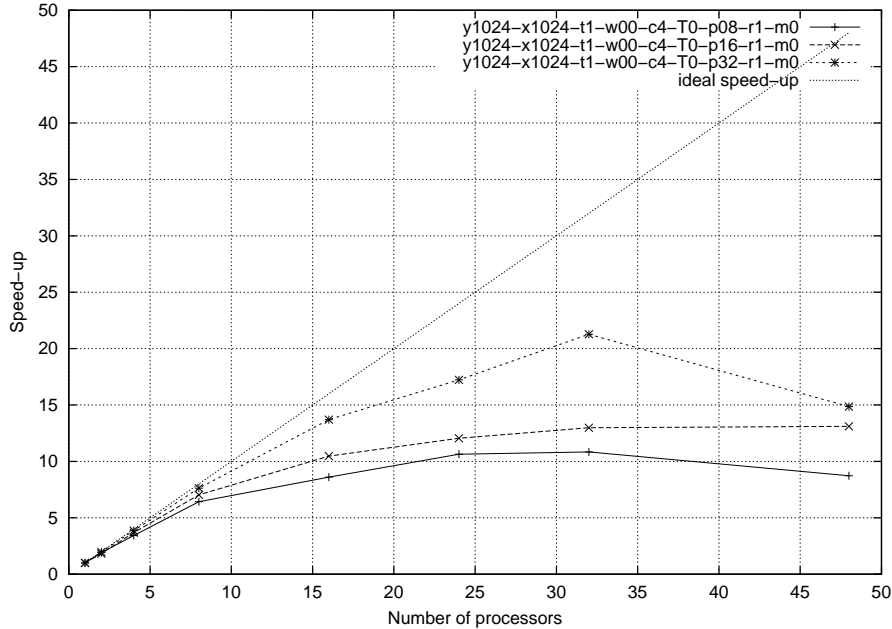


Figure 8.32: Speed-up of breadth-first with fixed domain decomposition, for various values for resource potential, showing that the speed-up increases with the resource potential (DSM Origin 2000).

<i>Potential</i>	<i>Execution time (ms.)</i>	
	<i>Breadth</i>	<i>Iterative</i>
8	5850	5890
16	13500	5820
32	28080	5830

Again, the breadth method is highly influenced by the resource potential, since the potential field size, hence the number of square updates, changes. The iterative method does not change its execution time. The methods have the same execution time for a potential of 8, which corresponds to about 1.45 resources per square (formula 8.1, page 172, with 1% of resources). An average of 1 resource per square is obtained for a potential of 6–7 (same formula), and in this case the breadth method is slightly faster than the iterative one.

Figures 8.32 and 8.33 present the speed-ups of the methods. For the breadth method, the greater resource potential, the more computations, hence the better the speed-up. For the iterative method, the speed-up is not much influenced.

5. *Number of obstacles.* The execution times of 10 runs in 1-thread case when varying only the number of obstacles (in percentage of squares), randomly put in the environment, are given in the following table:

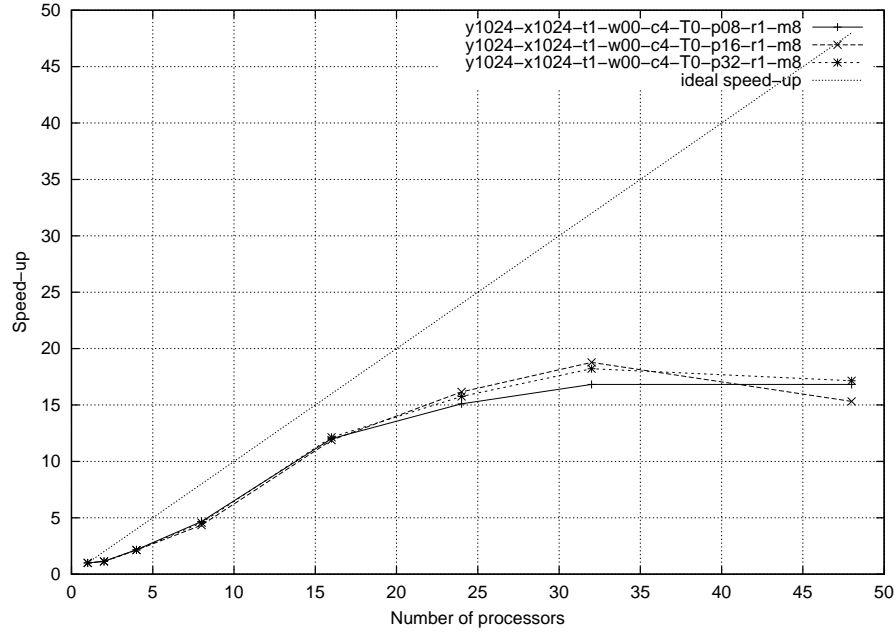


Figure 8.33: Speed-up of iterative-variable with fixed domain decomposition, for various values for resource potential, showing not very much influences (DSM Origin 2000).

<i>Obstacles (%)</i>	<i>Execution time (ms.)</i>	
	<i>Breadth</i>	<i>Iterative</i>
0	13500	5860
1	13470	8390
4	13250	10930
16	12080	13340

The greater the number of obstacles, the less the potential field size, hence the fewer square updates. We can see that the execution time of the breadth method slightly decreases with the number of obstacles. On the contrary, the iterative method is highly influenced by this parameter. It is worthwhile to note that the configuration of obstacles influences also the performance of the iterative method. As shown in its explanations (section 8.6.2), 3 sweeps are sufficient to completely propagate the potential on an environment without obstacles, while in the worst case numerous sweeps are necessary.

Figures 8.34 and 8.35 present the speed-up. For the breadth method, generally the greater the number of obstacles, the less scalable is the algorithm. Instead, the iterative method is not very much influenced by the number of obstacles. Moreover, in our cases its sequential execution time is almost always better compared to that of breadth.

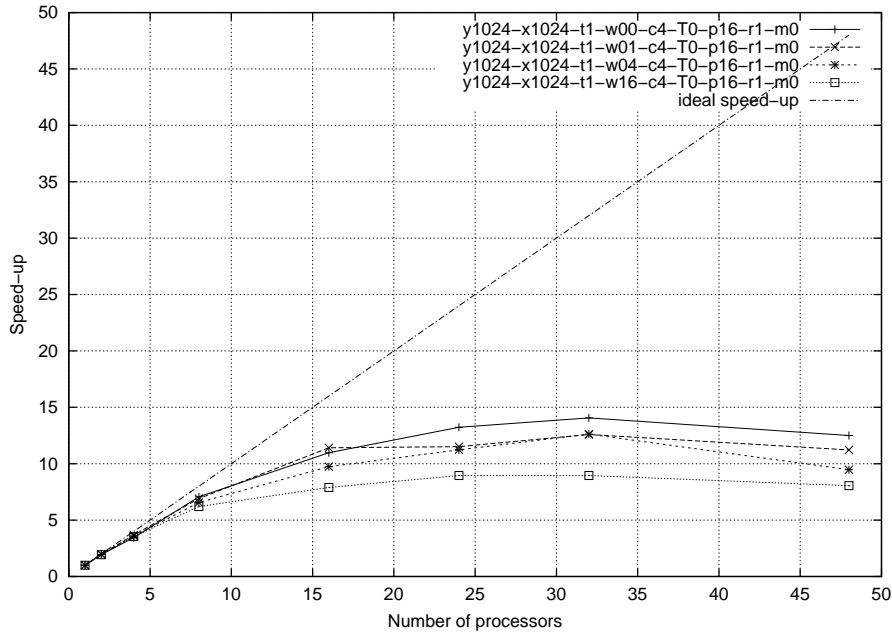


Figure 8.34: Speed-up of breadth-first with fixed domain decomposition, for various number of random obstacles, showing that generally the speed-up decreases with the number of obstacles (DSM Origin 2000).

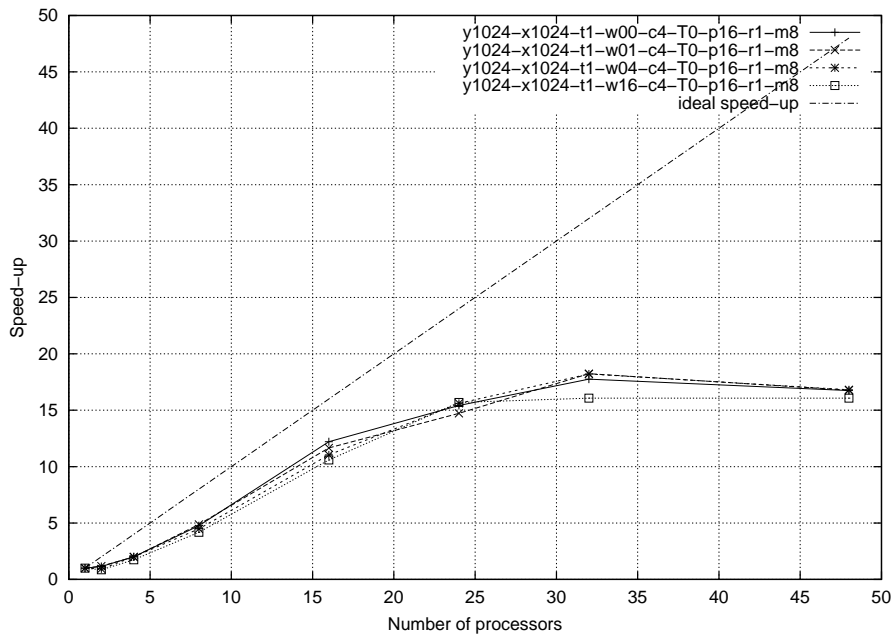


Figure 8.35: Speed-up of iterative-variable with fixed domain decomposition, for various number of random obstacles, showing that the speed-up is not much influenced by the number of obstacles (DSM Origin 2000).

8.8.5 Final best method

Based on the results presented in the previous section, we arrive to the following conclusions concerning the breadth and iterative methods:

1. Only the iterative method is highly influenced by the positive cache effects.
2. Both methods have a sequential execution time proportional to the number of squares.
3. The breadth method has an overall value where its sequential performance starts to degrade. This value depends on all the parameters discussed above, and corresponds to the case when *all* the environment is filled with *non overlapping* potential fields, i.e. an average influence of 1 resource per square (see also formula 8.1, page 172). This degradation is given by the inutile updates and may be very high.
4. The iterative method is much faster for small number of obstacles (with random positions).
5. On our system architecture (Origin 2000), similar sequential execution times have been obtained for an environment in 4-connectivity, without obstacles, and an average influence of about 1.1–1.45 resources per square (occurring when the data does not enter the cache).
6. With our horizontal decomposition the performance of the frontier propagation with the breadth-first method is not much influenced by the number of processors, while for iterative propagation it increases with the number of processors.

Therefore, in 1-thread case, a 4-connected environment, without obstacles, for parameters giving an influence of more than 1.5 resources per square and when the data does not enter 1 cache, the iterative method is faster than the breadth one (on our Origin 2000 system architecture). The 1-thread case corresponds to domain propagation. In our domain decomposition parallelisation method, the parallel case corresponds to frontier propagation. Because of the cache effects appearing for our parameters, the best performance in parallel depends on the number of processors. Therefore, the general best method is a combination of the two methods, depending on the parameters needed and on the number of processors used.

We have implemented the combination of iterative domain propagation and breadth frontier propagation. The sequential time of 10 runs, corresponding to domain propagation, is, of course, equal to the iterative method, i.e. about 6 seconds, hence we can compare their speed-up. The speed-up is presented in figure 8.36. The first curve on the legend represents our combination, while the second one only the iterative method (the same as

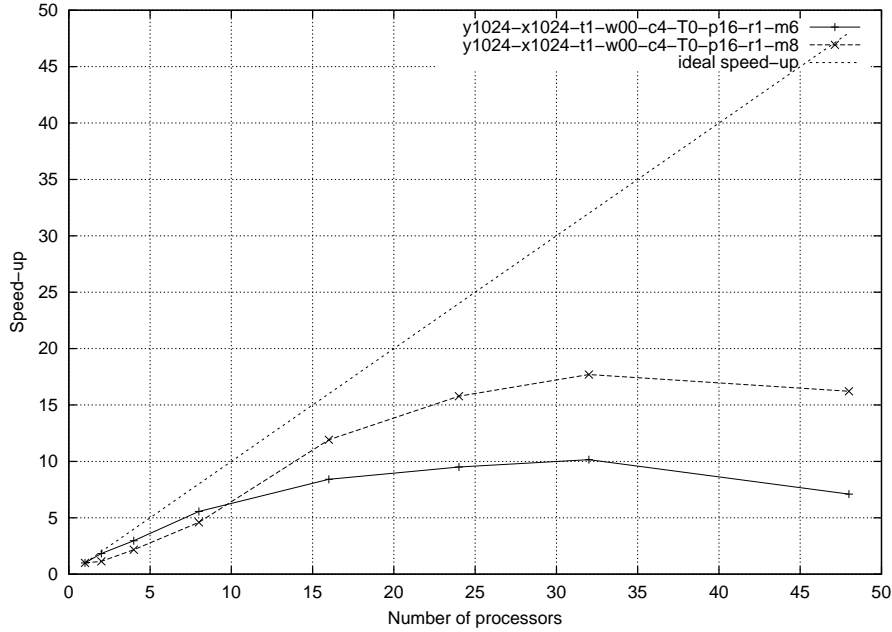


Figure 8.36: Speed-up of the best method, a combination of iterative sequential domain propagation and either breadth-first for a few processors (first curve of the legend), or iterative for higher number of processors (second curve on the legend) (DSM Origin 2000).

figure 8.21 on page 198). As expected, we notice that the combination is better up to about 10 processors, while afterwards the iterative method is better. Therefore, for these parameters, the best method is yet another combination of the two methods. In our example, the best method is to use iterative method for domain propagation, and either breadth-first propagation if a few processors (less than about 10 processors, corresponding to a small parallel machine for example), or iterative propagation if higher number of processors (a better parallel machine). It is worthwhile to note that we can consider this speed-up as the speed-up from the user point of view, since it involves the best sequential execution time of our methods.

8.9 Inexact potential propagation

A method to reduce the execution time of simulations of wave propagation algorithms is their parallelisation. We present in this section a promising direction to reduce even more the execution time of simulations, and/or its memory requirements. The basic idea is to allow the tolerance of minor errors in potential propagation, provided that they do not affect agent's

performance from MAS point of view⁴.

This idea is not new. Rabin's work on the notion that "certain programs *should* be designed to produce errors on rare occasions" [72, page 68] has led to the idea of acceptable uncertainty. Particularly, Hayes [47] considers the gradient-following algorithm, "which the program deliberately makes imperfect by adding a random 'wiggle' to the cells' direction-finding procedure."

Surely, some systems are not appropriate for inexactitudes. However, the agents by definition have to adapt themselves to environment and they are autonomous, hence they will cope with inexactitudes provided that they are not too important.

What the term "important" means depends on agents' behaviour. As an example, let $e(S)$ be the error function equal to the difference between the exact and the given inexact propagation in square S ($e : \text{squares} \rightarrow Z$). Generally, we try to avoid nonmonotonic functions e , as they introduce false directions to resources, while monotonic ones do not suffer from this property.

Inexactitudes can appear not only on some regions of the environment, but also on the whole environment.

In order to provide actual potential of resources, the propagation is done each cycle of simulation. With the price of being not always actual, the simulation can be sped up by doing fewer propagations: the propagation can be done only once, at the beginning of the simulation, or each k cycles. This optimisation has been implemented in our tool (the user may define the k parameter).

Also, they can appear either in the sequential, and/or the parallelisation method. An example of sequential inexactitude is doing only a part of the propagation in iterative methods, i.e. stopping the iteration before the end of the propagation.

The following two examples of inexactitudes involve the parallelisation methods:

1. Since the parallelisation methods complete the propagation, doing only the sequential part in the methods presented in this chapter gives a strongly incomplete propagation. Also, doing only one or two frontier propagations gives a light incomplete propagation. Generally, the nearer the frontier of decomposition, the higher the inexactitude.
2. Another source of inexactitude appears in the context of mutex-based parallelisation. In fact, the mutexes were needed in order to still put the correct potential in case of simultaneous access from several threads. Therefore, not using mutexes at all avoids the overhead in execution time and memory requirements of mutex-based parallelisation method, but can generate some inexactitudes in the potential field storage.

⁴This idea was confirmed by the agent community who attended the "COLINE Day" (*Journées COLINE*) at LORIA, June 8–9, 2000.

It is difficult to give exact results about the appropriateness of these inexact methods, since this depends on many factors. For example, doing only the sequential part can rise no problem for very few obstacles randomly put in the environment, while it can be unacceptable for some given configurations of obstacles. Nevertheless, the tolerance of inexactitudes in propagation fields seems to be a promising direction of research, allowing potentially both less execution times and less memory requirements.

8.10 Conclusions

This section has presented the wave propagation model we use in our multi-agent simulations. This model allows resources to generate a potential which is used by agents in order to find them. As the potential of resources changes during the simulation, the potential fields need to be updated frequently and can occupy a significant part of the simulation time. Several sequential and parallelisation algorithms have also been presented. The theoretical analysis of these algorithms has not been sufficient to compare them. The practical performance analysis of the algorithms we have implemented has led us to focus on two sequential methods, a recursive one and an iterative one, parallelised with the fixed domain decomposition method. After a detailed comparison, we have detected the cases where each of them is better than the other, and have implemented a combination of them, giving the best performance, both in sequential and parallel. Its performance has been presented.

A promising way of research seems to be the use of methods doing slightly inexact propagation. They would allow a yet greater performance on execution time, without affecting the agents' quality from MAS point of view.

Chapter 9

ParSSAP features and implementation

ParSSAP (PARallel Simulator of Situated Agent Populations) is the library we provide to implement situated multi-agent systems. It is based on the simulation model presented in chapter 6. Its main goals are fast development time and time-efficient execution. The applications written with this library can be executed on all sequential machines and on parallel machines providing a shared-memory interface. In this chapter we present its features and explain some basic applications written with it. The library reference manual is found in appendix A.

9.1 Features of our library

We present in this section only the differences between the model presented in chapter 6 and our *current* implementation.

9.1.1 General features

In our library the environment is two-dimensional and can be toroidal or edged. For edged environments, obstacles are automatically added on edges (agents are forced to stay into the environment).

It is not possible to have two resources in the same square. Each resource has its own load and its own function which gives its potential based on its load. The propagation of the potential is automatically done by the library using one of the wave propagation algorithms presented in chapter 8 (only some of them are implemented). In order to speed up the simulation, the potential fields can be calculated either each cycle, or each k cycles (for a given k), or only at the beginning of the initialisation.

The memory of agents is given by a simple pointer to a memory space. This is a flexible approach, since the user is free to use it as he needs. In fact,

the user initialises the agent's memory and, during the creation of agent, he gives the pointer to the agent's memory and its memory size. The library then creates the real agent's memory by allocating a memory space with size equal to that provided by the user, and copying in it the memory space given by the user. The memory given by the user is no longer needed. The library automatically deallocates its memory when the agent is destroyed or at the end of the simulation.

Because of time constraints, currently agents have only two percepts: odour and vision¹. Functions dealing with local and global perception are clearly differentiated through their names.

The actions an agent can do are move, take/drop an object and create/destroy agents. Objects can be found only in resources, and take/drop actions are local to the agent square, therefore these two actions work only in resource squares. Additionally, a special action is provided by the library, random move (for the reproducibility of random numbers, see section 9.2.3).

Agents can move only one square, in its neighbourhood of 4 or 8 squares. They do not have an implicit direction and they do not rotate, they just move in a given direction.

In our library an agent can do only *one* of the following actions during a cycle: move, take and drop. Additionally, it may execute any number of create/destroy agents during the same cycle. The number of agents used in the simulation is not limited by the library.

The spatial conflicts are automatically taken into account by the library. The choice of the winner among agents in conflict is influenced by their priority. Two priorities have been implemented: high and normal. Agents which lose in a conflict are forced to stay.

The initialisation of the system may be done in code, in initialisation files or both.

9.1.2 Information about system's state and evolution

The library provides functions to save information about the simulation. The frequency of savings is left to the user, an appropriate place is the user function. Three kinds of information can be saved in files:

1. Statistics about the state of the system on each cycle, mainly the number of agents, the number of empty agents, the number of empty resources and the number of unvisited resources.
2. Information about all the agents, such as their move, and all the resources, such as their load, useful to visualise the evolution of the

¹Another simulator written before the beginning of this thesis has been supported mark sensing [57].

system. A tool² to visualise this information is also provided.

3. The state of the whole system (checkpoint). The simulation may be resumed afterwards using this information. The system may also be initialised from such information. There is one general file, with information about parameters of the system, and one or several local files, containing information about domains of environment. The domains are given by the decomposition of the environment, as described in parallel issues next section. The local files contain relative coordinates, which eases their use. Currently, checkpoints do not save/restore the memory of agents.

9.1.3 Parallel issues

Our library is parallel. The library uses threads and supports two thread interfaces: POSIX threads and Irix-native one. It can also be executed on sequential machines without multi-threading libraries, a C compiler being sufficient.

With the exception of the user function, the parallelism is involved in all the simulation steps: behaviour of agents, spatial conflict processing, action execution, and vision and propagation algorithms. Among the savings, only the checkpoints are saved in parallel. The system initialisation is also sequential.

The technique of parallelisation used by the library is the statical domain decomposition. The number of processors used remains unchanged during all the simulation. The environment is decomposed horizontally, each processor being given an equal number of lines of the environment (or differing by 1 if the division is not an integer number):

```
1  topline = nlines * threadid / nbthreads;
2  bottomline = nlines * (threadid + 1) / nbthreads;
```

The use of the library is very much eased by hiding almost entirely the parallelism to the user of the library. The only function dealing with parallelism is the one specifying the number of processors to use.

Compared to a sequential implementation our library presents two minor limitations:

1. During a cycle, no warranty is given about the *order* of execution of the behaviour functions of the agents³. In fact, behaviour functions may be executed concurrently.

²This tool has been written at our school by another person, before the beginning of this thesis.

³Of course, as already mentioned, each behaviour function is assured to be executed exactly once during a cycle.

2. Due to parallelism restrictions, the functions giving the behaviour of agents, and the functions giving for each resource its potential based on its load (see section 9.8) cannot use static or global variables, in the sense of the C language⁴. The only functions involved are those used by *several* entities, where a race condition might appear if the functions are executed concurrently by different processors.

A special attention has been paid to allow *randomness* and fully *reproducibility* of simulation. More information is given in the next section.

9.2 Simultaneity processing and parallelism in the library

Macroscopically, our model of simulation is synchronous, based on cycles. However, during a cycle agents must have an asynchronous view, more precisely they are activated simultaneously, without any synchronisation. So, on each processor we have to simulate this simultaneity and we have to deal with real concurrent access to data structures from different processors. In order to solve these issues, specific structures are used, and each cycle of simulation is divided in several steps. This section presents them and explains how simultaneity and parallelism have been taken into account in our library.

9.2.1 Internal structures

Agents can be created and destroyed run-time. Also, agents may be processed by different processors during simulation. In order to cope with the dynamical number of agents, we have implemented expandable vectors (which grow automatically when elements are inserted past their end).

Several structures will be used in our discussion:

- Agents: vector containing all the information about the agents, such as its actual and its planned position, and if it is to be killed. Its size can change during run-time and it can reuse deleted elements. It is shared by all the processors. Processors work on different agents, hence no mutex is used.
- Local requested created agents: expandable vector containing information about agents which are to be created in the local domain. It is private to each processor.
- Global requested created agents: expandable vector containing the agents to be created in the domain of another processor than the one

⁴This is due to the special properties of this kind of variables in a parallel program. However, you can use them if the application is sequential.

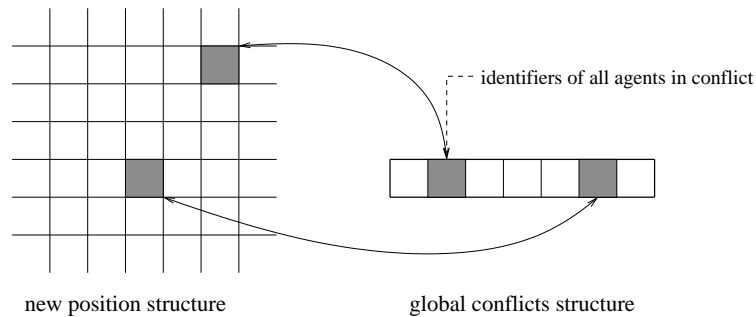


Figure 9.1: New position and conflicts structures.

which want to create the agent. It is shared by all the processors and its access is protected by a mutex.

- Global requested move agents: expandable vector containing the agents which are in a domain and move into another domain. It is shared by all the processors and its access is protected by a mutex.
- Global conflicts: vector containing the agents involved in conflicts. Each element corresponds to a conflict, and contains all the agents involved in the conflict (figure 9.1). It is shared by all the processors, but each processor accesses only the conflicts appearing in its domain, hence no mutex is used.
- Global new positions: static matrix of the same size as the environment, whose elements contain mainly the number of the conflict element of global conflicts structure, if any (figure 9.1). Each time an agent is added in this structure (“mark new position”), the library checks if there is already an agent inside and, if affirmative, creates a conflict. It is shared by all the processors, but each processor accesses only its domain, hence no mutex is used.
- Potential fields: static matrix containing the potential of resources. It is shared by all the processors, but each processor accesses only its domain, hence no mutex is used.

9.2.2 Execution flow

The following steps form a cycle and are executed by each processor:

1. Plan step: execute behaviour of all agents and plan agent creation and destruction.
2. Synchronisation barrier (wait for the global created agents and global move agents to be completely filled).

3. Transfer global requested created agents in local structures.
4. Transfer global requested move agents in local structures.
5. Spatial conflict step: solve all conflicts. Create winner agents. For loser agents, refuse their creation or change their move action in stay.
6. Create local agents.
7. Synchronisation barrier (wait for conflict solving to be completed).
8. Action step: kill agents and execute the planned action of the agents (move, take or drop).
9. Synchronisation barrier (wait for all agents' actions to be finished).
10. User step: execute sequentially the user function. This is the normal place to specify the end of simulation and to save information about the system. Also, the cycle number is incremented.
11. Synchronisation barrier (wait for the user to complete its a priori unknown processing).
12. Parallel saving step: if specified in previous step, the whole system is saved in parallel.
13. Environment percept update step: update if necessary the potential fields.
14. Synchronisation barrier (wait for the propagation of *all* the potential fields, since an agent may perceive information into another domain).

The whole procedure is presented in figure 9.2 on the next page and is explained below.

During the first step, each processor execute the behaviour of the agents found in its domain. If they request to create agents (through the call to the appropriate function) in the local domain, then the requested created agents are put in its local structure of requested created agents and their position is stored in the global new position structure. Elsewhere, if the position of requested created agents is in another domain, they are put in a global structure to be dispatched afterwards in a local structure. If they move and their new position is again in its domain, their new move is stored either in the global new position structure. If they move and change the domain, they are put in the global requested move agent structure.

After a synchronisation among all the processors, each processor reads the global requested created agent structure and copy the agents belonging to its domain into its local requested created agent structure. All the requested created agents are now in the appropriate local requested created agent structures.

9.2. SIMULTANEITY PROCESSING AND PARALLELISM IN THE LIBRARY221

```
repeat
  for all squares in its domain (PLAN STEP)
    if it contains an agent
      execute its behaviour
      [for all agents it creates
        if its new position is in its domain
          put it in local requested created agent structure
          mark new position
        else
          put it in global requested created agents structure
      for all agents it kills
        plan it to be killed]
    if it moves
      if its new position is in its domain
        mark new position
      else
        put it in global requested move agent structure
  SYNC
  for all global requested created agents (GLOBAL -> LOCAL CREATED AGENTS)
    if it is in its domain
      put it in local requested created agent structure
      mark its position
  for all global requested move agents (GLOBAL -> LOCAL REQUESTED MOVE AGENTS)
    if its new position is in its domain
      mark the new position
  for all (local) conflicts (SPATIAL CONFLICT STEP)
    if created agents exist
      choose randomly the winner among them
      create the winner
    else if high priority agents exist // only move agents
      choose randomly the winner among them
    else // normal priority only
      choose randomly the winner among them
    for all loser agents
      if it was to be created
        refuse its creation
      else // it wanted to move
        change its action into stay
  for all local requested created agents not in conflict (CREATE LOCAL REQUESTED AGENTS)
    create them
  SYNC
  for all agents whose new position is in its domain (ACTION STEP)
    if agent was to be killed
      kill it
    else
      execute its action
  SYNC
  if thid==0 (USER STEP)
    execute user function
    if it returns STOP
      mark end simulation
  cycle = cycle + 1
  SYNC
  save system if needed (in parallel) (PARALLEL SAVING STEP)
  if not end simulation
    recalculate potential fields if needed (ENVIRONMENT PERCEPT UPDATE STEP)
  SYNC
  clear all vectors
until end simulation
```

Figure 9.2: Execution flow of each processor.

Afterwards, the same processing is done for requested move agents. Now all the move agents are also in local requested move agent structures.

The next step solves the spatial conflicts among agents. The conflicts can be solved separately by each processor, since any conflict involves agents whose new position is identical, hence processed by a same processor. For any conflict case, in order, a random winner is chosen among the created agents (the priority is not used in this case). If there are no requested created agents in the conflict, a random winner is chosen among agents with high priority, if there are any, elsewhere among all the agents. Winner agents are created, while for loser agents, their creation is refused or their action is changed into stay (does nothing).

The next step creates all the agents not in conflict in the local requested created agent structure. Now, all the requested created agents appear in the agent structure.

After a synchronisation among all the processors, all the agents carry out their action. Also the requested killed agents are killed in this step.

After another synchronisation among all the processors, one thread executes the user function. If it returns `STOP`, then the simulation will end. The cycle number is incremented.

After another synchronisation among all the processors, the state of all the system is saved if it has been specified. The saving is done in this step because it is done in parallel by all the processors, while the previous step is executed in sequential. The files saved are the general file, saved by one of the processors, and the local files, one for each processor, saved in parallel by all the processors.

The last step updates if needed the percept data, in our case the potential fields. Also, all the vectors with cycle-life information are cleared.

These steps are executed repeatedly until the end of the simulation.

9.2.3 Randomness and reproducibility of simulations

Our tool provides fully reproducibility of simulations when using random numbers and different number of processors. Randomness refers to that all the functions involving randomness (see below) use random numbers. This means equal probability⁵ for all the choices, which is given by the simultaneity of actions. Reproducibility refers to that the simulation generates a fully reproducible simulation: the results of simulation are the same (provided that the initial conditions are the same).

Randomness appears in several parts of the model: put randomly agents in the environment, random move of agents (among all neighbour squares or among some of them), take/drop a random number of objects in resources, random choice of the winner in conflicts, creation of an agent in a random

⁵With respect to the quality of random numbers provided by the operating system libraries.

square of the environment, or simply get a random number for an agent. Generally, it involves no agent, one agent or several agents.

We need reproducible results for different number of processors. Therefore, in our implementation we cannot count on the order of elements in global tables, since it depends on number and speed of processors. We can neither count on the order of elements in local tables, since it changes for different number of processors. On the contrary, the behaviour of each agent is always processed sequentially.

Our method is to use parallel random number generators and several seeds (section 4.5). We use a global seed and numerous temporary seeds. The successfulness of our method is given by the following very important property: For an agent, even if we choose seeds by simple determinist formulae (e.g. the seed is the current number of cycle), as long as a same random number generator is used, the results are guaranteed to be random and reproducible. Therefore, for several agents we need to use different seeds.

The user specifies only one seed for the simulation, the global seed. It is used to allow diversity within a given simulation parameter space, and will be used every time a random number is generated. All the temporary seeds are generated from this seed.

The seeds are needed in three cases, depending on the number of agents involved:

- No agent is involved. This case appears only during the initialisation, where a specified number of agents, resources or obstacles are put randomly in the environment. This is executed in sequential, hence we can use safely the global seed.
- One agent is involved. This case appears when an agent does a random move, creates an agent somewhere in the environment or needs a random number. We use in this case a temporary seed generated by a dedicated function, as explained below. Each processor has one temporary seed for these cases. This seed is initialised for each agent immediately *before* executing the behaviour of the agent. It may be used several times during the execution of the behaviour function, if the agent does several times calls to functions involving random numbers. Since during a cycle an agent is processed sequentially, the reproducibility is thus guaranteed.
- Several agents are involved. This case appears in conflicts. We use again temporary seeds, created during simulation. Here, they are used only once. For each conflict we do the following processing. Firstly, we construct the table with *all* the agents involved. Then, as the order of elements in this table is not reproducible (some data may come from global vectors), we sort its elements in ascending order of agent's identifier (for move agents) or agent parent's identifier (for agents to

be created, which may come from global tables). Any such identifier is unique and reproducible⁶, therefore the order is now guaranteed. Finally, we choose the winner by a random number generator which uses a temporary seed generated by the same dedicated function, as explained in the next paragraph.

Identical seeds lead to generation of identical random numbers. Therefore, the seeds generated by the function dedicated to conflict cases must have four properties: parametrisable, spatial, temporal and “spatio-temporal”. Parametrisable means it depends on the initial seed given by the programmer, in order to allow diversity (as described above). Spatial means it depends on the position of the conflict (the position is unique to each conflict), otherwise during a cycle all the conflicts with the same number of agents involved would lead to the same rank for the winner. Temporal means that it depends on the cycle number, otherwise all the conflicts appearing in the same position during the simulation would lead to the same rank for the winner. “Spatio-temporal” means that it does not give the same result in different cycle *and* position, as exemplified below. In our implementation this dedicated function is:

```
gseed+7·cycle+y·dimx+x,
```

where (y, x) is the position of the conflict and `dimx` is the dimension of the environment on the x-axis. The multiplication of the cycle with 7, a prime number, tries to avoid generation of the same result for different cycles (the “spatio-temporal” property above). For instance, a function such as `cycle+x` would give the *same* result during a given cycle (`cycle+x`) and the next cycle if the agent moves at the left (`cycle+1+x-1`), which is avoided by introducing the multiplication with 7.

Finally, it is worthwhile to notice that our method of using temporary seeds avoids any memory overheads.

9.3 Compiling information

9.3.1 Compiling the library

In order to compile the ParSSAP library you have to edit the `Makefile` file and do the following changes appropriate for your system: the compiler to use, the file appropriate for your multi-threading library (`pthread`s or Irix threads), and a flag corresponding to run-time checks.

⁶Any information which is unique to the agent and reproducible might be used here, another example is their position in the environment. Note that even if several agents have the same parent's identifier, results are still reproducible, since their creation is requested by the *same* processor.

9.3.2 Compiling an application

In order to write an application you have to include the `agents.h` file of the library in your appropriate sources and link with the agents library (you can do this with the `-lagents` flag during the link stage). Depending on your system, you may also need to link with the multi-threading library (for instance with `-lpthread` on glibc-based systems, such as GNU/Linux).

An example of execution line is the following (the order of the parameters in this line is important!):

```
gcc myappli.c -lagents -lpthread
```

9.4 Execution of a simulation from the user point of view

Any application using ParSSAP library must call in order the following functions provided by the library:

- `pmInitSystem`, which allocates and initialises (with default values) internal data of the library.
- Either `pmCreateEnv`, which creates the environment, or `pmLoadSystem`, which creates the environment and optionally initialises the system.
- Optionally, miscellaneous functions, such as set the number of threads and put agents or resources in the environment. Note that, during this initialisation, some steps cannot be redone, for example a resource or an agent cannot be removed from the environment once it was put in.
- `pmRunSystem`, which starts the simulation.
- `pmEndSystem`, which deallocates the internal data of the library.

This sequence can be repeated indefinitely.

The following two sections describe two applications based on this sequence of function calls.

9.5 “Hello world” example

This section introduces you in the philosophy of the library by writing a very simple application: print a “Hello world” message to the screen. For that, we will create a MAS consisting of an environment containing one agent. The agent simply prints the message above. The full listing will be presented at the end of the section.

9.5.1 Implementation and explanations

First of all, in order to use the `ParSSAP` library we have to include the header `agents.h` and to initialise it before using it:

```
2  #include "agents.h"
```

```
27  pmInitSystem (0);
```

The parameter of the `pmInitSystem` function is the global seed, as explained in section 9.2.3. Also, notice that at the end of the simulation we have also to destroy the data used by the library:

```
31  pmEndSystem ();
```

Now, once the library is initialised, we can write the code for our simulation. Firstly, we create a 50×50 environment:

```
28  pmCreateEnv (50, 50, TORUS);
```

And we populate it with an agent:

```
29  pmPutAgent (0, 0, 1, NORM_PR, 0, NULL, 0, hello);
```

Notice that every function provided by `ParSSAP` starts with `pm` in order to avoid conflicts with other functions of the system. The environment has type `TORUS`, which means the left and right edges are glued, and also the top and bottom edges.

The creation of the agent needs several parameters. In order, we specify: its coordinates in the environment ($(0, 0)$), its slowness (1 means it moves every cycle of simulation), its priority (used in spatial conflicts between agents; here, we give it normal priority), its maximum load of miscellaneous objects (0, unused in this example), and the function which gives its behaviour. This function, called `hello`, is the following:

```
5  // behaviour function of the agent
6  // print the message "Hello world"
7  pmAction_t hello (void *mem)
8  {
9      pmAction_t act;
10     act.type = MOVERANDOM;
11     printf ("Hello_world!\n");
12     return act;
13 }
```

This function, called at the beginning of every cycle of simulation, has a precise header: it has one parameter (a pointer to the memory of the agent, which is unused in this example), and returns an action which corresponds to the desired action of the agent during the current cycle. The agent of this example simply does a random move and prints the “Hello world!” message.

We have now all the information about the world and its agent. It remains to start the simulation:

```
30  pmRunSystem (runfunc);
```

For flexibility, a parameter is given to `pmRunSystem`, which is a function to be called at the end of every cycle. We use it to specify when we want to end the simulation, in our case immediately (after the first cycle):

```

16 // called at the end of every cycle of simulation
17 // stop the simulation after one cycle
18 int runfunc (void)
19 {
20     return STOP;
21 }
```

The full listing of the application is the following:

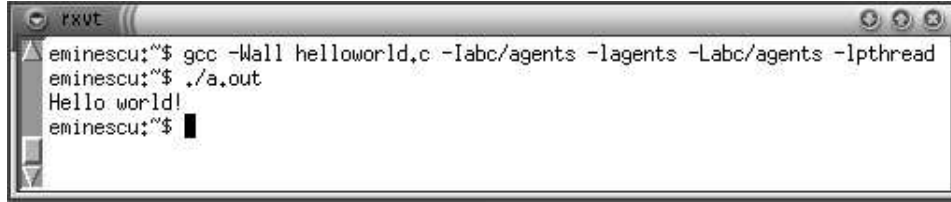
```

1 #include <stdio.h>
2 #include "agents.h"
3
4
5 // behaviour function of the agent
6 // print the message "Hello world"
7 pmAction_t hello (void *mem)
8 {
9     pmAction_t act;
10    act.type = MOVERANDOM;
11    printf ("Hello_world!\n");
12    return act;
13 }
14
15
16 // called at the end of every cycle of simulation
17 // stop the simulation after one cycle
18 int runfunc (void)
19 {
20     return STOP;
21 }
22
23
24 // the main function – beginning of the program
25 int main (void)
26 {
27     pmInitSystem (0);
28     pmCreateEnv (50, 50, TORUS);
29     pmPutAgent (0, 0, 1, NORM_PR, 0, NULL, 0, hello);
30     pmRunSystem (runfunc);
31     pmEndSystem ();
32     return 0;
33 }
```

The execution output is presented in figure 9.3 on the following page.

9.5.2 Richer “Hello world”

Suppose we want to run the simulation for more cycles, say 10. Our agent will then print 10 times the message, once per cycle. The only modification needed is the function `runfunc`, which becomes:



```

eminescu:~$ gcc -Wall helloworld.c -Iabc/agents -lagents -Labc/agents -lpthread
eminescu:~$ ./a.out
Hello world!
eminescu:~$

```

Figure 9.3: Execution output of the “Hello world” program.

```

int runfunc (void)
{
    if (pmgGetCycle() >= 10)
        return STOP;
    else
        return CONT;
}

```

It is possible to give agents to print also the number of the current cycle. The only function that changes is `hello`, which becomes:

```

pmAction_t hello (void *mem)
{
    pmAction_t act;
    act.type = MOVERANDOM;
    printf ("Hello_world!_The_current_cycle_is:_%d\n", pmgGetCycle());
    return act;
}

```

Notice that the `pmgGetCycle` function starts with `pmg`, with `g` letter standing for global. The user is thus noticed by the fact that this function uses global information, and not local one, as used by a pure MAS⁷.

Finally, if we want to have several agents in the world, say 15, we can use the function `pmPutRandomAgents` instead of `pmPutAgent` above:

```
pmPutRandomAgents (15, 1, NORM_PR, 0, NULL, 0, hello);
```

The parameters are the same, except that instead of the position we specify the number of agents desired.

9.6 “Diamond collector”, a more complex example

This section presents all the steps to implement and execute a simple program, the “Diamond collector”, using our library. The full listing will be shown in figure 9.4 on page 232.

9.6.1 Specifications

Here is the application to be written:

⁷However, some people can object against this affirmation by saying that every agent has a local notion of time.

Suppose a 2D maze with 50×100 squares, containing 50 squares already occupied (inaccessible obstacles). There are 15 fixed diamonds and 5 mobile workers. The goal of the workers is to collect all the diamonds. Any diamond/worker occupies one square. During every cycle of simulation, a worker has two choices: either collect a diamond if the square where it is contains a diamond, or move into one of the four neighbouring squares. We have to write a simulator of such a system, when the workers have totally random movement. The simulation ends after a specified number of cycles (say 1000), and we are interested by the number of diamonds collected during this time.

9.6.2 Implementation and explanations

Several steps can be seen in this example. First of all, as seen above, in order to use the `ParSSAP` library we have to include the header `agents.h`, to initialise it (the parameter is the global seed) before using it, and to destroy the data it uses at the end of the simulation:

```
2  #include "agents.h"

42 pmInitSystem (0);

50 pmEndSystem ();
```

Optionally, we can specify the number of threads used, which otherwise defaults to 1:

```
43 pmSetThreadsNo (4);
```

Now, once the library is initialised, we can write the code for our simulation. Firstly, there are the parameters of the environment: a 2D maze with 50×100 squares, 50 obstacles, 15 diamonds and 5 workers. To have a more readable program, we prefer to define these parameters:

```
4  // parameters of the simulation
5  #define DIMY 50      // world dimension (y)
6  #define DIMX 100    // world dimension (x)
7  #define OBSTACLES 50 // number of obstacles
8  #define DIAMONDS 15 // number of diamonds
9  #define WORKERS 5   // number of workers
```

Once defined, we create the world and populate it with objects:

```
44 pmCreateEnv (DIMY, DIMX, EDGES);
45 pmPutRandomObstacles (OBSTACLES);
46 pmPutRandomResources (DIAMONDS, 0, 1, 0, NULL);
47 pmPutRandomAgents (WORKERS, 1, NORM_PR, 100, NULL, 0, take_or_random);
```

The environment has type `EDGES`, which means obstacles are added automatically at the borders of the environment (we have a closed world).

In order to be more flexible, the library allows the creation of resources containing a certain load. We think thus of diamonds as resources of load 1 (only 1 diamond into every resource). Their creation needs a few parameters. The first is the number of diamonds to be created, and the second is their type (the library allows several types of objects to be defined). The next parameter specifies the initial load of the resource. We want only one diamond in them. The last two parameters are its initial potential⁸ and the function giving its potential as a function of its load. They are used in more complex simulations, for example in the case where diamonds propagate potential fields to attract workers.

The creation of the workers needs several parameters. In order, we specify: the number of workers, their slowness (1 means they move at every cycle of simulation), their priority (used in spatial conflicts between workers; here, all the workers have the same normal priority), their maximum load of diamonds, their memory (data address and data size, unused in this simple example) and the function giving their behaviour. This function, called `take_or_random`, will be written in the following. Here is its implementation:

```

13 // behaviour function of the workers
14 // take a diamond if possible, else move randomly
15 pmAction_t take_or_random (void *mem)
16 {
17     pmAction_t act;
18     // if found a resource with a diamond, take it
19     if ((pmlmGetSquareType() == RESSQ) && (pmlmGetSquareResLoad() == 1)){
20         act.type = TAKE;
21         act.param = 1;
22     }else // else do random movement
23         act.type = MOVERANDOM;
24     return act;
25 }

```

The implementation of this example simply checks if there is a source in its square and if it contains a diamond. In this case, it takes the diamond, else it does a random movement.

We have now all the information about the world and its workers. It remains to start the simulation:

```

48 pmRunSystem (runfunc);

```

The function `runfunc` allows us to specify when we want to end the simulation, i.e. when the desired number of cycles is reached:

```

28 // called at the end of every cycle of simulation
29 // stop the simulation after the specified number of cycles
30 int runfunc (void)
31 {
32     if (pmgGetCycle() >= CYCLES) // end the simulation, if max count reached

```

⁸See appendix A.4 for the reason of why we need to specify also the initial potential.

```

33     return STOP;
34     else
35     return CONT;
36 }

```

The constant `CYCLES` needs to have already the value of the desired number of cycles, 1000:

```

10 #define CYCLES 1000 // number of cycles

```

The full listing of the application is shown in figure 9.4 on the following page.

9.7 “Carrier robots”, an example with potential fields

This section presents an application using the `ParSSAP` library, mainly in order to illustrate environment configuration, user function and wave propagation use. More information about the functions provided by the library are given in the reference (appendix A). Suppose the following application:

There are 1024 robots in a 256×256 world. The world is a discrete torus (the right and left edges are glued, the same for bottom and top edges) and contains resources of ore (mines) and resources which catch the ore (factories). Agents’ goal is to carry all the ore from mines to factories, using the potential spread by resources and avoiding the obstacles. They can move one square in one of the four adjacent squares, or take or drop several units of ore if they are in a mine, respectively in a factory. Many behaviours for these agents exist, and we are interested by the emergence of a collective behaviour for this society of agents. We need also a statistic file providing the state of the agents during each cycle.

In order to implement such an application we will use a few development steps, described in the following.

9.7.1 System initialisation

This section presents an initialisation by code:

```

7  #define MINE 0
8  #define FACTORY 99

93  pmInitSystem (0);           // library initialisation , seed=0
94  pmSetThreadsNo (4);        // 4 threads for the simulation
95
96  pmCreateEnv(256, 256, TORUS);
97  pmPutRandomObstacles (4000);
98  pmPutRandomResources (2000, MINE, 100, 15, l2pmine);
99  pmPutRandomResources (2000, FACTORY, 0, 15, l2pfact);
100 pmPutRandomAgents (1024, 1, HIGH_PR, 5, NULL, 0, behaviour);

```



```

1  #include <stdio.h>
2  #include "agents.h"
3
4  // parameters of the simulation
5  #define DIMY 50      // world dimension (y)
6  #define DIMX 100    // world dimension (x)
7  #define OBSTACLES 50 // number of obstacles
8  #define DIAMONDS 15 // number of diamonds
9  #define WORKERS 5   // number of workers
10 #define CYCLES 1000 // number of cycles
11
12
13 // behaviour function of the workers
14 // take a diamond if possible, else move randomly
15 pmAction_t take_or_random (void *mem)
16 {
17     pmAction_t act;
18     // if found a resource with a diamond, take it
19     if ((pmlmGetSquareType() == RESSQ) && (pmlmGetSquareResLoad() == 1)){
20         act.type = TAKE;
21         act.param = 1;
22     }else // else do random movement
23         act.type = MOVERANDOM;
24     return act;
25 }
26
27
28 // called at the end of every cycle of simulation
29 // stop the simulation after the specified number of cycles
30 int runfunc (void)
31 {
32     if (pmgGetCycle() >= CYCLES) // end the simulation, if max count reached
33         return STOP;
34     else
35         return CONT;
36 }
37
38
39 // the main function – beginning of the program
40 int main (void)
41 {
42     pmInitSystem (0);
43     pmSetThreadsNo (4);
44     pmCreateEnv (DIMY, DIMX, EDGES);
45     pmPutRandomObstacles (OBSTACLES);
46     pmPutRandomResources (DIAMONDS, 0, 1, 0, NULL);
47     pmPutRandomAgents (WORKERS, 1, NORM_PR, 100, NULL, 0, take_or_random);
48     pmRunSystem (runfunc);
49     printf ("Number_of_diamonds_collected_is_%d\n", pmgGetResTotalLoad (0));
50     pmEndSystem ();
51     return 0;
52 }

```

Figure 9.4: Full listing of the “Diamond collector” program.

Firstly, the call to `pmInitSystem` initialises the library, giving 0 as the global seed. In the next line the number of threads (4) used for the simulation is given. By default, the agents can move only in the four squares in its close proximity, therefore no function call is needed. The statistic file also is initialised by default.

The world is a 256×256 toroidal environment, with 4000 obstacles (6% of the environment) at random positions. There are 2000 mines, with initial potential of 15, initial load of 100 and `l2pmine` as function of potential evolution. There are also 2000 factories, with initial potential of 15, initial load of 0 and `l2pmine` as function of potential evolution. All the mines and the resources are put at random positions in the environment. Finally, there are 1024 agents with the same priority at random positions, carrying maximum 5 units of ore and with `behaviour` as behaviour function. They do not have internal memory.

9.7.2 User function of end of cycle

The user function is called at the end of each cycle and is executed sequentially. This function will be used to decide the end of the application (by returning `STOP`) and to save information about the agents in the statistic file (by calling the `saveStat` function):

```

82 // callback function called at the end of each cycle
83 static int userfunction (void)
84 {
85     pmSaveStat ();           // save statistic information about the agents
86     if (pmgGetCycle() < 1000) // 0..1000, so 1001 cycles for the simulation
87         return CONT;
88     return STOP;
89 }
```

9.7.3 Functions related to resources

Suppose that the potential propagated by mines is a function of their load, while the potential of factories is constant. Thus two different functions are needed. Figure 9.5 on the next page presents the source codes and the plots of the two functions chosen: a linear function for mines and a constant one for factories.

9.7.4 Agent behaviour function

The behaviour of agents is the most difficult part of the application. Suppose the agents have the following behaviour (in pseudo-code):

```

1  if it is not full and it is into a non-empty mine, then
2    TAKE
3    return
4  if it is not empty and it is into a factory, then
```

```

70 // load->potential function for mines (linear)
71 static int l2pmine (int load, int initload , int initpot )
72 {
73     return load * initpot / initload ;
74 }
75
76 // load->potential function for factories (constant)
77 static int l2pfact (int load, int initload , int initpot )
78 {
79     return initpot;
80 }

```

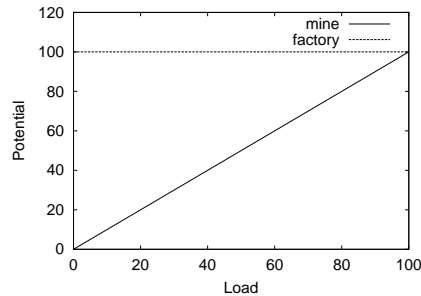


Figure 9.5: Resource functions in carrier robots simulation: function plots and corresponding ParSSAP code.

```

5     DROP
6     return
7     if its load is 0, then
8         use mine potential
9     else
10        use factory potential
11    if there is a square with the potential different than 0, then
12        choose it
13    else
14        do a random movement

```

As noticed, the agents do not need internal memory, but use potential fields. This can be implemented easily by using some percept functions provided by the ParSSAP library. The complete implementation of this behaviour in the ParSSAP library is presented in figure 9.6 on the facing page.

9.7.5 Simulation

The simulation is started by the call to function `runSystem`, giving it the above user function as parameter. When this function returns, a call to the `endSystem` function is necessary:

```

102 pmRunSystem (userfunction);
103 pmEndSystem ();

```

9.7. “CARRIER ROBOTS”, AN EXAMPLE WITH POTENTIAL FIELDS 235

```

5  #define DIRNB 4 // the connectivity

10 // agent' behaviour function, following increasing potentials
11 static pmAction_t behaviour (void *mem)
12 {
13     int dir, d;
14     int load, maxload;
15     int odourtype, odour, maxodour;
16     pmAction_t act;
17
18     load = pmlmGetAgentLoad ();
19     maxload = pmlmGetAgentMaxload ();
20
21     // if it is not full and it is into a non-empty mine, then TAKE
22     if ((load != maxload) && (pmlmGetSquareType() == RESSQ)
23         && (pmlmGetSquareResType() == MINE) && (pmlmGetSquareResLoad() != 0)){
24         int resload = pmlmGetSquareResLoad();
25         act.type = TAKE;
26         act.param = (maxload-load < resload) ? maxload-load : resload;
27         return act;
28     }
29
30     // if it is not empty and it is into a factory, then DROP
31     if ((load > 0) && (pmlmGetSquareType() == RESSQ)
32         && (pmlmGetSquareResType() == FACTORY)){
33         act.type = DROP;
34         act.param = load;
35         return act;
36     }
37
38     // if its load is 0, then search for mines, else for factories
39     odourtype = (load == 0) ? MINE : FACTORY;
40
41     // try potential first
42     dir = pmtmRand() % DIRNB;
43     // get the max odour
44     maxodour = 0;
45     for (d=0 ; d<DIRNB ; d++){
46         dir = (dir + 1) % DIRNB;
47         odour = pmlmGetSquarePotential (odourtype, dir); // if can move, get the odour
48         if (odour > maxodour)
49             maxodour = odour;
50     }
51     // found an odour
52     if (maxodour != 0){
53         for (d=0 ; d<DIRNB ; d++){
54             dir = (dir + 1) % DIRNB;
55             if (pmlmCanMove(dir) && (pmlmGetSquarePotential(odourtype, dir) == maxodour)){
56                 act.type = MOVE; // action type
57                 act.param = dir; // direction
58                 return act;
59             }
60         }
61         act.type = STAY;
62         return act;
63     }
64
65     // no odour found
66     act.type = MOVERANDOM;
67     return act;
68 }

```

Figure 9.6: Agent behaviour implementation in the carrier robots simulation, using potential fields.

9.8 Generic design of an application

As described above, in order to write an application using the `ParSSAP` library you have to follow these steps:

1. Write code for the initialisation of the environment and of the agents and call the function which starts the simulation (`pmRunSystem`). The initialisation can also be *entirely* made by files, see appendix A.
2. Write the *user-function*, which is called by the library at the end of every cycle (as shown in the “Hello world” example above). It allows to take control over the application at the end of every cycle. It can be used for example to end the simulation, to print or write files with information about the status of the simulation, such as miscellaneous statistics. For simplicity of the user, this function is executed in sequential.
3. Optionally, for every type of resource (the diamonds in the example above), write the function allowing to compute its potential based on its load. During the creation of the resources, you can specify that you do not need such functions by passing `NULL` as parameter. If you specify such functions, the system will calculate regularly the potential field of the resources concerned. The agents can use then these fields to guide themselves to resources.
4. Write the behaviour function for every agent. This function is called every cycle (if the slowness of the agent is 1) and specifies the action that the agent wants to do during that cycle. Notice that this action can be changed to nothing (i.e. the agent simply stays) if it is involved in a spatial conflict with other agents and it loses. The priority in such conflicts can be controlled by a parameter at the agent creation.

9.9 Conclusions

In this chapter we have presented our library and our solution to several problems, especially actions’ simultaneity, parallel execution, randomness and reproducibility. It consists mainly of breaking agents’ actions in three steps: plan, conflict resolution and execution steps. We have also presented how the global data access has been dealt in a parallel execution.

We have explained the use of our library by an in-depth description of a few applications using the library. In the next chapter we will present more complex applications, involving more advanced features of the library.

Chapter 10

Domains of application and global performance

We present in this chapter two applications. The first one is the well-known Conway's game of life, where cells are static and consists basically of creation and destruction of cells. The second one is a multi-consumer simulator, where people basically move between different stores, where they buy goods, and their home, where they bring their goods.

10.1 Conway's game of life

Description

The Game of life¹ is a two-dimensional cellular automaton [79] governed by a simple set of birth, death and survival rules. It was invented in 1970 by John Conway and popularised by Martin Gardner [39]. Each of the cells in the 2D universe can be in one of two states: alive or dead. Beginning with any given initial pattern of live cells, Conway's rules can be employed in order to determine the behaviour of the universe over any number of generations (time steps, cycles). Whether a cell survives, dies or comes into being is determined by the number of live neighbours the cell has. Each cell has eight possible neighbours (our 8-connectivity). The rules for survival, death and birth are as follows:

- Survival: if a live cell has two or three live neighbours, then it survives.
- Death: if a live cell has less than two or more than three live neighbours, then it dies.
- Birth: if a dead cell has exactly three live neighbours, it is born.

¹This description is adapted from the Web page "The Game of Life": <http://www.reed.edu/alife/classes/models/life/gameoflife.html>.

When these rules are employed to any given initial pattern of live cells, the results can be startling. Complex behaviour consisting of various life forms, composed of several or more living cells, often occurs [39].

Implementation models

In our library, Conway's universe can be modelled in two ways. For both of them, an 8-connectivity is assumed. The first model is to put an agent/resource in each cell, and execute their behaviour each cycle. One possible method is to put an agent and a resource (with load equal to 1) in each cell. The potential propagation is not used. Agents do not move, and they can take or drop the unit of the resource where they are. A live cell is represented by a loaded agent (or empty resource), and inversely. An interesting model, with several states for a cell (not only dead or alive), is obtained if we use greater potential for resources.

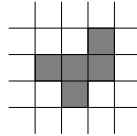
Another method is to have a resource in each cell. All the resources have a potential of 0 or 1, so that the potential fields do not overlap and the potential propagation is not needed. A live cell is represented by a resource with potential of 1, while a dead cell by a resource with potential of 0. The resource function is called every cycle and, based on the potentials of its neighbouring cells, each resource can compute its correct potential. An interesting model, with larger neighbourhoods, is obtained when the potential of resources can be greater than 1, with possibly different types of potential.

We can imagine yet another method, with only agents, one per cell. The communication among agents can be done through their memory, which we deliberately make shared to all agents (this can be done by giving the same memory address to all the agents' memory).

All these methods can be parallelised efficiently, since each cell do mainly the same processing and hence the processor load is approximately balanced. A drawback of these methods when implemented with our library is that they are not natural models, hence a specific external program has to be used to show the results of simulation.

By contrast, the second model is to have agents only in live cells, and dynamically create/destroy agents. The advantage compared to the first model is that the execution time is smaller, since the number of agents is smaller. As an example, for a 512×512 environment with few live cells, we may have only a few agents instead of 256 thousands agents. This model and its implementation are described in the next section.

As shown above, the library is sufficiently rich to allow several models to be used for an application such as Conway's universe.



1	<i>#agents</i>							
2	<i>#</i>	<i>y</i>	<i>x</i>	<i>slownes</i>	<i>priorit</i>	<i>load</i>	<i>maxload</i>	<i>funcInd</i>
3	A	20	22	1	0	0	0	0
4	A	21	20	1	0	0	0	0
5	A	21	21	1	0	0	0	0
6	A	21	22	1	0	0	0	0
7	A	22	21	1	0	0	0	0

Figure 10.1: Methuselah pattern and the corresponding data file.

Implementation chosen

Based on the fact that cells born only near live cells, we can use the natural model, where a live cell is represented by an agent, and a dead cell by an empty square. Agents will be dynamically created and destroyed. This model also allows us to use the already written external program to visualise the agents' position for each cycle of simulation.

The initial state of the environment contains an agent for each live cell in the initial pattern. Figure 10.1 presents an initial pattern known as Methuselah² and the corresponding data file, called `conway-methuselah`, used in our tool (lines starting with `#` character are comments).

The initialisation of the simulation can be done in code or in data file. We have chosen the latter, which is simpler. Here is the file, called `conway.ini`:

```

1  #number of threads used
2  nbThreads 1
3
4  #number of lines of the environment
5  dimy 512
6
7  #number of columns of the environment
8  dimx 512
9
10 #type of the environment: 0=TORUS, 1=EDGES
11 envtype 0
12
13 #connectivity
14 nbdirs 8
15
16 #local files
17 file 0 0 methuselah

```

We have specified the number of threads used, the dimension of the

²More precisely, “R-pentomino” [39]. A pattern which stabilises after a large number of cycles is called Methuselah, and there are several such patterns.

environment, its type, the connectivity and finally the file name with the initial state of the environment.

It remains to code the cells' behaviour. The processing done by each live cell (agent) is divided in two parts. The first part is to count the number of neighbours and, if appropriate, to die the next cycle. The second part is to check each of its neighbours and, for each dead cell, to check if it must be born or not. With such a behaviour, cells which will be born are processed by several neighbouring live cells. The library anyway creates one and only one agent, so the simulation is still correct. However, in order to program properly and to avoid multiple births, we have added code so that any born cell be created by only one live cell, namely the first in its neighbours in the order of directions checked (from UP to DOR). The full code in C language is presented in figure 10.2 on the facing page. Snapshots of the external program for a smaller universe are given in figure 10.3 on page 242.

A drawback of this implementation is that its parallel performance is satisfactory only in some cases. In fact, for a life form of a few and contiguous live cells, it is highly difficult to obtain an efficient parallelisation. The reason is that modern MIMD parallel machines are not appropriate for fine-grained parallelism, due to memory constraints. Nevertheless, if the use of our library does not allow a gain in parallel execution speed, it has allowed a gain in development time: the main thing written has been the cells' behaviour.

Possible extensions to Conway's game of life

Compared to a specific implementation, our implementation of Conway's universe allows an easy implementation of some exotic features. Such a feature is the use of a toroidal environment. A toroidal environment acts like an infinite environment for some patterns, and introduces exotic issues for others. Also, test agents with different slownesses, i.e. cells' computation is done regularly, but not every cycle.

A special case appears when obstacles are used in Conway's universe. New rules, taking into account obstacles, need to be written. New interesting patterns might appear in this case, for example a life form which transforms into another life form when it meets an obstacle.

10.2 Multi-consumer simulator

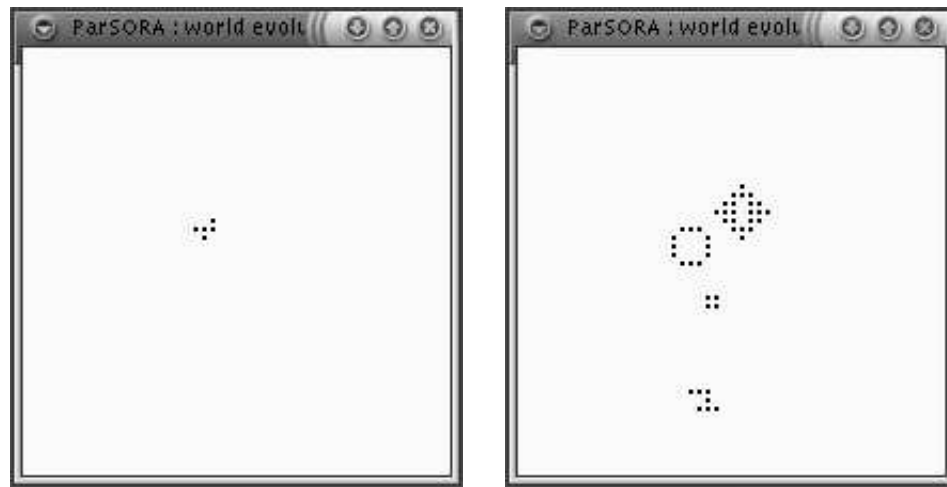
The second application we present is a multi-consumer simulator. Several people live in a town. The town contains homes where people live, shops where people buy from, and uninteresting buildings (buildings for short) which prevent people to enter them. The goal of simulation is to study the efficiency of these people to buy goods and bring them at their home, i.e. how fast they do it.

```

1  #include <stdlib.h>
2  #include "agents.h"
3
4  #define CYCLES 1024 // maximum number of cycles allowed
5
6  // cells ' behaviour function
7  static pmAction_t cell (void *mem)
8  {
9      pmAction_t act;
10     int dir, neighbours = 0; // number of neighbour cells alive
11     for (dir=UP ; dir<=DOR ; dir++)
12         if (pmlGetSquareState (dir) == OCCUP) // found a live cell
13             neighbours ++;
14     else{ // unoccupied cell => try to check if it will be born
15         int nbdir, nbneighbours = 0;
16         for (nbdir=UP ; nbdir<=DOR ; nbdir++){
17             int yrel = pmtDir2y (dir) + pmtDir2y (nbdir);
18             int xrel = pmtDir2x (dir) + pmtDir2x (nbdir);
19             if (pmlrGetSquareState (yrel, xrel) == OCCUP){
20                 if ((nbneighbours == 0) && ((yrel != 0) || (xrel != 0)))
21                     break; // cell processed by another live cell
22                 nbneighbours ++;
23             }
24         }
25         if (nbneighbours == 3) // this cell will be born
26             pmlCreateAgent (dir, 1, NORM_PR, 0, NULL, 0, cell);
27     }
28     if ((neighbours != 2) && (neighbours != 3)) // the cell will die
29         pmlKillAgent ();
30     act.type = STAY; // cells do not move
31     return act;
32 }
33
34 // function called at the end of each cycle, controlling the end of simulation
35 static int runfunc (void)
36 {
37     pmSaveChanges ();
38     if ((pmgGetCycle () >= CYCLES) || (pmgGetAgentNumber () == 0))
39         return STOP;
40     return CONT;
41 }
42
43 int main (void)
44 {
45     pmpfAgent af[] = {cell, NULL};
46     pmpfSource rf[] = {NULL};
47     // init environment
48     pmInitSystem (0);
49     pmLoadSystem ("initial-state", af, rf);
50     // start simulation
51     pmRunSystem (runfunc);
52     pmEndSystem ();
53     return 0;
54 }

```

Figure 10.2: Full code of Conway's game of life.



(a) Methuselah initial pattern.

(b) State of the world after cycle 44.

Figure 10.3: Snapshots of Conway's game of life on a world of 50×50 with a Methuselah as initial pattern.

Description of the town

In our simulations we have considered an edged environment for the town. About the size, for our explanations we have used an environment of 25×25 squares. Such a small world allows us to better introduce the system principles. On the contrary, for performance measurements we have used a more realistic case, an environment of 256×256 squares.

In this town we have created two regions: a region without buildings at the periphery of the town, containing the homes where people live, and a region with many buildings, in the centre of the town, where shops are. The resulted town is depicted in figure 10.4 on the facing page, taken with the external program.

The general file, called `multi-consumer.ini`, contains global information about the simulation (figure 10.5 on the next page). It includes the local file, called `multi-consumer-town`, containing the town configuration, i.e. all the information about homes, shops and other buildings (figure 10.6 on page 244).

Each shop contains 10 objects to be bought, simulated through a resource load of 10. Their potential is constant, and the potential propagation can be done only at the beginning of the simulation (figure 10.7 on page 245).

Description of the people

We have created three types of people:

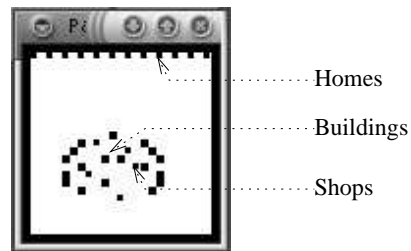


Figure 10.4: The town used in multi-consumer simulation, showing the periphery of the town with people homes (at the top), and the centre of the town with several buildings and five shops (smaller than buildings in the figure).

```

1  #number of threads used
2  nbThreads 1
3
4  #number of lines of the environment
5  dimy 25
6
7  #number of columns of the environment
8  dimx 25
9
10 #type of the environment: 0=TORUS, 1=EDGES
11 envtype 1
12
13 #connectivity
14 nbdirs 4
15
16 #local files
17 file 0 0 multi-consumer-town

```

Figure 10.5: System parameters file for multi-consumer simulation.

```

1  #WALLS
2  #      y      x
3  #buildings
4  W      11     11
5  W      12     7
6  W      12     15
7  W      13     6
8  W      13     16
9  W      14     5
10 W      14     10
11 W      14     12
12 W      14     17
13 W      15     7
14 W      15     15
15 W      16     5
16 W      17     10
17 W      16     17
18 W      17     5
19 W      17     17
20 W      18     7
21 W      18     16
22
23 #RESOURCES
24 #      y      x      type      load      initloa      pot      initpot      func_id
25 #homes
26 S      1      1      0      0      0      29      29      0
27 S      1      3      1      0      0      29      29      0
28 S      1      5      2      0      0      29      29      0
29 S      1      7      3      0      0      29      29      0
30 S      1      9      4      0      0      29      29      0
31 S      1      11     5      0      0      29      29      0
32 S      1      13     6      0      0      29      29      0
33 S      1      15     7      0      0      29      29      0
34 S      1      17     8      0      0      29      29      0
35 S      1      19     9      0      0      29      29      0
36 S      1      21     10     0      0      29      29      0
37 S      1      23     11     0      0      29      29      0
38 #shops
39 S      12     9      20     10     10     29      29      1
40 S      16     8      22     10     10     29      29      1
41 S      19     12     24     10     10     29      29      1
42 S      13     13     21     10     10     29      29      1
43 S      15     14     23     10     10     29      29      1

```

Figure 10.6: Town configuration for multi-consumer simulation.

```

23 // home potential spreading
24 static int home (int load, int initload , int initpot)
25 {
26     return initpot;
27 }
28
29 // shop potential spreading
30 static int shop (int load, int initload , int initpot)
31 {
32     return initpot; // return load * initpot / initload ;
33 }
179 pmSetResRefreshPeriod (0);

```

Figure 10.7: Resource functions for multi-consumer simulation.

1. Non-consumers, who walk throughout the town without any precise direction and without buying (they have no money). We consider them as *children*.
2. Unexperimented and deterministic consumers, who go to all the shops *in a predefined order*, buy from them and bring the goods at home. In order to find the way to shops and homes, they use the odour percept. They cannot carry all the goods at once, so they need to go several times to shops and to home. Once they have finished all the shops, they rest at home indefinitely. We consider them as *men*.
3. Experimented consumers, who act like the previous type except that if they are involved in a traffic jam, then they try other shops. We consider them as *women*.

The people behaviour and implementation are explained in detail in the following.

No person has a global perception of the town. All of them use only local perception and their memory in order to guide to shops or homes, and to buy goods.

Each person has a memory containing the “odour” of his home, his dynamic goal, the next shop where he goes, the visited shops and the doggedness (the latter is used only by women). This is implemented with the following structure:

```

15 typedef struct {
16     int homeid;           // its home: type of potential of its home
17     int goal;            // its goal: go to SHOP, HOME or REST
18     int shopid;         // next shop to go (potential type: SHOPID..SHOPID+4)
19     int visited [5];    // bool giving shops already visited
20     int doggedness;     // doggedness to a resource (0 = no doggedness)
21 }person_t;

```

Firstly, we present a pseudo-code procedure, `gotoshop`, to go to a specific resource, either home or shop, based on the potential perceived:

```
PROCEDURE gotoshop
get max potential among all neighbourhood squares
if max potential > 0
  if there are unoccupied squares with max potential
    choose randomly one of them
    move inside
  else
    wait (stay)
else
  move random
```

This procedure tries to go to a specific resource by finding the square with maximum potential $p > 0$. If there is no such square, it moves randomly. If there are such squares and all of them are occupied, it waits for them to be freed. Elsewhere it chooses randomly one of the available squares with maximum potential. Its implementation with our library is given in figure 10.8 on the facing page.

Men's behaviour is to go to resources in a predefined order, buy until they become full and bring the goods to home. They repeat this behaviour until they have bought from all the shops. Afterwards, they rest at home. Here is their behaviour in pseudo-code:

```
MAN:
if goal==shop
  if he is in the current shop
    buy
    mark shop as visited
    next shop becomes current shop
    if he is full or has finished all shops
      goal=home
  else
    gotoshop current shop
else if goal==home
  if he is at home
    leave goods
    if he has finished all shops
      goal=rest
  else
    goal=shop
else
  gotoshop home
else {goal==rest}
  rest (do nothing)
```

Its implementation with our library is given in figure 10.9 on page 248.

Women's behaviour is identical to that of men, except that they can change the order of shops. In fact, if they have unsuccessfully tried a few times to go towards a shop (they have been blocked in a traffic jam), then

```

35  /* follow potential of the resource of given type and return the new move:
36  * if it finds the way to the shop, then
37  * find all the squares which bring it nearer to the shop
38  * if there are such squares and it can go inside one (they are not occupied)
39  * choose *randomly* among them
40  * else wait for the other agents to free the place
41  * else move randomly
42  */
43  static pmAction_t gotoshop (int shopid)
44  {
45      int dir, maxpot = 0;
46      int move[4], count = 0;
47      pmAction_t act;
48      // get squares with max potential where it can go
49      for (dir=0 ; dir<4 ; dir++){
50          int pot = pmlGetSquarePotential (shopid, dir);
51          if (maxpot < pot){ // found a shorter way to resource
52              maxpot = pot;
53              count = 0;
54          }
55          if ((maxpot <= pot) && pmtdCanMove (dir)) // unoccupied square => remember it
56              move[count++] = dir;
57      }
58      // do the appropriate action
59      if (maxpot > 0){ // found the way to the shop
60          if (count > 0){ // found unoccupied squares => choose one of them randomly
61              act.type = MOVE;
62              act.param = move[pmtmRand () % count];
63          }else // traffic jam, all the squares are occupied => wait
64              act.type = STAY;
65      }else // did not find the way to the shop => move randomly
66          act.type = MOVERANDOM;
67      return act;
68  }

```

Figure 10.8: Gotoshop procedure for multi-consumer simulation.


```

70 // return 1 if end of shopping, else 0
71 static int endshopping (person_t *mem)
72 {
73     int id;
74     for (id=0 ; id<5 ; id++)
75         if (! mem->visited[id])
76             return 0;
77     return 1;
78 }

88 // man's behaviour function
89 // visit shops and persist to shopping order
90 static pmAction_t man (void *memory)
91 {
92     pmAction_t act;
93     person_t *mem = (person_t *)memory;
94     if (mem->goal == SHOP){ // go shopping
95         if ((pmlmGetSquareType () == RESSQ)
96             && (pmlmGetSquareResType () == mem->shopid)){ // found the right shop => do shopping
97             act.type = TAKE; // do shopping
98             act.param = 1;
99             mem->visited[mem->shopid - SHOPID] = 1; // has visited this shop
100            mem->shopid ++; // go to next shop (in order)
101            if ((pmlmGetAgentLoad () + act.param == pmlmGetAgentMaxload ())
102                || endshopping (mem))
103                // is full or has finished the shopping => go home
104                mem->goal = HOME;
105            }else // go to the wanted shop
106                act = gotoshop (mem->shopid);
107        }else if (mem->goal == HOME){ // go home
108            if ((pmlmGetSquareType () == RESSQ)
109                && (pmlmGetSquareResType () == mem->homeid)){ // found the home => rest
110                act.type = DROP; // leave its goods at home
111                act.param = pmlmGetAgentLoad ();
112                if (endshopping (mem))
113                    mem->goal = REST; // and rest
114                else
115                    mem->goal = SHOP; // or continue shopping
116            }else // go home
117                act = gotoshop (mem->homeid);
118        }else // rest endlessly
119            act.type = STAY;
120        return act;
121    }

```

Figure 10.9: Man's behaviour for multi-consumer simulation.

```

13  #define MAXDOGGEDNESS 1 // max doggedness

123 // woman's behaviour function
124 // visit shops but do not persist in shopping order
125 static pmAction_t woman (void *memory)
126 {
127     int id;
128     pmAction_t act;
129     person_t *mem = (person_t *)memory;
130     act = man (memory); // the same as man, with one exception...
131     if ((mem->goal == SHOP) && (act.type == STAY)){ // shopping; impatient, instead of long waiting, try another shop
132         if (mem->doggedness == 0){ // do not stay
133             for (id=0 ; id<5 ; id++)
134                 if (! mem->visited[id] && (mem->shopid != id+SHOPID)){ // try other shops
135                     act = gotoshop (id+SHOPID);
136                     if (act.type != STAY){ // no longer dogged
137                         mem->doggedness = MAXDOGGEDNESS;
138                         break;
139                 }
140             }
141         }else
142             mem->doggedness --;
143     }else
144         mem->doggedness = MAXDOGGEDNESS;
145     return act;
146 }

```

Figure 10.10: Woman's behaviour for multi-consumer simulation.

they change their mind and they go towards another shop. Here is their behaviour:

```

WOMAN:
man +
if goal==shop and involved twice in a traffic jam
    try to go to another shop

```

Its implementation with our library is given in figure 10.10, and uses men's behaviour.

Children walk randomly throughout all the town, without buying anything:

```

CHILD:
random movement

```

Their implementation with our library is very simple and is presented in figure 10.11 on the next page.

People's efficiency

In each test we have created 2 children, and 10 men and women. Men and women may carry at most 3 objects at a time. We will vary the number

```

80 // child's behaviour function
81 static pmAction_t child (void *memory)
82 {
83     pmAction_t act;
84     act.type = MOVERANDOM;
85     return act;
86 }

```

Figure 10.11: Child's behaviour for multi-consumer simulation.

of men/women from 0 to 10 and we will notice its influence on the people's efficiency. By their efficiency we mean the percentage of objects brought to homes. Children do not buy anything, and a man/woman buys one object from any of the shops. The 10 men and women will therefore need to bring to homes 50 objects. As given above, the shops contain 50 objects, sufficient for all the people.

The implementation with our library of the creation of people is given in figure 10.12 on the facing page.

The end of simulation appears when all the goods have been brought at homes (50 objects) or when a maximum number of cycles (250 in our example, as shown below) has been reached. We also save people's movement during each cycle of simulation (figure 10.13 on page 252) in order to discover possible problems.

For these parameters, the people block themselves in many cases. The blocking appears in the centre of the town, because of the buildings (figure 10.14 on page 252). We present in the following table the total number of goods brought to homes as a function of the number of men and women (number of cycles needed also printed):

<i>Women</i>	<i>Men</i>	<i>Goods</i>	<i>Cycles</i>
0	10	50 (100%)	178
1	9	50 (100%)	168
2	8	13 (26%)	251
3	7	25 (50%)	251
4	6	25 (50%)	251
5	5	0 (0%)	251
6	4	0 (0%)	251
7	3	0 (0%)	251
8	2	0 (0%)	251
9	1	0 (0%)	251
10	0	0 (0%)	251

From the results presented in this table, we notice the interest of having agents able to adapt in unknown situations, since increasing the ratio of agents able to change their goal has furnished better results. This is a basic

```

161 int main (int argc, char *argv[])
162 {
163     person_t mem;
164     int i, homeid = 0, mennb = -1;
165     pmpfAgent af[] = {child, man, woman, NULL};
166     pmpfSource rf[] = {home, shop, NULL};
167     // parameter parsing
168     if (argc == 2)
169         mennb = atoi (argv[1]);
170     if ((mennb <0) || (mennb > 10)){
171         printf ("Usage:_%s_menNumber(0..10)\n", argv[0]);
172         return 1;
173     }
174     printf ("%d_men,_%d_women\n", mennb, 10-mennb);
175     // init environment
176     pmInitSystem (0);
177     if (pmLoadSystem ("multi-consumer.ini", af, rf) != OK)
178         return 1;
179     pmSetResRefreshPeriod (0);
180     pmSetPropagationMethod (ITERATIVE_ITERATIVE_DOMAIN);
181     // agent initialisation
182     mem.goal = SHOP;
183     mem.shopid = SHOPID;
184     for (i=0 ; i<5 ; i++)
185         mem.visited[i] = 0;
186     // first child
187     mem.homeid = homeid;
188     homeid ++;
189     pmPutAgent (1, 1, 1, HIGH_PR, 0, NULL, 0, child); // higher priority
190     // men
191     for (i=0 ; i<mennb ; i++){
192         mem.homeid = homeid;
193         pmPutAgent (1, 2*homeid+1, 1, NORM_PR, 3, &mem, sizeof (mem), man);
194         homeid ++;
195     }
196     // women
197     mem.doggedness = MAXDOGGEDNESS;
198     for (i=mennb ; i<10 ; i++){
199         mem.homeid = homeid;
200         pmPutAgent (1, 2*homeid+1, 1, NORM_PR, 3, &mem, sizeof (mem), woman);
201         homeid ++;
202     }
203     // second child
204     mem.homeid = homeid;
205     pmPutAgent (1, 2*homeid+1, 1, HIGH_PR, 0, NULL, 0, child); // higher priority
206     // start simulation
207     pmRunSystem (runfunc);
208     pmEndSystem ();
209     return 0;
210 }

```

Figure 10.12: People's creation for multi-consumer simulation.

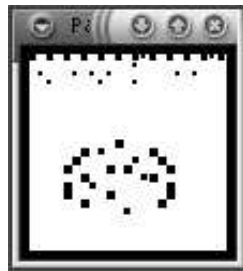
```

5  #define CYCLES 500 // maximum number of cycles allowed

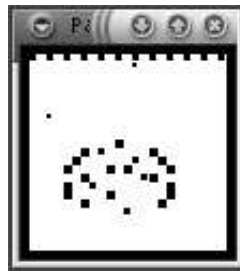
148 // function called at the end of each cycle, controlling the end of simulation
149 static int runfunc (void)
150 {
151     int i, homeload = 0;
152     // pmSaveChanges ();
153     for (i=0 ; i<12 ; i++)
154         homeload += pmgGetResTotalLoad (i);
155     // printf ("%2d ", homeload); fflush (stdout);
156     if ((homeload == 50) || (pmgGetCycle () >= CYCLES))
157         return STOP;
158     return CONT;
159 }

```

Figure 10.13: User function for multi-consumer simulation, giving the end of simulation.



(a) Position near the beginning of simulation (cycle 4), for 10 men and no woman, and for 5 men and 5 women.



(b) Final position of a successful simulation, for 10 men and no woman (note the two wandering children).



(c) Blocked position, for 5 men and 5 women (note the traffic jam in the centre).

Figure 10.14: Snapshots of the multi-consumer simulator.

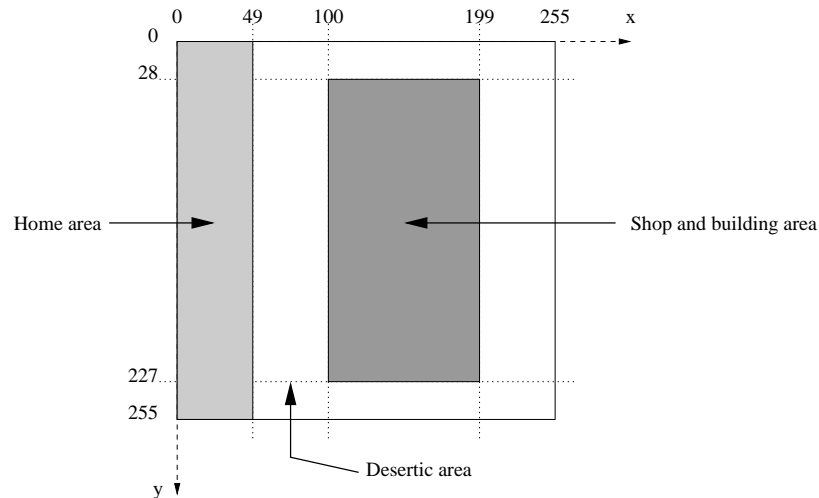


Figure 10.15: Environment of the multi-consumer simulation used for performance measurement.

observation, but a real research about agent behaviour (outside the scope of this thesis) could be done with our library in order to optimise such a multi-agent system.

Execution performance of the application

We present a basic performance measurement of this application. In our tests we have used a town discretised in 256×256 (i.e. 65536) squares. Due to our horizontal decomposition in parallel, better parallel performance is obtained when people and homes are in several domains. Therefore we have *rotated* the world by 90 degrees. The simulation results do not change. The residential region is at the left. The centre of the town is near the right, and it has been heighten. Figure 10.15 presents this new town.

In this town we have put 655 people (1% of the total squares), 655 homes (1% of the total squares), 163 shops (0.25% of the total squares) and 327 buildings (0.5% of the total squares). Among the people, 20% are men and the other 80% are women. The potential field of any resource covers all the environment. All the resources have constant potential (equal to 512), hence we have done the potential propagation only at the beginning of simulation. We have used the iterative method of potential propagation (`ITERATIVE_ITERATIVE_DOMAIN`). People and homes have been randomly put in the residential region, each person in his own home. Also, shops and buildings have been randomly put in the centre of the town. We do not have used configuration files. The number of source code lines is 240.

We have executed the simulation for 400 cycles, less than the number of cycles needed by agents to carry all the goods in their homes. On the

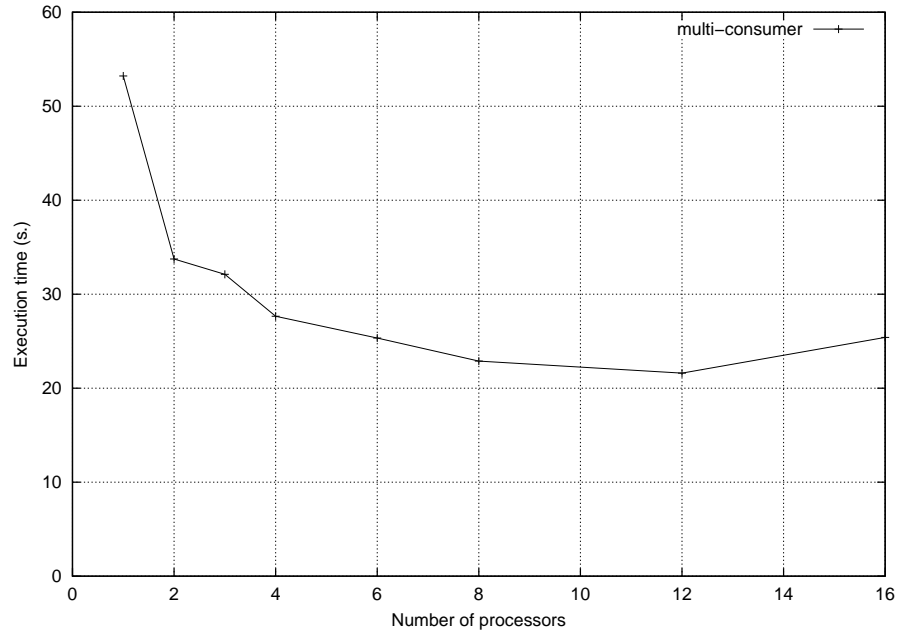


Figure 10.16: Execution time of the multi-consumer simulation, showing decreasing execution time up to 12 processors (DSM Origin 2000).

contrary, on the small size problem 250 cycles were sufficient, because agents could cross its entire world in a few cycles.

The system information is presented in section 7.10.1 on page 135. We have removed the I/O part of the application (save statistic file). We have measured the wall-clock time of the *whole* application for a number of processors ranging from 1 to 16. The tool used for measurement is the `time` command available on Unix machines.

The execution of the application on the DSM Origin 2000 machine gives the following results. Figure 10.16 presents the execution time. We notice the smallest execution time for 12 processors. The speed-up is presented in figure 10.17. It increases slowly up to 12 processors. The efficiency is presented in figure 10.18.

We have also executed it on the SMP Sun 450 machine with 4 processors. The comparative execution time on both machines is presented in figure 10.19. The comparative speed-up when executed on both machines (but different execution times) is given in figure 10.20. Compared to the DSM machine (different processors, different architectures), the execution time is higher, but both speed-ups are close.

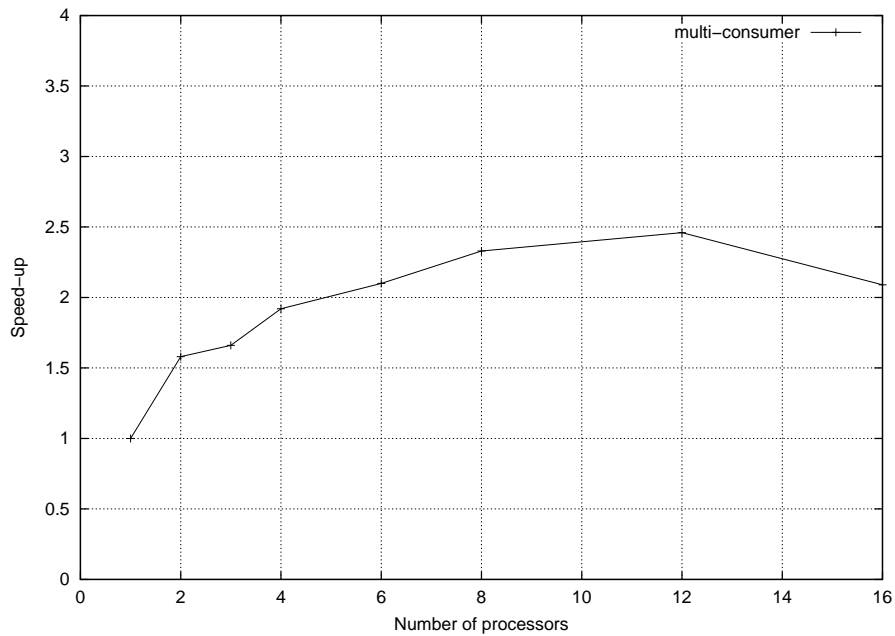


Figure 10.17: Speed-up of the multi-consumer simulation, showing slowly increasing speed-up, with a speed-up of about 2 for 4 processors (DSM Origin 2000).

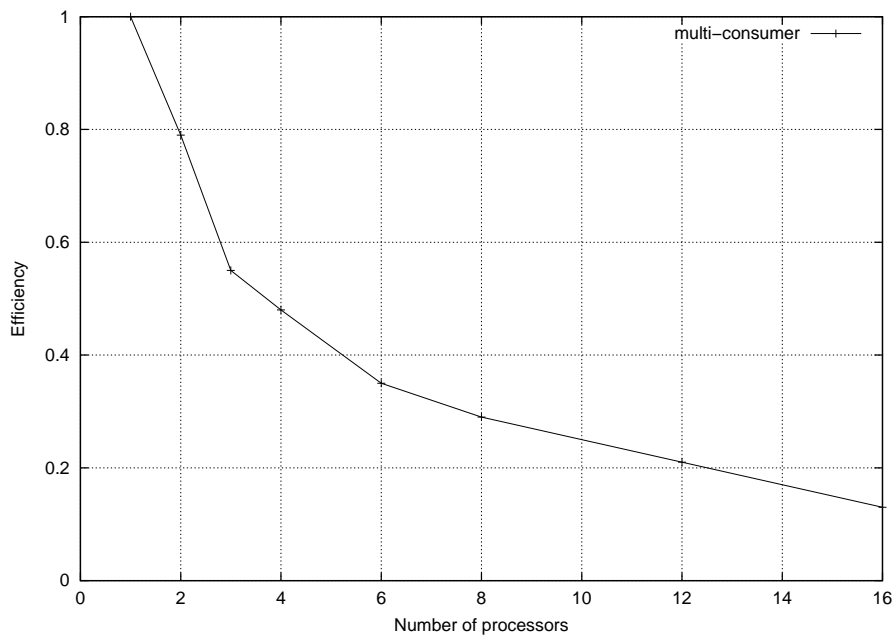


Figure 10.18: Efficiency of the multi-consumer simulation (DSM Origin 2000).

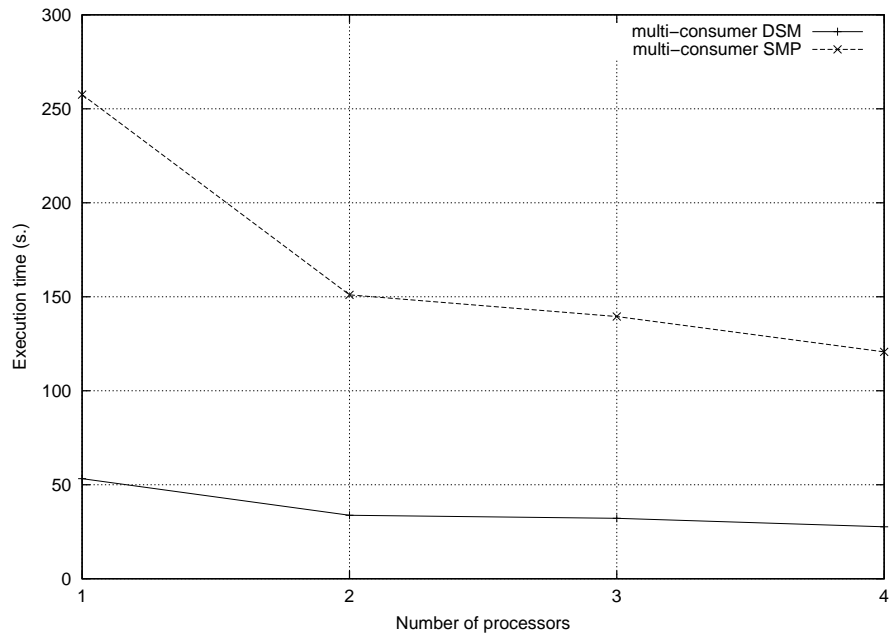


Figure 10.19: Execution time of the multi-consumer simulation, showing decreasing execution time for the SMP machine too (4 processors only).

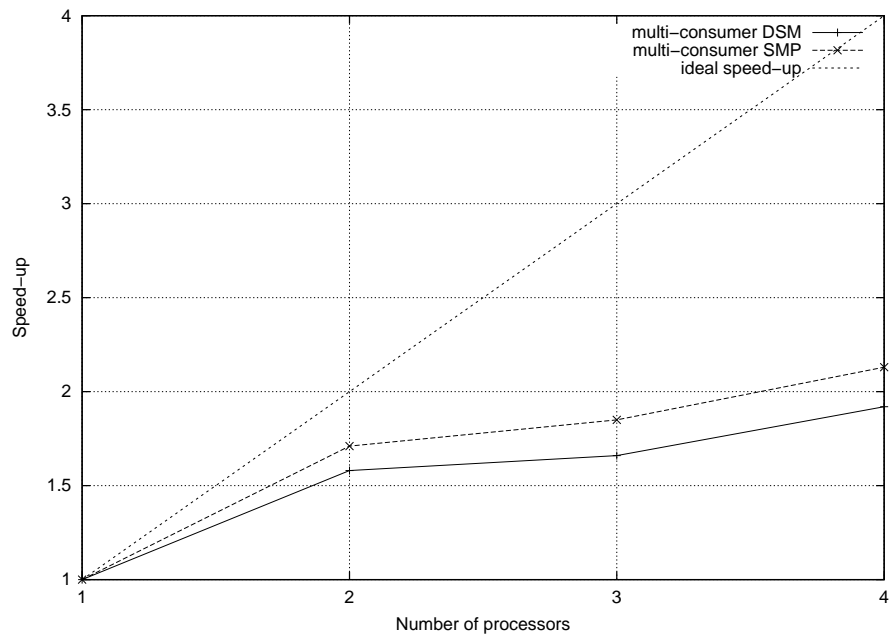


Figure 10.20: Speed-up of the multi-consumer simulation, showing close speed-ups for both the SMP machine and the DSM machine.

Conclusions

This section has presented a basic simulation of several people living in a town and doing their shoppings. Some people just walk, others buy from shops in a predefined order, while others are stubborn in a shop order for a few times only. Depending on the number of each type of people, the presented behaviours give successful or unsuccessful results. This example has shown that our library is sufficiently flexible to quickly implement and simulate such an application. All the application used in our performance measurements consists of only 240 lines of source code.

On the other hand, this application is not very well adapted to parallelisation. The potential field of any resource covers all the environment, which implies several domain propagations when executed in parallel. Moreover, there is only one resource for each potential type, and our wave propagation algorithm has not been optimised in this case. Finally, the environment is not too homogeneous: agents tend to concentrate to a region of the environment, hence decreasing the load balancing among processors. Nevertheless, we have obtained a speed-up of about 2 for 4 processors for this original application.

10.3 Conclusions

We have presented two applications written with our library. The first one is Conway's game of life, based on static cells and dynamical creation and destruction of cells. The library is sufficiently rich to allow several methods to implement this application, as described. We have chosen a method optimising the execution time.

The second application is a more complex multi-consumer simulator. It involves much more functionalities, such as people travelling, traffic jams and shopping. Its implementation is much easier than a version written from scratch, since many features, such as resource potential propagation and traffic jams, are already taken into account by the library. This application is not very appropriate to our parallelisation method, nevertheless its parallel execution time decreases, with a speed-up of 2 for 4 processors.

This chapter has shown that despite its simplicity our tool can be used in several domains of simulation. The implementation is easy and may have satisfactory parallel performance while decreasing development time.

Chapter 11

Conclusions and future work

11.1 Main results of our research

Multi-agent systems are an increasing domain of research. They allow to model many real problems. An interesting way of research is the simulation of large populations of agents, in order to discover emergence of global behaviour from simple individual behaviours (chapter 3).

In this context, this thesis has had three main contributions: it has presented a model of simulation of situated multi-agent systems, it has developed parallel algorithmic of multi-agent systems, and it has supplied an implementation of the model on parallel computers. Performance of this implementation (a C library) has been measured and is satisfactory.

Simulation model

The model, introduced in chapter 6, is destined to simulation of large populations of agents and pays special attention to agent move processing. The model has four components: environment, resources, agents and arbitrator. The environment is discrete and may contain obstacles, which prevent agents to enter some squares. Resources are entities guiding agents through the environment. They do so by propagating potential fields sensed by agents. Agents are the mobile entities of the model. They act on the environment by moving through it and they perceive it through their percepts. Agent percepts and actions are by definition local, but for flexibility the model allows any agent to use global information through specific calls. They have also memory. They may be created and destroyed dynamically, during the simulation. The fourth component, the arbitrator, maintains the coherence of the system by forbidding illegal agent moves and avoiding spatial conflicts among agents.

Statistics and information about the system can be regularly written in files and analysed by an external program. Checkpoints of the whole system are also possible.

The execution engine of the simulation is based on cycles. Each cycle is divided in several steps. This allows to deal with simultaneity of actions, such as percept data updates and spatial conflict avoiding. It also allows the user to control its application on a per-cycle basis.

Parallel algorithmic

The second contribution of this thesis consists of the work done on parallel algorithmic of multi-agent systems. We have been particularly interested in the implementation and analysis of two percepts of our agents: the vision and the detection of potential fields.

The vision percept, presented in chapter 7, is based on ray-tracing. A new algorithm of drawing supercover lines has been proposed. It is an example of regular algorithm, and its parallel performance is very high: a speed-up greater than 28 on 32 processors (an efficiency greater than 90% up to 32 processors). We have also noticed a minor load unbalancing due to domain decomposition.

The potential percept, presented in chapter 8, uses a wave propagation model based on Huygens' principle [51]. It is an irregular model, and several algorithms have been presented, both sequential and parallel. After carefully performance measurement, a comparison between the best two algorithms has been done. A combination between them has been shown to have the best performance, and it has been implemented and analysed. The parallel performance of the best algorithm obtained is acceptable: a speed-up of 17 on 32 processors (an efficiency greater than 50% up to 32 processors).

A particular attention has also been paid to allow random and *fully* reproducible simulations, regardless of the number of processors used.

Portable parallel implementation of the model

The third contribution of this thesis is the validation of our simulation model through its operational and portable parallel implementation. The library is presented in chapter 9 (features) and appendix A (reference manual). During the limited time of the thesis we have implemented the most part of our simulation model, including two percepts: vision, simulated by a visibility field, and odour detection, simulated by a wave propagation of potential.

The library also fills a real gap in multi-agent research (chapter 5), as it allows to easily simulate large populations of agents. The library focuses on simplicity of programming from the user point of view, and on performance of execution. The simplicity is provided by providing a methodology of creating simulations along with several common functionalities used in agent simulations, and by almost completely hiding the difficult parallel programming (chapter 4) from the user of the library. The performance is given by the parallelisation of our library (chapter 4) on shared-memory machines

and by a set of algorithms carefully implemented and analysed, as explained above.

Nevertheless, in order to achieve these goals, it assumes some hypotheses (chapter 2). The hypotheses are simple enough to be optimised on actual machines, and sufficient enough to allow the simulation of a broad range of applications. As our implementation of the simulation model is operational, we have used it to implement two applications, presented in chapter 10. The first is the well-known Conway game of life, based mainly on cell creation and destruction. It shows that our model and its implementation can be used to implement systems with immobile agents. The second is a multi-consumer simulator, which allows to simulate the traffic jams among people living and doing their shopping in a town. It validates our model and our library for the implementation of situated MAS simulators.

The multi-consumer application has a speed-up of about 2 on 4 processors. On the other side, we consider that the development time has been greatly reduced. For instance, the multi-consumer application consists of only 240 lines of source code.

11.2 Future work

Our future work has several directions. It involves the model enhancement, algorithm optimisation, programming environment enhancement and exploitation.

Model enhancement

Concerning our model of simulation, we plan to introduce explicit agent communication. Also, an interesting feature will be the merge of the simulated entities: agents and resources will be gathered in a single type. This would allow agents to be sources of potential, and resources to be mobile and to born and die.

Algorithm optimisation

Concerning the model implementation, we plan firstly to enhance the wave propagation model and its current implementation. In increasing order of difficulty, a feature enhancement of our implementation is to provide propagation in toroidal environments too, as specified in our model. An execution performance enhancement is to better optimise the propagation when potential fields do not change or change partially. Also, we plan to implement and analyse other methods, such as distance-based sequential methods and mutex-based parallelisation method. Finally, a complex enhancement of our model of wave propagation is to allow inexact propagation that remains acceptable from agent point of view and runs faster.

Secondly, we plan to optimise the vision model and algorithm by reducing the memory requirements in 4-connectivity. A promising approach is to implement the computing of visibility fields through a cache-based approach. We also plan to implement another method of computing visibility fields and compare it with the actual solution. It would work by creating a database with the visibility fields of all the cases where just one obstacle is involved, and afterwards computing the fields by superposing the database results that match the real configuration of obstacles.

Thirdly, at the agent level, we plan to integrate other agent percepts while keeping the simplicity and performance of the library at acceptable values. Such an example is the mark sensing (sort of pheromone) and mark taking/dropping actions.

Finally, at the simulation level, we plan to optimise its parallel performance. In this context, the most important feature is to implement a dynamic load balancing. An idea is to automatically grow or decrease the number of lines processed by each processor, based on the processing time it spent in previous cycles of simulation. With a dynamic load balancing, the parallel performance may become much greater for not so regular simulations, for example when the majority of the agents are concentrated in a region of environment.

Programming environment enhancement

Another direction of work is the creation of external graphical programs to automatically generate environments to be simulated (world initialisation), and to better visualise simulation results, such as agents' performance, by using the already implemented I/O features of the library. The checkpoints would also need to store all the information about the system, such as the memory of agents.

Exploitation

Until now we have been involved in providing this library. Nevertheless, a very important future work is the real use of this tool by artificial intelligence researchers to search for emergence of behaviour. This is planned to be done at Supélec starting from September 2002 through our graduate students' projects or Ph.D. thesis.

By providing such a tool to easily and efficiently test behaviours of large populations of agents, we hope to have successfully contributed to a part of the multi-agent research.

Appendix A

ParSSAP reference manual

This appendix presents all the structures, types and functions provided by our library. The functions have been grouped in several categories:

- Simulation start and end functions.
- Environment creation functions.
- Simulation parameters functions.
- Input/output functions.
- Agents dynamic creation and destruction functions.
- System information functions.

We present first some general considerations about the library, afterwards all the functions provided by our library and finally we explain the files processed by `ParSSAP`.

A.1 General considerations

Many functions of the `ParSSAP` library return a value. Generally, this value is `OK`, for successful return, or `ERR`, in case of error. For save/load functions, two other return values exist: `FILEERR` (error when using the file) and `FUNCERR` (the function found in file/code does not exist in code/file, see Input/output functions).

Each square of the environment has one and only one of the following types: `FREESQ` (if the square is free), `RESSQ` (if there is a resource in the square) and `WALLSQ` (if there is an obstacle in the square). It has also one of the two followings states: `OCCUP` (if there is an agent in the square) and `UNOCCUP` (if the square is not occupied by an agent).

The action an agent can do is specified in the structure `pmAction_t`.

The action of the agents may be read in order to know if during that cycle they have done the planned action or not. This can be done in two places. The first is the behaviour function of the agent, through the call to the function `pmgiGetAgentActiontype`, which gives the action done in the previous cycle of simulation. (For agents with slowness different than 1, this returns always `PASS`, as shown below, since during the previous cycle its behaviour function has not been called.) The second is the user function, through calls to the functions `pmgiGetAgentActiontype` and its variants. As the user function is executed at the end of cycle, it may be used to obtain the action for agents with slowness different than 1 too. The only part which can change the action of the agents is the conflict part. Thus:

- `MOVE` may be changed into `MOVE_CONFLICT` (if it has won in a conflict case), `STAY_CONFLICT` or `STAY_CA_CONFLICT` (if it has lost in a conflict case where the winner has been a move agent, respectively a created agent).
- `MOVERANDOM` may be changed in `MOVERANDOM_STAY` (if the agent has no square to move), `MOVERANDOM_MOVE` (if it has randomly moved), `MOVERANDOM_MOVE_CONFLICT` (if it has randomly moved and it has won a conflict), `MOVERANDOM_STAY_CONFLICT`, or `MOVERANDOM_STAY_CA_CONFLICT`. (if it has randomly planned a move, but it has lost in the conflict case where the winner has been a move agent, respectively a created agent).
- `PASS` means that the behaviour function of the agent was not called during that cycle because of its slowness.

A.2 Data structures and types

```
pmAction_t
typedef struct{
    int type;
    int param;
}pmAction_t;
```

The action of an agent.

type the type of the action. It may be `STAY`, `MOVE`, `TAKE`, `DROP`, or `MOVERANDOM`.

param the parameter associated to the type. It may be a direction (`UP`, `DOWN`, `LEFT`, `RIGHT`, `UPL`, `UPR`, `DOL`, `DOR`, in this order) if the action is `MOVE`, or the number of units taken/dropped if `TAKE/DROP`. `DOWN` is equivalent to incrementing the y coordinate, and `RIGHT` is equivalent to incrementing the x coordinate¹.

¹This information is needed in vision.

```
pmAction_t pmpfAgent (void *memory)
```

The behaviour function of an agent.

memory a pointer to the internal memory specific to the agent. This is the pointer passed as parameter during the `pmPutAgent` call.

returned value the planned action of the agent. It may be changed by the arbitrator in conflict cases, see section A.1.

```
int pml2pFunc (int load, int initload, int initpot)
```

The resource function giving its potential as a function of its load (and the initial values of its load and its potential).

load the dynamic load of the resource.

initload the initial load of the resource.

initpot the initial potential of the resource.

For any such function the following relation is true:

```
pml2pFunc (initload, initload , initpot) == initpot
```

Note: The reason to have also `initload` and `initpot` as parameters is to allow to have only one function for resources with different `initload` and/or `initpot`. See also the note in the description of function `pmPutResource`.

```
int pmpfCallback (void)
```

The user function. Called (sequentially) at the end of every cycle.

returned value `STOP` to stop the simulation, or `CONT` to continue it.

A.3 Simulation start and end functions

```
void pmInitSystem (unsigned int randomseed)
```

Initialises the system. It must be called before any other function of the library.

randomseed the global seed used for all the random number generation.

Two executions give the same result if the same value for `randomseed` is used and, instead of classical random functions, `pmAgRand` function is used.

```
void pmRunSystem (pmpfCallback userfunc)
```

Starts the simulation. It must be called when the initialisation of the environment is complete (see section A.4).

userfunc The function to be called at the end of every cycle. Here you can write the code you need to be executed every cycle. See the type `pmpfCallback` for more information.

```
void pmEndSystem (void)
```

Deallocates all the memory used by the system.

A.4 Environment creation functions

These functions must be called before starting the simulation with the `pmRunSystem` function.

```
void pmCreateEnv (int dimy, int dimx, int type)
```

Creates the environment.

dimy the dimension (number of squares) on y axis. The coordinate starts from 0 ($0..dimy-1$) and grows from top to bottom.

dimx the dimension (number of squares) on x axis. The coordinate starts from 0 ($0..dimx-1$) and grows from left to right.

type the type of the environment: **EDGES** (obstacles are added automatically on all the edges of the environment to prevent the agents to leave the environment) or **TORUS** (the last column is continued with the first column, the same for lines).

```
void pmPutObstacle (int y, int x)
```

Puts an obstacle at a precise square of the environment. This must be called after the creation of the environment. If the square is already occupied, the program aborts.

y the y coordinate of the obstacle.

x the x coordinate of the obstacle.

```
void pmPutRandomObstacles (int nb)
```

Puts randomly **nb** obstacles in the environment. This must be called after the creation of the environment. If **nb** is greater than the number of free squares, this function aborts.

nb the number of obstacles to put in the environment.

```
void pmPutAgent (int y, int x, int slowness, int prio, int maxl,  
void *mem, int memsize, pmAgFunc func)
```

Puts an agent in the environment. This must be called after the creation of the environment. There is no imposed limit on the number of agents.

y the y coordinate of the agent.

x the x coordinate of the agent.

slowness a positive number specifying the speed of the agent. An agent with slowness **n** means that its behaviour function is called every **n** cycles.

prio the priority of the agent. It may be **NORM_PR** or **HIGH_PR**. It is used only in spatial conflict cases, when the winner is chosen randomly among the high priority agents in conflict. If high priority agents do not exist, they are randomly chosen among the agents.

maxl the maximum load of the agent. Actually, an agent may carry only one type of ore at the same time.

mem pointer to an initialised area which is copied (**memsize** bytes) to the memory of the agent.

memsize the size, in bytes, of the memory of the agent.

func the behaviour function of the agent, which is called every **slowness** cycles.

```
void pmPutRandomAgents (int nb, int slowness, int prio, int  
maxl, void *mem, int memsize, pmAgFunc func)
```

Puts randomly **nb** agents in the environment with the given parameters (see the previous function for their meaning). This must be called after the creation of the environment. If **nb** is greater than the number of free squares, this function aborts.

```
void pmPutResource (int y, int x, int type, int initload, int
initpot, pml2pFunc func)
```

Puts a resource in the environment. This must be called after the creation of the environment. A square cannot contain more than one resource. If the square is already occupied by another resource, this function aborts the program. This kind of resource propagates a potential field using a wave propagation algorithm described in chapter 8. There is no imposed limit on the number of resources created.

y the y coordinate of the agent.

x the x coordinate of the agent.

type the type of the potential propagated by the resource. There is no imposed limit for the number of types. Also, the type numbers may be discontinuous (e.g. 1 and 3).

initload the initial load of the resource.

initpot the initial potential of the resource.

func function which has the load as parameter and returns the potential. It NULL, then the resource will have always the potential **initpot**.

Note: The parameter **initpot** is needed because the function **func** takes also the initial load and initial potential as parameters (see the note in the description of type **pml2pFunc**). If the **func** function had depended only on load, then the initial potential would not have need to be given as parameter here (in fact, it would have been implicitly known in **func** function).

```
void pmPutRandomResources (int nb, int type, int initload, int
initpot, fctl2p func)
```

Puts randomly **nb** resources in the environment with the given parameters (see the previous function for their meaning). This must be called after the creation of the environment. If **nb** is greater than the number of free squares, this function aborts.

A.5 Simulation configuration functions

These functions must be called before starting the simulation with the **pmRunSystem** function.

The default values of the functions represent the value which is used if that function is not called.

```
void pmSetDirsNo (int n)
```

Sets the connectivity of the system. The connectivity is taken into account by the agents (the number of directions allowed for movement) and by the wave propagation and the vision algorithms. This is used in computing the vision field and the gradient propagation fields.

n the connectivity (4 or 8). For agents movement, 4 allows only the UP, RIGHT, DOWN, and LEFT directions, while 8 allows all the them.

default 4.

```
void pmSetThreadsNo (int nbth)
```

Sets the number of threads used during the simulation. The program will create $\text{nbth}-1$ threads ($\text{nbth} \geq 1$). The simulation will be done in parallel by all the nbth threads. This is the only function dealing with the parallelism.

default 1.

```
void pmSetResRefreshPeriod (int upd)
```

Sets the period of update of the potential fields of the resources.

upd the period of update. A value of 0 means initialisation only, otherwise update every **upd** cycles (useful to speed-up the simulation).

default 1.

```
void pmSetPropagationMethod (int method)
```

Sets the method of potential propagation of resources. Actually, three sequential and two parallelisation methods are implemented. The sequential methods are: recursive with breadth-first propagation (**BREADTH**), recursive with depth-first propagation (**DEPTH**), and iterative with variable potential (**ITERATIVE**). The parallelisation methods are: fixed domain decomposition (**DOMAIN**), and private environments (**PRIVATE**). All these methods propagate exactly the potential fields, and hence yield the same potential fields. Their difference is only in execution time and memory requirements.

method The combination between sequential and parallel methods. When using domain decomposition, two sequential methods are used: one for domain propagation, the other for frontier propagation (in this order in the name below). The method may be one of the following: **BREADTH_BREADTH_DOMAIN**, **BREADTH_DEPTH_DOMAIN**, **BREADTH_ITERATIVE_DOMAIN**, **DEPTH_BREADTH_DOMAIN**, **DEPTH_DEPTH_DOMAIN**, **DEPTH_ITERATIVE_DOMAIN**, **ITERATIVE_BREADTH_DOMAIN**, **ITERATIVE_DEPTH_DOMAIN**, **ITERATIVE_ITERATIVE_DOMAIN**, **BREADTH_PRIVATE**, and **ITERATIVE_PRIVATE**.

default BREADTH_BREADTH_DOMAIN.

void pmSetVisionRadius (int vr)

Sets the vision radius. Note that its square is always seen.

vr if 0, the vision is not used in the program, leading to a lot of memory save. If greater than 0, this sets the vision radius. A vision of 1 means the agent sees the 4/8 squares (depending on the connectivity) in its close proximity.

default 0.

void pmSetVisionType (int vt)

Sets the vision type. Each square S of the environment has associated a matrix $(2vr+1) \times (2vr+1)$ with boolean elements, which says if that square is visible from the square S or not. The vision is influenced only by obstacles, so this matrix may be computed once for all at the beginning of the simulation. However, this matrix needs a lot of memory, hence it may greatly influence the execution time of the application. Greater the vision radius is, more it needs memory. The memory needed for the vision is about $dimy * dimx * 2vr^2$ bytes (the visibility relation is commutative), which, for 1024×1024 world with $rv = 16$, gives 512MB. The goal of the function `setVisionType` is to allow the user to specify if the matrix must be allocated and computed at the beginning of the simulation, or the vision fields must be computed dynamically (i.e. no storage necessary).

vt if `STATIC`, the matrix is allocated and filled at the beginning of the simulation. If `DYNAMIC`, no matrix is needed and the vision fields are computed every time they are needed.

default `DYNAMIC`.

void pmSetVisionSymmetry (int sym)

The vision relation is symmetric. If you use this property, the memory needed and the initialisation time are halved. Note that this adds a slight execution time overhead when the vision is used, since a check needs to be made to each vision read in order to access the correct half of the memory.

sym 1 to use the symmetry, 0 to not use it.

default 1.

A.6 Input/output functions

These functions must be called either before starting the simulation with the `pmRunSystem` function, or in the user function. However, the `pmInitChanges` and `pmInitStat` initialisation functions must be called before starting the simulation.

```
int pmSaveSystem (char *filename, pmpfAgent *useragfunc[], pml2fFunc
*userl2pfunc [])
```

Saves all the state of the system (checkpoint). The name of files written are `filename` for the general file, and one or more local files with name `filename.X`, where `X` is a number starting from 0. See appendix A.9 for a full-featured example of the files saved.

filename the base name of the files to save the system state in.

useragfunc a vector ended with NULL value, containing the behaviour functions of the agents.

userl2pfunc a vector ended with NULL value, containing the resource functions.

returned value OK if successful, ERR if the number of functions exceeds the current limit (100 on actual implementation), FUNCERR if no function in `useragfunc` or `userl2pfunc` sets corresponds to the index found in the local files included by file `filename`, or FILEERR for input/output or parsing errors.

```
int pmLoadSystem (char *filename, pmpfAgent *useragfunc[], pml2fFunc
*userl2pfunc [])
```

Loads a previous saved system (checkpoint). The parameters and the returned value are identical to the previous function. The parameters `useragfunc` and `userl2pfunc` must be the same for save/load of the same files. The only thing required in initialisation file is the parameters of `pmCreateEnv`, i.e. dimensions and type of the environment. See appendix A.9 for a full-featured example of the file saved.

```
void pmSetParallelSaving (int type)
```

Sets the type of the saving done by `pmSaveSystem`. In parallel, each processor saves in a different file its domain.

type 0 for a sequential saving and 1 for a parallel saving.

default 1.

int pmInitChanges (char *basename)

Specifies the name of the file to save information about the agents. See the next function for details.

basename the base name of the files to save information in. The files written are: **basename.env** for environment information, and **basename.agents** for agents information.

returned value OK if successful, ERR if the basename has too many characters.

default "Changes".

int pmSaveChanges (void)

Saves information about environment (position of the obstacles, resources), resources (their dynamic load) and agents (their position and load) for the current cycle. This information may be used by an external program to visualise the evolution of the system.

returned value OK if successful.

int pmInitStat (char *fn)

Initialises the statistic file. See the next function for details.

fn the file name of the statistic file.

returned value OK if successful, ERR if the basename has too many characters.

default "Statistics".

void pmSaveStat (void)

Saves statistic information about the current cycle in the file previously initialised with **pmInitStat**. The information saved is: the percentage of objects in each type of resources, the percentage of empty resources, the percentage of unvisited (by the agents) resources, the number of agents alive during that cycle, the number of agents in random movement, and the number of agents which are empty. See appendix A.10 for a full-featured example of this file.

A.7 Agents dynamic creation and destruction functions

These functions must be called in agent behaviour functions.

Important note: Functions in this section have generally several variants. The name of each function starts with `pm`. The third letter may be one of the following:

- `l` (local), then it is a local function, in the spirit of agent principles.
- `g` (global), then it is global.
- `t` (tool), then it is a tool function.

The fourth (and fifth sometimes) letter may be one of the following:

- `a` (absolute), then the coordinates are absolute.
- `r` (relative), then the coordinates are relative to the coordinates of the current agent.
- `d` (direction), then the coordinates are given by a parameter specifying the direction relative to the coordinates of the current agent.
- `m` (me), then the coordinates are not given, but implicitly taken as the coordinates of the current agent.

```
void pmgaCreateAgent (int y, int x, int slowness, int priority,
int maxload, void *memory, int memorysize, pmAgFunc agfunc)
```

Creates an agent *during the simulation* with the specified parameters (see the `pmPutAgent` function for a description of these parameters). If more than one agent are planned to be created in the same square, only one is created, and a warning for each agent not created is printed on the screen. This one is chosen randomly among the first eight agents appeared in conflict. In conflicts between created and move agents, the former have always higher priority than the latter, this means that if there is a conflict among a created agent and move agents, all the move agents will be forced to `STAY`.

Available also as: `pmLrCreateAgent`, `pmLdCreateAgent`, `pmLCreateAgentDist` (create an agent at a specified distance from the current agent), `pmLCreateAgentAnywhere` (create an agent at a random position in the environment).

```
void pmgiKillAgent (int id)
```

Kills, *during the simulation*, the agent with the specified identifier. The identifier becomes free, and it may be used later, when agents are created.

Available also as: `pmgaKillAgent`, `pmLrKillAgent`, `pmLdKillAgent`, `pmLmKillAgent`.

A.8 System information functions

Please see the note of the previous section concerning variants of functions.

These functions must be called either in agent behaviour functions, or in the user function. Additionally, functions involving the current agent cannot be used in the user function. Note that, while behaviour functions are called *before* doing agents' action, the user function is called *after* it.

`int pmgGetEnvDimy (void)`

Gets the dimension of the environment on y axis.

return the y dimension of the environment.

Available also as: `pmgGetEnvDimx`.

`int pmgGetCycle (void)`

Gets the current cycle.

`int pmgGetAgentNumber (void)`

Gets the number of agents in the environment.

`int pmgaGetAgentId (int y, int x)`

Gets the identifier number of the agent in square (y, x).

Available also as: `pmlrGetAgentId`, `pmldGetAgentId`, `pmlmGetAgentId`.

`int pmgiGetAgenty (int id)`

Gets the y coordinate of the agent id.

Available also as: `pmgmGetAgenty`, `pmgiGetAgentx` and `pmgmGetAgentx`.

`int pmgiGetAgentActiontype (int id)`

Get the previous action type done by the agent id. It can be used for example to find out if it has succeeded to carry out its action.

Available also as: `pmgaGetAgentActiontype`, `pmgrGetAgentActiontype`, `pmlmGetAgentActiontype`.

`int pmgiGetAgentMaxload (int id)`

Gets the maxload of the agent id.

Available also as: `pmgaGetAgentMaxload`, `pmgrGetAgentMaxload`, `pmlmGetAgentMaxload`.

```
int pmgiGetAgentLoad (int id)
```

Gets the dynamic load of the agent `id`.

Available also as: `pmgaGetAgentLoad`, `pmgrGetAgentLoad`, `pmlmGetAgentLoad`.

```
int pmgaGetSquarePotential (int type, int y, int x)
```

Gets the potential of type `type` in the square `(y, x)`.

Available also as: `pmlrGetSquarePotential`, `pmlmGetSquarePotential`.

```
int pmgaGetSquareType (int y, int x)
```

Gets the type (`FREESQ`, `RESSQ` or `WALLSQ`) of the square `(y, x)`.

Available also as: `pmlrGetSquareType`, `pmlmGetSquareType`, `pmlmGetSquareType`.

```
int pmgaGetSquareState (int y, int x)
```

Gets the state (`OCCUP` or `UNOCCUP`) of the square `(y, x)`, i.e. if there is an agent in the square or not.

Available also as: `pmlrGetSquareState`, `pmlmGetSquareState`, `pmlmGetSquareState`.

```
int pmgGetResTotalLoad (int type)
```

Gets the total load of resources of type `type`.

```
int pmgaGetSquareResType (int y, int x)
```

Gets the type of the resource in `(y, x)`.

Available also as: `pmlrGetSquareResType`, `pmlmGetSquareResType`, `pmlmGetSquareResType`.

```
int pmgaGetSquareResLoad (int y, int x)
```

Gets the load of the resource in `(y, x)`.

Available also as: `pmlrGetSquareResLoad`, `pmlmGetSquareResLoad`, `pmlmGetSquareResLoad`.

```
int pmgaaIsVisibleSquare (int y1, int x1, int y2, int x2)
```

Checks if the square `(y2, x2)` is visible by the square `(y1, x1)`. Returns 1 if yes, else 0.

Available also as: `pmgiaIsVisibleSquare`, `pmgmaIsVisibleSquare`, `pmlrIsVisibleSquare`, `pmlmIsVisibleSquare`.

`int pmtmRand (void)`

Gets a *random* and *reproducible* number. The use of this function for all random numbers guarantees that the results are fully reproducible, regardless of the number of processors. This function may be used also in the user function.

`int pmtDir2y (int dir)`

Gets the y relative coordinate (-1, 0, or 1) for the direction `dir`.

Available also as: `pmtDir2x`.

`int pmtaCanMove (int y, int x)`

Returns 1 if an agent can move in square (y, x) , else 0. An agent may move in a square if and only if the square is not an obstacle and it is not occupied by an agent. Note that it returns only if an agent can occupy the square (i.e. it does not contains an obstacle and it is unoccupied), not if the square is near the current agent.

Available also as: `pmtmrCanMove`, `pmtdCanMove`, `pmtmGetAgentRandomMove` (gets a random move for an agent).

`int pmtarCoordy (int)`

Transforms a relative y-coordinate in absolute y-coordinate.

Available also as: `pmtraCoordy`, `pmtraCoordx` and `pmtraCoordx`.

A.9 Full-featured example of system files

System files contain all the information about the system (with a few exceptions). The information is saved in two kinds of files: one general file and one or several local files. Both files are preprocessed by the C-preprocessor. Additionally, the `#` character starts a comment.

The general file stores information about the parameters of the system. Here is a full-featured example of general file:

```

1 //start with cycle number :
2 cycle 0
3
4 //number of threads used
5 nbThreads 1
6
7 #global seed (an unsigned int)
8 gseed 0
9
10 //number of lines of the environment
11 dimy 128
12
```

```

13 //number of columns of the environment
14 dimx 128
15
16 //type of the environment: 0=TORUS, 1=EDGES
17 envtype 1
18
19 //number of directions used (4 or 8)
20 nbdirs 4
21
22 //period of gradient sources updates: 0= initialization only, k=every k cycles
23 gradsrcUpdate 1
24
25 #vision type (0=STATIC, 1=DYNAMIC)
26 visiontype 1
27
28 #Changes file name
29 changesfilename bouton
30
31 //local files (ytop xtop filename) – environment coordinates start with 0
32 file 0 0 area–plain
33 file 128 0 area–plain
34 file 256 0 area–desert
35 file 0 128 area–forest
36 file 128 128 area–city
37 file 0 0 area–city
38 file 256 128 area–plain
39 file 0 256 area–mountain
40 file 128 256 area–forest
41 file 256 256 area–plain

```

The general file contains information about the cycle number, the number of threads, the global seed, the size and the type of the environment, the connectivity, the vision type and the name of statistic and trace file. Finally, the names of the local files and their coordinates relative to the environment are given. This file contains nine local files of 128x128, corresponding to an environment of 384x384.

Local files store information about resources and agents. Here is a full-featured example of local file:

```

1 // CITY AREA file, 128x128
2
3 // Resources codes
4 #define GoldMine 0
5 #define GoldFactory 1
6
7 // Load to potential function codes
8 #define Const 0
9 #define Rung 1
10
11 // Agent behaviour function codes
12 #define Worker 0
13 #define Superv 1
14

```

```

15 //The walls
16 //W   y   x
17 W    50  54
18 W    77  54
19 W    54  50
20 W    54  77
21 W    50  55
22 W    77  55
23
24 //The resources
25 //S   y   x   Type   Load   Load0   Pot   Pot0   FuncInd
26 S    10  20   GoldMine  100   100    8    8    Rung
27 S    20  40   GoldFactory 0    0     8    8    Const
28 S    70 120   GoldMine  100   100    8    8    Rung
29 S   100  80   GoldMine  100   100    8    8    Rung
30 S   110  20   GoldFactory 0    0     8    8    Const
31 S    90  10   GoldFactory 0    0     8    8    Const
32
33 //The agents
34 //A   y   x   Slownes Priorit Load   LoadMax FuncInd
35 A    11  13   1     0     0    13   Worker
36 A    13  13   1     0     0    13   Worker
37 A    15  13   1     0     0    13   Superv

```

The first lines define some words in order to facilitate the comprehension.

Any line of local file defines one of the following entities: obstacle, resource and agent. Each one starts with the corresponding letter. The information needed by walls is given in the following:

- For obstacles: its position.
- For resources: position, potential type, load, initial load, potential, initial potential², and the index of its function. The index is the element index of the table given to the `pmLoadSystem`.
- For agents: position, slowness, priority, load, maximum load and behaviour function index. The index is the element index of the table given to the `pmLoadSystem`.

A.10 Full-featured example of statistic file

Statistic file contains miscellaneous statistic information about the evolution of the system. Here is a full-featured example of statistic file:

```

1 #cycle load0 load9 empty0 empty9 init0 init9 nbAg randAg emptyAg
2 0 99.20 0.00 0.00 100.00 92.02 100.00 1310 0.00 99.01

```

²The same file format is used for both system initialisation and simulation continuation, and, while for initialisation the potential and the load are always equal to initial potential respectively to initial load (hence superfluous), for simulation continuation they are not identical. See description of the `pm12pFunc` type for why the initial potential also is needed.

3	10	92.94	0.25	0.00	97.55	44.17	97.55	1310	1.83	91.53
4	20	88.96	0.61	0.00	93.87	28.22	93.87	1310	0.00	87.02
5	30	85.28	0.92	0.00	90.80	18.40	90.80	1310	95.42	82.82
6	40	81.60	2.33	0.00	79.14	12.88	79.14	1310	6.26	80.00
7	50	78.90	3.07	0.00	73.62	8.59	73.62	50	4.00	84.00
8	60	78.53	3.13	0.00	73.62	8.59	73.62	44	52.27	75.00
9	70	78.34	3.19	0.00	73.62	8.59	73.62	40	47.50	70.00
10	80	78.22	3.25	0.00	73.62	8.59	73.62	32	9.38	65.62
11	90	78.10	3.25	0.00	73.62	8.59	73.62	13	61.54	69.23
12	100	78.10	3.31	0.00	73.62	8.59	73.62	4	25.00	100.00
13	110	78.10	3.31	0.00	73.62	8.59	73.62	2	0.00	100.00
14	120	78.10	3.31	0.00	73.62	8.59	73.62	2	100.00	100.00
15	130	78.10	3.31	0.00	73.62	8.59	73.62	1	0.00	100.00
16	140	78.10	3.31	0.00	73.62	8.59	73.62	1	0.00	100.00

The first line of the file contains information about the data. The following lines contain the data corresponding to each cycle. Columns in each line are separated by tab characters, and data is right justified.

The first column represents the cycle number. We notice that in this example the statistics have been saved for each cycle multiple of 10. The rest of the columns provide information about the system as it was at the corresponding cycle.

The following columns provide information about each type of resources. Three kinds of information are provided for each type of resource:

1. *load*: Percentage of the current load of all the resources of this type compared to the total initial load of all the resources of *all* the types.
2. *empty*: Percentage of empty resources of this type compared to the total number of resources of this type.
3. *init*: Percentage of resources of this type whose current load is identical to its initial load (if the resource function is monotonic, this corresponds to full/unvisited resources) compared to the total number of resources of this type.

In this example we have created two resources, with types 0 and 9, and columns 2–7 provide this information.

The last three columns provide information about agents. In order, the dynamical number of agents, the percentage of agents in a random movement and the percentage of empty agents.

This example of statistics file comes from a simulation where agents' goal was to carry objects of the same type from resources of type 0 (mines) to resources of type 9 (factories). Several things may be noted in this simulation. The simulation finished before the 150th cycle, because of the death of all the agents. Between the 50th and the 60th cycle most of the agents died. Agents have taken about a quarter of the objects from mines, and have successfully deposited 3% of the total number of objects in factories (the other

objects have disappeared when agents died). Finally, almost all the mines were visited, and only a quarter of the factories were visited.

Appendix B

Implementation of the supercover line tracing algorithm

Unlike ordinary line, the supercover line is formed by *all* the points the ideal line pierces. The full implementation of our algorithm for drawing supercover line in the C language is the following (the lines which are changed from Bresenham algorithm are printed in bold):

```
1 void useVisionLine (int y1, int x1, int y2, int x2)
2 {
3     int i;                // loop counter
4     int ystep, xstep;    // the step on y and x axis
5     int error;          // the error accumulated during the increment
6     int errorprev;      // the previous value of the error variable
7     int y = y1, x = x1; // the line points
8     int ddy, ddx;       // compulsory variables: the double values of dy and dx
9     int dx = x2 - x1;
10    int dy = y2 - y1;
11    POINT (y1, x1); // first point
12    // NB the last point can't be here, because of its previous point (which has to be verified)
13    if (dy < 0){
14        ystep = -1;
15        dy = -dy;
16    }else
17        ystep = 1;
18    if (dx < 0){
19        xstep = -1;
20        dx = -dx;
21    }else
22        xstep = 1;
23    ddy = 2 * dy; // work with double values for full precision
24    ddx = 2 * dx;
25    if (ddx >= ddy){ // first octant (0 <= slope <= 1)
26        // compulsory initialization (even for errorprev, needed when dx==dy)
27        errorprev = dx;
```

```

28     error = dx; // start in the middle of the square
29     for (i=0 ; i < dx ; i++){ // do not use the first point (already done)
30         x += xstep;
31         error += ddy;
32         if (error > ddx){ // increment y if AFTER the middle ( > )
33             y += ystep;
34             error -= ddx;
35             // three cases (octant == right->right-top for directions below):
36             if (error + errorprev < ddx) // bottom square also
37                 POINT (y-ystep, x);
38             else if (error + errorprev > ddx) // left square also
39                 POINT (y, x-xstep);
40             else{ // corner: bottom and left squares also
41                 POINT (y-ystep, x);
42                 POINT (y, x-xstep);
43             }
44         }
45         POINT (y, x);
46         errorprev = error;
47     }
48 }else{ // the same modifications as above
49     errorprev = dy;
50     error = dy;
51     for (i=0 ; i < dy ; i++){
52         y += ystep;
53         error += ddx;
54         if (error > ddy){
55             x += xstep;
56             error -= ddy;
57             if (error + errorprev < ddy)
58                 POINT (y, x-xstep);
59             else if (error + errorprev > ddy)
60                 POINT (y-ystep, x);
61             else{
62                 POINT (y, x-xstep);
63                 POINT (y-ystep, x);
64             }
65         }
66         POINT (y, x);
67         errorprev = error;
68     }
69 }
70 }

```

Appendix C

Publications written during this thesis

This Ph.D. thesis allowed me to publish articles in international and national conferences, presented below together with their abstract. It also allowed me to present them and to do other dissertations to researcher groups.

C.1 International conferences with lecture committee

- [27] Eugen Dedu, Stéphane Vialle, and Claude Timsit. Comparison of OpenMP and classical multi-threading parallelization for regular and irregular algorithms. In H. Fouchal and R. Y. Lee, editors, *Proceedings of Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*, pages 53–60, Reims, France, May 2000.

Abstract: The new emerging Distributed Shared Memory architecture promises to be more scalable than Symmetric Multiprocessor architecture, and leads to a regain of interest for parallel shared-memory programming paradigms. This paper compares two such important paradigms: classical multi-threading and multi-threading based on compiler directives (with OpenMP). Several implementations of regular and irregular algorithms, taken from artificial intelligence field, were made on an SGI-Origin2000 (a DSM architecture) and compared both in terms of development time and of execution time. Finally, we identify the most appropriate paradigm for each kind of algorithm.

- [28] Eugen Dedu, Stéphane Vialle, and Claude Timsit. Parallelisation of wave propagation algorithms for odour propagation in multi-agent systems. In D. Grigoras, A. Nicolau, B. Toursel, and B. Folliot, editors, *Advanced Environments, Tools and Applications for Cluster Computing (IWCC), LNCS 2326*, pages 92–102, Mangalia, Romania, Sept. 2001. NATO, Springer.

Abstract: One of the algorithms used in multi-agent systems is based on the wave propagation model. This article discusses some sequential (recursive, iterative, and based on distance) and parallel methods (frontier exchanging, domain decomposition changing, private environments, and mutex-based) to implement it. The mixing between these sequential and parallel methods is also shown, and the performance of some of them on two shared-memory parallel architectures is introduced.

- [83] Stéphane Vialle and Eugen Dedu. Long parallel algorithm design vs. quick parallel implementation. In *Proceedings of European Workshop on OpenMP (EWOMP)*, pages 145–150, Edinburgh, Scotland, UK, Sept. 2000.

Abstract: Some applications have a parallel natural algorithm well adapted to modern MIMD parallel computers, while others seem at first look to be sequential and need a parallel algorithm design. In this last case, we have to choose between doing just a parallel implementation of a poor parallel algorithm, or doing a parallel algorithmic effort before the parallel implementation. As today it is possible to quickly parallelize sequential source code using OpenMP, the temptation is great to avoid to design new parallel algorithms. This paper relates some new and previous experiences [27], and points out the difference between parallel algorithmic and parallel implementation, and some contributions of OpenMP.

C.2 National conference with lecture committee

- [26] Eugen Dedu. Bibliothèque parallèle pour l’implantation de systèmes multi-agent à composantes connexionnistes. In *Proceedings of Rencontres Francophones du Parallélisme, des Architectures et des Systèmes (RENPAR)*, pages 211–216, Besançon, France, June 2000. In French.

Abstract: 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.

Bibliography

- [1] C. Adami. *Introduction to Artificial Life*. Springer-Verlag, 1998.
- [2] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
- [3] P. E. Agre. *The Dynamic Structure of Everyday Life*. PhD thesis, Massachusetts Institute of Technology (MIT), Department of Electrical Engineering and Computer Science, USA, 1988.
- [4] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 268–272. Morgan Kaufmann, 1987.
- [5] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, second edition, 1994.
- [6] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, N.J., USA, Apr. 1967. AFIPS Press.
- [7] S. L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*, 32(2):221–251, June 1990.
- [8] É. Andres, P. Nehlig, and J. Françon. Tunnel-free supercover 3D polygons and polyhedra. *Computer Graphics Forum (Eurographics)*, 16(3):C3–C13, month 1997.
- [9] R. Bagrodia. Parallel languages for discrete-event simulation models. *IEEE Computational Science & Engineering*, 5(2):27–38, Apr.-June 1998.
- [10] H. E. Bal and D. Grune. *Programming Language Essentials*. Addison-Wesley, 1994.
- [11] S. Bandi. *Discrete Object Space Methods for Computer Animation*. PhD thesis, École Polytechnique Fédérale de Lausanne, Swiss, 1998.

- [12] Y. Boniface. *Etude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD*. PhD thesis, Université Henri Poincaré – Nancy 1, France, Oct. 2000. In French.
- [13] L. Bouton. ParMASS : un simulateur de systèmes multi-agents parallèle. Master's thesis, Université de Nancy-I, Supélec, LORIA, France, June 1998. In French.
- [14] M. Bouzid. *Contribution à la modélisation de l'interaction agent/environnement : Modélisation stochastique et simulation parallèle*. PhD thesis, Université Henri Poincaré—Nancy 1, France, Nov. 2001. In French.
- [15] M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. A stochastic model of interaction for situated agents and its parallel implementation. In *Proceedings of International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA)*, pages 26–31, Monastir, Tunisia, Mar. 2000.
- [16] M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. Parallel simulation of a stochastic agent/environment interaction model. *Integrated Computer-Aided Engineering (ICAE)*, 8(3), 2001.
- [17] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
- [18] M. Bull. A hierarchical classification of overheads in parallel programs. In I. Jelly, I. Gorton, and P. Croll, editors, *Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, Mar. 1996. Available also at <http://www.epcc.ed.ac.uk/~markb/pubs.html>.
- [19] A. Burns. *Programming In occam 2*. Addison-Wesley, 1988.
- [20] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.
- [21] A. Choudhary. Parallel I/O systems: Guest editor's introduction. *Journal of Parallel and Distributed Computing*, 17(1–2):1–3, Jan.-Feb. 1993.
- [22] R. Couturier. Couplage OpenMP/MPI: une expérimentation. In *Proceedings of Rencontres Francophones du Parallélisme, des Architectures et des Systèmes (RENPAR)*, pages 133–138, Besançon, France, June 2000. In French.
- [23] J. Cownie and S. Browne. Portable OpenMP debugging with TotalView. In *Proceedings of European Workshop on OpenMP (EWOMP)*, pages 23–27, Edinburgh, Scotland, UK, Sept. 2000.

- [24] T. Dagaëff, 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., Apr. 1997.
- [25] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan.-Mar. 1998.
- [26] E. Dedu. Bibliothèque parallèle pour l’implantation de systèmes multi-agent à composantes connexionnistes. In *Proceedings of Rencontres Francophones du Parallélisme, des Architectures et des Systèmes (REN-PAR)*, pages 211–216, Besançon, France, June 2000. In French.
- [27] E. Dedu, S. Vialle, and C. Timsit. Comparison of OpenMP and classical multi-threading parallelization for regular and irregular algorithms. In H. Fouchal and R. Y. Lee, editors, *Proceedings of Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*, pages 53–60, Reims, France, May 2000. Association for Computer and Information Science (ACIS).
- [28] E. Dedu, S. Vialle, and C. Timsit. Parallelisation of wave propagation algorithms for odour propagation in multi-agent systems. In D. Grigoras, A. Nicolau, B. Toursel, and B. Folliot, editors, *Advanced Environments, Tools and Applications for Cluster Computing (IWCC), LNCS 2326*, pages 92–102, Mangalia, Romania, Sept. 2001. NATO, Springer.
- [29] S. Descamps, S. Nussbaum, and O. Rochel. Parallel I/Os: Study on an Origin 2000 bay disk. Technical report, Supélec, Metz, France, 1999. Supervisor: Stéphane Vialle. Report available at <http://www.esse-metz.fr/~ersidp/Projects/Paralgo/Pario/Root.html>.
- [30] A. Drogoul, J. Ferber, and E. Jacopin. Pengi: Applying eco-problem-solving for behavior modelling in an abstract eco-system. In E. Mosekilde, editor, *Proceedings of European Simulation Multiconference (ESM)*, pages 337–342, Copenhagen, Denmark, 1991.
- [31] J. Ferber. *Les systèmes multi-agents. Vers une intelligence collective*. InterEditions, 1995. In French.
- [32] J. Ferber. Les systèmes multi-agents : un aperçu général. *Technique et science informatiques*, 16(8):979–1012, Oct. 1997. In French.
- [33] J. Ferber and J. P. Briot. Design of a concurrent language for distributed artificial intelligence. In *FGCS*, volume 2, pages 755–762, 1988.

- [34] 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>.
- [35] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966.
- [36] M. J. Flynn. Parallel processors were the future... and may yet be. *Computer*, 29(12):151–152, Dec. 1996.
- [37] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [38] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, Apr. 1986.
- [39] M. Gardner. The fantastic combinations of John Conway’s new solitaire game “life”. *Scientific American*, 223(4):120–123, Oct. 1970.
- [40] L. Gasser. Agents and concurrent objects. *IEEE Concurrency*, 6(4):74–77, 81, Oct.-Dec. 1998. Interview taken by Jean-Pierre Briot.
- [41] J.-M. Geib, C. Gransart, and P. Merle. *Corba: Des concepts à la pratique*. Dunod, second edition, 1999. In French.
- [42] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994. Available also at <http://www.netlib.org/pvm3/book/pvm-book.html>.
- [43] C. Germain-Renaud and J.-P. Sansonnet. *Les Ordinateurs Massivement Parallèles*. Armand Colin, 1991. In French.
- [44] A. Grujić, M. Tomašević, and V. Milutinović. A simulation study of hardware-oriented DSM approaches. *IEEE Parallel & Distributed Technology*, 4(1):74–83, Spring 1996.
- [45] Z. Guessom and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, July-Sept. 1999.
- [46] J. L. Gustafson. Reevaluating Amdahl’s law. *Communications of the ACM*, 31(5):532–533, May 1988.
- [47] B. Hayes. E pluribus unum. *American Scientist*, 87(1):10–14, Jan.-Feb. 1999. Available also at <http://www.amsci.org/amsci/issues/comsci99/compsci1999-01.html>.

- [48] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers, second edition, 1996.
- [49] M. D. Hill and J. R. Larus. Cache considerations for multiprocessor programmers. *Communications of the ACM*, 33(8):97–102, Aug. 1990.
- [50] A. Holub. *Timing Java Threads*. Apress, 2000.
- [51] C. Huygens. *Traité de la lumière*. 1678. Translated into English: *Treatise on Light*, Dover Publications, New York, 1962.
- [52] INMOS Limited. *Transputer Reference Manual*. Prentice Hall, 1988.
- [53] N. Jennings and M. Wooldridge. Applications of intelligent agents. In N. Jennings and M. Wooldridge, editors, *Agent Technology. Foundations, Applications, and Markets*. Springer, 1998.
- [54] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, Mar. 2000.
- [55] L. Kale. Programming languages for CSE: The state of the art. *IEEE Computational Science & Engineering*, 5(2):18–26, Apr.-June 1998.
- [56] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [57] L. Kipp. Application du parallélisme aux systèmes multi-agents. Master's thesis, Université de Nancy-I, Supélec, CRIN-LORIA, Sept. 1997. In French.
- [58] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.
- [59] J. Kodjabachian and J.-A. Meyer. Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. *IEEE Transactions on Neural Networks*, 9(5):796–812, Sept. 1998.
- [60] B. Lester. *The Art of Parallel Programming*. Prentice Hall, 1993.
- [61] T. G. Lewis. *Foundations of Parallel Programming. A Machine-Independent Approach*. IEEE Computer Society Press, 1993.
- [62] N. Memon, D. L. Neuhoff, and S. Shende. An analysis of some common scanning techniques for lossless image coding. *IEEE Transactions on Image Processing*, 9(11):1837–1848, Nov. 2000.
- [63] A. Milenković. Achieving high performance in bus-based shared-memory multiprocessors. *IEEE Concurrency*, 8(3):36–44, July-Sept. 2000.

- [64] B. Nichols, D. Buttlar, and J. P. Farrel. *Pthreads Programming*. O'Reilly & Associates, Sept. 1996.
- [65] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.
- [66] F. Polat and R. Alhajj. A comparative study of the Linda model with other alternative models of parallel programming. In H. Fouchal and R. Y. Lee, editors, *Proceedings of Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*, pages 61–68, Reims, France, May 2000. Association for Computer and Information Science (ACIS).
- [67] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology*, 4(2):63–79, Summer 1996.
- [68] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, Computer Science and Engineering, June 1998.
- [69] S. H. Roosta. *Parallel Processing and Parallel Algorithms. Theory and Computation*. Springer, 2000.
- [70] J. Sahuquillo and A. Pont. Splitting the data cache: A survey. *IEEE Concurrency*, 8(3):30–35, July-Sept. 2000.
- [71] C. B. Sears. The elements of cache programming style. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 283–298, Atlanta, Georgia, USA, Oct. 2000. USENIX: The Advanced Computing Systems Association. Available also at http://www.usenix.org/publications/library/proceedings/als2000/full_papers/sears/sears_html.
- [72] D. Shasha and C. Lazere. *Out of Their Minds. The Lives and Discoveries of 15 Great Computer Scientists*. Springer-Verlag, 1998.
- [73] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60(1):51–92, Mar. 1993.
- [74] J. Šilc, B. Robič, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer, 1999.
- [75] M. Sipper. The emergence of cellular computing. *Computer*, 32(7):18–26, July 1999.
- [76] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

- [77] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.
- [78] M. Tabme, J. Adibi, Y. Al-Onaizan, A. Erdem, G. A. Kaminka, S. C. Marsella, and I. Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110(2):215–239, June 1999.
- [79] D. Talia. Cellular processing tools for high-performance simulation. *IEEE Computer*, 33(9):44–52, Sept. 2000.
- [80] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.
- [81] S. Vialle. *ParCeL-1: un langage parallèle d’acteurs autonomes synchrones*. PhD thesis, Université Paris XI Orsay, France, June 1996. In French.
- [82] S. Vialle, M. Bouzid, V. Chevrier, and F. Charpillat. ParCeL-3: A parallel programming language based on concurrent cells and multiple clocks. In H. Fouchal and R. Y. Lee, editors, *Proceedings of Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*, pages 624–627, Reims, France, May 2000. Association for Computer and Information Science (ACIS).
- [83] S. Vialle and E. Dedu. Long parallel algorithm design *vs.* quick parallel implementation. In *Proceedings of European Workshop on OpenMP (EWOMP)*, pages 145–150, Edinburgh, Scotland, UK, Sept. 2000.
- [84] S. Vialle, Y. Lallement, and T. Cornu. Design and implementation of a parallel cellular language for MIMD architectures. *Computer languages*, 24(3):125–153, 1998.
- [85] H. Wang and C. Wang. Intelligent agents in the nuclear industry. *Computer*, 30(11):28–34, Nov. 1997.
- [86] G. M. Werner and M. G. Dyer. BioLand: A massively parallel simulation environment for evolving distributed forms of intelligent behavior. In H. Kitano and J. A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 316–349. MIT Press, 1994.
- [87] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1988.
- [88] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, Sept. 1993.

- [89] R. Yagel, D. Cojen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(5):19–28, Sept. 1992.

Written at Supélec, Metz campus, France, January–December 2001.

Text editor: $\text{\LaTeX} 2_{\epsilon}$.

Figure editor: xfig 3.2.

Plot editor: gnuplot 3.7p1.

Snapshot grabbing tool: gimp 1.1.17.

Discipline: computer science, parallelism.

This thesis treats about parallel simulation of multi-agent systems (MAS), in particular behaviours of situated agents: autonomous entities acting in an environment. Simulation of traffic jams, people or animal populations are some of its applications.

Nowadays, there is a lack of efficient parallel simulators of these systems, useful to reduce the execution time of high-scale simulations. In this context, our contribution is threefold: (1) provide a simulation model of high-scale MASs, called **ParSSAP**, (2) do a parallel algorithmic work in MASs, and (3) provide an implementation of this model through a parallel library.

In this dissertation we start by introducing MASs, their parallelisation issues and the state of the art in MAS simulation. Afterwards, we detail our contribution: the simulation model we have conceived, the parallel algorithmics we have designed in two agent percepts provided in our library (computing of vision fields and potential field propagation), the documentation of our library and a few applications together with their execution performance. Finally, we present the conclusions of our work.

Our model and its parallel implementation are targeted to easy utilisation and efficient execution. They may yet be enriched, nevertheless our library allows already to easily build efficient applications on modern parallel computers.

Keywords: parallelism, situated multi-agent systems, algorithmics, wave propagation, performance measurement.

Discipline : informatique, parallélisme.

Nous nous intéressons dans cette thèse à la simulation parallèle des systèmes multi-agent (SMA), plus particulièrement à des comportements d'agents situés : entités autonomes agissant dans un environnement. La simulation du trafic routier, de populations de personnes ou d'animaux sont quelques exemples de ses applications.

Actuellement, il y a un manque de simulateurs parallèles efficaces pour ces systèmes, qui seraient très utiles, compte tenu des temps d'exécution pour des simulations à grande échelle. Dans ce contexte, notre apport se divise en trois parties : (1) fournir un modèle de simulation de SMAs à grande échelle, appelé **ParSSAP**, (2) faire un travail d'algorithmique parallèle dans les SMAs et (3) fournir une implantation de ce modèle sous la forme d'une bibliothèque parallèle.

Dans cette thèse nous commençons par introduire les SMAs, les problèmes de parallélisation qu'ils posent et l'état de l'art dans la simulation des SMAs. Nous présentons ensuite nos travaux et apports : le modèle de simulation que nous avons conçu, l'algorithmique parallèle utilisée dans deux percepts d'agents fournis dans notre bibliothèque (calcul des champs de visibilité et propagation des champs de potentiel), la documentation sur notre bibliothèque et quelques applications avec leurs performances à l'exécution. Finalement, nous présentons le bilan, positif, de nos travaux.

Notre modèle et son implantation parallèle sont destinés à une utilisation facile et à des exécutions efficaces. Ils peuvent encore être enrichis, néanmoins notre bibliothèque permet déjà de construire rapidement des applications efficaces à l'exécution sur des machines parallèles modernes.

Mots-clé : parallélisme, systèmes multi-agent situés, algorithmique, propagation par vagues, mesure de performances.