UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Parallélisation d'Algorithmes Génétiques fondée sur le Modèle BSP

Agnès Braud, Christel Vrain
Université d'Orléans, LIFO

Rapport N^o **99-06**

Résumé

Nous nous intéressons à l'apport des algorithmes génétiques dans le domaine de l'Extraction de Connaissances dans les Bases de Données. Ces algorithmes réalisent une exploration stochastique de l'espace de recherche ; ils sont donc adaptés à des problèmes pour lesquels cet espace est vaste, ce qui est notamment le cas dans ce domaine d'application. Néanmoins, le processus d'évolution permettant de faire émerger une bonne solution au problème est en général très long, ce qui représente une limite importante à leur application. Des solutions ont été proposées pour diminuer les coûts d'éxécution de ces algorithmes, mais le problème reste ouvert et nous étudions ici l'une d'elles, à savoir la parallélisation.

Après une présentation des algorithmes génétiques, nous proposons trois algorithmes parallèles ne modifiant pas le fonctionnement de l'algorithme séquentiel. Ces algorithmes sont conçus selon le modèle BSP, ce qui permet d'étudier leur coût. Il est calculé dans le cadre d'une application classique d'Extraction de Connaissances dans les Bases de Données, à savoir l'apprentissage d'une description de concept à partir d'exemples stockés dans une base de données. Les résultats de cette étude théorique prédisent une accélération par rapport à l'algorithme séquentiel pour deux des trois algorithmes proposés. De plus l'un de ces algorithmes a été implanté et les expérimentations menées sur le prototype confirment ces prédictions.

Abstract

We are interested in the contribution of Genetic Algorithms in the field of Knowledge Discovery in Databases. These algorithms carry out a stochastic exploration of the search space; they are thus adapted to problems for which this space is large, which is particularly the case in this domain of application. Nevertheless, the process of evolution that makes a good solution for the problem at hand emerge is in general very long, which represents a significant limit for their usefulness to solve problems. Solutions were proposed to reduce the execution costs of these algorithms, but the problem remains open and we study one of these solutions, namely parallelization.

After a presentation of Genetic Algorithms, we propose three parallel algorithms that do not modify the behaviour of the sequential algorithm. These algorithms are designed according to the BSP model, which makes possible to study their cost. It is calculated within the framework of a classical application to Knowledge Discovery in Databases, namely Concept Learning from examples stored in a Database. Results of this theoretical study predict a speed-up in comparison with the sequential algorithm for two of the three algorithms proposed. Moreover one of these algorithms was implemented and experiments carried out on the prototype confirm these predictions.

Table des matières

1 Introduction **5**

2 Les algorithmes génétiques **6**
2.1 Présentation 6
2.2 Principes 7
2.3 Définitions 8
2.4 Les étapes d'un algorithme génétique 9
2.5 Les opérateurs génétiques 9
2.5.1 La sélection 9
2.5.2 Les opérateurs liés à la reproduction 11
2.6 Convergence d'un algorithme génétique 12
2.6.1 Définitions 12
2.6.2 Les "clés" de la convergence 13
2.7 L'exploration et l'exploitation de l'espace de recherche 14
2.8 L'algorithme séquentiel 14
2.9 Résultat théorique sur la convergence : la théorie du schéma . . . 15
2.9.1 Le concept de schéma 16
2.9.2 Théorème du schéma 16
2.9.3 Le parallélisme implicite 17
2.10 Problèmes rencontrés lors de l'utilisation d'algorithmes génétiques 18

3 Parallélisation **20**
3.1 Parallélisation des algorithmes génétiques traditionnels 20
3.1.1 La parallélisation globale 20
3.1.2 La parallélisation à gros grain 21
3.1.3 La parallélisation à grain fin 21
3.2 Le parallélisme avec le modèle BSP 22
3.2.1 Le modèle PRAM 22
3.2.2 Le modèle BSP 23
3.3 Algorithme parallèle proposé 26
3.3.1 Description de la parallélisation de l'algorithme général . . 27
3.3.2 Etude du coût moyen d'une itération de l'algorithme . . . 32

4 Application au Data Mining **35**
4.1 Apprentissage de concepts à partir d'exemples 35
4.1.1 Description de l'application 35
4.1.2 La notion de couverture 35
4.1.3 Codage des individus 36
4.1.4 Prétraitement de la base de données 36
4.1.5 Base de données des champignons 38

4.2 Coût de l'algorithme parallèle pour l'application 38
4.2.1 Evaluation du nombre d'instructions nécessaires pour calculer la qualité d'un individu 38
4.2.2 Comparaison du coût de l'algorithme parallèle avec celui de l'algorithme séquentiel 39
4.2.3 Etude de l'accélération de l'algorithme 42
4.2.4 Expérimentations 42
4.3 Etude de deux algorithmes utilisant une base de données répartie 43
4.3.1 Population centralisée et base de données répartie 43
4.3.2 Population et base de données réparties 46

5 Conclusion **47**

Bibliographie **51**

Algorithmes **53**
A L'algorithme parallèle avec répartition de la population 55
B L'algorithme parallèle avec répartition de la base de données et de la population 58
C L'algorithme parallèle avec répartition de la base de données . . . 60

1 Introduction

Les algorithmes génétiques sont des algorithmes itératifs s'inspirant des principes de la sélection naturelle (les organismes les mieux adaptés à l'environnement ont plus de chances de survivre) et de la propagation du patrimoine génétique (celui d'un nouvel organisme est un mélange des gènes de ses deux parents). Ainsi ils simulent l'évolution d'une population d'individus au fil des générations, en leur appliquant des opérateurs génétiques de sélection (grâce à une fonction permettant d'évaluer leur adaptation à l'environnement) et de reproduction (mutation et croisement).

Ils se révèlent être des techniques efficaces pour la résolution d'un certain nombre de problèmes (des problèmes d'optimisation, par exemple), mais ils sont très coûteux en temps. Il est donc très intéressant d'essayer de les paralléliser pour réduire leur temps d'exécution. Ainsi, de nombreux travaux ont déjà porté sur la parallélisation de ces algorithmes, et il s'en dégage trois approches. La première de ces approches est la parallélisation globale de l'algorithme : une seule population évolue, un processeur maître alloue des tâches à un ensemble de processeurs esclaves, afin de réaliser l'évaluation des individus et l'application des opérateurs génétiques de reproduction en parallèle. Cette méthode préserve le fonctionnement de l'algorithme séquentiel, car la sélection des meilleurs individus prend en compte toute la population et chaque individu a une chance de se reproduire avec n'importe quel autre. La seconde méthode est la parallélisation à gros grain, dans laquelle plusieurs populations évoluent indépendamment la plupart du temps, échangeant occasionnellement un ou plusieurs individus sélectionnés selon des critères liés à leur adaptation à l'environnement. Enfin, la troisième méthode est la parallélisation à grain fin ; elle associe à la population une structure spatiale qui définit des voisinages, la sélection et la reproduction n'interviennent qu'entre individus d'un même voisinage. Les deux dernières méthodes modifient beaucoup le fonctionnement de l'algorithme séquentiel car sélection et reproduction sont alors locales.

Le travail que nous présentons ici consiste à étudier des méthodes de parallélisation d'un algorithme génétique séquentiel en utilisant le modèle BSP. Il s'agit d'un modèle de programmation parallèle, dont l'intérêt principal est de permettre de prédire théoriquement le coût d'un algorithme réalisé selon ce modèle.

L'objectif est de paralléliser l'algorithme séquentiel en ne modifiant pas son fonctionnement, afin de préserver les propriétés théoriques des algorithmes génétiques. Nous proposons ici trois méthodes : distribution de la population et des traitements, distribution de la base de données, distribution de la base de données et de la population. Ces méthodes se rapprochent de la parallélisation globale, dans la mesure où le fonctionnement de l'algorithme séquentiel est préservé. L'utilisation du modèle BSP nous a permis de comparer les performances, en terme de temps d'exécution, des algorithmes parallèles avec celles de l'algorithme séquentiel, et d'évaluer sous quelles conditions il est possible d'améliorer les temps d'exécution d'un algorithme génétique parallélisé selon nos méthodes. Nous en avons retenu une, que nous avons implantée, afin de vérifier les calculs théoriques faits avec BSP. Nous l'avons utilisée dans le cadre de l'apprentissage de concepts à partir d'exemples.

Ce rapport est organisé comme suit. Tout d'abord, nous commencerons par une présentation des algorithmes génétiques : principes, fonctionnement, propriétés et problèmes liés à leur fonctionnement. Ensuite nous présenterons les méthodes courantes de parallélisation de ces algorithmes, puis une première méthode que nous proposons en commençant par une présentation du modèle BSP sur lequel elle est basée. Puis, nous décrirons l'application choisie et nous étudierons les performances de notre algorithme pour cette application ; enfin nous proposerons deux autres méthodes de parallélisation utilisant une base de données répartie.

2 Les algorithmes génétiques

2.1 Présentation

L'évolution biologique a engendré des systèmes vivants autonomes extrêmement compliqués qui peuvent résoudre des problèmes très difficiles, tels que l'adaptation continuelle à un environnement complexe, incertain et en constante transformation. La grande variété des situations auxquelles la vie s'est adaptée laisse penser que le processus de l'Evolution est robuste, c'est-à-dire capable de résoudre de nombreuses classes de problèmes. Ainsi, on peut espérer que l'utilisation en Informatique d'algorithmes inspirés de ce mécanisme permette de résoudre des problèmes complexes.

Dans les années 60-70, trois écoles modélisant l'Evolution de façon différente ont émergé indépendamment :

- les algorithmes génétiques (AG), d'abord introduits par J.H. Holland [12], puis utilisés pour la première fois pour des tâches d'optimisation par K.A. De Jong [5] ;
- la programmation évolutionniste, développée par L.G. Fogel, A.J. Owens et M.J. Walsh [7] ;
- les stratégies évolutionnistes, d'abord introduites par I. Rechenberg [15], puis développées par H.-P. Schwefel [17].

Ces trois approches font partie de la classe des algorithmes évolutionnistes. Un aperçu des motivations de ce domaine et des différences entre ces trois écoles est donné dans [18]. D'autres modélisations sont apparues, comme la programmation génétique ([13], [14]).

Les principes fondamentaux des algorithmes génétiques ont été énoncés pour la première fois en 1975 par John Holland [12]. Ils utilisent les mécanismes de la sélection naturelle et de la génétique des organismes biologiques pour résoudre des problèmes complexes, en simulant sur une population d'individus les lois de l'évolution des espèces établies par Charles Darwin : survie des individus les mieux adaptés et propagation du patrimoine génétique. L'étude de l'évolution des organismes biologiques a en effet permis de constater l'apparition de nouvelles espèces résultant d'un croisement ou de transformations rendant ces nouveaux organisnes plus résistants, alors que d'autres espèces, trop inadaptées au milieu, vont disparaître. L'idée dévéloppée dans le cadre des algorithmes génétiques est de résoudre un problème en faisant subir des transformations semblables à une population de solutions potentielles à ce problème. On peut ainsi espérer qu'une évolution similaire à celle constatée dans la nature se produise dans notre population, et nous mène à la découverte d'une solution particulièrement adaptée au problème.

Ces algorithmes peuvent être utilisés pour résoudre des problèmes d'optimisation et de recherche de solution dans un espace complexe.

2.2 Principes

Les algorithmes génétiques reposent sur une analogie directe avec l'environnement naturel : le problème traité est assimilé à un environnement, les individus évoluant dans cet environnement représentant des solutions potentielles pour le problème.

On utilise ainsi dans le domaine des algorithmes génétiques des termes empruntés à la biologie. Les caractéristiques de chaque individu sont représentées par un ensemble de paramètres (*gènes*), qui peuvent prendre des valeurs différentes (*allèles*). Un individu est représenté par une chaîne de caractères, appelée *chromosome*, rassemblant ces caractéristiques.

Les AG sont des algorithmes itératifs simulant sur une population la reproduction des individus et le mécanisme de sélection naturelle qui se traduit par le fait que les individus les mieux adaptés à l'environnement ont les plus grandes chances de survie. Pour mettre en place le mécanisme de sélection, il est nécessaire de pouvoir estimer la qualité d'un individu. Cette mesure est fonction de la capacité de l'individu à résoudre le problème. Le but de ces algorithmes est en fait d'optimiser une fonction définie par l'utilisateur, appelée *fitness* ou *fonction d'adaptation*, qui permet d'évaluer la qualité des individus, en cherchant la combinaison des allèles qui donne lieu au maximum d'adéquation.

A chaque génération, on donne aux individus les mieux adaptés à l'environnement un plus grand nombre d'opportunités d'être sélectionnés pour être présents à la génération suivante et/ou effectuer des croisements avec d'autres individus. On obtient ainsi une nouvelle population. Chaque descendant possède des caractéristiques héritées de ses parents, ce qui doit permettre d'obtenir dans la nouvelle population une plus grande proportion des caractéristiques possédées par les meilleurs individus, échangées et mélangées avec d'autres caractéristiques, et ainsi de propager de génération en génération celles qui sont susceptibles de contribuer à la bonne qualité d'un individu.

La sélection des meilleurs individus permet l'exploration des zones de l'espace de recherche les plus prometteuses pour la découverte d'une solution optimale.

Au bout d'un certain nombre de générations, on peut espérer que les individus restants soient particulièrement adaptés à l'environnement, donnant ainsi des solutions très proches de la solution optimale du problème. Si l'algorithme a été bien conçu, il y aura alors convergence de la population vers une solution optimale. Nous verrons par la suite que les deux points cruciaux dans la conception des algorithmes génétiques sont le choix de la représentation des individus (codage) et celui de la *fitness*.

Les algorithmes génétiques représentent une technique robuste pouvant traiter un grand nombre de problèmes. De tels algorithmes ne trouveront pas toujours la solution optimale, mais seront en général capables de trouver des solutions de qualité acceptable assez rapidement.

2.3 Définitions

individu : dans le cadre d'un AG traditionnel, un individu est réduit à son *génome*, c'est-à-dire un ensemble de caractéristiques élémentaires. Dans la nature, un organisme biologique est plus qu'un simple génome, c'est aussi une forme (un *phénotype*), produit de l'activité des gènes. En cela, la notion d'individu utilisée dans le domaine des algorithmes génétiques est donc plus simple que celle d'organisme rencontrée en biologie.

gène : caractéristique d'un individu, et donc d'une solution potentielle au problème que traite l'AG. Ces caractéristiques pourront prendre différentes valeurs appartenant à un alphabet qui dépend du problème à résoudre.

génome : collection complète des gènes d'un organisme.

chromosome : il s'agit d'une suite de symboles, généralement de longueur fixe, codant les informations génétiques d'un individu (son génome). En général, un individu sera formé d'un seul ensemble de gènes, mais il peut arriver qu'il soit composé de deux (on parle alors de chromosome *diploïde*) ou plus.

population : ensemble de n individus. En faisant travailler les algorithmes génétiques sur une population d'individus plutôt que sur un seul, on effectue des recherches sur un domaine de possibilités plus important.

génération : population à un instant t, c'est-à-dire la population résultant d'une itération de l'algorithme génétique.

fonction d'adaptation ou de qualité (fitness) : fonction qui mesure l'adaptation d'un individu d'une population donnée à l'environnement, c'est-à-dire sa capacité à résoudre le problème. Il est important de définir une bonne fonction d'adaptation pour un problème donné, qui représente bien la qualité des solutions et permette d'évoluer vers la meilleure d'entre elles ou tout au moins une de qualité acceptable.

2.4 Les étapes d'un algorithme génétique

La figure 1 présente le déroulement d'un algorithme génétique. On distingue trois grandes phases dans le processus d'évolution de cet algorithme :
- une phase d'initialisation pendant laquelle est générée la population initiale ;
- une phase de sélection de bons individus qui simule le principe de survie des individus les mieux adaptés ;
- une phase pendant laquelle des opérateurs génétiques (mutation et croisement) peuvent être appliqués aux individus sélectionnés, simulant la reproduction et la propagation du patrimoine génétique.

Les deux dernières phases sont répétées tant que la population n'a pas convergé (test de terminaison).

2.5 Les opérateurs génétiques

2.5.1 La sélection

La sélection est la tâche qui consiste à allouer à chaque individu des opportunités de se reproduire. Généralement les individus sont copiés dans une *mating pool*, le nombre de copies de chacun étant fonction de sa qualité. C'est ensuite dans cette *mating pool* que seront pris au hasard les couples d'individus qui, après l'application éventuelle des opérateurs de croisement et de mutation, donneront naissance aux individus de la nouvelle génération.

On peut classer les techniques de sélection en deux catégories :
- les techniques qui consistent à réévaluer pour chaque individu la fonction que l'on souhaite optimiser sur la base d'une nouvelle échelle. La valeur déterminée par cette méthode correspond au nombre de copies de l'individu à mettre dans la *mating pool*. Parmi les plus employées, on peut citer notamment la méthode appelée **"fitness ranking"** dans laquelle les individus sont classés en fonction de leur qualité, de façon à leur assigner une valeur proportionnelle à leur rang. Cette méthode permet de limiter les écarts

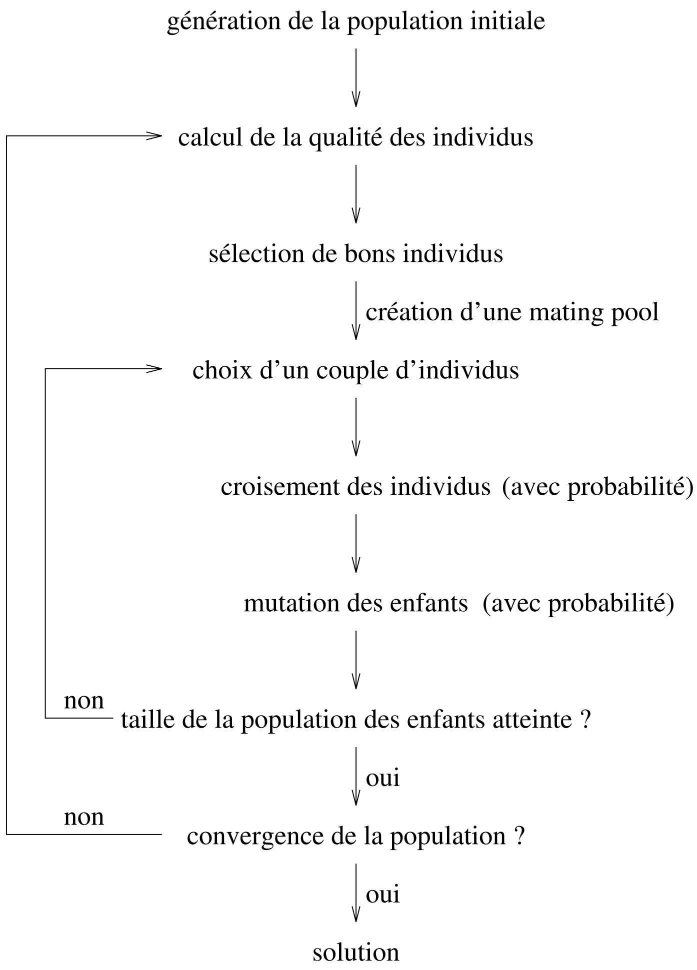

FIG. 1 – *Déroulement de l'algorithme génétique*

entre les qualités d'individus de rangs voisins (ces écarts seront réguliers), et donc de maîtriser les effets produits par la présence d'individus de trop bonne ou trop mauvaise qualité. On évite ainsi en partie les problèmes rencontrés lors de la mise en oeuvre des algorithmes génétiques et qui seront présentés dans la section 2.10. La probabilité de sélection pour un individu est égale à la nouvelle valeur qui lui a été attribuée divisée par la somme des valeurs assignées à chacun des individus de la population ;

- les techniques qui permettent de remplir la *mating pool* sans passer par l'étape intermédiaire de réévaluation de la fonction de qualité. On peut citer notamment la **sélection par tournoi** qui consiste, dans le cas le plus simple, à tirer au hasard deux individus et à copier dans la *mating pool* celui dont la qualité est la plus élevée. Il est possible de modifier cette technique en tirant un plus grand nombre d'individus et en ne copiant celui de meilleure qualité qu'avec une certaine probabilité ; d'autres variantes

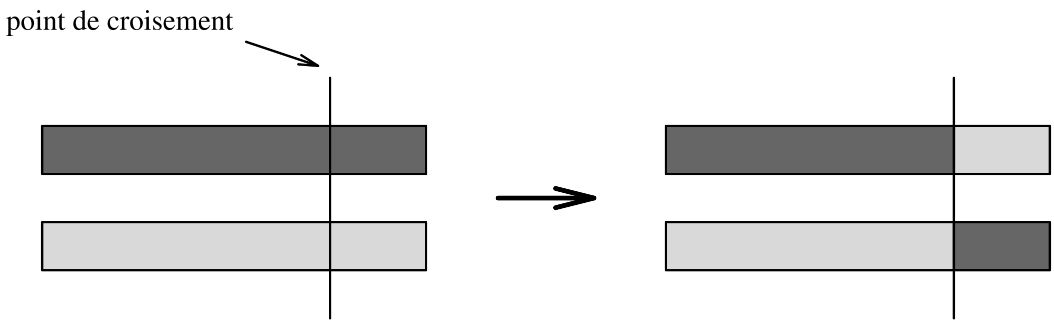

FIG. 2 – *Croisement "un point"*

reposent sur le fait de replacer ou non les individus tirés.

2.5.2 Les opérateurs liés à la reproduction

2.5.2.1 Le croisement. Cet opérateur combine les caractéristiques de deux individus pour en obtenir deux nouveaux. Soient deux individus représentés par leur chromosome, l'opération se déroule en deux étapes (voir figure 2) :

- choix aléatoire d'un point de coupure identique sur les deux chromosomes ;
- coupure des deux chromosomes et échange des deux fragments situés à droite.

Dans l'algorithme génétique, le croisement symbolise la reproduction et correspond à une recherche globale par combinaison de "blocs de construction". Les deux individus participant à cette opération sont assimilés aux parents et les deux individus résultants aux descendants.

Le croisement ne s'applique pas systématiquement à tous les individus sélectionnés pour la phase de reproduction, il est réalisé avec une probabilité qui est en général égale à 0.6. Si le croisement n'est pas fait entre deux individus, les descendants sont produits directement par duplication des parents. Ce croisement avec probabilité donne une chance à chaque individu de passer d'une génération à une autre sans modification de son patrimoine génétique.

Remarque : la technique présentée ci-dessus décrit en fait le croisement "un point" ou croisement simple, c'est la plus simple et la plus classique des techniques de croisement. Il est en effet possible de réaliser un croisement en plusieurs points, c'est le cas par exemple du croisement "deux points" et du croisement uniforme.

2.5.2.2 La mutation. C'est le changement aléatoire d'une (ou plusieurs) des caractéristiques d'un individu (voir figure 3). Cet opérateur s'applique avec une probabilité assez faible sur un individu. Si la probabilité était trop élevée, cela reviendrait à une recherche au hasard dans l'espace, ce qui ferait perdre à l'algo-

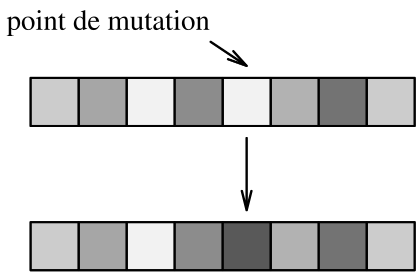

FIG. 3 – *Mutation simple*

rithme génétique son efficacité.

La mutation assure une recherche locale aléatoire autour de chaque individu. Les actions combinées de la sélection et du croisement peuvent faire disparaître certains gènes et rendre l'opérateur de croisement moins efficace avec le temps, car les individus deviennent similaires et la population n'est plus en mesure d'évoluer. Nous verrons dans la section 2.10 qu'une telle situation peut poser des problèmes. Le phénomène de mutation prend alors toute son importance, car il est capable de réintroduire ces gènes dans la population et évite ainsi une perte irréparable de diversité, permettant d'assurer qu'aucun point de l'espace n'a une probabilité 0 d'être exploré. Ces mutations ne créent généralement pas de meilleures solutions au problème, mais elles évitent l'établissement de populations uniformes incapables d'évoluer.

2.6 Convergence d'un algorithme génétique

2.6.1 Définitions

- On parle de **convergence d'un gène** si 95% de la population possède ce gène.
- On parle de **convergence d'une population** si tous les gènes ont convergé.

Si l'algorithme génétique a été correctement implanté, la population évoluera au fur et à mesure des générations de telle sorte que la qualité moyenne des individus et celle du meilleur augmenteront vers le maximum global. La convergence est une progression vers l'uniformité de la population, les meilleures caractéristiques auront en effet tendance à apparaître de plus en plus dans les individus alors que les mauvaises auront tendance à disparaître.

2.6.2 Les “clés” de la convergence

2.6.2.1 Le choix du codage des individus. Un des problèmes importants dans l'implantation des AG concerne le codage des chromosomes. En effet, comment coder les caractéristiques d'une solution potentielle au problème posé de façon adaptée au traitement par un AG ?

Plusieurs difficultés se présentent dans le choix du codage. Celui-ci doit être adapté au problème, de façon à limiter au mieux la taille de l'espace de recherche et aussi de façon à ce que les individus engendrés par application des opérateurs génétiques représentent des solutions valides compte tenu des contraintes du problème. D'autre part, il doit être tel que les opérateurs de recherche soient suffisamment efficaces pour produire les solutions recherchées avec une bonne probabilité et les détruire avec une faible probabilité.

Initialement, ces solutions ont été codées sous forme de chaînes de longueur fixe sur l'alphabet $\{0,1\}$. On parle alors de codage binaire. Cependant, d'autres formes de codage sont apparues, qui peuvent se révéler plus adaptées à la résolution de certains types de problèmes.

Par exemple, le codage binaire apparaît comme peu adapté à la résolution de problèmes dont l'espace de recherche est continu et de grande dimension, et pour lesquels on souhaite une précision numérique importante (c'est le cas pour des problèmes d'optimisation). En effet, les algorithmes génétiques procèdent de telle façon que les bits les moins significatifs seraient traités comme les bits les plus significatifs, gaspillant ainsi des efforts sur des bits dont la valeur dépendra des bits les plus significatifs. Par conséquent, pendant la manipulation des bits les plus significatifs, celle de ceux qui le sont le moins est inutile ; de même une fois que les bits les plus significatifs ont convergé, il devient inutile de revenir dessus. Il est donc plus judicieux de choisir un codage avec des nombres réels. Le chromosome est alors un vecteur de nombres réels.

On pourra ainsi utiliser des alphabets de plus ou moins grande cardinalité. Les algorithmes génétiques utilisent généralement une représentation binaire.

2.6.2.2 Le choix de la fonction d'évaluation. Comme elle conduit à l'élimination d'un certain nombre d'individus, elle constitue également un des aspects cruciaux des algorithmes génétiques et il est très important qu'elle soit bien adaptée à la résolution du problème. Ainsi, si on souhaite qu'un algorithme génétique converge et conduise à une solution de qualité satisfaisante, il faut trouver une fonction de qualité qui n'ait pas trop de maxima locaux et qui évite une convergence prématurée vers un maximum local.

2.7 L'exploration et l'exploitation de l'espace de recherche

Un algorithme d'optimisation efficace doit utiliser deux techniques pour trouver un maximum global : l'*exploration* pour étudier des zones de l'espace de recherche inconnues et regarder si des points intéressants s'y trouvent, et l'*exploitation* pour utiliser les connaissances acquises en des points de l'espace déjà visités afin de déterminer quelles zones il serait profitable d'étudier par la suite. Ces deux besoins sont contradictoires et un bon algorithme doit faire un compromis entre les deux.

Holland [12] a montré qu'un algorithme génétique combine ces deux techniques de manière optimale. D'un côté, les opérateurs de sélection et de croisement favorisent l'exploitation. En effet, une population d'individus représente un ensemble de points de l'espace de recherche, et parmi ces individus, la sélection retient ceux qui sont susceptibles de posséder de bons gènes, ces bons gènes pouvant apparaître dans une solution optimale. Le croisement permet alors de combiner des blocs de gènes de deux de ces individus, de manière à essayer de regrouper en un seul individu les bons gènes de chacun. D'un autre côté, en modifiant des caractéristiques de certains individus, l'opérateur de mutation permet d'orienter la recherche des AG vers des zones de l'espace de recherche qui n'ont peut-être pas encore été étudiées et favorise ainsi l'exploration de l'espace.

Cependant ceci est vrai en théorie, mais pose quelques problèmes en pratique, puisque Holland a fait un certain nombre de suppositions :

1. la population est de taille infinie ;
2. la fonction de qualité représente exactement l'utilité d'une solution ;
3. les gènes d'un chromosome n'interagissent pas de façon significative.

La première de ces suppositions ne peut jamais être satisfaite dans la pratique, les deux autres le seront très difficilement dans le cas de problèmes réels.

2.8 L'algorithme séquentiel

Nous avons choisi d'utiliser une *mating pool*, qui regroupe des individus sélectionnés pour leur bonne qualité et susceptibles de participer à la phase de reproduction. Lors de la phase de reproduction, la sélection est déjà faite et il suffit alors de piocher au hasard des couples d'individus dans cette *mating pool*. D'autres algorithmes n'en utilisent pas et sélectionnent, au moment de la reproduction, les couples d'individus dans la population de parents en fonction de leur qualité . Son utilisation permet de bien séparer les étapes, ce qui nous sera utile lors de la parallélisation.

Pour remplir cette *mating pool*, nous avons choisi d'utiliser la technique de sélection par tournoi entre deux individus. C'est une technique simple et nous verrons qu'elle se prête bien à la parallélisation de l'algorithme.

L'algorithme est le suivant :

début
engendrer la population initiale P_0 de taille n
calculer la qualité de chaque individu
terminé ← **faux**

tant que non terminé **faire**
début /* *produire une nouvelle génération* */

/* *phase de sélection: remplissage de la mating pool* */
/* *par la technique des tournois* */
pour i de 1 à n **faire**
début
tirer deux individus aléatoirement
mettre dans la *mating pool* l'individu dont la qualité
est la plus élevée
fin

/* *phase de reproduction* */
pour i de 1 à $\frac{n}{2}$ **faire**
début /* *cycle de reproduction* */
sélection aléatoire de 2 individus + point de croisement
croisement réalisé avec probabilité p_c (entre 0.6 et 1)
mutation réalisée sur les nouveaux individus
avec probabilité p_m (0.001)
calculer la qualité des descendants
fin

si la population a convergé **alors**
terminé ← **vrai**
fin si
fin tant que

fin

2.9 Résultat théorique sur la convergence : la théorie du schéma

Aucune théorie générale n'explique les propriétés des algorithmes génétiques, cependant plusieurs hypothèses ont été formulées qui peuvent expliquer partiellement leur succès. Le **théorème du schéma** proposé par Holland [12] est la première explication rigoureuse de leur fonctionnement.

2.9.1 Le concept de schéma

Etant donné un alphabet $\mathcal{A}$ utilisé pour le codage des individus, un *schéma* est une chaîne de caractères définie sur l'alphabet $\mathcal{A} \cup \{*\}$, où l'étoile est un symbole non spécifié qui peut être remplacé par n'importe quel symbole de l'alphabet $\mathcal{A}$. Un schéma peut être vu comme un motif décrivant un sous-ensemble de chaînes ayant des caractéristiques similaires en certaines positions.

Un chromosome particulier est une instance d'un schéma si pour toutes les positions définies du schéma (positions pour lesquelles la valeur associée n'est pas l'étoile), la valeur du chromosome en ces positions est la même que pour le schéma.

On dit qu'un chromosome contient un schéma s'il en est instance.

Considérons par exemple l'alphabet binaire $\mathcal{A} = \{0,1\}$ alors :
- Le chromosome (1010) contient entre autres les schémas (10**), (*0*0), (**1*) et (101*).
- Le schéma (*11*010) a comme instances les chromosomes : {(0110010), (1110010), (0111010), (1111010)}.

Remarque :
La qualité d'un schéma est définie comme la moyenne des qualités des individus qui sont des instances de celui-ci dans la population. Ainsi la qualité d'un individu contribue directement à la qualité des schémas dont il est instance. Notons l la longueur d'un chromosome (c'est-à-dire son nombre de gènes), la qualité d'un individu donne donc une information sur la qualité des 2^l schémas dont il est instance.

Réciproquement, plus un individu contient de bons schémas, plus il est probable qu'il ait une qualité élevée.

2.9.2 Théorème du schéma

Théorème (Holland) : les meilleurs schémas observés (schémas qui possèdent au moins une instance dans la population) auront en moyenne un nombre croissant exponentiellement d'instances dans la génération suivante.

En effet, la phase de sélection qui débute chaque itération fait qu'un individu a d'autant plus de chances de participer souvent à la phase de reproduction qu'il a une qualité élevée. On a donc plus de chances de retrouver ses gènes dans la population suivante. De plus on a vu qu'un individu qui est instance de bons schémas a lui-même de grandes chances d'être de bonne qualité. En reproduisant les meilleurs individus, on propage donc les meilleurs schémas et on retrouve un plus grand nombre de ces schémas à la génération suivante. Ainsi on augmente les chances de trouver de bonnes solutions par recombinaison de blocs de chromosomes de bonne qualité.

L'exemple suivant (tableau 1), issu de [10], montre l'évolution d'une population d'individus, codés sur l'alphabet binaire, d'une génération à une autre. Il considère le processus de traitement d'un algorithme génétique sur les individus eux-mêmes, puis sur des schémas qui ont des instances dans la population (tableau 2). La fonction que l'on cherche à optimiser est $f(x) = x^2$. Le mécanisme de sélection est celui de la sélection proportionnelle, où la probabilité pour un individu d'être sélectionné est égale à sa qualité divisée par la somme des qualités de tous les individus de la population, associée à un tirage aléatoire avec replacement des individus tirés. Les paramètres de l'AG sont les suivants : $p_c = 0.5$, $p_m = 0.001$ et $n = 4$. Les chromosomes sont des chaînes de 0 et de 1 de longueur 4, représentant les nombres entre 0 et 15 selon le codage en base 2.

Mécanisme de sélection				Recombinaison		
Population	Fitness	Probabilité de sélection	Copies obtenues	Population intermédiaire	Nouvelle population	Fitness
0110	36	0.19	1	01|10	0101	25
0010	4	0.02	0	10|01	1010	100
1001	81	0.43	2	1001	1001	81
1000	64	0.34	1	1000	1000	64
Moins bon	4					25
Moyen	46.25					67.5
Meilleur	81					100

TAB. 1 – *Exemple d'itération d'un algorithme génétique avec codage binaire*

Le tableau 1 nous montre le pouvoir des AG, par la nette augmentation des qualités du meilleur individu, du moins bon et de la qualité moyenne de la population d'une génération à l'autre, et met en évidence le fait qu'un plus grand nombre de copies est attribué aux individus de meilleure qualité.

Le tableau 2 illustre ensuite l'effet produit par la même itération de l'AG sur des schémas présents dans la population. On peut ici vérifier le bien fondé du théorème du schéma : le nombre d'instances des meilleurs schémas observés a tendance à augmenter, alors que celui des moins bons diminue.

2.9.3 Le parallélisme implicite

Bien que l'AG semble seulement traiter les individus présents dans la génération courante, il traite en fait, en parallèle, un grand nombre d'informations utiles concernant les schémas non visibles. Chaque individu contenant un grand

Schémas	Nombres d'instances à la génération t	Qualité	Nombre d'instances à la génération $t+1$	Qualité
1***	2	72.5	3	81.66
01**	1	36	1	25
1**1	1	81	1	81
0**0	2	20	0	

TAB. 2 – *Traitement des schémas*

nombre de schémas, le nombre de schémas traités à chaque génération est très important. Cette interprétation revient en fait à considérer qu'un AG traite des schémas plutôt que les chromosomes.

2.10 Problèmes rencontrés lors de l'utilisation d'algorithmes génétiques

Les individus de la population initiale étant choisis aléatoirement, il existe en général au début de l'exécution d'un algorithme génétique un large éventail d'individus de qualité différente. Cependant, au fur et à mesure de l'exécution, des valeurs particulières vont avoir tendance à prédominer pour certains gènes. Ainsi la population va converger et l'écart entre les qualités des individus se réduire. Cette réduction de l'intervalle des qualités pendant l'exécution conduit souvent aux problèmes de convergence prématurée ou de terminaison lente.

- **convergence prématurée :** le processus de recherche se retrouve coincé dans un optimum local avant d'avoir trouvé l'optimum global ;
- **terminaison lente :** au bout d'un certain nombre de générations, la population a largement convergé, mais n'a toujours pas localisé le maximum global. La qualité moyenne est élevée et il y a peu de différence entre le meilleur individu et les individus moyens. Par conséquent, l'inclinaison de la fonction de qualité est insuffisante pour amener l'AG vers le maximum global.

D'autres types de problèmes pouvant être rencontrés sont liés à l'interaction des gènes dans le chromosome.

- **"epistasis" :** c'est l'interaction entre différents gènes à l'intérieur d'un chromosome, c'est-à-dire que la contribution d'un gène à la fonction de qualité dépend de la valeur d'autres gènes du chromosome. Le degré d'interaction sera, en général, différent pour chaque gène ;
- **problèmes "trompeurs"** *(deception)* **:** la structure de l'espace de recherche empêche l'AG de converger vers un optimum global.

  Pour illustrer ce type de problème, prenons l'exemple proposé dans [10]. Considérons le tableau 3. Soit f la fonction de qualité et ϵ un schéma, $f(\epsilon)$

Chromosomes	Fitness
000	28
001	26
010	22
100	14
110	0
011	0
101	0
111	30

Schémas	Performances
$0 * *$	25.33
$0\ 0\ *$	27
$0 * 0$	25
$0\ 1\ *$	11
$0 * 1$	13
$*\ 0\ *$	22.67
$*\ 0\ 0$	21
$1\ 0\ *$	7
$*\ 0\ 1$	13
$* * 0$	21.33
$1 * 0$	7
$*\ 1\ 0$	11
$1 * *$	11
$1\ 1\ *$	15
$1 * 1$	15
$*\ 1\ *$	13
$*\ 1\ 1$	15
$* *\ 1$	14

TAB. 3 – *Un problème "trompeur"*

représente la performance de ce schéma, c'est-à-dire la moyenne des qualités de ses instances dans la population. On a les relations suivantes :

$$\begin{array}{ll} f(0**) > f(1**) & f(00*) > f(11*), f(01*), f(10*) \\ f(*0*) > f(*1*) & f(0*0) > f(1*1), f(0*1), f(1*0) \\ f(**0) > f(**1) & f(*00) > f(*11), f(*01), f(*10) \end{array}$$

Bien que le chromosome 111 soit l'optimum global, on peut prédire que l'algorithme génétique aura des difficultés à converger vers celui-ci et que la recherche sera plutôt dirigée vers le chromosome 000. En effet, il découle du théorème du schéma que le nombre d'instances d'un schéma aura tendance à augmenter à la génération suivante s'il a une bonne performance ; or les performances des schémas contenant le chromosome 111 sont moins bonnes que celles des schémas contenant le chromosome 000 et ils auront donc moins tendance à proliférer.

3 Parallélisation

3.1 Parallélisation des algorithmes génétiques traditionnels

Nous allons voir rapidement les trois principales méthodes. Une quatrième consiste à construire un AG hybride combinant celles-ci. Toutes ces méthodes sont présentées dans [3]. Une présentation des algorithmes génétiques parallèles et d'autres techniques utilisées pour l'Extraction de Connaissances dans les grosses bases de données est également proposée dans [8].

3.1.1 La parallélisation globale

Ce type d'AG parallèle fait évoluer une seule population, comme dans l'algorithme séquentiel, mais l'évaluation des individus et/ou l'application des opérateurs génétiques se fait en parallèle (voir figure 4). Le calcul de la qualité des individus de la population est parallélisé en assignant à chaque processeur une fraction de la population qu'il sera chargé d'évaluer. Ce calcul étant indépendant du reste de la population, aucune communication entre les processeurs n'est nécessaire entre le moment où un processeur reçoit ses individus et celui où il renvoie leurs qualités. L'application des opérateurs génétiques peut être faite en parallèle en suivant le même principe, mais elle nécessite de nombreuses communications entre processeurs et il est probable que leur coût compense les gains obtenus.

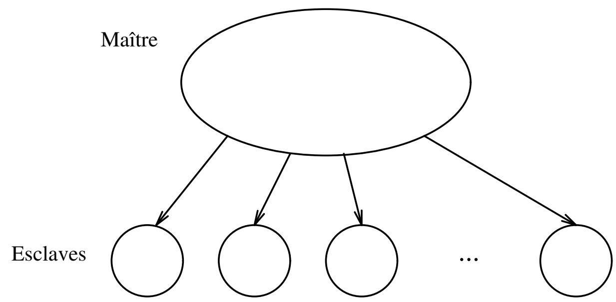

FIG. 4 – *Parallélisation globale*

Comme il n'y a qu'une population, chaque individu est en compétition avec tous les autres et a des chances de participer à la phase de reproduction avec n'importe quel autre. Ainsi, si l'implantation est telle que le processeur maître attend que tous les processeurs esclaves aient terminé leurs calculs avant de passer à la génération suivante (c'est le cas en général), la sélection et la reproduction restent globales et le comportement de l'algorithme séquentiel n'est pas modifié ; on peut donc lui appliquer la théorie de l'algorithme séquentiel.

Cette méthode est efficace lorsque l'évaluation des individus nécessite un traitement assez long et que le coût des communications n'est pas trop important.

3.1.2 La parallélisation à gros grain

Dans l'algorithme parallèle à gros grain (*coarse-grained parallel GA*), la population est divisée en quelques sous-populations (*demes*), qui évoluent isolées les unes des autres, mais échangent des individus occasionnellement (voir figure 5). Cet échange d'individus est appelé *migration*.

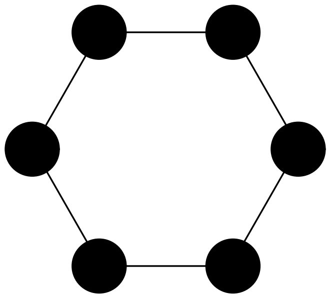

FIG. 5 – *Parallélisation à gros grain*

Il existe différentes stratégies pour contrôler la migration. Ces algorithmes introduisent donc des paramètres supplémentaires par rapport à l'algorithme séquentiel : nombre de sous-populations (et donc taille des sous-populations), taux de migration (nombre d'individus échangés), fréquence de migration ; ils nécessitent également des choix de stratégie pour la sélection des individus échangés lors de la migration, le replacement de ces individus dans la sous-population qui les accueille (choix des individus remplacés) et le choix d'une topologie pour la migration des individus (déterminant les connections entre les sous-populations). Ces paramètres ont un impact très important sur l'efficacité de l'algorithme, tant du point de vue de la rapidité que de la qualité des solutions, ce qui rend complexe la mise au point de tels algorithmes.

Ce type d'AG introduit donc des changements fondamentaux par rapport à l'algorithme séquentiel et a un comportement différent. Il se rapproche du modèle des “îlots” de la programmation génétique.

3.1.3 La parallélisation à grain fin

Ce modèle associe à la population totale une structure spatiale. L'algorithme génétique fait évoluer la population entière, mais la sélection et la reproduction sont locales. En effet, la qualité d'un individu est calculée en fonction des individus se trouvant dans un voisinage restreint (relativement à la structure spatiale de la population) et la reproduction ne se fait qu'entre individus voisins. Cependant, le recoupement des voisinages permet de propager les bonnes caractéristiques dans la population entière (voir figure 6). Ce modèle est adapté aux machines massivement parallèles.

Ce type d'AG peut être implanté de différentes façons, suivant la structure du voisinage (topologie, rayon) et les stratégies de sélection des meilleurs individus

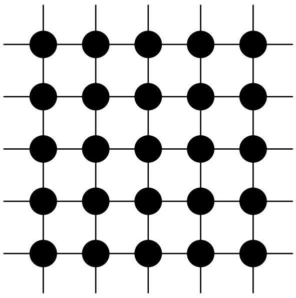

FIG. 6 – *Parallélisation à grain fin*

et des couples pour la reproduction. Là encore le fonctionnement de l'algorithme séquentiel est modifié.

3.2 Le parallélisme avec le modèle BSP

3.2.1 Le modèle PRAM

Le modèle PRAM (*Parallel Random Access Machine*) est un concept abstrait de machine parallèle, sorte d'ordinateur parallèle idéal avec un nombre illimité de processeurs travaillant de façon synchrone et de bancs mémoire. Il s'agit d'un ensemble de RAM (*Random Access Machine* ou couples processeur-mémoire) travaillant sur une mémoire partagée (voir figure 7).

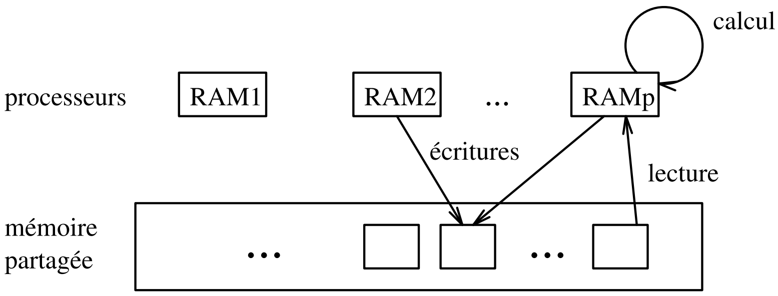

FIG. 7 – *La machine PRAM*

Caractéristiques du modèle. La PRAM consiste en un nombre non borné de processeurs indexés par les entiers naturels. Ces processeurs sont des RAM munies des opérations et instructions usuelles, qui travaillent sur le même programme de façon synchrone, avec un coût uniforme, et partagent une mémoire globale de taille non bornée au travers de laquelle ils communiquent. Le coût des opérations arithmétiques est constant. Bien que traitant les mêmes instructions, les processeurs peuvent travailler sur des données différentes, un tel modèle est dit SIMD (Single Instruction, Multiple Data-stream). A chaque pas, chaque processeur peut accéder à un banc mémoire, soit pour une lecture soit pour une écriture, ce qui pose le problème de la résolution des conflits.

Ce modèle néglige les contraintes imposées par une architecture plus spécifique (nombre non borné de processeurs...). Il n'y a donc aucune limitation dans le parallélisme, ce qui implique de pouvoir faire tous les liens possibles entre processeurs et bancs mémoire, ceci n'étant pas physiquement réalisable. Cependant, des méthodes existent pour simuler un tel ordinateur idéal sur des ordinateurs parallèles plus réalistes. Des techniques de gestion du routage des messages et de la mémoire ont été développées, qui montrent qu'une implantaion efficace est en effet possible dans de nombreux cas.

Conditions nécessaires pour la réalisation d'un algorithme PRAM :

- l'architecture doit comporter autant de processeurs que la PRAM comporte de processus ;
- les opérations locales aux processus PRAM doivent être réalisées par des opérations en mémoire locale des processeurs ;
- nécessité d'un protocole complet sur la MIMD (architectures les plus courantes) pour la mémoire partagée : actions suffisantes pour réaliser une étape globale de lecture/écriture, c'est-à-dire les synchronisations et communications point-à-point.

Certaines des hypothèses faites pour le modèle PRAM ne peuvent être satisfaites dans la réalité, par exemple l'hypothèse de communication uniforme n'est pas vérifiée pour des raisons physiques. Ainsi L. Valiant [20] a proposé un modèle qui est un raffinement de la PRAM, c'est le modèle BSP (*Bulk Synchronous Parallelism*).

3.2.2 Le modèle BSP

Ce modèle précise les coûts d'accès à la mémoire partagée (ou, ce qui revient au même, des communications inter-processeurs) et les coûts de synchronisation par des paramètres g et l décrivant les capacités du matériel. Une machine BSP est ainsi décrite par un triplet (g, p, l).

3.2.2.1 La machine BSP.

Description. Elle est composée (voir figure 8) :

- d'un ensemble de processeurs séquentiels à mémoire locale ;
- d'un réseau de communication reliant ces processeurs et permettant les opérations mémoire à distance ;
- d'un mécanisme de synchronisation globale.

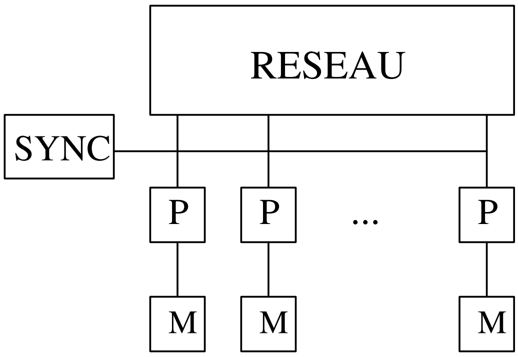

Fig. 8 – *La machine BSP*

Les paramètres définissant les performances. Trois paramètres sont nécessaires pour définir les performances d'une machine BSP :

- l: délai requis pour synchroniser tout le système ;
- p : nombre de processeurs ;
- g : coût nécessaire pour délivrer un message de taille égale à un mot mémoire.

Le tableau 4 présente les paramètres BSP disponibles pour trois machines parallèles.

Machines	p	$l(flops)$	$g(flops/word)$
Multiprocessor Sun Sparc 20	4	118	4.1
IBM SP2 (Ethernet)	8	88795	1224.1
Cray T3D	256	387	2.4

Tab. 4 – *Paramètres de machines BSP*

3.2.2.2 Structure d'un algorithme BSP. C'est une séquence de phases appelées *supersteps*, chacune complétée par une synchronisation globale et un échange de données (voir figure 9). Durant une phase, chaque processeur peut réaliser des calculs locaux et/ou émettre des requêtes non bloquantes qui ne sont réalisées qu'avant le début de la phase suivante. Les requêtes doivent donc anticiper les besoins de la phase suivante pour ses calculs locaux. Considérer les communications en masse permet de simplifier les traitements et de borner le temps nécessaire pour délivrer les données.

Un algorithme BSP possède des propriétés particulièrement intéressantes :

- simplicité d'écriture ;
- indépendance par rapport à l'architecture cible ;
- possibilité de prédiction de la performance d'un programme sur une architecture donnée.

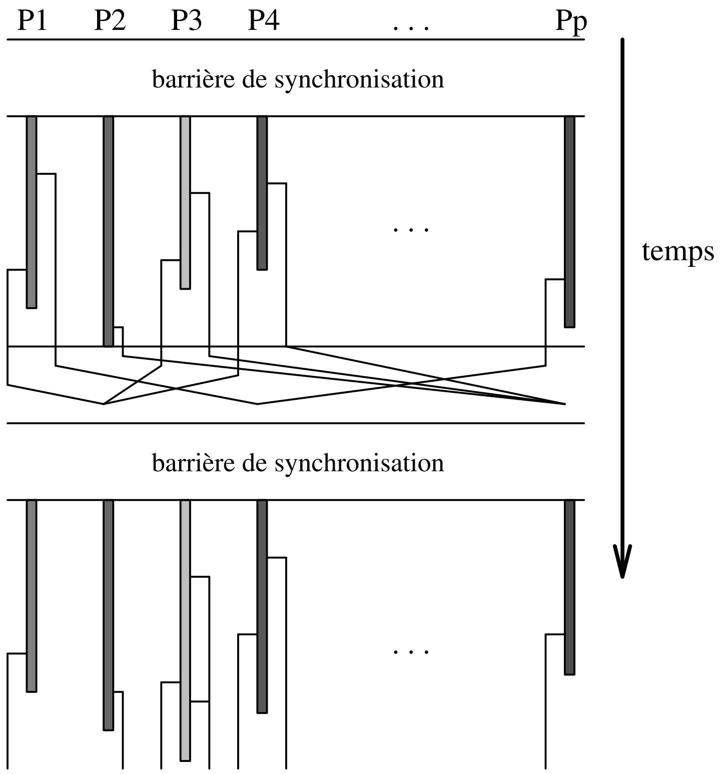

FIG. 9 – *Structure d'un algorithme BSP*

3.2.2.3 Prévision du coût d'un algorithme BSP. On mesure les communications collectivement. Les ordres d'écriture à distance et les demandes/réponses de lecture engendrent des messages. Tout ensemble de messages échangés établit une relation sur l'ensemble des processeurs : de ceux qui émettent des messages vers ceux qui les reçoivent. On mesure ces relations par leur degré maximal en tant que graphes. Une h-relation est ainsi une relation binaire sur $\{1,\dots,p\}$ où tout sommet est de degré entrant au plus h et de degré sortant au plus h. On constate alors que si g est le temps requis pour acheminer une 1-relation alors toute h-relation pourra être acheminée en temps $g.h$.

On note :

- w : maximum sur $i \in \{1,\dots,p\}$ du nombre d'opérations locales effectuées dans la phase courante par le processeur i (le coût des opérations de base étant constant, on leur associe un coût égal à 1 dans nos calculs) ;
- h_s : le nombre maximum de requêtes émises par un processeur ;
- h_r : le nombre maximum de requêtes reçues par un processeur.

Alors selon notre définition de la machine BSP, la phase i sera réalisée en temps

$$t_i = w + g.max\{h_s,h_r\} + l$$

ou, si on pose $h = max\{h_s,h_r\}$,

$$t_i = w + g.h + l$$

Ainsi, si on considère la première étape de la figure 9, on a $h_s = 2$ et $h_r = 3$ d'où un coût de $3.g$ pour les communications.

En répétant ce raisonnement pour chacune des phases de l'algorithme, on obtient un temps total de

$$t = \sum_i t_i = \sum_i (w_i + g.h_i + l) = W + H.g + S.l$$

où $W = \sum_i w_i$, $H = \sum_i h_i$ et S est le nombre de phases. Le nombre W mesure ainsi le travail local de calcul séquentiel, H mesure le volume des communications et S la fréquence des synchronisations.

Remarques :

- le paramètre g correspondant au temps nécessaire pour acheminer un message de taille égale à un mot mémoire, la formule décrite ci-dessus représente le cas où les messages sont de taille 1. En fait, BSP ne fait pas de différence entre un message de taille m et m messages de taille 1 : le coût est $m.h.g$ dans les deux cas. Ainsi, lorsque les messages sont de taille supérieure ou égale à 1, il faut définir h_s comme le nombre maximum de mots émis par un processeur, et h_r comme le nombre maximum de mots reçus par un processeur ;
- il est préférable de répartir équitablement entre les processeurs les quantités totales de travail et de communication pour minimiser le coût en temps. On minimise $w = max_j(w_j)$ (donc W) lorsque $w_1 = w_2 = \ldots = w_p$ et $h = max_j(h_j)$ lorsque $h_1 = h_2 = \ldots = h_p$;
- le modèle BSP présenté ici ne tient pas compte de l'espace mémoire utilisé, ni des coûts de démarrage occasionnés par l'envoi de petits messages (l'envoi d'un message requiert un certain nombre d'opérations préalables, ceci occasionnant un coût qui devient non négligeable lorsqu'on envoie des messages de peu de mots, messages qui seront acheminés plus rapidement que ceux de taille plus importante). Il existe cependant des versions plus fines du modèle.

3.3 Algorithme parallèle proposé

Nous présentons ici une méthode pour paralléliser les algorithmes génétiques, sans modifier leur fonctionnement. La population est répartie sur différents processeurs, mais se comporte comme une population unique dans laquelle chaque individu a une chance de se reproduire avec n'importe quel autre. L'intérêt de préserver le fonctionnement de l'algorithme séquentiel est double : du point de vue parallélisme, nous étudions les performances de deux algorithmes (l'un séquentiel, l'autre parallèle) qui réalisent exactement le même travail, de plus nous préservons le comportement de l'algorithme et donc ses propriétés (théorème du schéma...). Cette méthode est basée sur le modèle BSP, ce qui nous permet de proposer une étude de son coût en temps.

3.3.1 Description de la parallélisation de l'algorithme général

Dans un premier temps, notre but est uniquement d'améliorer les performances d'un algorithme génétique séquentiel en terme de temps en utilisant le modèle BSP, et non de modifier la qualité ou le nombre de ses solutions. On souhaite donc paralléliser cet algorithme séquentiel en essayant de respecter au maximum son fonctionnement. Pour arriver à cela, on s'attache notamment à conserver exactement les mêmes étapes que dans l'algorithme séquentiel et à permettre à chaque processeur de travailler sur la population totale, et non sur une sous-population comme dans une technique telle que celle des "îlots", bien que la population soit répartie sur les différents processeurs. On espère ainsi minimiser, voire éliminer, les effets de la parallélisation sur le fonctionnement de l'algorithme et peut-être sur les résultats ; mais ceci implique des communications d'individus fréquentes entre les processeurs, afin de fournir à chacun les individus dont il a besoin pour réaliser ses traitements. On se rapproche de la parallélisation globale dans laquelle l'algorithme travaille sur la population totale et les tâches de calcul de qualité et d'application des opérateurs génétiques sont réparties, tout en préservant le comportement de l'algorithme séquentiel. Dans notre cas, aucun processeur n'apparaît comme le maître des autres, alors que dans la parallélisation globale un processeur gère la répartition des tâches à effectuer parmi les autres processeurs. On a choisi de distribuer la population, ce qui évite de stocker toute la population sur un seul processeur et permet d'équilibrer les charges de travail de chacun ainsi que les communications (le nombre maximum de messages émis/reçus par un processeur est alors moins important, car les processeurs émettent et reçoivent des messages provenant de différents processeurs) qui sont donc réalisées plus rapidement.

Reprenons les étapes de l'algorithme séquentiel et regardons celles qui peuvent être parallélisées :

1. génération de la population initiale et évaluation des individus
2. boucle
   2.1. sélection (création d'une *mating pool*)
   2.2. boucle
      2.2.1. choix d'un couple d'individus dans la *mating pool*
      2.2.2. cycle de reproduction
         - croisement

- mutation

2.2.3. évaluation des nouveaux individus

2.3 test de convergence

Dans notre algorithme, la sélection des individus se fait par tournois entre deux individus. Nous verrons plus loin que cette technique peut-être implantée avec un nombre peu important de communications et de calculs par rapport à ce qui est nécessaire pour d'autres méthodes, et sans modifier le fonctionnement de l'algorithme séquentiel, malgré la répartition de la population sur différents processeurs. Nous avons également choisi d'utiliser une *mating pool*, qui regroupe des individus sélectionnés pour leur bonne qualité et susceptibles de participer à la phase de reproduction. Lors de la phase de reproduction, la sélection est déjà faite et il suffit alors de piocher au hasard des couples d'individus dans cette *mating pool*.

Dans ce qui suit, nous présentons pour chacune des étapes de cet algorithme, une version parallèle ne modifiant pas son fonctionnement séquentiel.

3.3.1.1 La génération de la population initiale. Elle peut être aisément parallélisée. Il suffit de répartir équitablement la création des individus sur chaque processeur. Si on considère une population de n individus à créer et p processeurs, chaque processeur aura donc $\frac{n}{p}$ individus à créer. On a bien alors une population de n individus répartie équitablement sur tous les processeurs.

3.3.1.2 La sélection d'individus de bonne qualité. Ce sont ces individus qui pourront intervenir dans la reproduction pour créer les individus de la génération suivante. Dans l'algorithme séquentiel, tous les individus sélectionnés sont stockés dans une *mating pool* de taille n, d'où ils pourront être pris pour participer à un cycle de reproduction. On utilise comme technique de sélection celle du tournoi entre deux individus, qui consiste à en prendre deux au hasard dans la population courante et à garder celui dont la qualité est la meilleure.

Soit `pop` un tableau contenant les individus de la population et leur qualité, et `mating_pool` un tableau contenant les individus choisis lors de la phase de sélection. La partie de l'algorithme séquentiel correspondante est la suivante :

pour i de 0 à $n-1$ **faire**
    tirer aléatoirement deux nombres entre 0 et $n-1 \rightarrow$ ind1 et ind2
    si pop[ind1].qualité $\geq$ pop[ind2].qualité **alors**
        mating_pool[i] $\leftarrow$ pop[ind1].individu
    sinon
        mating_pool[i] $\leftarrow$ pop[ind2].individu
    fin si
fin pour

L'idée proposée pour la parallélisation de cette étape est de répartir les individus de la *mating pool* de l'algorithme séquentiel, en créant sur chaque processeur une *mating pool* locale de taille $\frac{n}{p}$. La création de ces *mating pool* se fait en parallèle, chaque processeur pouvant sélectionner des individus venant d'autres processeurs. Cette étape se déroule en deux phases : une première au cours de laquelle chaque processeur lance des requêtes de lecture à destination de processeurs éventuellement distants pour remplir un tableau `tournoi` local avec des couples d'individus, une seconde durant laquelle ces individus sont mis en compétition pour finalement ne garder dans la *mating pool* que le meilleur de chaque couple. La structure des algorithmes BSP nous impose ces deux phases, car la première se termine par un certain nombre de communications dont les données seront utilisées lors de la deuxième.

L'algorithme sera exécuté par tous les processeurs en parallèle (`tournoi` est un tableau de taille $2 \times \frac{n}{p}$ contenant, pour chaque processeur, les individus choisis pour participer aux tournois locaux ainsi que leur qualité) :

/ remplissage du tableau tournoi du processeur */*
pour i de 0 à $\frac{n}{p} - 1$ **faire**
    tirer aléatoirement un nombre entre 0 et $n - 1 \rightarrow$ ind1
    lire l'individu d'indice (ind1 mod $\frac{n}{p}$) de la population
        du processeur (ind1 div $\frac{n}{p}$) ainsi que sa qualité
            $\rightarrow$ tournoi[2i].individu, tournoi[2i].qualité
    tirer aléatoirement un nombre entre 0 et $n - 1 \rightarrow$ ind2
    lire l'individu d'indice (ind2 mod $\frac{n}{p}$) de la population
        du processeur (ind2 div $\frac{n}{p}$) ainsi que sa qualité
            $\rightarrow$ tournoi[2i+1].individu, tournoi[2i+1].qualité
fin pour

synchronisation

/ tournoi et remplissage de la mating pool locale */*
pour i de 0 à $\frac{n}{p} - 1$ **faire**
    si tournoi[2i].qualité $\geq$ tournoi[2i+1].qualité **alors**
        mating_pool[i] $\leftarrow$ tournoi[2i].individu
    sinon
        mating_pool[i] $\leftarrow$ tournoi[2i+1].individu
    fin si
fin pour

On aurait pu ne faire qu'une phase, mais alors chaque processeur ne travaillerait que sur sa population locale. En permettant la sélection d'individus sur d'autres processeurs, on ne modifie pas le fonctionnement de l'algorithme séquentiel. Pour choisir un individu au hasard, on tire aléatoirement un nombre entre 0 et $n-1$. La division entière de ce nombre par la taille de la population locale donne un nombre compris entre 0 et $p-1$, qui représente l'indice du processeur où choisir l'individu, et le nombre tiré modulo la taille de la population locale indique le numéro de l'individu choisi. Il suffit alors d'émettre une requête de lecture à destination du processeur d'indice calculé. On choisit donc l'indice à partir d'un seul tirage aléatoire, ce qui correspond bien à l'algorithme séquentiel. La sélection par tournoi présente l'avantage de ne pas prendre en compte toute la population, mais uniquement les deux individus mis en compétition. Ainsi on peut l'implanter dans le modèle BSP avec un nombre peu important de communications et de calculs par rapport à ce qui aurait été nécessaire pour des méthodes tenant compte de la population totale, et sans modifier son fonctionnement malgré la répartition de la population sur différents processeurs. Pour une méthode de sélection telle que celle appelée *fitness ranking*, par exemple, il aurait fallu soit se limiter à la population locale et perdre ainsi en diversité, soit effectuer de nombreuses communications pour évaluer le rang de l'individu dans la population totale puis le lire.

Quand tous les processeurs ont terminé cette phase, on a donc une *mating pool* de n individus répartie sur les p processeurs.

3.3.1.3 La phase de reproduction. En séquentiel, cette étape se présente comme suit (k est la longueur d'un chromosome) :

pour i de 0 à $\frac{n}{2}-1$ **faire**
  / croisement */*
  tirer aléatoirement deux nombres entre 0 et $n-1 \rightarrow$ ind1 et ind2
  tirer aléatoirement un nombre entre 1 et $k-1 \rightarrow$ pt_crois
  croiser les individus mating_pool[ind1] et mating_pool[ind2]
    au point pt_crois avec une probabilité de 0.6
  / mutation */*
  muter un gène des descendants avec une probabilité de 0.001
  calculer la qualité des nouveaux individus
fin pour

Dans la version parallèle, on souhaite effectuer les opérations en parallèle sur les différents processeurs, en leur laissant la possibilité de choisir des individus provenant de processeurs distants, toujours dans le but de préserver le fonctionnement de l'algorithme séquentiel. Là encore, deux phases sont nécessaires. Lors de la première, chaque processeur doit choisir $\frac{n}{2p}$ couples d'individus qui participeront à un cycle de reproduction parmi les *mating pool* de tous les processeurs. Lors de la seconde, chaque processeur réalise pour ces couples, en séquentiel, le cycle qui consiste à réaliser le croisement des deux individus avec probabilité et la mutation de leurs deux descendants avec probabilité. La version proposée est la suivante :

> */* remplissage du tableau pop du processeur par les couples */*
> */* d'individus qui pourront se reproduire */*
> **pour** i de 0 à $\frac{n}{2p} - 1$ **faire**
>   */* sélection de deux individus */*
>   tirer aléatoirement un nombre entre 0 et $n - 1 \rightarrow$ ind1
>   lire l'individu d'indice (ind1 mod $\frac{n}{p}$) de la *mating pool*
>   du processeur (ind1 div $\frac{n}{p}$)
> $\rightarrow$ pop[2i].individu
>   tirer aléatoirement un nombre entre 0 et $n - 1 \rightarrow$ ind2
>   lire l'individu d'indice (ind2 mod $\frac{n}{p}$) de la *mating pool*
>   du processeur (ind2 div $\frac{n}{p}$)
> $\rightarrow$ pop[2i+1].individu
> **fin pour**
>
> **synchronisation**
>
> **pour** i de 0 à $\frac{n}{2p} - 1$ **faire**
>   */* croisement */*
>   tirer aléatoirement un nombre entre 1 et $k - 1 \rightarrow$ pt_crois
>   croiser les individus contenus dans pop[2i].individu et
>   pop[2i+1].individu au point pt_crois avec une probabilité de 0.6
>   */* mutation */*
>   muter un gène des descendants avec une probabilité de 0.001
>   calculer la qualité des nouveaux individus
> **fin pour**

Là encore, le fait de permettre à chaque processeur de choisir les individus parmi toutes les *mating pool* lors de la première étape, permet de se conformer autant que possible au fonctionnement de l'algorithme séquentiel et d'utiliser la diversité offerte par l'ensemble de la population, mais impose des communications et donc deux phases séparées par une barrière de synchronisation, car les données échangées à la fin de la première sont utilisées dans la seconde.

3.3.1.4 Le test de convergence. Pour réaliser ce test, il est nécessaire d'avoir, pour chaque gène, des résultats globaux sur sa valeur dominant dans la population entière. Par conséquent, il faut collecter et rassembler les résultats de tous les processeurs sur un seul. Comme on utilise un codage binaire, il suffit de faire, pour chaque gène, la somme de toutes les valeurs qu'il prend dans les sous-populations (c'est-à-dire la somme case à case de tous les individus). Les résultats de chaque processeur sont mis dans un tableau local, puis après fusion parallèle de tous ces tableaux, on pourra tester la convergence.

Le fonctionnement de la fusion est donné par l'algorithme suivant et la figure 10 :

pour i de 0 à $log_2(p) - 1$ **faire**
  lire le tableau de résultats du processeur à distance 2^i
  faire la somme case à case des résultats du tableau reçu
    avec ceux du tableau local
fin pour

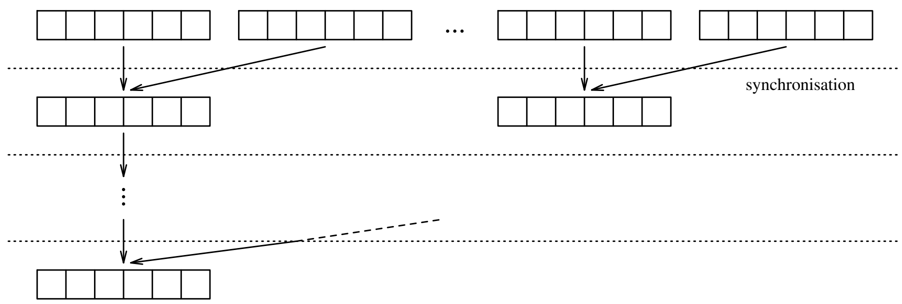

FIG. 10 – *Fusion de tableaux en parallèle*

La fusion terminée, la somme case à case de tous les individus de la population sera disponible sur le processeur disposant du tableau résultant de la fusion. On peut alors vérifier si, pour tous les gènes, cette somme est supérieure ou égale à 95% de la taille de la population totale (convergence de la population vers la valeur 1 pour le gène), ou si cette somme est inférieure ou égale à 5% de la taille de la population (convergence du gène vers la valeur 0). Si tous les gènes ont convergé, alors la population a convergé.

L'algorithme parallèle avec sélection par tournoi est proposé en annexe.

3.3.2 Etude du coût moyen d'une itération de l'algorithme

On souhaiterait, bien entendu, que les performances en temps de notre algorithme parallèle soient meilleures que celles de l'algorithme séquentiel. Pour ce qui est de la qualité des solutions, on ne l'a pas modifiée car on a conservé le fonctionnement de l'algorithme séquentiel.

Dans notre algorithme parallèle, la génération des individus est faite en parallèle et ne nécessite ni calcul séquentiel supplémentaire par rapport à ce qui est fait en séquentiel, ni communications. Nous gagnons donc du temps. Les phases de communication et de synchronisation sont en général coûteuses car les valeurs des paramètres g et l peuvent être très importantes, mais le calcul de la qualité peut aussi prendre beaucoup de temps, surtout pour les applications que nous avons choisi de traiter et qui utilisent des informations stockées dans une base de données. Nous espérons donc pouvoir gagner du temps sur le reste de l'algorithme, car le calcul de la qualité des individus de la population totale revient souvent dans l'algorithme séquentiel et se fera ici sur plusieurs processeurs. Les étapes de notre algorithme parallèle, qu'il est important de quantifier en temps pour les comparer à celles de l'algorithme séquentiel, sont donc celles qui comprennent des communications, synchronisations ou le calcul de la qualité des individus. On identifie ainsi deux phases qui semblent être les plus coûteuses du point de vue de l'algorithme parallèle :

- la phase de sélection des individus, qui nécessite des communications pour pouvoir prendre des individus à distance ;
- la phase de reproduction, avec le choix des couples de "parents" sur chaque processeur, où là encore les processeurs pourront prendre des individus venant d'une autre *mating pool* que la leur.

Etudions donc le coût de ces deux phases et comparons le au temps nécessaire à l'algorithme séquentiel pour effectuer le même travail.

3.3.2.1 Coût de la phase de sélection par tournoi. On étudie le cas moyen où les communications sont équitablement réparties entre les p processeurs, hypothèse raisonnable dans la mesure où l'on effectue un tirage aléatoire. Chaque processeur effectue lors de cette phase $\frac{n}{p}$ tournois pour lesquels il a besoin de deux individus et de leur qualité. Comme on l'a vu, une première étape consiste à obtenir ces couples. Pour cela, chacun commence par tirer aléatoirement deux nombres à partir desquels il calculera l'indice des processeurs (voir la description de la phase de sélection) sur lesquels prendre ces individus, puis les indices de ceux-ci sur le processeur. Ensuite il lance les requêtes lui permettant d'obtenir les individus et leurs qualités. Ceci sera fait $\frac{n}{p}$ fois. On prend un coût de 1 pour le tirage d'un nombre, 1 pour remettre ce nombre dans l'intervalle $[0,n-1]$, 1 également pour chacun des calculs d'indices ; ceci donne un coût total de $2 \times 4 \times \frac{n}{p}$ pour le choix de tous les indices. Puis chaque processeur procède à l'envoi des $2 \times \frac{n}{p}$ requêtes de lecture des individus et de leur qualité, et comme on considère les communications équitablement réparties, chacun recevra également $2 \times \frac{n}{p}$ de ces requêtes. Il s'agit d'une $(2 \times \frac{n}{p})$-relation, mais les objets communiqués sont des chaînes de caractères de la taille d'un individu acompagnées d'un entier représentant la qualité de l'individu, et non des objets de taille 1. Si on note ici t la taille d'un individu (nombre de gènes), on aura donc en moyenne un coût en temps de $(2 \times (t+1) \times \frac{n}{p}) \times g$ pour le remplissage du tableau "tournoi".

La qualité des individus étant transmise, le tournoi se limite à un test et une affectation pour remplir la *mating pool*. Si on prend un temps de 1 pour les tests et 1 pour les affectations, la deuxième étape réalise $\frac{n}{p}$ tournois et remplit la *mating pool* avec un coût en temps de $2 \times \frac{n}{p}$.

Ces deux étapes sont séparées par une barrière de synchronisation, ce qui fait un temps moyen de $8 \times \frac{n}{p} + 2 \times (t+1) \times \frac{n}{p} \times g + l + 2 \times \frac{n}{p}$. On parle de temps moyen car cette valeur suppose que les individus sont répartis équitablement sur les processeurs, ce qui est le cas en moyenne.

3.3.2.2 Coût de la phase de reproduction. Chaque processeur doit générer une nouvelle sous-population de taille $\frac{n}{p}$. Pour cela il doit composer, lors d'une première étape, $\frac{n}{2p}$ couples d'individus qui seront les parents des individus de la génération suivante. Ces parents peuvent provenir de n'importe quel processeur, ainsi $\frac{n}{p}$ calculs d'indices seront effectués (selon la méthode décrite précédemment), puis $\frac{n}{p}$ requêtes de lecture seront émises par chaque processeur, et en considérant toujours une répartition à peu près équitable des communications, le même nombre sera reçu par chacun. Le calcul des indices a un coût total de $4 \times \frac{n}{p}$. De plus, les requêtes induisent une $\frac{n}{p}$-relation, qui entraîne une phase de communication d'individus dont le coût moyen en temps sera de $t \times \frac{n}{p} \times g$.

La seconde étape consiste, pour chaque processeur, à effectuer les croisements et mutations nécessaires, puis à calculer la qualité des individus de sa nouvelle sous-population. Considérons tout d'abord l'application des opérateurs génétiques, en posant que la copie d'une chaîne de caractères a un coût de 1 (ce qui donne un coût de 4 pour l'opérateur de croisement), et que l'application de l'opérateur de mutation a un coût de 1 également. Pour chaque couple, il faut tirer aléatoirement un nombre qui, après avoir été remis dans l'intervalle [0,1], nous permet de savoir s'il faut ou non réaliser le croisement entre les deux individus considérés (le croisement est réalisé seulement si ce nombre est inférieur ou égal à 0.6). Si le croisement doit être réalisé, il faut ensuite tirer un second nombre, replaçé cette fois dans l'intervalle $[1,t-1]$ pour être le point de croisement. Il ne reste plus alors qu'à réaliser le croisement. En ce qui concerne la mutation, on tire deux nombres si la mutation a lieu, un seul sinon. Le premier permet de savoir si la mutation doit être réalisée ou non ; si oui, le second est tiré et indique le point de mutation. Pour tenir compte des probabilités, on estime le coût d'application des opérateurs génétiques à $\frac{n}{2p} \times (3 + 0.6 \times 6 + 2 \times (3 + 0.001 \times 3))$.

Considérons maintenant le calcul de la qualité des individus. Si on note K le coût du calcul de la qualité d'un individu, le coût moyen nécessaire pour exécuter cette étape est de $\frac{n}{p} \times K$.

Ces deux étapes sont séparées par une barrière de synchronisation. Ainsi on obtient un temps moyen d'exécution de la phase de choix des individus parents de $4 \times \frac{n}{p} + t \times \frac{n}{p} \times g + l + \frac{n}{2p} \times (3 + 0.6 \times 6 + 2 \times (3 + 0.001 \times 3)) + \frac{n}{p} \times K$.

3.3.2.3 En résumé. La somme des coûts des deux étapes que nous venons d'étudier (étapes les plus coûteuses) est de $(14 + 10 \times T) \times \frac{n}{p} + (3 \times t + 2) \times \frac{n}{p} \times g + \frac{n}{p} \times 6.303 + 2 \times l$, ce qui nous donne un coût de $n \times \frac{20.303 + 10 \times T + (3 \times t + 2) \times g}{p} + 2 \times l$.

La première phase dans l'algorithme séquentiel a un coût de $4 \times n$, la seconde en a un beaucoup plus élevé, en raison du calcul de la qualité des individus, qui est de $\frac{n}{2} \times (7 + 0.6 \times 6 + 2 \times (3 + 0.001 \times 3)) + 10 \times n \times T$. La somme des deux nous donne un coût de $4 \times n + n \times 8.303 + n \times 10 \times T$, soit $n \times (12.303 + 10 \times T)$.

Pour espérer un gain de temps avec notre algorithme parallèle, il faudrait donc avoir :

$$\frac{20.303+10\times T+(3\times t+2)\times g}{p} + \frac{2\times l}{n} \leq 12.303 + 10 \times T \quad .$$

K correspond au nombre moyen d'instructions nécessaire pour calculer la qualité d'un individu et dépend donc de l'application. Afin d'avoir une estimation pour K, nous avons choisi de tester notre algorithme génétique sur un problème d'apprentissage de règles à partir d'exemples stockés dans une base de données.

4 Application au Data Mining

Dans ce chapitre, nous présenterons le problème utilisé comme application pour notre algorithme génétique parallèle. Nous estimerons alors une valeur pour K pour ce problème, ceci nous permettant de comparer le coût d'une itération de l'algorithme séquentiel avec celui d'une itération de l'algorithme parallèle que nous avons proposé, et de prédire une éventuelle amélioration des performances. Nous présenterons ensuite les résultats des premières expérimentations. Enfin nous proposerons deux autres algorithmes spécifiques au Data Mining.

4.1 Apprentissage de concepts à partir d'exemples

4.1.1 Description de l'application

On souhaite tester notre algorithme génétique parallèle sur un problème d'apprentissage très simple, à savoir l'apprentissage de règles en attribut-valeur à partir d'exemples stockés dans une base de données. Chacun de ces exemples est associé à une classe. On souhaite trouver une règle permettant de reconnaître les individus d'une certaine classe et de rejeter les autres.

Pour cela, nous allons définir la notion de couverture.

4.1.2 La notion de couverture

Soit R le schéma relationnel de la base (classe de l'exemple non comprise), $R = \{A_1, \ldots, A_k\}$ où chaque A_i désigne un nom d'attribut. On pose $dom(A_i) = D_i$ et $D = \cup_{i=1}^{k} D_i$.

Un **tuple** est une application t, $t : R \rightarrow D$ telle que $\forall A_i \; t(A_i) \in D_i$.

Un **concept** est défini par un ensemble d'exemples (POS) et de contre-exemples (NEG) représentés par des tuples de la base ayant, selon le cas, une classe positive ou négative.

Une **description de concept** I est une expression de la forme $L_1 \times L_2 \times \ldots \times L_k$, où $\forall i \in \{1,\ldots,k\}$ ($L_i \subseteq D_i$ et $L_i \neq \emptyset$) (sinon I ne pourrait pas être une solution valide pour le problème).

Un tuple, représenté par une application t, est **couvert** par une description I si $\forall i \; t(A_i) \in L_i$.

Etant donné un concept, le **problème d'apprentissage** consiste à trouver une description I du concept couvrant POS et ne couvrant aucun élément de NEG.

4.1.3 Codage des individus

L'espace de recherche pour notre algorithme génétique est l'ensemble des descriptions de concepts codables par un individu, on utilise ici un codage binaire, inspiré de celui proposé dans [6]. Les individus sont donc des chaînes (ou tableaux) de caractères '0' ou '1', qui représentent des règles potentielles pour reconnaître les exemples (tuples de la classe voulue). Chacun de ces caractères est associé à une valeur du domaine d'un attribut, le nombre de caractères de la chaîne est donc égal à la somme des cardinaux des domaines des attributs. Si pour la description considérée un attribut peut être égal à une certaine valeur, alors la case correspondante dans l'individu contiendra le caractère '1', '0' sinon (voir exemple). Ce codage permet de représenter les disjonctions de valeurs pour un attribut. Il s'agit du cas où plusieurs cases correspondant à un même attribut contiennent le caractère '1'. Quand aucun des caractères correspondant au domaine d'un attribut n'est égal à '1', l'individu n'est pas valide. Ce codage est simple et adapté à une représentation en attribut-valeur.

Exemple :
Considérons une base de données dont les tuples possèdent 3 attributs a_1, a_2 et a_3, tels que :

$dom(a_1) = \{a_{11},a_{12},a_{13}\}$
$dom(a_2) = \{a_{21},a_{22},a_{23},a_{24},a_{25}\}$
$dom(a_3) = \{a_{31},a_{32},a_{33}\}$

Les individus codant les descriptions du concept associé à cette base possèdent 11 cases, et l'individu codé par la chaîne de la figure 11 correspond aux tuples tels que

$$(a_1 = a_{13}) \wedge ((a_2 = a_{21}) \vee (a_2 = a_{22}) \vee (a_2 = a_{24})) \wedge ((a_3 = a_{31}) \vee (a_3 = a_{33})).$$

4.1.4 Prétraitement de la base de données

Dans notre application, un individu sera d'autant meilleur qu'il couvrira un plus grand nombre d'exemples positifs (tuples de la classe souhaitée) de la base et

a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}
0	0	1	1	1	0	1	0	1	0	1

$\underbrace{a_{11}\ a_{12}\ a_{13}}_{a_1}\ \underbrace{a_{21}\ a_{22}\ a_{23}\ a_{24}\ a_{25}}_{a_2}\ \underbrace{a_{31}\ a_{32}\ a_{33}}_{a_3}$

FIG. 11 – *Exemple de codage d'un individu*

rejettera un plus grand nombre de contre-exemples (tuples d'une autre classe). La qualité sera donc déterminée par une fonction dépendant des ensembles POS et NEG. L'objectif de ce travail étant de réduire les coûts en temps de l'algorithme séquentiel, et non d'augmenter la qualité des solutions obtenues, on utilise une fonction d'évaluation très simple, mais peu efficace pour la découverte de bonnes solutions, utilisant ces deux paramètres. Nous avons choisi celle qui sera égale, pour un individu, à la somme du nombre d'exemples couverts et du nombre de contre-exemples rejetés par cet individu. Il sera donc nécessaire de parcourir tous les tuples de la base pour chaque calcul de qualité.

Pour simplifier les calculs et limiter les accès à la base de données, on charge la base en mémoire vive au début de l'exécution sous forme de deux tableaux (un pour les exemples, un second pour les contre-exemples) à deux dimensions dont les lignes correspondent aux tuples et les colonnes aux attributs. Notons que ceci ne sera pas possible pour des bases de données de taille très importante. Pendant ce chargement, un prétraitement est effectué, qui consiste à associer un nombre à chaque valeur d'attribut et à remplacer chaque valeur des tableaux par le nombre qui lui correspond. La figure 12 illustre ce propos en reprenant les attributs et domaines utilisés dans la section précédente.

base de données

a_1	a_2	a_3
a_{11}	a_{23}	a_{32}
a_{13}	a_{24}	a_{31}
a_{11}	a_{21}	a_{32}
a_{12}	a_{25}	a_{32}
a_{12}	a_{24}	a_{33}
a_{11}	a_{22}	a_{31}

$\xrightarrow{\text{prétraitement}}$

base traitée

0	1	2
0	5	9
2	6	8
0	3	9
1	7	9
1	6	10
0	4	8

FIG. 12 – *Prétraitement de la base de données*

Par la suite on note `val(i,j)` la valeur de l'attribut j pour l'exemple i. Le nombre représentant la valeur d'un attribut dans le tableau correspond en fait au numéro de la case associée à cette valeur dans le chromosome d'un individu. Il est donc très simple de tester si un individu `IND` a une valeur donnée pour un attribut donné (`IND(val(i,j))=1`), c'est-à-dire si la règle codée par ce chromosome autorise ou non une telle valeur pour l'attribut.

Par exemple, l'individu de la figure 11 couvre le deuxième tuple de la base de données de la figure 12 et rejette tous les autres.

4.1.5 Base de données des champignons

La base de données utilisée pour les tests est celle des champignons [16]. Elle comporte 8124 tuples représentant chacun un champignon, et 22 attributs nominaux permettant de décrire leurs caractéristiques (forme, couleur...). A chaque tuple est associé une classe : e pour les comestibles, p pour les poisons. L'ensemble des champignons de classe e (notre ensemble POS) représente 51.8% du nombre total de tuples (4208 tuples), l'ensemble NEG 48.2% (3916 tuples).

L'objectif de notre algorithme sera de trouver une description de concepts couvrant les tuples de classe e et rejetant les tuples de classe p, permettant ainsi de reconnaître les champignons comestibles. Avec notre codage, il manipulera des chaînes de 117 caractères.

4.2 Coût de l'algorithme parallèle pour l'application

4.2.1 Evaluation du nombre d'instructions nécessaires pour calculer la qualité d'un individu

Notre algorithme parallèle sera d'autant plus avantageux que le temps nécessaire pour réaliser le calcul de la qualité d'un individu sera grand, puisque la population totale est répartie sur les processeurs et que chacun calcule seulement la qualité des individus de sa sous-population. Afin de nous situer dans le cas le plus défavorable, nous allons donc donner une borne inférieure pour le nombre d'opérations de la fonction d'évaluation.

Rappelons que pour notre application, l'algorithme calculant la qualité d'un individu doit calculer le nombre d'exemples de la base de données couverts et de contre-exemples rejetés par un individu. Grâce au prétraitement effectué sur la base de données, cet algorithme est simple. L'algorithme consiste donc en deux boucles qui parcourent successivement les lignes des deux tableaux résultant du prétraitement (donc parcourant tous les tuples de la base). Rappelons que `val(i,j)` représente la valeur de l'attibut `j` pour l'exemple `i`. Soit A le nombre d'attributs de la base et T_e le nombre d'exemples qu'elle contient (T_c est le nombre de contre-exemples). L'algorithme calculant le nombre d'exemples couverts par un individu `IND` est le suivant (celui calculant le nombre de contre-exemples rejetés est basé sur le même principe) :

nombre_de_couverts $\leftarrow$ 0

```
pour i de 1 à T_e faire
    couvert ← vrai
    j ← 0
    tant que couvert et j < A faire
        si IND(val(i,j))=0 alors
            couvert ← faux
        sinon
            j ← j + 1
        fin si
    fin tant que
    si couvert alors
        nombre_de_couverts ← nombre_de_couverts + 1
    fin si
fin pour
```

Cet algorithme consiste à parcourir le tableau des exemples, tester si le tuple courant est couvert par l'individu dont on calcule la qualité et à incrémenter ou non, en fonction de ce résultat, un compteur correspondant au nombre d'exemples couverts. Pour savoir si un tuple est couvert ou non par un individu, on utilise une autre boucle qui parcourt les attributs (c'est-à-dire les colonnes du tableau) pour regarder si leur valeur est compatible avec l'individu considéré, et s'arrête lorsque tous les attributs ont été parcourus ou lorsqu'on trouve un attribut dont la valeur ne correspond pas à celles autorisées par la description que code l'individu (le tuple n'est alors pas couvert par l'individu). On peut estimer raisonnablement qu'un minimum de 10 opérations de base (dont le coût est 1) sera nécessaire pour vérifier si un tuple est couvert ou non par l'individu considéré. On atteint déjà presque ce nombre en considérant tous les tests des conditionnelles faux (on ne rentre alors pas dans la boucle interne), ce qui correspond au cas très improbable où aucun tuple de la base n'est couvert par l'individu considéré et où la non compatibilité intervient dès le test du premier attribut. Ainsi on obtient un minimum de 10×T opérations de base pour calculer la qualité d'un individu.

4.2.2 Comparaison du coût de l'algorithme parallèle avec celui de l'algorithme séquentiel

Rappelons que pour cet algorithme, il faudrait, pour espérer un gain de temps, avoir :

$$\frac{20.303+10\times T+(3\times t+2)\times g}{p} + \frac{2\times l}{n} \leq 12.303 + 10 \times T.$$

Nous allons maintenant reprendre les trois machines dont les paramètres BSP sont présentés dans le tableau 4, ceci nous donnant des valeurs réelles pour les paramètres g, p et l. Ensuite nous pourrons étudier, pour chacune de ces machines, pour quelles valeurs des paramètres T (nombre de tuples de la base de données),

n (taille de la population) et t (taille d'un individu) l'inégalité est vérifiée. Nous allons donc d'abord fixer les paramètres g, p et l en utilisant les valeurs que nous possédons. On peut alors donner aux paramètres T et n des valeurs réalistes. Tous ces paramètres étant fixés, la seule inconnue restant dans notre inégalité est t. Nous pouvons donc déterminer un intervalle de valeurs pour ce paramètre pour lequel notre algorithme parallèle serait avantageux :

- pour la Multiprocessor Sun Sparc 20, $p = 4$, $l = 118 flops$ et $g = 4.1 flops/word$. Notre inégalité devient alors :

$$\frac{20.303+10\times T+(3\times t+2)\times 4.1}{4} + \frac{2\times 118}{n} \leq 12.303 + 10 \times T$$
$$\text{soit } t \leq \frac{20.709+30\times T-\frac{944}{n}}{12.3}$$

  Le tableau 5 présente les valeurs utilisées pour T et n, ainsi que l'intervalle de valeurs pour t pour lequel l'inégalité est vérifiée une fois T et n fixés. On commence les essais avec $n = 50$ et $T = 10$, ce qui représente des valeurs faibles pour ces paramètres.

T	n	t
10	50	[0,24]
	[100,1000]	[0,25]
50	[50,100]	[0,122]
	[500,1000]	[0,123]
100	[50,100]	[0,244]
	[500,1000]	[0,245]
500	50	[0,1219]
	100	[0,1220]
	[500,1000]	[0,1221]
1000	[50,100]	[0,2439]
	[500,1000]	[0,2440]
10000	50	[0,24390]
	[100,1000]	[0,24391]
100000	50	[0,243902]
	[100,500]	[0,243903]
	1000	[0,243904]

TAB. 5 – *Résultats pour la Multiprocessor Sun Sparc 20*

Dans le cas de la base des champignons, on a $T = 8124$ et $t = 4word$. La valeur de T se situe donc entre 1000 et 10000. La taille de l'intervalle pour t augmentant constamment, les paramètres de notre application vérifient l'inégalité. On peut en déduire que la parallélisation nous permettra d'obtenir un gain de temps.

– pour la IBM SP2, $p = 8$, $l = 88795 flops$ et $g = 1224.1 flops/word$. Notre inégalité devient alors :

$$\frac{20.303+10\times T+(3\times t+2)\times 1224.1}{8} + \frac{2\times 88795}{n} \leq 12.303 + 10 \times T$$
$$\text{soit } t \leq \frac{70\times T - 2370.079 - \frac{1420720}{n}}{3672.3}$$

T	n	t
[10,100]	[50,1000]	
500	50 100 [500,1000]	[0,1] [0,5] [0,8]
1000	50 100 500 1000	[0,10] [0,14] [0,17] [0,18]
10000	50 100 [500,1000]	[0,182] [0,186] [0,189]
100000	50 100 500 1000	[0,1897] [0,1901] [0,1904] [0,1905]

TAB. 6 – *Résultats pour la IBM SP2*

Les résultats sont présentés dans le tableau 6. Si aucune valeur de t ne vérifie l'inégalité, on laisse la case vide, cela signifie que l'on n'a pas, dans ce cas, intérêt à utiliser l'algorithme. Là encore, lorsqu'on applique l'algorithme parallèle à la base des champignons, on obtient un gain de temps par rapport à l'algorithme séquentiel, mais il est moins important que celui qu'on aurait avec la Multiprocessor Sun Sparc 20.

– enfin pour la Cray T3D, $p = 256$, $l = 387 flops$ et $g = 2.4 flops/word$. Notre inégalité devient alors :

$$\frac{20.303+10\times T+(3\times t+2)\times 2.4}{256} + \frac{2\times 387}{n} \leq 12.303 + 10 \times T$$
$$\text{soit } t \leq \frac{3124.465+2550\times T - \frac{198144}{n}}{7.2}$$

Les résultats sont présentés dans le tableau 7. Les paramètres de notre application se situent de nouveau dans de bons intervalles. C'est avec cette machine qu'on a la plus grande marge pour t, et donc avec elle qu'on peut espérer les meilleures performances.

Ainsi on peut espérer que notre algorithme parallèle nous fera gagner du temps par rapport à l'algorithme séquentiel pour ces machines. Nous allons maintenant étudier plus finement de quel ordre est l'accélération que nous pouvons espérer.

4.2.3 Etude de l'accélération de l'algorithme

Dans notre étude précédente, nous avons tenu compte de termes qui étaient négligeables devant d'autres. Nous allons ici les éliminer afin d'obtenir une formule plus simple, nous permettant d'estimer l'accélération de notre algorithme (l'objectif d'un algorithme parallèle étant bien sûr de se rapprocher autant que possible d'une accélération linéaire).

Une étude moins fine des étapes de sélection de bons individus et de reproduction, nous donne pour ces deux étapes une complexité de l'ordre de $2 \times (t + 1) \times g + l + 2 \times \frac{n}{p}$ et $t \times \frac{n}{p} \times g + l + \frac{n}{p} \times K$ respectivement. Nous ne gardons ici que les coûts des communications, synchronisations et calculs de qualités, les autres opérations pouvant être négligées car de faible coût comparés à ceux-ci. Les complexités de ces deux phases sont de l'ordre de $4 \times n$ et $n \times K$ respectivement en séquentiel. La complexité de l'algorithme parallèle est donc de l'ordre de $\frac{n}{p} \times ((3 \times t + 2) \times g + 2 + K) + 2 \times l$, elle est de l'ordre de $(K + 4) \times n$ pour le séquentiel. Sous la condition que $3 \times (t + 2) \times g$ soit négligeable comparé à K et $2 \times l$ soit négligeable comparé à $\frac{n \times K}{p}$, nous avons une complexité de l'ordre de $\frac{n \times K}{p}$ pour notre algorithme parallèle et nous pouvons donc prédire une accélération presque linéaire.

Nous avons déjà vu que pour les applications à des problèmes d'Extraction de Connaissances dans les Bases de Données, cette condition est satisfaite. Nous pouvons donc espérer une accélération presque linéaire lors de nos tests.

4.2.4 Expérimentations

L'algorithme séquentiel a été implanté en C, et nous avons utilisé la version C de la librairie BSPLib (version 1.3) fournie par l'Université d'Oxford [11] pour implanter le premier algorithme parallèle proposé. Nous avons fait des tests en appliquant ces algorithmes à la base des champignons, afin d'apprendre une règle permettant de reconnaître un champignon comestible. Ces tests ont été effectués sur une biprocesseur Sun Ultra Sparc2, 166MHz et 128Mo de mémoire partagée. Pour faire nos mesures, nous avons fait varier le nombre de tuples. Les résultats pour 100 générations d'une population de 1000 individus sont présentés sur la figure 13.

Ces premiers résultats sont conformes aux prédictions faites avec le modèle BSP. Notre algorithme parallèle exécuté sur deux processeurs est à peu près deux fois plus rapide que l'algorithme séquentiel. En fait, il est même plus de deux fois plus rapide que l'algorithme séquentiel, ce qui peut être expliqué par des problèmes de swap.

Des Tests sont actuellement en cours sur une Origin 2000 (Silicon Graphics), afin d'observer le comportement de l'algorithme parallèle lorsque le nombre de processeurs augmente et de vérifier si, comme nous le permet d'espérer l'étude faite avec BSP, l'accélération de cet algorithme est presque linéaire.

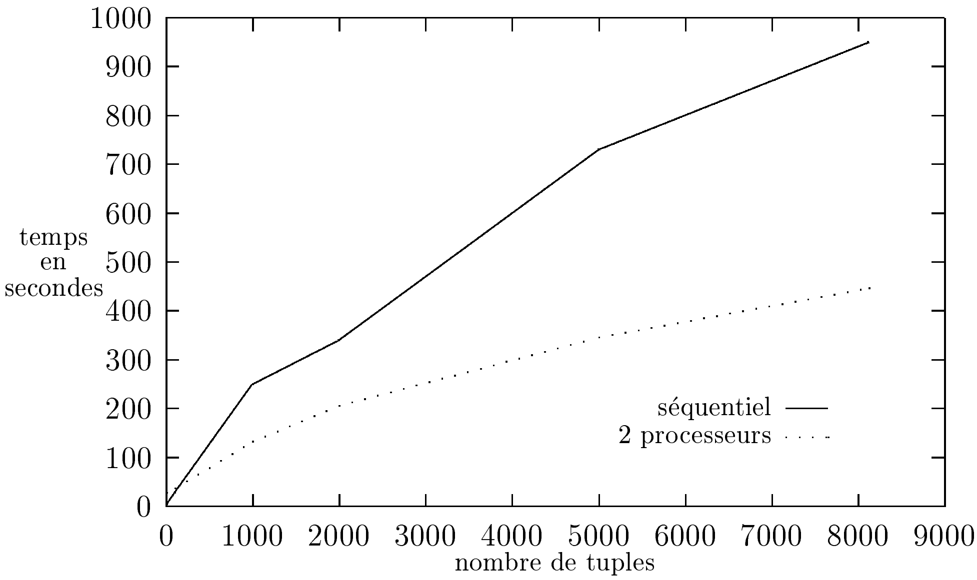

FIG. 13 – *Tests sur la base de données des champignons*

4.3 Etude de deux algorithmes utilisant une base de données répartie

4.3.1 Population centralisée et base de données répartie

4.3.1.1 Description. Dans cet algorithme, on répartit la base de données, mais la population se trouve sur un seul processeur et les traitements autres que le calcul de la qualité s'effectuent en séquentiel sur ce processeur. On peut ainsi paralléliser le calcul de la fonction de qualité pour chaque individu ; il suffit pour cela de réaliser sur chaque processeur le travail correspondant aux tuples qui lui sont attribués, puis de regrouper les résultats. Dans le cas d'une base de données composée d'un grand nombre de tuples, le temps nécessaire pour le calcul de la qualité d'un individu peut devenir très important, et cette méthode nous permettrait peut-être de le réduire de façon significative. Cette façon de calculer la qualité est présentée sur la figure 14.

Description des étapes de la figure :

1. le processeur 1 envoie tous les individus de la population à tous les autres processeurs ;
2. chaque processeur calcule, pour chacun des individus de la population, le nombre d'exemples de la base de données locale couverts et le nombre de contre-exemples rejetés ;
3. les processeurs envoient les résultats des calculs de couverture au processeur 1 ;

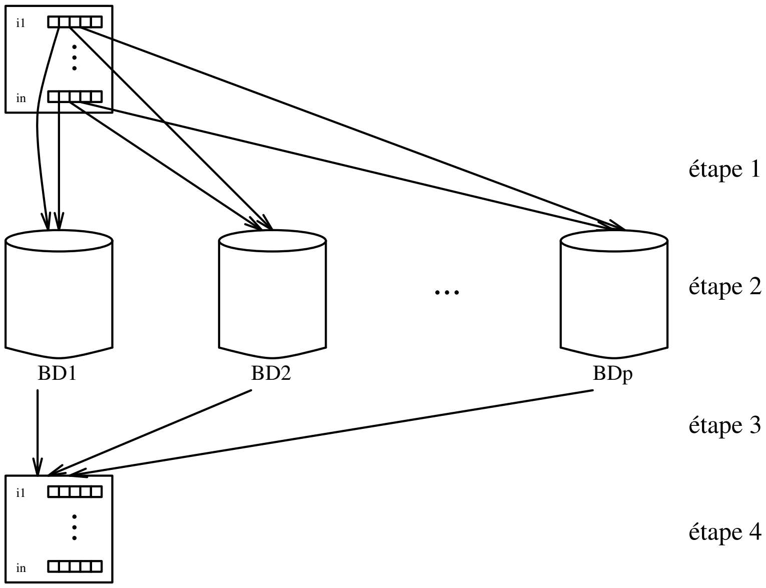

FIG. 14 – *Evaluation de la population dans le cas d'une base de données répartie et d'une population centralisée*

4. le processeur 1 fusionne les résultats qu'il a reçus et obtient ainsi la qualité des individus de la population.

Le reste de l'algorithme est identique à l'algorithme séquentiel.

4.3.1.2 Etude du coût. Nous allons étudier le coût de la phase de calcul de la qualité des individus de la population, afin de pouvoir le comparer à celui de l'algorithme séquentiel. Pour cela, reprenons chacune des étapes de ce calcul et regardons son coût :

1. le processeur 1 envoie n individus à tous les processeurs (donc $n \times p$ en tout) et chacun en reçoit n. Il s'agit donc d'une $(n \times p)$-relation qui aura un coût de $n \times p \times t \times g$ et sera suivie d'une barrière de synchronisation, d'où un coût de $n \times p \times t \times g + l$;
2. chaque processeur effectue pour les n individus un calcul de couverture sur sa base locale de taille $\frac{T}{p}$. Le coût pour un individu est de $i \times \frac{T}{p}$, où i est le nombre d'instructions de base nécessaires pour calculer la qualité d'un individu. Cette étape a donc un coût de $i \times n \times \frac{T}{p}$ ($10 \times n \times \frac{T}{p}$ pour notre application) ;
3. chaque processeur envoie ses n résultats au processeur 1, qui en reçoit par conséquent $n \times p$. Cela induit une $(n \times p)$-relation d'un coût de $n \times p \times g$. Il y a ensuite une barrière de synchromisation, d'où un coût total pour l'étape de $n \times p \times g + l$;
4. le processeur 1 fait la somme des p résultats qu'il a reçus pour chaque individu, ce qui donne un coût de $n \times p$.

Ainsi, la somme de ces étapes a un coût de $n \times p \times t \times g + l + 10 \times n \times \frac{T}{p} + n \times p \times g + l + n \times p$. La phase de calcul des qualités des individus de la population a donc un coût de $n \times p \times (g \times (t+1) + 1) + 10 \times n \times \frac{T}{p} + 2 \times l$.

La même phase dans l'algorithme séquentiel ayant un coût de $10 \times n \times T$, il faudrait $n \times p \times t \times g + 2 \times l + 10 \times n \times \frac{T}{p} \leq 10 \times n \times T$ pour espérer un gain de temps.

Simplifions cette inégalité :

$$n \times p \times (g \times (t+1) + 1) + 10 \times n \times \frac{T}{p} + 2 \times l \leq 10 \times n \times T$$
$$\text{soit } n \times p \times (g \times (t+1) + 1) + 2 \times l \leq 10 \times n \times T \times (1 - \tfrac{1}{p})$$
$$\text{soit } p \times (g \times (t+1) + 1) + \tfrac{2 \times l}{n} \leq 10 \times T \times (1 - \tfrac{1}{p})$$

Nous allons maintenant fixer les paramètres g, p et l en utilisant les valeurs du tableau 4, puis nous étudierons pour des valeurs réalistes des paramètres T et n, quelles sont les valeurs de t pour lesquelles notre algorithme parallèle serait avantageux :

- pour la Multiprocessor Sun Sparc 20, on a $p = 4$, $l = 118 flops$ et $g = 4.1 flops/word$. Notre inégalité devient alors :
$$4 \times (4.1 \times (t+1) + 1) + \tfrac{2 \times 118}{n} \leq 10 \times T \times (1 - \tfrac{1}{4})$$
$$\text{soit } t \leq \frac{7.5 \times T - 20.4 - \frac{236}{n}}{16.4}$$
Les résultats présentés dans le tableau 8 nous permettent d'espérer de meilleurs temps avec cet algorithme parallèle qu'avec l'algorithme séquentiel pour notre application.
- pour la IBM SP2, on a $p = 8$, $l = 88795 flops$ et $g = 1224.1 flops/word$. Notre inégalité devient :
$$8 \times (1224.1 \times (t+1) + 1) + \tfrac{2 \times 88795}{n} \leq 10 \times T \times (1 - \tfrac{1}{8})$$
$$\text{soit } t \leq \frac{8.75 \times T - 9800.8 - \frac{177590}{n}}{9792.8}$$
Pour cette machine, on peut espérer, dans certains cas, de meilleurs résultats qu'avec l'algorithme séquentiel pour notre application (voir tableau 9). Cependant, il faut utiliser des bases de données de taille importante pour obtenir un gain de temps ; notons tout de même que nous avons estimé une borne inférieure pour le nombre d'instructions nécessaires pour calculer la qualité d'un individu.
- enfin pour la Cray T3D, on a $p = 256$, $l = 387 flops$ et $g = 2.4 flops/word$. Notre inégalité devient :
$$256 \times (2.4 \times (t+1) + 1) + \tfrac{2 \times 387}{n} \leq 10 \times T \times (1 - \tfrac{1}{256})$$
$$\text{soit } t \leq \frac{\frac{2550}{256} \times T - 870.4 - \frac{774}{n}}{614.4}$$
Cette fois encore, les paramètres de notre application nous permettent d'espérer de meilleures performances avec cet algorithme qu'avec l'algorithme séquentiel (voir tableau 10). Cependant, on constate que, contrairement à l'algorithme dans lequel on répartit la population et les tâches à effectuer, cette machine ne nous donne pas les meilleurs résultats.

Ainsi dans la plupart des cas, il semble qu'on gagne du temps en parallélisant de cette manière. Il serait donc intéressant d'implanter un tel algorithme. Cependant il faut que la base de données sur laquelle on travaille soit de taille assez importante, et les gains seront probablement moins importants qu'avec le premier algorithme étudié.

4.3.2 Population et base de données réparties

4.3.2.1 Description. Une autre solution possible serait de répartir la population et la base de données sur les différents processeurs. On peut alors paralléliser les traitements comme dans le premier algorithme proposé et le calcul de la qualité de chaque individu comme dans le deuxième.

L'évaluation de la population pour cette approche est présentée sur la figure 15.

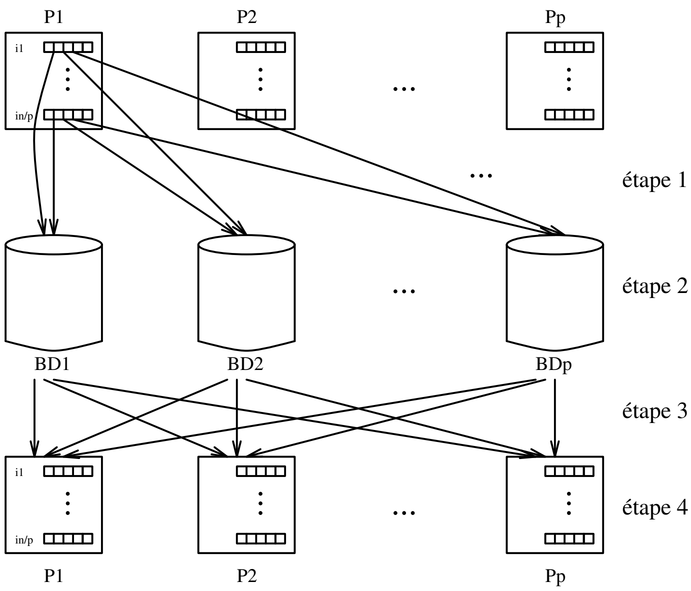

FIG. 15 – *Evaluation de la population dans le cas d'une base de données et d'une population réparties*

Description des étapes de la figure :

1. chaque processeur envoie tous ses individus à tous les autres processeurs. Ainsi chaque processeur possède tous les individus de l'ensemble de la population ;
2. chaque processeur calcule, pour chacun des individus de la population totale, le nombre d'exemples de la base de données locale couverts et le nombre de contre-exemples rejetés ;
3. les processeurs envoient les résultats des calculs de couverture aux processeurs concernés ;

4. les processeurs fusionnent les résultats des individus de leur sous-population et obtiennent ainsi leur qualité.

Le reste de l'algorithme est identique au premier proposé.

4.3.2.2 Etude du coût.

Nous allons étudier si cet algorithme serait plus avantageux que celui où la population est répartie et la base de données ne l'est pas. Seule la phase de calcul de la qualité des individus diffère. Pour le premier algorithme, elle était de $10 \times \frac{n}{p} \times T$.

Dans ce nouvel algorithme, il faut d'abord procéder à l'envoi par tous les processeurs de tous leurs individus à tous les autres processeurs. Ainsi, chaque processeur envoie $\frac{n}{p}$ individus à tous les autres, soit n en tout, et en reçoit n également ($\frac{n}{p}$ de chaque processeur). Il s'agit donc d'une n-relation qui a un coût de $n \times t \times g$ et est suivie d'une barrière de synchronisation.

Ensuite les p processeurs calculent, pour chacun des n individus de la population totale, le nombre d'exemples de la base locale couverts et le nombre de contre-exemples rejetés. Le coût pour un individu sur un processeur est de $10 \times \frac{T}{p}$. Comme chaque processeur doit le faire pour les n individus de la population, on arrive à un coût de $10 \times n \times \frac{T}{p}$ pour l'étape.

L'étape suivante consiste, pour chaque processeur, à envoyer les résultats de ses calculs aux processeurs concernés. Chaque processeur envoie ainsi n résultats et en reçoit $\frac{n}{p}$ de chaque processeur, donc n en tout. C'est une n-relation d'un coût de $n \times g$, qui est suivie d'une barrière de synchronisation.

Enfin, chaque processeur calcule la qualité des individus de sa sous-population en faisant pour chacun de ses $\frac{n}{p}$ individus la somme des p résultats qu'il a reçus. Cette étape a un coût de n.

La somme de ces différentes étapes nous donne un coût total de $n \times t \times g + l + 10 \times n \times \frac{T}{p} + n \times g + l + n$. Le troisième terme est équivalent au coût de l'évaluation dans notre premier algorithme, on peut donc en conclure immédiatement que celui-ci est plus lent. En effet, on réalise le même travail que dans le premier algorithme présenté, mais avec des communications en plus.

5 Conclusion

Ce rapport présente une méthode possible pour paralléliser un algorithme génétique général sans modifier son fonctionnement, afin de conserver les propriétés connues dans le cas séquentiel. L'objectif est uniquement d'améliorer les temps d'exécution de ces algorithmes qui peuvent être très coûteux. L'utilisation du modèle BSP est particulièrement intéressante, car elle nous permet de faire une prédiction du coût en temps d'un algorithme écrit en suivant ce modèle, et donc de déterminer quelles pistes il peut être intéressant d'explorer. Nous avons étudié une application à l'Extraction de Connaissances dans les bases de données, car pour ce type d'applications l'évaluation des individus est souvent très coûteuse. Nous proposons deux autres méthodes spécifiques au Data Mining, car utilisant une base de données répartie pour le calcul de la qualité des individus, qui préservent également le fonctionnement de l'algorithme séquentiel.

Les trois méthodes proposées sont les suivantes : répartition de la population et des traitements, répartition de la base de données, et enfin répartition de la population et de la base de données. Pour chacun des algorithmes proposés, nous avons fait une étude de coût avec le modèle BSP qui nous permet d'espérer un bénéfice pour les deux premiers. Nous avons choisi d'étudier plus finement le premier, notamment son accélération, et nous l'avons implanté afin de vérifier les prédictions faites avec le modèle BSP. Les premières expérimentations confirment notre étude théorique.

Cette méthode est proche du travail présenté dans [4]. Dans leur approche, chaque processeur divise sa population en p parties et envoie chacune à un processeur différent. Dans notre cas, chaque fois qu'un processeur émet une requête à destination d'un autre processeur afin de recevoir un de ses individus, le numéro du processeur et celui de l'individu sont tirés aléatoirement ; il pourrait même arriver qu'un processeur demande tous ses individus à un même processeur. Ainsi le comportement stochastique de l'algorithme génétique séquentiel est bien préservé.

Nous remercions Gaétan Hains et Lionel Martin pour leurs discussions fructueuses sur ce sujet.

T	n	t
10	50	[0,3425]
	100	[0,3700]
	500	[0,3920]
	1000	[0,3948]
50	50	[0,17591]
	100	[0,17867]
	500	[0,18087]
	1000	[0,18114]
100	50	[0,35300]
	100	[0,35575]
	500	[0,35795]
	1000	[0,35823]
500	50	[0,176966]
	100	[0,177242]
	500	[0,177462]
	1000	[0,177489]
1000	50	[0,354050]
	100	[0,354325]
	500	[0,354545]
	1000	[0,3541970]
10000	50	[0,3541550]
	100	[0,3541825]
	500	[0,3542045]
	1000	[0,3542073]
100000	50	[0,35416550]
	100	[0,35416825]
	500	[0,35417045]
	1000	[0,35417073]

TAB. 7 – *Résultats pour la Cray T3D*

T	n	t
10	[50,1000]	[0,3]
50	[50,1000]	[0,21]
100	[50,1000]	[0,44]
500	[50,1000]	[0,227]
1000	[50,100] [500,1000]	[0,455] [0,456]
10000	[50,1000]	[0,4571]
100000	[50,1000]	[0,45730]

TAB. 8 – *Résultats pour la Multiprocessor Sun Sparc 20*

T	n	t
[10,1000]	[50,1000]	
8000	[50,100] [500,1000]	[0,5] [0,6]
10000	[50,1000]	[0,7]
100000	50 [100,1000]	[0,87] [0,88]

TAB. 9 – *Résultats pour la IBM SP2*

T	n	t
[10,100]	[50,1000]	
500	[50,1000]	[0,6]
1000	[50,1000]	[0,14]
10000	[50,1000]	[0,160]
100000	[50,1000]	[0,1619]

TAB. 10 – *Résultats pour la Cray T3D*

Références

[1] David Beasley, David R. Bull, Ralph R. Martin. *An Overview of Genetic Algortihms: Part 1, Fundamentals.* University Computing, 15(2) 58-69, 1993.

[2] David Beasley, David R. Bull, Ralph R. Martin. *An Overview of Genetic Algortihms: Part 2, Research Topics.* University Computing, 15(4) 170-181, 1993.

[3] Erick Cant-Paz. *A Survey of Parallel Genetic Algorithms.* Calculateurs Paralleles, Reseaux et Systems Repartis. Vol. 10, No. 2. pp. 141-171. Paris: Hermes. 1998.

[4] E. Cant-Paz, D.E. Goldberg. *Parallel genetic algorithms with distributed panmictic populations.* IlliGAL Report No. 99006, 1999.

[5] Kenneth A. De Jong. *An analysis of the behavior of a class of genetic adaptative systems.* Ph.D. Thesis, University of Michigan, 1975.

[6] Kenneth A. de Jong, William M. Spears. *Learning Concept Classification Rules Using Genetic Algorithms.* Proceedings of the Int'l Joint Conference on Artificial Intelligence, 651-656, 1991.

[7] Lawrence J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution.* Wiley, New York, 1966.

[8] A.A. Freitas, S.H. Lavington. *Mining Very Large Databases with Parallel Processing.* Kluwer Academic Publishers, 1998.

[9] Goldberg, D.E. *Genetic algorithms in search, optimization, and machine learnng.* Addison-Wesley, 1989.

[10] F. Herrera, M. Lozano, J.L. Verdegay. *Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis*, 1995.

[11] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. *BSPlib: The BSP Programming Library.* Technical report May, Oxford University Computing Laboratory, 1997.

[12] J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1975.

[13] J.R. Koza, *Genetic Programming.* The MIT Press, 1992.

[14] J.R. Koza, *Genetic Programming II.* The MIT Press, 1994.

[15] Ingo Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog Verlag, Stuttgart, 1973.

[16] J.S. Schlimmer. *Concept Acquisition Through Representational Adjustment.* Technical Report 87-19, Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine, 1987.

[17] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 26 of Interdisciplinary systems research, Birkhäuser, Basel, 1977.

[18] Hans-Paul Schwefel. *On the Evolution of Evolutionary Computation.* University of Dortmund.

[19] David B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. *Questions and Answers about BSP*. Technical report 15-96, Programming Research Group, Oxford University Computing Laboratory, August 1996.

[20] Leslie G. Valiant. *A bridging model for parallel computation.* Communications of the Association for Computing Machinery, 33(8):103-111, 1990.

Algorithmes

Annexe A
L'algorithme parallèle avec répartition de la population

La population est répartie équitablement sur les processeurs. Chaque processeur exécute l'algorithme suivant, où *get* est une opération de lecture qui permet de lire le contenu de la mémoire locale d'un autre processeur.

début
    terminé ← **faux**
    engendrer $\frac{n}{p}$ individus aléatoirement et les ranger dans les champs individu de pop
    calculer la qualité de chaque individu de pop et la ranger dans le champ qualité

    tant que non terminé **faire**
    début /* produire une nouvelle génération */

        /* phase de sélection : remplissage de la *mating pool* par */
        /* la technique des tournois */
        /* remplir le tableau tournoi */
        pour i de 0 à $\frac{n}{p} - 1$ **faire**
            /* tirage d'un couple pour un tournoi */
        début
            /* tirage du 1er individu */
            tirer aléatoirement un nombre entre 0 et n-1 → ind1
            get pop[ind1 mod $\frac{n}{p}$].individu à partir du processeur (ind1 div $\frac{n}{p}$)
                → tournoi[2i]
            /* tirage du 2eme individu */
            tirer aléatoirement un nombre entre 0 et n-1 → ind2
            get pop[ind2 mod $\frac{n}{p}$].individu à partir du processeur (ind2 div $\frac{n}{p}$)
                → tournoi[2i+1]
        fin pour
        synchronisation

        /* tournoi et remplissage de la *mating pool* */
        pour i de 0 à $\frac{n}{p} - 1$ **faire**
        début
            si tournoi[2i].qualité ≥ tournoi[2i+1].qualité **alors**
                mating_pool[i] ← tournoi[2i].individu
            sinon
                mating_pool[i] ← tournoi[2i+1].individu
            fin si
        fin pour

        /* phase de reproduction */
        /* remplissage du tableau pop par les couples d'individus qui pourront */

```
/* se reproduire */
pour i de 0 à n/(2p) − 1 faire
début /* sélection de deux individus */
    tirer aléatoirement un nombre entre 0 et n-1 → ind1
    get mating_pool[ind1 mod n/p].individu à partir du processeur (ind1 div n/p)
            → pop[2i].individu
    tirer aléatoirement un nombre entre 0 et n-1 → ind2
    get mating_pool[ind2 mod n/p].individu à partir du processeur (ind2 div n/p)
            → pop[2i+1].individu
fin pour
synchronisation

pour i de 0 à n/(2p) − 1 faire
début /* cycle de reproduction */
    /* croisement */
    tirer aléatoirement un nombre entre 0 et 1 → p_c
    si p_c ≤ 0.6 alors
        /* croisement des deux individus */
        tirer aléatoirement un nombre entre 1 et k-1 → pt_crois
        croiser pop[2i].individu et pop[2i+1].individu au point pt_crois
        mettre les nouveaux individus dans pop[2i].individu
            et pop[2i+1].individu
    fin si
    /* mutation */
    /* mutation du 1er descendant */
    tirer aléatoirement un nombre entre 0 et 1 → p_m
    si p_m ≤ 0.001 alors
        tirer aléatoirement un entier entre 0 et k-1 → pt_mut
        effectuer une mutation sur l'individu pop[2i].individu
            du gène de la case pt_mut
    fin si
    /* mutation du 2eme descendant */
    tirer aléatoirement un nombre entre 0 et 1 → p_m
    si p_m ≤ 0.001 alors
        tirer aléatoirement un entier entre 0 et k-1 → pt_mut
        effectuer une mutation sur l'individu pop[2i+1].individu
            du gène de la case pt_mut
    fin si
    calculer la qualité des individus pop[2i].individu et pop[2i+1].individu
fin pour
synchronisation

si la population a convergé alors
    terminé ← vrai
fin si
```

fin tant que

fin

Le tirage d'un individu sur un processeur quelconque pourrait se faire en lançant deux dés, l'un dont la valeur serait comprise entre 0 et p-1 correspondant au numéro du processeur, l'autre dont la valeur serait comprise entre 0 et $\frac{n}{p} - 1$ qui correspondrait au numéro de l'individu sélectionné sur le processeur. Cependant pour se conformer au maximum à l'algorithme séquentiel dans lequel il n'y a qu'un seul tirage et pour pallier la fragilité des dés, on préfère ne tirer qu'un seul dé dont les valeurs sont comprises entre 1 et n. Notons v la valeur tirée, proc=v DIV $\frac{n}{p}$ correspondra au numéro du processeur sélectionné et v MOD $\frac{n}{p}$ correspondra au numéro de l'individu sélectionné sur le processeur proc.

Annexe B
L'algorithme parallèle avec répartition de la base de données et de la population

La base de données est répartie sur les processeurs, la population également. Chaque processeur effectue les instructions de l'algorithme suivant.

début
    terminé $\leftarrow$ **faux**
    engendrer $\frac{n}{p}$ individus aléatoirement et les ranger dans les champs individu de pop
    calculer la qualité des individus i de pop $\rightarrow$ pop[i].qualité :
        envoi des individus de la population locale aux autres processeurs
        synchronisation
        calcul du nombre d'exemples de la base locale couverts et de contre-exemples
            rejetés pour tous les individus de la population
        envoi des résultats aux processeurs concernés
        synchronisation
        fusion des résultats concernant la population locale
            $\rightarrow$ qualité des individus de la sous-population

    tant que non terminé **faire**
    début /* produire une nouvelle génération */

        /* phase de sélection : remplissage de la *mating pool* par */
        /* la technique des tournois */
        /* remplir le tableau tournoi */
        pour i de 0 à $\frac{n}{p} - 1$ **faire**
            /* tirage d'un couple pour un tournoi */
        début
            /* tirage du 1er individu */
            tirer aléatoirement un nombre entre 0 et n-1 $\rightarrow$ ind1
            get pop[ind1 mod $\frac{n}{p}$].individu à partir du processeur (ind1 div $\frac{n}{p}$)
                $\rightarrow$ tournoi[2i].individu
            /* tirage du 2eme individu */
            tirer aléatoirement un nombre entre 0 et n-1 $\rightarrow$ ind2
            get pop[ind2 mod $\frac{n}{p}$].individu à partir du processeur (ind2 div $\frac{n}{p}$)
                $\rightarrow$ tournoi[2i+1].individu
        fin pour
        synchronisation

        /* tournoi et remplissage de la *mating pool* */
        pour i de 0 à $\frac{n}{p} - 1$ **faire**
        début
            si tournoi[2i].qualité $\geq$ tournoi[2i+1].qualité **alors**
                mating_pool[i] $\leftarrow$ tournoi[2i].individu

```
    sinon
        mating_pool[i] ← tournoi[2i+1].individu
    fin si
fin pour

/* phase de reproduction */
/* remplissage du tableau pop par les couples d'individus qui pourront */
/* se reproduire */
pour i de 0 à n/(2p) − 1 faire
début /* sélection de deux individus */
    tirer aléatoirement un nombre entre 0 et n-1 → ind1
    get mating_pool[ind1 mod n/p].individu à partir du processeur (ind1 div n/p)
            → pop[2i].individu
    tirer aléatoirement un nombre entre 0 et n-1 → ind2
    get mating_pool[ind2 mod n/p].individu à partir du processeur (ind2 div n/p)
            → pop[2i+1].individu
fin pour
synchronisation

pour i de 0 à n/(2p) − 1 faire
début /* cycle de reproduction */
    /* croisement */
    tirer aléatoirement un nombre entre 0 et 1 → p_c
    si p_c ≤ 0.6 alors
        /* croisement des deux individus */
        tirer aléatoirement un nombre entre 1 et k-1 → pt_crois
        croiser pop[2i].individu et pop[2i+1].individu au point pt_crois
        mettre les nouveaux individus dans pop[2i].individu et
            pop[2i+1].individu
    fin si
    /* mutation */
    /* mutation du 1er descendant */
    tirer aléatoirement un nombre entre 0 et 1 → p_m
    si p_m ≤ 0.001 alors
        tirer aléatoirement un entier entre 0 et k-1 → pt_mut
        effectuer une mutation sur l'individu pop[2i].individu du gène
            de la case pt_mut
    fin si
    /* mutation du 2eme descendant */
    tirer aléatoirement un nombre entre 0 et 1 → p_m
    si p_m ≤ 0.001 alors
        tirer aléatoirement un entier entre 0 et k-1 → pt_mut
        effectuer une mutation sur l'individu pop[2i+1].individu du gène
            de la case pt_mut
    fin si
```

```
        fin pour
        synchronisation

        calculer la qualité des individus de pop :
            envoi des individus de la population locale aux autres processeurs
            synchronisation
            calcul du nombre d'exemples de la base locale couverts et de
                contre-exemples rejetés pour tous les individus de la population
            envoi des résultats aux processeurs concernés
            synchronisation
            fusion des résultats concernant la population locale
                → qualité des individus de la sous-population

        synchronisation

        si la population a convergé alors
            terminé ← vrai
        fin si

    fin tant que

fin
```

Annexe C
L'algorithme parallèle avec répartition de la base de données

La base de données est répartie sur les p processeurs, les n individus de la population sont sur un seul processeur.L'algorithme est le suivant.

```
début
    terminé ← faux
    le processeur d'indice 0 engendre n individus aléatoirement et les range
        dans les champs individu de pop
    calcul de la qualité des individus de pop :
        le processeur d'indice 0 envoie les individus de pop aux autres processeurs
        synchronisation
        calcul du nombre d'exemples de la base locale couverts et de
            contre-exemples rejetés pour tous les individus de la population
        envoi des résultats au processeur 0
        synchronisation
        fusion des résultats par le processeur d'indice 0
            → qualité des individus de la population

    tant que non terminé faire
    début /* le processeur d'indice 0 produit une nouvelle génération */

        /* phase de sélection : remplissage de la mating pool par */
        /* la technique des tournois */
        pour i de 0 à n-1 faire
            /* tirage d'un couple pour un tournoi */
        début
            /* tirage du 1er individu */
            tirer aléatoirement un nombre entre 0 et n-1 → ind1
            /* tirage du 2eme individu */
            tirer aléatoirement un nombre entre 0 et n-1 → ind2
            si pop[ind1].qualité ≥ pop[ind2].qualité alors
                mating_pool[i] ← pop[indi1].individu
        fin pour
        synchronisation

        /* phase de reproduction */
        pour i de 0 à n/2 − 1 faire
            /* croisement */
            tirer aléatoirement un nombre entre 0 et n-1 → ind1
            tirer aléatoirement un nombre entre 0 et n-1 → ind2
            tirer aléatoirement un nombre entre 0 et 1 → p_c
            si p_c ≤ 0.6 alors
                /* croisement des deux individus */
```

```
            tirer aléatoirement un nombre entre 1 et k-1 → pt_crois
            croiser mating_pool[ind1] et mating_pool[ind2]
                au point pt_crois
            mettre les nouveaux individus dans pop[2i].individu
                et pop[2i+1].individu
        sinon
            /* duplication des individus pour les mettre */
            /* dans la nouvelle population */
            pop[2i].individu ← mating_pool[ind1]
            pop[2i+1].individu ← mating_pool[ind2]
        fin si
        /* mutation du premier descendant */
        tirer aléatoirement un nombre entre 0 et 1 → p_m
        si p_m ≤ 0.001 alors
            tirer aléatoirement un entier entre 0 et k-1 → pt_mut
            effectuer une mutation sur l'individu pop[2i].individu du gène
                de la case pt_mut
        fin si
        /* mutation du second descendant */
        tirer aléatoirement un nombre entre 0 et 1 → p_m
        si p_m ≤ 0.001 alors
            tirer aléatoirement un entier entre 0 et k-1 → pt_mut
            effectuer une mutation sur l'individu pop[2i+1].individu du gène
                de la case pt_mut
        fin si
    fin pour

    calcul de la qualité des individus de pop :
        le processeur d'indice 0 envoie les individus de pop aux autres processeurs
        synchronisation
        calcul du nombre d'exemples de la base locale couverts et de
            contre-exemples rejetés pour tous les individus de la population
        envoi des résultats au processeur 0
        synchronisation
        fusion des résultats par le processeur d'indice 0
            → qualité des individus de la population

    si la population a convergé alors
        terminé ← vrai
    fin si

  fin tant que

fin
```