UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉVELOPPEMENT ET PARALLÉLISATION D'ALGORITHMES BIOINFORMATIQUES POUR LA RECONSTRUCTION D'ARBRES PHYLOGÉNÉTIQUES ET DE RÉSEAUX RÉTICULÉS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR
ALPHA BOUBACAR DIALLO

SEPTEMBRE 2007

REMERCIEMENTS

Je présente mes remerciements au Dr. Vladimir Makarenkov, mon directeur de recherche, pour son suivi, ses conseils et ses suggestions lors de la réalisation de ce projet de maîtrise. Qu'il trouve ici toute l'expression de ma gratitude et de ma profonde reconnaissance.

Je remercie également mes collègues Alix Boc, Abdoulaye Baniré Diallo, Abdellah Mazouzi et Pablo Zentilli pour leurs points de vue et leur soutien qui ont été d'une grande utilité.

Mes remerciements s'adressent aussi à ma famille ainsi qu'à mes amis pour leur encouragement et leur soutien moral.

J'adresse ma totale reconnaissance à Génome Québec qui a contribué au financement de ce projet d'études.

À tout ceux qui ont contribué de près ou de loin à la réalisation de ce projet, qu'ils trouvent ici mes remerciements les plus sincères.

TABLE DES MATIÈRES

TABLE DES MATIÈRES II

LISTE DES FIGURES VI

LISTE DES TABLEAUX XI

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES XIII

RÉSUMÉ XIV

INTRODUCTION 1

CHAPITRE I 4

RECONSTRUCTION DE PHYLOGÉNIES 4

1.1 Introduction 4

1.2 Définitions de base 6

1.3 Arbres phylogénétiques 10

1.4 Réseaux réticulés 13

1.5 Méthodes de reconstruction d'arbres phylogénétiques 15

1.6 Références 27

CHAPITRE II 31

PARALLÉLISATION DES PROGRAMMES INFORMATIQUES 31

2.1 Parallélisme *31*

2.2 Machine virtuelle parallèle (Parallel Virual Machine) *41*

2.3 Interface de Passage de Messages (Message Passing Interface) *42*

2.4 Références *60*

CHAPITRE III **62**

ARTICLES 62

3.1 Une nouvelle méthode pour l'estimation de nucléotides manquants en vue de l'inférence phylogénétique *64*

3.2 New efficient algorithm for modeling partial and complete gene transfer scenarios *70*

3.3 Algorithms for detecting complete and partial horizontal gene transfers: theory and practice *79*

CHAPITRE IV **103**

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT DES MÉTHODES DE DISTANCES 103

4.1 Introduction *103*

4.2 Forme de parallélisme employée *104*

4.3 Neighbor-Joining *106*

4.4 Unweighted Neighbor-Joining *111*

4.5 ADDTREE *112*

4.6 Reconstruction par ordre circulaire *116*

4.7 Moindres carrés pondérés *120*

4.8 BioNJ *124*

4.9 FITCH *125*

4.10 Références *131*

CHAPITRE V **132**

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT LA MÉTHODE DE MAXIMUM DE VRAISEMBLANCE 132

5.1 Introduction *132*

5.2 Forme de parallélisme employée *133*

5.3 PHYML *134*

5.4 DNAML *137*

5.5 DnaMLK *141*

5.6 ProML *144*

5.7 ProMLK *148*

5.8 Références *152*

CHAPITRE VI **153**

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT LA MÉTHODE DE MAXIMUM DE PARCIMONIE 153

6.1 Introduction *153*

6.2 DNAPARS *154*

6.3 PARS *158*

6.4 DOLLOP *162*

6.5 Références *167*

CHAPITRE VII **168**

PARALLÉLISATION DES ALGORITHMES DE DÉTECTION DE TRANSFERTS HORIZONTAUX DE GÈNES ET DE RECONSTRUCTION DE RÉTICULOGRAMMES 168

7.1 Transferts horizontaux de gènes *168*

7.2 Reconstruction de réticulogrammes *177*

7.3 Références *180*

CONCLUSION ET PERSPECTIVES 181

ANNEXE A 183

ANNEXE B 194

LISTE DES FIGURES

Figure I-1 Arbre d'évolution de Haeckel (1866). 5

Figure I-2 Un arbre phylogénétique. 10

Figure I-3 Deux arbres enracinés (a et b) et un arbre non enraciné (c). 12

Figure I-4 Quatre représentations d'un arbre phylogénétique (obtenues à l'aide du logiciel T-Rex (Makarenkov, 2001)). 13

Figure I-5 Un réseau réticulé. 14

Figure I-6 Processus d'inférence d'arbres phylogénétiques par la méthode de distances 17

Figure I-7 Recouvrement de toutes les topologies par recherche exhaustive. 22

Figure I-8 Longueur d'arbre pour trois topologies (Harry, 2001). 22

Figure II-1 Architecture SISD. 35

Figure II-2 Architecture SIMD. 35

Figure II-3 Architecture MISD. 36

Figure II-4 Architecture MIMD. 37

Figure II-5 Architecture « share everything ». 38

Figure II-6 Architecture « share nothing ». 39

Figure II-7 Architecture « shared disks ». 41

Figure II-8 Système à mémoire distribuée. 45

Figure II-9 Structure d'un programme MPI. 46

Figure II-10 Communicateur ou groupe MPI. 47

Figure II-11 Envoi et réception de messages. 51

Figure II-12 Envoi de messages avec un tampon de mémoire. 53

Figure IV-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de distances. 104

Figure IV-2 Arbre en étoile (a) et son développement (b) (Saitou et Nei, 1987). 107

Figure IV-3 Algorithme NJ : Temps écoulé en secondes en fonction de la taille des données d'entrée. 109

Figure IV-4 Configurations possibles d'arbres additifs binaires avec quatre feuilles. 113

Figure IV-5 Algorithme ADDTREE : Temps écoulé en secondes en fonction de la taille des données d'entrée. 116

Figure IV-6 Algorithme de reconstruction par ordre circulaire : Temps écoulé en secondes en fonction de la taille des données d'entrée. 120

Figure IV-7 Algorithme MW : Temps écoulé en secondes en fonction de la taille des données d'entrée. 123

Figure IV-8 Reconstruction par la méthode Fitch. 126

Figure IV-9 Interface de T-Rex avec la méthode Fitch. 127

Figure IV-10 Algorithme Fitch : Temps écoulé en secondes en fonction de la taille des données d'entrée. 129

Figure V-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de maximum de vraisemblance. 133

Figure V-2 Interface de T-Rex avec la méthode PHYML. 135

Figure V-3 Algorithme PHYML : Temps écoulé en secondes en fonction de la taille des données d'entrée. 137

Figure V-4 Interface de T-Rex avec la méthode DNAML 139

Figure V-5 Algorithme DNAML : Temps écoulé en secondes en fonction de la taille des données d'entrée. 141

Figure V-6 Interface de T-Rex avec la méthode DNAMLK. 142

Figure V-7 Algorithme DNAMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée. 144

Figure V-8 Interface de T-Rex avec la méthode PROML 146

V-9 Algorithme PROML : Temps écoulé en secondes en fonction de la taille des données d'entrée 148

Figure V-10 Interface de T-Rex avec la méthode PROMLK. 149

Figure V-11 Algorithme PROMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée. 151

Figure VI-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de maximum de parcimonie. 154

Figure VI-2 Interface de T-Rex avec la méthode DNAPARS 156

VI-3 Algorithme DNAPARS : Temps écoulé en secondes en fonction de la taille des données d'entrée.....158

Figure VI-4 Interface de T-Rex avec la méthode PARS.....160

Figure VI-5 Algorithme PARS : Temps écoulé en secondes en fonction de la taille des données d'entrée.....162

Figure VI-6 Interface de T-Rex avec la méthode DOLLOP.....164

Figure VI-7 Algorithme DOLLOP : Temps écoulé en secondes en fonction de la taille des données d'entrée.....166

Figure VII-1 Transfert horizontal de gènes (3 mécanismes).....169

Figure VII-2 Mécanisme de transformation.....170

Figure VII-3 Mécanisme de conjugaison.....171

Figure VII-4 Mécanisme de transduction.....172

Figure VII-5 Phylogénie d'espèces (a) différente de celle de gène (b) (tiré de Boc et al., 2004 page 6).....173

Figure VII-6 Arbre d'espèces (a) transformé en arbre de gène (d) en trois étapes (tiré de Boc et al., 2004 page 7).....174

Figure VII-7 Algorithme HGT: Temps écoulé en secondes en fonction de la taille des données d'entrée.....176

Figure VII-8 Algorithme de reconstruction de reticulogrammes : Temps écoulé en secondes en fonction de la taille des données d'entrée.....179

Figure ANNEXE A-1 Page d'accueil du serveur Web de T-REX..... 183

Figure ANNEXE A-2 Inférence d'un arbre phylogénétique avec la méthode NJ.. 184

Figure ANNEXE A-3 Arbre inféré avec la méthode NJ.. 185

Figure ANNEXE A-4 Inférence d'un réticulogramme avec la méthode ADDTREE..... 186

Figure ANNEXE A-5 Réticulogramme inféré avec la méthode ADDTREE..... 187

Figure ANNEXE A-6 Détection de transferts horizontaux de gènes..... 188

Figure ANNEXE A-7 Premier transfert horizontal détecté..... 189

Figure ANNEXE A-8 Troisième transfert horizontal détecté.. 190

Figure ANNEXE A-9 Cinquième transfert horizontal détecté.. 191

Figure ANNEXE A-10 Inférence d'un arbre phylogénétique avec la méthode PHYML..... 192

Figure ANNEXE A-11 Arbre inféré avec la méthode PHYML.. 193

LISTE DES TABLEAUX

Tableau II-1 Type de données de MPI. 56

Tableau II-2 Routines de MPI. 59

Tableau IV-1 Algorithme NJ : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster. 109

Tableau IV-2 Algorithme NJ : Temps écoulé en secondes en fonction du nombre de lames du cluster pour 400 et 800 espèces. 110

Tableau IV-3 Algorithme ADDTREE : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 115

Tableau IV-4 Algorithme de reconstruction par ordre circuclaire : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 119

Tableau IV-5 MW : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 123

Tableau IV-6 Algorithme Fitch : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 129

Tableau V-1 Algorithme PHYML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 136

Tableau V-2 Algorithme DNAML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster 140

Tableau V-3 Algorithme DNAMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....143

Tableau V-4 Algorithme PROML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....147

Tableau V-5 Algorithme PROMLK: Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....150

Tableau VI-1 Algorithme DNAPARS : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....157

Tableau VI-2 Algorithme PARS : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....161

Tableau VI-3 Algorithme DOLLOP : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....165

Tableau VII-1 Algorithme HGT : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....175

Tableau VII-2 Algorithme de reconstruction de reticulogrammes : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.....179

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ADDTREE	Additive Similarity Trees
ADN	Acide DésoxyriboNucléique
ARN	Acide RiboNucléique
DNAML	DNA Maximum Likelihood Program
DNAMLK	DNA Maximum Likelihood Program with mocelular clock
DNAPARS	DNA Parcimony Program
DOLLOP	Dollo and Polymorphism Parsimony Program
HGT	Horizontal Gene Transfert
MPI	Message Passing Interface
MW	Method of Weighted least-squares
NJ	Neighbor-Joining
NNI	Nearest Neighbor Interchange
PARS	Parsimony Program
PHYLIP	PHYLogeny Inference Package
PROML	Protein Maximum Likelihood Program
PROMLK	Protein Maximum Likelihood Program with Molecular Clock
PROTPARS	Protein Parcimony Program
PVM	Parallel Virtual Machine
SPR	Subtree Pruning and Regrafting
TBR	Tree Bissection and Recombination
UPGMA	Unweighted Pair-Group Method using Arithmetic Averages
UNJ	Unweighted Neighbor-Joining

RÉSUMÉ

Dans ce mémoire nous abordons de prime abord la reconstruction d'arbres et de réseaux phylogénétiques, à travers deux méthodes d'inférence. Les arbres et les réseaux sont deux supports pour la représentation de l'évolution d'un groupe d'espèces étudiées.

Les modèles d'évolution d'espèces qui seront traités sont les suivants :

- Le modèle arborescent classique qui a longtemps été le seul support formel pour la représentation des relations génétiques entre les espèces.
- Le modèle en réseau qui permet de représenter des mécanismes phylogénétiques importants pouvant jouer un rôle clé dans l'évolution et pouvant s'expliquer par le phénomène de l'évolution réticulée. Nous nous sommes particulièrement intéressé aux algorithmes d'inférence de réseaux de transferts horizontaux de gènes. Un transfert horizontal de gènes permet à deux espèces de s'échanger, partiellement ou totalement, différents gènes au cours de l'évolution.

Le travail effectué sur la reconstruction d'arbres et de réseaux phylogénétiques a mené à la publication de trois articles.

Ensuite, nous abordons le problème de réduction du temps d'exécution de différents programmes bioinformatiques. Ce problème a pris de l'ampleur à cause de la croissance du volume de données biologiques et du blocage de la puissance des ordinateurs autour de 3,4GHZ depuis environ deux ans.

Nous décrivons un procédé d'accélération des calculs effectués par différents algorithmes d'inférence et de représentation de l'évolution des espèces, en utilisant le parallélisme. Le parallélisme mis en place a été réalisé à travers une librairie standard de passage de messages (Message Passing Interface).

Nous montrons les différentes formes de parallélisme, les architectures de systèmes parallèles, quelques environnements qui permettent de supporter l'exécution des applications de façon à exprimer le parallélisme, ainsi que les approches utilisées pour paralléliser différents modèles d'évolution. Les versions parallèles des algorithmes d'évolution ont été développées et installées sur une « grappe » (i.e. *cluster*) Linux ayant 16 lames possédant chacune deux processeurs et sa propre mémoire.

Mots clés : algorithmes d'évolution, arbre phylogénétique, réseau phylogénétique, transferts horizontaux de gènes, programmation parallèle, Message Passing Interface (MPI).

INTRODUCTION

La bioinformatique est un domaine multidisciplinaire qui associe la biologie, l'informatique et les mathématiques. Le développement récent et rapide de la génomique et de la protéomique suscite une collaboration de plus en plus étroite entre les spécialistes des sciences de la vie et de l'informatique, afin de développer de nouvelles approches et de nouvelles méthodes analytiques permettant d'examiner la quantité massive de données biologiques disponibles.

La croissance du volume de données biologiques et le plafonnement de la puissance des ordinateurs (autour de 3,4GHZ depuis environ deux ans) nécessitent le développement de méthodes informatiques efficaces pour réduire le temps d'exécution de différentes applications bioinformatiques.

La réduction du temps d'exécution d'un programme peut être réalisée de deux manières :

- La réécriture du programme en optimisant les opérations.
- La parallélisation du programme à l'aide des outils disponibles.

Lors de nos travaux, nous nous sommes intéressés à la deuxième approche, à savoir, paralléliser les programmes afin de diminuer le temps d'exécution.

Actuellement, il n'existe aucun outil de parallélisation autonome. Il existe par contre des environnements qui permettent de supporter l'exécution des applications de façon à réaliser le

parallélisme. Ces environnements ont été développés pour les applications de calcul numérique intensif. Il s'agit de PVM (Parallel Virtual Machine) qui existe depuis de nombreuses années et d'un récent standard, MPI (Message Passing Interface), que nous avons utilisé dans le cadre de ce travail.

Dans ce mémoire, nous présentons principalement les différentes formes de parallélisme utilisées pour réduire le temps d'exécution de divers algorithmes bioinformatiques.

Pour mieux comprendre les différentes méthodes d'inférence d'arbres phylogénétiques et de réseaux réticulés, que nous avons parallélisées, nous les présenterons sommairement au chapitre I.

Dans le chapitre II, nous présenterons les différentes formes de parallélisme, les architectures de systèmes parallèles, ainsi que les environnements qui permettent de supporter l'exécution des applications de façon à réaliser le parallélisme.

Au chapitre III, nous présenterons les trois articles écrits dans le cadre de notre projet de maîtrise. Le premier traite le problème d'inférence d'arbres phylogénétiques à partir des séquences de nucléotides (i.e. séquences d'ADN ou d'ARN) contenant des entrées manquantes. Un algorithme permettant d'estimer des données nucléotidiques partielles a été développé en vue de la reconstruction phylogénétique. Le deuxième article décrit une nouvelle méthode pour prédire et visualiser les possibles transferts horizontaux de gènes dans le cadre des modèles complet et partiel. Dans le troisième article nous détaillons notre recherche sur des modèles de transferts horizontaux. Une procédure de validation statistique des transferts inférés a été conçue et présentée à l'aide d'un exemple de données réelles. Cette procédure peut être appliquée pour estimer la robustesse d'un transfert horizontal particulier ou d'un scenario de transferts au complet.

Au chapitre IV, nous présenterons les algorithmes d'inférence phylogénétique basés sur l'approche de distances. Les statistiques concernant le gain obtenu grâce à l'utilisation des versions parallèles de ces algorithmes seront mises en évidence.

Au chapitre V, nous présenterons les méthodes d'inférence phylogénétique basées sur l'approche du maximum de vraisemblance et les statistiques caractérisant les versions parallèles de ces algorithmes.

Au chapitre VI, nous présenterons les méthodes d'inférence phylogénétique utilisant le principe du maximum de parcimonie. Les statistiques concernant le gain obtenu grâce à l'utilisation des versions parallèles de ces algorithmes seront exposées.

Au chapitre VII, nous présenterons l'algorithme de détection de transferts horizontaux de gènes et la méthode d'ajout de réticulations que nous avons parallélisés. Les statistiques caractérisant la version parallèle de cet algorithme seront présentées.

En conclusion, nous effectuerons une synthèse du travail accompli tout en mettant en évidence les améliorations possibles de la version parallèle des différentes méthodes présentées.

Ce mémoire est clôturé par deux annexes. La première annexe fournit des captures d'images de l'interface Web du logiciel T-Rex qui contient tous les programmes parallélisés. La seconde annexe contient le code source de tous les programmes parallélisés.

CHAPITRE I

RECONSTRUCTION DE PHYLOGÉNIES

1.1 Introduction

En 1859, Charles Darwin (Darwin, 1859) a introduit la notion d'arbre phylogénétique à travers sa théorie d'évolution par sélection. Ses travaux ont contribué grandement à l'utilisation d'arbres comme support formel de la représentation des relations entre les espèces. Ce support est devenu un élément incontournable dans leur classification. Sept ans plus tard, en 1866, Ernst Haeckel présenta le premier arbre universel du vivant (figure I-1), représentant toutes les espèces et tous les groupes connus à l'époque. Étonnamment, cet arbre fut remarquablement correct parce qu'il ressemble en plusieurs points à ce que l'on connaît actuellement de l'évolution des organismes, sauf en ce qui concerne les microorganismes et, en particulier, les procaryotes. Au cours des décennies suivantes, les seuls moyens de classification des espèces étaient les comparaisons entre leur morphologie, leur comportement et leur répartition géographique.

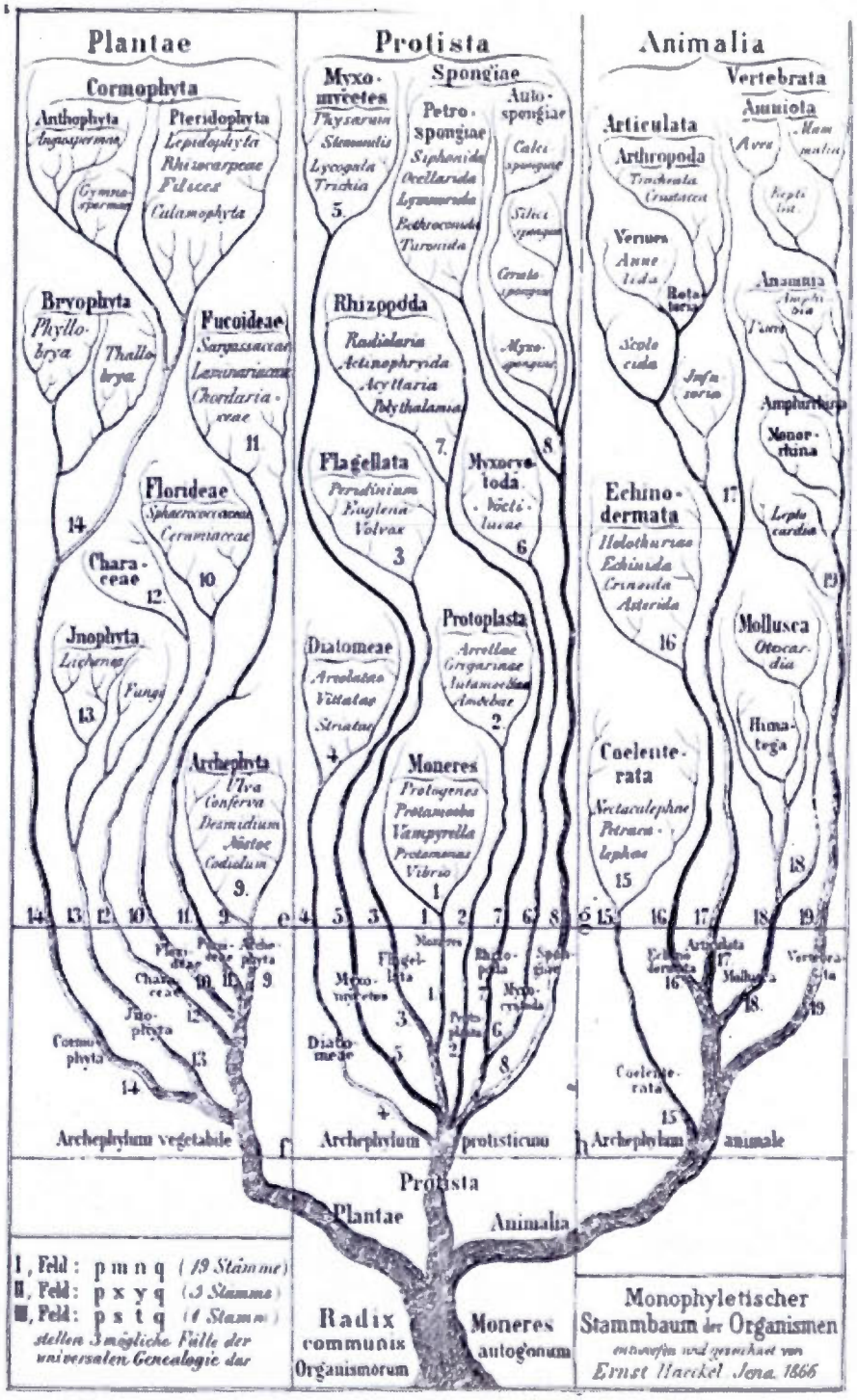


Figure I-1 Arbre d'évolution de Haeckel (1866).

Il a fallu attendre l'arrivée des données moléculaires, vers la fin des années 1970 et au début des années 1980, pour assister à la découverte d'importants mécanises d'évolution, tels que la découverte de l'échange de matériel génétique entre des individus appartenant à différentes espèces (Doolittle, 1999). Actuellement, des arbres phylogénétiques peuvent être inférés à partir d'un ensemble varié de données telles que les caractères discrets, les séquences moléculaires, les fréquences des gènes, les sites de restriction, l'ordre des gènes, les microsatellites, etc.

Cependant, dans de tels arbres classiques, chaque espèce ne peut être reliée qu'avec son ancêtre le plus proche. Plusieurs importants mécanismes phylogénétiques, pouvant jouer un rôle clé dans l'évolution et pouvant s'expliquer par le phénomène de l'évolution réticulée (e.g., transfert horizontal de gènes, homoplasie, hybridation), ne sont pas représentables par un modèle arborescent traditionnel. Ceci a conduit à la mise au point de différents modèles réticulés ou modèles en réseau. Au cours des dernières années, les modèles de réseaux sont devenus de plus en plus utilisés.

Dans ce mémoire, nous présentons uniquement les phylogénies inférées à partir de données moléculaires (phylogénies moléculaires). Dans ce chapitre, nous présentons sommairement la notion de phylogénie, quelques définitions de base pour se familiariser avec l'analyse phylogénétique ainsi que les principales méthodes d'inférence d'arbres phylogénétiques et de réseaux réticulés.

1.2 Définitions de base

1.2.1 ADN

L'acide désoxyribonucléique (souvent abrégé en ADN) est une macromolécule de poids moléculaire élevé, formée de polymères de nucléotides dont le sucre et deux désoxyriboses. L'ADN se présente sous la forme d'une double chaîne hélicoïdale dont les deux brins sont complémentaires. Il est composé d'un assemblage linéaire de nucléotides renfermant chacun dans sa structure une base azotée qui l'identifie. Il existe quatre types de nucléotides au

niveau de l'ADN : l'adénine (A), la guanine (G), la cytosine (C) et la thymine (T). Ces quatre nucléotides se subdivisent en deux groupes : les purines composées de l'adénine et de la guanine; les pyrimidines composées de la cytosine et de la thymine. L'ADN est présent dans le noyau des cellules eucaryotes, dans le cytoplasme des cellules procaryotes, dans la matrice des mitochondries ainsi que dans les chloroplastes. Certains virus possèdent également de l'ADN encapsulé dans leur capside, coque constituée de protéines qui entoure et protège l'acide nucléique d'un virus. On dit que l'ADN est le support de l'hérédité. Il se transmet en en partie ou en totalité lors des processus de reproduction. Il est à la base de la synthèse des protéines.

1.2.2 Alignement

L'alignement est une opération qui permet de comparer des séquences de nucléotides pour repérer les éléments correspondants. Cette opération est nécessaire pour toutes les comparaisons de séquences.

1.2.3 ARN

L'acide ribonucléique (souvent abrégé ARN) est une macromolécule formée par la polymérisation de nombreux nucléotides dont le sucre et le ribose. Tout comme l'ADN, l'ARN est composé d'un assemblage linéaire de nucléotides renfermant chacun dans sa structure une base azotée qui l'identifie. Il existe quatre types de nucléotides au niveau de l'ARN : l'adénine (A), la guanine (G), la cytosine (C) et l'uracile (U). Ces quatre nucléotides se subdivisent en deux groupes : les purines composées de l'adénine et de la guanine; les pyrimidines composées de la cytosine et de l'uracile. L'ARN est présent dans le cytoplasme, les mitochondries ainsi que dans le noyau cellulaire. Il sert d'intermédiaire dans la synthèse des protéines.

1.2.4 Bootstrap

Le « boostrap » est une méthode statistique utilisée pour évaluer la robustesse d'un arbre phylogénétique. Elle part de l'hypothèse que les sites évoluent de manière indépendante.

1.2.5 Branch and Bound

Le « Branch and bound » est une méthode informatique utilisée pour résoudre des problèmes complexes. Pour trouver la solution à un problème, le « Branch and bound » effectue des parcours en profondeur de l'espace de solutions. Il ignore tous les sous ensembles de l'espace de solution qui ne remplissent pas un critère fixé au départ.

1.2.6 Gène

Un gène est une séquence ordonnée de nucléotides qui occupe une position précise sur un chromosome déterminé. Il constitue une information génétique dont la transmission est héréditaire, c'est-à-dire transmise par un individu à sa descendance par reproduction sexuée ou asexuée. Le gène le plus simple consiste en un segment d'acide désoxyribonucléique (ADN) codant un seul acide ribonucléique (ARN), à l'origine d'une seule protéine (en dehors de l'épissage alternatif). L'ensemble des gènes d'un individu constitue une partie de son génome.

1.2.7 Lignée

Une lignée est le regroupement de toutes les espèces issues du même ancêtre.

1.2.8 Matrice de dissimilarités ou matrice de distances

Une matrice de dissimilarités est un tableau à double entrée comprenant horizontalement comme verticalement la même série d'espèces. Chacune des cases de la matrice contient la distance de dissimilarité qui sépare les deux espèces concernées. La valeur de dissimilarité est égale à zéro pour deux espèces identiques.

1.2.9 Maximum de parcimonie

L'approche de maximum de parcimonie, souvent dénommée parcimonie, est une approche statistique non-paramétrique très utilisée, notamment pour l'inférence phylogénétique. Sous

l'hypothèse du maximum de parcimonie, l'arbre phylogénétique inféré est celui qui requiert le plus petit nombre de changements évolutifs.

1.2.10 Maximum de vraisemblance

L'estimation du maximum de vraisemblance est une approche statistique courante utilisée pour inférer les paramètres de la distribution de probabilité d'un échantillon donné. Elle est très employée dans le domaine de la bioinformatique.

1.2.11 Phylogénie

La phylogénie ou phylogenèse se définit, en général, dans l'encyclopédie comme « *l'histoire de la formation et de l'évolution d'une espèce* ». Le terme phylogenèse provient du grec phûlon « tribu » et genesis « origine » et il a été présenté par Haeckel dès 1860 (Haeckel, 1860), qui l'a définie comme « *l'histoire du développement paléontologique des organismes par analogie avec l'ontogénie ou histoire du développement individuel* ». La phylogénie constitue un procédé pour construire des classifications d'espèces. La phylogénie moléculaire, dont certaines méthodes de reconstruction seront présentées par la suite, étudie l'histoire évolutive des espèces en se basant sur une portion de leur séquence moléculaire.

1.2.12 Transition

La transition est une mutation au cours de laquelle une base purique est remplacée par une autre base purique ou par une base pyrimidique.

1.2.13 Transversion

La transversion est une mutation au cours de laquelle une base purique est remplacée par une base pyrimidique ou vice-versa.

1.3 Arbres phylogénétiques

1.3.1 Définition

Un arbre est dit phylogénétique (figure I-2) si le concept de descendance des espèces avec modification des sites des séquences des espèces a été utilisé lors de l'inférence de l'arbre. L'arbre phylogénétique présente les relations de parenté entre les organismes vivants. Il montre le rapprochement entre les différentes espèces et pas forcément leur descendance. L'arbre est une forme de classification des espèces.

Un arbre est composé de quatre éléments principaux :

- la racine qui indique l'ancêtre commun des espèces représentées dans l'arbre. On considère les arbres phylogénétiques enracinés ou non-enracinés.
- les nœuds externes ou feuilles qui représentent les espèces contemporaines pour lesquelles les informations ont été utilisées lors de la construction de l'arbre;
- les nœuds internes qui représentent des ancêtres hypothétiques;
- les branches qui montrent les relations de descendance entre les nœuds de l'arbre.

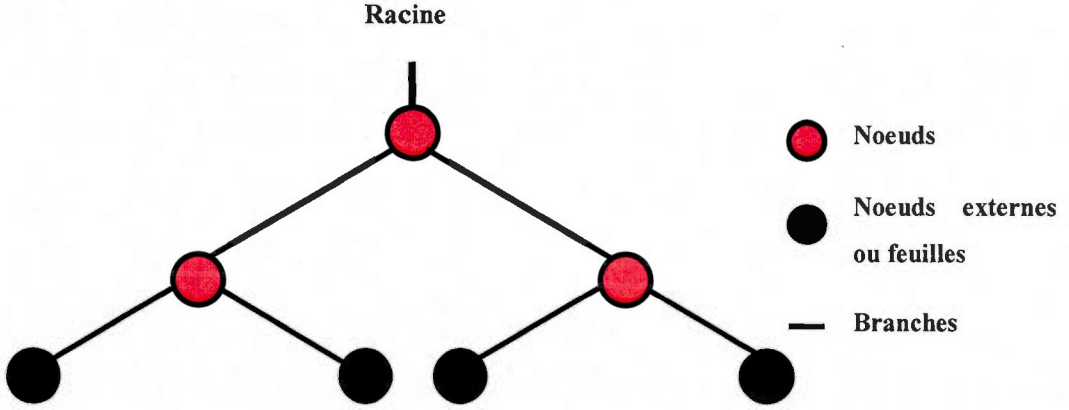


Figure I-2 Un arbre phylogénétique.

1.3.2 Caractéristiques

Le degré d'un nœud de l'arbre est défini comme le nombre de branches adjacentes à ce nœud. Tout nœud de degré supérieur à trois est appelé « non résolu » sinon le nœud est « résolu »., Un arbre phylogénétique binaire *non-enraciné* ayant n espèces (feuilles) dont tous les nœuds internes sont résolus est composé de :

- $2n$-2 nœuds (n-2 nœuds internes et n feuilles);
- $2n$-3 branches.

Lorsque l'ancêtre commun de toutes les espèces de l'arbre est identifié, l'arbre est enraciné. Il est orienté dans le sens du temps d'évolution des espèces. L'arbre enraciné permet de définir une relation de descendance entre deux nœuds successifs. Généralement, il est impossible d'identifier l'origine de diversification des espèces. Au niveau d'un arbre non enraciné, on perd la notion de temps d'évolution et la relation d'ancêtres. L'arbre non enraciné est habituellement utilisé pour la classification d'un groupe d'espèces sans considérer le sens d'évolution. La figure I-3 représente deux arbres enracinés et un arbre non enraciné ayant chacun quatre espèces α, β, γ, δ. La figure I-3(c) représente l'arbre non enraciné tandis que les figures I-3(a) et I-3(b) représentent des arbres enracinés et montrent la différence dans l'évolution entraînée par un changement de la position de la racine. Sur la figure I-3(a) le regroupement ψ est composé des espèces α et β tandis que sur la figure I-3(b), le regroupement ψ est composé des espèces β, γ et δ. Remarquons aussi dans ces deux derniers cas que le sous arbre enraciné par ω regroupe exactement les mêmes espèces, à savoir γ et δ.

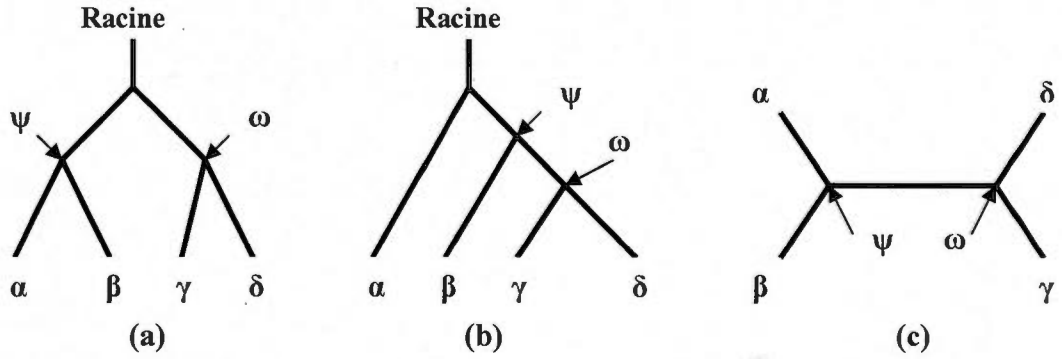
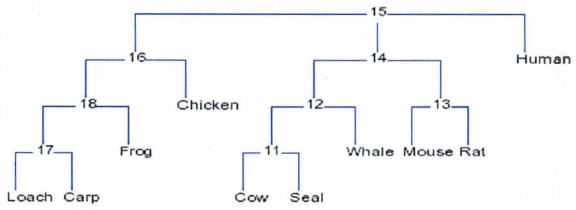


Figure I-3 Deux arbres enracinés (a et b) et un arbre non enraciné (c).

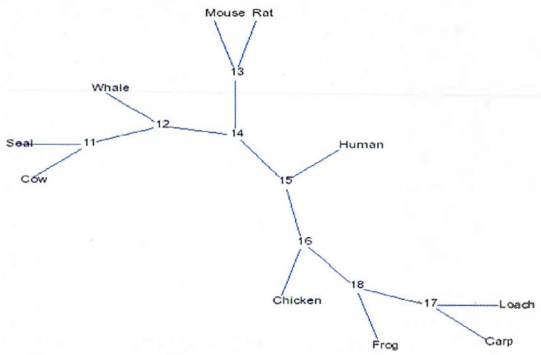
1.3.3 Les représentations des arbres

Il existe un bon nombre de représentations pour les arbres phylogénétiques dans la littérature scientifique. Barthélémy et Guénoche (1991) ont présenté les types de tracé les plus populaires et connus à savoir : le tracé hiérarchique, le tracé axial et le tracé radial.

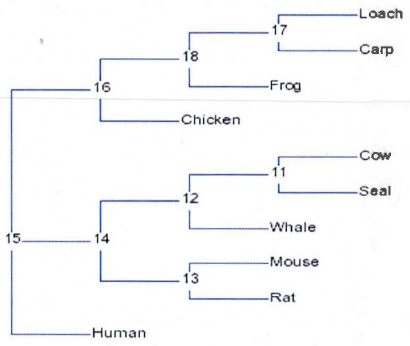
Les espèces contemporaines sont les feuilles de l'arbre et les ancêtres sont représentés par les nœuds internes. Le tracé hiérarchique représente la hiérarchie des espèces étudiées. Dans le tracé radial, les espèces sont disposées à l'intérieur d'un cercle imaginaire. Dans le tracé axial les espèces sont disposées le long du chemin le plus long de l'arbre.



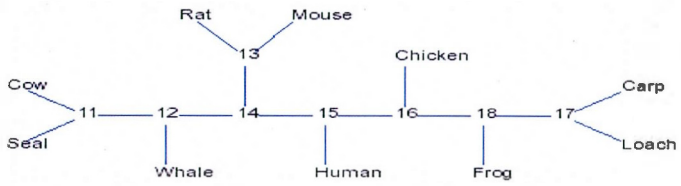
Tracé hiérarchique vertical



Tracé radial



Tracé hiérarchique horizontal



Tracé axial

Figure I-4 Quatre représentations d'un arbre phylogénétique (obtenues à l'aide du logiciel T-Rex (Makarenkov, 2001)).

1.4 Réseaux réticulés

Il a longtemps été supposé que l'évolution des espèces est un processus de branchement ne pouvant être représenté que par une topologie d'arbre. Dans cette dernière, une espèce est liée

à son ancêtre le plus proche et toutes les autres relations entre les espèces ne peuvent être prises en compte. L'évolution réticulée reflète la partie de l'évolution des espèces qui ne peut être représentée par le modèle arborescent classique.

Pour construire un réticulogramme, ou réseau réticulé, la première étape consiste à reconstruire l'arbre original représentant l'évolution des espèces puis la seconde à ajouter les réticulations (ou branches supplémentaires) nécessaires (figure I-5). La réticulation est représentée par le trait ajouté entre les branches 1 et 2 de l'arbre original. La méthode d'ajout de réticulations sera présentée au chapitre VII.

Les modèles d'évolution réticulée ont été retrouvés dans des contextes d'évolution très variés, ce qui a donné lieu à une augmentation des cas étudiés. Dans l'évolution des bactéries par exemple, le transfert horizontal de gènes est un mécanisme qui permet aux bactéries d'échanger des gènes (Sonea et Panisset, 1976 et 1981; Doolittle, 1999; Sonea et Mathieu, 2000; Sneath, 2000). Les réticulations apparaissent comme résultat de l'allopolyploïdie chez les plantes, ce qui peut mener à l'apparition instantanée d'une nouvelle espèce possédant le complément chromosomique des deux espèces parentes. L'évolution réticulée est aussi présente dans la microévolution à l'intérieur des espèces chez les eucaryotes à reproduction sexuée (Smouse, 2000), après différentiation génétique des populations allopatriques suivie d'échanges de gènes lors des migrations. Elle se retrouve aussi au niveau de l'homoplasie qui est la portion de la similarité phylogénétique provenant de la convergence évolutive.

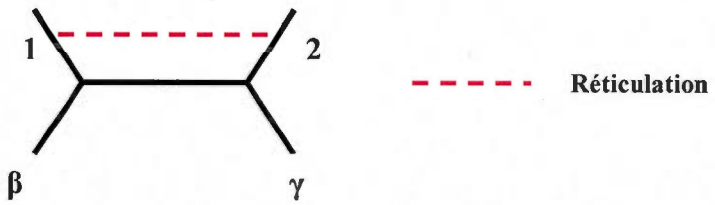


Figure I-5 Un réseau réticulé.

1.5 Méthodes de reconstruction d'arbres phylogénétiques

La manière classique d'illustrer les relations phylogénétiques entres les espèces est de les modéliser en utilisant un arbre phylogénétique. La manière moderne consiste en la modélisation d'un réseau phylogénétique comprenant les réticulations nécessaires. A ce niveau nous présenterons seulement quelques méthodes d'inférences d'arbres phylogénétiques. Les méthodes d'inférence phylogénétique présentées sont exposées en détails dans Swofford et al. (1996), Li (1997) et Felsenstein (2004).

La reconstruction d'un arbre phylogénétique débute par ce que l'on appelle « l'alignement » qui consiste à mettre en correspondance les sites des séquences des espèces de manière à pouvoir les comparer les unes aux autres. Les séquences utilisées pour la reconstruction peuvent être de l'ADN ou de l'ARN.

Il existe deux approches principales pour l'inférence de phylogénies. La première, appelée approche phénétique, ne tient pas compte des relations historiques entre les espèces. Elle fonctionne en calculant les distances entre les espèces (matrice de dissimilarités) à partir des séquences alignées, puis en reconstruisant l'arbre phylogénétique en utilisant une procédure de regroupement hiérarchique. La deuxième approche, appelée approche cladistique, considère plusieurs pistes de l'évolution, en inférant un dispositif d'ancêtres au niveau de chaque nœud et en choisissant un arbre optimal selon le modèle d'évolution. L'approche phénétique est basée sur la similarité tandis que l'approche cladistique est basée sur la généalogie. Nous présenterons quatre méthodes principales pour la construction de phylogénies : la méthode basée sur les distances (approche phénétique), la méthode basée sur le maximum de parcimonie (approche cladistique), la méthode basée sur le maximum de vraisemblance (approche cladistique) et la méthode bayésienne (approche cladistique).

1.5.1 Approche phénétique

1.5.1.1 Méthode basée sur les distances

Cette méthode d'inférence d'arbres phylogénétiques est basée sur les matrices de dissimilarités qui ont été introduites par Cavalli-Sforza et Edwards (1967) et Fitch et Margoliash (1967). Cette méthode calcule les distances de dissimilarité entre chaque paire d'espèces avant d'inférer l'arbre phylogénétique dont les longueurs des branches se rapprochent le mieux aux distances calculées. La somme des longueurs des branches entre deux espèces *i* et *j*, de l'arbre phylogénétique est appelée distance évolutive entre les espèces *i* et *j*. La distance évolutive correspond au nombre d'évènements mutationnels réels contrairement à la distance de dissimilarité qui est juste une estimation obtenue par comparaison des séquences alignées. La distance évolutive est également appelée distance additive et respecte les propriétés d'une distance arborée voir Barthélémy et Guénoche (1991).

Si les distances estimées sont assez proches du nombre d'évènements évolutifs entre les espèces, la méthode reconstruit un arbre correct (Kim et Warnow, 1999). Cette supposition est vraie pour plusieurs modèles d'évolution de séquences biomoléculaires, dans lesquels les méthodes basées sur les distances donnent des résultats suffisamment précis (Li, 1997). L'avantage principal des méthodes basées sur les distances est que leur temps d'exécution est assez rapide, ce qui leur permet d'analyser de larges ensembles de données.

Le taux d'évolution étant constant au niveau de tout l'arbre, la prise en compte de l'hypothèse de l' « horloge moléculaire » entraîne des corrections mineures à la matrice de dissimilarités au cours de l'inférence de l'arbre phylogénétique. Cependant, cette hypothèse est habituellement inappropriée pour les espèces séparées par de grandes distances. Par conséquent, la reconstruction d'un arbre phylogénétique correct devient très problématique sous l'hypothèse de l' « horloge moléculaire ». Si cette dernière n'est pas prise en compte, les différences observées (entre les espèces) ne reflètent pas de manière exacte les distances évolutives. Par conséquent plusieurs substitutions au niveau d'un même site biaisent les vraies distances et font en sorte que les séquences semblent plus proches les unes des autres

qu'elles ne le sont réellement. Dans ce cas, une correction des distances évolutives entre les paires d'espèces est nécessaire. Il existe plusieurs modèles de Markov pour modéliser l'évolution des séquences et chacune de ces modélisations implique une manière spécifique d'estimer et de corriger les distances évolutives. Ces corrections occasionnent des variations substantielles si les distances sont trop élevées. Parmi les plus populaires modèles de transformation des séquences en distances, nous avons la distance de Hamming (1950), de Jukes et Cantor (Jukes et Cantor, 1969), Kimura 2-paramètres (Kimura, 1981). L'inférence d'arbre phylogénétique à partir de séquences très différentes produit une problématique quant à la fiabilité des valeurs de la matrice de distances.

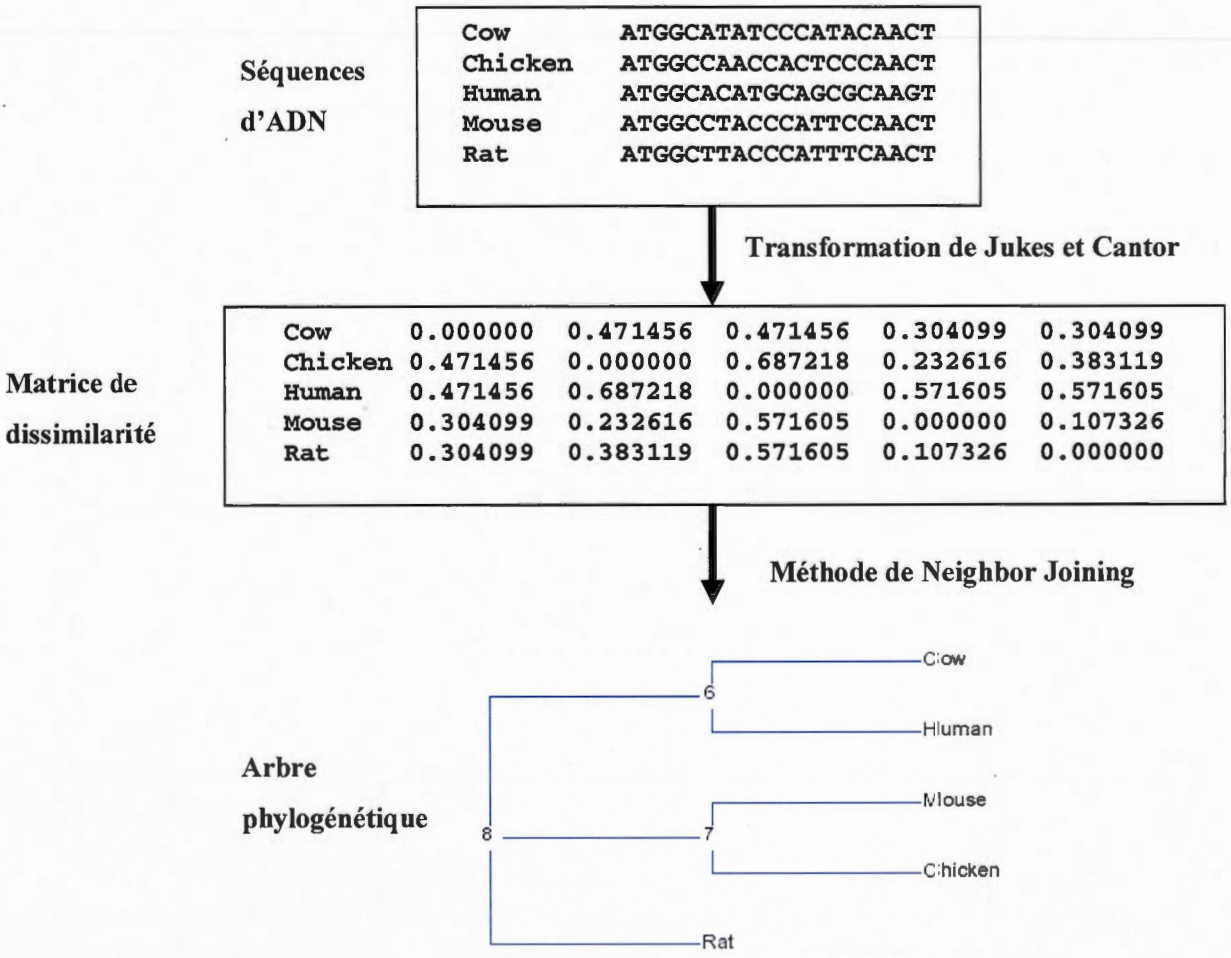


Figure I-6 Processus d'inférence d'arbres phylogénétiques par la méthode de distances

Les distances de la matrice de dissimiliratés sont calculées en appliquant un modèle d'évolution aux séquences alignées (ou à d'autres types d'informations sur les séquences). Toutefois la distance obtenue par une simple application de certains modèles d'évolution est considérée comme « non corrigée » et elle comprend deux biais majeurs : (1) la probabilité d'avoir plus d'une mutation à un site donné augmente avec le temps de divergence entre deux séquences. Ainsi des mutations multiples peuvent survenir sans être observables ; (2) la probabilité d'un changement dans un site peut être différente selon les types de données. Ces biais peuvent être corrigés pour mieux estimer la distance évolutive des séquences. La correction des ces biais peut être effectuée par des méthodes de PHYLIP (Felsenstein, 1993) : DNADIST pour les séquences d'ADN et PROTDIST pour les séquences de protéines. Les corrections obtenues selon les modèles d'évolution sont le plus souvent utilisées pour inférer les phylogénies, à la place des distances observées « non corrigées ».

a. Reconstitution de la distance évolutive à partir de la distance observée

Les méthodes de distances utilisent la matrice de dissimilarités pour reconstituer la matrice de distances évolutives (distance d'arbres) qui peut être représentée sous forme d'un arbre unique. Pour reconstruire l'arbre phylogénétique de la matrice de distances évolutives, les méthodes de distances se servent de différentes techniques dont les plus citées sont les suivantes :

- **Les méthodes d'ajustement**

Les méthodes d'ajustement reposent sur le choix de la distance arborée la plus proche de la matrice de dissimilarité, en utilisant un critère dont la forme générale est la suivante :

$$Q = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(a_{ij} - d_{ij})^2 \qquad \textbf{(1.1)}$$

où W représente souvent la matrice de poids accordés à la comparaison des paires de séquences;

n représente le nombre d'espèces;

a_{ij} représente la distance évolutive entre les espèces i et j;
d_{ij} j représente la dissimilarité entre les espèces i et j.

Le critère des moindres carrés non pondérés (Cavalli-Sforza et Edwards, 1967) correspond à l'initialisation de toutes les valeurs de la matrice de poids W à 1. Fitch et Margoliash (1967) utilisent le même critère où $w_{ij} = \frac{1}{d_{ij}^2}$, tandis que Beyer et al. (1974) ont proposé de considérer le cas où $w_{ij} = \frac{1}{d_{ij}}$.

- **Les méthodes d'évolution minimum**

Les méthodes d'évolution minimum se basent sur la longueur totale des branches de l'arbre reconstruit. Une première méthode de ce type a été proposée par Kidd et Sgaramella-Zonta (1971) et puis raffinée par Saitou et Nei (1987) et Rzhetsky et Nei (1992). Ces méthodes utilisent la somme des longueurs de branches de l'arbre et exploitent deux critères. L'arbre est ajusté aux données et la longueur des branches est déterminée en utilisant la méthode des moindres carrés non pondérés. Les méthodes d'évolution minimum requièrent un temps de calcul similaire à celui des méthodes d'ajustement. Ce temps peut être amélioré par les méthodes de Bryant et Waddell (1998) et Makarenkov et Leclerc (1999) qui réduisent le temps de calcul des moindres carrés. Le temps de recherche des topologies d'arbres peut également être diminué en utilisant une recherche gloutonne comme proposée par Desper et Gascuel (2002).

a) Les méthodes de regroupement (clustering)

Ces méthodes n'utilisent pas un critère explicite pour trouver l'arbre qui correspond à la matrice de dissimilarités. Elles utilisent un algorithme particulier sur la matrice de dissimilarités pour inférer directement un arbre. Elles peuvent être très rapides, mais elles ne garantissent pas l'utilisation de toutes les distances (Felsenstein, 2004). Parmi les algorithmes de reconstruction d'arbres phylogénétiques basés sur les distances, la méthode UPGMA (Unweighted Pair-Group Method using Arithmetic Averages) fut l'une des premières à avoir

été proposée (Sokal et Sneath, 1973) et la méthode Neighbor-Joining (NJ) de Saitou et Nei (1987) est la plus populaire.

1.5.2 Approche cladistique

L'approche cladistique inclut des méthodes qui infèrent les arbres directement à partir des données des séquences sans passer par le calcul de matrices de dissimilaraités. Elles sont généralement plus robustes statistiquement que les méthodes de distances. Mais elles sont en contrepartie très lentes. Cette approche regroupe les méthodes de parcimonie, de maximum de vraisemblance et nouvellement les méthodes bayésiennes.

1.5.2.1 Le maximum de parcimonie

L'idée générale de maximum de parcimonie a été donnée quand la première méthode de ce type a été introduite dans la littérature scientifique par Edwards et Cavalli-Slforza (1963). Les méthodes de maximum de parcimonie infèrent des arbres phylogénétiques en évaluant les mutations possibles entres les séquences. En terme général, le but de ces méthodes est de trouver l'arbre phylogénétique de longueur totale minimale, c'est-à-dire l'arbre qui a le plus petit nombre d'évènements mutationnels. Ceci implique deux problématiques : (1) La capacité d'effectuer la reconstruction impliquant le moins d'évènements possibles; (2) La capacité de chercher parmi toutes les phylogénies celle ou celles qui minimisent le nombre d'évènements.

Il existe plusieurs variations de la parcimonie et les plus utilisées sont les parcimonies de Wagner (Farris, 1970) et de Fitch (Fitch, 1971). Habituellement, on peut trouver plus d'un arbre qui minimise le nombre d'évènements mutationnels en appliquant la méthode de maximum de parcimonie. Afin de garantir de retrouver le meilleur arbre, une évaluation exhaustive de toutes les topologies possibles doit être effectuée. L'approche de maximum de parcimonie reconstruit correctement un arbre phylogénétique si le rapport entre le nombre d'évènements mutationnels par la position de la séquence est petit. Dans le cas d'un nombre d'évènements mutationnels élevé, la proportion de changement homoplastique augmente et peut entrainer des erreurs au cours de la reconstruction de l'arbre, spécifiquement au cours de

l'analyse des longues lignées non ramifiées ou si l'arbre contient un mélange de longues et de courtes branches. La méthode reconstruit efficacement les arbres phylogénétiques dans lesquels de multiples changements au niveau du même site se produisent rarement (Hillis, 1996; Kim, 1996). Les méthodes de parcimonie sont d'habitude plus lentes que celles basées sur les distances. Ces méthodes s'appuient sur deux hypothèses principales (Darlu et Tassy, 1993):

- Tous les sites évoluent indépendamment les uns des autres ;
- La vitesse d'évolution est lente et constante à travers les lignées évolutives.

Plusieurs approches peuvent être utilisées pour rechercher l'arbre le plus parcimonieux, c'est-à-dire l'arbre de longueur totale minimale. Lorsque le nombre de taxons est inférieur à dix, il est possible d'effectuer une recherche exhaustive. Dans le cas contraire, pour avoir un résultat dans un temps raisonnable, il faut se contenter des constructions heuristiques d'un arbre parcimonieux ou utiliser une approche de type « branch-and-bound ».

a) Recherche exhaustive

La recherche exhaustive examine toutes les topologies possibles par une approche basée sur des algorithmes voraces. La première étape consiste à former un arbre en regroupant trois espèces choisies aléatoirement ou selon des critères heuristiques (figure I-7). À la deuxième étape, une quatrième espèce est ajoutée sur chacune des trois branches de cet arbre, ce qui nous donne trois arbres possibles. Lors de la troisième étape, une cinquième espèce est ajoutée sur chacune des cinq branches possibles des trois arbres produits au cours de l'étape précédente, ce qui nous donne quinze arbres possibles. On procède de la sorte, par la suite, pour placer toutes les espèces étudiées sur les différentes branches. Ce qui nous conduit finalement à l'ajout de la $j^{ème}$ espèce à chacune des $(2j - 5)$ branches possibles (nombre de branches d'un arbre possédant j-1 feuilles) de tous les arbres produits à l'étape j-1. Le nombre total de topologies (non enracinées) de j feuilles explorées par cette approche est $\frac{(2j-3)!}{2^{(j-2)}(j-2)}$.

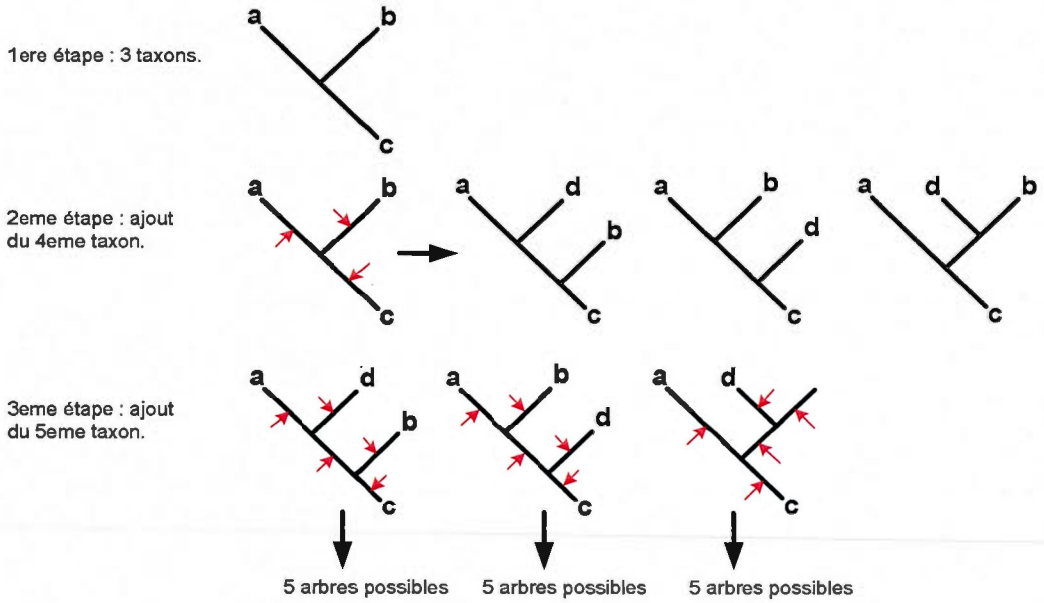


Figure I-7 Recouvrement de toutes les topologies par recherche exhaustive.

Cette approche calcule le nombre d'évènements mutationnels pour tous les arbres explorés. Le nombre d'évènements mutationnels détermine la longueur totale de l'arbre. Parmi tous les arbres reconstruits, on retient celui ou ceux de longueur minimale.

Par exemple, considérons un ensemble E = {a, b, c, d} et la séquence nucléotidique suivante pour chacun de ces taxons : a- ATTA, b- TAAC, c- AAAT, d- TTTT. La figure I-8 montre trois arbres formés à travers la recherche exhaustive. L'arbre qui sera retenu selon le critère de longueur minimale est le premier en partant de la gauche, car sa longueur est la plus petite (6 mutations).

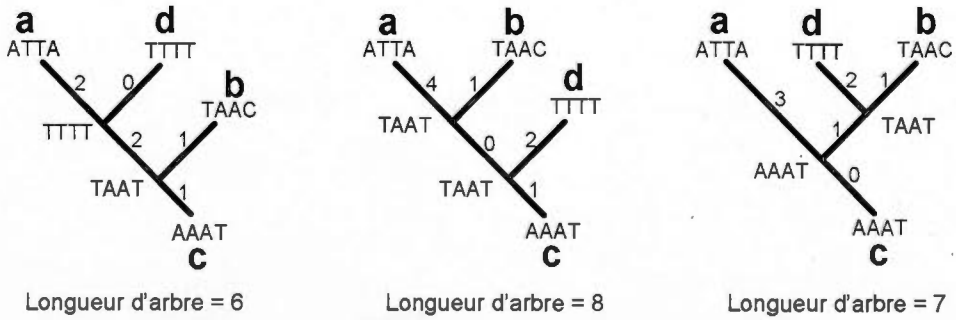


Figure I-8 Longueur d'arbre pour trois topologies (Harry, 2001).

b) Construction heuristique d'un arbre parcimonieux

Ne pouvant pas trouver le meilleur arbre en examinant tous les arbres possibles, on peut faire une recherche dans un espace plus restreint d'arbres. La technique de recherche heuristique consiste à trouver le(s) meilleur(s) arbre(s) sans examiner tous les arbres possibles. Cette méthode ne garantit pas de trouver le(s) meilleur(s) arbre(s). Dans les constructions heuristiques d'un arbre parcimonieux, des algorithmes itératifs s'ajoutent à la construction vorace de la recherche exhaustive. Ces algorithmes sont utilisés en général lorsque le nombre de taxons est supérieur à dix. La technique fondamentale des constructions heuristiques est de partir d'un arbre initial et d'effectuer des réarrangements sur les branches afin de trouver d'autres arbres « voisins ». Si un des « voisins » est plus parcimonieux que l'arbre initial, ce voisin est retenu et d'autres réarrangements sont examinés. Finalement, nous atteindrons un arbre, appelé optimum local dans l'espace des arbres, que des réarrangements n'amélioreront pas. Cependant, nous n'avons aucune garantie qu'il corresponde à l'optimum global. L'arbre initial est souvent inféré par des méthodes qui présentent une solution acceptable en un temps très court. Toutefois ces approches heuristiques n'explorent généralement que les topologies similaires à la topologie initiale. Par ailleurs, il existe plusieurs stratégies de réarrangements et les plus connues sont les suivantes (Swofford et al., 1996) :

- le réarrangement local (Nearest-Neighbor Interchanges "NNI");
- le réarrangement global (Subtree Pruning and Regrafting "SPR");
- la bissection et reconnexion d'arbres (Tree Bisection and Reconnection "TBR");
- les fusions d'arbres (tree-fusing);
- les algorithmes génétiques (Genetic algorithms).

1.5.2.2 Le maximum de vraisemblance

L'approche de maximum de vraisemblance a été introduite par Edwards et Cavalli-Sforza (1964) dans le cadre de l'étude des données sur la fréquence des gènes. La première application d'une méthode de maximum de vraisemblance aux séquences moléculaires a été effectuée par le statisticien Jerzy Neyman (1971), imposant des contraintes sur le taux d'évolution. L'approche de maximum de vraisemblance introduite par Felsenstein (1981) n'imposait, quant à elle, aucune contrainte par rapport à la constance du taux d'évolution

entre les lignées. Elle assigne des probabilités aux évènements mutationnels, plutôt que de les compter. Cette méthode compare les arbres phylogénétiques possibles sur la base de leur habileté à prédire les données observées. L'arbre qui a la plus grande probabilité de produire les séquences observées est choisi. La vraisemblance (ou le score de vraisemblance) est exprimée sous forme d'un logarithme naturel. Pour trouver l'arbre le plus vraisemblable, les bases de toutes les séquences à chaque site sont considérées séparément et le logarithme de la vraisemblance est calculé pour une topologie donnée, en utilisant un modèle d'évolution particulier. Ce logarithme de la vraisemblance est cumulé sur tous les sites et sa somme est maximisée pour estimer la longueur des branches de l'arbre. Cette procédure est répétée pour toutes les topologies possibles et la topologie ayant la plus grande vraisemblance est choisie. Il faut noter que le logarithme naturel de la vraisemblance est négatif car la probabilité calculée est inférieure à 1.

De façon similaire au maximum de parcimonie, la méthode de maximum de vraisemblance reconstruit un dispositif d'ancêtres au niveau de tous les nœuds de l'arbre considéré, mais elle assigne aussi aux branches une longueur basée sur les probabilités de mutation. Pour chaque topologie d'arbre possible, les taux de substitution supposés sont variés dans le but de trouver les paramètres qui produiront la plus grande vraisemblance par rapport aux séquences observées.

Sous plusieurs points de vue, le maximum de vraisemblance semble être une manière attrayante pour construire des phylogénies (Whelan et al., 2001). Toutes les mutations possibles et compatibles avec les données sont considérées. Les fonctions de vraisemblance sont connues pour avoir des bases solides et conformes pour l'inférence statistique (Edwards, 1972). Ces méthodes représentent bien les relations d'évolution entres les espèces. Elles prennent en considération plusieurs paramètres du processus d'évolution tels que les probabilités relatives des transitions par rapport aux transversions ou le degré avec lequel le taux d'évolution change entre les sites. Tous ces paramètres sont estimés au cours du processus d'évaluation.

Le principal obstacle dans l'utilisation des méthodes de maximum de vraisemblance est le temps de calcul. Les algorithmes qui trouvent le score de vraisemblance font des recherches dans un espace multidimensionnel de paramètres, ceci fait en sorte que la solution des

problèmes à grande échelle (> 100 séquences) est extrêmement longue à trouver. Une méthode d'estimation de maximum de vraisemblance peut être l'objet d'erreurs systématiques quand le modèle d'évolution utilisé pour évaluer la vraisemblance d'un arbre donné ne reflète pas le processus actuel d'évolution.

L'approche de maximum de vraisemblance est plus fiable que les méthodes de distances et de maximum de parcimonie. Théoriquement, c'est l'approche qui conduit au résultat le plus proche de l'arbre évolutif réel. Comparée au maximum de parcimonie, elle est aussi beaucoup plus robuste. Cette approche permet également d'appliquer les différents modèles d'évolution et d'estimer la longueur des branches en fonction des changements évolutifs.

Felsenstein (1981) a développé un des premiers programmes de maximum de vraisemblance, DNAML (DNA maximum likelihood program). Ce programme est présentement d'une grande utilité dans les analyses phylogénétiques. Des simulations effectuées ont montré que ce programme est extrêmement efficace dans l'estimation de vraies phylogénies dans des situations variées impliquant une violation de la constance du taux d'évolution dans une lignée. Il existe plusieurs autres programmes de maximum de vraisemblance, par exemple : ProtML (Adachi et Hasegawa, 1992), PUZZLE (Strimmer et Von Haeseler, 1996) et PHYML (Guindon et Gascuel, 2003). La méthode PHYML permet de traiter un très grand nombre d'espèces en un temps raisonnable.

1.5.2.3 Méthodes bayésiennes

Les méthodes bayésiennes sont étroitement liées aux méthodes de maximum de vraisemblance. À la différence de ces dernières, les méthodes bayésiennes utilisent une distribution *a priori* sur la quantité inférée. L'utilisation d'une distribution *a priori* permet d'interpréter le résultat comme une distribution de la quantité de données. Les méthodes bayésiennes datent des années 1970 et l'utilisation récente des méthodes de Chaîne de Markov de Monte Carlo (Markov chain Monte Carlo) leur a donné une nouvelle impulsion (Rannala et Yang, 1996; Rannala, 1997; Larget et Simon, 1999; Li et al 2000; Huelsenbeck et Ronquist. 2001). L'hypothèse optimale est celle qui maximise la probabilité *a posteriori*. La probabilité *a posteriori* de l'hypothèse est proportionnelle à la vraisemblance multipliée par

la probabilité *a priori* de cette hypothèse. Les probabilités *a priori* des différentes hypothèses dépendent des suppositions des scientifiques concernant les relations phylogénétiques possibles au niveau des données. Dans plusieurs cas, les chercheurs n'ont aucune information à propos des distributions de la probabilité *a priori*. La manière la plus simple de résoudre ce problème est de spécifier une probabilité *a priori* uniforme c'est-à-dire avec laquelle toutes les valeurs possibles d'un paramètre ont la même probabilité *a priori*.

Dans l'analyse bayésienne, le résultat final ne dépend pas d'une valeur spécifique mais de la considération de toutes les valeurs possibles des paramètres. Même s'il y'a assez de données pour estimer plusieurs paramètres, les algorithmes d'escalade « hill climbing », utilisés pour trouver le score de maximum de vraisemblance, peuvent être très lents et de moins en moins fiables au fur et à mesure que le nombre de paramètres augmente. Contrairement à cela, les méthodes bayésiennes reposent sur un algorithme qui ne cherche pas forcément à trouver le point le plus haut dans l'espace des valeurs des paramètres (optimum global). Par rapport au maximum de vraisemblance, les méthodes bayésiennes ont l'avantage d'avoir une vitesse de calcul élevée et la possibilité d'incorporer des modèles d'évolution complexes.

Les deux logiciels les plus connus réalisant les méthodes bayésiennes sont : MrBAYES developpé par Huelsenbeck (Huelsenbeck et Ronquist, 2001) et BAMBE (Larget et Simon, 1999).

1.6 Références

Adachi, J. et Hasegawa, M: (1992). Amino acid substitution of proteins coded for in mitochondrial DNA during mammalian evolution. *Jpn. J. Genet.* **67** : 187-197.

Barthélemy, J.-P. et Guénoche A. (1991). Trees and proximity representations. *New York. John Wiley & Sons.*

Beyer, W., Stein, M., Smith, T., et Ulam, S. (1974). A molecular sequence metric and evolutionary trees. *Mathematical Biosciences* **19**: 9-25

Bryant, D. et Waddell, P. J. (1998). Rapid evaluation of least squares and minimum evolution criteria on phylogenetic trees. *Molecular Biology and Evolution* **15**: 1346-1359.

Cavalli-Sforza, L. L. et Edwards A. W. F. (1967). Phylogenetic analysis: models and estimation procedures. *American Journal of Human Genetics* **19**: 233-257.

Darlu, P. et Tassy, P. (1993). Reconstruction phylogénétique: concepts et méthodes. *Collection Biologie Théorique, Masson, Paris*: 245.

Darwin, C. (1859). On the Origin of Species. *John Murray, London.*

Desper, R. et Gascuel, O. (2002). Fast and Accurate Phylogeny Reconstruction Algorithms Based on the Minimum-Evolution Principle. *Journal of Computational Biology* **19 (5)**: 687-705.

Doolitlle, W. F. (1999). Phylogenetic classification and the universal tree. *Science* **284**: 2124-2128.

Edwards, A. W. F. (1972). Likelihood. *Cambridge University Press, Cambridge.*

Edwards, A. W. F. et Cavalli-Sforza, L. L. (1963). The reconstruction of evolution. *Annals of Huma Genetic* **27:** 105-106.

Edwards, A. W. F. et Cavalli-Sforza, L. L. (1964). Reconstruction of evolutionary trees. *Systematics Association Publication* **6:** 67-76.

Farris, J. S. (1970). Methods for computing wagner trees. *Systematic Zoology* **19**: 83-92.

Felsenstein, J. (1981). Evolutionary trees from dna sequences: A maximum likelihood approach. *Journal of Molecular Evolution* **17**: 368-376.

Felsenstein, J. (1993). PHYLIP (PHYLogeny Inference Package) version 3.6a2, *Distributed by the author, Department of Genetics, University of Washington, Seattle, WA* .

Felsenstein, J. (2004). Inferring phylogenies. *Sunderland (MA): Sinauer Associates*: 664.

Fitch, W. M. et Margoliash, E. (1967). Construction of phylogenetic trees. *Science* **155**: 279-284.

Fitch, W. M. (1971). Toward defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology* **20**: 406-416.

Guindon, S., et Gascuel, O. (2003). A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology* **52**: 696-704.

Haeckel, E. (1860). "Über neue, lebende Radiolarien des Mittlmeeres," *Monatsberichte der Königlichen.* 794-817.

Hamming, R. W. (1950). Error-detecting and error-correcting codes, *Bell System Technical Journal* **29 (2)**: 147-160.

Harry, M. (2001). Espèce phylogénie et evolution. *Chap in Génétique moléculaire et évolutive. Paris* Éditions Maloine. 263-305.

Hillis, D. M. (1996). Inferring complex phylogenies. Nature 383: 130-131.

Huelsenbeck, J. P. et Ronquist, F. (2001). MRBAYES : Bayesian inference of phylogenetic trees. *Bioinformatics* **17**: 754-755.

Jukes, T. H. et Cantor, C. (1969). Mammalian Protein Metabolism, *Chap in Evolution of protein molecules. Academic Press New York.* 21-132.

Kidd, K. K. et Sgaramella-Zonta, L. A. (1971). Phylogenetic analysis concepts and methods. *American Journal of Human Genetics* **23**: 235-252.

Kim, J. H. (1996). General inconsistency conditions for maximum parsimony: effects of branch lengths and increasing numbers of taxa. *Systematic Biolology* **45**: 363-374.

Kim, J. et Warnow, T. (1999). Tutorial on phylogenetic tree estimation. *In: Proc. 7th Int'l Conf. on Intelligent Systems for molecular Biology (USNB99).*

Kimura, M. (1981). Estimation of the evolutionary distances between homologous nucleotide sequences. Proc Natl Acad Sci USA 78: 454-458.

Larget, B. et Simon, D. L. (1999). Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees. *Molecular Biology and Evolution* **16**: 750-759.

Legendre, P. et Makarenkov, V. (2002), Reconstruction of biogeographic and evolutionary networks using reticulograms, *Systematic Biology* **51 (2)**: 199-216.

Li W-H. (1997). Molecular evolution. *Sunderland, Massachussets: Sinauer Assoc.* 487.

Li, S., Pearl, D. K. et Doss, H. (2000), Phylogenetic tree construction using Markov chain Monte Carlo. *J Am Stat Assoc* **95**: 493-508.

Makarenkov, V. et Leclerc, B. (1999). An algorithm for the fitting of a phylogenetic tree according to a weighted least-squares criterion. *Journal of Classification* **16**: 3-26.

Makarenkov, V. (2001). T-REX: reconstructing and visualizing phylogenetic trees and reticulation networks. *Bioinformatics* **17**: 664-668.

Neyman, J. (1971). Molecular studies of evolution: A source of novel statistical problems. *In Statistical Decision Theory and Related Topics, ed. S. S. Gupta and J. Yackel. Academic Press, New York.* 1-27.

Rannala, B. et Yang, Z. (1996). Probability distribution of molecular evolutionary trees: a new method of phylogenetic inference. *Journal of. Molecular. Evolution* **43**: 304-311.

Rannala, B. (1997). Gene genealogy in a population of variable size. *Heredity* **78**: 890-896.

Rzhetsky, A. et Nei, M. (1992). A simple method for estimating and testing minimum evolution trees. *Comput. Appl. Biosci.* **10**: 409-412.

Saitou, N. et Nei, M. (1987). The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution* **4**: 406-425.

Smouse,. P. E. (2000), Reticulation inside the species boundary. *Journal of Classification* **17**: 165-173.

Sneath, P. H. A. (2000). Reticulate evolution in bacteria and other organisms : how can we study it ? *Journal of Classification* **17**: 159-163.

Sokal, R. R et Sneath, P. H. A. (1973). Numerical Taxonomy. *Freeman, San Francisco.*

Sonea, S. et Mathieu, L. G. (2000). Prokaryotology- A coherent view. *Presses de l'Université de Montréal, Montréal.*

Sonea, S. et Panisset, M. (1976). Pour une nouvelle bactériologie. Revue Canadienne de Biologie **35**: 103-167.

Sonea, S. et Panisset, M. (1981). Introduction à la nouvelle bactériologie. *Presse de l'Université de Montréal, Montréal and Masson, Pari* : 127.

Strimmer, K. et von Haeseler. (1996). Quartet puzzling: A quartet maximum Likelihood method for reconstructing tree topologies. *Molecular Biology and Evolution* **13**: 964-969.

Swofford, D. L., Olsen, G. J., Waddell, P. J. et Hillis, D. M. (1996). Phylogenetic inference. *In D. M. Hillis, C. Moritz, and B. Mable (eds.) Molecular Systematics (2nd ed.), Sinauer Associates, Sunderland, Massachusetts*: 407-514.

Whelan, S. et Goldman, N. (2001). A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Eveolution* **18**: 691-699.

CHAPITRE II

PARALLÉLISATION DES PROGRAMMES INFORMATIQUES

2.1 Parallélisme

2.1.1 Introduction

Dans ce chapitre nous ferons une introduction générale du parallélisme. Nous présenterons les différentes formes de parallélisme, les architectures de systèmes parallèles, ainsi que les environnements qui permettent de supporter l'exécution des applications de façon à réaliser le parallélisme. Ce chapitre devrait permettre de comprendre le choix des formes de parallélismes que nous avons effectués, le choix de l'architecture de la machine pour laquelle les programmes parallèles ont été implémentés, ainsi que la librairie utilisée (MPI) pour effectuer la parallélisation.

Dans la perpétuelle recherche de performances élevées, et sachant que les besoins de certaines applications en terme de performance dépassent la capacité des microprocesseurs standard, les concepteurs de systèmes informatiques se sont tournés, depuis la fin des années 1960, vers des structures multiprocesseurs qui mettent en œuvre plusieurs processeurs au sein d'un même système. Ces concepteurs se sont intéressés au parallélisme en fondant leurs

systèmes sur des processeurs standard au lieu de créer et de maintenir leurs propres microprocesseurs.

En 1966, Philip Bernstein a défini les conditions exprimant la possibilité d'exécution des programmes en parallèle :

> Soit P_1 et P_2 deux programmes; on suppose que chacun des programmes utilise des variables en entrée (E_1 et E_2) et produit des valeurs en sortie (S_1 et S_2). Les programmes P_1 et P_2 sont exécutables en parallèle (noté $P_1 \mid\mid P_2$) si et seulement si les conditions suivantes sont respectées : $E_1 \cap S_2 = \emptyset$, $E_2 \cap S_1 = \emptyset$ et $S_1 \cap S_2 = \emptyset$.
>
> De façon plus générale, un ensemble de programmes P_1, P_2,.., P_n, est exécutable en parallèle si et seulement si les conditions de Bernstein sont satisfaites.
>
> (Chevance, 2000, page 56).

Néanmoins, dans la pratique, le respect de ces conditions est peu probable. De plus, dans le domaine du parallélisme, on parle de la notion de « grain » de parallélisme qui fait référence à la taille des sections du code susceptibles d'être exécutées en parallèle. On parle de « grain fin » lorsque la taille de la section ne représente que quelques instructions et de « gros grain » lorsque les séquences représentent un nombre important d'instructions.

La source du parallélisme se trouve au niveau de la formulation des problèmes. Il existe trois formes de parallélisme :

- Parallélisme de données : « La même opération est effectuée par des processeurs différents sur des ensembles disjoints de données » (Chevance, 2000, page 56);
- Parallélisme de contrôle : « Différentes opérations sont réalisées simultanément. Ce parallélisme est utilisé lorsque le programme est composé de parties indépendantes ou bien lorsque certaines structures de contrôle (telles que les boucles) sont susceptibles d'être exécutées en parallèle » (Chevance, 2000, page 56);
- Parallélisme de flux : « Les opérations sur un même flux de données peuvent être enchaînées avec un certain niveau de recouvrement, c'est-à-dire que l'opération suivante peut être amorcée avant que la précédente ne soit terminée. C'est le mode de travail à la chaîne ou de pipeline » (Chevance, 2000, page 56).

La difficulté majeure dans les systèmes parallèles réside dans la programmation des applications car cette dernière implique des opérations de synchronisation pour les accès aux données partagées et de rendez-vous entre les différentes parties du programme.

Les environnements qui permettent de supporter l'exécution des applications de façon à exprimer le parallélisme qui seront présenté dans ce chapitre sont :

- PVM (présenté de façon sommaire).
- MPI (présenté plus en détails).

La programmation parallèle consiste à la conception, à l'exécution et à l'agencement des programmes sur les machines parallèles afin de tirer profit des systèmes de calcul parallèles. Elle consiste également à l'application des méthodes de programmation parallèle aux programmes séquentiels existants. Le principe général de la programmation parallèle est de diviser le problème global en charges de travail, d'assigner ces charges aux processeurs et de synchroniser les travaux des processeurs afin d'obtenir des résultats valides. La programmation parallèle peut seulement être appliquée aux problèmes qui sont parallélisables, la plupart du temps sans dépendance de données (Grama et al. 2003).

Il y a deux approches principales à la programmation parallèle :

- Le parallélisme implicite : le système (le compilateur ou un autre programme) divise le problème et assigne les charges aux processeurs automatiquement.
- Le parallélisme explicite : le programmeur doit annoter son programme pour montrer comment il doit être divisé. Plusieurs facteurs et techniques de parallélisation influent grandement sur la programmation parallèle.

2.1.2 Architectures des systèmes parallèles

Donner une classification des architectures parallèles est un exercice auquel se sont essayés de nombreux experts depuis fort longtemps. Il existe entre autres :

- La classification de Flynn en 1972 (Flynn, 1972);
- La classification de Hockney en 1987 (Hockney, 1987);
- La classification de Treleaven en 1988 (Treleaven, 1988);
- La classification de Skillicorn en 1988 (Skillicorn, 1988);
- La classification de Krishnamurthy en 1989 (Krishnamurthy, 1989);
- La classification de Duncan en 1990 (Duncan, 1990);
- La classification de Germain-Renaud en 1991 (Germain-Renaud et Sansonnet, 1991);
- La classification d'Ibett en 1992 (Ibett, 1992);
- La classification d'Esclangon en 1992 (Escalangon, 1992).

Je présenterai ici seulement la classification de Flynn (Flynn, 1972) puisqu'elle est la plus connue. Cette classification permet de comprendre le système utilisé pour réaliser la parallélisation. Par la suite, je présenterai les différentes manières existant pour associer la mémoire (ou le disque) à un système multiprocesseur.

2.1.2.1 Classification de Flynn

La classification de Flynn se décompose en quatre catégories.

a. SISD (Single Instruction Single Data Stream, ou flot unique d'instructions et flot unique de données)

Dans cette architecture, un seul flot d'instructions opère une transformation sur un même flot de données.

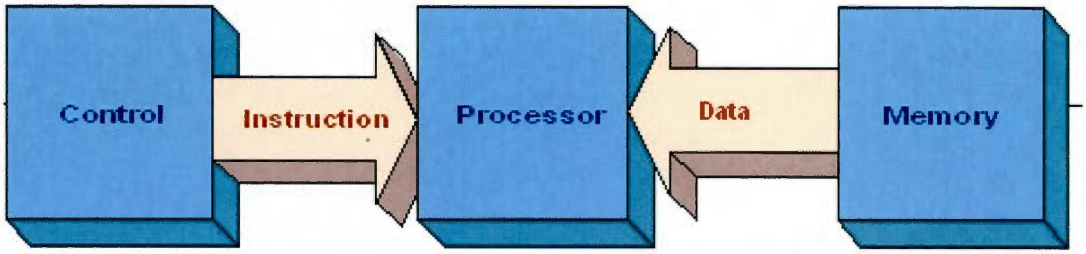


Figure II-1 Architecture SISD.

b. SIMD (Single Instruction Multiple Data Stream, ou flot unique d'instructions et flots multiples de données) :

Dans cette architecture, le même flot d'instructions est appliqué à des ensembles disjoints de données.

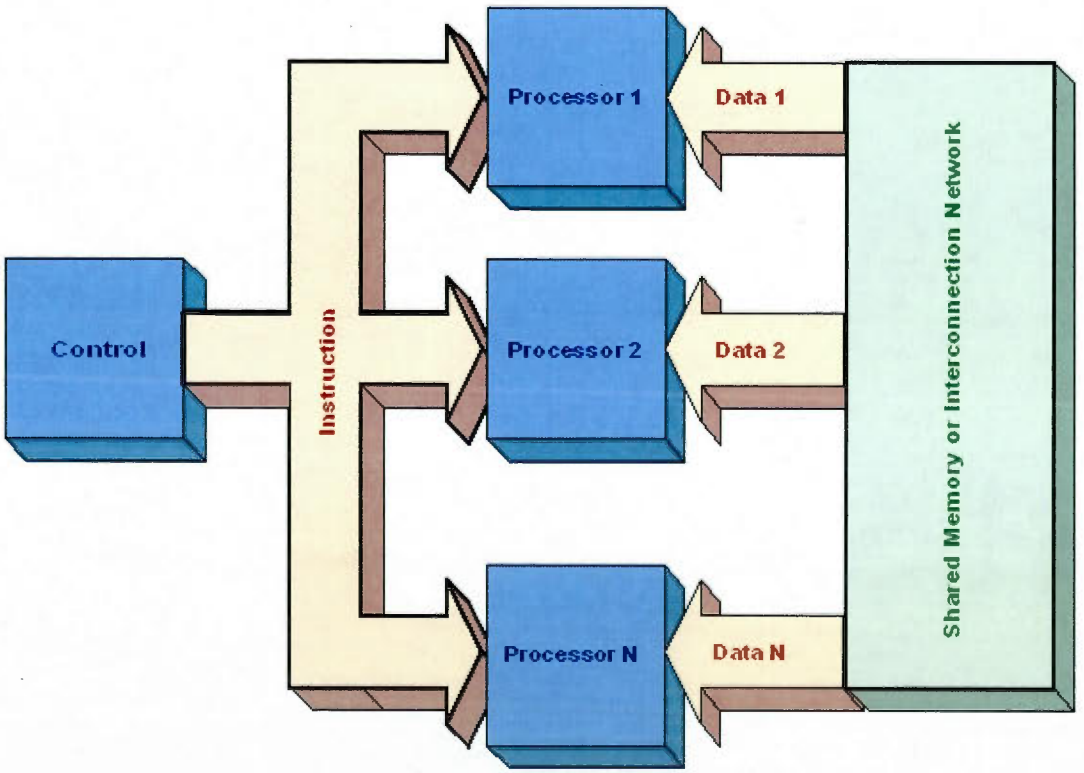


Figure II-2 Architecture SIMD.

c. MISD (Multiple Instruction Single Data Stream, ou flots d'instructions et flot unique de données) :

Dans cette architecture, plusieurs flots d'instructions sont appliqués à un même flot de données.

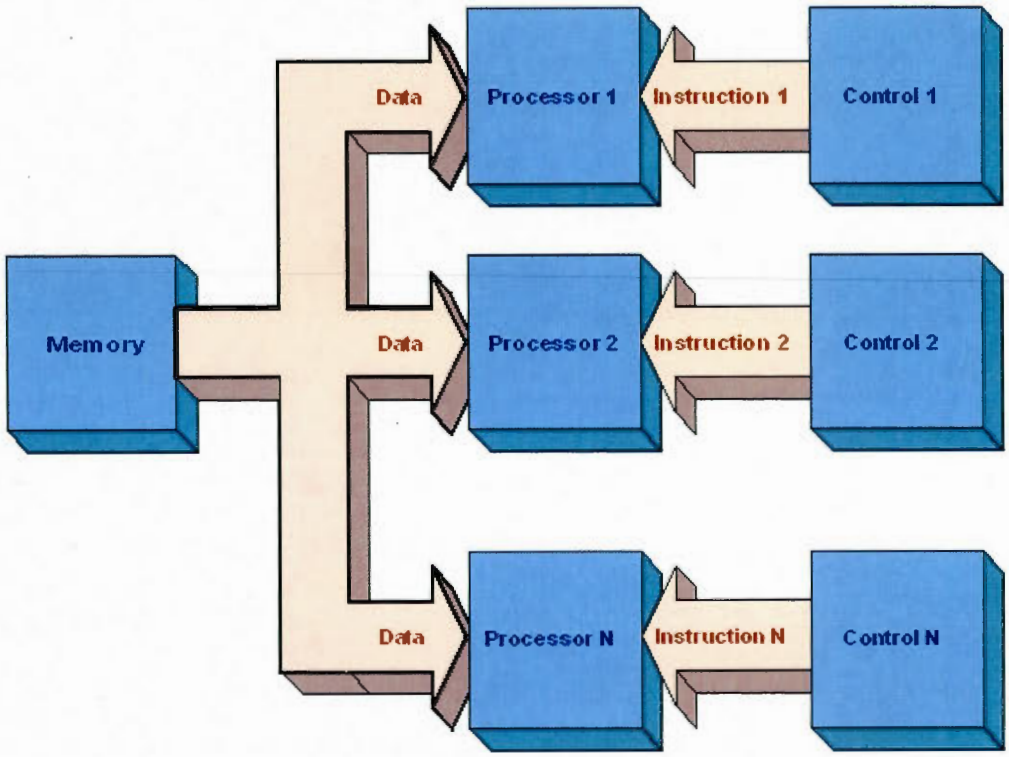


Figure II-3 Architecture MISD.

d. MIMD (Multiple Instruction Multiple Data Stream, ou flot multiple d'instructions et flots multiples de données) :

Dans cette architecture, des flots d'instructions indépendants sont appliqués à des ensembles de données indépendants.

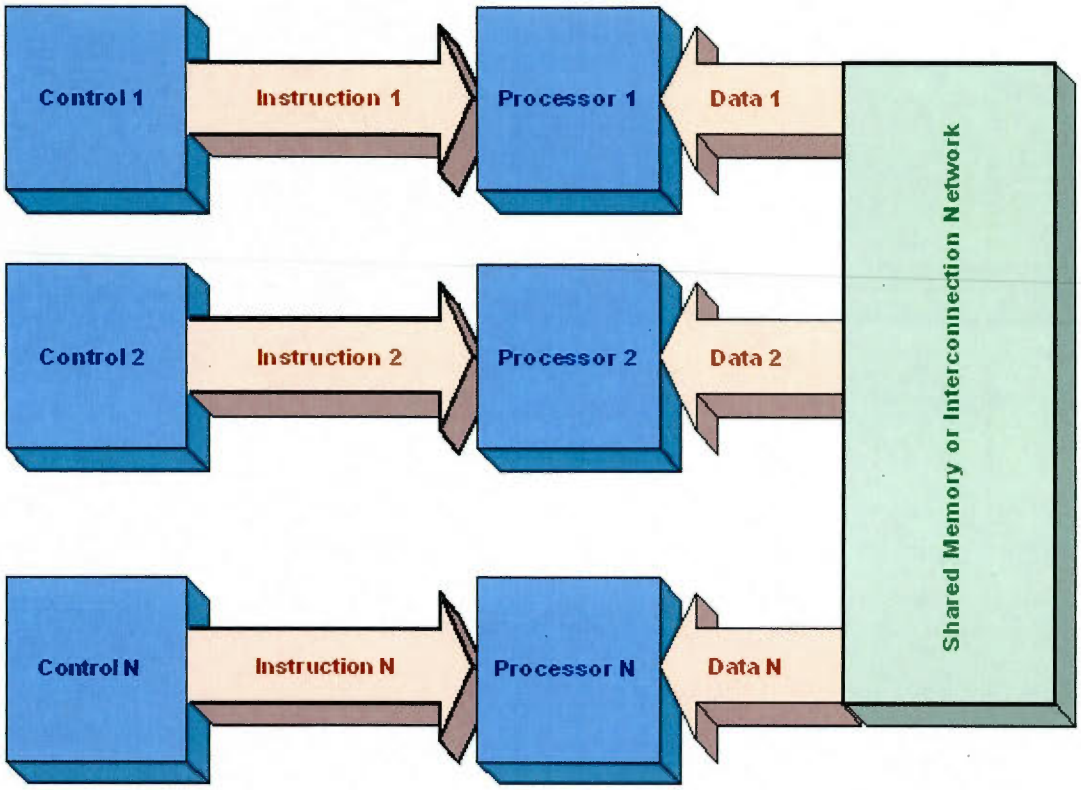


Figure II-4 Architecture MIMD.

2.1.2.2 Architecture standard d'un système parallèle

En matière de liaison avec les disques, on distingue trois grandes classes d'architectures (Chevanche, 2000) :

a. Architecture « share everything » ou « tout partagé » :

Dans cette architecture, l'ensemble des processeurs du système fonctionne sous le contrôle d'un seul système d'exploitation. Une opération d'entrée-sortie peut être initialisée par n'importe quel processeur.

Les avantages de cette architecture sont :

- Simplicité du parallélisme;
- Bonne utilisation des ressources;
- Équilibrage de charge naturelle;
- Communication inter-processeurs efficace (car fondée sur une mémoire partagée cohérente);
- Solution en cours de banalisation (baisse des prix).

Les inconvénients de cette architecture sont :

- Disponibilité du système difficile à assurer;
- Scalabilité limitée (quelques dizaines de processeurs).

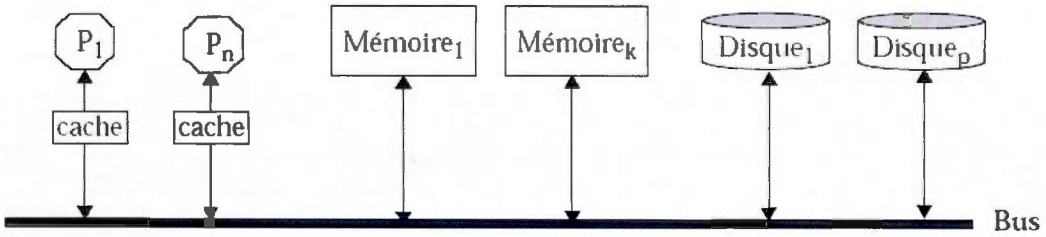


Figure II-5 Architecture « share everything ».

b. Architecture « share nothing » ou « sans partage » :

Dans cette architecture, chacun des nœuds qui composent le système fonctionne sous le contrôle de sa propre copie du système d'exploitation et a un accès exclusif aux disques qui lui sont rattachés.

Les avantages de cette architecture sont :

- Bonne disponibilité;
- Bonne scalabilité (100 processeurs et au-delà);
- Faible coût grâce à la réutilisation des composants standard;
- Bon équilibrage de charge (les données à fort taux de partage peuvent être répliquées).

Les inconvénients de cette architecture sont :

- Interaction entre les nœuds pour la synchronisation des mises à jour des données;
- Saturation du réseau d'interconnexion due aux transferts entre nœuds et disques;
- Coût du maintien de la cohérence des copies multiples, en particulier si les mises à jour sont fréquentes.

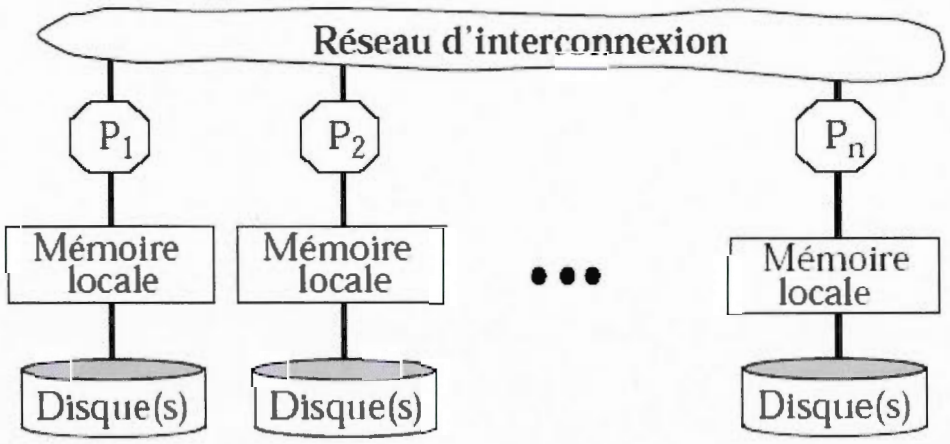


Figure II-6 Architecture « share nothing ».

La machine que nous avons utilisée pour effectuer la parallélisation est un cluster composé de 16 lames. Un cluster se caractérise par :

- L'utilisation d'une technologie de réseau d'interconnexion standard (Fibre Channel, Réseau local rapide).
- Un nombre maximal de nœuds limité, dans la pratique, à une dizaine de nœuds.

Cette machine repose sur une architecture « share nothing ». Chaque lame est composée de deux processeurs de 3.2 GHZ chacun, et possède sa propre mémoire sur laquelle elle a un accès exclusif.

c. Architecture « shared disks » ou « disque partagé » :

Dans cette architecture, chacun des nœuds qui composent le système fonctionne sous le contrôle de sa propre copie du système d'exploitation et peut accéder directement aux autres disques (mémoire) qui sont partagés entre les différents nœuds. Cette architecture n'exclut pas qu'un nœud ait ses propres disques, ce qui est d'ailleurs souvent le cas, avec au moins un disque, dit système, par nœud. Un disque système contient le système d'exploitation, les zones réservées à la pagination, à la demande ainsi que, souvent, les programmes d'applications les plus fréquemment utilisés.

Les avantages de cette architecture sont :

- Bonne disponibilité;
- Excellente scalabilité (plusieurs centaines de processeurs);
- Faible coût, grâce à la réutilisation de composants standards.

Les inconvénients de cette architecture sont :

- Équilibrage en charge difficile;
- Difficile à administrer et à optimiser du fait du partitionnement des données;
- Forte dépendance des performances vis-à-vis des caractéristiques du réseau d'interconnexion;
- Coût de la parallélisation, même pour les requêtes simples;

- Coût de maintien de la cohérence des copies multiples, en particulier si les mises à jour sont fréquentes.

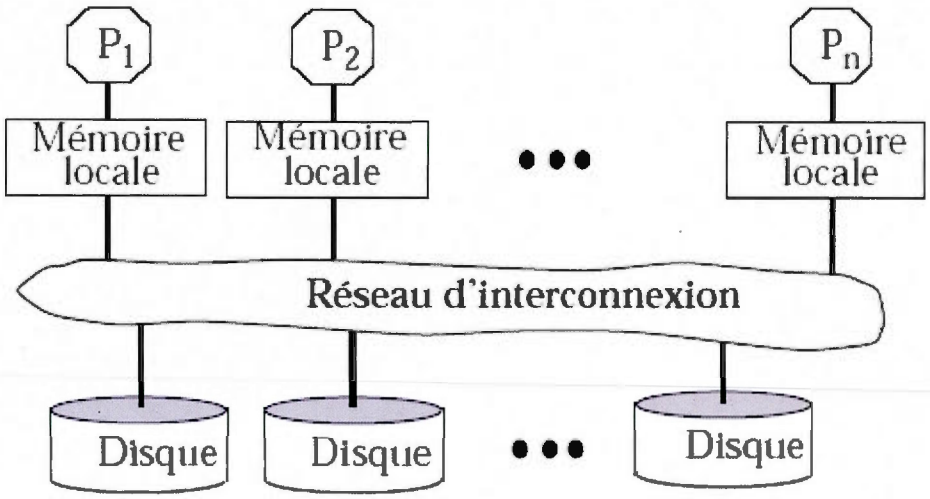


Figure II-7 Architecture « shared disks ».

2.2 Machine virtuelle parallèle (Parallel Virual Machine)

Pour comprendre le modèle de programmation « Parallel Virtual Machine » (PVM, 2006), il faut tout d'abord comprendre le concept de « machine virtuelle ». Une « machine virtuelle » est composée d'une ou de plusieurs machines hôtes qui communiquent les unes avec les autres à travers le protocole standard TCP/IP. Les machines hôtes peuvent être une collection de postes de travail hétérogènes, des superordinateurs, des PCs (Personal Computer), etc. Du point de vue du programmeur, la machine virtuelle est un ordinateur unique avec une grande mémoire distribuée. Dans le modèle PVM, la « machine virtuelle » peut être constituée d'un nombre illimité d'hôtes et ces derniers peuvent être ajoutés ou supprimés de façon dynamique. Les processus qui s'exécutent sur les machines parallèles sont appelés tâches et le modèle PVM suppose l'exécution de plusieurs tâches au niveau d'un seul hôte et leur communication les unes avec les autres. Ces tâches peuvent être gérées de manière dynamique aussi.

Le système PVM est issu d'un projet interne de l'Oak Ridge National Laboratory (ORNL) aux Etats-Unis en 1989. La première version a servi de base, en 1991, à un projet plus vaste commun à trois universités des États-Unis et a été soutenue par plusieurs programmes, tant gouvernementaux qu'industriels.

Les techniques de calcul parallèle du modèle de programmation PVM sont maintenant utilisées dans plusieurs disciplines scientifiques. La mise à disposition des codes sources de PVM dans le « domaine public » ainsi que le fort développement des réseaux de télécommunications, en particulier la couverture mondiale réalisée par les protocoles IP et leur présence sur toutes sortes de plates-formes, ont été des éléments déterminants de son succès.

2.3 Interface de Passage de Messages (Message Passing Interface)

2.3.1 Introduction

« Message Passing Interface » (MPI, 2006) est une librairie standard de passage de messages. Le passage de messages est une méthode par laquelle les données de la mémoire d'un processeur donné sont copiées dans la mémoire d'un autre processeur. La librairie standard de passage de messages est constituée d'une collection de routines servant à faciliter la communication entre les processeurs d'une machine parallèle. MPI n'est pas un standard en tant que tel parce qu'il n'a pas été publié par des organisations telles que ISO ou AINSI. C'est un standard parce qu'il a été conçu dans un forum ouvert qui incluait des fournisseurs de matériels informatiques, des chercheurs, des universitaires, des représentants de plus de 40 organisations, etc.

Le but principal de MPI était de concevoir une librairie assez complète permettant :

- La communication : Une communication de haut niveau et de bas niveau;
- La portabilité : Elle est très importante avec les changements rapides dans les environnements des machines à haute performance et dans domaine de la technologie;
- La modularité (dans le développement des différentes librairies);

- Hétérogénéité;
- Gestion des groupes de processus;
- Gestion des processus.

Cette librairie reste efficace sur un maximum de machines parallèles. Elle a de bonnes chances de devenir la librairie standard qui remplacera toutes les autres existantes dans le domaine de la programmation parallèle au cours des prochaines années. Il existe des implémentations de MPI pour les systèmes Unix et Windows. Ces implémentations supportent les programmes écrits en C, C++, Java et Fortran.

2.3.2 Historique de MPI

MPI résulte des efforts de nombreux groupes d'individus entre les années 1992 et 1994. Déjà, dans les années 1980 et au début des années 1990, la mémoire distribuée et la programmation parallèle ont conduit au développement de plusieurs logiciels qui étaient pour la plupart incompatibles et dédiés à l'écriture de tels programmes. Habituellement, les incompatibilités se trouvaient au niveau de la portabilité, de la performance, de la portabilité et du prix. Toutes ces incompatibilités ont entraîné le besoin d'un standard.

En avril 1992, un atelier sur les standards de passage de messages pour les machines à mémoire distribuée fut commendité par le centre de recherche sur la programmation parallèle (Willamsburg, Virginie). Cet atelier a permis de discuter des dispositifs basiques qui sont essentiels à une interface standard de passage de messages et une équipe de développement a été mise en place pour continuer la standardisation. En novembre 1992, l'équipe de développement se rencontra à Minneapolis, la version MPI-1 fut présentée et le forum MPI fut créé. En novembre 1993 (Supercomputing 93 conference), MPI standard a été dévoilé et une version finale a été rendue disponible en mai 1994. MPI-2 est la continuation de MPI-1. Il traite des sujets qui vont au delà de ceux de la première spécification de MPI.

Aujourd'hui, les différentes implémentations de MPI sont une combinaison de MPI-1 et MPI-2 et peu d'implémentations incluent la totalité des fonctionnalités de MPI-1 et de MPI-2.

2.3.3 Raisons principales d'utilisation de la librairie MPI

Les principales raisons sont les suivantes :

- La standardisation : MPI est la seule librairie de passage de messages qui peut être considérée standard. Cette librairie remplace progressivement toutes les anciennes librairies de passage de messages;
- La portabilité : Pas besoin de faire une modification sur le code à la suite d'un changement de plateforme qui supporte le standard MPI;
- La performance : Il y a la possibilité d'utiliser le «Native Hardware Features » pour optimiser la performance;
- La fonctionnalité : Plus de 115 routines ont été définies seulement pour MPI-1;
- La disponibilité : Plusieurs implémentations de MPI sont disponibles (dans le domaine industriel et public;
- L'existence de programmes bioinformatiques parallélisés à travers la librairie MPI : mpi-Blast (Lin et al., 2005) et clustal-mpi (Li, 2003).

2.3.4 Modèle de programmation

MPI est pratiquement utilisé sur la plupart des modèles de programmation parallèle à mémoire distribuée (figure II-8). Les plateformes matérielles sont :

- Mémoire distribuée : À l'origine, MPI visait juste les machines à mémoires distribuées.
- Mémoire partagée : Comme les machines à mémoire partagée sont devenues très populaires (architectures SMP / NUMA), des implémentations MPI apparurent pour ces genre de systèmes.
- Hybride : MPI est maintenant utilisé pour tout système à architecture parallèle comme les machines massivement parallèles, les clusters SMP, les clusters de postes de travail et les réseaux hétérogènes.

Au niveau de MPI, tout le parallélisme est explicite, c'est-à-dire que le programmeur est responsable de l'indentification correcte du parallélisme et de l'implémentation de l'algorithme parallèle en utilisant MPI.

Le nombre de tâches dédiées à s'exécuter en parallèle est statique. Cependant, l'engendrement de nouvelles tâches durant l'exécution est impossible.

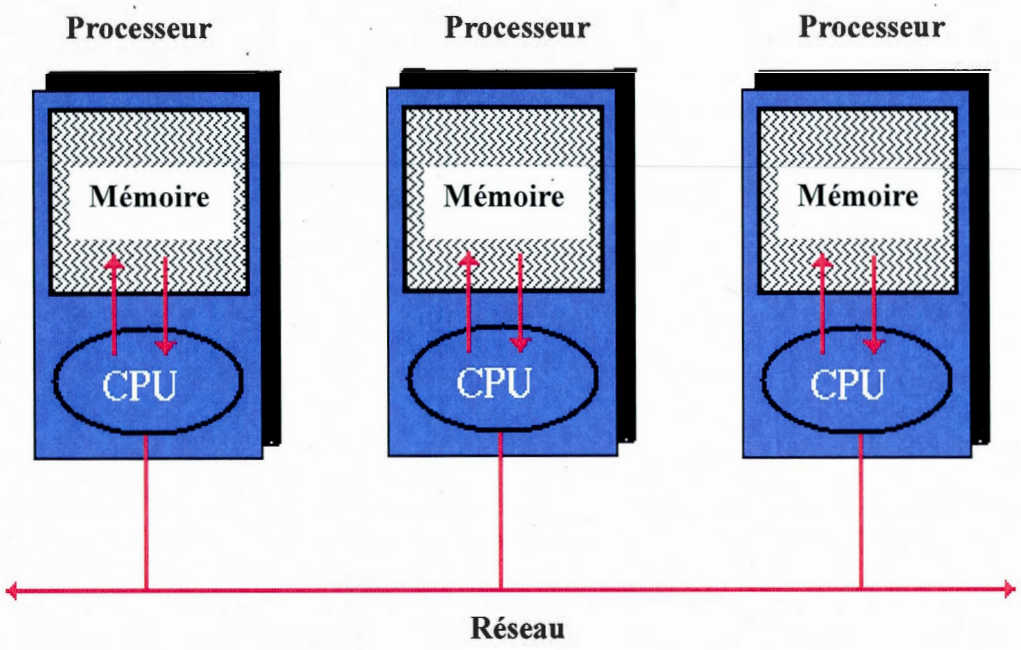


Figure II-8 Système à mémoire distribuée.

2.3.5 Structure générale d'un programme MPI

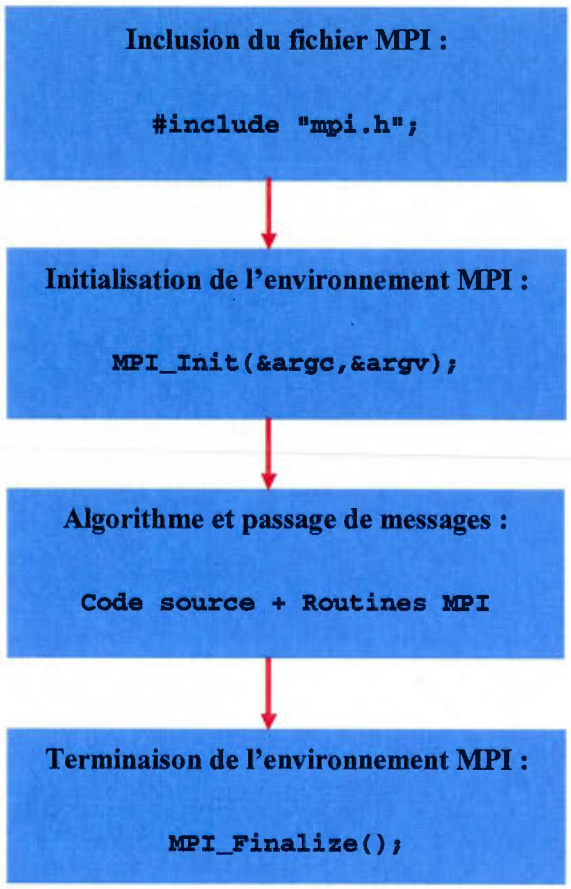


Figure II-9 Structure d'un programme MPI.

2.3.6 Communicateurs, groupes et rang

MPI utilise des objets appelés « communicateurs » ou « groupes » pour définir le groupe de processus (i.e., ensemble de processeurs qui exécutent les processus du programme) qui peuvent communiquer avec tous les autres. La plupart des routines MPI exige la spécification du communicateur en argument. Par défaut, la constante MPI_COMM_WORLD qui comprend tous les processus actifs, est le communicateur qui inclut tous les processus MPI.

Dans un communicateur, chaque processeur est identifié de façon unique par un entier qui lui est assigné par le système lors de l'initialisation des processus. Le rang est aussi appelé

parfois identifiant du processeur. Les rangs sont contigus et commencent à zéro. Les rangs sont généralement utilisés pour spécifier la source et la destination des messages. Ils peuvent aussi être utilisés pour le contrôle de l'exécution du programme juste par certains processeurs (par exemple, si *rang* = 0 alors exécuter la suite d'instructions suivante).

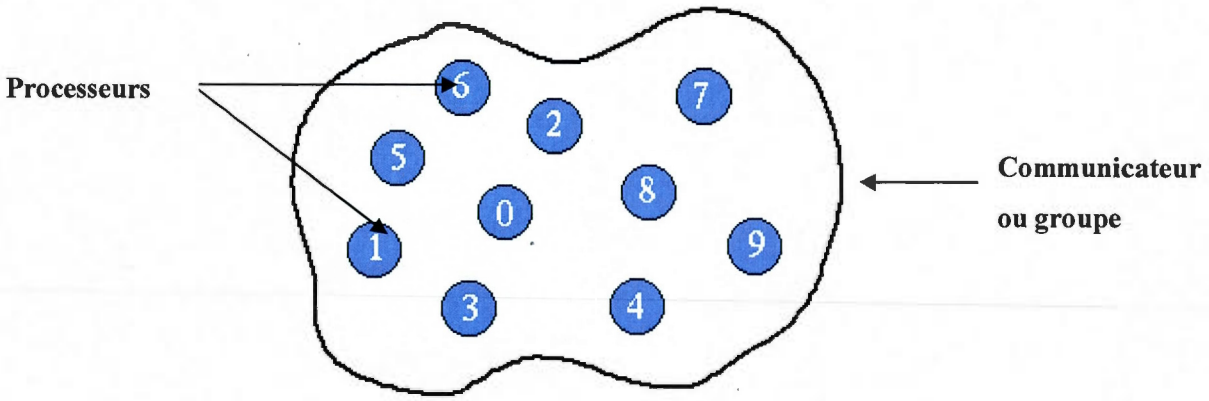


Figure II-10 Communicateur ou groupe MPI.

2.3.7 Routines MPI

Les routines MPI peuvent être divisées en trois grandes catégories selon leur fonction :

2.3.7.1 Routines de gestion de l'environnement d'exécution

Les routines de gestion de l'environnement d'exécution sont utilisées afin d'atteindre des buts spécifiques tels que l'initialisation et la terminaison de l'environnement MPI, le questionnement de l'environnement tel que son identité, etc. Les routines de gestion d'environnement les plus couramment utilisées sont :

a- MPI_init

Cette routine permet de procéder à l'initialisation de l'environnement d'exécution de MPI. Elle doit être appelée une seule fois dans tous les programmes MPI et avant toutes les autres routines. Pour les programmes en C, cette routine peut être utilisée pour passer les arguments de la ligne de commande à tous les processeurs, cependant les implémentations peuvent varier.

Exemple :

```
MPI_Init (&argc, &argv)
```

Les arguments *argc* et *argv* sont requis seulement quand la base du programme est écrite en C ou C++.

b- MPI_Comm_size

Cette routine détermine le nombre de processus associés à un communicateur. Elle est généralement utilisée avec le communicateur par défaut MPI_COMM_WORLD pour déterminer le nombre de processus qui sont utilisés dans l'application.

Exemple :

```
MPI_Comm_size (comm, &size),
```

où *comm* : le communicateur et *size* : nombre de processus du communicateur.

c- MPI_Comm_rank

Cette routine permet de déterminer le rang des processeurs. Initialement, un entier unique entre 0 et le nombre de processeurs-1 est assigné à chaque processeur au niveau du communicateur par défaut. Si un processeur est associé à d'autres communicateurs, il aura un rang unique avec chacun de ces communicateurs.

Exemple :

MPI_Comm_rank (comm, &rank),

où *comm :* communicateur et *rank* : le rang communicateur.

d- MPI_Abort

Cette routine termine tous les processus associés au communicateur. Cependant dans la plupart des implémentations de MPI, cette routine termine tous les processus sans tenir compte du communicateur.

Exemple :

MPI_Abort (comm, errorcode),

où *comm* : communicateur et *errorcode* : code d'erreur retourné à l'environnement.

e- MPI_Get_processor_name

Cette routine, comme son nom l'indique, retourne le nom du processeur ainsi que sa longueur. Le nom d'un processeur peut avoir une longueur maximale qui correspond à la constante MPI_MAX_PROCESSOR_NAME égale à 256.

Exemple :

MPI_Get_processor_name (&name, &resultlenght),

où *name* : est le nom du processeur et *resultlenght* : longueur du nom du processeur.

f- MPI_Initialized

Cette routine indique si la fonction MPI_init a été appelée et retourne la valeur vraie (1) ou fausse (0).

Exemple :

MPI_initialized (&flag),

où *flag est vrai si MPI_init a été déjà appelé et faux sinon.*

g- MPI_Wtime

Cette routine retourne le temps écoulé en seconde sur le processeur appelant. C'est cette routine qui a été utilisée pour calculer le gain en temps engendré par l'exécution des programmes parallélisés.

Exemple :

MPI_Wtime ()

h- MPI_Wtick

Cette routine retourne la résolution en seconde de la routine MPI_Wtime.

Exemple :

MPI_Wtick ()

i- MPI_Finalize

Cette routine permet de terminer l'environnement d'exécution de MPI. Elle doit être en principe la dernière routine appelée dans tout programme MPI. Aucune autre routine MPI ne peut être appelée après le MPI_Finalize.

Exemple :

MPI_Finalize ()

2.3.7.2 Routines de communication point à point

Les opérations MPI point à point impliquent le passage de messages entres deux différentes tâches de MPI. La première tâche exécute l'opération d'envoi du message et la seconde exécute l'opération de réception du message correspondant (figure II-11).

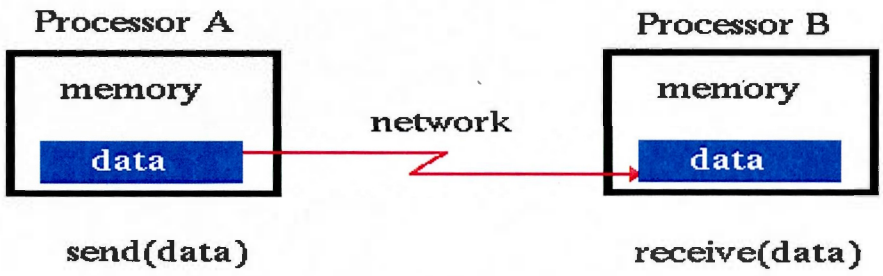


Figure II-11 Envoi et réception de messages.

La plupart des routines de communication point à point peuvent être utilisées en mode bloqué ou débloqué.

a- Routines de communication point à point en mode bloqué

Les routines de communication point à point utilisent les paramètres suivants : *buf* représente l'adresse du tampon d'envoi, *count* le nombre d'éléments à envoyer, *datatype* le type de données à envoyer, *dest* le rang du processeur de destination, *tag* le message d'étiquette, *comm* le communicateur, *status* le statut de l'objet et *request* la requête du communicateur.

Les routines les plus communes sont les suivantes :

- **MPI_Send**

C'est la routine de base d'envoi de messages en mode bloqué. La routine se termine après que le tampon de la tache d'envoi soit libre pour une réutilisation. Cette routine peut être implémentée de différentes manières en fonction des systèmes.

Exemple :

MPI_Send (&buf, count, datatype, dest, tag, comm)

- **MPI_Recv**

Cette routine permet de recevoir un message et d'effectuer un blocage jusqu'à ce que la donnée demandée soit disponible dans le tampon de la tâche réceptrice.

Exemple :

MPI_Recv (&buf, count, datatype, source, tag, comm, &status)

b- Routines de communication point à point en mode non bloqué

Les routines de communication point à point en mode non bloqué les plus communes sont les suivantes :

- **MPI_Isend**

Cette routine commence par identifier un espace dans la mémoire qui servira comme tampon d'envoi. Le traitement continue immédiatement sans attendre que le message ne soit copié hors du tampon d'application (figure II-12). Un descripteur de la demande de communication est retourné pour gérer le statut du message en suspend.

Exemple :

MPI_Isend (&buf, count, datatype, dest, tag, comm, &request)

- **MPI_Irecv**

Cette routine commence aussi par identifier un espace dans la mémoire qui servira comme tampon d'envoi. Le traitement continue immédiatement sans attendre que le message ne soit reçu et copié dans du tampon d'application (figure II-12). Un descripteur de la demande de communication est retourné pour gérer le statut du message en suspens.

Exemple :

MPI_Irecv (&buf, count, datatype, source, tag, comm, &request)

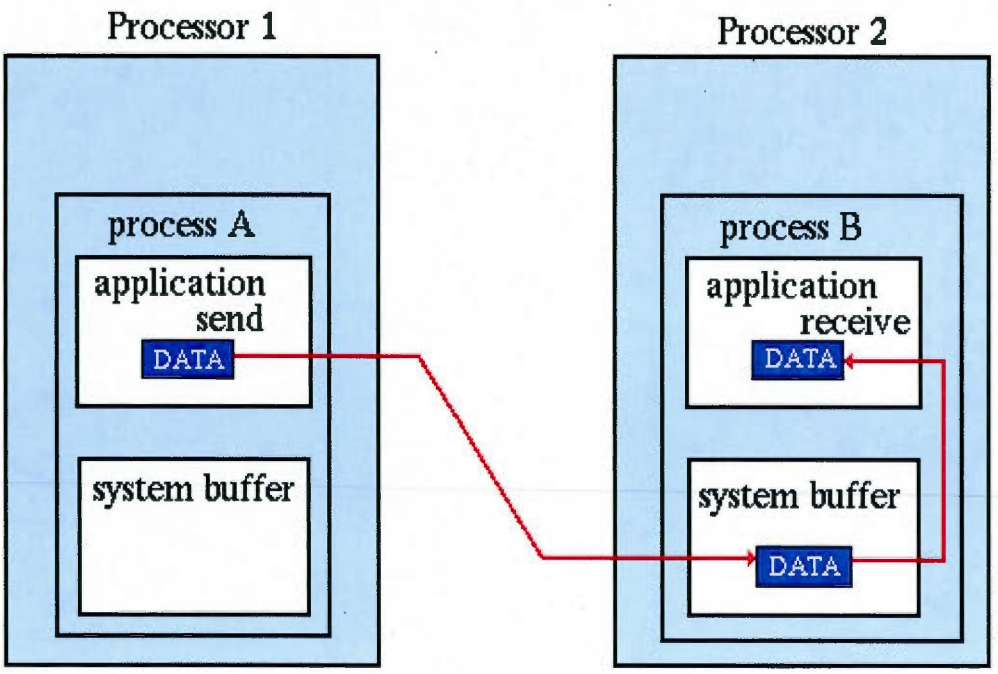


Figure II-12 Envoi de messages avec un tampon de mémoire.

2.3.7.3 Routines de communication collectives

La communication collective doit impliquer tous les processeurs d'un communicateur donné. Par défaut, ce sont tous les processeurs qui appartiennent au communicateur MPI_COMM_WORLD. C'est la responsabilité du programmeur de s'assurer que tous les processeurs sans communicateur, participent aux opérations collectives. Les communications collectives peuvent être de trois types :

a- Synchronisation :

Les processus attendent que tous les membres du groupe aient atteint le point de synchronisation.

b- Mouvement de données :

Ce sont les opérations du genre broadcast, scatter/gather (distribution et regroupement des données), etc.

c- Calcul collectif :

Un membre du groupe collecte les données provenant de tous les autres processeurs du groupe et effectue une opération (minimum, maximum, addition, etc.) sur les données collectées.

Toutes les opérations collectives sont bloquantes. Les routines des opérations collectives ne passent pas en paramètre d'argument « tag » et peuvent être utilisées juste avec les types prédéfinis de MPI.

Les routines de communication collectives utilisent quelques paramètres supplémentaires par rapport aux routines vues précédemment qui sont : *root* qui représente le rang du processus d'envoi et *op* qui représente l'opération de réduction utilisée.

Les routines de communication collectives les plus communes sont les suivantes :

- **MPI_Barrier**

Cette routine crée une barrière de synchronisation dans un groupe (communicateur). Quand une tâche atteint l'appel de cette routine, elle entre en attente jusqu'à ce que les autres tâches du groupe atteignent l'appel de la même routine.

Exemple :

MPI_Barrier (comm)

- **b- MPI_Bcast**

Cette routine émet un message du processeur racine « root » à tous les autres processeurs du même groupe.

Exemple :

MPI_Bcast (&buffer, count, datatype, root, comm)

- **c- MPI_Scatter**

Cette routine distribue des messages distincts d'une source unique à tous les processeurs du groupe.

Exemple :

MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, ...comm)

- **d- MPI_Gather**

Cette routine regroupe les messages distincts de chaque processeur du groupe dans un seul processeur de destination. Cette routine est l'inverse de la routine MPI_Scatter.

Exemple :

MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, ... comm)

- **e- MPI_Reduce**

Cette routine applique une opération de réduction sur tous les processeurs du groupe et convertit le résultat (correspondant à la réduction) au niveau d'une seule tâche (la tâche *root*). L'utilisateur peut définir ses propres opérations de réduction ou utiliser celles prédéfinies par MPI telles que MPI_MAX, MPI_MIN, etc.

Exemple :

MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)

2.3.8 Types de données et ensemble des routines de MPI pour l'environnement C.

2.3.8.1 Type de données

Type de données de MPI pour l'environnement C	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack() / MPI_Unpack

Tableau II-1 Type de données de MPI.

Pour des raisons de portabilité, MPI a défini ses types de données élémentaires et permet de définir des types dérivés avec MPI_PACKED (tableau II-1).

2.3.8.2 Ensemble des routines de MPI

Mis à part les routines citées précédemment, MPI fournit aussi le nécessaire afin de définir ses propres structures de données en se basant sur les types primaire de MPI. Ces structures sont appelées types de données dérivées. Les types de données primaires MPI sont continus, les types de données dérivées permettent de définir des types de données non continus.

MPI fournit plusieurs méthodes de construction de types de données dérivées (Contiguous, Vector, Indexed, Struct).

Le tableau II-2 contient la totalité des routines de MPI pour l'environnement de programme C.

Routines MPI		
MPI_Abort	MPI_Errhandler_create	MPI_Errhandler_free
MPI_Errhandler_get	MPI_Errhandler_set	MPI_Error_class
MPI_Error_string	MPI_Finalize	MPI_Get_processor_name
MPI_Init	MPI_Initialized	MPI_Wtick
MPI_Wtime	MPI_Start	MPI_Test
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_attach
MPI_Buffer_detach	MPI_Cancel	MPI_Get_count
MPI_Get_elements	MPI_Ibsend	MPI_Iprobe
MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv
MPI_Recv_init	MPI_Request_free	MPI_Rsend

MPI_Rsend_init	MPI_Send	MPI_Send_init
MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend
MPI_Ssend_init	MPI_Reduce	MPI_Startall
MPI_Scatterv	MPI_Test_cancelled	MPI_Testall
MPI_Testany	MPI_Testsome	MPI_Wait
MPI_Waitall	MPI_Waitany	MPI_Waitsome
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv	MPI_Barrier
MPI_Bcast	MPI_Gather	MPI_Gatherv
MPI_Op_create	MPI_Op_free	MPI_Scatter
MPI_Reduce_scatter	MPI_Scan	MPI_Group_translate_ranks
MPI_Group_union	MPI_Group_difference	MPI_Group_excl
MPI_Group_compare	MPI_Group_union	MPI_Group_rank
MPI_Group_free	MPI_Group_incl	MPI_Group_intersection
MPI_Group_range_excl	MPI_Group_range_incl	MPI_Group_size
MPI_Comm_split	MPI_Comm_test_inter	MPI_Intercomm_create
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank
MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_size
MPI_Type_commit	MPI_Type_extent	MPI_Type_contiguous
MPI_Intercomm_merge	MPI_Type_free	MPI_Type_hindexed
MPI_Type_count	MPI_Type_vector	MPI_Type_ub
MPI_Type_hvector	MPI_Type_indexed	MPI_Type_lb

MPI_Type_size	MPI_Type_struct	MPI_Cart_shift
MPI_Cart_coords	MPI_Cart_create	MPI_Cart_get
MPI_Cart_map	MPI_Cart_rank	MPI_Pack
MPI_Cart_sub	MPI_Cartdim_get	MPI_Dims_create
MPI_Graph_create	MPI_Graph_get	MPI_Graph_map
MPI_Graph_neighbors	MPI_Graph_neighbors_count	MPI_Graphdims_get
MPI_Topo_test	MPI_Pack_size	MPI_Unpack
MPI_Address	MPI_Attr_delete	MPI_Attr_get
MPI_Attr_put	MPI_DUP_FN	MPI_Keyval_create
MPI_Keyval_free	MPI_NULL_COPY_FN	MPI_NULL_DELETE_FN
MPI_Pcontrol		

Tableau II-2 Routines de MPI.

2.4 Références

Chevance, R. J. (2000). Serveurs multiprocesseurs, clusters et architectures parallèles. *Eyrolles*.

Duncan, R. (1990). (Control Data Corporation) A survey of Parallel *Computer Architectures, IEEE Computer*: 5-16.

Escalangon, P. (1992). Le Marché des Calculateurs Parallèles. *EDF/DER/IMA/MMN, Note technique HI-70/8102*.

Flynn, M.J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* **C-21 (9)**.

Germain-Renaud, C. et Sansonnet, J.P. (1991). Les ordinateurs Massivement parallèles. *collection 2aI, Armand Colin, Paris*.

Grama, A., Gupta, A. , Karypis, G. et Kumar, V. (2003) Introduction to Parallel Computing

Hockney, R.W. (1987). Classification and Evaluation of Parallel Computer Systems, *Springer-Verlag Lecture Notes in Computer Science* **295** : 13-25.

Ibbet, R.N. (1992). Parallel Computer Architectures: Where Next ? *Transputers '92, M. Becker et al., Editors, IOS Press* : 364-380.

Krishnamurthy, E.V. (1989). Parallel Programming, Principles and Practice. *International Computer Science Series, Addison-Wesley.*

Li, K.-B.(2003). ClustalW-MPI: ClustalW Analysis Using Distributed and Parallel Computing. *Bioinformatics* **19 (12)**:1585--1586.

Lin, H., Ma, X., Chandramohan, P., Geist, A. et Samatova, N. (Avril 2005). Efficient Data Access for Parallel BLAST. *IEEE International Parallel & Distributed Processing Symposium, Denver, CO.*

MPI (2006). http://www.llnl.gov/computing/tutorials/mpi/.

PVM (2006). http://www.csm.ornl.gov/pvm/.

Skillicorn, D.B. (November 1988). A Taxonomy for Computer Architecture. *Computer:* **21 (11)** : 46-57.

Treleaven, P.C. (1988). Parallel Architecture Overview. *Parallel Computing* **8**, North-Holland : 59-70.

Wilkinson, B. et Allen, M. (2005) Parallel Programming

CHAPITRE III

ARTICLES

« Une nouvelle méthode pour l'estimation de nucléotides manquants en vue de l'inférence phylogénétique ».

Cet article traite du problème d'inférence d'arbres phylogénétiques à partir des séquences de nucléotides (i.e. séquences d'ADN ou d'ARN) contenant des entrées manquantes. Un algorithme permettant d'estimer des données nucléotidiques partielles a été développé en vue de la reconstruction phylogénétique.

« New efficient algorithm for modeling partial and complete gene transfer scenarios »

Cet article décrit une nouvelle méthode pour prédire et visualiser les possibles transferts horizontaux de gènes dans le cadre des modèles complet et partiel.

« Algorithms for detecting complete and partial horizontal gene transfers: theory and practice»

Le dernier article détaille notre recherche sur des modèles de transferts horizontaux. Une procédure de validation statistique des transferts inférés a été conçue et présentée à l'aide d'un exemple de données réelles.

Par rapport à ces trois articles, j'ai participé au développement des algorithmes, à leur implémentation, ainsi qu'à la réalisation des simulations nécessaires pour effectuer la validation des résultats.

3.1 Une nouvelle méthode pour l'estimation de nucléotides manquants en vue de l'inférence phylogénétique

Cet article traite le problème de la reconstruction d'arbres phylogénétiques à partir de séquences contenant des nucléotides manquants ou indéterminés. Nous proposons une méthode permettant d'estimer ces nucléotides manquants dans les séquences d'ADN ou d'ARN en se basant sur des modèles d'évolution de Jukes-Cantor [JUK 69] et de Kimura 2-paramètres [KIM 80]. Cette méthode permet d'améliorer la qualité de l'inférence phylogénétique comparativement aux méthodes "Ignore Missing Sites" (IMS) et "Proportional Distribution of Missing and Ambiguous Bases" (PDMAB) incluses dans le logiciel PAUP [SWO 01]. Par ailleurs, dans la partie application, nous montrons l'utilité de la nouvelle méthode en considérant un arbre phylogénétique pour un groupe de 10 eucaryotes.

Diallo, Ab. B., Diallo, Al. B. et Makarenkov, V. (2005), Une nouvelle méthode pour l'estimation de nucléotides manquants en vue de l'inférence phylogénétique, Actes de la 12-ième Conférence de la Société Francophone de Classification, Montréal, 121-125.

Une nouvelle méthode pour l'estimation de nucléotides manquants en vue de l'inférence phylogénétique

Abdoulaye Baniré Diallo, Alpha Boubacar Diallo et Vladimir Makarenkov

Département d'informatique,
Université du Québec à Montréal,
Case postale 8888, succursale Centre-ville
Montréal (Québec) Canada, H3C 3P8

RÉSUMÉ. Cet article adresse le problème de la reconstruction d'arbres phylogénétiques à partir de séquences contenant des nucléotides manquants ou indéterminés. Nous proposons une méthode permettant d'estimer ces nucléotides manquants dans les séquences d'ADN ou d'ARN en se basant des modèles d'évolution de Jukes-Cantor [JUK 69] et de Kimura 2-paramètres [KIM 80]. Cette méthode permet d'améliorer la qualité de l'inférence phylogénétique comparativement aux méthodes "Ignore Missing Sites" (IMS) et "Proportional Distribution of Missing and Ambiguous Bases" (PDMAB) incluses dans le logiciel PAUP [SWO 01]. Par ailleurs, dans la partie application, nous montrons l'utilité de la nouvelle méthode en considérant un arbre phylogénétique pour un groupe de 10 eucaryotes.

MOTS-CLÉS : arbre phylogénétique, donnée manquante, modèle d'évolution

1 Introduction

La présence des nucléotides indéterminés dans les séquences d'ADN peut être due à la difficulté de séquencer certaines régions du génome d'un spécimen donné ou a une mauvaise conservation des spécimens. Ce problème peut représenter un grand obstacle pour la classification phylogénétique. Huelsenbeck [HUE 91] et Makarenkov et Lapointe [MAK 04] ont indiqué que les taxons qui contiennent de nombreux caractères inconnus diminuent considérablement la qualité de l'inférence phylogénétique. Dans cette étude nous proposons une nouvelle méthode, appelée *PEMV* (*Probabilistic estimation of missing values*), utilisant une approche probabiliste et visant à estimer les nucléotides manquants avant le calcul des distances d'évolution. Ici nous présentons cette méthode dans le cadre des modèles d'évolution de Jukes-Cantor [JUK 69] et de Kimura 2-paramètres [KIM 80], mais elle peut être généralisée à toute autre transformation séquences-distances. Dans la section suivante nous introduirons la nouvelle méthode et présenterons ses performances en inférant une phylogénie d'eucaryotes à partir de leur ADN mitochondriaux auxquels nous ajoutons des données manquantes. Nous comparons les performances de la nouvelle méthode à celles des méthodes *IMS* (« *Ignoring missing sites* ») et *PDMAB* (« *Proportional distribution of missing and ambiguous bases* »), qui sont incluses dans le populaire logiciel PAUP de Swofford [SWO 01]. La qualité d'inférence phylogénétique est estimée à l'aide de la distance topologique de Robinson et Foulds [ROB 80]. La méthode Neighbor Joining (NJ) de Saitou et Nei [SAI 87] a été utilisée dans les simulations pour reconstruire les arbres phylogénétiques.

2 La nouvelle méthode d'estimation des nucléotides manquants PEMV

La méthode *PEMV* est présentée ici dans le cadre des modèles d'évolution de Jukes-Cantor [JUK 69] et Kimura 2-paramètres [KIM 80]. Pour calculer les distances entre les paires de séquences, selon le modèle de Jukes-Cantor, la formule de correction suivante a été utilisée: $D = -3/4\ ln(1 - 4/3d)$, où d représente la distance observée. Selon le modèle de Kimura, la formule de correction est la suivante: $D = -1/2\ ln((1 - 2P - Q)\sqrt{(1-2Q)})$, où P représente le taux de transition et Q le taux de transversion entre les séquences.

Considérons un ensemble de séquences d'ADN (composées des nucléotides A, C, G et T). Admettons que le site k (i.e. colonne k) de la séquence i est inconnu (i.e. manquant). Pour calculer la distance entre la séquence i et toutes les autres séquences considérées, *PEMV* estime, à l'aide de l'équation 1, les probabilités P_{ik}(A), P_{ik}(C), P_{ik}(G) et P_{ik}(T) d'avoir respectivement le nucléotide A, C, G ou T au site k de la séquence i. La probabilité qu'une base manquante corresponde à un nucléotide spécifique dépend du nombre de séquences ayant le même nucléotide au site k, de même que de la distance entre i et toutes les autres séquences ayant des nucléotides connus au site k. On évalue en premier lieu le score de parité (i.e. la similarité) δ entre toutes les séquences (en ignorant les données manquantes). Ce score représente le rapport entre le nombre de paires de nucléotides distinctes et le nombre de sites comparables dans une paire de séquences. La probabilité P_{ik} se calcule comme suit :

$$P_{ik}(\mathrm{V}) = \frac{1}{N_k}\left(\sum_{\substack{j,\ \text{tels que} \\ C_{jk}=\mathrm{V}}} \delta_{ij} + \frac{1}{3} \sum_{\substack{j,\ \text{tels que} \\ C_{jk}\neq\mathrm{V}}} (1-\delta_{ij}) \right) \qquad (1)$$

où le caractère V remplace l'un des quatre nucléotides A, C, G ou T; N_k – est le nombre de valeurs existantes dans la colonne k; δ_{ij} – est le score de parité entre les séquences i et j; **C** – est la matrice des séquences des nucléotides,

Le théorème suivant caractérisant les probabilités P_{ik}(A), P_{ik}(C), P_{ik}(G) et P_{ik}(T) pour une séquence i et un site donné k peut être formulé (sa preuve n'est pas présentée ici):

Théorème. *Pour toute séquence i, tout site k de la matrice* **C**, *tels que* c_{ik} *est un nucléotide manquant, nous avons :* $P_{ik}(A) + P_{ik}(C) + P_{ik}(G) + P_{ik}(T) = 1$.

Une fois les différentes probabilités P_{ik} trouvées, nous calculons la matrice de distances **D** entre toutes les séquences en appliquant l'équation 2. Dans le modèle de Jukes-Cantor, la distance *PEMV* non corrigée entre les séquences i et j se calcule comme suit :

$$d_{ij} = \frac{N_{ij}^{c} - N_{ij}^{m} + \sum_{k=1}^{N-N_{ij}^{c}} (1 - P_{ij}^{k})}{N}, \qquad (2)$$

où d_{ij} – est la distance observée entre les séquences i et j ; N – est le nombre de sites considérés; N_{ij}^{m} - est le nombre de paires de nucléotides identiques dans i et j, N_{ij}^{c} - est le nombre de paires de nucléotides comparables (i.e. quand les deux nucléotides sont présents dans les sites correspondants) dans i et j ; P_{ij}^{k} - la probabilité, calculée en utilisant l'équation 1, d'avoir une paire de nucléotides identiques au site k dans i et j. La distance de Jukes-Cantor s'obtient à partir de d par l'application de la formule logarithmique.

Dans le cadre du modèle d'évolution de Kimura 2-paramètres nous calculons les taux de transition $P(i,j)$ et de transversion $Q(i,j)$ en utilisant l'équation 3 avant d'appliquer la formule logarithmique correspondante:

$$P(i,j) = \frac{P'(i,j) + \sum_{k=1}^{N-N_{ij}^{c}} P'(i,j,k)}{N}, \qquad Q(i,j) = \frac{Q'(i,j) + \sum_{k=1}^{N-N_{ij}^{c}} Q'(i,j,k)}{N}, \tag{3}$$

où $P'(i,j)$ – est le nombre de transitions entre les séquences i et j calculé en ignorant les sites incomplets; $P'(i,j,k)$ – est la probabilité de transition entre i et j au site k quand le nucléotide au site k est manquant soit dans i soit dans j; $Q'(i,j)$ – représente le nombre de transversions entre i et j calculé en ignorant les sites incomplets; $Q'(i,j,k)$ – est la probabilité d'obtenir une transversion entre i et j au site k lorsque ce nucléotide est manquant soit dans i soit dans j.

3 Application à des données réelles

La méthode introduite dans cet article a été utilisée pour reconstruire des arbres phylogénétiques inférés à partir des séquences de 10 espèces d'eucaryotes. Ces séquences de longueur 705 nucléotides chacune, ne contiennent pas de données manquantes et proviennent de l'ADN mitochondrial des espèces choisies. Elles sont disponibles sur la page web du Workshop on Molecular Biology à l'adresse URL suivante : <http://workshop.molecularevolution.org/resources/fileformats/fasta_dna_al.php>. La figure 1 présente deux arbres phylogénétiques possibles pour représenter l'évolution de ce groupe d'espèces et le pourcentage de bootstrap pour leurs branches internes. Le pourcentage de bootstrap permet d'estimer la robustesse de chacune des branches internes d'un arbre phylogénétique.

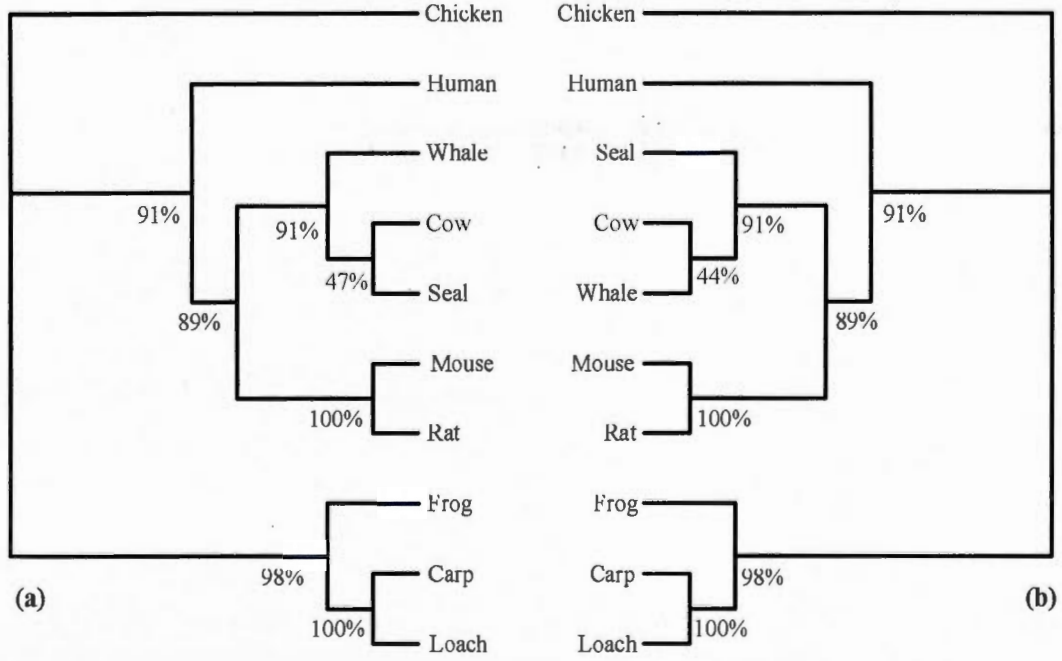


Figure 1. Deux arbres phylogénétiques inférés par la méthode NJ. Les séquences utilisées pour inférer ces arbres ne contiennent pas de données manquantes. Le modèle d'évolution de Kimura 2-paramètres a été utilisé pour calculer la matrice de distances entre les espèces.

Ces arbres sont légèrement différents au niveau du cluster regroupant les espèces Whale, Cow et Seal. Le premier arbre (figure 1a) soutient le groupe {Cow, Seal} avec le pourcentage de bootstrap de 47%, alors que le deuxième contient le cluster {Whale, Cow} dont le pourcentage de bootstrap est 44%.

Les deux topologies sont donc presque équiprobables. Donc, dans nos simulations nous les considérons comme les arbres initiaux corrects. Ces deux arbres ont été comparés aux phylogénies obtenues par les méthodes *IMS*, *PDMAB* et *PEMV* appliquées aux séquences d'ADN d'origine perturbées par l'ajout des données manquantes. Pour évaluer les performances des trois méthodes, de 0 à 80% de nucléotides ont été retirés des séquences initiales. Les nucléotides manquants ont été simulés par blocs pour mieux refléter la réalité génomique. Les séquences incomplètes obtenues, ont été soumises aux trois méthodes de calcul des matrices de distances d'évolution (*IMS*, *PDMAB* et *PEMV*) mentionnées ci-dessus. Pour chaque matrice de distance ainsi obtenue, nous avons reconstruit une phylogénie *T'* en utilisant la méthode NJ [SAI 87]. L'arbre phylogénique *T'* a été ensuite comparé à l'aide de la distance de Robinson et Foulds [ROB 81] aux deux arbres initiaux *T* présentés sur la figure 1 et la plus petite valeur (des deux valeurs possibles) de la distance de Robinson et Foulds a été retenue. Sur la figure 2, nous présentons les résultats moyens obtenus après 500 itérations (i.e. ensembles de séquences partielles différents). La figure 2a présente le pourcentage de la distance de Robinson et Foulds entre *T'* est le plus proche des deux arbres *T*. Deux arbres sont identiques si ce pourcentage est égal à 0 et leur différence s'accentue lorsqu'il augmente.

Les simulations fournissent des résultats similaires pour *IMS* et *PDMAB*. Pour tous les pourcentages de nucléotides manquants, *PEMV* assure une meilleure inférence phylogénétique comparativement aux méthodes de PAUP (figure 2a). Le nombre d'arbres identiques aux arbres initiaux était toujours plus élevé pour *PEMV* que pour *IMS* et *PDMAB* (figure 2b). Les performances de la nouvelle méthode sont les plus marquées pour 20, 30, 40 et 50% de valeurs manquantes.

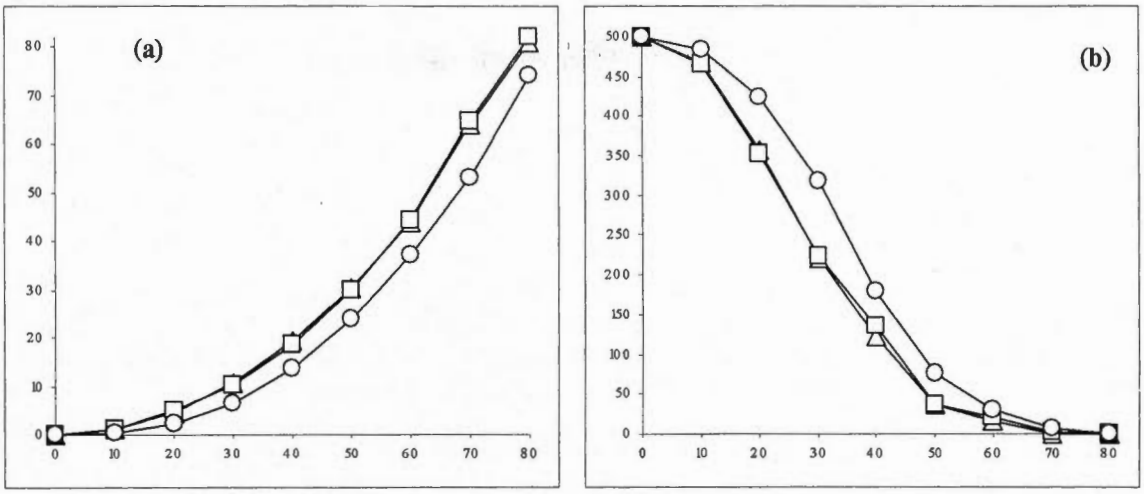


Figure 2. Les résultats moyens obtenus pour 500 itérations. Le pourcentage des données manquantes varie de 0 à 80% (axe des abscisses). Les courbes traduisent (a) les variations de la distance de Robinson et Foulds et (b) le nombre de fois que l'une des deux phylogénies initiales a été recouvrée. Les méthodes testées sont *IMS* (Δ), *PDMAB* (□) et *PEMV* (○).

4 Conclusion

Dans cet article nous avons décrit la méthode *PEMV* permettant d'estimer les probabilités des nucléotides manquants dans des séquences d'ADN et d'ARN en vue d'inférence d'un arbre phylogénétique. Dans nos simulations, le nombre de topologies identiques aux topologies initiales retrouvées par *PEMV* était supérieur à ceux des méthodes connues *PDMAB* et *IMS* de plus de 15, 30 et 25% pour, respectivement, 20, 30 et 40% de nucléotides manquants (figure 2b). Dans ces situations, l'élimination de sites manquants, comme le préconise la méthode *IMS*, ou leur estimation par la méthode *PDMAB* supprime des spécificités importantes des données traitées. La méthode *PEMV* a été incluse dans le logiciel T-Rex [MAK 01] disponible à l'URL suivant : <http://www.info.uqam.ca/~makarenv/trex.html>. Cette méthode a été

présentée ici dans le cadre des modèles de Jukes-Cantor [JUK 69] et de Kimura 2-parameter [KIM 80]. Il serait intéressant de continuer le développement de cette approche en la généralisant pour d'autres modèles d'évolution connus tels que : Tajima – Nei [TAJ 84], LogDet [STE 94] et d'autres. Il serait également nécessaire de comparer les performances de la stratégie *PEMV* à celles de maximum de vraisemblance et de maximum de parcimonie.

5 Bibliographie

[HUE 91] HUELSENBECK, J. P., "When are fossils better than existent taxa in phylogenetic analysis?" *Sys. Zool.*, vol. 40, 1991, p. 458-469.

[JUK 69] JUKES, T. H., CANTOR, C.. "Mammalian Protein Metabolism", *chapter Evolution of protein molecules*, 1969, p. 21-132. Academic Press, New York.

[KIM 80] KIMURA, M., "A simple method for estimating evolutionary rate of base substitutions through comparative studies of nucleotide sequences", *Jour. of Mol. Evol.*, vol. 16, 1980, p. 111-120.

[MAK 01] MAKARENKOV, V., "T-REX: reconstructing and visualizing phylogenetic trees and reticulation networks", *Bioinformatics*, vol. 17, 2001, p. 664-668.

[MAK 04] MAKARENKOV, V., LAPOINTE, F-J., "A weighted least-squares approach for inferring phylogenies from incomplete distance matrices", *Bioinformatics*, vol. 20, 2004, p. 2113-2121.

[ROB 81] ROBINSON D. R., FOULDS L. R. "Comparison of phylogenetic trees", *Mathematical Biosciences*, vol. 53, 1981, p. 131-147.

[SAI 87] SAITOU, N., M. NEI. "The neighbor-joining method: A new method for reconstructing phylogenetic trees", *Mol. Biol. Evol.* , vol. 4, 1987, p. 406-425.

[STE 94] STEEL, M. A. "Recovering a tree from the leaf colorations it generates under a Markov model", *Applied Math Letters*, vol. 72, 1994, p.19-24.

[SWO 01] SWOFFORD, D. L. "*PAUP**. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4.", *Sinauer Associates*, 2001, Sunderland, Massachusetts.

[TAJ 84] TAJIMA F, NEI M. "Estimation of evolutionary distance between nucleotide sequences." *Mol. Biol. Evol.*, vol. 3, 1984, p. 269-285.

3.2 New efficient algorithm for modeling partial and complete gene transfer scenarios

Dans cet article, publié aux actes du congrès biannuel de la Société Internationale de Classification (IFCS 2006, Ljubljana, Slovénie), nous décrivons une nouvelle méthode pour prédire et visualiser les possibles transferts horizontaux de gènes. La méthode peut utiliser soit une optimisation métrique soit une optimisation topologique pour estimer une possibilité d'avoir un transfert horizontal de gènes dans une phylogénie d'espèces. La classification d'espèces est examinée dans le contexte des modèles de transferts complet et partiel.

Makarenkov, V., Boc, A., Delwiche, C. F, Diallo, Al. B. and Philippe, H. (2006), New efficient algorithm for modeling partial and complete gene transfer scenarios, Data Science and Classification, V. Batagelj, H.-H. Bock, A. Ferligoj, and A. Ziberna (Eds.), IFCS 2006, Series: Studies in Classification, Data Analysis, and Knowledge Organization, Springer Verlag, 341- 349.

New Efficient Algorithm for Modeling Partial and Complete Gene Transfer Scenarios

Vladimir Makarenkov[1], Alix Boc[1], Charles F. Delwiche[2], Alpha Boubacar Diallo[1], and Hervé Philippe[3]

[1] Département d'informatique, Université du Québec à Montréal, C.P. 8888, Succ. Centre-Ville, Montréal (Québec), H3C 3P8, Canada,
[2] Cell Biology and Molecular Genetics, HJ Patterson Hall, Bldg. 073, University of Maryland at College Park, MD 20742-5815, USA.
[3] Département de biochimie, Faculté de Médecine, Université de Montréal, C.P. 6128, Succ. Centre-ville, Montréal, QC, H3C 3J7, Canada.

Abstract. In this article we describe a new method allowing one to predict and visualize possible horizontal gene transfer events. It relies either on a metric or topological optimization to estimate the probability of a horizontal gene transfer between any pair of edges in a species phylogeny. Species classification will be examined in the framework of the complete and partial gene transfer models.

1 Introduction

Species evolution has long been modeled using only phylogenetic trees, where each species has a unique most recent ancestor and other interspecies relationships, such as those caused by horizontal gene transfers (HGT) or hybridization, cannot be represented (Legendre and Makarenkov (2002)). HGT is a direct transfer of genetic material from one lineage to another. Bacteria and Archaea have sophisticated mechanisms for the acquisition of new genes through HGT, which may have been favored by natural selection as a more rapid mechanism of adaptation than the alteration of gene functions through numerous mutations (Doolittle (1999)). Several attempts to use network-based models to depict horizontal gene transfers can be found (see for example: Page (1994) or Charleston (1998)). Mirkin et al (1995) put forward a tree reconciliation method that combines different gene trees into a unique species phylogeny. Page and Charleston (1998) described a set of evolutionary rules that should be taken into account in HGT models. Tsirigos and Rigoutsos (2005) introduced a novel method for identifying horizontal transfers that relies on a gene's nucleotide composition and obviates the need for knowledge of codon boundaries. Lake and Rivera (2004) showed that the dynamic deletions and insertions of genes that occur during genome evolution, including those introduced by HGT, may be modeled using techniques similar to those used to model nucleotide substitutions (e.g. general Markov models). Moret et al (2004) presented an overview of the network modeling in phylogenetics. In this paper we continue the work started in Makarenkov

et al (2004), where we described an HGT detection algorithm based on the least-squares optimization. To design a detection algorithm which is mathematically and biologically sound we will consider two possible approaches allowing for complete and partial gene transfer scenarios.

2 Two different ways of transferring genes

Two HGT models are considered in this study. The first model, assumes partial gene transfer. In such a model, the original species phylogeny is transformed into a connected and directed network where a pair of species can be linked by several paths (Figure 1a). The second model assumes complete transfer; the species phylogenetic tree is gradually transformed into the gene tree by adding to it an HGT in each step. During this transformation, only tree structures are considered and modified (Figure 1b).

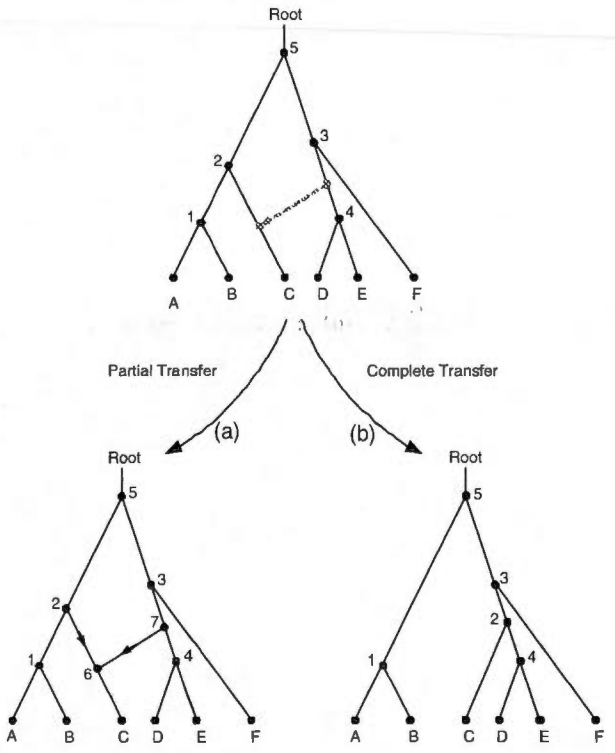


Fig. 1. Two evolutionary models, assuming that either a partial (a) or complete (b) HGT has taken place. In the first case, only a part of the gene is incorporated into the recipient genome and the tree is transformed into a directed network, whereas in the second, the entire donor gene is acquired by the host genome and the species tree is transformed into a different tree.

3 Complete gene transfer model

In this section we discuss the main features of the HGT detection algorithm in the framework of the complete gene transfer model. This model assumes that the entire transferred gene is acquired by the host (Figures 1b). If the homologous gene was present in the host genome, the transferred gene can supplant it. Two optimization criteria will be considered. The first of them is the least-squares (LS) function Q:

$$Q = \sum_i \sum_j (d(i,j) - \delta(i,j))^2, \tag{1}$$

where $d(i,j)$ is the pairwise distance between the leaves i and j in the species phylogenetic tree T and $\delta(i,j)$ the pairwise distance between i and j in the gene tree T_1. The second criterion that can be useful to assess the incongruence between the species and gene phylogenies is the Robinson and Foulds (RF) topological distance (1981). When the RF distance is considered, we can use it as an optimization criterion as follows: All possible transformations (Figure 1b) of the species tree, consisting of transferring one of its subtrees from one edge to another, are evaluated in a way that the RF distance between the transformed species tree T' and the gene tree T_1 is computed. The subtree transfer providing the minimum of the RF distance between T' and T_1 is retained as a solution. Note that the problem asking to find the minimum number of subtree transfer operations necessary to transform one tree into another has been shown to be NP-hard but approximable to within a factor of 3 (Hein et al (1996)).

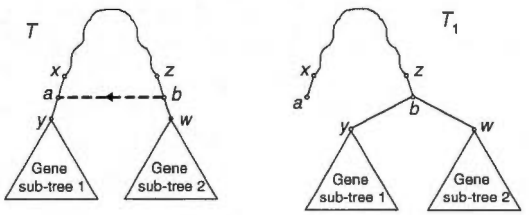


Fig. 2. Timing constraint: the transfer between the edges (z,w) and (x,y) of the species tree T can be allowed if and only if the cluster regrouping both affected subtrees is present in the gene tree T_1.

Several biological rules have to be considered in order to synchronize the way of evolution within a species phylogeny (Page and Charleston (1998)). For instance, transfers between the species of the same lineage must be prohibited. In addition, our algorithm relies on the following timing constraint: The cluster combining the subtrees rooted by the vertices y and w must be present in the gene tree T_1 in order to allow an HGT between the edges (z,w) and (x,y) of the species tree T (Figure 2). Such a constraint enables us,

first, to arrange the topological conflicts between T and T_1 that are due to the transfers between single species or their close ancestors and, second, to identify the transfers that have occurred deeper in the phylogeny. The main steps of the HGT detection algorithm are the following:

Step 0. This step consists of inferring the species and gene phylogenies denoted respectively T and T_1 and labeled according to the same set X of n taxa (e.g. species). Both species and gene trees should be explicitly rooted. If the topologies of T and T_1 are identical, we conclude that HGTs are not required to explain the data. If not, either the RF difference between them can be used as a phylogeny transformation index, or the gene tree T_1 can be mapped into the species tree T fitting by least-squares the edge lengths of T to the pairwise distances in T_1 (see Makarenkov and Leclerc (1999)).

Step 1. The goal of this step is to obtain an ordered list L of all possible gene transfer connections between pairs of edges in T. This list will comprise all different directed connections (i.e. HGTs) between pairs of edges in T except the connections between adjacent edges and those violating the evolutionary constraints. Each entry of L is associated with the value of the gain in fit, computed using either LS function or RF distance, found after the addition of the corresponding HGT connection. The computation of the ordered list L requires $O(n^4)$ operations for a phylogenetic tree with n leaves. The first entry of L is then added to the species tree T.

Steps 2 ... k. In the step k, a new tree topology is examined to determine the next transfer by computing the ordered list L of all possible HGTs. The procedure stops when the RF distance equals 0 or the LS coefficient stops decreasing (ideally dropping to 0). Such a procedure requires $O(kn^4)$ operations to add k HGT edges to a phylogenetic tree with n leaves.

4 Partial gene transfer model

The partial gene transfer model is more general, but also more complex and challenging. It presumes that only a part of the transferred gene has been acquired by the host genome through the process of homologous recombination. Mathematically, this means that the traditional species phylogenetic tree is transformed into a directed evolutionary network (Figure 1a). Figure 3 illustrates the case where the evolutionary distance between the taxa i and j may change after the addition of the edge (b,a) representing a partial gene transfer from b to a.

From a biological point of view, it is relevant to consider that the HGT from b to a can affect the distance between the taxa i and j if and only if a is located on the path between i and the root of the tree; the position of j is assumed to be fixed. Thus, in the network T (Figure 3) the evolutionary distance $dist(i,j)$ between the taxa i and j can be computed as follows:

$$dist(i,j) = (1-\mu)d(i,j) + \mu(d(i,a) + d(j,b)), \qquad (2)$$

where μ indicates the fraction (unknown in advance) of the gene being transferred and d is the distance between the vertices in T before the addition of the HGT edge (b,a). A number of biological rules, not discussed here due to the space limitation, have to be incorporated into this model (see Makarenkov et al (2004) for more details). Here we describe the main features of the network-building algorithm:

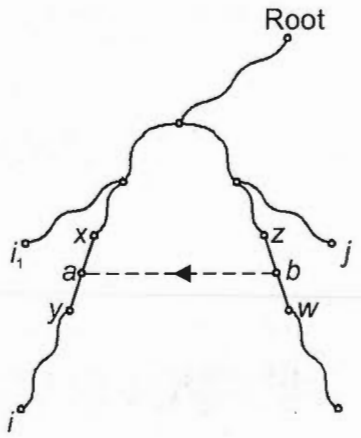


Fig. 3. Evolutionary distance between the taxa i and j can be affected by the addition of the edge (b,a) representing a partial HGT between the edges (z,w) and (x,y). Evolutionary distance between the taxa i_1 and j cannot be affected by the addition of (b,a).

Step 0. This step corresponds to Step 0 defined for the complete gene transfer model. It consists of inferring the species and gene phylogenies denoted respectively T and T_1. Because the classical RF distance is defined only for tree topologies, we use the LS optimization when modeling partial HGT.

Step 1. Assume that a partial HGT between the edges (z,w) and (x,y) (Figure 3) of the species tree T has taken place. The lengths of all edges in T should be reassessed after the addition of (b,a), whereas the length of (b,a) is assumed to be 0. To reassess the edge lengths of T, we have first to make an assumption about the value of the parameter μ (Equation 2) indicating the gene fraction being transferred. This parameter can be estimated either by comparing sequence data corresponding to the subtrees rooted by the vertices y and w or by testing different values of μ in the optimization problem. Fixing this parameter, we reduce to a linear system the system of equations establishing the correspondence between the experimental gene distances and the path-length distances in the HGT network. This system having generally more variables (i.e. edge lengths of T) than equations (i.e. pairwise distances in T; number of equations is always $n(n\text{-}1)/2$ for n taxa)

can be solved by approximation in the least-squares sense. All pairs of edges in T can be processed in this way. The HGT connection providing the smallest value of the LS coefficient and satisfying the evolutionary constraints will be selected for the addition to the tree T' transforming it into a phylogenetic network.

Steps 2 ... k. In the same way, the best second, third and other HGT edges can be added to T, improving in each step the LS fit of the gene distance. The whole procedure requires $O(kn^5)$ operations to build a reticulated network with k HGT edges starting from a species phylogenetic tree with n leaves.

5 Detecting horizontal transfers of PheRS synthetase

In this section, we examine the evolution of the PheRS protein sequences for 32 species including 24 Bacteria, 6 Archaea, and 2 Eukarya (see Woese et al (2000)). The PheRS phylogenetic tree inferred with PHYML (Guindon and Gascuel (2003)) using G-law correction is shown in Figure 4.

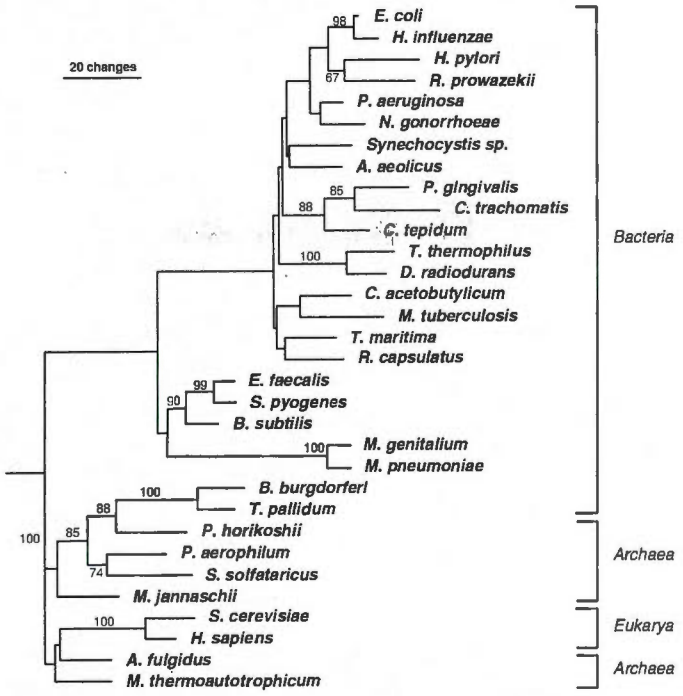


Fig. 4. Phylogenetic tree of PheRS sequences (i.e. gene tree). Protein sequences with 171 bases were considered. Bootstrap scores above 60% are indicated.

This tree is slightly different from the phylogeny obtained by Woese et al (2000, Fig. 2); the biggest difference involves the presence of a new clus-

ter formed by two Eukarya (*H. sapiens* and *S. cerevisiae*) and two Archaea (*A. fulgidus* and *M. thermoautotrophicum*). This 4-species cluster with a low bootstrap support is probably due to the reconstruction artifacts. Otherwise, this tree shows the canonical pattern, the only exception being the spirochete PheRSs (i.e. *B. bugdorferi* and *T. pallidum*). They are of the archaeal, not the bacterial genre, but seem to be specifically related to *P. horokoshii* within that grouping (Figure 4). The species tree corresponding to the NCBI taxonomic classification was also inferred (Figure 5, undirected lines). The computation of HGTs was done in the framework of the complete gene transfer model. The five transfers with the biggest bootstrap scores are represented.

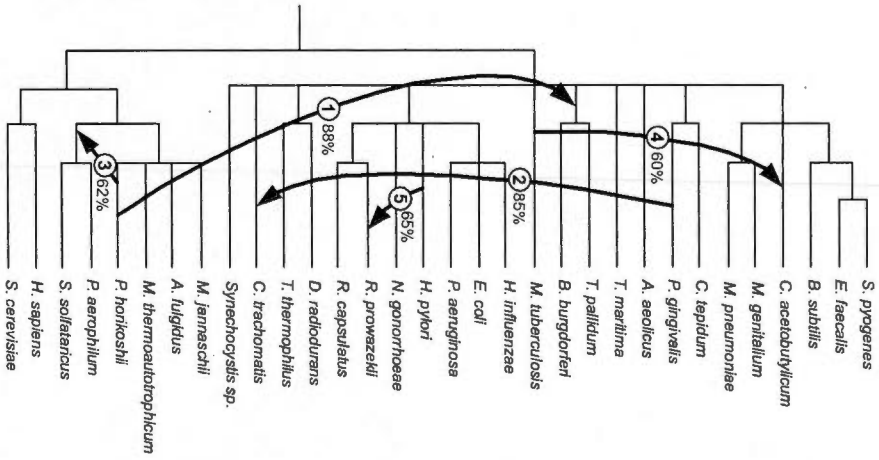


Fig. 5. Species phylogeny corresponding to the NCBI taxonomy for the 32 species in Figure 4. HGTs with bootstrap scores above 60% are depicted by arrows. Numbers on the HGT edges indicate their order of appearance in the transfer scenario.

The bootstrap scores for HGT edges were found fixing the topology of the species tree and resampling the PheRS sequences used to obtain the gene tree. The transfer number 1, having the biggest bootstrap support, 88%, links *P. horokoshii* to the clade of spirochetes. This bootstrap score is the biggest one that could be obtained for this HGT, taking into account the identical 88% score of the corresponding 3-species cluster in the PheRS phylogeny (Figure 4). In total, 14 HGTs, including 5 trivial connections, were found; trivial transfers occur between the adjacent edges. Trivial HGTs are necessary to transform a non-binary tree into a binary one. The non-trivial HGTs with low bootstrap score are most probably due to the tree reconstruction artifacts. For instance, two HGT connections (not shown in Figure 5) linking the cluster of Eukarya to the Archaea (*A. fulgidus* and *M. thermoautotrophicum*) have a low bootstrap support (16% and 32%, respectively). In this example, the solution found with the RF distance was represented. The usage of the LS

function leads to the identical scenario differing from that shown in Figure 5 only by the bootstrap scores found for the HGT edges 3 to 5.

6 Conclusion

We described a new distance-based algorithm for the detection and visualization of HGT events. It exploits the discrepancies between the species and gene phylogenies either to map the gene tree into the species tree by least-squares or to compute a topological distance between them and then estimate the probability of HGT between each pair of edges of the species phylogeny. In this study we considered the complete and partial gene transfer models, implying at each step either the transformation of a species phylogeny into another tree or its transformation into a network structure. The examples of the evolution of the PheRS synthetase considered in the application section showed that the new algorithm can be useful for predicting HGT in real data. In the future, it would be interesting to extend and test this procedure in the framework of the maximum likelihood and maximum parsimony models. The program implementing the new algorithm was included to the T-Rex package (Makarenkov (2001), http://www.trex.uqam.ca).

References

CHARLESTON, M. A. (1998): Jungle: a new solution to the host/parasite phylogeny reconciliation problem. *Math. Bioscience, 149, 191-223.*

DOOLITTLE, W. F. (1999): Phylogenetic classification and the universal tree. *Science, 284, 2124-2129.*

GUINDON, S. and GASCUEL, O. (2003): A simple, fast and accurate algorithm to estimate large phylogenies by maximum likelihood. *Syst. Biol., 52, 696-704.*

LAKE, J. A. and RIVERA, M. C. (2004): Deriving the genomic tree of life in the presence of horizontal gene transfer: conditioned reconstruction. *Mol. Biol. Evol., 21, 681-690.*

LEGENDRE, P. and V. MAKARENKOV. (2002): Reconstruction of biogeographic and evolutionary networks using reticulograms. *Syst. Biol., 51, 199-216.*

MAKARENKOV, V. and LECLERC, B. (1999): An algorithm for the fitting of a tree metric according to a weighted LS criterion. *J. of Classif., 16, 3-26.*

MAKARENKOV, V. (2001): reconstructing and visualizing phylogenetic trees and reticulation networks. *Bioinformatics, 17, 664-668.*

MAKARENKOV, V., BOC, A. and DIALLO, A. B. (2004): Representing lateral gene transfer in species classification. Unique scenario. In: D. Banks, L. House, F. R. McMorris, P. Arabie, and W. Gaul (eds.): *Classification, Clustering and Data Mining Applications.* Springer Verlag, proc. IFCS 2004, Chicago 439-446

MIRKIN, B. G., MUCHNIK, I. and SMITH, T.F. (1995): A Biologically Consistent Model for Comparing Molecular Phylogenies. *J. of Comp. Biol., 2, 493-507.*

MORET, B., NAKHLEH, L., WARNOW, T., LINDER, C., THOLSE, A., PADOLINA, A., SUN, J. and TIMME, R. (2004): Phylogenetic Networks: Modeling, Reconstructibility, Accuracy. *Trans. Comp. Biol. Bioinf., 1, 13-23.*

PAGE, R. D. M. (1994): Maps between trees and cladistic analysis of historical associations among genes, organism and areas. *Syst. Biol., 43, 58-77.*

PAGE, R. D. M. and CHARLESTON, M. A. (1998): Trees within trees: phylogeny and historical associations. *Trends Ecol. Evol., 13, 356-359.*

ROBINSON, D. R. and FOULDS, L. R. (1981): Comparison of phylogenetic trees. *Math. Biosciences, 53, 131-147.*

TSIRIGOS, A. and RIGOUTSOS, I. (2005): A Sensitive, Support-Vector-Machine Method for the Detection of Horizontal Gene Transfers in Viral, Archaeal and Bacterial Genomes. *Nucl. Acids Res., 33, 3699-3707.*

WOESE, C., OLSEN, G., IBBA, M. and SÖLL, D. (2000): Aminoacyl-tRNA synthetases, genetic code, evolut. process. *Micr. Mol. Biol. Rev., 64, 202-236.*

3.3 Algorithms for detecting complete and partial horizontal gene transfers: theory and practice

L'évolution réticulée est un processus complexe, incluant les mécanismes de transfert horizontal de gènes (i.e., transfert latéral de gènes), de recombinaison homologue et d'hybridation. Dans cet article, soumis au Centre de Recherche en Mathématiques de Montréal (CRM) pour le livre édité à la suite du colloque international "Data Mining and Mathematical Programming", nous présentons une nouvelle méthode permettant de détecter des transferts horizontaux de gènes d'un groupe d'espèces données. Nous faisons une présentation détaillée des modèles de transferts horizontaux complet et partiel. Un algorithme glouton est proposé pour inférer des transferts horizontaux. Une procédure de bootstrap (i.e., validation statistique des transferts inférés) a été conçue et présentée à l'aide d'un exemple de données réelles (évolution du gène rpl12e dans une phylogénie des archéobactéries). Cette procédure peut être appliquée pour estimer la robustesse d'un transfert horizontal particulier ou d'un scenario de transferts au complet. Les simulations Monte Carlo présentent les résultats de l'algorithme dans le cadre du transfert horizontal complet en considérant trois différents modèles d'évolution.

Makarenkov, V., Boc, A., Diallo, Al. B., Diallo, Ab. B. (2006). Algorithms for detecting complete and partial horizontal gene transfers: theory and practice À paraitre dans les Actes de Workshop : Exploration de données et programmation mathématique. Centre de recherche en mathématiques de Montréal. Octobre 2006.

Algorithms for detecting complete and partial horizontal gene transfers: theory and practice

Vladimir Makarenkov[1], Alix Boc[1], Alpha Boubacar Diallo[1] and Abdoulaye Baniré Diallo[2]

[1] Département d'informatique, Université du Québec à Montréal, C.P. 8888, Succ. Centre-Ville, Montréal (Québec), Canada, H3C 3P8.

[2] McGill Centre for Bioinformatics and School of Computer Science, McGill University, 3775 University Street, Montréal, Québec, H3A 2B4, Canada.

e-mails: makarenkov.vladimir@uqam.ca, boc.alix@courrier.uqam.ca, diallo.alpha_boubacar@courrier.uqam.ca and banire@mcb.mcgill.ca

Corresponding author: Vladimir Makarenkov (makarenkov.vladimir@uqam.ca).

Abstract. We describe two methods for detecting horizontal gene transfers in the framework of the complete and partial gene transfer models. In case of a complete gene transfer model a new fast backward selection algorithm for predicting horizontal gene transfer events is presented. The latter algorithm can rely either on the metric or on the topological optimization to identify horizontal gene transfers between branches of a given species phylogeny. In case of the topological optimization, we use the well-known Robison and Foulds (RF) topological distance, whereas in case of the metric optimization, the least-squares (LS) criterion is considered. We also formulate and prove the NP-hardness of the partial gene transfer problem. Second, an efficient algorithm for predicting partial transfers, using the Gauss and Seidel optimization, is discussed. We also show how to assess the reliability of a specific gene transfer or a whole gene transfer scenario. In the application section, we apply the new algorithm to detect possible gene transfers occurred during the evolution of the gene *rpl12e*.

Submitted to the proceedings of the CRM 2006 workshop "Data Mining and Mathematical Programming".

INTRODUCTION

Horizontal gene transfer (HGT) is a direct transfer of genetic material from one lineage to another. The understanding that horizontal gene transfer might have played a key role in biological evolution is one of the most fundamental changes in our perception of general aspects of molecular biology in recent years (Doolittle 1999, Legendre 2000, Legendre and Makarenkov 2002). Bacteria and Archaea have sophisticated mechanisms for the acquisition of new genes through HGT which may have been favored by natural selection as a more rapid mechanism of adaptation than the alteration of gene functions through numerous point mutations. If the donor DNA and the recipient chromosome display some homologous sequences, the donor sequences can be stably incorporated into the recipient chromosome by homologous recombination. The three main mechanisms of HGT are the following: transformation, consisting of uptake of naked DNA from the environment; conjugation, which is mediated by conjugal plasmids or conjugal transposons; and transduction, consisting of DNA transfer by phage. These transferring mechanisms can introduce sequences of DNA that display little similarity with the remaining DNA of the recipient cell (Doolittle 1999).

There are a few ways to identify the genes that have been transferred horizontally. First, sequence analysis of the host genome may reveal areas with GC content or codon usage patterns atypical to it (Lawrence and Ochman 1997). Second, if a sequence is found in only one organism and is absent from all other closely related organisms, it is more likely that it has been introduced horizontally into this organism rather than deleted from all the others. Third, the comparison of a morphology-based species tree or a molecular tree based on a molecule that is assumed to be refractory to horizontal gene transfer (e.g. 16S rRNA or 23S rRNA) against a phylogeny of an observed gene may reveal topological conflicts which can be explained by horizontal transfers.

Several attempts to use network-based models to depict horizontal gene transfers can be found (see for example: von Haeseler and Churchill 1993, Page 1994, Charleston 1998, Hallett and Lagergren 2001, or Hallett et al. 2004). A model of horizontal gene transfer that maps gene phylogenies into a species tree has been introduced by Hallett and Lagergren 2001. Mirkin et al. (2003) and Hallett et al. (2004) have developed algorithms allowing for simultaneous identification of gene duplications, gene losses, and horizontal gene transfers. The papers by Moret et al. (2004) and Nakhleh et al. (2005) give an overview of the network modeling in phylogenetics. In a recent paper published in the SFC2004 proceedings, Mirkin (2004) considered some approaches for biologically meaningful mapping of data of individual gene families into an evolutionary species tree. One approach first produces a gene tree, then maps it into the species tree, whereas the other approach first takes the gene phyletic profile, maps it into the species tree and then tunes it into a directed scenario based on the similarity data.

In this article we continue the work started in Boc and Makarenkov (2003), where we described a HGT model based on least-squares, and in Makarenkov et al. (2006), where we showed the difference between complete and partial gene transfer models. First, we describe a polynomial-time HGT algorithm for the detection of complete transfers and test it with respect to the two optimization criteria: Least-squares (LS), and Robinson and Foulds (RF) topological distance. We also suggest how to assess the reliability of

horizontal gene transfers identified by our algorithm. In the application section, we show how the new algorithm predicts transfers of the gene *rpl12e* for the group of 14 Archaea organisms which were originally examined in Matte-Taillez et al. (2002).

ALGORITHMS FOR PREDICTING HORIZONTAL GENE TRANSFERS

2.1 Basic definitions

We start this section with some basic definitions about phylogenetic trees and tree metrics, generally following the terminology of Barthélemy and Guénoche (1988, 1991). The *distance* $\delta(x\ y)$ between two vertices x and y in a phylogenetic (i.e. additive) tree T is defined as the sum of the edge lengths in the unique path linking x and y in T. Such a path is denoted (x,y). A *leaf* is a vertex of degree one.

Definition 1
Let X be a finite set of n taxa. A *dissimilarity* d on X is a non-negative function on $(X\text{x}X)$ such that for any x, y from X:

(1) $d(x,y) = d(y,x)$, and

(2) $d(x,y) = d(y,x) \geq d(x,x) = 0$.

Definition 2
A dissimilarity d on X satisfies the *four-point condition* if for any x, y, z, and w from X:

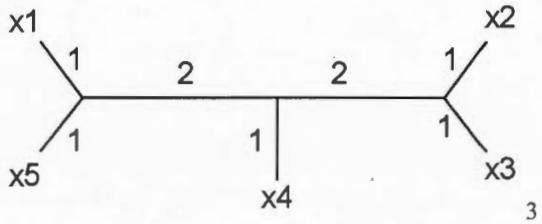
$$d(x,y) + d(z,w) \leq \text{Max}\ \{\ d(x,z) + d(y,w);\ d(x,w) + d(y,z)\ \}.$$

Definition 3
For a finite set X, a **phylogenetic tree** (i.e. an additive tree or an X-tree) is an ordered pair (T, φ) consisting of a tree T, with vertex set V, and a map $\varphi: X \rightarrow V$ with the property that, for all $x \in X$ with degree at most two, $x \in \varphi(X)$. A phylogenetic tree is **binary** if φ is a bijection from X into the leaf set of T and every interior vertex has degree three.

The main theorem relating the four-point condition and dissimilarity representability by a phylogenetic tree (i.e., phylogeny) is as follows:

Theorem 1 (*Zarestskii, Buneman, Patrinos & Hakimi, Dobson*)
Any dissimilarity satisfying the four-point condition can be represented by a *phylogenetic tree* such that for any x, y from X, $d(x,y)$ is equal to the length of the path linking the leaves x and y in T. This dissimilarity is called a *tree metric*. Furthermore, this tree is unique.
Here is an example of an example of a tree metric on the set X of 5 taxa and the associated phylogenetic tree.

	x2	x3	x4	x5
x1	6	6	4	2
x2		2	4	6
x3			4	6
x4				4



2.2 Optimization criteria

Here we present a fast greedy algorithm for predicting complete horizontal gene transfers. The algorithm for identifying HGTs proceeds by a progressive reconciliation of the given species and gene phylogenetic trees, denoted T and T' respectively. Usually, the species tree T is inferred from the genes that are refractory to horizontal gene transfer and genetic recombination (e.g., 16sRNA sequences). This tree represents the direct or tree-like evolution. The gene tree T' represents the evolution of a given gene which is supposed to undergo horizontal transfers.

At each step of the algorithm, all pairs of branches in T are tested against the hypothesis that a horizontal gene transfer has occurred between them. The considered HGT model assumes that the transferred gene supplants the entire homologous gene of the host or that the homologous gene is simply absent at the host genome. In such a model, the original species phylogenetic tree T is gradually transformed into the gene phylogenetic tree T' through a series of subtree moves (i.e., gene transfers or HGTs). The topology of the gene tree T' is kept fixed. The goal is to find the minimum possible sequence of trees T, T_1, T_2, ..., T' that transforms T into T'. Obviously, a number of necessary biological rules should be taken into account. For instance, the transfers within the same lineage as well as some double-crossing transfers should be prohibited (for more detail, see Maddison 1997, Page and Charleston 1998, or Hallett and Lagergren 2001).

We consider two optimization criteria which can be used at each algorithmic step to select the best HGT. The first optimization criterion that we consider is the *least-squares (LS) function Q*. It is computed as follows:

$$Q = \sum_i \sum_j (d(i,j) - \delta(i,j))^2 \ , \qquad (1)$$

where $d(i,j)$ is the pairwise distance between the leaves i and j in the species tree T (or in the tree T_1 obtained from T after the first subtree move) and $\delta(i,j)$ the pairwise distance between i and j in the gene tree T'. The second criterion that can be useful for assessing discrepancy between the species and gene phylogenies is the *Robinson and Foulds (RF) topological distance*. The RF metric (Robinson and Foulds 1981) is an important and frequently used tool to compare the topologies of phylogenetic trees. This distance is equal to the minimum number of elementary operations, consisting of merging and splitting nodes, necessary to transform one tree into the other. This distance is also the number of bipartitions or Buneman's splits belonging to exactly one of the two trees. When the RF distance is considered, we can use it as an optimization criterion as follows: all possible transformations of the species tree, consisting of transferring one of its subtrees from one branch to another, are evaluated in a way that the RF distance between the transformed species tree T_1 and the gene tree T' is computed. The subtree transfer providing the minimum of the RF distance between T_1 and T' is retained. Note that the problem asking to find the minimum number of subtree transfer operations necessary to transform one tree into another (i.e. also known as *Subtree Transfer Problem*) has been *shown to be NP-hard* (Hein et al. 1996).

2.3 Greedy backward algorithm for predicting complete horizontal gene transfers

In this section we discuss the main features of our algorithm based on the backward selection of horizontal gene transfers. Consider a gene transfer in the species tree T going from b to a and transforming it into the tree T_1 (Figure 1). The following timing constraint is considered (see also Makarenkov et al. 2006): to allow the transfer between the branches (z,w) and (x,y) of the species tree T, the cluster combining the subtrees rooted by the vertices y and w must be present in the gene tree T'. Such a constraint enables us, first, to arrange the topological conflicts between T and T' that are due to the transfers between single species or their close ancestors and, second, to identify the transfers that have occurred deeper in the phylogeny (i.e., closer to the tree root). The usage of this constraint allows the method to follow the order that is opposite to the order of evolution and infer first the most recent HGTs which are easier to detect.

Proposition 1. *If all bipartitions corresponding to the branches of the path (x,z) in the transformed species tree T_1 (Figure 1) can be found in the bipartition table of the gene tree T', then the transfer from b to a, transforming the species tree T into T_1, is a part of any minimum cost HGT scenario transforming T into T'.*

This Proposition can be easily proved by induction on the number of branches of the path (x,z).

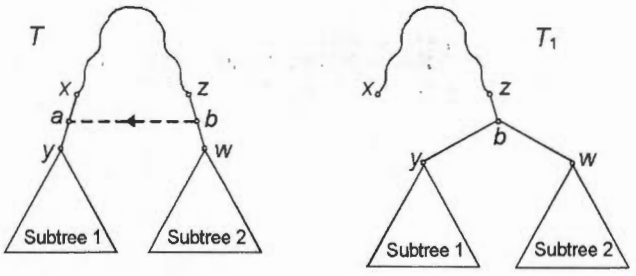


Figure 1. Subtree constraint: the transfer between the branches (z,w) and (x,y) of the species tree T can be allowed if and only if the cluster regrouping both affected subtrees is present in the gene tree; here, a single branch is depicted by a plane line and a path is depicted by a wavy line.

The main steps of the HGT detection algorithm are the following:

Preliminary Step

Infer species and gene phylogenies, denoted respectively T and T', whose leaves are labeled by the same set of n species. Both species and gene trees must be rooted. If there exist identical subtrees with two or more leaves belonging to both T and T', reduce the size of the problem by replacing these subtrees with the same auxiliary taxa in both T and T'.

Steps 1 ... *k*

Test all possible HGTs between pairs of branches in T_{k-1} ($T_{k-1} = T$ at Step 1) except the transfers between adjacent branches and those violating the evolutionary and subtree constraints. If no such a transfer exists, relax the subtree constraint. In our simulations described in the section Simulation study, this relaxation was necessary on average in 1.2% of cases. Search for the transfers satisfying the conditions of Proposition 1. If no such

transfers exist, choose the best HGT with respect to the selected optimization criterion that can be in our case: the least-squares (LS) or the Robinson and Foulds (RF) metric. Reduce the size of the problem by contracting the newly-formed subtree in the transformed species tree T_k and the gene tree T'. In the list of the obtained HGTs, search for and eliminate the idle transfers using a backward procedure. An idle transfer is the transfer whose removal does not change the topology of the tree T_k.

Stopping condition and time complexity

The procedure stops when the LS or RF coefficient drops to zero. Such a computation requires $O(kn^4)$ time to generate k transfers in a phylogenetic tree with n leaves. However, because of the progressive size reduction of the species and gene trees, the practical time complexity of this algorithm is rather $O(kn^3)$.

Proposition 2. *If the subtree constraint is not relaxed, the HGT detection algorithm requires at most $n-3$ steps to transform a binary species tree with n leaves into a binary gene tree with the same set of n leaves.*

The proof of this Proposition is based on the fact that the maximum value of the RF distance between two binary trees with n leaves is $2n-6$ and that each subtree transfer satisfying the subtree constraint decreases the value of the RF distance by at least 2.

2.4. Partial gene transfer model

The partial gene transfer model is more general, but also more complex and challenging. It presumes that only a part of the transferred gene has been acquired by the host species through the process of homologous recombination (Makarenkov et al. 2006). This means that the traditional species phylogenetic tree is transformed into a directed phylogenetic network (i.e. a directed connected graph). For example, Denamur et al. (2000) proposed a method to identify gene segments being transferred horizontally. This method was applied to detect partial HGTs of the *mutU* and *mutS* genes within *E. coli* evolutionary trees. Because many analyses are now directed at understanding the evolution of complete genomes, the partial gene transfer model could be also useful if one wanted to model the transfer of a portion of a genome.

In a phylogenetic tree, there is always a unique path connecting a pair of nodes. Adding to it an HGT branch creates an extra path between certain nodes. Figure 2 illustrates the case where the evolutionary distance between the taxa i and j can be affected by the addition of the HGT branch (b,a) representing partial gene transfer from b to a. It is relevant to assume that the HGT from b to a can affect the evolutionary distance between the taxa i and j if and only if the destination point a is located on the path between i and the root of the tree; the position of j is fixed. Thus, in the reticulate phylogeny T in Figure 2 the evolutionary distance $d_1(i,j)$ between the taxa i and j can be computed as follows:

$$d_1(i,j) = (1 - \alpha)\, d(i,j) + \alpha\,(d(i,a) + d(j,b)), \qquad (2)$$

where α indicates the fraction, unknown in advance, of the transferred gene and d is the internode distance in the species tree before the addition of the HGT branch (b,a).

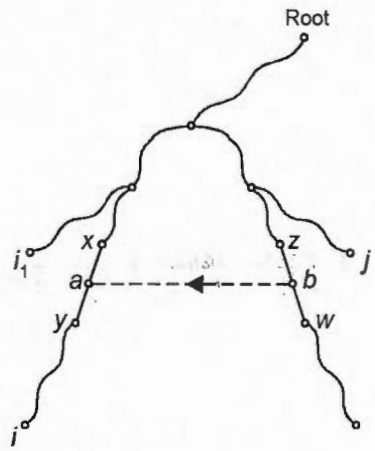


Figure 2. Evolutionary distance between the taxa i and j can be allowed to change after the addition of the branch (b,a) representing a partial HGT between the branches (z,w) and (x,y). Evolutionary distance between the taxa i_1 and j must not be affected by the addition of (b,a).

On the contrary, the distance between the taxa i_1 and j (Figure 2) must not be affected by the addition of (b,a). Figure 3 illustrates the other cases where the addition of a HGT branch must not affect the length of the evolutionary path between i and j.

The least-squares loss function Q to be minimized with the *unknown vector of edge lengths* $\boldsymbol{l}$ in T and the *unknown fraction of the transferred gene* α is as follows:

$$Q(L,\alpha) = \sum_{ij\in S}((1-\alpha)\sum_{k\in path(ij)} l_{ij}^k + \alpha(\sum_{k\in path(ia)} l_{ia}^k + \sum_{k\in path(jb)} l_{jb}^k) - \delta(i,j))^2 + \sum_{ij\notin S}(\sum_{k\in path(ij)} l_{ij}^k - \delta(i,j))^2 \rightarrow min, \quad (3)$$

where $\delta(i,j)$ is the given gene dissimilarity between i and j; l_{ij}^k is the length of the branch k of the path (ij) in T; α is the fraction of the transferred gene $(0 \le \alpha \le 1)$; and S is the set of pairs of taxa $\{ij\}$ such that the transfer (ba) can affect the evolutionary distance between them.

To show the NP-hardness of the least-squares optimization in the context of the partial gene transfer the following problem can be stated:

Given: *Species phylogenetic tree T (with the associated tree metric d on the set of taxa X), gene dissimilarity δ on X, and a fixed non-negative value ε*

Problem: *Find the minimum number of partial gene transfers k such that:*

$$Q = \sum_i \sum_j (d_k(i,j) - \delta(i,j))^2 \le \varepsilon, \quad (4)$$

where $d_k(i,j)$ is the network distance between i and j, computed using Formulae 2 and 3, in the phylogenetic network T_k obtained from T after the addition of k partial gene transfers.

Theorem 2. *The minimum number of partial gene transfer problem (MNPGT problem) is NP-hard.*

The proof of this Theorem is based on a polynomial-time reduction from the *Subtree Transfer Problem* (STR problem) that consists of finding the minimum number of complete gene transfers to transform a given species tree T into a given gene tree T'. The STR problem is identical to the problem of adding to T the minimum number of complete gene transfers such that $Q = \sum_i \sum_j (d_k(i,j) - \delta(i,j))^2 \leq 0$ (i.e., the case of $\varepsilon = 0$ is considered), where $d_k(i,j)$ is the pairwise distance between i and j in the phylogenetic tree (i.e., a particular case of a phylogenetic network). Here, the tree T_k is obtained from T after the addition of k complete gene transfers (i.e., a particular case of a partial transfer) and $\delta(i,j)$ is the given tree metric associated with T'.

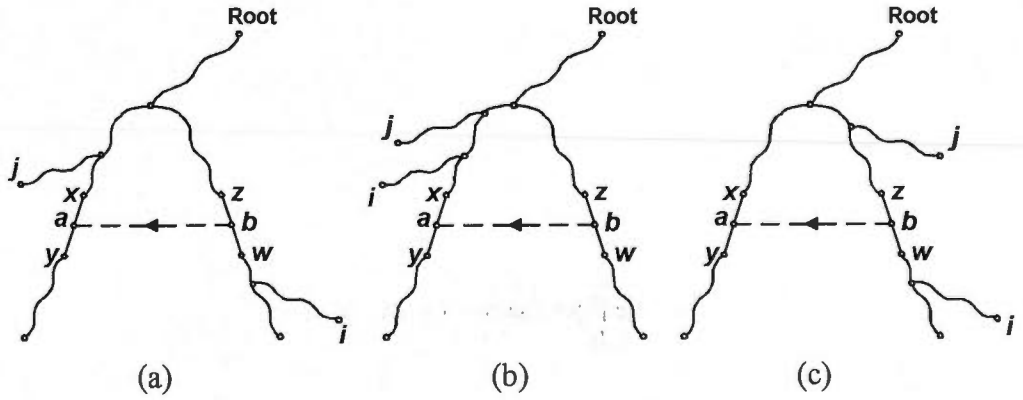


Figure 3 (a-c). Three situations when the evolutionary distance between the taxa i and j must not be affected by the addition of the new branch (b,a) representing a partial HGT between the branches (z,w) and (x,y). Path between the taxa i and j cannot to go through the branch (b,a).

Several important timing constraints have to be incorporated into this model, in addition to those taken into account in the complete HTS model, to identify the interactions between HGTs that are not intelligible from an evolutionary point of view. Some of these constraints, but not all of them, were initially pointed out by Page and Charleston (1998a and b). For instance, double-crossing transfers between two lineages (Figures 4a and b) must be forbidden. In this case, the HGT events affect the ancestor of the species from the previous transfer. Making the source and destination lineages contemporaneous for one HGT makes the other transfer impossible (Figure 4).

Note that the rule illustrated in Figure 4a is automatically taken into account in the complete gene transfer model, where its violation would be equivalent to the violation of the same lineage constraint (see Page and Charleston 1998). For instance (Figure 4a), the HGT from (z,w) to (x,y) cannot be followed by the transfer from (z_1,w_1) to (x_1,y_1) because after the first HGT the branches (z_1,w_1) and (x_1,y_1) will be located on the same lineage (Lineage 2). We also identify two cases, where the evolutionary distance between the taxa i and j can be affected by multiple transfers (Figures 5a and b); and, two cases, where this distance must not be affected by them (Figures 5c and d). Failure to

take these constraints into account can result in postulating transfers that are mutually incompatible.

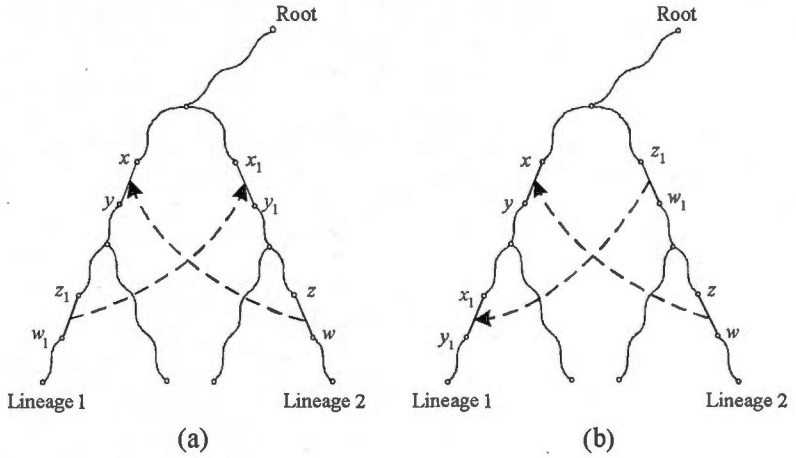


Figure 4. Transfers between two lineages crossing in such ways must be prohibited.

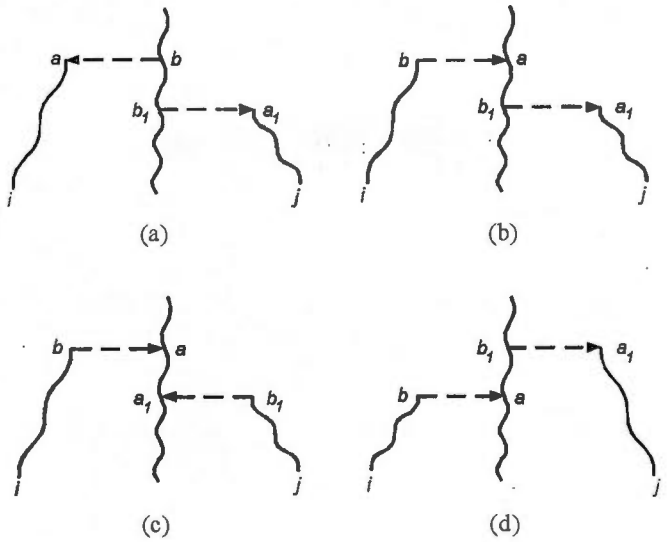


Figure 5. Cases (a and b): evolutionary path between the taxa i and j can go through both HGT branches (b,a) and (b_1,a_1). Cases (c) and (d): evolutionary path between the taxa i and j cannot go through both HGT branches (b,a) and (b_1,a_1).

Assume that a partial gene transfer between the branches (z,w) and (x,y) (i.e., from b to a in Figure 2) of the species tree T has taken place. The lengths of all branches in T are reassessed in the least-squares after the addition of (b,a), whereas the length of (b,a) is assumed to be 0. To reassess the branch lengths of T, we have first to make an assumption about the value of the parameter α (Equation 2), indicating the gene fraction being transferred. This parameter can be estimated either by comparing sequence data corresponding to the subtrees rooted by the vertices y and w, or different values of α can be tested in the optimization problem.

Fixing the parameter α, we reduce to a linear system the system of equations establishing the correspondence between the experimental gene distances and the path-length distances in the HGT network. This system having generally more variables (i.e. branch lengths of T) than equations (i.e. pairwise distances in T; the number of equations is always $n(n\text{-}1)/2$ for n taxa) can be solved by approximation in the least-squares sense. Let us now show how the approximation problem can be stated and efficiently solved.

Let $\mathbf{A}_\alpha$ be the matrix of dimension $n(n\text{-}1)/2 \times m$, each row of which is associated with one pair of taxa of X, where n is the number of taxa and m is a number of edges in T. The value $a_{ij,e}$ of this matrix corresponding to the pair of taxa ij and the edge e is equal either to 1, or to α, or to $1\text{-}\alpha$ if the edge e is in the path (ij) in T, and is equal to 0 if not. Let ℓ be the vector of edge lengths of m elements and $\mathbf{d}$ be given vector of gene distances of $n(n\text{-}1)/2$ elements.

Fixing the value of α (e.g., values 0, 0.1, 0.2, ..., and 1.0 can be tested in turn), we obtain a linear system of $n(n\text{-}1)/2$ equations with m unknowns: $\mathbf{A}_\alpha \times \ell = \mathbf{d}$.
When $n \geq 4$, this system has more equations than unknowns. It can be solved by approximation in the least-squares sense:

$$(\mathbf{A}_\alpha \times \ell - \mathbf{d})^2 \rightarrow \min. \tag{4}$$

After taking the gradient we have:

$$\mathbf{A}_\alpha^t \times (\mathbf{A}_\alpha \times l - \boldsymbol{\delta}) = 0 \tag{5}$$

Following algebraic manipulations, we obtain:

$$\mathbf{A}_\alpha^t \times \mathbf{A}_\alpha \times l = \mathbf{A}_\alpha^t \times \boldsymbol{\delta} \tag{6}$$

Thus, we have: $\mathbf{B} \times \ell = \mathbf{c}$, where $\mathbf{B}$ is an $(m \times m)$ matrix, and $\mathbf{c}$ is a vector with m components.

Following Barthélemy and Guénoche (1988) and Makarenkov and Leclerc (1999), we apply a slightly modified Gauss-Seidel method to solve the above system. The method consists of decomposing $\mathbf{B}$ into its diagonal (Δ), its strictly upper triangular component ($-\mathbf{F}$), and its strictly lower triangular component ($-\mathbf{E}$):

$$\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mm} \end{pmatrix} = \begin{pmatrix} & & -\mathbf{F} \\ & \Delta & \\ -\mathbf{E} & & \end{pmatrix} = \Delta - \mathbf{E} - \mathbf{F}. \tag{7}$$

Then, we apply the iterative procedure:

$$\Delta \times \ell^{(k+1)} = \mathbf{E} \times \ell^{(k+1)} + \mathbf{F} \times \ell^{(k)} + \mathbf{c}, \tag{8}$$

which allows us to compute successively the components of the vector $\ell(j)^{(k+1)}$, corresponding to the edge lengths at the k+1-th iteration, from those of $\ell(j)^{k}$. If the computed value of $\ell(j)^{(k+1)}$ is negative, it is replaced with the value 0. This operation is equivalent to the projection on the cone $\mathbf{L} \geq 0$, which ensures an appropriate solution.

The exact equation used in this method is the following for all $j = 1, 2, \dots, m$:

$$\ell(j)^{(k+1)} = (- (\textstyle\sum_{j+1 \leq i \leq m} b_{ij} \ell(j)^{(k)}) - (\sum_{1 \leq i \leq j-1} b_{ij} \ell(j)^{(k+1)}) + c_j) / b_{jj}. \tag{9}$$

Thus, the main steps of the partial gene transfer algorithm can be stated as follows:

Preliminary Step

This step corresponds to the preliminary step discussed in the context of the complete gene transfer model. It consists of inferring the species and gene phylogenies denoted respectively T and T' whose leaves are labeled by the same set X of n taxa. Because the classical Robinson and Foulds distance is defined only for tree topologies, we use the least-squares as a unique optimization criterion when modeling partial HGTs.

Step 1. Test all connections between pairs of branches in the species tree T. For each HGT connexion satisfying evolutionary constraints, carry out the following optimization:

- **a)** Fix the value of the fraction of the gene being transferred α (e.g., one can try in turn the values of 0, 0.1, 0.2, ..., and 1.0). Compute using the Gauss-Seidel method the optimal lengths $\boldsymbol{l}$ of the edges in the species tree (or network, starting from Step 2) T.
- **b)** Go back to the original equation system: $\mathbf{A}_a \times \boldsymbol{l} = \mathbf{d}$. Fix the values of the vector $\boldsymbol{l}$ found using the Gauss-Seidel method and solve this problem by least-squares considering as unknown the parameter α.
- **c)** Then, fix the optimal value of α found and repeat the computation until both unknown parameters $\boldsymbol{l}$ and α converge to a certain solution.

All eligible pairs of branches in T can be processed in this way. The HGT connection providing the smallest value of the LS coefficient Q and satisfying the defined evolutionary constraints should be selected for the addition to the species tree T, transforming it into a phylogenetic network.

Steps 2...*k*. Run the algorithm until a fixed number k of partial gene transfers is found and added to T or the value of the LS criterion Q is lower than a pre-established threshold ε.

Time complexity of this algorithm is $O(kn^5)$ to add k partial horizontal gene transfers to the species tree with n leaves.

2.5 Bootstrap validation of horizontal gene transfers

Bootstrap analysis can be used to place confidence intervals on internal branches of evolutionary trees (Felsenstein 1985). We designed a bootstrap validation procedure for computing the bootstrap scores either for a specific gene transfer or a whole gene transfer scenario. The following strategy was adopted to assess the reliability of obtained HGTs. Because we are mostly interested in the evolution of a given gene or a group of genes, the sequences used to build the species tree are not resampled. The species tree is taken as an *a priori* assumption of the method and held constant. The sequence data used to build the gene tree are drawn with replacement in order to create a series of pseudo-replicates. The HGT detection algorithm is then carried out on the bootstrapped pseudo-replicates. Thus, for all HGT branches appearing in the original scenario, we verify if they appear in the obtained transfer scenarios, using as input the original species tree and the gene tree inferred from the sets of pseudo-replicates. It is worth noting that among resampled datasets only those that give rise to a gene phylogenetic tree such that it contains the root branch separating this tree into exactly the same bipartition sets as the root branch of the original gene tree does, are eligible for the HGT bootstrap analysis.

SIMULATION STUDY

A Monte Carlo study was conducted to test the ability of the new method to recover correct gene transfers. In the framework of *the complete HGT model only* we examined how the detection procedure performed depending on the model of sequence evolution, number of observed species, and sequence length. The results illustrated in Figures 6 and 7, and reported in Tables 1 and 2 (see Appendix) were obtained from simulations carried out with random binary phylogenetic trees with 8, 16, 24, 32, 48, and 64 leaves, whereas the sequence length varied from 125 to 1000 sites. The simulation procedure consisted of the five basic steps described below:

1. A true tree topology, denoted T, was obtained using the random tree generation procedure proposed by Kuhner and Felsenstein (1994). The branch lengths of T were computed using an exponential distribution. Following the approach of Guindon and Gascuel (2002), we added some noise to the branches of the true phylogenies to create a deviation from the molecular clock hypothesis. All the branch lengths of T were multiplied by $1+ax$, where the variable x was obtained from a standard exponential distribution ($P(x>k) = \exp(-k)$), where the constant a was a tuning factor for the deviation intensity. Following Guindon and Gascuel (2002), a was fixed to 0.8. The random trees generated by this procedure are chosen to have the depth of $O(\log(n))$, where n is the number of species (i.e. number of leaves in a binary phylogenetic tree).

2. Each random phylogeny was then submitted to the SeqGen program (Rambaut and Grassly 1996) to simulate sequence evolution along its branches according to the Jukes and Cantor (1969), Kimura 2-parameter (1980), and Jin-Nei Gamma (1990) models.

3. To assess the quality of HGT detection by the new method, we developed a simulation program using the results of SeqGen. For each considered rooted tree, viewed as an organismal phylogeny, our program created one random horizontal gene transfer that respected the evolutionary constraints discussed in the algorithmic section. During this operation, the program regenerated the DNA sequences for each tree node located in the subtree affected by the HGT. As the simulations were carried out for the complete gene transfer model, the HGT destination sequence was set identical to the source sequence and the new sequences were regenerated from it according to the selected evolutionary model.

4. The sequence to distance transformation corresponding to the considered model of evolution was then applied to the DNA sequences associated with the leaves of the phylogeny affected by the gene transfer. The NJ method (Saitou and Nei 1987) was used to infer the gene trees from the obtained distance matrix. The topology of the organismal phylogeny (i.e. true tree T) was supposed to be known.

5. The HGT detection method was then carried out to infer the transfer. The experiments were conducted using the procedures based on the RF and LS optimization. The simulations were carried out for 500 random rooted phylogenies with 8 and 16 leaves and 100 random rooted phylogenies with 24 to 64 leaves.

Figures 6 and 7 present the average simulation results obtained for random phylogenies with 8 to 64 leaves, using as optimization criteria the RF topological distance and LS function, respectively. These figures illustrate how the detection rate changes as the number of sites varies from 125 to 1000. As expected, the detection rate grows as the number of sites increases and the number of species decreases. Note that for the phylogenies with 8 to 32 leaves the best results were obtained under the Kumura and Jukes-

Cantor models. For the phylogenies with 48 to 64 species the best performances were regularly obtained under the Kimura model, whereas the results found under the Jukes-Cantor model were the worst of the three evolutionary models.

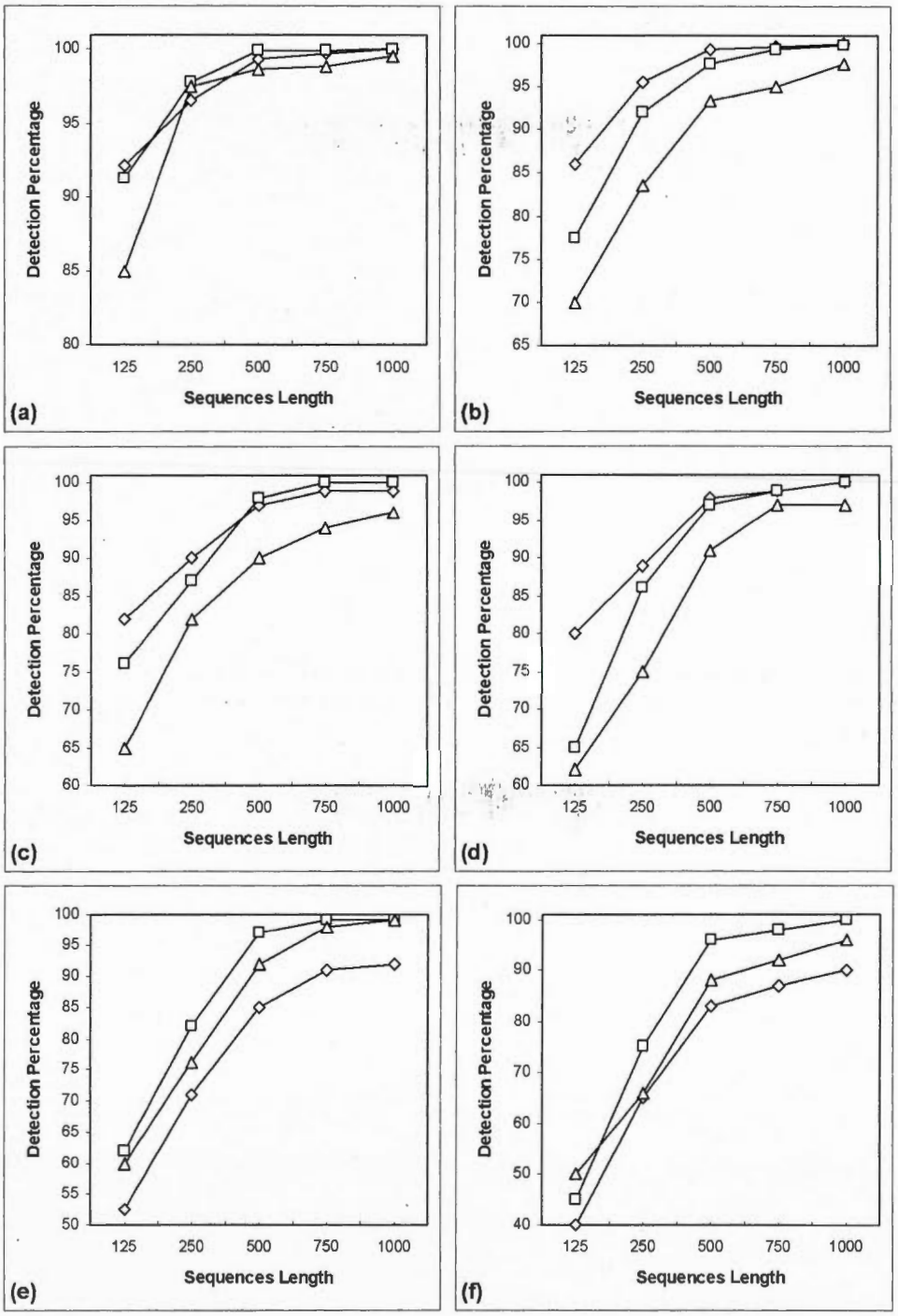


Figure 6. HGT detection rates obtained for random phylogenies with 8 to 64 leaves (8-a, 16-b, 24-b, 32-d, 48-e, 64-f) using the RF topological distance for optimization. Jukes and Cantor (◊), Kimura 2-parameter (□), and Jin-Nei Gamma (Δ) models were used for the tree generation.

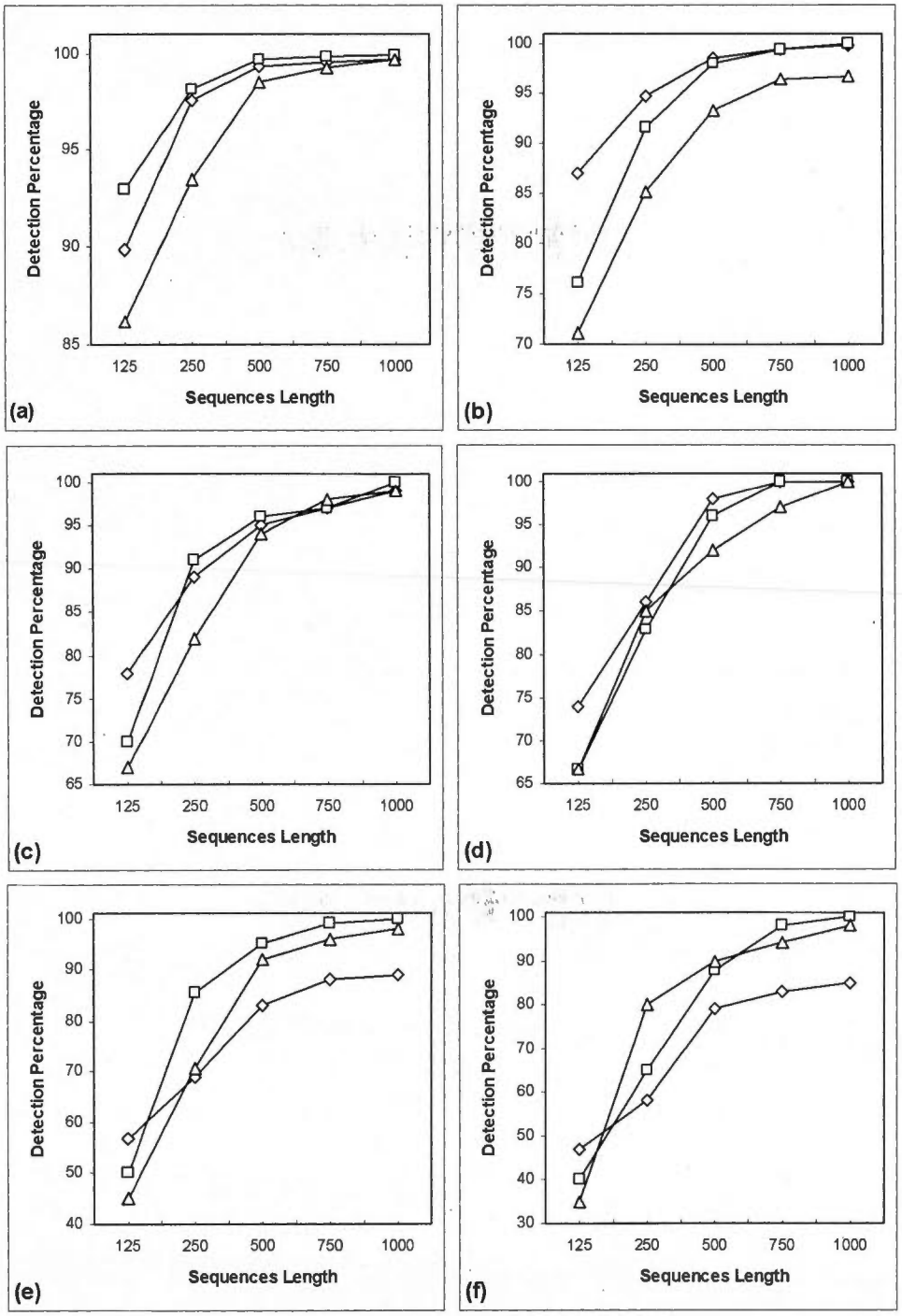


Figure 7. HGT detection rates obtained for random phylogenies with 8 to 64 leaves (8-a, 16-b, 24-b, 32-d, 48-e, 64-f) using the LS function for optimization. Jukes and Cantor (◊), Kimura 2-parameter (□), and Jin-Nei Gamma (Δ) models were used for the tree generation.

This trend can be observed in the case of both optimization criteria. Obviously, with the short sequences we have a bigger phylogenetic error that can either appear like an

HGT, when it does not occur, or disguise a real HGT. Tables 1 and 2 (see Appendix) report the false positive and false negative (indicated in parentheses) detection rates obtained using as optimization criteria the RF distance and LS function, respectively. A false positive HGT is an incorrect transfer found by the algorithm and a false negative HGT is the right transfer that has not been detected. A false positive HGT will always occur if the gene tree inferred by NJ (see Step 4 above) is different from the true gene tree (see Step 3 above), but it can also take place when both trees are identical but a transfer going to the direction opposite to the correct HGT disguises it, leading to the same gene tree (see Maddison 1997).

False negative HGTs are mostly due to the error of inferring the gene tree, but can also happen when a transfer going to the opposite direction disguises the correct HGT. As defined, the false positive detection rate is always bigger or equal to the negative one. The analysis of Tables 1 and 2 shows that the false negative rate is almost as big as the false positive rate when the tests were conducted with large phylogenies (48 and 64 species) and short sequences (125 and 250 sites). The false negative rate was noticeably lower than the false positive one in the case of the large phylogenies and long sequences. Furthermore, we have measured the recovery rates for the HGT source, destination, and source and destination combined (i.e. the latter parameter corresponds to the detection rate depicted in Figures 6 and 7). These tests were carried out under the Jukes and Cantor model of sequence evolution and using the RF distance for the algorithmic optimization. Note that the transfer destinations were generally better detectable than their sources. The difference in the source-destination detection was more important for the short sequence. For example, for the sequences with 125 sites it varied, on average, from 6% (for 8 species) to 1% (for 64 species). However, for the longer sequences the source and destination rates were very similar.

Generally, the procedure based on the RF distance provided better results than that based on the LS function. Nevertheless, some noticeable exceptions (e.g. under the Kimura model for the phylogenies with 8 leaves or under the Jin-Nei model in the case of the short sequences) can be pointed out. The simulation study suggested that the accuracy of the transfer detection is highly dependable on the model of sequence evolution, number of considered species, and length of observed sequences.

RESULTS and DISCUSSION

Detecting horizontal transfers of the gene *rpl12e*

We first tested our algorithm on the phylogeny of 14 species of Archaea originally considered by Matte-Tailliez et al. (2002). The latter authors discuss problems encountered when reconstructing some parts of the archaeal phylogeny, pointing out the evidence of HGT events perturbing the evolution of a number of considered genes. Matte-Tailliez et al. inferred the maximum likelihood tree (Figure 9, undirected lines) based on the concatenated 53 ribosomal proteins (7,175 positions) and compared it to the maximum likelihood phylogeny of the gene *rpl12e* (Figure 8) built for the same 14 organisms. The calculations of the best ML tree and its branch lengths for the 53 concatenated proteins were conducted using the PUZZLE program with Γ-law correction.

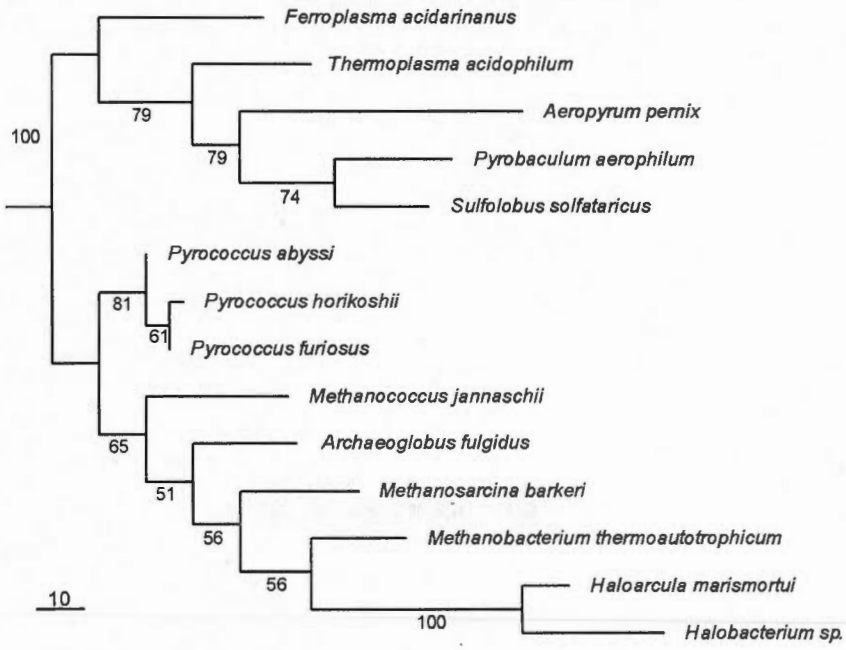


Figure 8. Maximum likelihood phylogenetic tree for the protein *rpl12e* (89 positions). Numbers close to branches are ML bootstrap scores obtained from the sampled protein sequences using the SeqBoot and Proml (JTT model) programs from the PHYLIP package (Felsenstein, 1989). Its topology is identical to the tree found by Matte-Taillez et al. (2002, Figure 3).

Given the topological incongruence of the obtained phylogenies, the authors hypothesized a few cases of lateral transfers of the gene *rpl12e*. More precisely, the case of the transfer between the clades of Thermoplasmatales (*Ferroplasma acidarmanus* and *Thermoplasma acidophilum*) and Crenarchaeota (*Aeropyrum pernix*, *Pyrobaculum aerophilum* and *Sulfolobus solfataricus*) was indicated as the most evident one.

In order to apply our method, we first reconstructed from the original sequences the topologies of the gene (Figure 8) and species trees (Figure 9, undirected lines). The computations were conducted in the framework of the complete gene transfer model, using the RF optimization and subtree constraint options (Figure 1). Five directed branches needed to reconcile the species and gene topologies have been found (Figure 9). The connection representing the transfer between the cluster of *Halobacterium sp.* and *Haloarcula marismortui* and the species *Methanobacterium thermoautotrophicum* was found in the first iteration. This transfer provided the biggest drop of the RF distance between the species and gene phylogenies; its bootstrap score is 55%.

In the second and third iterations, we found the reconciliation branches between the species *Pyrococcus horikoshii* and *Pyrococcus furiosus* and between *Sulfolobus solfataricus* and *Pyrobaculum aerophilum*. Both of these reconciliation branches link closely related species. Such kind of connections may be due to HGT as well as to local topological rearrangements necessary because of the tree reconstruction artifacts (e.g. attraction of long branches, unequal evolutionary rates, etc). The transfer branches 4 and 5 linking the cluster of Crenarchaeota to the species *Thermoplasma acidophilum* and *Ferroplasma acidarmanus* can be interpreted as HGT events that might have taken place between Thermoplasmatales and Crenarchaeota.

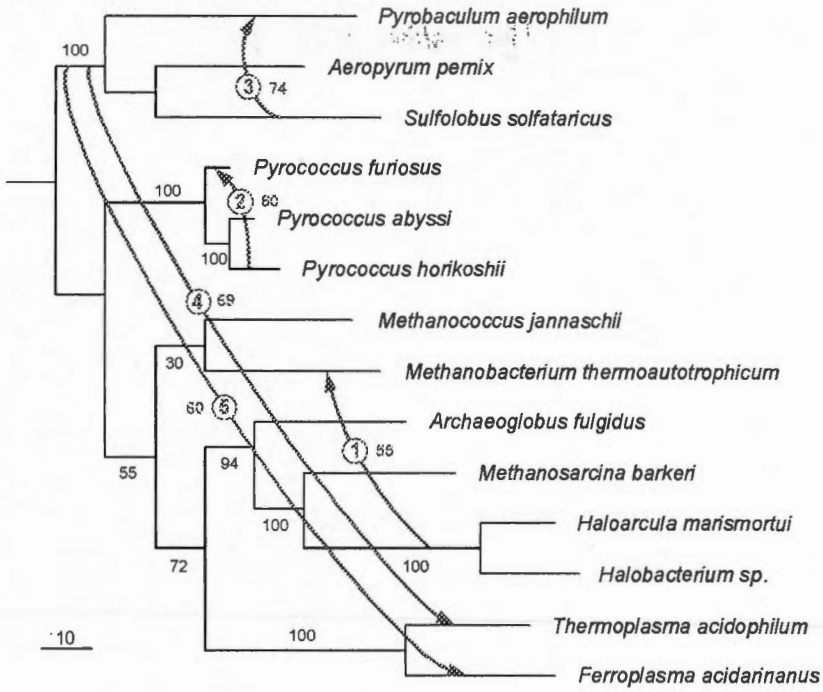


Figure 9. Species tree (Matte-Taillez et al. 2002, Figure 1a) with five reconciliation branches (denoted by arrows). Numbers close to branches are ML bootstrap scores computed by the RELL method upon 2,000 top-ranking trees using the MOLPHY program without correction for among-site variation. Numbers on HGT arrows indicate their order of appearance in the unique gene transfer scenario found by the HGT detection method. Bootstrap scores for transfers are indicated by numbers close to arrow circles. Arrows 4 and 5 depict the HGTs between the clades of Thermoplasmatales and Crenarchaeota also predicted by Matte-Taillez et al. (2002).

In the second and third iterations, we found the reconciliation branches between the species *Pyrococcus horikoshii* and *Pyrococcus furiosus* and between *Sulfolobus solfataricus* and *Pyrobaculum aerophilum*. Both of these reconciliation branches link closely related species. Such kind of connections may be due to HGT as well as to local topological rearrangements necessary because of the tree reconstruction artifacts (e.g. attraction of long branches, unequal evolutionary rates, etc). The transfer branches 4 and 5 linking the cluster of Crenarchaeota to the species *Thermoplasma acidophilum* and *Ferroplasma acidarmanus* can be interpreted as HGT events that might have taken place between Thermoplasmatales and Crenarchaeota.

Note, that HGT between these two groups was also predicted by Matte-Taillez et al. (2002). In fact, the transfers 4 and 5 could consist of a unique transfer between the clades of Thermoplasmatales and Crenarchaeota that was separated into two transfers by our method due to the application of the subtree constraint (Figure 1) and the presence of the tree reconstruction artifacts. Figure 10 illustrates the evolution of the newly formed Thermoplasmatales-Crenarchaeota clade involving the HGTs 4 and 5. The usage of the LS criterion instead of RF leads to the solution consisting of 6 HGTs including all transfers from Figure 9 except the HGT number 2 that goes in the opposite direction. Note that a new reconciliation branch found with LS brings the species *Methanococcus jannaschii* to the cluster of 4 species including *Archaeoglobus fulgidus*. This reconciliation branch turns out to be useless and have a low bootstrap score of 14%.

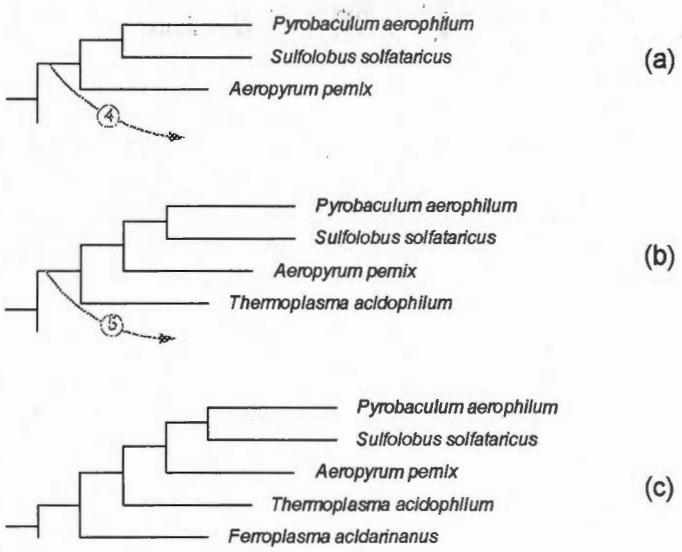


Figure 10. Changes in the Crenarchaeota-Thermoplasmatales cluster occurring after the addition of HGT branches 4 and 5. (a) This cluster after the transfer 3; the species *Thermoplasma acidophilum* joins the Crenarchaeota cluster. (b) This cluster after the transfer 4; the species *Ferroplasma acidarmanus* is added to the clade comprising three Crenarchaeota and *Thermoplasma acidophilum*. (c) This cluster after the transfer 5.

CONCLUSION

We presented two polynomial-time algorithms for detecting horizontal gene transfer events. We considered the complete and partial gene transfer models, implying at each step, either the transformation of a species phylogeny into another tree or its transformation into a network structure. The algorithm for inferring complete gene transfers exploits the discrepancies between the species and gene phylogenies either to map the gene tree into the species tree by least-squares or to compute a topological distance between them and then estimate the possibility of a HGT event between each pair of branches of the species phylogeny. The models based on the optimization of the least-squares function and the Robinson and Foulds topological distance were introduced.

Inferred HGTs should be carefully analyzed using all available information about the data in hand in order to select the transfers that will be represented as a final solution. Each gene transfer branch added to the species phylogeny aids to resolve a conflict between it and the gene tree (i.e. helps to reconcile the species and gene phylogenies). A bootstrap validation procedure allowing one to assess the reliability of a specific gene transfer or whole gene transfer scenario was proposed. A comprehensive Monte Carlo study was carried out to test the ability of the new method to recover correct HGTs. It provided very encouraging results especially when the Robinson and Foulds distance was used as an optimization criterion. The example of the evolution of the gene *rpl12e* was considered in the application section. More simulation work is required to investigate the properties of the algorithm intended to infer partial gene transfers.

As any method of phylogenetic inferring, the new HGT detection method is subject to a number of artifacts which generally affect phylogenetic analysis; the main of them being: Attraction of long branches, unequal evolutionary rates, and situations when the occurrence of some HGT events almost coincides with speciation events located closely to the recipient species. It is important to investigate in greater details the impact of these artifacts on the HGT detection technique introduced in this article. It would be also interesting to extend the presented model to the case, where the gene and species trees have different numbers of taxa; this situation can take place when some species have more than one copy of the gene under consideration.

The software implementing the new algorithms for detecting complete and partial horizontal gene transfers is freely available at the following URL address: < http://www.info2.uqam.ca/~boca05/software/> (this is a consol version running on the Unix and Windows platforms; it is distributed along with its C++ source code). A graphical version of this program has been also implemented and included in the T-Rex web server (Makarenkov 2001) at the following URL: < http://www.trex.uqam.ca >.

ACKNOWLEDGEMENTS

The authors are grateful to Dr. Hervé Philippe for his help in the analysis of the *rpl12e* data.

REFERENCES

Barthélemy, J.-P. and Guénoche, A.: *Les arbres et les représentations des proximités*, Paris: Masson (1988)

Barthelemy J.-P. and Guenoche A.: *Trees and proximity representations.* New York: Wiley (1991)

Boc, A., Makarenkov, V.: New Efficient Algorithm for Detection of Horizontal Gene Transfer Events. In: Benson, G. and Page, R. (eds.): *Algorithms in Bioinformatics.* WABI. LNCS. Springer Verlag, Budapest (2003) 190-201

Charleston, M. A.: Jungle: a new solution to the host/parasite phylogeny reconciliation problem. *Math. Bioscience* (1998) 149:191-223

Denamur, E., Lecointre, G. and Darlu, P. et al. (12 co-authors): Evolutionary implications of the frequent horizontal transfer of mismatch repair genes. *Cell* (2000) 103:711-721

Doolittle, W. F.: Phylogenetic classification and the universal tree. *Science* (1999) 284:2124-2129

Felsenstein, J.: Confidence limits on phylogenies: An approach using the bootstrap. *Evolution* (1985) 39:738-791

Felsenstein, J. PHYLIP - Phylogeny Inference Package (Version 3.2) *Cladistics* (1989) 5:164-166

Guindon, S. and Gascuel, O.: Efficient biased estimation of evolutionary distances when substitution rates vary across sites. *Mol. Biol. Evol.*, (2002) 19:534-543

Hallett, M. and Lagergren, J.: Efficient algorithms for lateral gene transfer problems. In: El-Mabrouk,N., Lengauer,T., Sankoff,D. (eds.): *RECOMB*, ACM, New-York (2001) 149-156

Hallett, M., Lagergren, J., Tofigh, A.: Simultaneous identification of duplications and lateral transfers. In: Bourne, P.E. and Gusfield, D. (eds.): *RECOMB*, ACM, San Diego (2004) 347-356

Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. *Discr. Appl. Math.* (1996) 71:153-169

Jin, L. and Nei, M.: Limitations of the evolutionary parsimony method of phylogenetic analysis *Mol. Biol. Evol.*, (1990) 7: 82-102

Jukes, T.H. and Cantor, C.: Mammalian Protein Metabolism. In *Evolution of protein molecules* (H. N. Munro, editor). New York: Academic Press. . (1969) 21-132

Kimura, M. A: Simple method for estimating evolutionary rate of base substitutions through comparative studies of nucleotide sequences. *J. Mol. Evol.* (1980) 16:111-120

Kuhner, M., Felsenstein, J.: A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Mol. Biol. Evol.* (1994) 11:459-468

Lawrence, J. G., Ochman, H.: Amelioration of bacterial genomes: rates of change and exchange. *J. Mol. Evol.* (1997) 44:383-397

Legendre, P. (Guest Editor): Special section on reticulate evolution. *J. of Classif* (2000) 17: 153-195

Legendre, P. and Makarenkov, V.: Reconstruction of biogeographic and evolutionary networks using reticulograms. *Syst. Biol.* (2002) 51:199-216

Maddison, W. P.: Gene trees in species trees. *Syst. Biol.* (1997) 46:523-536

Makarenkov, V. and Leclerc, B.: An algorithm for the fitting of a phylogenetic tree according to a weighted least-squares criterion. *J. of Classif* (1999) 16:3-26

Makarenkov, V. T-Rex: reconstructing and visualizing phylogenetic trees and reticulation networks. *Bioinformatics* (2001) 17:664–668

Makarenkov, V., Boc, A., Delwiche, C. F., Diallo, A.B. and Philippe, H.: New efficient algorithm for modeling partial and complete gene transfer scenarios. Data Science and Classification, V. Batagelj, H.-H. Bock, A. Ferligoj, and A. Ziberna (Eds.), IFCS 2006, Series: *Studies in Classification, Data Analysis, and Knowledge Organization*, Springer Verlag, (2006) 341-349

Matte-Tailliez, O., Brochier, C., Forterre, P. and Philippe, H.: Archaeal phylogeny based on ribosomal proteins. *Mol. Biol. Evol.* (2002) 19:631-639

Mirkin, B., Fenner T. I., Galperin M. and Koonin E.: Algorithms for computing parsimonious evolutionary scenarios for genome evolution, the last universal common ancestor and dominance of horizontal gene transfer in the evolution of prokaryotes. *BMC Evol. Biol.* 3 (2003)

Mirkin, B.: Mapping gene family data onto evolutionary trees, in M. Chavent, O. Dordan, C. Lacomblez, M. Langlais, and B. Patouille (Eds.), *Proceedings of the 11th Rencontres de la Société Francophone de Classification*, University of Bordeaux, (2004) 61-68

Moret, B. M. E., Nakhleh, L., Warnow, T., Linder, C. R., Tholse, A., Padolina, A., Sun, J. and Timme R.: Phylogenetic networks: modeling, reconstructibility, and accuracy. IEEE/ACM Trans. on Comput. *Biol. and Bioinf.* (2004) 1:13-23

Nakhleh, L., Ruths, D. and Innan, H.: Gene trees, species trees, and species networks. In: Guerra, R. and Allison, D. (eds.): *Meta-analysis and combining information in genetics* (2005) 1-27

Page, R.D.M. and Charleston, M.A.: From gene to organismal phylogeny: Reconciled trees *Bioinformatics* (1998a) 14:819-820

Page, R. D. M. and Charleston, M. A.: Trees within trees: phylogeny and historical associations. *Trends Ecol. Evol.* (1998b) 13:356-359

Rambaut, A. and Grassly, N.C.: Seq-Gen: An application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Comput. Appl. Biosci.* (1996)

Robinson, D. R. and Foulds, L. R.: Comparison of phylogenetic trees. *Math. Biosciences* (1981) 53:131-147

Saitou, N. and Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.* (1987) 4:406-425

von Haeseler, A., Churchill, G. A.: Network models for sequence evolution. *J. Mol. Evol.* (1993) 37:77-85

APPENDIX

This Appendix includes the results of the tests described in the section Simulation Study. The results reported in Tables 1 and 2 correspond to the graphics represented in Figures 6 (optimization using the RF distance) and 7 (optimization using the LS function). They were obtained from simulations carried out for random binary phylogenies with 8, 16, 24, 32, 48, and 64 leaves, whereas the sequence length varied from 125 to 1000 sites. Note that the sum of the HGT detection rate shown in Figures 6 and 7 and of the false negative detection rate reported in Tables 1 and 2 is always 100%.

Table 1. False positive and false negative (in parentheses) detection rates obtained for random phylogenies with 8 to 64 leaves using the RF distance as an optimization criterion. A false positive HGT is an incorrect transfer found by the algorithm and a false negative HGT is the right transfer that has not been found. For each sequence length, the simulations were carried out for 500 random phylogenies with 8 and 16 leaves and 100 random phylogenies with 24 to 64 leaves.

RF rates (in %)			Sequence length				
			125	250	500	750	1000
Species number	8	Jukes-Cantor	14.9 (7.8)	5.9 (3.5)	1.1 (0.7)	0.3 (0.3)	0.0 (0.0)
		Kimura	12.9 (8.7)	3.3 (2.2)	0.2 (0.1)	0.1 (0.1)	0.0 (0.0)
		Jin-Nei	20.1 (15.0)	3.9 (2.5)	1.6 (1.3)	1.1 (1.1)	0.5 (0.5)
	16	Jukes-Cantor	25.7 (14.0)	7.1 (4.5)	1.2 (0.7)	0.4 (0.3)	0.0 (0.0)
		Kimura	35.1 (22.5)	11.9 (7.9)	3.2 (2.3)	0.6 (0.6)	0.1 (0.1)
		Jin-Nei	43.0 (30.0)	22.5 (16.5)	7.6 (6.6)	5.3 (4.9)	2.3 (2.3)
	24	Jukes-Cantor	36 (18)	15 (10)	4 (3)	1 (1)	1 (1)
		Kimura	43 (24)	24 (13)	4 (2)	2 (0)	0 (0)
		Jin-Nei	55 (35)	33 (18)	19 (10)	9 (6)	5 (4)
	32	Jukes-Cantor	37 (20)	29 (11)	4 (2)	1 (1)	1 (0)
		Kimura	60 (35)	31 (14)	8 (3)	3 (1)	2 (0)
		Jin-Nei	70 (38)	47 (25)	16 (9)	8 (3)	8 (3)
	48	Jukes-Cantor	65 (48)	49 (29)	28 (15)	27 (9)	25 (8)
		Kimura	55 (38)	46 (18)	9 (3)	3 (1)	1 (1)
		Jin-Nei	70 (40)	58 (24)	19 (8)	9 (2)	4 (1)
	64	Jukes-Cantor	70 (60)	45 (35)	27 (17)	23 (13)	20 (10)
		Kimura	65 (55)	35 (25)	14 (4)	12 (2)	10 (0)
		Jin-Nei	60 (50)	44 (34)	22 (12)	18 (8)	14 (4)

Table 2. False positive and false negative (in parentheses) detection rates obtained for random phylogenies with 8 to 64 leaves using the LS function as an optimization criterion. A false positive HGT is an incorrect transfer found by the algorithm and a false negative HGT is the right transfer that has not been found. For each sequence length, the simulations were carried out for 500 random phylogenies with 8 and 16 leaves and 100 random phylogenies with 24 to 64 leaves.

LS rates (in %)			Sequence length				
			125	250	500	750	1000
Species number	8	Jukes-Cantor	17.2 (10.1)	5.0 (2.5)	0.8 (0.7)	0.8 (0.5)	0.3 (0.3)
		Kimura	10.8 (7.0)	2.8 (1.9)	0.3 (0.3)	0.2 (0.2)	0.1 (0.1)
		Jin-Nei	18.6 (13.8)	7.8 (6.5)	1.7 (1.5)	0.9 (0.8)	0.5 (0.3)
	16	Jukes-Cantor	25.5 (13.0)	7.6 (5.3)	2.2 (1.4)	0.8 (0.5)	0.1 (0.1)
		Kimura	37.6 (23.8)	11.9 (8.4)	2.3 (2.0)	0.6 (0.6)	0.0 (0.0)
		Jin-Nei	40.9 (28.8)	20.9 (14.8)	8.1 (6.7)	3.8 (3.6)	3.3 (3.3)
	24	Jukes-Cantor	43 (22)	13 (11)	5 (5)	3 (3)	1 (1)
		Kimura	59 (30)	26 (9)	7 (4)	4 (3)	1 (0)
		Jin-Nei	67 (33)	26 (18)	12 (6)	6 (2)	3 (1)
	32	Jukes-Cantor	47 (26)	21 (14)	5 (2)	0 (0)	0 (0)
		Kimura	56 (33)	31 (17)	9 (4)	0 (0)	0 (0)
		Jin-Nei	50 (33)	31 (15)	12 (8)	11 (3)	4 (0)
	48	Jukes-Cantor	53 (43)	38 (31)	33 (17)	22 (12)	19 (11)
		Kimura	60 (50)	34 (14)	16 (5)	5 (1)	2 (0)
		Jin-Nei	65 (55)	50 (29)	25 (8)	12 (4)	10 (2)
	64	Jukes-Cantor	63 (53)	52 (42)	41 (21)	27 (17)	25 (15)
		Kimura	70 (60)	45 (35)	22 (12)	15 (2)	10 (0)
		Jin-Nei	75 (65)	40 (20)	20 (10)	16 (6)	12 (2)

CHAPITRE IV

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT DES MÉTHODES DE DISTANCES

4.1 Introduction

Les programmes utilisant des méthodes de distances qui ont été parallélisés sont principalement ceux inclus dans T-Rex (Tree and Reticulogram reconstruction). Il s'agit de : Neighbor-Joining (Saitou et Nei, 1987), Unweighted Neighbor-Joining (Gascuel, 1997), Addtree (Sattath et Tversky, 1977), Reconstruction par ordre circulaire (Makarenkov et Leclerc, 1997), Weighted Least-Squares (Makarenkov et Leclerc, 1999) et BioNJ (Gascuel, 1997). De plus, le populaire programme FITCH (Fitch, 1971) a également été parallélisé. Dans ce chapitre nous expliquerons de façon sommaire les fonctionnalités importantes de ces programmes ainsi que la forme de parallélisme qui a été mise au point.

La figure IV-1 montre l'interface du serveur web de T-Rex pour les six programmes parallélisés.

Figure IV-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de distances.

4.2 Forme de parallélisme employée

Étant donné que les programmes présentés dans la figure IV-1 sont basés sur des algorithmes composés de parties indépendantes et que différents opérateurs de ces algorithmes (tels que les boucles) peuvent être exécutés simultanément, nous avons utilisé le parallélisme de contrôle (voir chapitre II, formes de parallélisme).

Le point commun des algorithmes implémentés dans ces sept programmes réside au niveau du choix successif des paires d'espèces à regrouper pour former un arbre. Le choix d'une paire d'espèces est basé sur des critères à maximiser ou à minimiser. Par conséquent, le critère est calculé pour chacune des paires d'espèces, puis la paire avec la meilleure valeur du critère est sélectionnée. Les calculs de critères au niveau de chaque paire d'espèces sont effectués à travers une série de boucles imbriquées et sont indépendants les uns des autres.

La parallélisation a consisté à diviser équitablement le calcul des critères au niveau des lames utilisées par le cluster (lors d'une exécution en parallèle). Cette division équitable a été réalisée à travers l'instruction de contrôle suivante : if (*indice % nombredelames = myid*); où *indice* représente l'indice des espèces (allant souvent de zéro au nombre d'espèces moins un), *%* représente l'opération « modulo », *nombredelames* représente le nombre de lames utilisées

pour exécuter le programme en parallèle et *myid* représente l'identificateur de la lame du cluster (voir chapitre II, Message Passing Interface).

Ensuite, une synchronisation a été effectuée entre toutes les lames utilisées afin de trouver la lame qui contient la paire avec la valeur optimale globale du critère (car chaque lame contient une paire avec une valeur optimale locale du critère). Au niveau de la synchronisation, une uniformisation des variables du programme a été effectuée (étant donné que les lames ont effectué des calculs sur des données différentes) afin de conserver la conformité des résultats obtenus au niveau de toutes les lames utilisées par le cluster lors de l'exécution de l'algorithme parallèle.

La forme générale du pseudo-code des programmes parallélisés est la suivante :

- Traitement séquentiel (Cas d'une seule lame utilisée) :

```
if (nombredelames = 1) {
        for (indice = 1; indice <= maxindice; indice++) {        ...}
}
```

- Traitement parallèle :

```
else {
        for (indice = 1; indice <= maxindice; indice++) {
                if (indice % nombredelames == myid) {...}
        }
}
```

- Synchronisation :

```
if (myid > 0) {
        Envoi de toutes les informations nécessaires à la première lame (myid=0.)
        Réception des nouvelles valeurs envoyées par la lame précédente.
        if (myid < nombredelames-1) {
                Envoi des nouvelles valeurs à la lame suivante.
        }
```

}

else {

Réception des informations nécessaires envoyées par les autres lames.

Recherche de la paire ayant la meilleure valeur critère.

Envoi des nouvelles informations à la lame suivante.

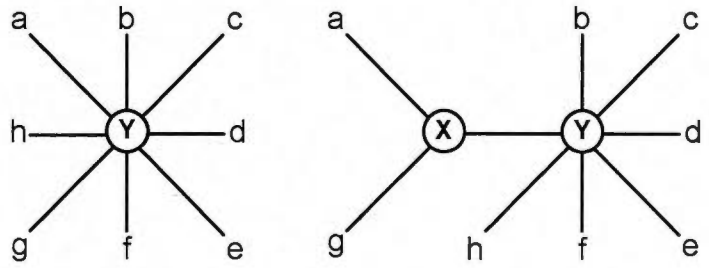
}

4.3 Neighbor-Joining

Neighbor-Joining (NJ; Saitou et Nei, 1987) est une méthode de reconstruction d'arbres phylogénétiques qui repose sur l'application du regroupement combiné à une approximation efficace du principe d'évolution minimum. Cette méthode permet un taux d'évolution différent entre les lignées étudiées. Le principe de cette méthode est de trouver les paires d'unités taxonomiques opérationnelles (OTUs) qui minimisent la longueur totale de l'arbre (i.e., la somme des longueurs des branches) à chaque étape du regroupement des unités taxonomiques, avec un arbre en étoile au départ.

NJ garantit de recouvrer la vraie phylogénie si la matrice de distance est une réflexion exacte de la phylogénie (Felsenstein, 2004). Il a une complexité algorithmique de $O(n^3)$.

Pour reconstruire une phylogénie T à partir d'une matrice de dissimilarités d, cette dernière est transformée afin d'obtenir un arbre en étoile T (figure IV-2(a)). Puis la paire de taxons qui minimise la longueur totale de l'arbre est choisie et remplacée par un nœud interne X (figure IV-2(b)). La dernière partie de la méthode est exécutée n-3 fois.



(a) **(b)**

Figure IV-2 Arbre en étoile (a) et son développement (b) (Saitou et Nei, 1987).

L'algorithme détaillé de Neighbor-Joining est le suivant (Saitou et Nei, 1987):

1. Pour chaque feuille, calculer : $u_i = \sum_{j:j\neq i}^{n} \frac{\delta_{ij}}{n-2}$; où δ représente la matrice de distance;
2. Choisir *i* et *j* pour lesquels la distance $\delta_{ij} - u_i - u_j$ est la plus petite;
3. Fusionner *i* et *j* et calculer les longueurs des branches menant au nœud (*vi*) et au nœud (*vj*) de la manière suivante :

$$v_i = \frac{1}{2}\delta_{ij} + \frac{1}{2}(u_i - u_j),$$
$$v_j = \frac{1}{2}\delta_{ij} + \frac{1}{2}(u_j - u_i).$$

4. Calculer toutes les distances entre le nouveau nœud (*ij*) et les feuilles restantes en utilisant la formule suivante : $\delta_{(ij)k} = \frac{\delta_{ik} + \delta_{jk}}{2}$;
5. Supprimer les feuilles *i* et *j* de la matrice de distances et les remplacer par le nouveau nœud (*ij*) qui sera considéré comme une feuille;
6. Revenir à l'étape 1 si le nombre de nœuds restant est supérieur à 2. Sinon, connecter les deux nœuds restants par une branche.

Pour la méthode NJ, le parallélisme de contrôle a été effectué au niveau de la deuxième étape de l'algorithme présenté ci-dessus (« Choisir i et j pour lesquels la distance $\delta_{ij} - u_i - u_j$ est la plus petite »). A cette étape, les données sont divisées en fonction du nombre de lames qui exécutent le programme. À partir du flot de données qui lui est dédié, chaque lame cherche i et j qui minimisent la distance $\delta_{ij} - u_i - u_j$. Finalement, une synchronisation est effectuée afin de choisir i et j qui minimisent cette distance au niveau de toutes les lames et pour mettre à jour les variables (critiques) au niveau de chaque lame.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme NJ ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'espèces	Temps moyen d'exécution (sec)
1	8	0.000020
1	16	0.000060
1	32	0.000250
1	64	0.001350
1	128	0.008600
2	8	0.002550
2	16	0.005000
2	32	0.008900
2	64	0.020000
2	128	0.107150
4	8	0.009200
4	16	0.015000
4	32	0.032500
4	64	0.055500
4	128	0.155000
8	8	0.017500

8	16	0.030000
8	32	0.069000
8	64	0.135000
8	128	0.287000
16	8	0.026800
16	16	0.058000
16	32	0.160000
16	64	0.322500
16	128	0.600000

Tableau IV-1 Algorithme NJ : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

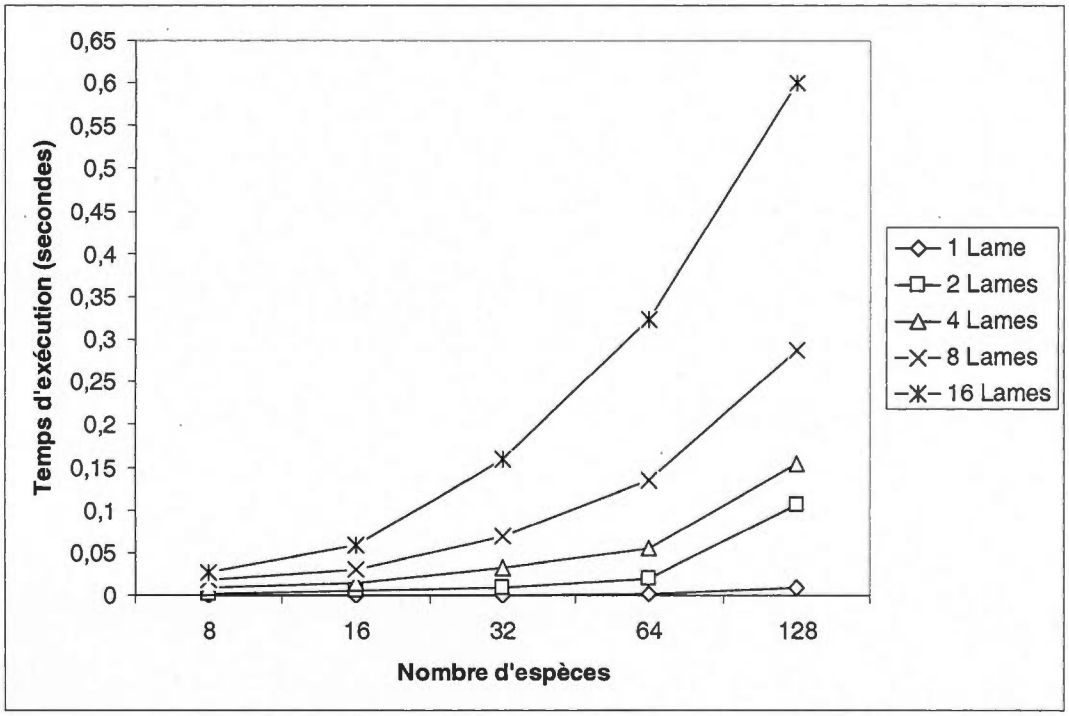


Figure IV-3 Algorithme NJ : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Nous remarquons, en observant le tableau IV-1, que l'algorithme séquentiel est plus rapide que l'algorithme parallèle pour le même nombre d'espèces considérées. Ceci est dû à la supériorité du temps de passage des informations entre les différentes lames au temps gagné par l'exécution parallèle de la boucle de sélection des paires. Cependant, à partir de 400 espèces, l'algorithme parallèle sur deux lames devient plus rapide que l'algorithme séquentiel (tableau IV-2). Plus le nombre d'espèces est élevé, plus la version parallèle est performante car le temps de passage des informations entre les différentes lames devient inférieur au temps gagné par l'exécution parallèle de la boucle de sélection des paires

Nombre de lames	Nombre d'espèces	Temps moyen d'exécution (sec)
1	400	0.300000
1	800	2.800000
2	400	0.260000
2	800	1.700000
4	400	0.450000
4	800	1.250000
8	400	0.640000
8	800	1.560000
16	400	1.220000
16	800	1.900000

Tableau IV-2 Algorithme NJ : Temps écoulé en secondes en fonction du nombre de lames du cluster pour 400 et 800 espèces.

Étant donné que l'algorithme séquentiel de NJ fournit des résultats satisfaisants, la version parallèle sera juste utilisée pour un grand nombre d'espèces (> 450 espèces).

4.4 Unweighted Neighbor-Joining

Unweighted Neighbor Joining (UNJ; Gascuel, 1997) est une adaptation de l'algorithme Neighbor-Joining. Tout comme NJ, le principe de UNJ consiste à trouver les paires d'unités taxonomiques opérationnelles (OTUs) qui minimisent la longueur totale de l'arbre (la somme des longueurs des branches) à chaque étape du regroupement des unités taxonomiques. La méthode a une complexité algorithmique de $O(n^3)$.

UNJ est défini par les trois formules suivantes :

- La formule du critère de correction :
  Dans cette formule r représente le nombre d'espèces contenus dans la matrice.

$$Q_{xy} = R_x + R_y + (r-2)\lambda_{xy}, \quad où \quad R_z = \sum_{i=1}^{r} \lambda_{zi};$$

- La formule du critère d'estimation :

$$\hat{d}_{xu} = \frac{1}{2}\lambda_{xy} + \frac{1}{2(n-n_u)} \sum_{i=1 \neq x,y}^{r} n_i(\lambda_{xi} - \lambda_{yi}) \; ; \; \hat{d}_{yu} \text{ est obtenu par symétrie.}$$

- La formule de réduction :

$$\lambda_{ui} = w_x\lambda_{xi} + w_y\lambda_{yi} - w_x\hat{d}_{xu} - w_y\hat{d}_{yu}, \text{ où } w_x = \frac{n_x}{n_u} \text{ et } w_y = \frac{n_y}{n_u}.$$

L'algorithme UNJ est le suivant (Gascuel 1997) :

1. Initialiser la matrice d'exécution : $\Lambda = (\lambda_{ij}) \leftarrow (\delta_{ij})$;
2. Initialiser le nombre de nœuds restants : $r \leftarrow n$;
3. Initialiser le nombre d'objets par nœud : $n_i \leftarrow 1, i \in \{1,...n\}$;
4. Tant que le nombre de nœuds est supérieur à 3 : Calculer les sommes $R_i, i \in \{1,...r\}$;
5. Trouver la paire $\{x, y\}$ à regrouper pour maximiser la valeur Q_{xy} ;

6. Créer un nœud u tel que : $n_u \leftarrow n_x + n_y$;
7. Estimer la longueur des branches (x,u) (y,u)en utilisant la formule d'estimation;
8. Réduire la matrice Λ en utilisant la formule de réduction;
9. Réduire le nombre de nœuds : $r \leftarrow r-1$;
10. Créer un nœud central et calculer les dernières longueurs de branches de l'arbre en utilisant la formule d'estimation.

Pour la méthode UNJ, le parallélisme de contrôle a été effectué au niveau de la cinquième étape de l'algorithme qui consiste à « Trouver la paire $\{x,y\}$ à regrouper pour maximiser la valeur du critère le critère Q_{xy} ». À cette étape, tout comme pour l'algorithme NJ, les données sont divisées en fonction du nombre de lames qui exécutent le programme. À partir du flot des données qui lui est dédié, chaque lame cherche x et y qui minimisent la valeur du critère Q_{xy}. Finalement, une synchronisation est effectuée afin de choisir x et y qui minimisent la valeur du critère Q_{xy} au niveau de toutes les lames et pour mettre à jour les variables au niveau de chaque lame.

Les résultats de la parallélisation en termes de gain de temps sont identiques à la méthode de Neighbor-Joining. A partir de 450 espèces la version parallèle sur 2 lames devient plus rapide que la version séquentielle. Cela s'explique par le fait que UNJ est une adaptation de l'algorithme Neighbor-Joining ayant la même complexité algorithmique $O(n^3)$.

4.5 ADDTREE

Rappelons tout d'abord que les données de la même origine peuvent être représentées par des arbres additifs (i.e., arbre phylogénétique). Dans ce modèle, les objets sont représentés par les nœuds externes de l'arbre et la dissimilarité entre les objets est la longueur du chemin qui les sépare. Les arbres additifs sont moins restrictifs que les arbres ultramétriques (appelés aussi regroupements hiérarchiques).

Un arbre additif constitué de quatre objets x, y, u et v apparaît forcément dans l'une des configurations de la figure IV-4. Les modèles de distances qui correspondent aux configurations de la figure IV-4 sont les suivants :

(a) $$d(x,y)+d(u,v) < d(x,u)+d(y,v) = d(x,v)+d(y,u)$$

(b) $$d(x,v)+d(y,u) < d(x,u)+d(y,v) = d(x,y)+d(u,v)$$

(c) $$d(x,u)+d(y,v) < d(x,y)+d(u,v) = d(x,v)+d(y,u)$$

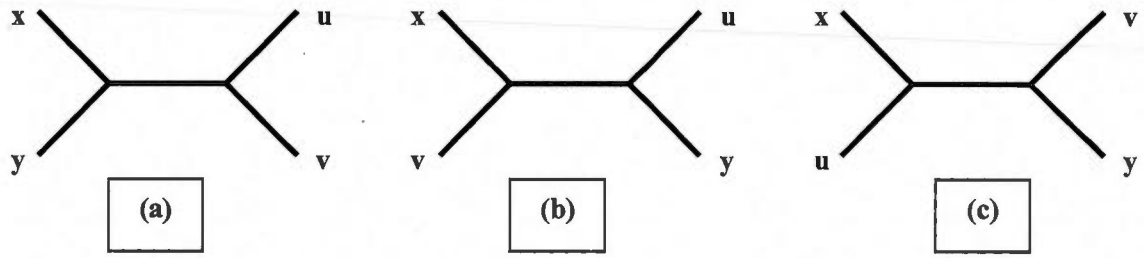


Figure IV-4 Configurations possibles d'arbres additifs binaires avec quatre feuilles.

Le but de la méthode ADDTREE (Sattath et Tversky, 1977) est de sélectionner la configuration la plus appropriée sur la base d'une mesure de dissimilarité (ou distance) observée δ.

Considérons la mesure de dissimilarité (pour quatre objets) telle que :

$$\delta(x,y)+\delta(u,v) \leq \delta(x,u)+\delta(y,v) \leq \delta(x,v)+\delta(y,u).$$

Il est évident, dans ce cas, que la configuration IV-4(a) soit plus représentative de cette dissimilarité que les configurations IV-4(b) et IV-4(c). Par conséquent, la loi (formule) pour choisir la meilleure configuration pour n'importe quel ensemble de quatre objets est la suivante : « Marquer ou noter les objets de manière à satisfaire l'inéquation ci-dessus et sélectionner la configuration IV-4(a) ».

Les objets x, y tout comme les objets u et v sont appelés « voisins ». La construction de l'arbre repose sur le regroupement des objets sur une base de relations de voisinage. Les étapes majeures de la construction sont esquissées plus bas.

Pour toutes les paires x, y, ADDTREE examine tous les objets u, v et compte le nombre de quadruplets dans lesquels x et y sont voisins. La paire x, y avec le plus haut score est sélectionnée et ses membres sont combinés pour former un objet z qui replacera x et y dans l'analyse subséquente.

La dissimilarité entre z et n'importe quel autre objet u sera égale à $(\delta(u,x)+\delta(u,y))/2$.

Par la suite, la paire avec le deuxième plus haut score est sélectionnée. Si ces éléments n'avaient pas été encore sélectionnés, ils sont combinés comme ci-dessus et le balayage des paires continue jusqu'à la sélection de tous les éléments.

Ce processus de regroupement est d'abord appliqué à un ensemble d'objets qui comprend les éléments récemment formés ainsi que les éléments originaux qui n'ont pas été combinés dans ce processus. Le processus de regroupement est par la suite appliqué à plusieurs reprises à l'ensemble jusqu'à ce qu'il ne reste que 3 éléments. Finalement ces trois derniers éléments sont combinés pour former le dernier élément.

La procédure de construction utilise des sommes de dissimilarités pour définir les voisins et pour calculer les distances des nouveaux éléments construits. Cette procédure est appliquée à des données cardinales (données mesurées sur des échelles d'intervalles ou de rapports). Pour les données ordinaires, il existe une version modifiée de la procédure. La méthode ADDTREE a une complexité algorithmique de $O(n^5)$.

Le parallélisme de contrôle a été appliqué au niveau de choix de la paire x et y avec le plus haut score. A cette étape, les données sont divisées en fonction du nombre de lames qui exécutent le programme. À partir des données qui lui sont affectées, chaque lame cherche x et y qui maximisent le score de ADDTREE. Finalement, une synchronisation est effectuée afin de choisir x et y qui maximisent le score au niveau de toutes les lames et pour mettre à jour les variables au niveau de chaque lame.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme ADDTREE ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'espèces	Temps moyen d'exécution (sec)
1	8	0.000060
1	16	0.001800
1	32	0.060000
1	64	1.830000
2	8	0.002100
2	16	0.005500
2	32	0.039000
2	64	0.960000
4	8	0.009500
4	16	0.013000
4	32	0.048000
4	64	0.550000
8	8	0.017500
8	16	0.029000
8	32	0.076000
8	64	0.375000
16	8	0.027500
16	16	0.060000
16	32	0.165000
16	64	0.400000

Tableau IV-3 Algorithme ADDTREE : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

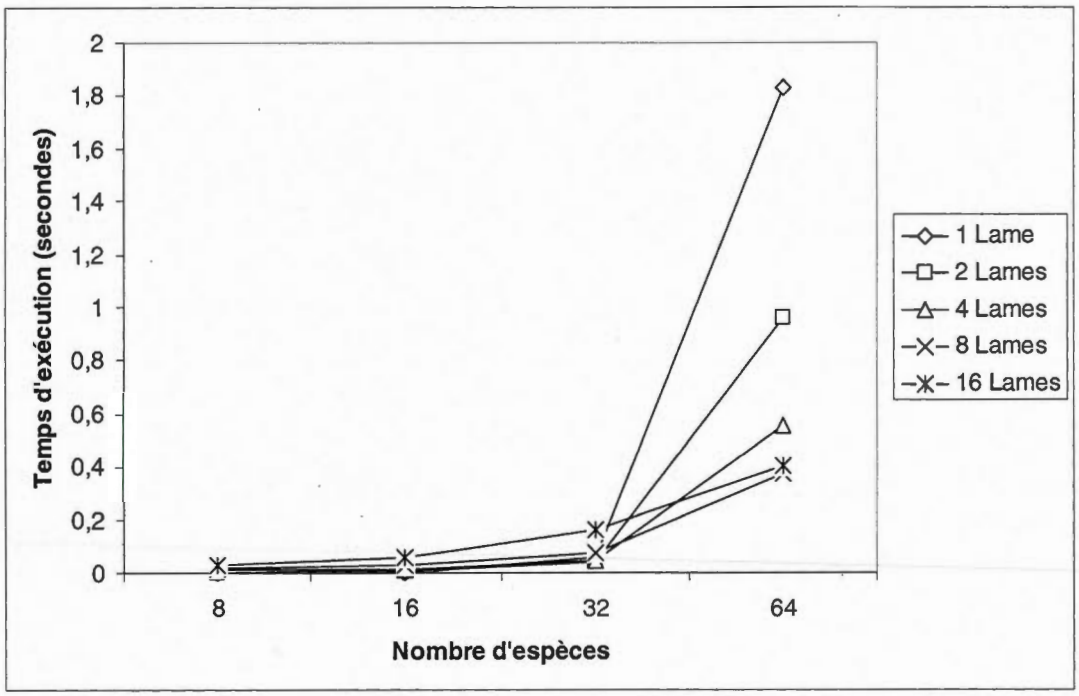


Figure IV-5 Algorithme ADDTREE : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Nous remarquons que l'algorithme séquentiel devient plus lent que l'algorithme parallèle à partir de 30 espèces, quel que soit le nombre de lames utilisées. Ces résultats sont explicables par le fait que la méthode a une complexité de $O(n^5)$.

4.6 Reconstruction par ordre circulaire

Soit X un ensemble de n éléments, la dissimilarité sur X est une fonction d de réels supérieurs ou égaux à zéro qui satisfait les deux conditions suivantes :

- Pour tous $x, y \in X, d(x,y) = d(y,x)$;
- Pour tous $x, y \in X, d(x,y) \geq d(x,x) = 0$.

La dissimilarité d est une métrique si elle satisfait l'inégalité triangulaire :

$$\forall x, y, z \in X, d(x,z) \leq d(x,y) + d(y,z).$$

La reconstruction par ordre circulaire est un algorithme basé sur l'ordre circulaire associé aux représentations planaires des *X-arbres*. Ces représentations planaires sont souvent utilisées pour le codage des *X-arbres* valués par $2n$-3 longueurs de chemins d'une feuille à une autre (n étant le nombre d'éléments de X).

La reconstruction par ordre circulaire est basée sur le théorème suivant : « Soit $X = \{x_1, x_2, ..., x_n\}$ un ensemble fini d'éléments. Pour toute séquence $d_{12}, d_{13}, d_{23}, ..., d_{1i}, d_{i,i+1}, ..., d_{1n}, d_{n-1,n}$ des $2n$-3 réels strictement positifs, il existe un *X-arbre* valué unique T_l tel que $d_{ij} = \sum_{l \in T(x_i, x_j)} l(e)$ et $x_1, x_2, ..., x_n$ est un ordre circulaire de T» (Makarenkov et Leclerc, 1997).

L'algorithme de construction d'un *X-arbre* à partir d'une dissimilarité d est le suivant (Makarenkov et Leclerc, 1997) :

Entrées : Un ensemble fini X contenant n éléments, une dissimilarité d sur X.

Sorties : Un *X-arbre* valué.

Initialisation : Calculer l'ordre de Yushmanov $(x_1, x_2, ..., x_n)$ sur X;

$$V(T) \leftarrow \{x_1, x_2\}; E(T) \leftarrow x_1x_2; l(x_1x_2) \leftarrow d(x_1, x_2); k \leftarrow 1$$

Répéter :

$$k \leftarrow k+1; S \leftarrow 0; i \leftarrow 0;$$

Le problème est d'ajouter la feuille x_{k+1} à l'arbre valué courant T_l^k avec les feuilles $x_1, x_2, ..., x_k$;

Résoudre le problème $P_{1,l}(\alpha, \gamma)$ pour le chemin $T_l^k(x_1 x_k)$:

Tant que $S < \alpha$ faire : $i \leftarrow i+1; S \leftarrow S + l(i, T(x_1 x_k))$;

$u \leftarrow w(i, T(x_1 x_k)); v \leftarrow w'(i, T(x_1 x_k))$;

Si $S = \alpha$ alors

$$V(T) \leftarrow V(T) \cup \{x_{k+1}\}; E(T) \leftarrow E(T) \cup \{ux_{k+1}\}; l(ux_{k+1}) \leftarrow \gamma;$$

Sinon

$$V(T) \leftarrow V(T) \cup \{a_{k+1}, x_{k+1}\}; E(T) \leftarrow (E(T) - \{uv\}) \cup \{ua_{k+1}, va_{k+1}, a_{k+1}x_{k+1}\};$$
$$l(a_{k+1}x_{k+1}) \leftarrow \gamma; l(ua_{k+1}) \leftarrow \alpha - S + l(i, T(x_1 x_k)); l(va_{k+1}) \leftarrow S - \alpha;$$

Jusqu'à $(k = n)$

Cette méthode a une complexité algorithmique de $O(n^4)$.

Ici, le parallélisme de contrôle a été effectué au niveau de la résolution du problème d'ajout d'une nouvelle feuille. A cette étape, les données sont encore divisées en fonction du nombre de lames qui exécutent le programme et, finalement, une synchronisation est effectuée afin de choisir la feuille à ajouter et pour mettre à jour les variables au niveau de chaque lame.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme de Reconstruction par ordre circulaire ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'espèces	Temps moyen d'exécution (sec)
1	8	0.0051
1	16	0.0720
1	32	1.0000
1	64	14.8000
2	8	0.0041
2	16	0.0410
2	32	0.5300
2	64	7.5000
4	8	0.0029
4	16	0.0230
4	32	0.2900
4	64	3.9000
8	8	0.0027
8	16	0.0150
8	32	0.1600
8	64	2.1000
16	8	0.0026
16	16	0.0110
16	32	0.0980
16	64	1.3000

Tableau IV-4 Algorithme de reconstruction par ordre circuclaire : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

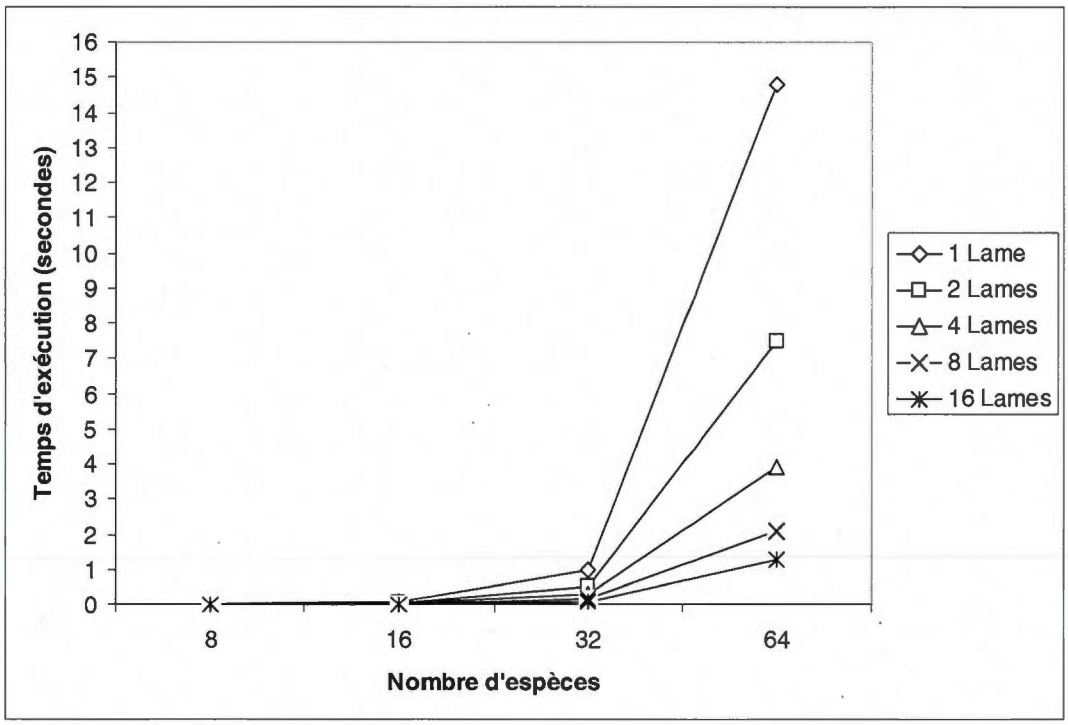


Figure IV-6 Algorithme de reconstruction par ordre circulaire : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Nous remarquons qu'en général, l'algorithme parallèle devient plus rapide que l'algorithme séquentiel à partir de 8 espèces, quel que soit le nombre de lames utilisées.

4.7 Moindres carrés pondérés

Le principe de la méthode des moindres carrés pondérés (MW) est de trouver un arbre phylogénétique qui minimise le critère des moindres carrés.

Étant donné que :

- d_{ij} est la dissimilarité donnée sur un ensemble fini d'éléments X ;
- T représente l'arbre valué ayant X comme ensemble de feuilles
- δ_{ij} représente les longueurs des chemins connectant les feuilles de l'arbre T.

Le critère des moindres carrées se présente comme suit :

$$Q = \sum_{1 \le i \le j \le n} w_{ij} \left(d_{ij} - \delta_{ij} \right)^2 \rightarrow MIN \ ;$$

où w_{ij} est le poids appliqué pour la séparation des éléments *i* et *j*.

Le critère des moindres carrés pondérés doit être minimisé pour l'ensemble des arbres valués. Ce critère peut être appliqué à plusieurs situations pratiques. Selon Swofford et Olsen (1990), il y'a principalement quatre arrangements de poids qui sont fréquemment utilisés dans le domaine :

$$w_{ij} = 1, w_{ij} = \frac{1}{d_{ij}}, w_{ij} = \frac{1}{d_{ij}^2}, w_{ij} = \frac{1}{\sigma_{ij}^2}$$

où σ_{ij} est la variance escomptée de la dissimilarité d_{ij}.

Étant donné *D* une matrice symétrique de dimension (*n* x *n*) sur un ensemble de *n* éléments $Y = \{y_1, y_2, ..., y_n\}$, *W* sera une matrice symétrique de poids.

L'algorithme des moindres carrés pondérés peut être divisé en deux étapes (Makarenkov et Leclerc, 1999) :

La première étape de l'algorithme consiste à considérer deux éléments y_i et y_j de l'ensemble *Y*, de telle sorte que $d(y_i y_j)$ soit la plus petite valeur positive dans la matrice *D*. Soit $x_1 = y_i$; soit $x_2 = y_j$; soit $X = \{x_1, x_2\}$; L'arbre T^2 se résume à une branche unique $x_1 x_2$ de longueur $d(x_1 x_2)$.

A l'étape *k*, T^k est l'arbre courant qui résulte des constructions précédentes. Cet arbre contient *k* feuilles qui correspondent à l'ensemble *X* qui est un sous-ensemble de *Y*. À cette étape, il faut sélectionner, à partir des *n-k* éléments de $Y \setminus X$ (ensemble *Y* privé de *X*), l'élément le plus approprié selon le critère des moindres carrés. Pour y arriver, nous ajoutons

une nouvelle feuille x_{k+1} à l'arbre T^k en considérant toutes les connections possibles de tous les éléments de l'ensemble $Y \setminus X$ sur toutes les branches de T^k.

La méthode MW a une complexité algorithmique de $O(n^5)$.

Pour MW, le parallélisme de contrôle a été réalisé au niveau de la sélection des éléments qui minimisent le critère des moindres carrés. A cette étape, les données sont subdivisées en fonction du nombre de lames qui exécutent le programme. En fonction de son flot de données, chaque lame cherche l'élément qui maximise le critère des moindres carrés. Puis, une synchronisation est effectuée afin de choisir l'élément qui maximise le critère choisi au niveau de toutes les lames et pour mettre à jour les variables au niveau de chaque lame.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme des moindres carrés pondérés (MW) ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'espèces	Temps moyen d'exécution (sec)
1	8	0.007000
1	16	0.190000
1	32	5.900000
2	8	0.006000
2	16	0.115000
2	32	3.240000
4	8	0.004700
4	16	0.076000
4	32	1.920000
8	8	0.004400
8	16	0.058000
8	32	1.240000
16	8	0.004400

16	16	0.049000
16	32	0.910000

Tableau IV-5 MW : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

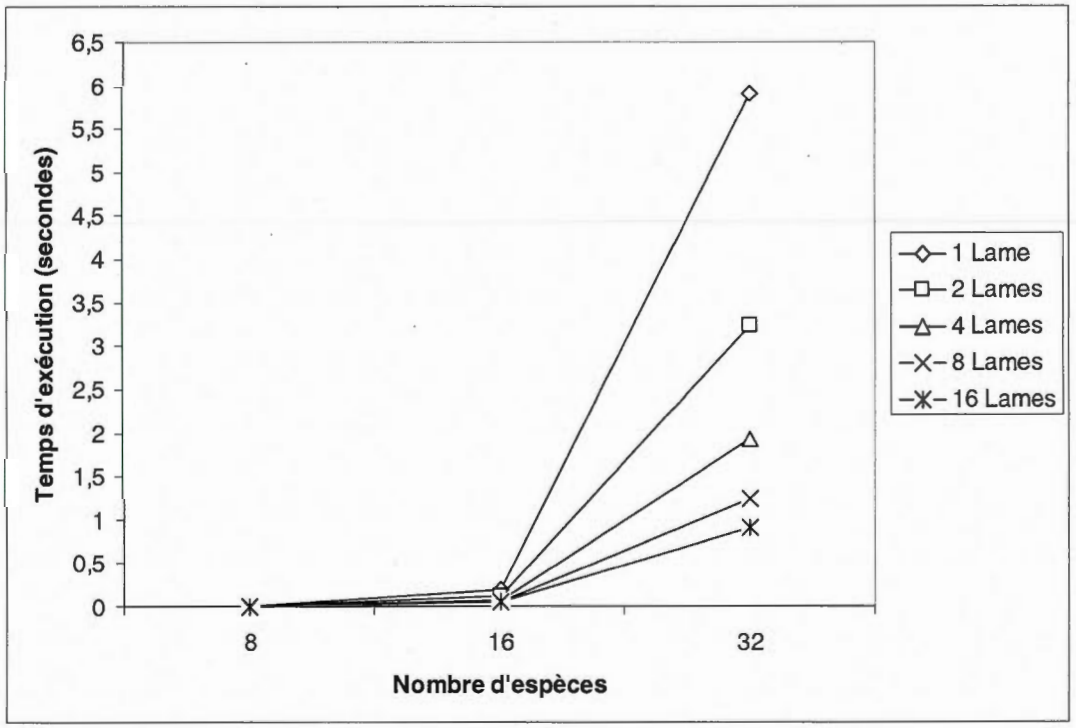


Figure IV-7 Algorithme MW : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Pour l'algorithme MW, la version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées. Pour un nombre d'espèces inférieur à 16, on peut utiliser un nombre de lames égal au nombre d'espèces, alors que pour un nombre d'espèces supérieur à 16, on peut utiliser les 16 lames au complet pour minimiser le temps d'exécution.

4.8 BioNJ

BioNJ (Gascuel, 1997) est un algorithme de reconstruction phylogénétique basé sur les distances. Cet algorithme est une version améliorée de la méthode de Neighbor-Joining (Saitou et Nei, 1987). Cet algorithme modifie Neighbor-Joining en considérant les variances et les covariances entre les distances dans un modèle d'évolution simple. Ces variances et covariances sont proportionnelles aux longueurs des branches. BioNJ est bien adapté pour les distances estimées à partir de séquences d'ADN ou de protéines.

BioNJ, suit le même arrangement agglomératif que NJ, qui consiste à sélectionner itérativement une paire de taxons, créer un nouveau nœud qui représente le regroupement de ces taxons et réduire la matrice de distances en remplaçant les deux taxons par un nouveau nœud. Le modèle de BioNJ est bien adapté quand les évaluations sont obtenues à partir des ordres alignés. À chaque étape, il permet le choix, à partir de la classe des réductions possibles, de la réduction qui minimise le désaccord de la nouvelle matrice de distances. De cette façon, nous obtenons de meilleures évaluations pour choisir la paire de taxons à agglomérer pendant étapes suivantes.

BioNJ infère des topologies plus précises que NJ dans toutes les conditions d'évolution, surtout quand les taux de substitutions sont élevés et changent entre les lignées. La méthode BioNJ une complexité de algorithmique de $O(n^3)$.

Tout comme pour NJ, le parallélisme de contrôle a été effectué au niveau de l'étape de la sélection de la paire d'éléments qui minimise la distance. A cette étape, les données sont divisées en fonction du nombre de lames qui exécutent le programme. En fonction du flot de données qui lui est dédié, chaque lame cherche la paire (i et j) qui minimise la distance. Finalement, une synchronisation est effectuée afin de choisir la paire qui minimise la distance au niveau de toutes les lames et pour mettre à jour les variables au niveau de chaque lame.

Les résultats de la parallélisation en termes de gain de temps sont identiques à la méthode Neighbor-Joining et à partir de 450 espèces la version parallèle sur 2 lames est plus rapide

que la version séquentielle. Cela s'explique par le fait que BioNJ est une adaptation de l'algorithme Neighbor-Joining ayant la même complexité $O(n^3)$.

4.9 FITCH

L'algorithme Fitch (1971) a été conçu pour compter le nombre de changements dans un arbre binaire avec des séquences de nucléotides, dans lequel chacune des quatre bases (A, C, G, T) peut être transformée en toute autre. Il fonctionne pour tout nombre d'états, sachant que tout état peut changer en un autre. L'algorithme Fitch fonctionne bien pour un changement d'état $0 \Leftrightarrow 1$.

L'algorithme Fitch considère les sites (ou caractères) un par un. À chaque bout de l'arbre, il crée un ensemble qui contient les nucléotides qui ont été observés ou qui sont compatibles avec l'observation. Ainsi, si on observe un A par exemple, on crée l'ensemble $\{A\}$ et on descend au niveau de l'arbre.

En terme algorithmique, Fitch effectue un parcours postordonné de l'arbre. À chaque nœud interne, il crée un ensemble qui est l'intersection des ensembles de deux nœuds descendants (Figure IV-8). Si cet ensemble est vide, alors l'algorithme crée un ensemble qui est l'union des ensembles des deux nœuds descendants. Chaque fois que l'ensemble créé est une union, il compte un changement d'état (i.e., l'algorithme incrémente le nombre de changements d'états).

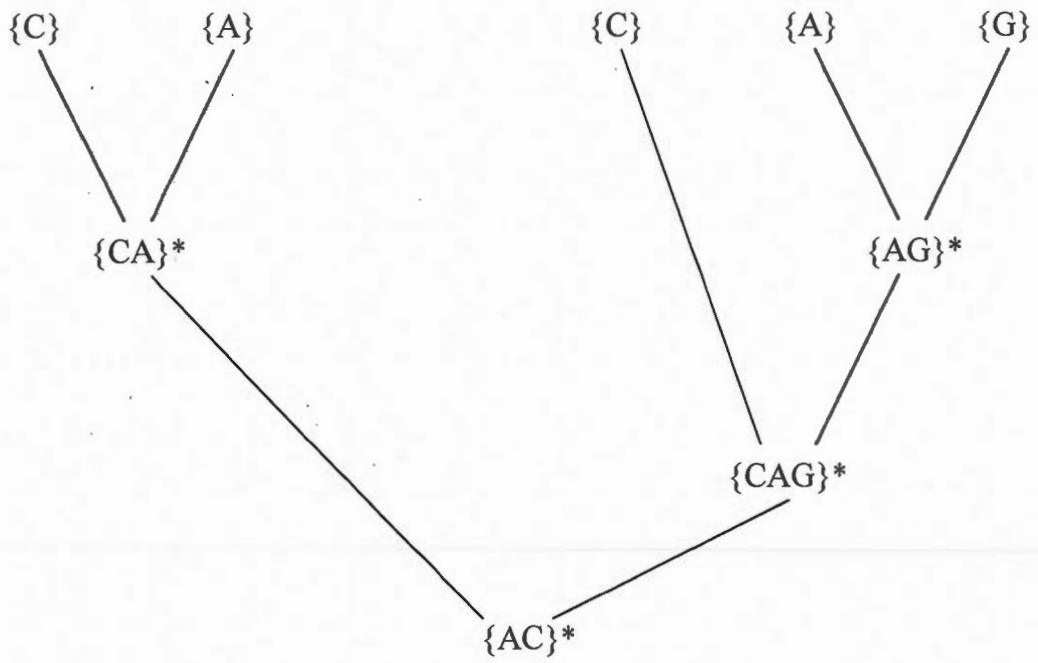


Figure IV-8 Reconstruction par la méthode Fitch.

La figure IV-8, montre un arbre à cinq espèces. Mettons qu'au niveau d'un site particulier, nous ayons observé les bases C, A, C, A et G chez les cinq espèces. Ces bases sont ordonnées en fonction de l'ordre dans lequel les espèces apparaissent au niveau de l'arbre (de gauche à droite).

Tout d'abord, nous considérons les deux premières espèces de gauche au niveau du nœud qui est leur ancêtre commun immédiat. Étant donné que l'intersection des deux ensembles est vide ($\{C\} \cap \{A\} = \emptyset$), nous considérons plutôt leur union ($\{C\} \cup \{A\} = \{AC\}$) et comptons un changement d'état. Pour les deux espèces à droite ($\{A\}$ et $\{G\}$) pour les mêmes raisons que précédemment, leur ancêtre commun immédiat sera ($\{A\} \cap \{G\} = \{AG\}$) et nous comptons un autre changement d'état. En procédant de la même manière, nous comptons trois changements d'états et nous aurons un ancêtre commun qui est l'ensemble ($\{AC\}$).

L'algorithme Fitch peut être réalisé en un nombre d'opérations qui est proportionnel au nombre d'espèces de l'arbre.

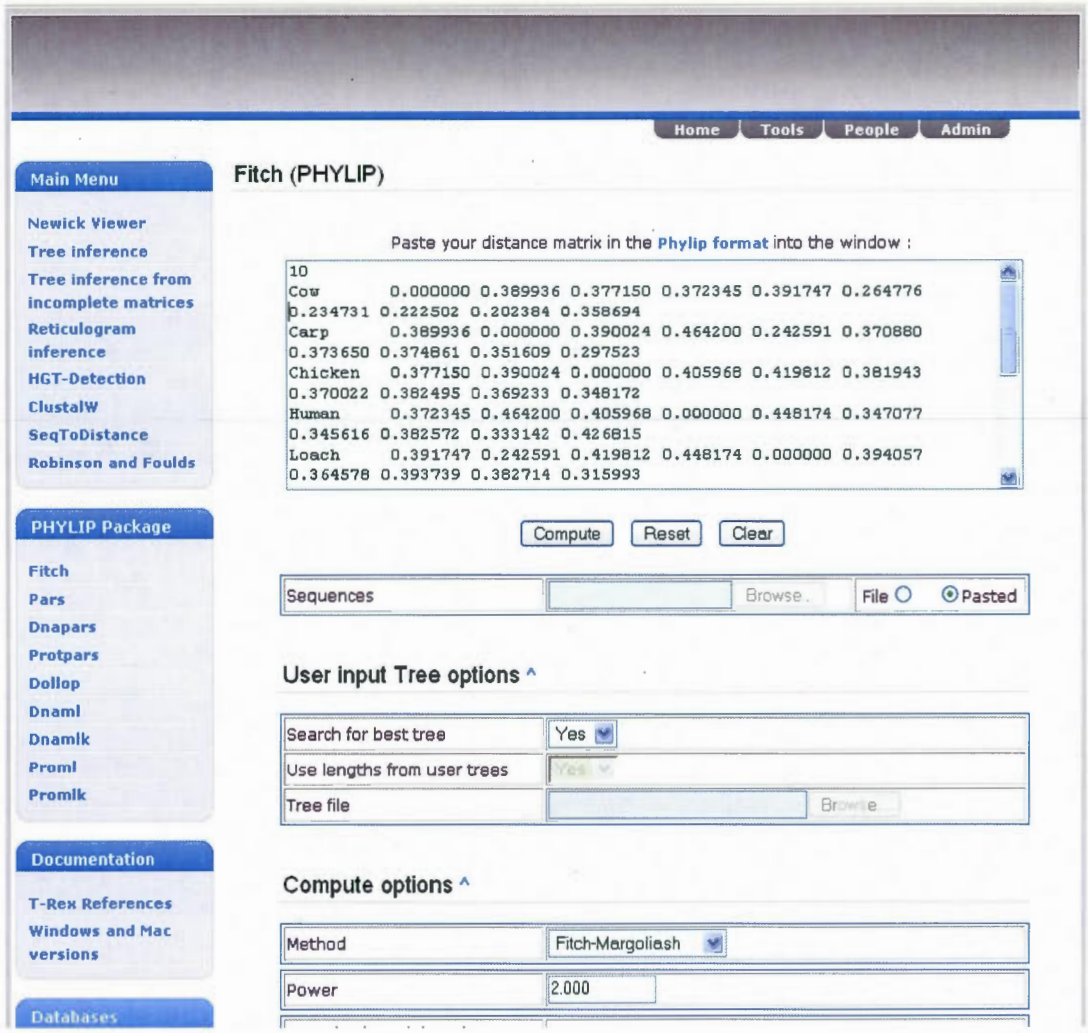


Figure IV-9 Interface de T-Rex avec la méthode Fitch.

Au niveau de la parallélisation de l'algorithme Fitch, un parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Prenant en compte le nombre d'arbres à inférer, ils sont divisés de manière équitable en fonction du nombre de lames du cluster qui exécutent le programme. Les arbres produits

seront localement contenus sur chaque lame. Le passage des données de ces arbres entre les différentes lames étant vraiment coûteux en termes de temps, un script de regroupement et de synchronisation des résultats a été implémenté. Le principe de ce script est simple. Il consiste à chercher la lame qui a trouvé le meilleur arbre et de copier les résultats de cette lame sur la lame principale (*myid* = 0).

Le même principe de parallélisme est appliqué en fonction du nombre d'ensembles de données. Le script regroupement et de synchronisation consiste cette fois à copier tous les résultats des lames au niveau de la lame principale (*myid* = 0). Finalement, ces résultats sont concaténés en tenant compte de l'ordre des ensembles de données.

Des simulations effectuées pour calculer le gain de temps obtenu, (en fonction d'ensembles de données) suite à la parallélisation de l'algorithme Fitch ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre de d'ensembles de données	Temps moyen d'exécution (sec)
1	1	0.012000
1	8	0.100000
1	16	0.195000
1	32	0.390000
2	1	0.012000
2	8	0.045000
2	16	0.087500
2	32	0.180000
4	1	0.012000
4	8	0.026000
4	16	0.052000
4	32	0.100000
8	1	0.012000
8	8	0.014000

8	16	0.027200
8	32	0.052200
16	1	0.012000
16	8	0.140000
16	16	0.140000
16	32	0.027000

Tableau IV-6 Algorithme Fitch : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

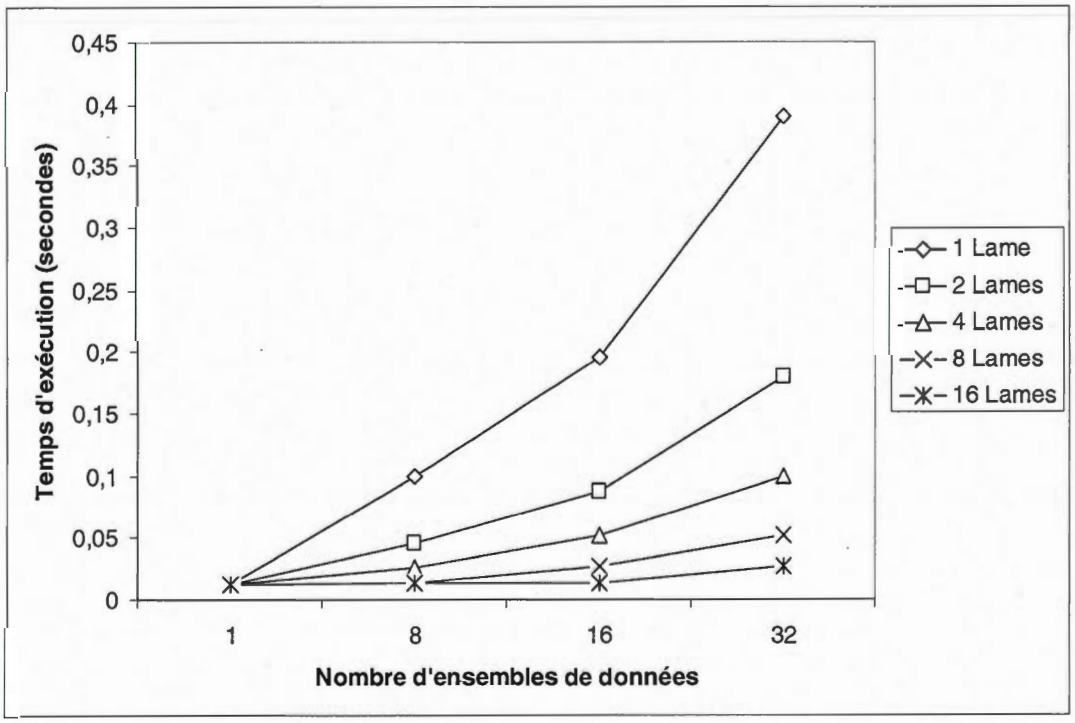


Figure IV-10 Algorithme Fitch : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données. Cela s'explique par le fait que les lames infèrent les arbres

sensiblement à la même vitesse. Par conséquent, le temps d'exécution reste presque invariable en doublant simultanément le nombre d'arbres à inférer et le nombre de lames utilisées.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement, on peut utiliser les 16 lames au complet pour minimiser le temps d'exécution.

4.10 Références

Felsenstein, J. (1993). PHYLIP (PHYLogeny Inference Package) version 3.6a2, *Distributed by the author, Department of Genetics, University of Washington, Seattle, WA* .

Fitch, W. M. (1971). Towards defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology* **20**: 406-416.

Gascuel, O. (1997). The NJ algorithm and its unweighted version UNJ, *In Mathematical hierarchies and Biology (B. Mirkin et al., eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **37**, *Amer. Math. Soc., Providence, RI* : 149170.

Gascuel, O. (1997). BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data. *Molecular Biology and Evolution* **14** : 685-695.

Makarenkov, V. et Leclerc, B. (1997). Circular orders of tree metrics, and their uses for the reconstruction and fitting of phylogenetic trees, *Mathematical Hierarchies and Biology (B. Mirkin, F.R. McMorris, F. Roberts, A. Rzhetsky, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Amer. Math. Soc., Providence, RI,* 183-208.

Makarenkov, V. et Leclerc, B. (1999). An algorithm for the fitting of a phylogenetic tree according to a weighted least-squares criterion, *Journal of Classification* **16 (1)**: 3-26.

Makarenkov, V. (2001). T-REX: reconstructing and visualizing phylogenetic trees and reticulation networks, *Bioinformatics* **17 (7)**: 664-668.

Saitou, N. et Nei, M. (1987). The neighbor-joining method: a new method for reconstructing phylogenetic trees, *Molecular Biology and Evolution* **4 (4)** : 406-25.

Sattath, S. et Tversky, A. (1977). Additive similarity trees, *Psychometrika* **42 (3)**: 319-345.

Swofford, D.L. et Olsen, G.J. (1990). Phylogeny construction. In Molecular Systematics, Hillis, *D.M. and Moritz, C., eds* : 411-501, Sinauer Associates.

CHAPITRE V

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT LA MÉTHODE DE MAXIMUM DE VRAISEMBLANCE

5.1 Introduction

Les programmes utilisant la méthode de maximum de vraisemblance qui ont été parallélisés sont principalement ceux inclus dans le package PHYLIP (Felsenstein, 1993). Il s'agit de : DNAML, DNAMLK, PROML et PROMLK. De plus, le populaire programme PHYML (Guindon, 2003) a également été parallélisé.

La figure V-1 montre l'interface du serveur web de T-Rex pour les cinq algorithmes (parallélisés) utilisant la méthode de maximum de vraisemblance.

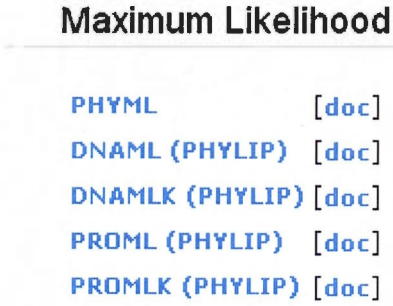


Figure V-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de maximum de vraisemblance.

5.2 Forme de parallélisme employée

Le point commun des tous ces cinq algorithmes réside dans l'application des mêmes opérations sur des ensembles disjoints de données. Par conséquent, nous avons utilisé le parallélisme de données (voir chapitre II, formes de parallélisme).

Le parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Prenant en compte le nombre d'arbres à inférer, ils sont divisés de manière équitable en fonction du nombre de lames du cluster qui exécutent le programme. Les arbres produits seront localement contenus sur chaque lame. Le passage des données de ces arbres entre les différentes lames étant vraiment coûteux en termes de temps, un script de regroupement et de synchronisation des résultats a été implémenté. Le principe de ce script est simple. Il consiste à chercher la lame qui a trouvé le meilleur arbre et de copier les résultats de cette lame sur la lame principale (*myid* = 0).

Le même principe de parallélisme est appliqué en fonction du nombre d'ensembles de données. Le script de regroupement et de synchronisation consiste cette fois à copier tous les résultats des lames au niveau de la lame principale (*myid* = 0). Finalement, ces résultats sont concaténés en tenant compte de l'ordre des ensembles de données.

5.3 PHYML

PHYML (Guindon, 2003) est un logiciel qui implémente une nouvelle méthode de reconstruction de phylogénie à partir de séquences en utilisant le principe de maximum de vraisemblance. Cette méthode démarre avec un arbre initial fourni par l'utilisateur ou construit à partir d'un algorithme rapide basé sur les distances, puis elle améliore cet arbre à travers des réarrangements topologiques. La méthode peut être décomposée en six étapes :

1. Une matrice de distance est calculée en fonction des séquences reçues en entrée avec un algorithme analogue à DNADIST (Felsenstein 1993). Cette étape nécessite une comparaison de toutes les paires de séquences et nécessite un temps d'exécution $O(n^2)$, où n représente le nombre de taxons.
2. Un arbre initial est construit à partir de la matrice de distance de l'étape 1 en utilisant la méthode BioNJ (Gascuel 1997). Cette étape nécessite un temps d'exécution de $O(n^3)$.
3. La vraisemblance est calculée au niveau de tous les sites et de tous les sous-arbres pour obtenir la vraisemblance de l'ensemble de l'arbre.
4. Les valeurs des paramètres libres du modèle de substitution (taux de transition, taux de transversion, etc.) sont ajustées de manière à améliorer la vraisemblance de l'arbre de départ.
5. L'arbre courant est raffiné itérativement jusqu'à la convergence.
6. Le raffinement s'arrête quand il n'y a plus de permutations possibles et quand les longueurs de branches deviennent stables. L'arbre courant est retourné par la méthode.

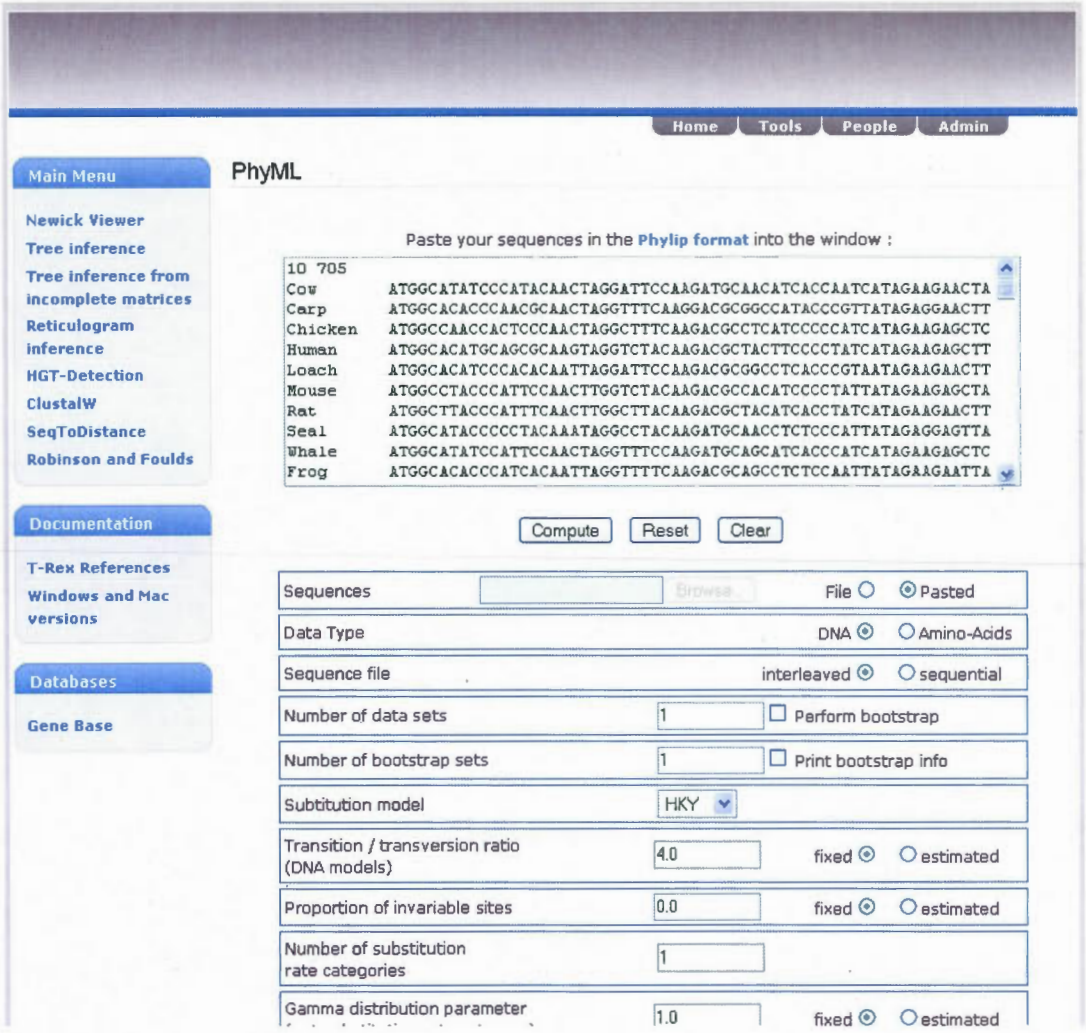


Figure V-2 Interface de T-Rex avec la méthode PHYML.

Au niveau de la parallélisation du programme PHYML, un parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme PHYML ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	1.350000
1	8	7.500000
1	16	27.00000
1	32	55.80000
2	1	1.350000
2	8	10.650000
2	16	20.80000
2	32	42.00000
4	1	1.350000
4	8	7.400000
4	16	15.50000
4	32	30.50000
8	1	1.350000
8	8	5.750000
8	16	11.20000
8	32	23.40000
16	1	1.350000
16	8	5.750000
16	16	5.80000
16	32	12.00000

Tableau V-1 Algorithme PHYML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

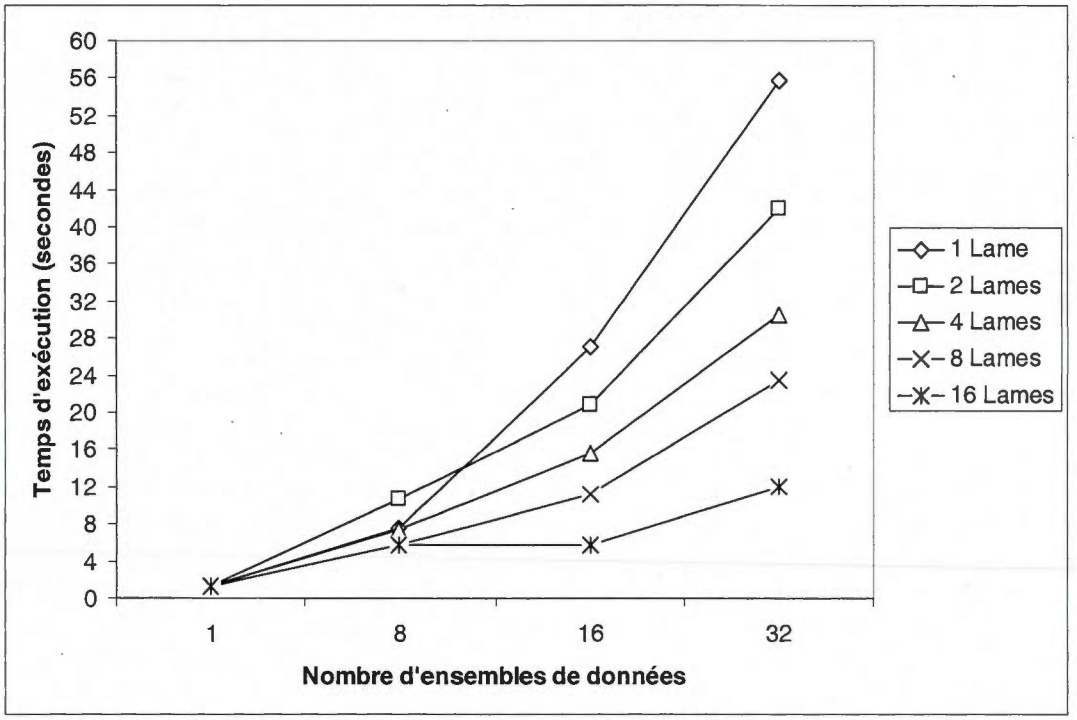


Figure V-3 Algorithme PHYML : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement, on peut utiliser les 16 lames au complet pour minimiser le temps d'exécution.

5.4 DNAML

DNAML (Felsenstein, 1993) est un programme d'inférence d'arbres phylogénétiques, pour les séquences d'ADN basé sur le maximum de vraisemblance. Le modèle de substitution des

bases permet d'avoir des fréquences différentes au niveau des quatre bases, des transversions et des transitions. Ce modèle possède différentes manières pour permettre des taux d'évolution différents au niveau de sites différents.

Les suppositions faites pour ce modèle sont les suivantes :

- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- Chaque site subit une substitution à un taux qui est choisi parmi une série de taux spécifiés;
- Tous les sites appropriés sont inclus dans la séquence, pas seulement ceux qui subissent des changements ou ceux qui ont une information phylogénétique;
- Une substitution comprend les deux événements suivants :
    - Le premier événement consiste à remplacer la base par une du lot des bases de purine ou une du lot des bases de pyrimidine pour aboutir soit à une transition, soit à aucun changement;
    - Le seçond évènement consiste à remplacer la base par une base aléatoire d'un lot des bases à fréquences connues, indépendamment de l'identité de la base qui a été remplacée, pour aboutir soit à une transversion, soit à une transition, soit à aucun changement.

Ce programme utilise un modèle de Markov caché pour inférer différents taux d'évolution pour différents sites. Ceci permet de spécifier au programme : la possibilité de plusieurs taux d'évolution, la probabilité *a priori* de chacun, la longueur moyenne du chemin des sites ayant tous le même taux. Les taux peuvent aussi être choisis par le programme pour approximer une distribution Gamma des taux ou une distribution Gamma avec une classe de sites invariants.

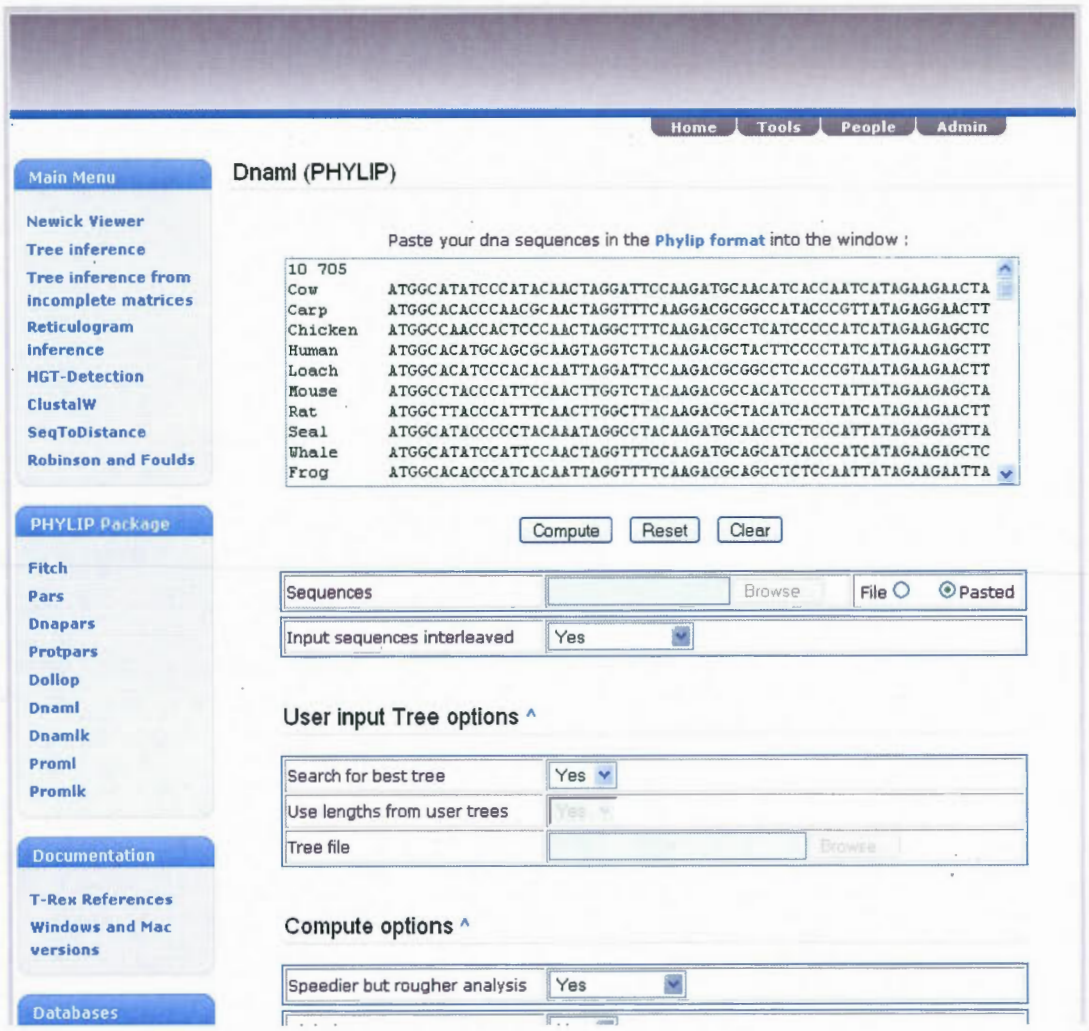


Figure V-4 Interface de T-Rex avec la méthode DNAML.

Au niveau de la parallélisation du programme DNAML, un parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme DNAML ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	1.550000
1	8	12.30000
1	16	26.50000
1	32	56.50000
2	1	1.550000
2	8	6.100000
2	16	18.20000
2	32	24.90000
4	1	1.550000
4	8	3.150000
4	16	8.250000
4	32	12.70000
8	1	1.550000
8	8	1.650000
8	16	3.200000
8	32	6.400000
16	1	1.550000
16	8	1.650000
16	16	1.700000
16	32	3.300000

Tableau V-2 Algorithme DNAML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

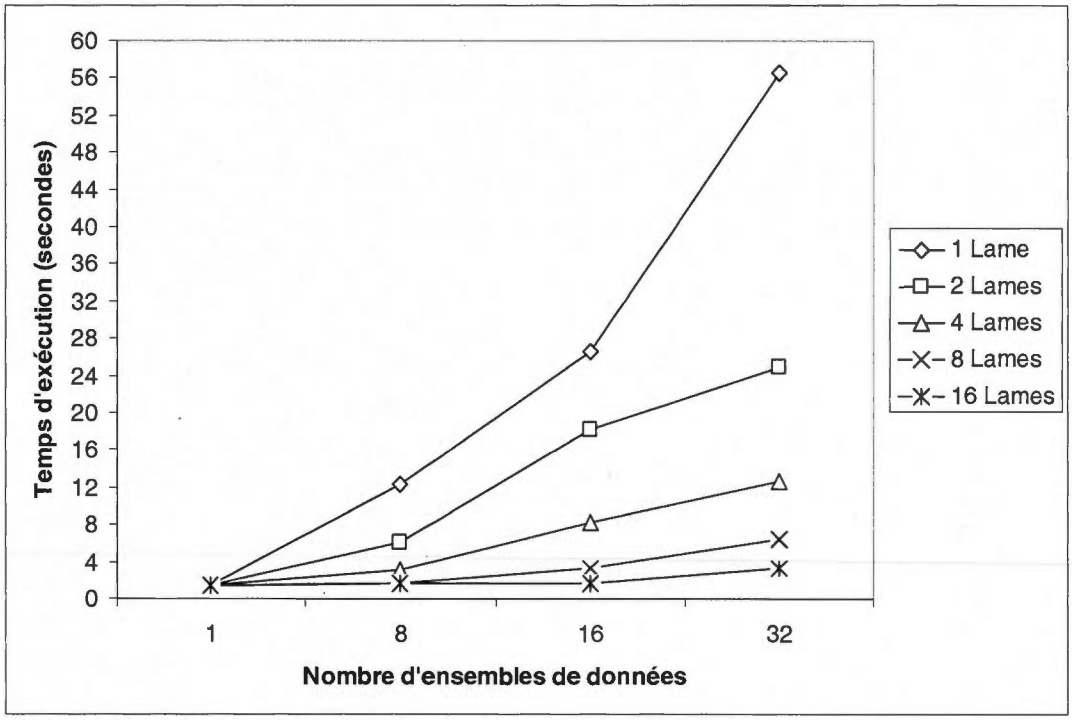


Figure V-5 Algorithme DNAML : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement on peut utiliser les 16 lames pour minimiser le temps d'exécution.

5.5 DnaMLK

DNAMLK (Felsenstein, 1993) est un programme d'inférence d'arbres phylogénétiques, pour les séquences d'ADN, basé sur le maximum de vraisemblance avec la contrainte que les arbres inférés soient en conformité avec l'horloge moléculaire. L'horloge moléculaire stipule

que toutes les feuilles de l'arbre sont équidistantes par rapport à la racine (au niveau de la longueur des branches). C'est un programme connexe à DnaML. Les mêmes suppositions au niveau du modèle sont valides (plus l'hypothèse de l'horloge moléculaire). Le programme utilise également un modèle de Markov caché pour inférer différents taux d'évolution pour des sites différents.

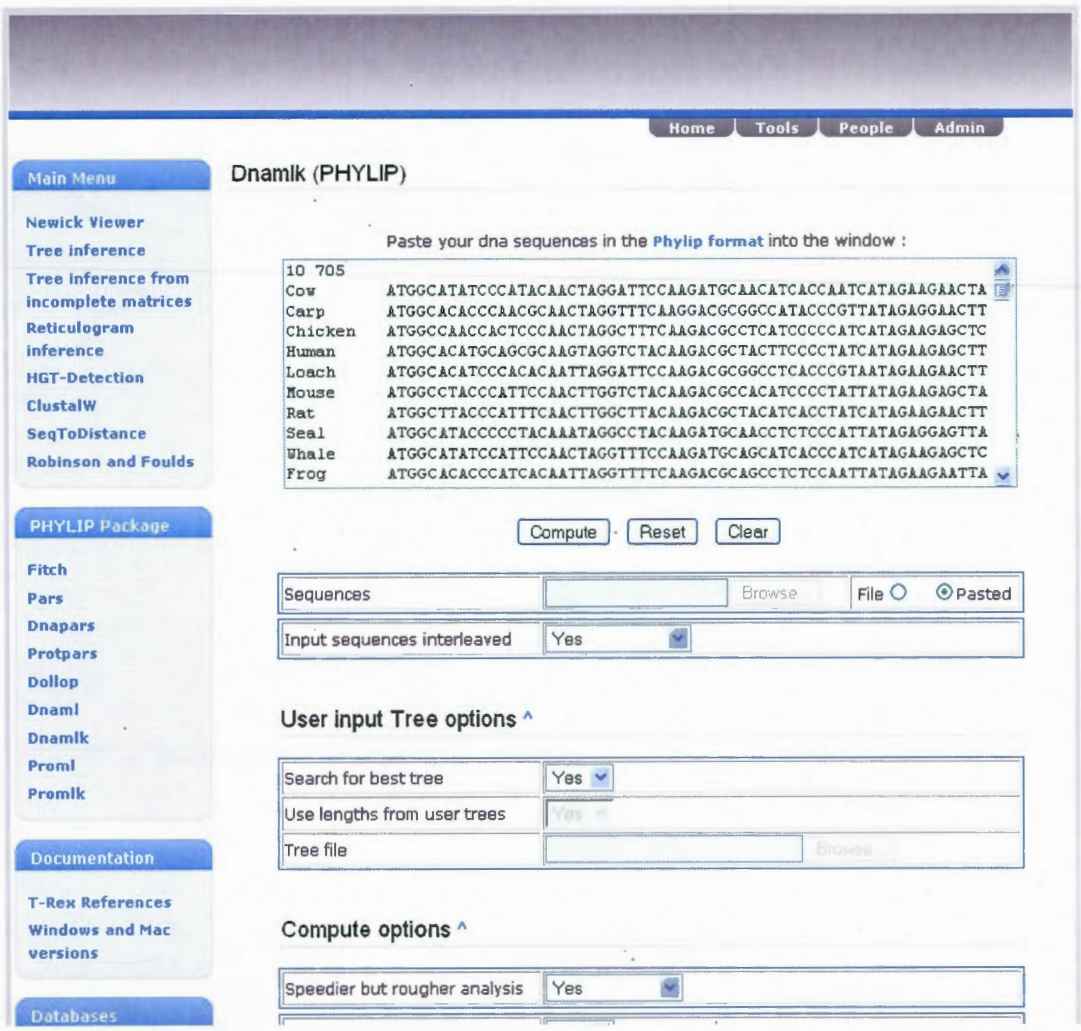


Figure V-6 Interface de T-Rex avec la méthode DNAMLK.

Au niveau de la parallélisation du programme DNAMLK, un parallélisme de données a été réalisé en fonction du nombre de d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme DNAMLK ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	4.000000
1	8	31.20000
1	16	62.50000
1	32	125.6000
2	1	4.000000
2	8	15.50000
2	16	31.25000
2	32	76.00000
4	1	4.000000
4	8	7.900000
4	16	16.00000
4	32	33.20000
8	1	4.000000
8	8	4.200000
8	16	7.950000
8	32	16.50000
16	1	4.000000
16	8	4.200000
16	16	4.200000
16	32	8.000000

Tableau V-3 Algorithme DNAMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

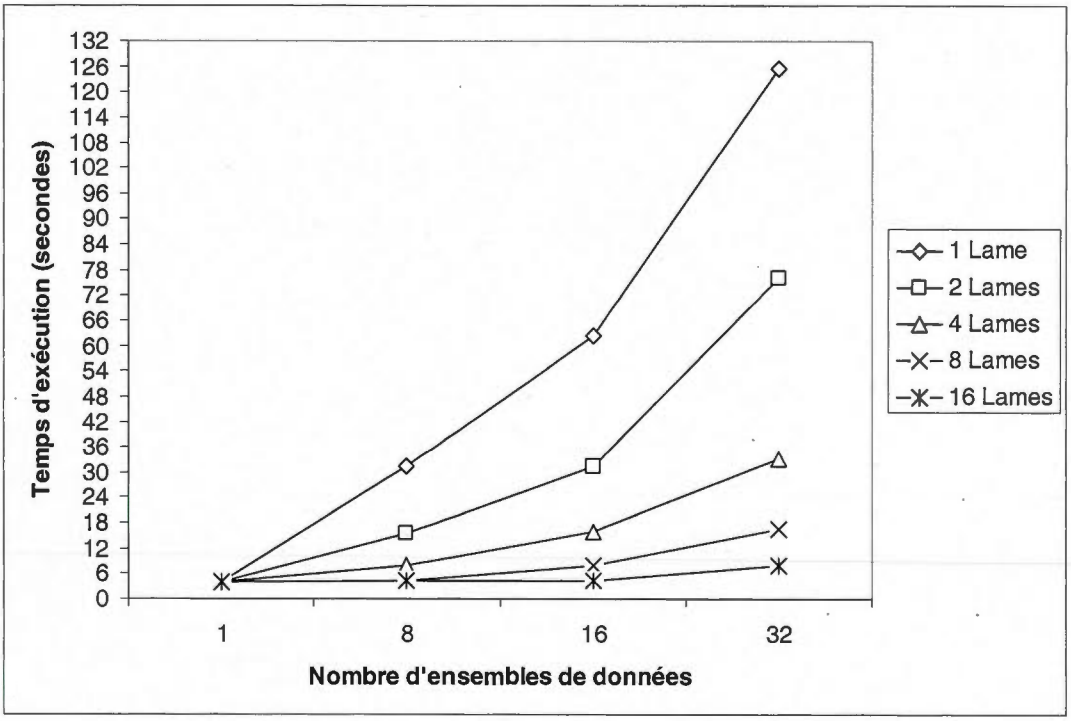


Figure V-7 Algorithme DNAMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement on peut utiliser les 16 lames pour minimiser le temps d'exécution.

5.6 ProML

ProML (Felsenstein, 1993) est un programme d'inférence d'arbres phylogénétiques, pour les séquences de protéines (acides aminés), basé sur le maximum de vraisemblance. Ce programme utilise le modèle probabiliste de Jones-Taylor-Thornton (1992) de même que

celui de Dayhoof (Dayhoff et al., 1979) pour effectuer les substitutions entre les acides aminés.

Les suppositions nécessaires pour ce modèle sont les suivantes :

- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- Chaque site subit une substitution à un taux qui est choisi parmi une série de taux spécifiés;
- Tous les sites appropriés sont inclus dans la séquence, pas seulement ceux qui subissent des changements ou ceux qui ont une information phylogénétique;
- La probabilité de changement entre les acides aminés est donnée par le modèle de Jones, Taylor et Thornton (1992) ou par le modèle PAM de Dayhoff (Dayhoff et al., 1979).

Tout comme les programmes DnaML et DnaMLK, ce programme utilise un modèle de Markov caché pour inférer différents taux d'évolution pour des sites différents.

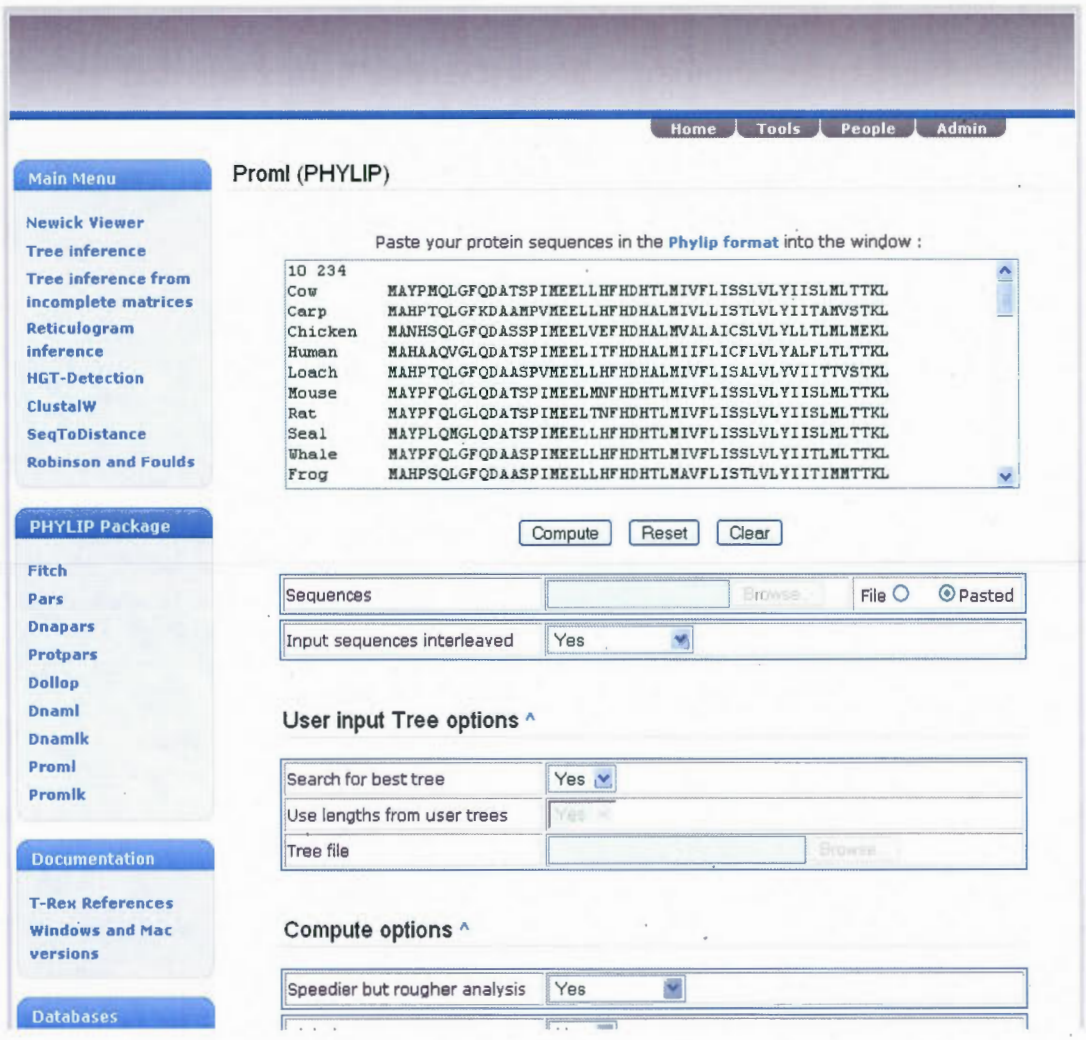


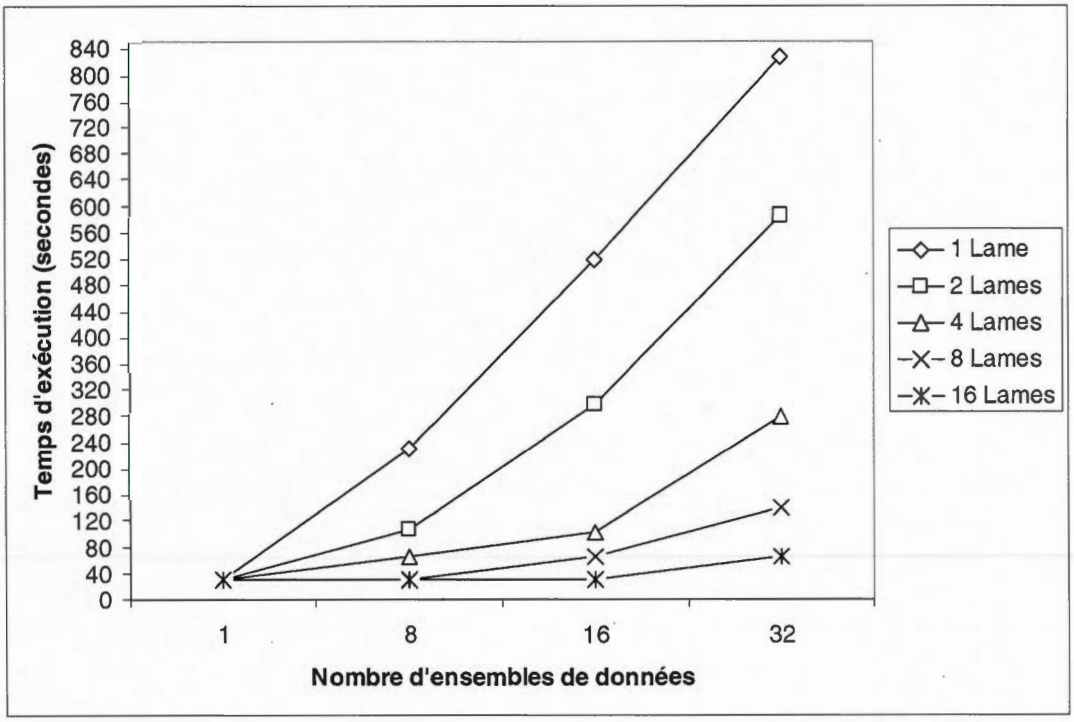
Figure V-8 Interface de T-Rex avec la méthode PROML.

Au niveau de la parallélisation du programme PROML, un parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme PROML ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	30.00000
1	8	230.0000
1	16	517.5000
1	32	827.0000
2	1	30.00000
2	8	107.5000
2	16	298.0000
2	32	585.5000
4	1	30.00000
4	8	65.75000
4	16	101.5000
4	32	278.0000
8	1	30.00000
8	8	31.00000
8	16	65.00000
8	32	140.00000
16	1	30.00000
16	8	31.00000
16	16	31.00000
16	32	64.00000

Tableau V-4 Algorithme PROML : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.



V-9 Algorithme PROML : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement on peut utiliser les 16 lames pour minimiser le temps d'exécution.

5.7 ProMLK

ProMLK (Felsenstein, 1993) est un programme d'inférence d'arbres phylogénétiques pour les séquences de protéines (acides aminés), basé sur le maximum de vraisemblance avec la contrainte que les arbres inférés soient en conformité avec l'hypothèse de l'horloge

moléculaire. C'est un programme connexe à ProML. Les mêmes suppositions au niveau du modèle sont valides (plus l'hypothèse de l'horloge moléculaire). Ce programme utilise également un modèle de Markov caché pour inférer différents taux d'évolution pour des sites différents.

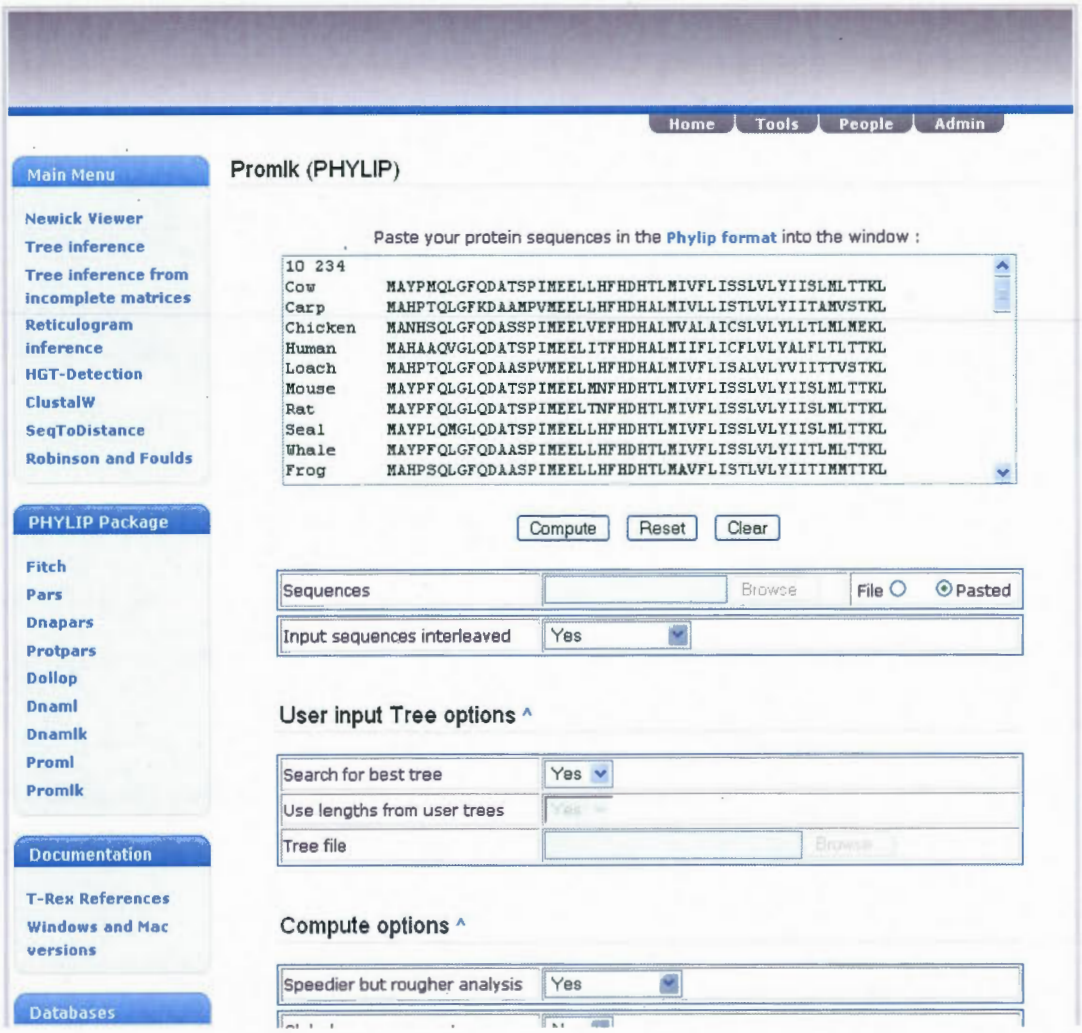


Figure V-10 Interface de T-Rex avec la méthode PROMLK.

Au niveau de la parallélisation du programme PROMLK, un parallélisme de données a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme PROMLK ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	60.000000
1	8	490.00000
1	16	1118.0000
1	32	2005.0000
2	1	60.000000
2	8	210.00000
2	16	390.50000
2	32	1095.0000
4	1	60.000000
4	8	140.00000
4	16	280.50000
4	32	490.00000
8	1	60.000000
8	8	71.500000
8	16	139.50000
8	32	210.00000
16	1	60.000000
16	8	62.000000
16	16	62.000000
16	32	112.00000

Tableau V-5 Algorithme PROMLK: Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

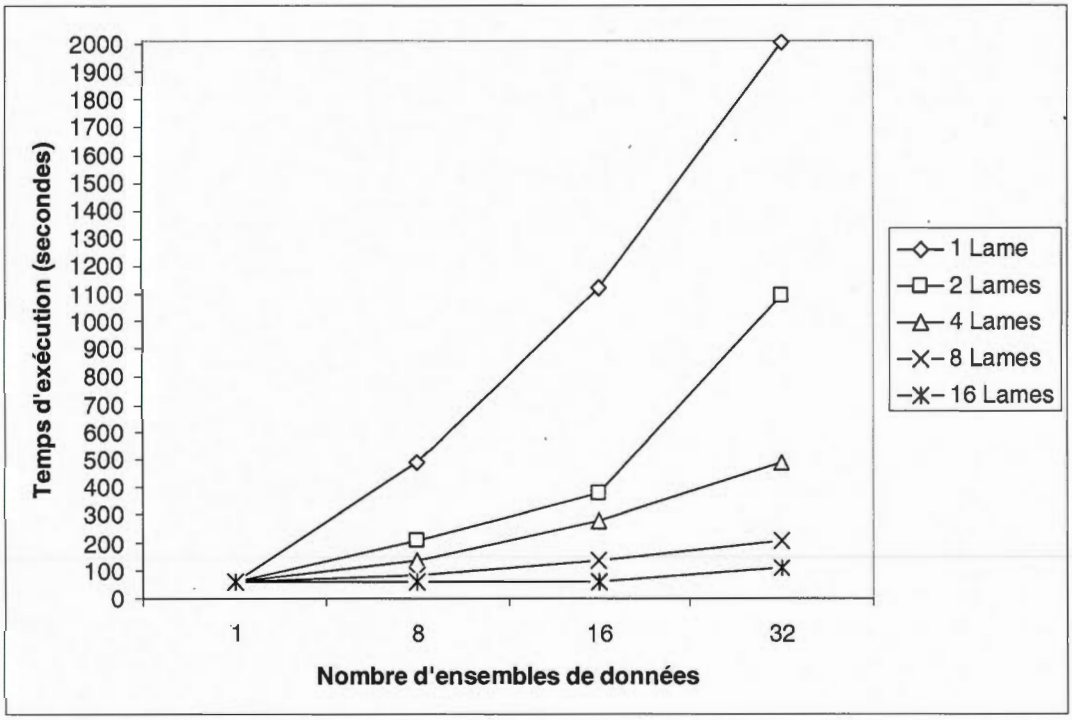


Figure V-11 Algorithme PROMLK : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement, on peut utiliser les 16 lames pour minimiser le temps d'exécution.

5.8 Références

Dayhoff, M. et al. (1979). Matrices for detecting distant relationships. *In Dayhoff, M., editor, Atlas of protein sequences*: 353 - 358. Natl. Biomed. Res. Found.

Felsenstein, J. (1993). PHYLIP (PHYLogeny Inference Package) version 3.6a2, Distributed by the author, Department of Genetics, University of Washington, Seattle, WA .

Guindon, S. et Gascuel, O. (2003). A simple, fast and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biolology*.**52** : 696-704.

Jones, D. T., Taylor, W. R. et Thornton, J. M. (1992). A new approach to protein fold recognition. *Nature*: 358, 86-89.

CHAPITRE VI

PARALLÉLISATION DES PROGRAMMES D'INFÉRENCE D'ARBRES PHYLOGÉNÉTIQUES UTILISANT LA MÉTHODE DE MAXIMUM DE PARCIMONIE

6.1 Introduction

Les programmes utilisant la méthode de maximum de parcimonie qui ont été parallélisés sont ceux inclus dans le package PHYLIP (Felsenstein, 1993). Il s'agit de : DNAPARS, PARS et DOLLOP.

La figure VI-1 montre l'interface du serveur web de T-Rex pour les trois algorithmes (parallélisés) utilisant la méthode de maximum de parcimonie.

Figure VI-1 Interface de T-Rex pour les programmes d'inférence d'arbres phylogénétiques utilisant la méthode de maximum de parcimonie.

6.2 DNAPARS

Ce programme infère un arbre parcimonieux non enraciné. La méthode de Fitch (1971) est utilisée pour compter des changements de bases nécessaires pour un arbre donné.

Les suppositions faites avec ce modèle sont les suivantes :

- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- La probabilité de substitution d'une base au niveau d'un site donné est petite par rapport à la durée associée à la longueur d'une branche de l'arbre phylogénétique;
- Les quantités escomptées de changement au niveau des différentes branches de la phylogénie ne varient pas beaucoup (deux changements sur une branche à haut taux sont plus probables qu'un changement sur une branche à bas taux);
- Les quantités escomptées de changement au niveau des différents sites de la phylogénie ne varient pas beaucoup (deux changements au niveau d'un site sont plus probables qu'un changement au niveau d'un autre site).

DNAPARS traite aussi bien les arbres bifurqués (i.e, binaires) que les arbres multifurqués. Dans sa recherche de l'arbre le plus parcimonieux, la méthode ajoute les espèces non seulement en créant de nouvelles bifurcations au milieu des branches existantes mais aussi

elle essaye de placer ces espèces à la fin des nouvelles branches qui ont été ajoutées aux bifurcations existantes. Ainsi la méthode cherche au niveau des arbres binaires et des arbres multifurqués. Si une branche de l'arbre n'a aucun caractère qui pourrait changer au niveau de l'arbre le plus parcimonieux, il ne le sauvegarde pas. Ainsi dans chaque arbre résultant, une branche existe seulement si certains caractères ont une reconstruction plus parcimonieuse qui pourrait entrainer des changements au niveau de la branche.

La méthode sauvegarde également le nombre d'arbres les plus parcimonieux obtenus. Quand elle réarrange les arbres, elle essaye de réarranger tous les arbres sauvegardés. Ceci rend cet algorithme plus lent que celui des versions précédentes. Cependant sa complexité algorithmique reste $O(n^2)$.

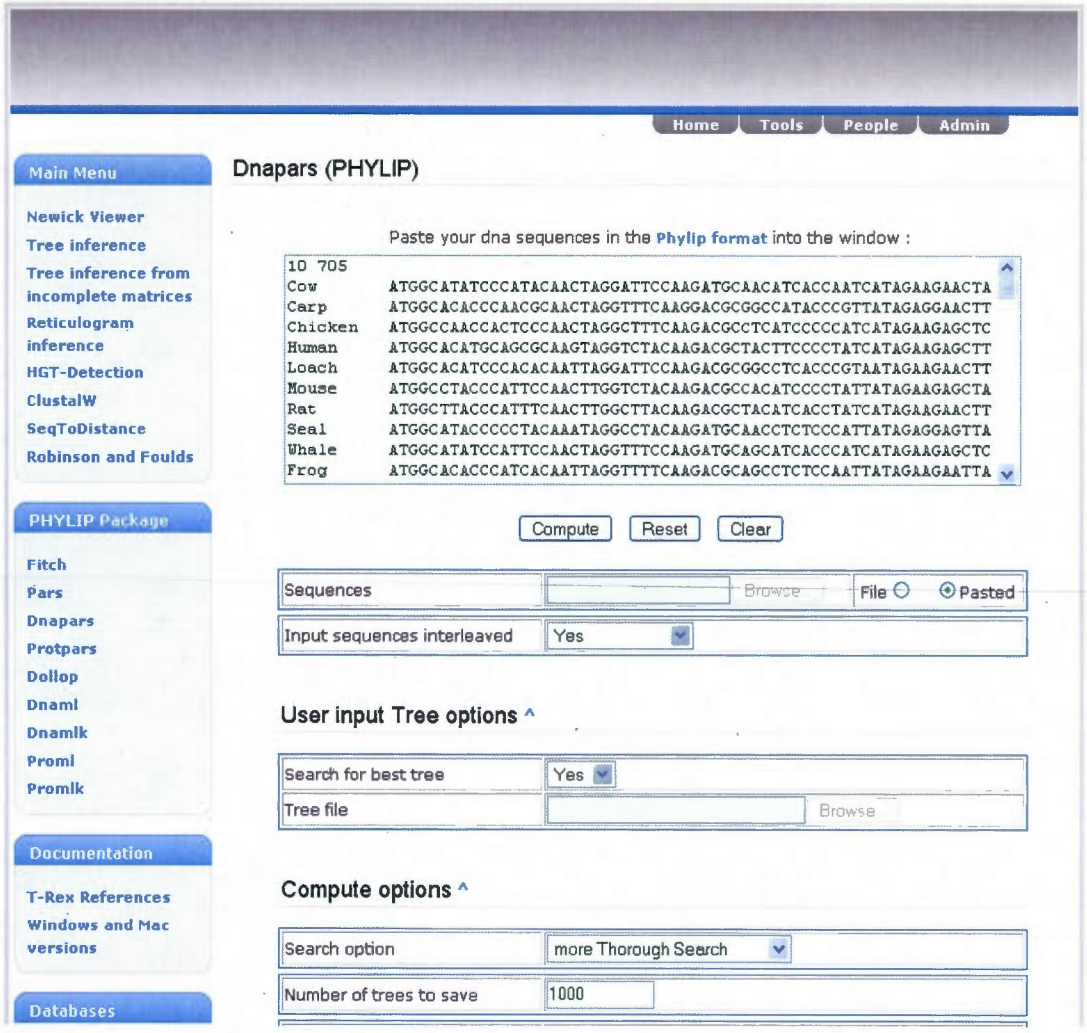


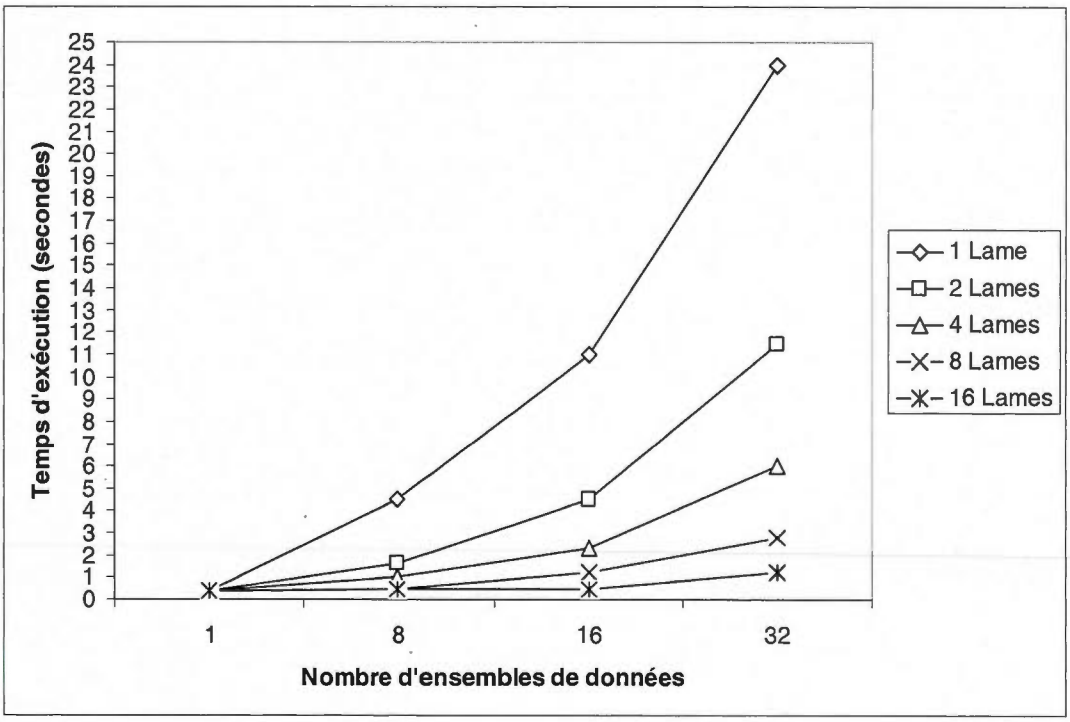
Figure VI-2 Interface de T-Rex avec la méthode DNAPARS.

Au niveau de la parallélisation du programme DNAPARS, un parallélisme de données (voir chapitre 4) a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme DNAPARS ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	0.385000
1	8	4.50000
1	16	11.000000
1	32	24.00000
2	1	0.400000
2	8	1.630000
2	16	4.500000
2	32	11.50000
4	1	0.400000
4	8	1.000000
4	16	2.300000
4	32	6.000000
8	1	0.400000
8	8	0.450000
8	16	1.200000
8	32	2.800000
16	1	0.400000
16	8	0.450000
16	16	0.500000
16	32	1.250000

Tableau VI-1 Algorithme DNAPARS : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.



VI-3 Algorithme DNAPARS : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement on peut utiliser les 16 lames pour minimiser le temps d'exécution.

6.3 PARS

PARS est un programme général de parcimonie qui est basé sur la méthode de parcimonie de Wagner. La parcimonie de Wagner permet des changements parmi tous les états. Le but est

de trouver l'arbre qui présente le nombre minimum de changements. La méthode de Wagner a été présentée par Eck et Dayhoff (1966) et par Kluge et Farris (1969).

Les suppositions faites pour ce modèle sont les suivantes :

- Les états ancestraux sont inconnus;
- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- Les changements d'un état à un autre ont la même probabilité;
- Ces changements sont *a priori* improbables au cours des périodes d'évolution impliquées dans la différentiation du groupe en question;
- Les autres événements d'évolution tels que par exemple la conservation du polymorphisme sont moins probables que les changements d'états;
- Les taux d'évolution entre les différentes lignées sont suffisamment bas pour que deux changements sur un long segment de l'arbre soient moins probables qu'un changement sur un segment court de l'arbre.

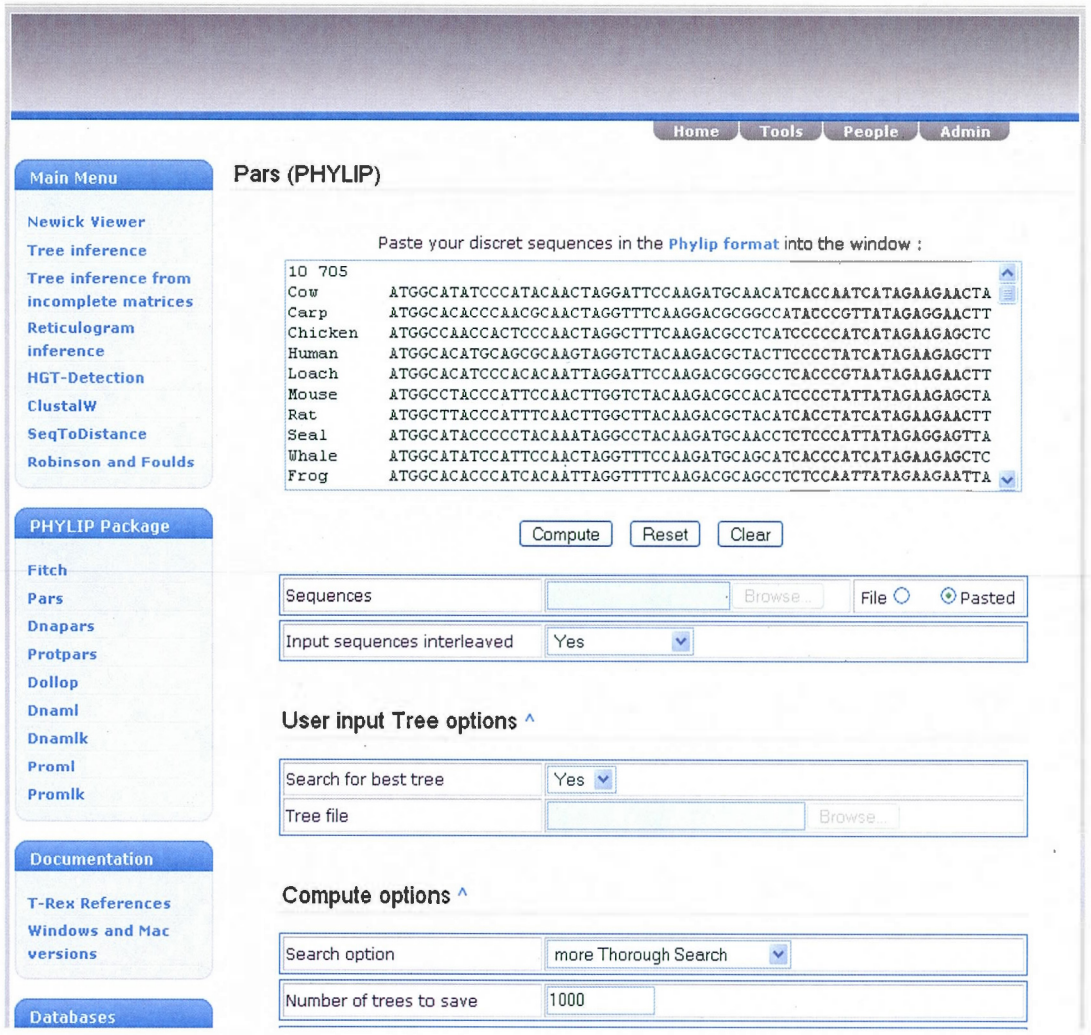


Figure VI-4 Interface de T-Rex avec la méthode PARS.

Au niveau de la parallélisation du programme PARS, un parallélisme de données (voir chapitre 4) a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme PARS ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	0.400000
1	8	5.000000
1	16	11.80000
1	32	25.00000
2	1	0.400000
2	8	1.750000
2	16	5.000000
2	32	12.50000
4	1	0.400000
4	8	0.930000
4	16	1.900000
4	32	4.000000
8	1	0.400000
8	8	0.500000
8	16	1.100000
8	32	2.200000
16	1	0.400000
16	8	0.500000
16	16	0.550000
16	32	1.200000

Tableau VI-2 Algorithme PARS : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

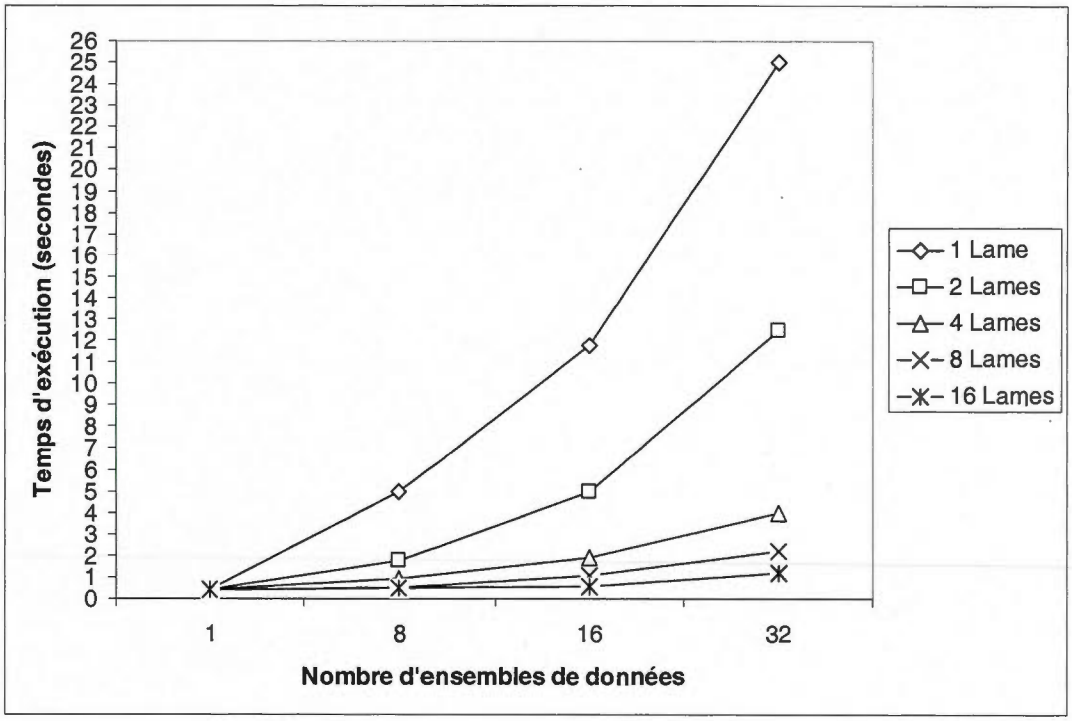


Figure VI-5 Algorithme PARS : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données ou un nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement on peut utiliser les 16 lames pour minimiser le temps d'exécution.

6.4 DOLLOP

Ce programme est basé sur les méthodes de parcimonie de Dollo et sur celles du polymorphisme. La méthode de parcimonie de Dollo a été suggérée la première fois sous forme verbale par Le Quesne (1974). Cette méthode a été par la suite spécifiée formellement

par Farris (1977). La méthode a été baptisée du nom de Louis Dollo puisqu'il fût l'un des premiers à affirmer que dans l'évolution il est plus difficile de gagner un dispositif complexe que de le perdre. L'algorithme explique la présence de l'état 1 en permettant jusqu'à un changement vers l'avant (0 →1) et autant de retours (1→0) nécessaires pour expliquer le modèle des états. Le programme a pour but de minimiser le nombre de changements retours (1→0) nécessaires.

Les suppositions faites pour ce modèle sont les suivantes :

- L'état ancêtre est connu (état 0);
- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- La probabilité d'un changement vers l'avant (0→1) est petite;
- La probabilité d'un retour (1→0) est également petite, mais plus grande que la probabilité d'un changement vers l'avant, de sorte qu'il soit plus facile envisager des changements retour que de réaliser un changement vers l'avant supplémentaire;
- La conservation du polymorphisme pour les deux états (0 et 1) est fortement improbable;
- Les longueurs des segments du vrai arbre sont presque égales de telle sorte que deux changements sur un long segment de l'arbre soient aussi probables qu'un changement sur un segment court.

Les suppositions faites pour le modèle de parcimonie de polymorphisme sont les suivantes :

- L'état ancêtre (état 0) est connu au niveau de chaque caractère;
- Chaque site d'une séquence peut évoluer indépendamment des autres sites;
- Les différentes lignées évoluent indépendamment les unes des autres;
- Le changement vers l'avant (0 →1) est fortement improbable au cours du temps d'évolution du groupe;
- La conservation du polymorphisme est également improbable, mais plus probable qu'un changement vers l'avant, de telle sorte qu'il est plus facile d'envisager plusieurs polymorphismes qu'un changement vers l'avant additionnel;

- Une fois que l'état 1 est atteint, le retour à l'état 0 est très improbable, beaucoup moins probable que des conservations multiples de polymorphisme;
- Les longueurs des segments du vrai arbre sont presque égales de telle sorte qu'il est plus facile d'envisager que des événements de conservation de polymorphisme se produisent dans deux longs segments qu'une conservation dans un court segment.

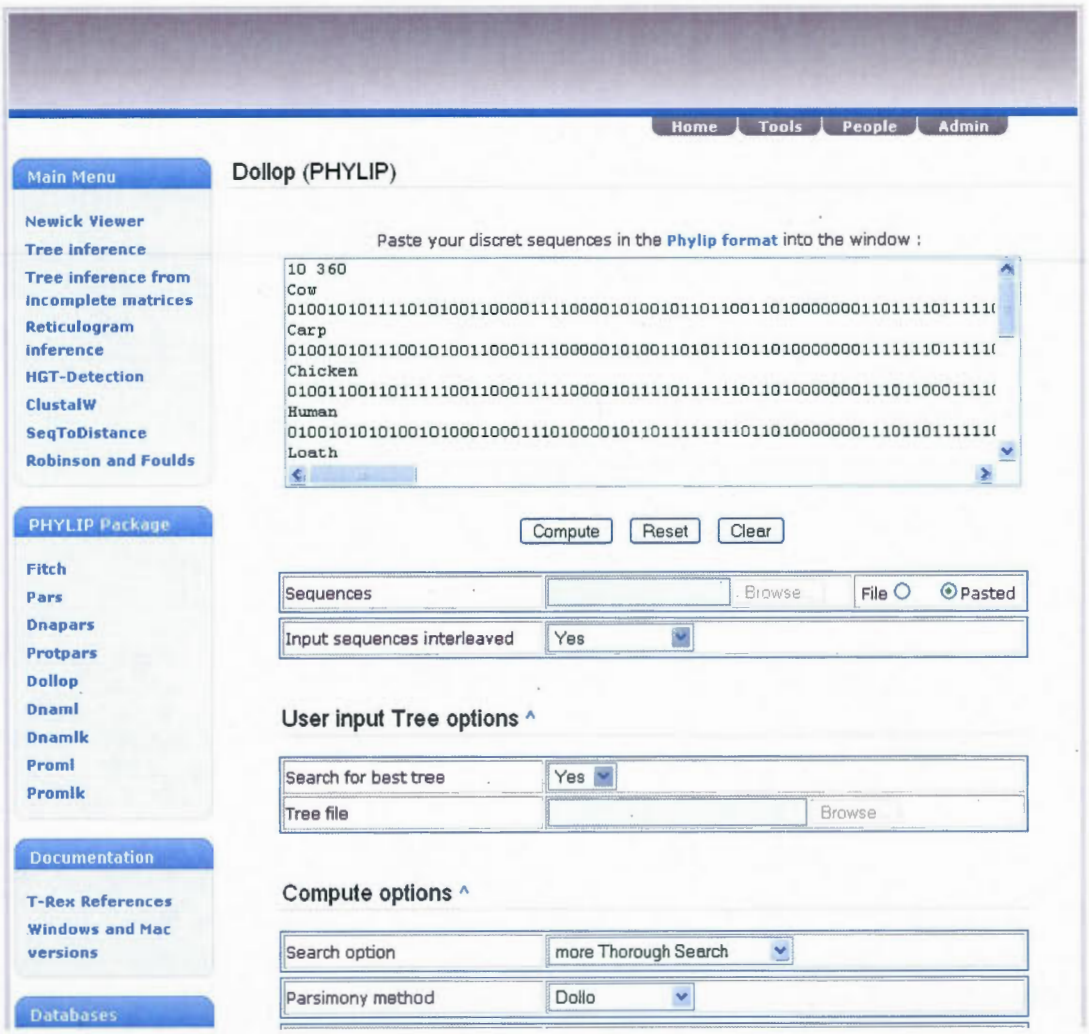


Figure VI-6 Interface de T-Rex avec la méthode DOLLOP.

Au niveau de la parallélisation du programme DOLLOP, un parallélisme de données (voir chapitre 4) a été réalisé en fonction du nombre d'ensembles de données reçues en entrée et en fonction du nombre d'arbres à inférer.

Des simulations effectuées pour calculer le gain de temps obtenu suite à la parallélisation de l'algorithme DOLLOP ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre d'ensembles de données	Temps moyen d'exécution (sec)
1	1	0.095000
1	8	0.620000
1	16	1.300000
1	32	3.300000
2	1	0.095000
2	8	0.285000
2	16	0.650000
2	32	1.420000
4	1	0.095000
4	8	0.185000
4	16	0.320000
4	32	0.700000
8	1	0.095000
8	8	0.110000
8	16	0.210000
8	32	0.400000
16	1	0.095000
16	8	0.115000
16	16	0.125000
16	32	0.220000

Tableau VI-3 Algorithme DOLLOP : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

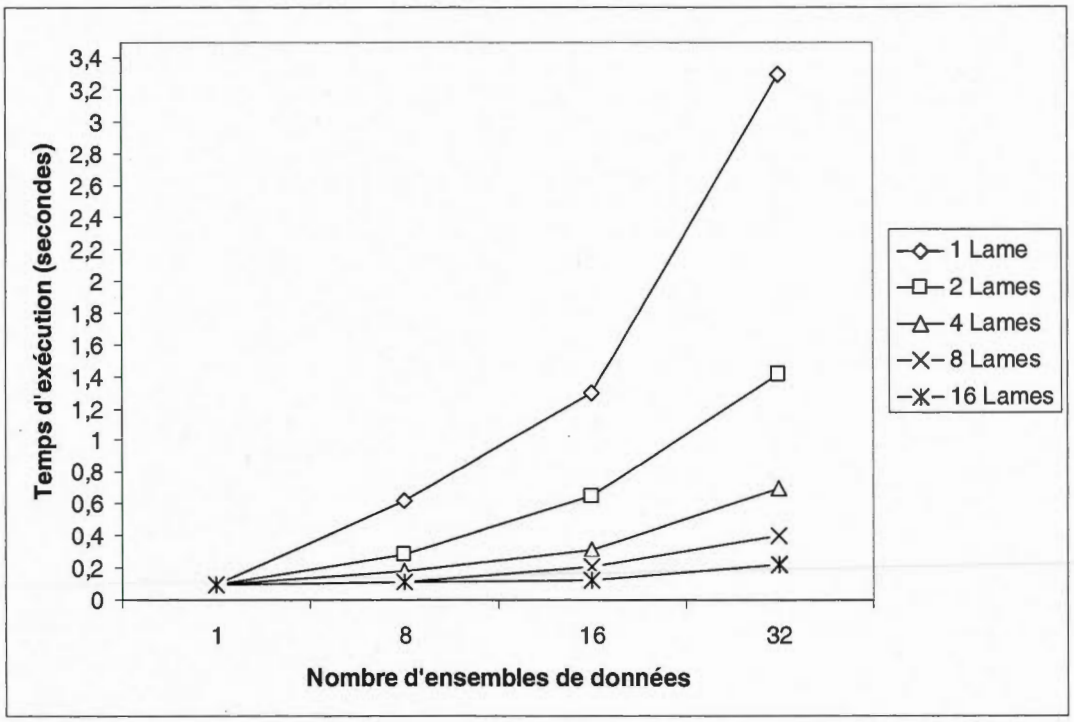


Figure VI-7 Algorithme DOLLOP : Temps écoulé en secondes en fonction de la taille des données d'entrée.

Les simulations effectuées pour calculer le gain de temps, en fonction d'arbres à inférer, donnent sensiblement les mêmes résultats que les simulations effectuées en fonction du nombre d'ensembles de données.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). Pour un nombre d'ensembles de données où le nombre d'arbres à inférer inférieur à 16, on peut utiliser le même nombre de lames. Autrement, on peut utiliser les 16 lames pour minimiser le temps d'exécution.

6.5 Références

Dayhoff, M.O. et Eck, R.V. (1968). A model of evolutionary change in proteins. *In : Atlas of Protein Sequence and Structure*, éd. par Fnd. (Natl. Biomed. Res.), chap. 4:. 33-41. Silver Spring MD.

Farris, J. S. (1977). Phylogenetic analysis under Dollo's Law. *Systematic Zoology* **26**: 77-88.

Felsenstein, J. (1993). PHYLIP (PHYLogeny Inference Package) version 3.6a2, Distributed by the author, Department of Genetics, University of Washington, Seattle, WA .

Fitch, W. M. (1971). Towards defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology* **20**: 406-416.

Kluge, A. G. et Farris, F. S. (1969). Quantitative phyletics and the evolution of anurans. *Systematic Zoology* **18**: 1–32.

Le Quesne, W. J. (1974). The uniquely evolved character concept and its cladistic application. *Systematic Zoology* **23**: 513-517.

CHAPITRE VII

PARALLÉLISATION DES ALGORITHMES DE DÉTECTION DE TRANSFERTS HORIZONTAUX DE GÈNES ET DE RECONSTRUCTION DE RÉTICULOGRAMMES

7.1 Transferts horizontaux de gènes

7.1.1 Introduction

Le transfert horizontal de gènes ou transfert latéral de gènes (HGT) se définit comme le passage d'une information génétique d'une espèce à une autre par des processus biologiques et l'insertion de cette information au sein du génome de l'hôte. Ces transferts sont très fréquents chez les bactéries. De ce fait, ces dernières deviennent de plus en plus résistantes aux antibiotiques en acquérant des gènes de résistance à partir d'autres espèces. Cependant, l'importance quantitative, au niveau du génome, de ces transferts horizontaux, par opposition aux transferts verticaux d'une cellule mère à une cellule fille, n'a pas pu être étudiée avant ces dernières années. Ce problème ne peut être abordé qu'avec la connaissance des génomes complets de plusieurs organismes. Plusieurs études ont suggéré que les transferts horizontaux étaient extrêmement fréquents chez les bactéries (Nelson et al. 1999). Certaines sont même allées jusqu'à rejeter le concept de phylogénie pour représenter l'évolution des Procaryotes, considérant que seul un réseau pourrait donner une représentation adéquate.

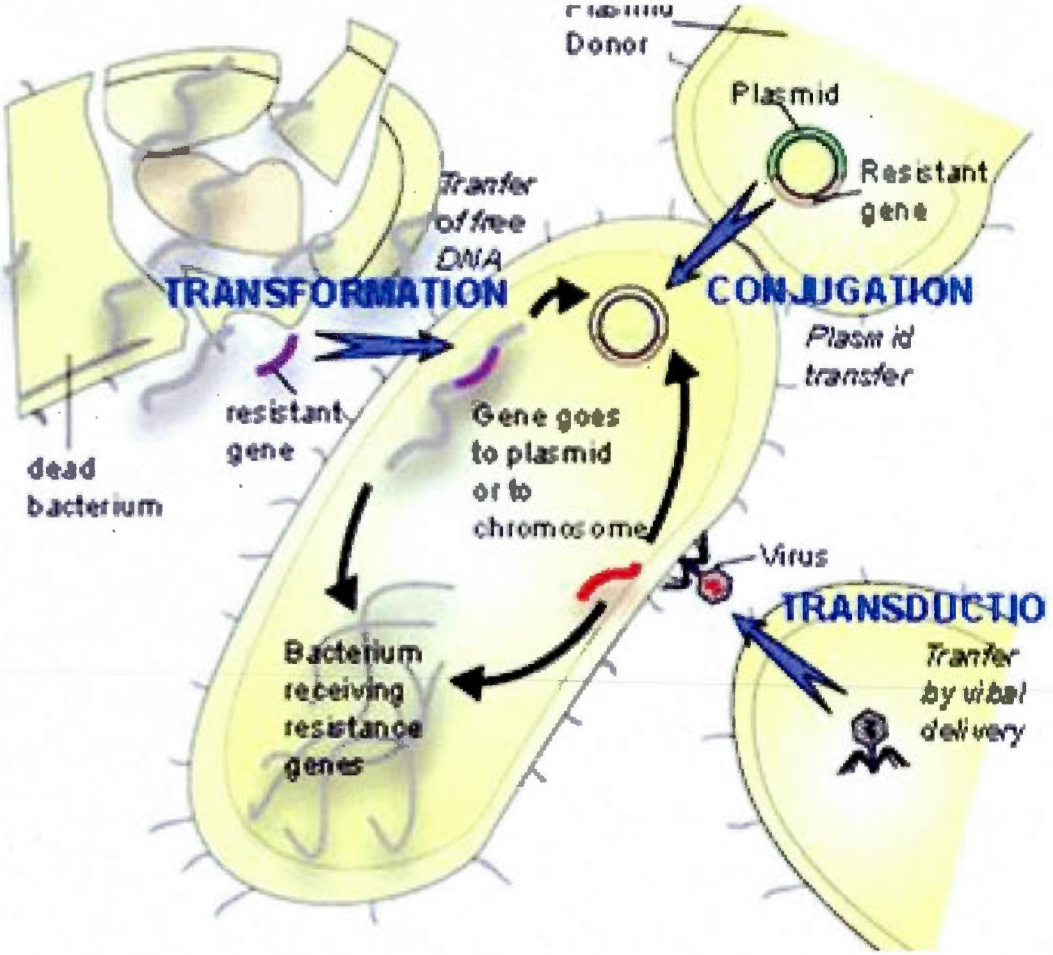


Figure VII-1 Transfert horizontal de gènes (3 mécanismes).

Il existe trois mécanismes pour le transfert latéral de gènes :

- La *transformation* est la mécanique la plus simple. Dans le milieu extérieur d'un organisme, se trouve de l'ADN libre qui résulte en général de la mort d'un organisme. Cet ADN libre peut être intégré à l'intérieur d'une cellule particulière, puis être intégré au génome de cet organisme. Il y a donc un transfert horizontal entre deux espèces qui peuvent être tout à fait différentes.

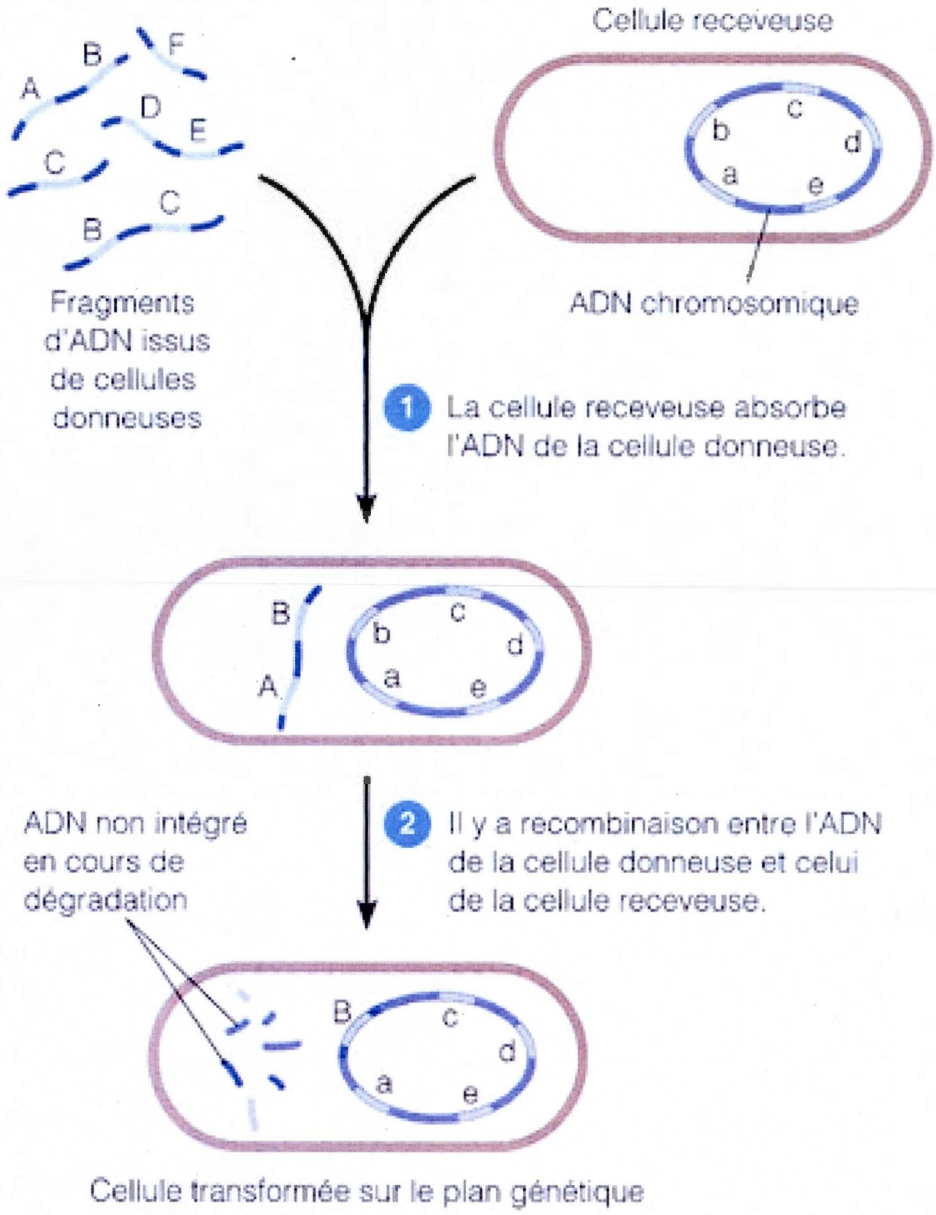
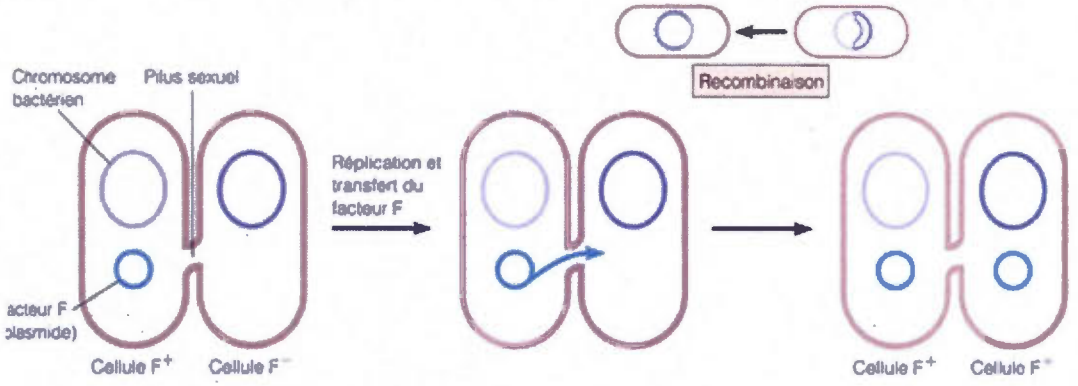


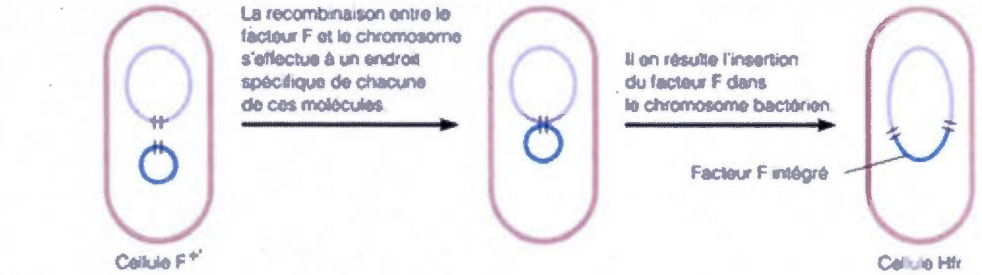
Figure VII-2 Mécanisme de transformation.

- Le second mécanisme est celui de la *conjugaison* où les organismes ont mis au point un système leur permettant de s'échanger du matériel génétique, en particulier des plasmides. Le processus est le suivant : deux cellules entrent en contact et s'échangent toute une partie de leur matériel génétique. Ce mécanisme est particulièrement important à

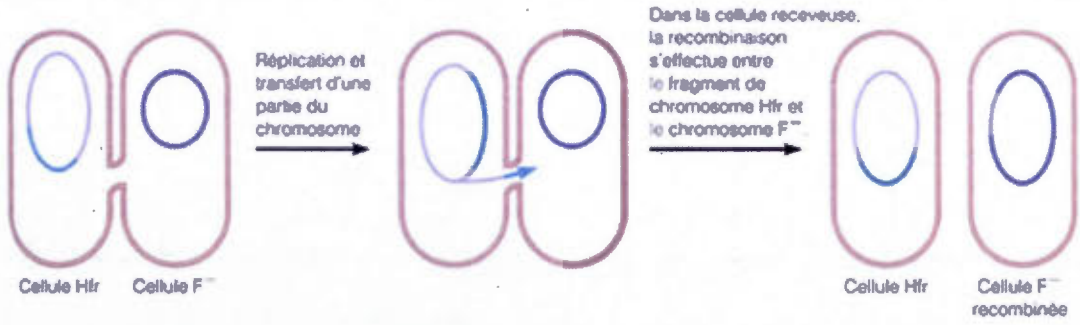
l'intérieur d'une même espèce, mais le phénomène de conjugaison peut également avoir lieu entre des organismes qui ne font pas partie de la même espèce.



Quand un facteur F (un plasmide) est transféré d'une cellule donneuse (F^+) à une cellule receveuse (F^-), la cellule F^- est convertie en cellule F^+.



Quand un facteur F s'intègre au chromosome d'une cellule F^+, la formation d'une cellule à haute fréquence de recombinaison (Hfr).



Quand une cellule Hfr donneuse transmet une partie de son chromosome à une cellule receveuse F^-, il en résulte une cellule F^- recombinée. Elle demeure F^- si le facteur F ne passe pas en entier lors de la conjugaison.

Figure VII-3 Mécanisme de conjugaison.

- Le dernier mécanisme est le mécanisme de *transduction*, où l'ADN est transféré d'une espèce à une autre via des virus ou des phages. Certains types de virus sont des organismes capables d'intégrer leurs matériels génétiques dans l'hôte et s'inspirent du génome de l'hôte pour donner naissance à de nouvelles particules virales. Ils peuvent amener par erreur une partie du matériel génétique de l'hôte et comme ces organismes n'ont pas une spécificité d'hôte très importante, ils sont capables de passer d'une espèce à une autre et donc de transférer du matériel génétique par erreur d'une espèce à une autre. Le virus ne peut pas, dans ce cas, infecter l'hôte, car il n'a plus tout son matériel génétique. Ce transfert horizontal est bénéfique pour l'hôte, car il n'est pas victime du bactériophage.

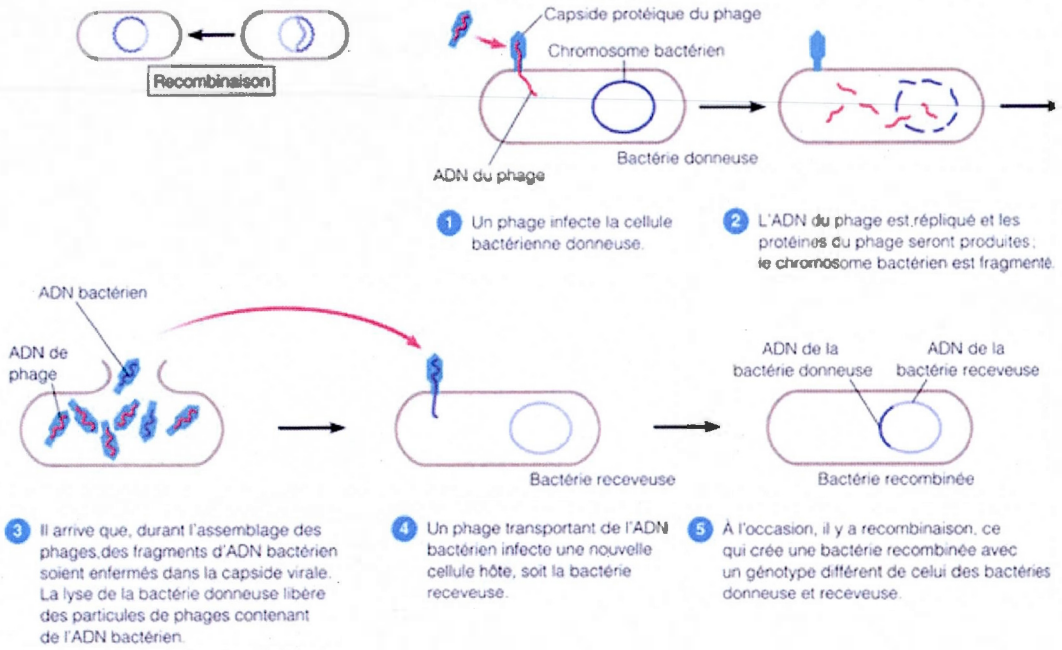


Figure VII-4 Mécanisme de transduction.

7.1.2 Algorithme

La méthode pour détecter des transferts horizontaux de gènes peut être divisée en deux parties principales :

- La première partie décrit la procédure de reconstruction des deux arbres phylogénétiques représentant l'évolution du gène considéré et de l'arbre d'espèces.
- La seconde partie de la méthode décrit le processus de réconciliation de ces deux arbres.

Partie 1 : Tel que présenté dans l'article (Boc et al., 2004)

> Soit T une phylogénie d'espèces dont les feuilles sont étiquetées selon l'ensemble X de n taxa. T peut être inféré à partir de données de séquences ou de distances à l'aide d'une méthode de reconstruction appropriée. Sans perte de généralité, nous supposons que T est un arbre binaire, dont les nœuds internes sont tous de degré 3 et qui possède $2n$–3 branches. Cet arbre doit être explicitement enraciné car la position de la racine est importante dans notre modèle.
>
> Soit T_1 un arbre de gène dont les feuilles sont étiquetées selon le même ensemble X de n taxa utilisé pour étiqueter l'arbre d'espèces. Comme la phylogénie d'espèces T, la phylogénie de gène T_1 peut être inférée à partir de données de séquences ou de distances caractérisant ce gène particulier. Si les topologies de T et T_1 sont identiques, aucun transfert horizontal du gène donné ne devrait être indiqué dans l'arbre d'espèces. Par contre, si les deux phylogénies sont topologiquement différentes, i.e. la distance de Robinson et Foulds entre les arbres T et T_1 est supérieure à zéro (voir Figure 5, par exemple), cela pourrait être dû aux transferts horizontaux de gène.
>
> (Boc et al., 2004, pages 5-6).

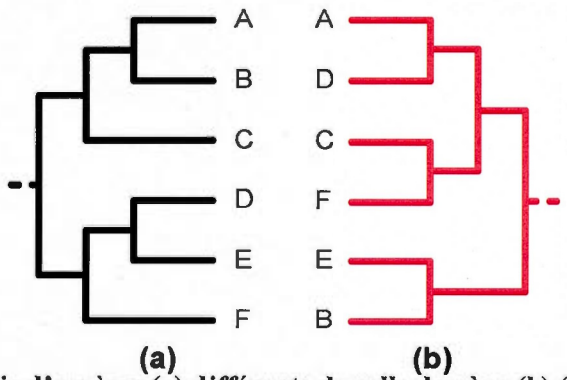


Figure VII-5 Phylogénie d'espèces (a) différente de celle de gène (b) (tiré de Boc et al., 2004 page 6).

Partie 2. Tel que présenté dans l'article (Boc et al., 2004). Notons que THGs signifie transfert horizontaux de gènes.

> Le but de cette étape est de déterminer un scénario de transferts horizontaux nécessaires pour transformer T en T_1. Tous les THGs possibles entre les paires de branches de l'arbre T qui sont en accord avec notre modèle d'évolution sont évalués. À la première itération, le transfert diminuant le plus la distance de Robinson et Foulds entre les deux arbres est considéré comme le plus probable. Il est par la suite ajouté à l'arbre T. Comme dans ce modèle nous considérons le transfert du gène au complet, la branche reliant l'espèce affectée par ce transfert et son ancêtre direct est supprimé de T.
>
> (Boc et a.l., 2004, pages 6-7).

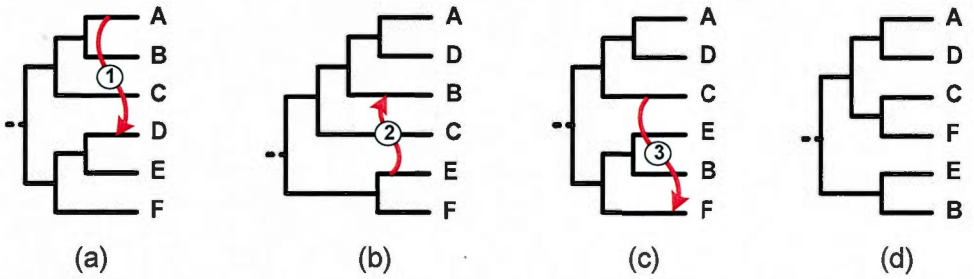


Figure VII-6 Arbre d'espèces (a) transformé en arbre de gène (d) en trois étapes (tiré de Boc et al., 2004 page 7).

> Ensuite, à la deuxième itération, cet arbre T modifié est considéré pour déterminer le prochain THG à ajouter. La procédure algorithmique s'arrête lorsqu'un nombre prédéfini de THGs est ajouté dans T ou lorsque la distance de Robinson et Foulds entre les arbres T modifié et T_1 atteint zéro. La liste des THGs produite par cette procédure représente un scénario possible de propagation du gène étudié. La Figure 6 ci-dessus illustre comment notre algorithme transforme l'arbre d'espèces (a) et l'arbre de gène (d). Chaque transfert trouvé tient compte des transferts ajoutés précédemment. Remarquons que le calcul de la distance topologique de Robinson et Foulds (pour le calcul optimal de cette distance, voir Makarenkov et Leclerc, 2000) entre l'arbre de gène T_1 et l'arbre d'espèces T modifié devient possible, car l'ajout d'une branche de transfert dans l'arbre d'espèces est toujours suivi par la suppression d'une branche, ce qui veut dire que T reste connexe est sans cycles et est donc toujours un arbre. La méthode décrite dans cette section nécessite $O(kn^4)$ opérations pour ajouter k transferts horizontaux dans l'arbre phylogénétique de n espèces.
>
> (Boc et al., 2004, pages 7).

Étant donné que cet algorithme est composé de parties indépendantes et que différentes opérations (telles que les boucles) peuvent être réalisées simultanément, nous avons utilisé le parallélisme de contrôle (voir chapitre 2).

Au niveau de l'algorithme, le choix de la paire d'espèces ayant un transfert horizontal de gènes est basé sur un critère à minimiser. Par conséquent, le critère est calculé pour chacune des paires d'espèces, puis la paire avec la meilleure valeur du critère est sélectionnée. Les calculs de critère au niveau de chaque paire d'espèces sont effectués à travers une série de boucles imbriquées et sont indépendants les uns des autres.

La parallélisation a consisté à diviser équitablement le calcul des critères au niveau des lames utilisées par le cluster (lors d'une exécution en parallèle). Cette division équitable a été réalisée à travers l'instruction de contrôle suivante : if (*indice % nombredelames = myid*); *indice* représente l'indice des espèces (allant souvent de zéro au nombre d'espèces moins un), *%* représente

l'opération « modulo », *nombredelames* représente le nombre de lames utilisées pour exécuter le programme en parallèle et *myid* représente l'identificateur de la lame du cluster (voir chapitre 2).

Ensuite, une synchronisation est effectuée entre toutes les lames utilisées afin de trouver la lame qui contient la paire avec la meilleure valeur du critère (chaque lame contient une paire optimale juste locale). Au niveau de la synchronisation, une uniformisation des variables du programme est effectuée (étant donné que les variables sont différentes entre les lames qui ont effectué des calculs sur des données différentes) afin d'avoir les mêmes arbres construits au niveau de toutes les lames.

Des simulations effectuées pour calculer le gain de temps produit à la suite de la parallélisation de l'algorithme HGT ont donné les résultats suivants (le temps moyen sur 100 jeux de données ayant chacun 14 espèces a été mesuré dans chaque cas) :

Nombre de lames	Nombre de transferts	Temps moyen d'exécution (sec)
1	1	780
1	8	6200
1	64	49500
2	1	450
2	8	3900
2	64	28500
4	1	230
4	8	1850
4	64	14720
8	1	114
8	8	910
8	64	7300
16	1	58
16	8	450
16	64	3700

Tableau VII-1 Algorithme HGT : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

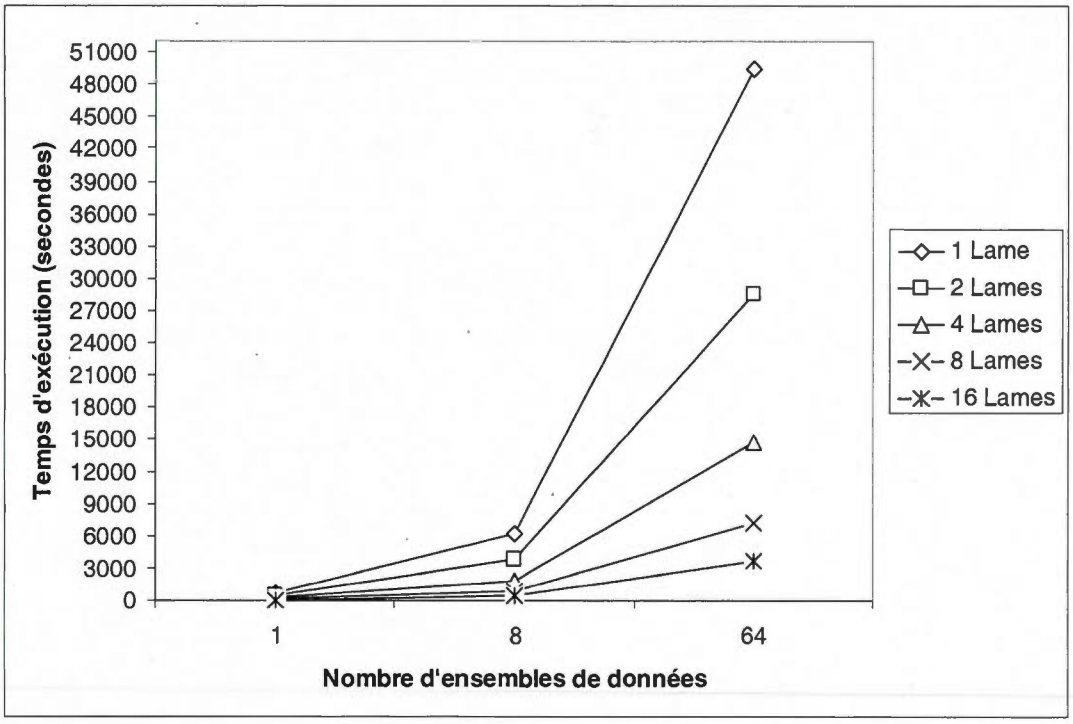


Figure VII-7 Algorithme HGT: Temps écoulé en secondes en fonction de la taille des données d'entrée.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). On peut toujours utiliser les 16 lames pour minimiser le temps d'exécution.

7.2 Reconstruction de réticulogrammes

7.2.1 Introduction

Un reticulogramme ou réseau réticulé (Makarenkov et al,. 2002, 2004, 2006) peut être défini comme un diagramme représentant la structure d'évolution des espèces, dans lequel ces dernières peuvent être reliées par plus d'un chemin à leurs ancêtres communs.

Un reticulogramme R est un triplet (N, B, l) tel que :

- N représente l'ensemble des nœuds.
- B représente l'ensemble de branches.
- l est la fonction de longueur de branche qui associe à un nombre non négatif à chaque branche.

La méthode débute par l'inférence d'un arbre phylogénétique à partir d'une matrice de distance par n'importe quelle méthode appropriée. Puis les branches de réticulations sont ajoutées.

L'algorithme de recherche de branches réticulées est le suivant (Makarenkov et al., 2002) :

1. Trouver la première branche xy à ajouter à l'arbre.
2. Essayer toutes les branches possibles :

   Recalculer les distances entres toutes les espèces X avec la présence de la nouvelle branche xy.

   Calculer le critère $Q = \sum_i \sum_j X(d_{ij} - r_{ij})^2$

3. Ajouter la branche xy, de longueur $l(x,y)$ qui minimise Q.
4. Répéter pour les nouvelles branches jusqu'à ce que le critère d'arrêt soit atteint.

Étant donné que cet algorithme est composé de parties indépendantes et que différentes opérations (telles que les boucles) peuvent être réalisées simultanément, nous avons utilisé le parallélisme de contrôle (voir chapitre 2).

Au niveau de l'algorithme, le calcul de toutes les distances en présence de la nouvelle branche *xy* et le calcul du critère sont effectués à travers une série de boucles imbriquées et sont indépendants les uns des autres.

La parallélisation a consisté à diviser équitablement le calcul des distances et du critère au niveau des lames utilisées par le cluster (lors d'une exécution en parallèle). Cette division équitable a été réalisée à travers l'instruction de contrôle suivante : if (*indice % nombredelames = myid*); *indice* représente l'indice des espèces (allant souvent de zéro au nombre d'espèces moins un), *%* représente l'opération « modulo », *nombredelames* représente le nombre de lames utilisées pour exécuter le programme en paralléle et *myid* représente l'identificateur de la lame du cluster (voir chapitre 2).

Ensuite, une synchronisation est effectuée entre toutes les lames utilisées afin de trouver la lame qui contient la branche avec la meilleure valeur du critère (car chaque lame contient une branche optimale juste locale). Au niveau de la synchronisation, une uniformisation des variables du programme est effectuée (étant donné que les variables sont différentes entre les lames qui ont effectué des calculs sur des données différentes) afin d'avoir les mêmes arbres construits au niveau de toutes les lames.

Des simulations effectuées pour calculer le gain de temps produit à la suite de la parallélisation de l'algorithme d'ajout de réticulations ont donné les résultats suivants (le temps moyen sur 100 jeux de données a été mesuré dans chaque cas) :

Nombre de lames	Nombre de sets de données	Temps moyen d'exécution (sec)
1	8	0.007600
1	16	0.230000
1	32	9.230000
2	8	0.006200
2	16	0.130000
2	32	5.200000
4	8	0.006300
4	16	0.089000
4	32	2.900000

8	8	0.006000
8	16	0.070000
8	32	1.750000
16	8	0.006000
16	16	0.070000
16	32	1.000000

Tableau VII-2 Algorithme de reconstruction de reticulogrammes : Temps écoulé en secondes en fonction de la taille des données d'entrée et du nombre de lames du cluster.

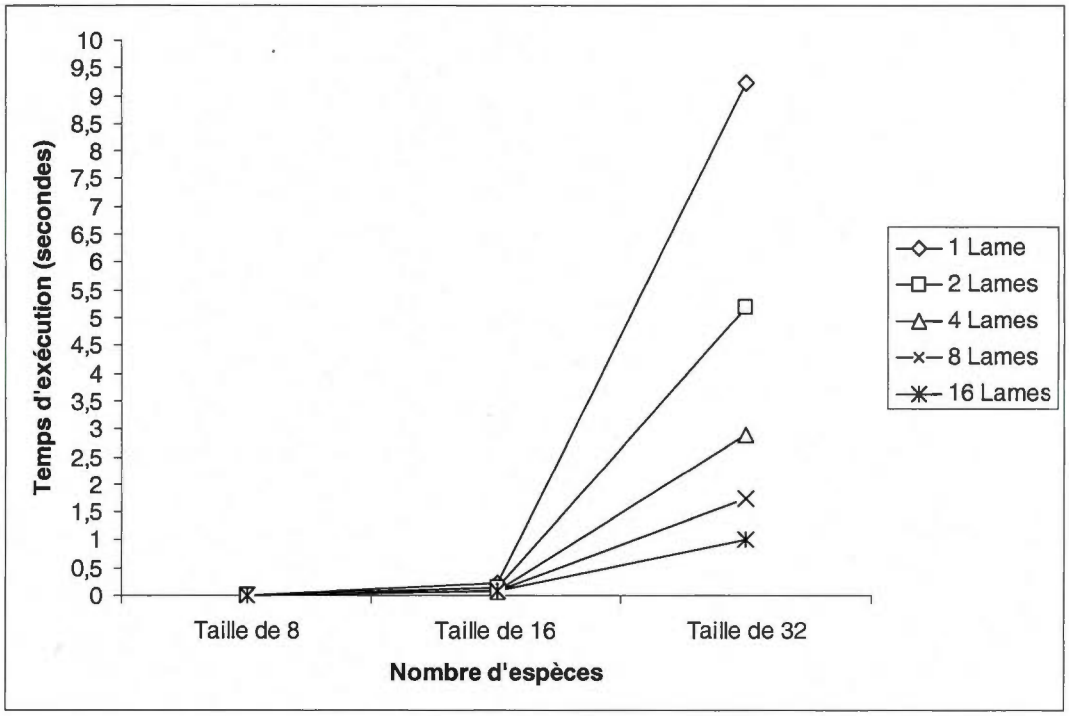


Figure VII-8 Algorithme de reconstruction de reticulogrammes : Temps écoulé en secondes en fonction de la taille des données d'entrée.

La version parallèle a toujours été plus rapide que la version séquentielle quel que soit le nombre de lames utilisées (à l'exception d'une lame). On peut toujours utiliser les 16 lames pour minimiser le temps d'exécution.

7.3 Références

Boc, A., Makarenkov, V. et Diallo, A. B. (2004). Une nouvelle methode pour la detection de transferts horizontaux de gene : la reconciliation topologique d'arbres de gene et d'especes, *JOBIM* 2004, Montreal, Canada.

Makarenkov, V. et Legendre, P. (2002). Nonlinear redundancy analysis and canonical correspondence analysis based on polynomial regression, *Ecology* **83 (4)** :1146-1161.

Makarenkov, V., Legendre, P. et Desdevises, Y. (2004). Modeling phylogenetic relationships using reticulated networks. *Zoologica Scripta* **33 (1)** : 89-96.

Makarenkov, V., Kevorkov, D. and Legendre, P. (2006). Phylogenetic Network Reconstruction Approaches, Applied Mycology and Biotechnology, International Elsevier, *Bioinformatics* **6** : 61-97.

Nelson et al. (1999). Evidence for lateral transfer between Archaea and bacteria from genome sequence of Thermotoga maritima. *Nature* **399(6734)** : 323-329.

CONCLUSION ET PERSPECTIVES

Dans ce mémoire, nous avons premièrement présenté trois nouvelles méthodes pour la reconstruction d'arbres phylogénétiques et de réseaux de transferts horizontaux de gènes. Ces méthodes ont été décrites dans les trois publications incluses dans le présent mémoire. Le premier traite du problème de la reconstruction d'arbres phylogénétiques à partir de séquences contenant des nucléotides manquants ou indéterminés. Le deuxième article décrit une nouvelle méthode pour prédire et visualiser les possibles transferts horizontaux de gènes dans le cadre des modèles complet et partiel. Dans le troisième article nous détaillons notre recherche sur des modèles de transferts horizontaux. Une procédure de validation statistique des transferts inférés a été conçue et présentée à l'aide d'un exemple de données réelles. Cette procédure peut être appliquée pour estimer la robustesse d'un transfert horizontal particulier ou d'un scenario de transferts au complet.

Deuxièmement, nous avons présenté un procédé d'accélération des calculs, effectués par différents algorithmes de reconstruction phylogénétique, en utilisant le parallélisme. Ce procédé a été mis au point à la suite de la croissance du volume de données génétiques disponibles et du plafonnement de la puissance des ordinateurs (autour de 3,4GHZ depuis environ deux ans).

Le parallélisme mis en place a été réalisé à travers une librairie standard de passage de messages (Message Passing Interface). Nous avons utilisé deux formes de parallélisme, à savoir :

- Parallélisme de données : « La même opération est effectuée par des processeurs différents sur des ensembles disjoints de données » ;
- Parallélisme de contrôle : « Différentes opérations sont réalisées simultanément. Ce parallélisme est utilisé lorsque le programme est composé de parties indépendantes ou lorsque certaines structures de contrôle (telles que les boucles) sont susceptibles d'être exécutées en parallèle ».

Le parallélisme a été réalisé sur une machine Linux de type « grappe » (i.e., cluster), composé de 16 lames (i.e., blades) possédant chacune deux processeurs et sa propre mémoire (architecture « share nothing » ou « sans partage »).

Les résultats obtenus pour les différents jeux de données ont permis de montrer que notre procédé améliore de façon considérable le temps d'exécution des différents algorithmes bioinformatiques.

Cependant, nous avons remarqué que le temps nécessaire pour passer des informations sur des grosses structures de données entre les différentes lames est un facteur important qui peut ralentir les calculs en parallèle et doit être absolument pris en compte.

À l'avenir, il serait intéressant de développer un procédé similaire, en utilisant une machine dont l'architecture est de type « Shared Disks » ou de type « Share Everything ». Dans ces architectures, chacune des lames qui compose le système fonctionne sous le contrôle de sa propre copie du système d'exploitation mais peut accéder directement aux autres disques (mémoire) qui sont partagés. Avec ces deux types d'architecture, aucun passage de messages entre les différentes lames du cluster ne sera nécessaire, ce qui pourrait se traduire en un gain de temps assez important. Toutefois, des opérations de gestion de l'accès à la mémoire devront être mises en place. Une autre possibilité d'améliorer la performance des versions parallèles de nos algorithmes pourrait être l'utilisation des «Native Hardware Features » qui font partie de la librairie de passage de messages (MPI).

ANNEXE A

SERVEUR WEB DU LOGICIEL T-REX

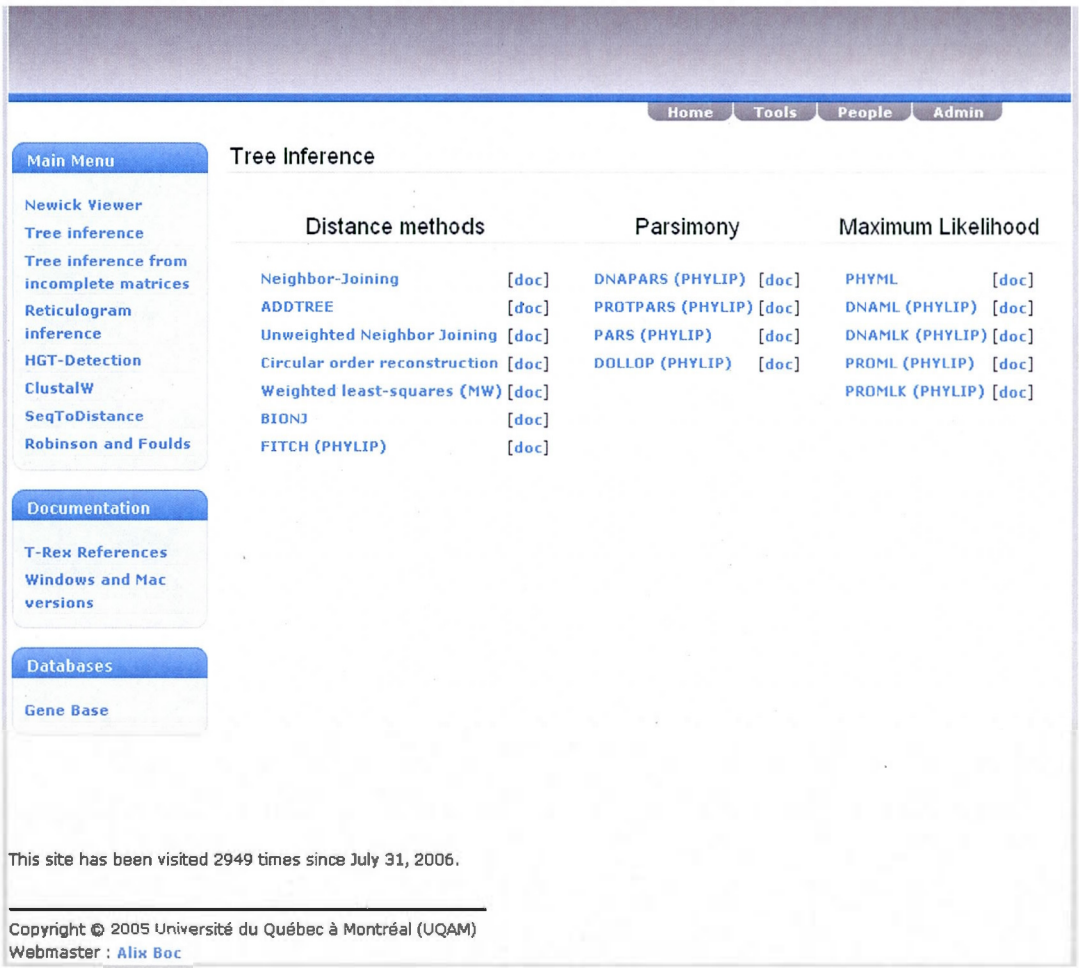


Figure ANNEXE A-1 Page d'accueil du serveur Web de T-REX.

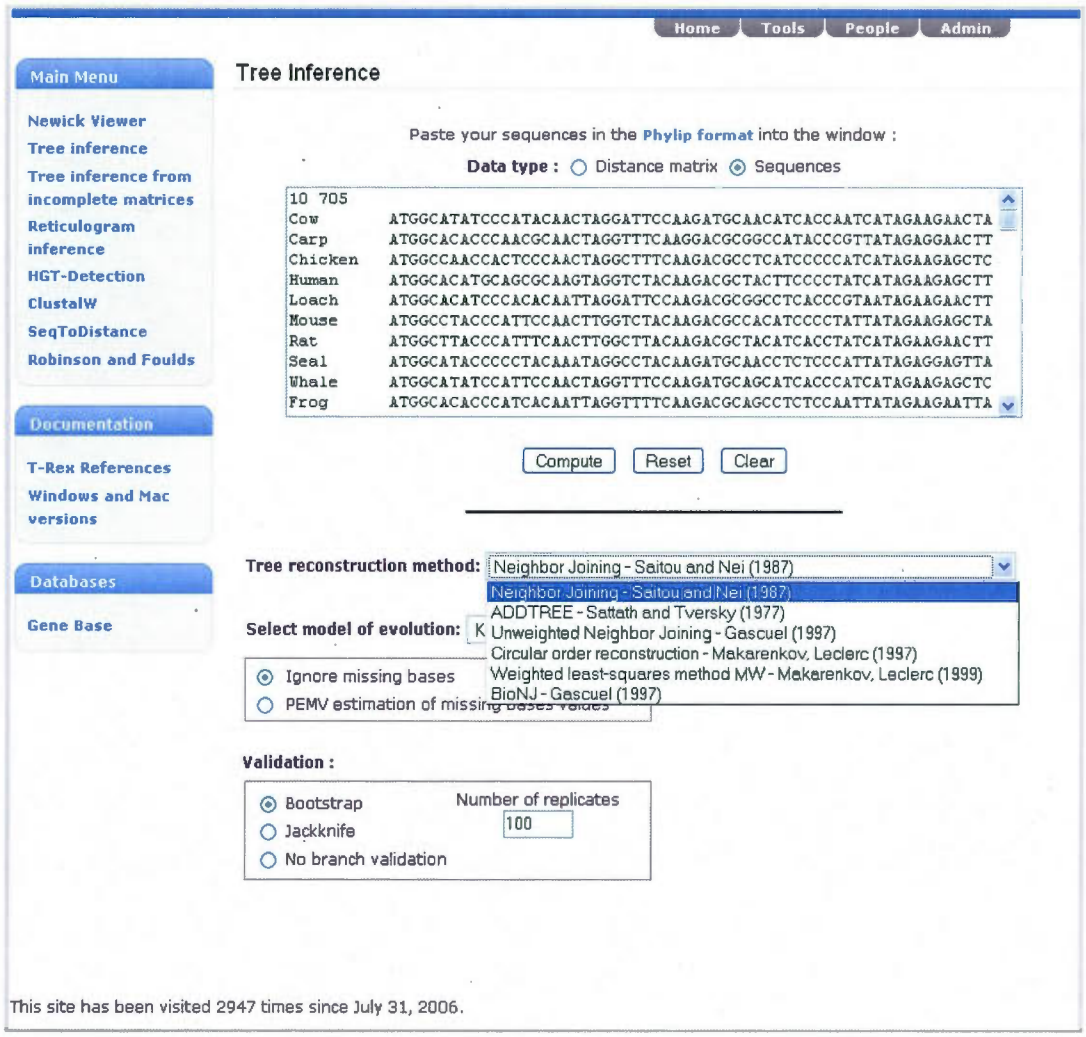


Figure ANNEXE A-2 Inférence d'un arbre phylogénétique avec la méthode NJ.

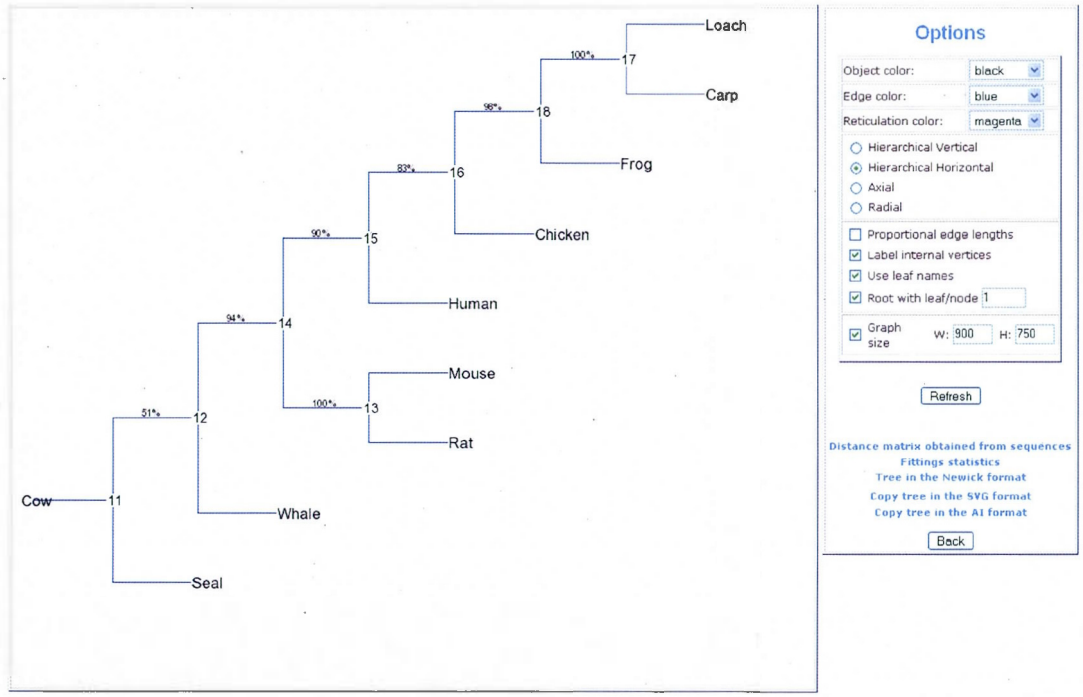


Figure ANNEXE A-3 Arbre inféré avec la méthode NJ.

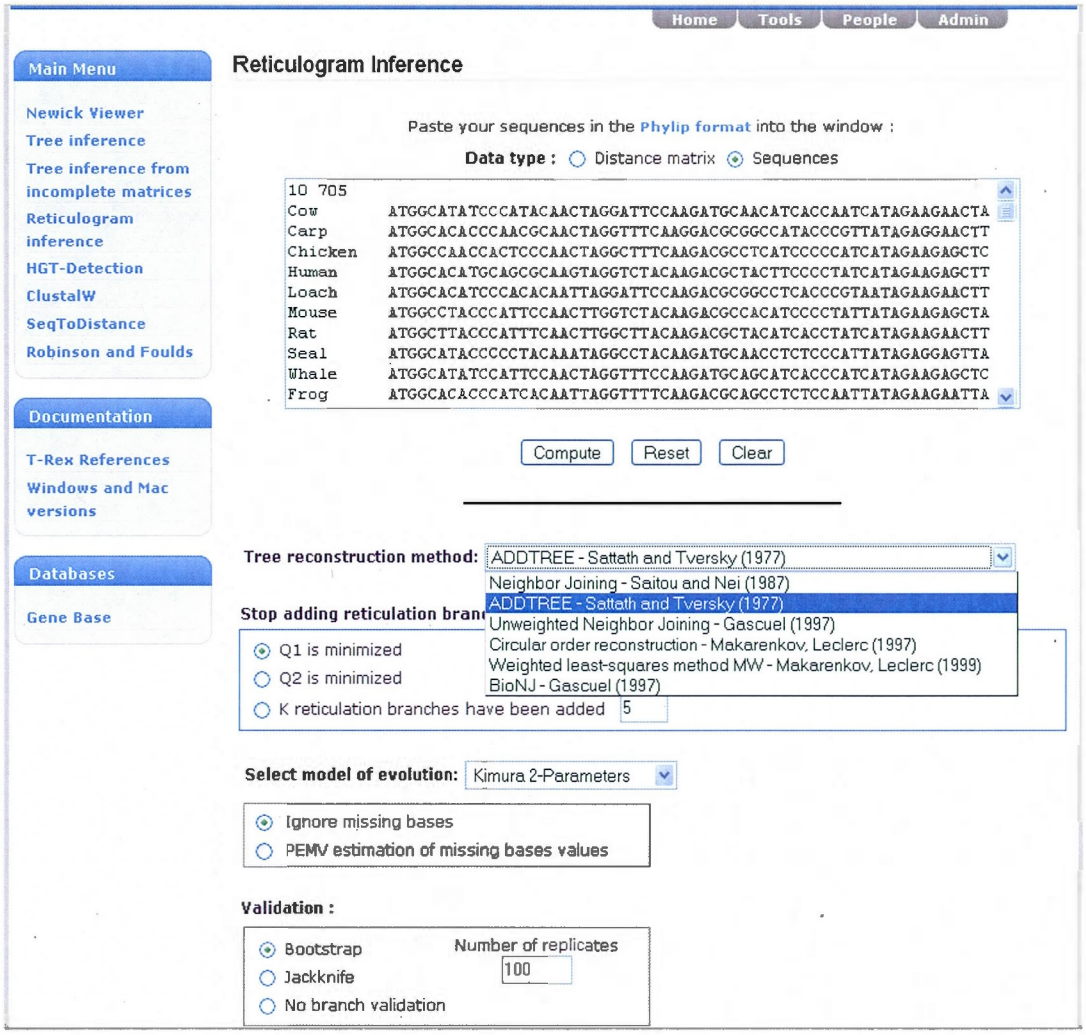


Figure ANNEXE A-4 Inférence d'un réticulogramme avec la méthode ADDTREE.

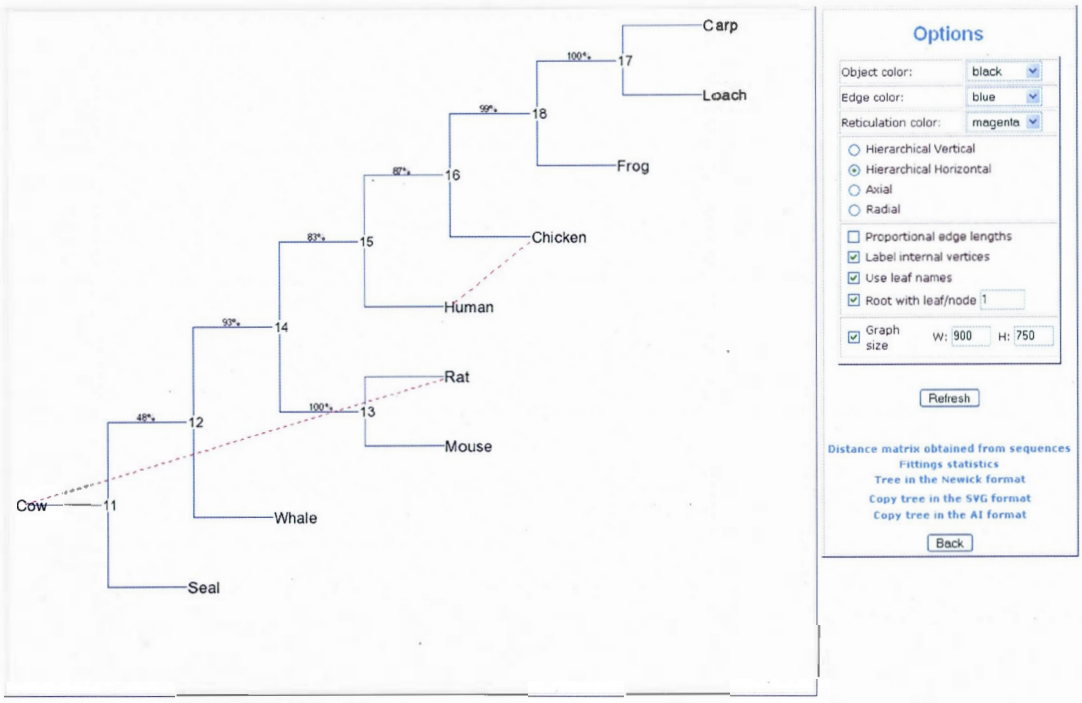


Figure ANNEXE A-5 Réticulogramme inféré avec la méthode ADDTREE.

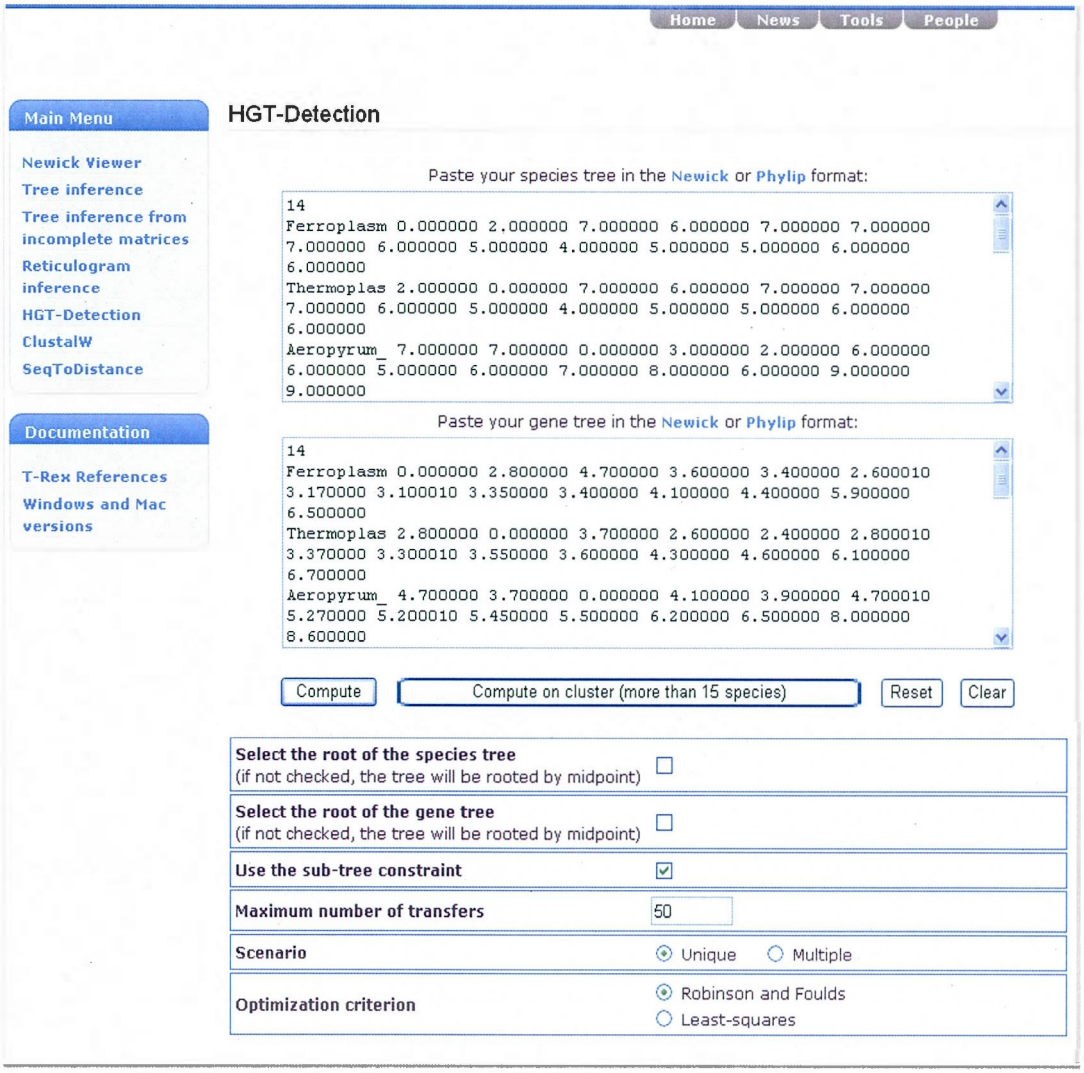


Figure ANNEXE A-6 Détection de transferts horizontaux de gènes.

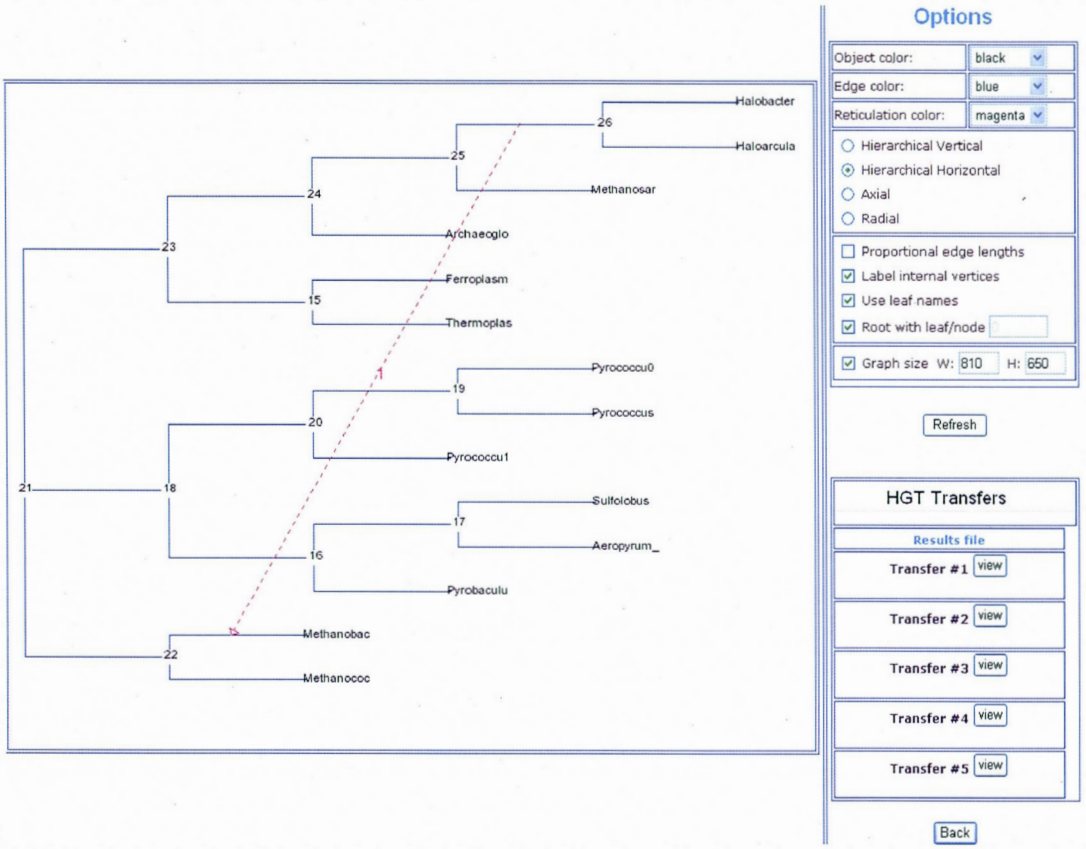


Figure ANNEXE A-7 Premier transfert horizontal détecté.

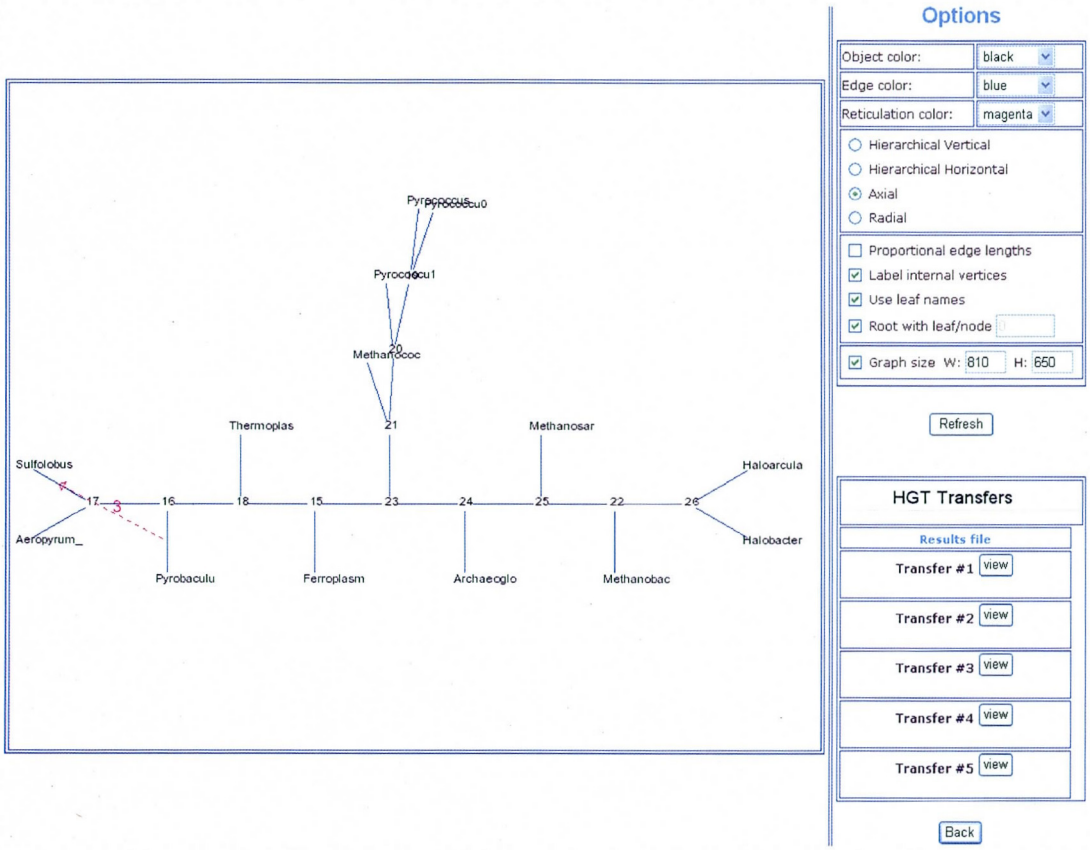


Figure ANNEXE A-8 Troisième transfert horizontal détecté.

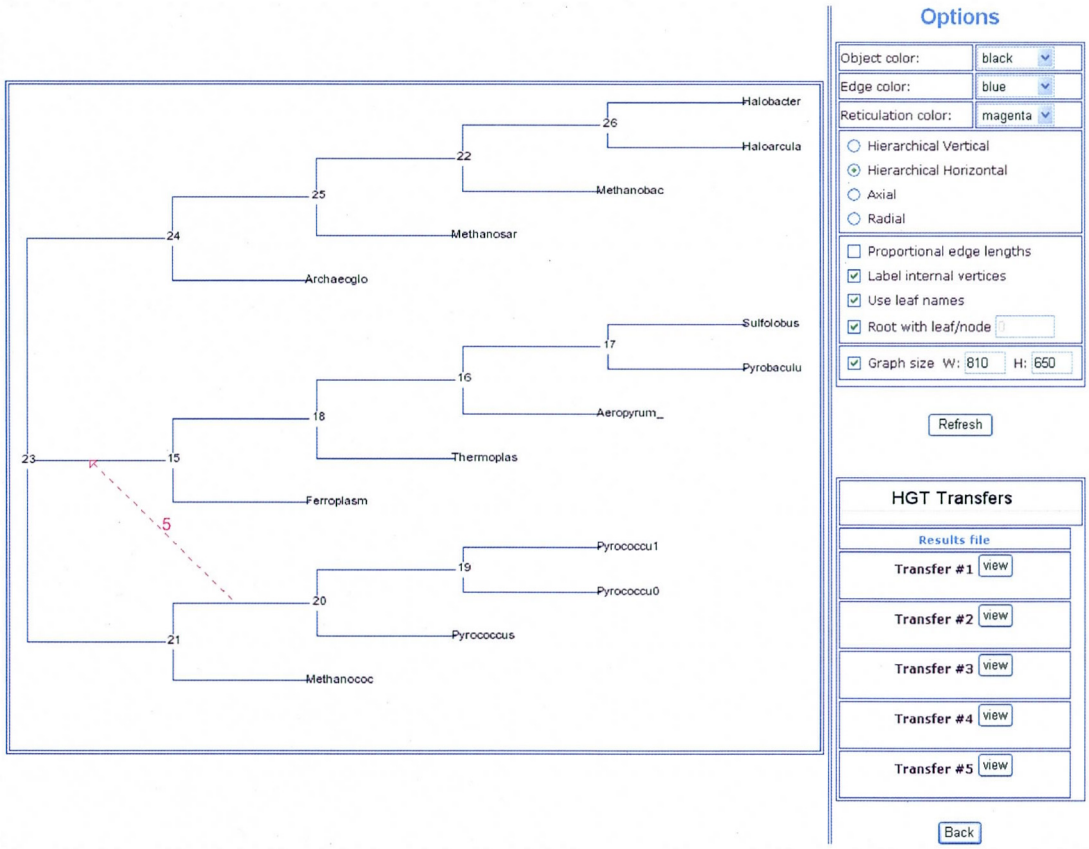


Figure ANNEXE A-9 Cinquième transfert horizontal détecté.

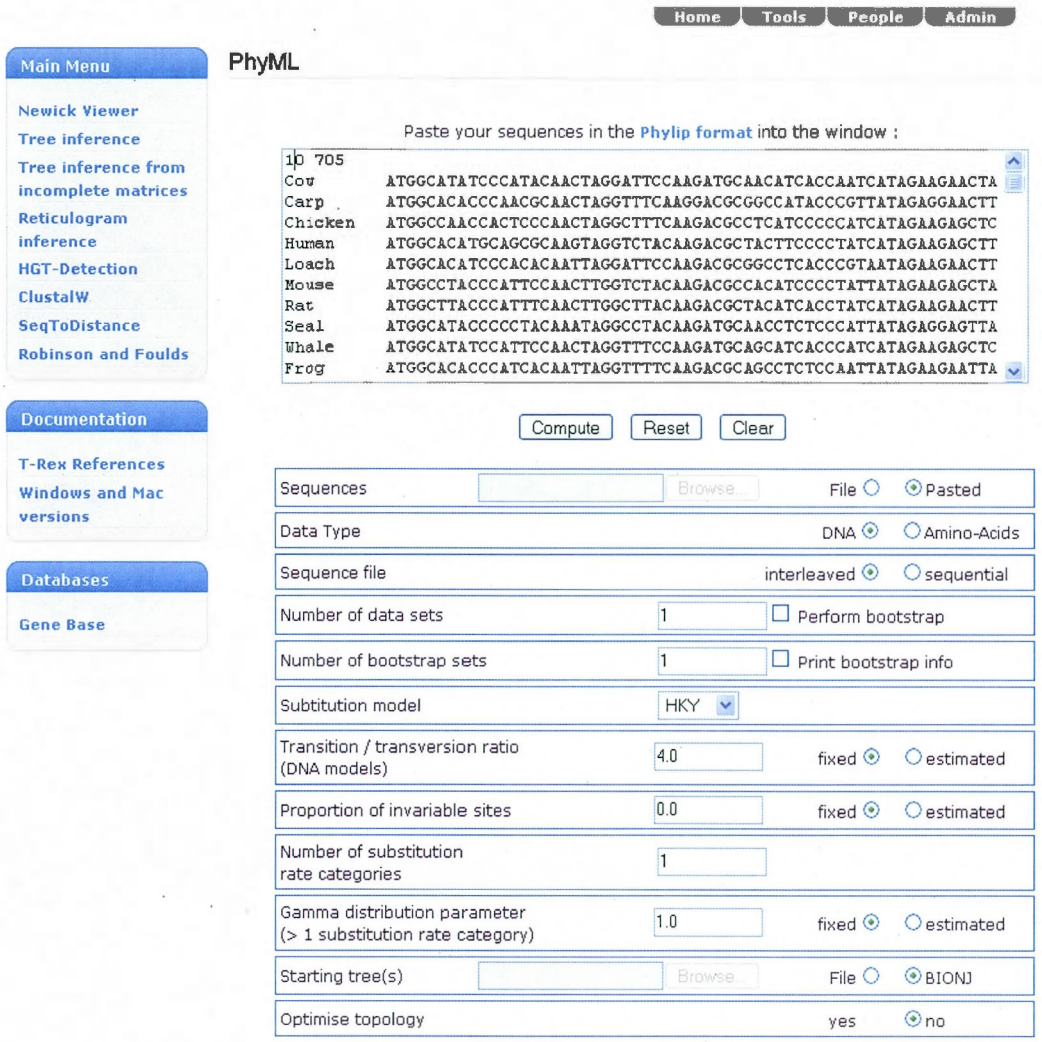


Figure ANNEXE A-10 Inférence d'un arbre phylogénétique avec la méthode PHYML.

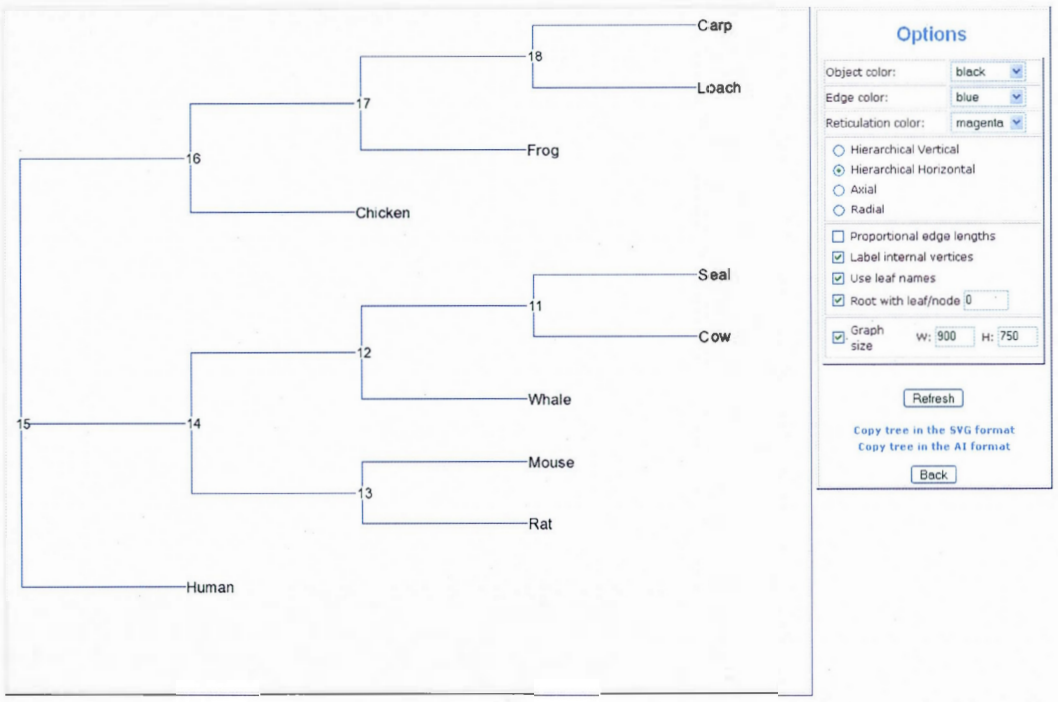


Figure ANNEXE A-11 Arbre inféré avec la méthode PHYML.

ANNEXE B

CODE SOURCE

A cause du volume du code, nous présenterons seulement quelques parties essentielles du travail. Le code source complet est disponible sur demande.

Nous présenterons dans la section suivante le code parallélisé du programme T-Rex (NJ, UNJ, ADDTREE, Circular Order Reconstruction, MW, BioNJ). Ensuite, nous présenterons le code parallélisé du package PHYLIP (FITCH, DNAML, DNAMLK, PROML, PROMLK, DNAPARS, PROTPARS, PARS et DOLLOP). Finalement, nous présenterons le code parallélisé de PHYML.

PROGRAMME T-REX

Le code source du programme T-Rex a été développé par Vladimir Makarenkov et par certains membres de notre laboratoire. Ma participation porte principalement sur la parallélisation des méthodes suivantes : NJ, UNJ, ADDTREE, Circular Order Reconstruction, MW, BioNJ ainsi que la méthode permettant de réaliser un bootstrap.

Ici, nous présenterons seulement des parties du fichier trexParallele.c qui contient toutes les méthodes parallélisées.

```
#include <limits.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>
#include "phylip.h"
#include "global.h"
#include "CalculDistances.c"
#include "DistanceMéthode.c"
#include "seqboot.c"

/*Import de la librairie de MPI */
#include "mpi.h"


...


/*********************
 *La fonction MAIN    *
**********************/
int main (int argc,char** argv)
{
        int retour = 0;
        long int *ARÉTÉ;
        double *LONGUEUR,**matDist;

        /*Initialisation des variables MPI*/
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        LireOptions(argv,argc);

        if((p_optionTR == 1)||(p_optionTR == 2)){ /*inference d'arbres ou de reticulogramme a partir de matrice*/
                if((m_dataType == 2)&&(p_boot == 2)){ /* sequences sans bootstrap*/
                        retour = seqToDistance(p_input,&matDist,1);
                        if(retour==0){
                                retour = trex(p_matrix,p_output,p_stat,&ARÉTÉ,&LONGUEUR,1,0);
                        }
                }
                if((m_dataType == 2)&&(p_boot < 2)){ /* sequences avec bootstrap*/
                        retour = bootstrap_function();
                }
                if(m_dataType == 1){
                        retour = trex(p_input,p_output,p_stat,&ARÉTÉ,&LONGUEUR,1,0);
                }
        }
        /* Matrices incomplétés */
        else{
                retour = trex(p_input,p_output,p_stat,&ARÉTÉ,&LONGUEUR,1,0);
        }
        MPI_Finalize();
        return retour;
}/* END main*/
```

Méthode trex :

```
/*=============================================================================*
 * Cette methode infere un arbre phylogenetique a partir de la matrice         *
 * de distances reçue en entree, en utilisant la methode selectionnee en       *
 * parametre                                                                   *
 *                                                                             *
 * param:                                                                      *
 *       inputFile              : Fichier de la matrice de distances           *
 *       treeFile               : Fichier de l'arbre infere                    *
 *       statFile               : Fichier sur les statisques de l'arbre        *
 *       ARÉTÉ                  : Tableau d'aretes de l'arbre                  *
 *       LONGUEUR_              : Tableau du longueur des aretes de l'arbre    *
 *       ecritureResultats:     : 1 (vrai) si les resultats sont ecrits dans un fichier
 *              sinon 0 (faux)                                                 *
 *       unproc                 : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux)
 *============================================================================*/

int trex(char *inputFile,char *treeFile, char *statFile,long int **ARÉTÉ_,double **LONGUEUR_,int ecritureResultats,
int unproc){
        double Puissance;
        int l,i,j,Iternum,ReticulationsNumber=0, OptionFunction=0, optionMiss=0,method=0,optionTR=0;
        double *LONGUEUR,power;
        long int *ARÉTÉ;
        double **DI,**DA,**DA1,**W,*R,*RAJ;
        int Option,Option1;
        double *CRITERION1, *CRITERION2;
        double RESULTATS[4];
        char mwFile[50];
        FILE *mod;
        FILE *data;
        FILE *newickFile;

        char symbol, PrintTreeMetric='Y';
        char **Names;
        double c;
        int k;
        char car;
        char *stringNewick = (char*)malloc(1000);
        char chCaractere;

        if ((data=fopen(inputFile,"r"))==0) { printf("File %s was not found \n", p_input); exit(8); }

        /* Reading the dissimilarity data file */
        fscanf(data,"%d",&n);
        if ((n>10000)||(n<1)) { printf("*Incorrect Phylip format or too many taxa in the distance matrix n = %d\n\n",
                                n); exit(5); }
        if(n<numprocs)
                maxprocs=n;
        else
                maxprocs = numprocs;
        /*Merory allocation*/
        DI=(double **) malloc((n+1)*sizeof(double*));
        DA=(double **) malloc((n+1)*sizeof(double*));
        DA1=(double **) malloc((n+1)*sizeof(double*));
        W=(double **) malloc((n+1)*sizeof(double*));
        R=(double *) malloc(4*sizeof(double));
        RAJ=(double *) malloc(4*sizeof(double));
        ARÉTÉ=(long int *) malloc((4*n-1)*sizeof(long int));
        LONGUEUR=(double *)malloc((2*n-1)*sizeof(double));
        Names=(char **)malloc((n+1)*sizeof(char*));
```

```
    for (i=0;i<=n;i++)
    {
            DI[i]=(double*)malloc((n+1)*sizeof(double));
            DA[i]=(double*)malloc((n+1)*sizeof(double));
            DA1[i]=(double*)malloc((n+1)*sizeof(double));
            W[i]=(double*)malloc((n+1)*sizeof(double));
            Names[i]=(char *)malloc((50)*sizeof(char));

            if ((DI[i]==NULL)||(DA[i]==NULL)||(DA1[i]==NULL)||(W[i]==NULL)||(Names[i]==NULL))
            {
                    printf("*Data matrix is too large\n ");
                    exit(5);
            }
    }

    /* Reading the dissimilarity data file */
    for (i=1; i<=n; i++)
    {
            while (((symbol=getc(data))==' ')||(symbol=='\n')||(symbol=='\t'));
            fseek(data,-1,1);
            k=0;
while (((symbol=getc(data))!=' ')&&(symbol!='\t'))
{
    if (k<30) Names[i-1][k] = symbol;
    k++;
}
Names[i-1][k] = '\0';

for (j=1; j<=n; j++)
  fscanf(data,"%lf",&DI[i][j]);
    }

    DI[0][0]=0;
    fclose (data);
    optionTR=p_optionTR;
    /*Reconstruction method:
    1. Infer an additive (phylogenetic) tree from D or Infer an additive tree from sequences
    2. Infer a reticulogram (reticulated network) from D
    3. Infer an additive tree from an incomplété matix D*/


    /*Tree reconstruction from incomplété matrices*/
    optionMiss=p_method;
    if (optionTR==3){
            optionMiss=p_method;
            /* Reconstruction method for an incomplété dissimilarity matrix D\n(missing entries in D should
               be indicated by -99)
                    1. Triangles method - Guenoche, Leclerc (2001");
                    2. Ultrametric procedure + MW - De Soété (1984));
                    3. Additive procedure + MW - Landry et al. (1996));
                    4. MW-modified - Makarenkov (2001);
                    5. MW* - Makarenkov, Lapointe (2004)*/
            GL_Main(DI, n, optionMiss, DA, RESULTATS, ARÉTÉ, LONGUEUR, W);
    }
    /* Tree reconstruction from comlété matrices and reticulogram reconstruction*/
    else {

            method=p_method;
      /* Tree reconstruction menu (5 methods available) :
                1. ADDTREE - Sattath and Tversky (1977)
                2. Neighbor Joining - Saitou and Nei (1987)
                3. Unweighted Neighbor Joining - Gascuel (1997)
                4. Circular order reconstruction - Makarenkov, Leclerc (1997)
                5. Weighted least-squares method MW - Makarenkov, Leclerc (1999)*/

if (method==5) /* options for the MW method*/
{
        if (n>3)
        {
                Option = p_option;
          /*The MW optimization options :
                  1. Global MW optimization
                  2. Local MW optimization*/
                DI[0][0]=Option;
        }
        else
                Option=1;

        Option1 = p_option1;
/*Choice of the weight matrix W :
                1. Of the form W=1/D^p
                2. In a file */

        if (Option1==1)
        {
                Puissance = p_p;
                for (j=1;j<=n;j++)
                {
                        for (i=j+1;i<=n;i++)
                        {
                                if (DI[j][i]>0)
                                        W[j][i]=pow(1/DI[j][i],Puissance);
                                else if (DI[j][i]<=0)
                                        W[j][i]=0.000001;
                                W[i][j]=W[j][i];
                        }
                        W[j][j]=1;
                }
        }

        else if (Option1==2) /* reading W from a file*/
        {
                FILE *weights;
                if ((weights=fopen(mwFile,"r"))==0) { printf("*File %s was not found ",
                  mwFile);
                exit(8); }
                for (j=2; j<=n; j++)
                {
                        for (i=1; i<=j-1; i++)
                        {
                                fscanf(weights,"%lf",&c);
                                W[i][j]=c;
                                W[j][i]=c;
                        }
                        W[j][j]=0;
                }
                fclose (weights);
        }
}

if (method!=5) /* Setting up W matrix if MW is not selected*/
{
        for (i=1;i<=n;i++){
```

```
                for (j=i+1;j<=n;j++)
                {
                        W[i][j]=1;
                        W[j][i]=W[i][j];
                }
                W[i][i]=1;
        }
}
W[0][0]=1;

if (n==2) /* the trivial case - of two objects*/
{
        DA[1][2]=DI[1][2];
        DA[2][1]=DI[1][2];
        DA[1][1]=0.0;
        DA[2][2]=0.0;
}
else
{
        if (n<=10) l=30;
        else l=3*n;
        Iternum=10;
        if (optionTR==1)
                Scaling(DI,&power);

        /* Blocage pour attendre les autres processeurs */
        if(unproc==0)
                MPI_Barrier(MPI_COMM_WORLD);

        if  (method==1)  /*method ADDTREE*/
                ADDTREE(DI,DA,unproc);

        else if  (method==2) /*method NJ*/
                NJ(DI,DA,unproc);

        else if (method==4)  /*method Circular Orders*/
                parcour1(DI,W,DA,&Iternum,ARÉTÉ,LONGUEUR,unproc);

        else if  (method==3)  /*method UNJ*/
                UNJ(DI,DA,unproc);

        else if (method==5) /*method MW*/
        {
                if (Option==1)
                        DI[0][0]=1;
                else
                        DI[0][0]=2;
                parcour2(DI,W,DA,&Iternum,ARÉTÉ,LONGUEUR,unproc);
        }
        else if(method==6)/*method bioNJ*/
                BioNJ_main (DI,DA);

        if (method<4 || method==6) /*edge lengths polishing for ADDTREE, NJ, and UNJ*/
                approx_arb2 (DI,DA1,DA,W,&l,ARÉTÉ,LONGUEUR);
}

/* Reticulogram reconstruction option */
if (optionTR==2)
{
        int Iternumber=(2*n-2)*(2*n-2-1)/2-2*n+3;

        /*Selection of the stopping rule*/
                OptionFunction = p_optionFunction;

                /*      Stopping rule for addition of new edges to the reticulogram:
                        1. When the criterion Q1 is minimized.
                        2. When the criterion Q2 is minimized.
                        3. Add a fixed number of edges K to the reticulogram.*/

                CRITERION2 = (double*)malloc((2*n-3 + Iternumber+1)*sizeof(double));
                CRITERION1 = (double*)malloc((2*n-3 + Iternumber+1)*sizeof(double));

                free(ARÉTÉ); free(LONGUEUR);
                ARÉTÉ=(long int *) malloc(((2*n-2)*(2*n-3)+1)*sizeof(long int));
                LONGUEUR=(double *)malloc(((2*n-2)*(2*n-3)/2+1)*sizeof(double));
                RETICULATIONS (n, DI, DA, ARÉTÉ, LONGUEUR, OptionFunction, W, Iternumber,
                &ReticulationsNumber, CRITERION1, CRITERION2, unproc);
        }
        else{
                CRITERION2 = (double*)malloc(sizeof(double));
                CRITERION1 = (double*)malloc(sizeof(double));
        }
}

/* Ecriture des resultats pour le premier processeur seulement */
if(ecrireResultats && myid==0){
        printf("\n %s", stringNewick);
        SAVEASNewick(stringNewick,LONGUEUR, ARÉTÉ, Names,n) ;

        newickFile = fopen(p_newick,"w+");
        fprintf(newickFile,"%s",stringNewick);
        fclose(newickFile);

        TreeFile = fopen(treeFile,"w+");
        if (TreeFile==NULL) {printf("\nFichier %s inexistant xxxxxxxxxxxxxxx",treeFile); exit(8);}

        mod = fopen(p_modele,"r");
        if (mod==NULL) {printf("\nFichier %s inexistant",p_modele); exit(8);}

        car = fgetc(mod);
        do{
                fputc(car,TreeFile);
                car = fgetc(mod);
        }while (car!=EOF);
        fclose(mod);
        fprintf(TreeFile,"drawRetic = 1\n");
        fprintf(TreeFile,"n = %d\n",n);

        Output1=fopen(statFile,"w+");
        if(Output1 == NULL) exit(9);

        if ((method==5)&&(Option1==1))
                fprintf(Output1,"Weight matix is of the form W=1/D^%f\n",Puissance);

        if (optionTR!=3)
        {
                if (method==1) fprintf(Output1,"\nTree reconstruction method - ADDTREE\n");
                else if (method==2) fprintf(Output1,"\nTree reconstruction method - Neighbor
                                Joining\n");
                else if (method==3) fprintf(Output1,"\nTree reconstruction method - Unweighted
                                Neighbor Joining\n");
```

```
                else if (method==4) fprintf(Output1,"\nTree reconstruction method - Circular order
                                reconstruction\n");
                else if (method==5)
                {fprintf(Output1,"\nTree reconstruction method - Weighted least-squares method MW");
                        if (Option==1) fprintf(Output1," (global optimization)\n");
                        else fprintf(Output1," (local optimization)\n");
                }
                else if (method==6) fprintf(Output1,"\nTree reconstruction method - BioNJ\n");
        }
        else
        {
                fprintf(Output1,"\nTree reconstruction from incomplété dissimilarity matrix D\n\n");
                if (optionMiss==1) fprintf(Output1,"\nTree reconstruction method - Triangles
                                method\n");
                else if (optionMiss==2) fprintf(Output1,"\nTree reconstruction method - Ultrametric
                                        procedure + MW\n");
                else if (optionMiss==3) fprintf(Output1,"\nTree reconstruction method - Additive
                                        procedure + MW\n");
                else if (optionMiss==4) fprintf(Output1,"\nTree reconstruction method - MW-
                                        modified\n");
                else if (optionMiss==5) fprintf(Output1,"\nTree reconstruction method - MW*\n");
        }

        if (power!=0.0) W[0][0]=2;
        else if (PrintTreeMetric=='Y') W[0][0]=0.1;
        else W[0][0]=0;

        if (optionTR==3) { RAJ[1]=RESULTATS[0]; RAJ[2]=RESULTATS[1];
                        RAJ[3]=RESULTATS[2]; RAJ[0]=RESULTATS[3]; }
        compute_criteres(DI,DA,RAJ,W,optionTR,Names);

        if ((power!=0.0)&&(n>2))
        {
                if (optionTR==1) Scaling1(DI,DA,RAJ,LONGUEUR,&power);
                if (PrintTreeMetric=='Y') W[0][0]=0.1;
                else W[0][0]=0;
                compute_criteres(DI,DA,RAJ,W,optionTR,Names);
        }

        if (n==2) { ARÉTÉ[0]=1;ARÉTÉ[1]=2;LONGUEUR[0]=DA[1][2]; }
        if (optionTR!=2) ReticulationsNumber=2*n-3;
        PrintEdges(ARÉTÉ,LONGUEUR,ReticulationsNumber,optionTR,CRITERION1,CRITERION2,
        OptionFunction);

        fclose (Output1);
        fclose (TreeFile);
        printf("\n\n");
    }

    (*ARÉTÉ_) = ARÉTÉ;
    (*LONGUEUR_) = LONGUEUR;

} /* END trex */
```

Méthode Neighbor-Joining :

```
/*=============================================================================*
* Cette methode infere un arbre phylogenetique a partir de la matrice          *
* de distances reçue en entree, en utilisant la methode Neighbor-Joining       *
* (Saitou et Nei 1987).                                                        *
*                                                                              *
* param:                                                                       *
*                                                                              *
*       D1              : Matrice de distances                                 *
*       DA              : Arbre phylogenetique infere                          *
*       unproc          : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux)  *
*=============================================================================*/

void NJ(double **D1,double **DA, int unproc)
{
        double **D,*T1,*S,*LP,Som,Smin,Sij,L,Lii,Ljj,l1,l2,l3;
        int *T,i,j,ii,jj,n1;
        double *Tsmin;          /* Tableau des Smin necessaire a la parallélisation */
        int *Tii, *Tjj;         /* Tableau des ii et jj necessaire a la parallélisation */

        /* Allocation de la memoire */
        D=(double **) malloc((n+1)*sizeof(double*));
        T1=(double *) malloc((n+1)*sizeof(double));
        S=(double *) malloc((n+1)*sizeof(double));
        LP=(double *) malloc((n+1)*sizeof(double));
        T=(int *) malloc((n+1)*sizeof(int));

        /* Allocation juste dans le cas d'une execution en parallele*/
        if(numprocs>1 || unproc==1){
                Tii=(int *) malloc((numprocs+1)*sizeof(int));
                Tjj=(int *) malloc((numprocs+1)*sizeof(int));
                Tsmin=(double *) malloc((numprocs+1)*sizeof(double));
        }
        Smin=(double)LONG_MAX;

        /* Allocation de la memoire */
        for (i=0;i<=n;i++){
                D[i]=(double*)malloc((n+1)*sizeof(double));
                if (D[i]==NULL){
                        printf("probleme de memoire");
                        exit(5);
                }
        }
        L=0;Som=0;

        /*Initialisation des valeurs*/
        for (i=1;i<=n;i++){
                S[i]=0; LP[i]=0;
                for (j=1;j<=n;j++){
                        D[i][j]=D1[i][j];
                        S[i]=S[i]+D[i][j];
                }
                Som=Som+S[i]/2;
                T[i]=i;
                T1[i]=0;
        }
```

```
/* Boucle principale */
for (n1=n;n1>3;n1--){
        /* Recherche de la meilleure paire (i,j) pour le regroupement*/
        Smin=2*Som;
        /* Cas d'une lame*/
        if(numprocs==1 || unproc==1){
                for (i=1;i<=n1-1;i++){
                        for (j=i+1;j<=n1;j++){
                                Sij=2*Som-S[i]-S[j]-D[i][j]*(n1-2);
                                if (Sij<Smin){
                                        Smin=Sij;ii=i;jj=j;
                                }
                        }
                }
        }
        /* Cas de plusieurs processeurs */
        else{
                for (i=1;i<=n1-1;i++){
                        if(i%numprocs==myid){
                                for (j=i+1;j<=n1;j++){
                                        Sij=2*Som-S[i]-S[j]+D[i][j]*(n1-2);
                                        if (Sij<Smin){
                                                Smin = Sij;ii=i; jj=j;Tsmin[myid] = Sij;
                                                Tii[myid]=i; Tjj[myid]=j;
                                        }
                                }
                        }
                }
                /* Syncrhonisation des resultats */
                if(myid>0){
                        MPI_Send( &Smin, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Send( &ii, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                        MPI_Send( &jj, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &ii, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                        MPI_Recv( &jj, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                        if(myid<numprocs-1){
                                MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                                MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        }
                }
                else{
                        for(i=1; i<numprocs;i++){
                                MPI_Recv( &Tsmin[i], 1, MPI_DOUBLE, i, 0,
                                        MPI_COMM_WORLD, &status );
                                MPI_Recv( &Tii[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status );
                                MPI_Recv( &Tjj[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status );
                        }
                        for(i=1; i<numprocs;i++){
                                if( Tsmin[i]==Smin && ( ii>Tii[i] || (ii==Tii[i]&& jj>Tjj[i]) )){
                                        ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                                }
                                if(Tsmin[i]<Smin){
                                        ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                                }
                        }
                        MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                }
        }
```

```
        /* Nouveau regroupement*/
        Lii=(D[ii][jj]+(S[ii]-S[jj])/(n1-2))/2-LP[ii];
        Ljj=(D[ii][jj]+(S[jj]-S[ii])/(n1-2))/2-LP[jj];

        /* Mise a jour de D */
        if (Lii<0.00001)
                Lii=0.00005;
        if (Ljj<0.00001)
                Ljj=0.00005;
        L=L+Lii+Ljj;
        LP[ii]=0.5*D[ii][jj];
        Som=Som-(S[ii]+S[jj])/2;
        for (i=1;i<=n1;i++){
                if ((i!=ii)&&(i!=jj)){
                        S[i]=S[i]-0.5*(D[i][ii]+D[i][jj]);
                        D[i][ii]=(D[i][ii]+D[i][jj])/2;
                        D[ii][i]=D[i][ii];
                }
        }
        D[ii][ii]=0;
        S[ii]=0.5*(S[ii]+S[jj])-D[ii][jj];
        if (jj!=n1){
                for (i=1;i<=n1-1;i++){
                        D[i][jj]=D[i][n1];
                        D[jj][i]=D[n1][i];
                }
                D[jj][jj]=0;
                S[jj]=S[n1];
                LP[jj]=LP[n1];
        }
        /* Mise a jour de DA */
        for (i=1;i<=n;i++){
                if (T[i]==ii)
                        T1[i]=T1[i]+Lii;
                if (T[i]==jj)
                        T1[i]=T1[i]+Ljj;
        }
        for (j=1;j<=n;j++){
                if (T[j]==jj){
                        for (i=1;i<=n;i++){
                                if (T[i]==ii){
                                        DA[i][j]=T1[i]+T1[j];
                                        DA[j][i]=DA[i][j];
                                }
                        }
                }
        }
        for (j=1;j<=n;j++)
                if (T[j]==jj)  T[j]=ii;

        if (jj!=n1){
                for (j=1;j<=n;j++)
                        if (T[j]==n1) T[j]=jj;
        }
}

/*Regroupement des 3 objets restants */
l1=(D[1][2]+D[1][3]-D[2][3])/2-LP[1];
l2=(D[1][2]+D[2][3]-D[1][3])/2-LP[2];
l3=(D[1][3]+D[2][3]-D[1][2])/2-LP[3];
```

```
        if (l1<0.00001)
                l1=0.00005;
        if (l2<0.00001)
                l2=0.00005;
        if (l3<0.00001)
                l3=0.00005;
        L=L+l1+l2+l3;

        for (j=1;j<=n;j++){
                for (i=1;i<=n;i++){
                        if ((T[j]==1)&&(T[i]==2)){
                                DA[i][j]=T1[i]+T1[j]+l1+l2;
                                DA[j][i]=DA[i][j];
                        }
                        if ((T[j]==1)&&(T[i]==3)){
                                DA[i][j]=T1[i]+T1[j]+l1+l3;
                                DA[j][i]=DA[i][j];
                        }
                        if ((T[j]==2)&&(T[i]==3)){
                                DA[i][j]=T1[i]+T1[j]+l2+l3;
                                DA[j][i]=DA[i][j];
                        }
                }
                DA[j][j]=0;
        }

        free(T);
        free(T1);
        free(S);
        free(LP);
        for (i=0;i<=n;i++){
                free(D[i]);
        }
        free(D);
} /* END NJ */
```

Méthode Unweighted Neighbor-Joining :

```
/*=============================================================================*
 * Cette methode infere un arbre phylogenetique a partir de la matrice         *
 * de distances reçue en entree, en utilisant la methode Unweighted            *
 * Neighbor-Joining (Gascuel 1997).                                            *
 *                                                                             *
 * param:                                                                      *
 *      D1          : Matrice de distances                                     *
 *      DA          : Arbre phylogenetique infere                              *
 *      unproc      : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux)  *
 *=============================================================================*/


void UNJ(double **D1,double **DA, int unproc)
{
        double **D,*T1,*S,*LP,Som,Smin,Sij,L,Lii,Ljj,l1,l2,l3,S1,*NUMBER;
        int *T,i,j,ii,jj,n1;

        double *Tsmin; /* Tableau des Smin necessaire a la parallélisation */
        int *Tii, *Tjj; /* Tableau des ii et jj necessaire a la parallélisation */
```

```
D=(double **) malloc((n+1)*sizeof(double*));
T1=(double *) malloc((n+1)*sizeof(double));
S=(double *) malloc((n+1)*sizeof(double));
LP=(double *) malloc((n+1)*sizeof(double));
T=(int *) malloc((n+1)*sizeof(int));
NUMBER=(double *) malloc((n+1)*sizeof(double));

/* Allocation juste dans le cas d'une execution en parallele*/
if(numprocs>1 || unproc==1){
        Tjj=(int *) malloc((numprocs+1)*sizeof(int));
        Tii=(int *) malloc((numprocs+1)*sizeof(int));
        Tsmin=(double *) malloc((numprocs+1)*sizeof(double));
}
Smin=(double)LONG_MAX;

for (i=0;i<=n;i++){
        D[i]=(double*)malloc((n+1)*sizeof(double));
        if (D[i]==NULL){
                printf("probleme de memoire");
                exit(5);
        }
}
L=0;
Som=0;
for (i=1;i<=n;i++){
        S[i]=0; LP[i]=0; NUMBER[i]=1.0;
        for (j=1;j<=n;j++){
                D[i][j]=D1[i][j];
                S[i]=S[i]+D[i][j];
        }
        T[i]=i;
        T1[i]=0;
}

/* Boucle Principale*/
for (n1=n;n1>3;n1--){
        /* Recherche de la paire d'objets (i,j) a regrouper  */
        Smin=-S[1]-S[2]+D[1][2]*(n1-2);

        /* Cas d'un seul processeur */
        if(numprocs==1 || unproc==1){
                for (i=1;i<=n1-1;i++){
                        for (j=i+1;j<=n1;j++){
                                Sij=-S[i]-S[j]+D[i][j]*(n1-2);
                                if (Sij<=Smin){
                                        Smin=Sij;ii=i;jj=j;
                                }
                        }
                }
        }


        /* Cas de plusieurs processeurs */
        else{
                for (i=1;i<=n1-1;i++){
                        if(i%numprocs==myid){
                                for (j=i+1;j<=n1;j++){
                                        Sij=-S[i]-S[j]+D[i][j]*(n1-2);
                                        if (Sij<=Smin){
                                                Smin = Sij;ii=i; jj=j;Tsmin[myid] = Sij;
                                                Tii[myid]=i; Tjj[myid]=j;
                                        }
```

```
                        }
                }
        }

        /* Synchronisation */
        if(myid>0){
                MPI_Send( &Smin, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                MPI_Send( &ii, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Send( &jj, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Recv( &ii, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                MPI_Recv( &jj, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                if(myid<numprocs-1){
                        MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                }
        }
        else{
                for(i=1; i<numprocs;i++){
                        MPI_Recv( &Tsmin[i], 1, MPI_DOUBLE, i, 0,
                                        MPI_COMM_WORLD, &status );
                        MPI_Recv( &Tii[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status );
                        MPI_Recv( &Tjj[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status );
                }
                for(i=1; i<numprocs;i++){
                        if(Tsmin[i]<=Smin){
                                ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                        }
                        if( Tsmin[i]==Smin && ( ii<Tii[i] || (ii==Tii[i]&& jj<Tjj[i]) )){
                                ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                        }
                }
                MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
        }
}
S1=0;

/* Mise a jour de D*/
for (i=1;i<=n1;i++){
        if ((i!=ii)&&(i!=jj))
                S1=S1+NUMBER[i]*(D[ii][i]-D[jj][i]);
}
Lii=S1/(2.0*(n-NUMBER[ii]-NUMBER[jj]))+0.5*D[ii][jj];
Ljj=-S1/(2.0*(n-NUMBER[ii]-NUMBER[jj]))+0.5*D[ii][jj];

if (Lii<0.00001) Lii=0.00005;
if (Ljj<0.00001) Ljj=0.00005;
L=L+Lii+Ljj;

S[ii]=0;
for (i=1;i<=n1;i++){
        if ((i!=ii)&&(i!=jj)){
                S1=D[i][ii];
                D[i][ii]=(NUMBER[ii]*(D[i][ii]-Lii)+NUMBER[jj]*(D[i][jj]-
                                Ljj))/(NUMBER[ii]+NUMBER[jj]);
                S[i]=S[i]+D[i][ii]-(D[i][jj]+S1);
                D[ii][i]=D[i][ii];
                S[ii]=S[ii]+D[i][ii];
        }
}
```

```
        S1=S[ii];
        D[ii][ii]=0;
        NUMBER[ii]=NUMBER[ii]+NUMBER[jj];

        if (jj!=n1){
                for (i=1;i<=n1-1;i++){
                        D[i][jj]=D[i][n1];
                        D[jj][i]=D[n1][i];
                }
                D[jj][jj]=0;
                S[jj]=S[n1];
                NUMBER[jj]=NUMBER[n1];
        }

        /* Mise a jour de DA */
        for (i=1;i<=n;i++){
                if (T[i]==ii) T1[i]=T1[i]+Lii;
                if (T[i]==jj) T1[i]=T1[i]+Ljj;
        }

        for (j=1;j<=n;j++){
                if (T[j]==jj){
                        for (i=1;i<=n;i++){
                                if (T[i]==ii){
                                        DA[i][j]=T1[i]+T1[j];
                                        DA[j][i]=DA[i][j];
                                }
                        }
                }
        }

        for (j=1;j<=n;j++)
                if (T[j]==jj)  T[j]=ii;
        if (jj!=n1){
                for (j=1;j<=n;j++)
                        if (T[j]==n1) T[j]=jj;
        }
}

/*Regroupement des trois derniers objets */

l1=(D[1][2]+D[1][3]-D[2][3])/2;
l2=(D[1][2]+D[2][3]-D[1][3])/2;
l3=(D[1][3]+D[2][3]-D[1][2])/2;
if (l1<0.00001) l1=0.00005;
if (l2<0.00001) l2=0.00005;
if (l3<0.00001) l3=0.00005;
L=L+l1+l2+l3;


for (j=1;j<=n;j++){
        for (i=1;i<=n;i++){
                if ((T[j]==1)&&(T[i]==2)){
                        DA[i][j]=T1[i]+T1[j]+l1+l2;
                        DA[j][i]=DA[i][j];
                }
                if ((T[j]==1)&&(T[i]==3)){
                        DA[i][j]=T1[i]+T1[j]+l1+l3;
                        DA[j][i]=DA[i][j];
                }
                if ((T[j]==2)&&(T[i]==3)){
                        DA[i][j]=T1[i]+T1[j]+l2+l3;
```

```
                        DA[j][i]=DA[i][j];
                    }
                }
                DA[j][j]=0;
        }
        free(T);
        free(T1);
        free(S);
        free(LP);
        free(NUMBER);
        for (i=0;i<=n;i++){
                free(D[i]);
        }
        free(D);
} /* END UNJ */
```

Méthode ADDTREE :

```
/*=============================================================================*
 * Cette methode infere un arbre phylogenetique a partir de la matrice         *
 * de distances reçue en entree, en utilisant la methode ADDTREE               *
 * (Sattath et Tversky 1977).                                                  *
 *                                                                             *
 * param:                                                                      *
 *      D1              : Matrice de distances                                 *
 *      DA              : Arbre phylogenetique infere                          *
 *      unproc          : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux)  *
 *=============================================================================*/
void ADDTREE(double **D1,double **DA, int unproc)
{
        double Som,Smin,Sij,L,Lii,Ljj,l1,l2,l3;
        int *T,i,j,ii,jj,n1,i1,j1;
        double **D,*LP,*S,*T1;
        double *Tsmin; /* Tableau des Smin necessaire a la parallélisation */
        int *Tii, *Tjj; /* Tableau des ii et jj necessaire a la parallélisation */


        D=(double **) malloc((n+1)*sizeof(double*));
        T1=(double *) malloc((n+1)*sizeof(double));
        S=(double *) malloc((n+1)*sizeof(double));
        LP=(double *) malloc((n+1)*sizeof(double));
        T=(int *) malloc((n+1)*sizeof(int));

        /* Allocation juste dans le cas d'une execution en parallele*/
        if(numprocs>1 || unproc==1){
                Tii=(int *) malloc((numprocs+1)*sizeof(int));
                Tjj=(int *) malloc((numprocs+1)*sizeof(int));
                Tsmin=(double *) malloc((numprocs+1)*sizeof(double));
        }
        Smin=(double)LONG_MAX;

        for (i=0;i<=n;i++)
        {
                D[i]=(double*)malloc((n+1)*sizeof(double));
                if (D[i]==NULL){
                        printf("probleme de memoire");
                        exit(5);
                }
        }
```

```
L=0;Som=0;
for (i=1;i<=n;i++){
        S[i]=0; LP[i]=0;
        for (j=1;j<=n;j++){
                D[i][j]=D1[i][j];
                S[i]=S[i]+D[i][j];
        }
        Som=Som+S[i]/2.0;
        T[i]=i;
        T1[i]=0;
}

/* Boucle principale */
for (n1=n;n1>3;n1--){
/* Recherche de la paire (i,j) pour le regroupement*/
        Smin=0.0;

        /* Cas d'une lame*/
        if(numprocs==1 || unproc==1){
                for (i=1;i<=n1-1;i++){
                        for (j=i+1;j<=n1;j++){
                                Sij=0.0;
                                for (i1=1;i1<=n1-1;i1++){
                                        if ((i1!=i)&&(i1!=j)){
                                                for (j1=i1+1;j1<=n1;j1++){
                                                        if ((j1!=i)&&(j1!=j)){
                                                                if ((D[i][i1]+D[j][j1]-D[i][j]-
                                                                        D[i1][j1]>=0)&&
                                                                        (D[i][j1]+D[j][i1]-
                                                                        D[i][j]-D[i1][j1]>=0))
                                                                        Sij=Sij+1.0;
                                                        }
                                                }
                                        }
                                }
                                if (Sij>Smin){
                                        Smin = Sij;ii=i; jj=j;
                                }
                        }
                }
        }
        /* Cas de plusieurs processeurs */
        else{
                for (i=1;i<=n1-1;i++){
                        if(i%numprocs==myid){
                                for (j=i+1;j<=n1;j++){
                                        Sij=0.0;
                                        for (i1=1;i1<=n1-1;i1++){
                                                if ((i1!=i)&&(i1!=j)){
                                                        for (j1=i1+1;j1<=n1;j1++){
                                                                if ((j1!=i)&&(j1!=j)){
                                                                        if ((D[i][i1]+D[j][j1]-
                                                                        D[i][j]-D[i1][j1]>=0)
                                                                        &&(D[i][j1]+D[j][i1]-
                                                                        D[i][j]-D[i1][j1]>=0))
                                                                                Sij=Sij+1.0;
                                                                }
                                                        }
                                                }
                                        }
                                        if (Sij>Smin){
```

```
                                        Smin = Sij;ii=i; jj=j;Tsmin[myid] = Sij;
                                        Tii[myid]=i; Tjj[myid]=j;
                                }
                        }
                }
        }
        /* Synchronisation des resultats */
        if(myid>0){
                MPI_Send( &Smin, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                MPI_Send( &ii, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Send( &jj, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Recv( &ii, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                MPI_Recv( &jj, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                if(myid<numprocs-1){
                        MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                }
        }
        else{
                for(i=1; i<numprocs;i++){
                        MPI_Recv( &Tsmin[i], 1, MPI_DOUBLE, i, 0,
                                MPI_COMM_WORLD, &status );
                        MPI_Recv( &Tii[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                &status );
                        MPI_Recv( &Tjj[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                &status );
                }
                for(i=1; i<numprocs;i++){
                        if( Tsmin[i]==Smin && ( ii>Tii[i] || (ii==Tii[i]&& jj>Tjj[i]) )){
                                ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                        }
                        if(  Tsmin[i]> Smin ){
                                ii = Tii[i];jj = Tjj[i];Smin = Tsmin[i];
                        }
                }
                MPI_Send( &ii, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                MPI_Send( &jj, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
        }
}
/* New clustering */
Lii=(D[ii][jj]+(S[ii]-S[jj])/(n1-2.0))/2-LP[ii];
Ljj=(D[ii][jj]+(S[jj]-S[ii])/(n1-2.0))/2-LP[jj];
if (Lii<0.00001) Lii=0.00005;
if (Ljj<0.00001) Ljj=0.00005;
L=L+Lii+Ljj;
LP[ii]=0.5*D[ii][jj];

Som=Som-(S[ii]+S[jj])/2.0;
/* Mise a jour de D*/
for (i=1;i<=n1;i++){
        if ((i!=ii)&&(i!=jj)){
                S[i]=S[i]-0.5*(D[i][ii]+D[i][jj]);
                D[i][ii]=(D[i][ii]+D[i][jj])/2;
                D[ii][i]=D[i][ii];
        }
}
D[ii][ii]=0;
S[ii]=0.5*(S[ii]+S[jj])-D[ii][jj];
if (jj!=n1){
        for (i=1;i<=n1-1;i++){
                D[i][jj]=D[i][n1];
                D[jj][i]=D[n1][i];
                }
                D[jj][jj]=0;
                S[jj]=S[n1];
                LP[jj]=LP[n1];
        }
        /* Mise a jour de DA */
        for (i=1;i<=n;i++){
                if (T[i]==ii) T1[i]=T1[i]+Lii;
                if (T[i]==jj) T1[i]=T1[i]+Ljj;
        }
        for (j=1;j<=n;j++){
                if (T[j]==jj){
                        for (i=1;i<=n;i++){
                                if (T[i]==ii){
                                        DA[i][j]=T1[i]+T1[j];
                                        DA[j][i]=DA[i][j];
                                }
                        }
                }
        }

        for (j=1;j<=n;j++)
                if (T[j]==jj)  T[j]=ii;

        if (jj!=n1){
                for (j=1;j<=n;j++)
                        if (T[j]==n1) T[j]=jj;
        }
}

/*3 objects remaining*/

l1=(D[1][2]+D[1][3]-D[2][3])/2.0-LP[1];
l2=(D[1][2]+D[2][3]-D[1][3])/2.0-LP[2];
l3=(D[1][3]+D[2][3]-D[1][2])/2.0-LP[3];
if (l1<0.00001) l1=0.00005;
if (l2<0.00001) l2=0.00005;
if (l3<0.00001) l3=0.00005;

L=L+l1+l2+l3;

for (j=1;j<=n;j++){
        for (i=1;i<=n;i++){
                if ((T[j]==1)&&(T[i]==2)){
                        DA[i][j]=T1[i]+T1[j]+l1+l2;
                        DA[j][i]=DA[i][j];
                }
                if ((T[j]==1)&&(T[i]==3)){
                        DA[i][j]=T1[i]+T1[j]+l1+l3;
                        DA[j][i]=DA[i][j];
                }
                if ((T[j]==2)&&(T[i]==3)){
                        DA[i][j]=T1[i]+T1[j]+l2+l3;
                        DA[j][i]=DA[i][j];
                }
        }
        DA[j][j]=0;
}
free(T);
free(T1);
free(S);
free(LP);
```

```
        for (i=0;i<=n;i++){
                free(D[i]);
        }
        free(D);
} /* END ADDTREE */
```

Méthode Circular order reconstruction:

```
/*=============================================================================*
 * Cette methode infere un arbre phylogenetique a partir de la matrice         *
 * de distances reçue en entree, en utilisant la methode de reconstruction     *
 * par ordre circulaire (Makarenkov et Leclerc 1997).                          *
 *                                                                             *
 * param:                                                                      *
 *      DISS        : Matrice de distances                                     *
 *      W           : Matrice de poids                                         *
 *      TM          : Arbre phylogenetique infere                              *
 *      Iternum     : Nombre d'interation dans la procedure Gauss-Seidel appliquee *
 *      ARRÊTÉ      : Tableau d'aretes (des branches de l'arbre)               *
 *      LONGUEUR    : Tableau des longueurs des aretes                         *
 *      unproc      : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux) *
 =============================================================================*/

void parcour1(double **DISS,double **W,double **TM,int *Iternum,long int *ARÉTÉ, double *LONGUEUR, int
unproc)
{
        int i,j,i1,j1,iopt,jopt,*X,Iternumber;
        double **TM1;
        double EQ,EQmin;

        double *Teqmin; /* Tableau des EQmin necessaire a la parallélisation */
        int *Tiopt, *Tjopt; /* Tableau des iopt et jopt necessaire a la parallélisation */


        X=(int *) malloc((n+1)*sizeof(int));
        TM1=(double **) malloc((n+1)*sizeof(double*));
        /* Allocation juste dans le cas d'une execution en parallele*/
        if(numprocs>1 || unproc==1){
                Tiopt=(int *) malloc((numprocs+1)*sizeof(int));
                Tjopt=(int *) malloc((numprocs+1)*sizeof(int));
                Teqmin =(double*)malloc((numprocs+1)*sizeof(double));
        }
        EQmin=(double)LONG_MAX;


        for (i=0;i<=n;i++){
                TM1[i]=(double*)malloc((n+1)*sizeof(double));
                if (TM1[i]==NULL){
                        printf("probleme de memoire");
                        exit(5);
                }
        }

        Iternumber=*Iternum;
        /* Cas d'une lame*/
        if(numprocs == 1 || unproc==1){
                for (i=1;i<=n;i++){
                        for (j=1;j<=n;j++){
                                if (i!=j){
                                        odp(DISS,X,&i,&j);
                                        construction2(DISS,TM1,X);
                                        approx_arb2 (DISS,TM1,TM,W,&Iternumber,ARÉTÉ,LONGUEUR);
                                        EQ=0.0;
                                        for (i1=1;i1<=n;i1++){
                                                for (j1=i1+1;j1<=n;j1++)
                                                        EQ=EQ+W[i1][j1]*(DISS[i1][j1]-
                                                        TM[i1][j1])*(DISS[i1][j1]-TM[i1][j1]);
                                        }
                                        if (((i==1)&&(j==2))||(EQ<EQmin)){
                                                EQmin=EQ;iopt=i;jopt=j;
                                        }
                                }
                        }
                }
        }
        /* Cas de plusieurs processeurs */
        else{
                for (i=1;i<=n;i++){
                        if(i%numprocs == myid){
                                for (j=1;j<=n;j++){
                                        if (i!=j){
                                                odp(DISS,X,&i,&j);
                                                construction2(DISS,TM1,X);
                                                approx_arb2 (DISS,TM1,TM,W,
                                                        &Iternumber,ARÉTÉ,LONGUEUR);
                                                EQ=0.0;
                                                for (i1=1;i1<=n;i1++){
                                                        for (j1=i1+1;j1<=n;j1++)
                                                                EQ=EQ+W[i1][j1]*(DISS[i1][j1]-
                                                                TM[i1][j1])*(DISS[i1][j1]-TM[i1][j1]);
                                                }
                                                if (((i==1)&&(j==2))||(EQ<EQmin)){
                                                        EQmin=EQ;iopt=i;jopt=j;Teqmin[myid]=EQ;
                                                        Tiopt[myid]=i;Tjopt[myid]=j;
                                                }
                                        }
                                }
                        }
                }
                /* Synchronisation des resultats */
                if(myid>0){
                        MPI_Send( &EQmin, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Send( &iopt, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                        MPI_Send( &jopt, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &iopt, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                        MPI_Recv( &jopt, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                        if(myid<numprocs-1){
                                MPI_Send( &iopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                                MPI_Send( &jopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        }
                }
                else{
                        for(i=1; i<numprocs;i++){
                                MPI_Recv( &Teqmin[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
                                        &status );
                                MPI_Recv( &Tiopt[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status );
                                MPI_Recv( &Tjopt[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status );

                        }
                        for(i=1; i<numprocs;i++){
                                if (((Tiopt[i]==1)&&(Tjopt[i]==2))||(Teqmin[i]<EQmin)){
```

```
                                        EQmin=Teqmin[i];iopt=Tiopt[i];jopt=Tjopt[i];
                                }
                        }
                        MPI_Send( &iopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        MPI_Send( &jopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                }

        }
        if (n>10) Iternumber=2*n-3;
        else Iternumber=15;
        odp(DISS,X,&iopt,&jopt);
        construction2(DISS,TM1,X);
        approx_arb2 (DISS,TM1,TM,W,&Iternumber,ARÉTÉ,LONGUEUR);

        free(X);
        for (i=0;i<=n;i++)
                free(TM1[i]);
        free(TM1);
}/* END parcour1 */
```

Méthode MW

```
/*=============================================================================
* Cette methode infere un arbre phylogenetique a partir de la matrice          *
* de distances reçue en entree, en utilisant la methode Weighted               *
* Least-Squares (Makarenkov et Leclerc 1999).                                  *
*                                                                              *
* param:                                                                       *
*       DISS            : Matrice de distances                                 *
*       W               : Matrice de poids                                     *
*       TM              : Arbre phylogenetique infere                          *
*       Iternum         : Nombre d'interation dans la procedure Gauss-Seidel appliquee *
*       ARRÉTÉ          : Tableau d'aretes (des branches de l'arbre)           *
*       LONGUEUR        : Tableau des longueurs des aretes                     *
*       unproc          : 1 (vrai) si le nombre de processeur utilise est 1, sinon 0 (faux) *
=============================================================================*/

void parcour2(double **DISS,double **W,double **TM,int *Iternum,long int *ARÉTÉ, double *LONGUEUR, int
unproc)
{
        int i,j,i1,j1,iopt,jopt,*X,Iternumber,Option;
        double **TM1;
        double EQ,EQmin;
        int *Tiopt, *Tjopt; /* Tableau des iopt et jopt necessaire a la parallélisation */
        double *Teqmin;   /* Tableau des Teqmin necessaire a la parallélisation */

        X=(int *) malloc((n+1)*sizeof(int));
        TM1=(double **) malloc((n+1)*sizeof(double*));
        /* Allocation juste dans le cas d'une execution en parallele*/
        if(numprocs>1 || unproc==1){
                Tiopt=(int *) malloc((numprocs+1)*sizeof(int));
                Tjopt=(int *) malloc((numprocs+1)*sizeof(int));
                Teqmin=(double*)malloc((numprocs+1)*sizeof(double));
        }
        EQmin=(double)LONG_MAX;


        for (i=0;i<=n;i++){
                TM1[i]=(double*)malloc((n+1)*sizeof(double));
        if (TM1[i]==NULL){
                printf("probleme de memoire");
                exit(5);
        }
}

Iternumber=*Iternum;
Option=floor1(DISS[0][0]);
X[0]=1;

/* Cas d'une lame*/
if(numprocs==1 || unproc==1){
        for (i=1;i<=n;i++){
                for (j=i+1;j<=n;j++)  {
                        X[1]=i;
                        X[2]=j;
                        construction(DISS,TM1,X,W);
                        approx_arb2 (DISS,TM1,TM,W,&Iternumber,ARÉTÉ,LONGUEUR);
                        EQ=0.0;
                        for (i1=1;i1<=n;i1++){
                                for (j1=i1+1;j1<=n;j1++)
                                        EQ=EQ+W[i1][j1]*(DISS[i1][j1]-TM[i1][j1])*
                                           (DISS[i1][j1]-TM[i1][j1]);
                        }
                        if (((i==1)&&(j==2))||(EQ<EQmin)){
                                EQmin=EQ;iopt=i;jopt=j;
                        }
                        if (Option==2) break;
                }
        }
}
/* Cas de plusieurs processeurs */
else{
        for (i=1;i<=n;i++){
                if(i%numprocs == myid){
                        for (j=i+1;j<=n;j++){
                                X[1]=i;
                                X[2]=j;
                                construction(DISS,TM1,X,W);
                                approx_arb2 (DISS,TM1,TM,W,&Iternumber,ARÉTÉ,LONGUEUR);
                                EQ=0.0;
                                for (i1=1;i1<=n;i1++){
                                        for (j1=i1+1;j1<=n;j1++)
                                        EQ=EQ+W[i1][j1]*(DISS[i1][j1]-TM[i1][j1])*
                                           (DISS[i1][j1]-TM[i1][j1]);
                                }
                                if (((i==1)&&(j==2))||(EQ<EQmin)){

EQmin=EQ;iopt=i;jopt=j;Teqmin[myid]=EQ;Tiopt[myid]=i;Tjopt[myid]=j;
                                }
                                if (Option==2) break;
                        }
                }
        }
        /* Synchronisation des resultats */
        if(myid>0){
                MPI_Send( &EQmin, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                MPI_Send( &iopt, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Send( &jopt, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                MPI_Recv( &iopt, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
                MPI_Recv( &jopt, 1, MPI_INT, myid-1, 0, MPI_COMM_WORLD, &status );
```

```
                        if(myid<numprocs-1){
                                MPI_Send( &iopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                                MPI_Send( &jopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        }
                }
                else{
                        for(i=1; i<numprocs;i++){
                                MPI_Recv( &Teqmin[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
                                        &status );
                                MPI_Recv( &Tiopt[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status );
                                MPI_Recv( &Tjopt[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status );

                        }
                        for(i=1; i<numprocs;i++){
                                if (((Tiopt[i]==1)&&(Tjopt[i]==2))||(Teqmin[i]<EQmin)){
                                        EQmin=Teqmin[i];iopt=Tiopt[i];jopt=Tjopt[i];
                                }
                        }
                        MPI_Send( &iopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                        MPI_Send( &jopt, 1, MPI_INT,myid+1, 0, MPI_COMM_WORLD );
                }
        }
        if (n>10) Iternumber=2*n-3;
        else Iternumber=15;
        X[1]=iopt;
        X[2]=jopt;
        construction(DISS,TM1,X,W);
        approx_arb2 (DISS,TM1,TM,W,&Iternumber,ARÊTÉ,LONGUEUR);

        free(X);
        for (i=0;i<=n;i++)
        free(TM1[i]);
        free(TM1);
} /* END parcour2 */
```

Méthode BioNJ :

```
/*=============================================================================*
*                                                                             *
* Cette methode infere un arbre phylogenetique a partir de la matrice         *
* de distances reçue en entree, en utilisant la methode BioNJ(Gascuel 1997)   *
*                                                                             *
*               Main program                                                  *
*                                                                             *
*               argc is the number of arguments                               *
*               **argv contains the arguments:                                *
*               the first argument has to be BIONJ;                           *
*               the second is the inptu-file;                                 *
*               the third is the output-file.                                 *
*               When the input and output files are                           *
*               not given, the user is asked for them.                        *
*                                                                             *
\*=============================================================================*/
void BioNJ_main (double **DI, double **DA)
{
        FILE *input=NULL;               /* pointer to input file*/
        FILE *output=NULL;              /* pointer to output file */
        POINTERS *trees=NULL;           /* list of subtrees*/
        char *Name_fich1;               /* name of the input file*/
```

```
char *Name_fich2;               /* name of the output file*/
char *chain1;                   /* stringized branch-length*/
char *chain2;                   /* idem                    */
int *a, *b;                     /* pair to be agglomerated*/
float **delta;                  /* delta matrix*/
float la;                       /* first taxon's branch-length*/
float lb;                       /* second taxon's branch-length*/
float vab;                      /* variance of Dab*/
float lamda;
int i;
int ok;
int r;                          /*number of subtrees*/
int x, y;
clock_t clock_start, clock_end;
double t;
int Pr,j;
int last1, last2, last3;

double *T1,L,Lii,Ljj,l1,l2,l3;
int *T,ii,jj,n1;

T1=(double *) malloc((n+1)*sizeof(double));
T=(int *) malloc((n+1)*sizeof(int));

for (i=1;i<=n;i++)
{
        T[i]=i;
        T1[i]=0.0;
}

/*  Allocation of memories    */

a=(int*)calloc(1,sizeof(int));
b=(int*)calloc(1,sizeof(int));

/*    Create the delta matrix    */

delta=(float **)calloc(n+1,sizeof(float*));
for(i=1; i<= n; i++)
{
        delta[i]=(float *)calloc(n+1, sizeof(float));
        if(delta[i] == NULL)
        {
                { printf("Data matrix is too large - BioNJ"); /*m_error=-1;*/ return;}
        }
}

for(i=1; i <= n; i++)
{
        for(j=1; j <= n; j++)
                delta[i][j]=DI[i][j];
}

for(i=1; i<=n; i++)
        delta[i][0]=0.0;


/*  initialise and symmetrize the running delta matrix    */
r=n;
*a=0;
*b=0;
```

```
while (r > 3)                                   /* until r=3            */
{
        Compute_sums_Sx(delta, n);              /* compute the sum Sx   */
        Best_pair(delta, r, a, b, n);           /* find the best pair by */
        vab=Variance(*a, *b, delta);            /* minimizing (1)       */
        la=Branch_length(*a, *b, delta, r);     /* compute branch-lengths */
        lb=Branch_length(*b, *a, delta, r);     /* using formula (2)    */
        lamda=Lamda(*a, *b, vab, delta, n, r);  /* compute lambda* using (9)*/

        for(i=1; i <= n; i++)
        {
                if(!Emptied(i,delta) && (i != *a) && (i != *b))
                {
                        if(*a > i)
                        {
                                x=*a;
                                y=i;
                        }
                        else
                        {
                                x=i;
                                y=*a;   /* apply reduction formulae */
                        }               /* 4 and 10 to delta        */
                        delta[x][y]=Reduction4(*a, la, *b, lb, i, lamda, delta);
                        delta[y][x]=Reduction10(*a, *b, i, lamda, vab, delta);
                }
        }

        /* Mise a jour de DA */
        if (la<0.00001) la=0.00001;
        if (lb<0.00001) lb=0.00001;
        n1=r; ii=*b; jj=*a; Lii=lb; Ljj=la;

        for (i=1;i<=n;i++)
        {
                if (T[i]==ii) T1[i]=T1[i]+Lii;
                if (T[i]==jj) T1[i]=T1[i]+Ljj;
        }
        for (j=1;j<=n;j++)
        {
                if (T[j]==jj)
                {
                        for (i=1;i<=n;i++)
                        {
                                if (T[i]==ii)
                                {
                                        DA[i][j]=T1[i]+T1[j];
                                        DA[j][i]=DA[i][j];
                                }
                        }
                }
        }

        for (j=1;j<=n;j++)
                if (T[j]==ii) T[j]=jj;
        /* end of bloc update DA */
        delta[*b][0]=1.0;                       /* make the b line empty */
        r=r-1;                                  /* decrease r           */
}

Finish(delta, n, trees, output,&l1,&l2,&l3,&last1,&last2,&last3);   /* compute the branch-lengths*/
        if (l1<0.00001) l1=0.00001;
        if (l2<0.00001) l2=0.00001;
        if (l3<0.00001) l3=0.00001;

        /*Il reste 3 sommets */
        for (j=1;j<=n;j++)
        {
                for (i=1;i<=n;i++)
                {
                        if ((T[j]==last1)&&(T[i]==last2))
                        {
                                DA[i][j]=T1[i]+T1[j]+l1+l2;
                                DA[j][i]=DA[i][j];
                        }
                        if ((T[j]==last1)&&(T[i]==last3))
                        {
                                DA[i][j]=T1[i]+T1[j]+l1+l3;
                                DA[j][i]=DA[i][j];
                        }
                        if ((T[j]==last2)&&(T[i]==last3))
                        {
                                DA[i][j]=T1[i]+T1[j]+l2+l3;
                                DA[j][i]=DA[i][j];
                        }
                }
                DA[j][j]=0;
        }

        free(T);
        free(T1);
        /* end of bloc Il reste 3 sommets */

        free(delta);
        return;
} /* END BioNJ_main */
```

Méthode de Bootstrap :

```
/*************************************************************
 * Algorithme de bootstrap                                   *
 * 1. sequence              -> matrice de distances (D)      *
 * 2. matrice de distances  -> arbre A(arete,longueur)       *
 * 3. bipartition(arete)                                     *
 * .....                                                     *
 *************************************************************/
int bootstrap_function(){

        /* Calcul pour la sequence initiale*/
        long int *ARÉTÉ;
        int *tabComparaison;
        int **TtabComparaison; /* Matrice de comparaison necessaire pour le parallelisme */
        double *LONGUEUR,Pr;
        int intNbreEspeces,intNbreSites,i,r,intBootstrap,intRetour=0;
        char **matSites;
        double **matDistances;
        char **toto;
        int intDimTab;
        double *m_Dl;
        int ** matBiPartition;
```

```
int ** matBiPartitionTmp;
char **matNomsEspeces;
int j,k,p,m = 0;

int tour;
FILE *bs;
int blnExact, blnExactInvers;

FILE * tmp = fopen(p_input,"r");
fscanf(tmp,"%d",&intNbreEspeces);
fscanf(tmp,"%d",&intNbreSites);
fclose(tmp);
if(intNbreEspeces<numprocs)
        maxprocs=intNbreEspeces;
else
        maxprocs = numprocs;
seqToDistance(p_input,&matDistances,1);
/* Appel de la fonction trex en mode multiprocesseur */
trex(p_matrix,"","",&ARÉTÉ,&LONGUEUR,0,0);

matBiPartition = (int **) malloc(((2*intNbreEspeces-3)+2)*sizeof(int*));
matBiPartitionTmp = (int **) malloc(((2*intNbreEspeces-3)+2)*sizeof(int*));
for (i=0;i<=2*intNbreEspeces-3+1;i++){
        matBiPartition[i]=(int *) malloc(((2*intNbreEspeces-3)+2)*sizeof(int));
        matBiPartitionTmp[i]=(int *) malloc(((2*intNbreEspeces-3)+2)*sizeof(int));
}
for (i=1; i<=2*intNbreEspeces-3; i++)
        for (j=1; j<=2*intNbreEspeces-3; j++)
                matBiPartitionTmp[i][j] = 0;
matNomsEspeces=(char **)malloc((intNbreEspeces + 1)*sizeof(char*));
for (i = 0; i <= intNbreEspeces; i++)
{       /* 20 correspend a la taille max du nom d'espèce*/
        matNomsEspeces[i] = (char*)malloc((20)*sizeof(char));
}

if (intNbreSites != 0) /* Matrice de sites*/
        matSites=(char **)malloc((intNbreEspeces +1)*sizeof(char*));
matDistances = (double **)malloc((intNbreEspeces + 1)*sizeof(double*));
for (i = 0; i <= intNbreEspeces; i++)
{       matSites[i] = (char*)malloc((intNbreSites + 1)*sizeof(char));
        matDistances[i] = (double*)malloc((intNbreEspeces + 1)*sizeof(double));
}

p_RecupererBipartition(ARÉTÉ,matBiPartition,intNbreEspeces);

strcpy(infilename, p_input);
infile = fopen(infilename,"r");

fseek(infile, 0, 0);
get_path(infilename,outfilename);
strcat(outfilename, "out");
ibmpc = IBMCRT;
ansi = ANSICRT;
if (p_boot == 0)
{       bootstrap = 1;
        jackknife = 0;
        intBootstrap = 1;
}
else
{       bootstrap = 0;
        jackknife = 1;
        intBootstrap = 2;   }
```

```
doinput(1, toto);

reps = p_nbRep;

if (outfilename[0] == '\0')
        {       printf("Error output file.."); return 8; }
else
{
        if(numprocs>1){
                TtabComparaison = (int **) malloc((numprocs+1)*sizeof(int));
                for (r=0;r<=numprocs;r++){
                        TtabComparaison[r]=(int *)malloc(((2*intNbreEspeces-3) + 1)*sizeof(int));
                        for(tour = 1; tour <= 2*intNbreEspeces-3; tour++)
                                TtabComparaison[r][tour] = 0;
                }
        }
        /* Allocation de memoire du vecteur des pourcentages des arêtes de bases*/
        tabComparaison = (int *)malloc(((2*intNbreEspeces-3) + 1)*sizeof(int));
        for (r = 1; r <= 2*intNbreEspeces-3; r++)
                tabComparaison[r] = 0;

        /* Cas de l'execution sur un seul processeur */
        if(numprocs==1){
                for (r = 1; r <= reps; r++) {
                        tototа = r;
                        Pr=floor((double)(1000 * (r+1)) / (reps + 2));

                        bootwrite();

                        fseek(outfile, 0, 0);
                        fscanf(outfile,"%d",&intNbreEspeces);
                        fscanf(outfile,"%d",&intNbreSites);
                        fclose(outfile);

                        intRetour = seqToDistance(outfilename,&matDistances,1);

                        if (intRetour != 0)
                                return -1;
                        /* Appel de la fonction trex en mode uniprocesseur */
                        trex(p_matrix,"","",&ARÉTÉ,&LONGUEUR,0,1);

                        p_RecupererBipartition(ARÉTÉ, matBiPartitionTmp, intNbreEspeces);

                        p_ComparerBipartition(matBiPartition, matBiPartitionTmp, tabComparaison,
                                        intNbreEspeces);

                        if (r < reps)
                        {       if ((outfile = fopen(outfilename,"w+")) == 0)
                                {       printf("Error output file.."); return -1; }
                        }
                }
        }
        else{
                for (r = 1; r <= reps; r++) {
                        /* Repartition des tâches au niveau de chaque processeur */
                        if(r%numprocs==myid){
                                tototа = r;
                                Pr=floor((double)(1000 * (r+1)) / (reps + 2));
                                bootwrite();
                                fseek(outfile, 0, 0);
                                fscanf(outfile,"%d",&intNbreEspeces);
                                fscanf(outfile,"%d",&intNbreSites);
```

```
                        fclose(outfile);
                        intRetour = seqToDistance(outfilename,&matDistances,1);
                        if (intRetour != 0)
                                return -1;
                        /* Appel de la fonction trex en mode uniprocesseur */
                        trex(p_matrix,"","",&ARÉTÉ,&LONGUEUR,0,1);

                        p_RecupererBipartition(ARÉTÉ, matBiPartitionTmp,
                                                intNbreEspeces);

                        p_ComparerBipartition(matBiPartition, matBiPartitionTmp,
                                                TtabComparaison[myid], intNbreEspeces);

                        if (r < reps)
                        {       if ((outfile = fopen(outfilename,"w+")) == 0)
                                {       printf("Error output file.."); return -1; }
                        }
                }
        }

        /*Synchronisation du resultat*/
        if(myid>0){
                /* Envoie de matrice de comparaison */
                for (r = 1; r <= 2*intNbreEspeces-3; r++)
                        MPI_Send( &TtabComparaison[myid][r], 1, MPI_INT,0, 0,
                                MPI_COMM_WORLD );
                /* Reception de table de comparaison */
                for (r = 1; r <= 2*intNbreEspeces-3; r++)
                        MPI_Recv( &tabComparaison[r], 1, MPI_INT, myid-1, 0,
                                MPI_COMM_WORLD, &status );
                /* Envoie de la table de comparaison au processeur suivant */
                if(myid<numprocs-1){
                        for (r = 1; r <= 2*intNbreEspeces-3; r++)
                                MPI_Send( &tabComparaison[r], 1, MPI_INT,myid+1, 0,
                                        MPI_COMM_WORLD );
                }

        }
        else{
                /* Reception de la matrice de comparaison envoyee par chaque processeur */
                for(tour=1; tour<numprocs;tour++){
                        for (r = 1; r <= 2*intNbreEspeces-3; r++)
                                MPI_Recv( &TtabComparaison[tour][r], 1, MPI_INT, tour,
                                        0, MPI_COMM_WORLD, &status );
                }

                /* Reconstruction du table de comparaison */
                for (r = 1; r <= 2*intNbreEspeces-3; r++){
                        for(tour=0; tour<numprocs;tour++)
                                tabComparaison[r] = TtabComparaison[tour][r] +
                                                        tabComparaison[r];
                }
                /* Envoie du tableau de comparaison au processeur suivant */
                for (r = 1; r <= 2*intNbreEspeces-3; r++)
                        MPI_Send( &tabComparaison[r], 1, MPI_INT,myid+1, 0,
                                MPI_COMM_WORLD );
        }
    }

}
```

```
        /* Calcul des pourcentages des arêtes internes*/
        for (i=1;i<=2*intNbreEspeces-3;i++)
                tabComparaison[i] = tabComparaison[i] * 100 / reps;

        seqToDistance(p_input,&matDistances,1);


        trex(p_matrix,"","",&ARÉTÉ,&LONGUEUR,0,0);
        bs = fopen(p_outboot,"w+");
        for (i=1;i<=2*intNbreEspeces-3;i++){
                fprintf(bs,"%d %d %d\n",ARÉTÉ[2*i-1],ARÉTÉ[2*i-2],tabComparaison[i]);
        }
        fclose(bs);

        trex(p_matrix,p_output,p_stat,&ARÉTÉ,&LONGUEUR,1,0);

        return 0;
} /* END bootstrap_function */
```

PACKAGE PHYLIP

Le code source du package PHYLIP (version 3.6. (c) Copyright 1993-2004 by the University of Washington.Written by Joseph Felsenstein) a été téléchargé à partir du site de Joseph Felsenstein et a été inclus dans T-Rex avec sa permission. Ma participation porte principalement sur la parallélisation des méthodes suivantes : FITCH, DNAML, DNAMLK, PROML, PROMLK, DNAPARS, PROTPARS, PARS et DOLLOP. Des scripts de synchronisation ont été developpés pour la mise à jour des fichiers résultats produits par les différentes lames.

Méthode FITCH :

Nous présenterons seulement des parties du fichier fitchclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "dist.h"
#include "limits.h"

/*Inclusion de la librairie de MPI*/
#include "mpi.h"

/* version 3.6. (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Akiko Fuseki, Sean Lamont, and Andrew Keeffe.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

#define zsmoothings     10                      /* number of zero-branch correction iterations */
#define epsilonf        0.000001                /* a very small but not too small number  */
#define delta           0.0001                  /* a not quite so small number */
#define MAXNUMTREES   100000000         /* a number bigger than conceivable numtrees */

/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"

...

/*=============================================================================*
 * Version sequentielle de l'inference de l'arbre phylogenetique               *
 *                                                                             *
 *=============================================================================*/
void inference(){

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n\n",ith);

                        if (progress)
                                printf("\nData set # %ld:\n\n",ith);
                }
                fitch_getinput();
                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++){
                        // Lecture lors du premier tour seulement
                        if (jumb == 1) {
                                inputdata(replicates, printdata, lower, upper, x, reps);
                                setuptree(&curtree, nonodes2);
                                setuptree(&priortree, nonodes2);
                                setuptree(&bestree, nonodes2);
                                if (njumble > 1)
                                        setuptree(&bestree2, nonodes2);
                        }
                        maketree();
                }
                firstset = false;
                if (eoln(infile) && (ith < ensembles de donnees))
                        scan_eoln(infile);
        }
} /* inference */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'ensembles de donnees                *
 *                                                                             *
 * param:                                                                      *
 *       param   : Fichier des parametres de la fonction                       *
 *=============================================================================*/
void inferencePdata(char **argv, char *chemin){
        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        for (ith = 1; ith <= ensembles de donnees; ith++) {
                if((ith-1)%numprocs==myid){
                        blade_out_fileC  = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC = (char*)malloc(100*sizeof(char));
                        blade_out_file  = (char*)malloc(100*sizeof(char));
                        blade_tree_file = (char*)malloc(100*sizeof(char));

                        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                        // Nom du fichier outfile sur la lame
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);
```

```
                // Nom du fichier d'arbre sur la lame
                strcpy(blade_tree_file,chemin);
                strcat(blade_tree_file,OUTTREEBLADE);
                strcat(blade_tree_file, num_blade);
                strcat(blade_tree_file,"_");
                strcat(blade_tree_file,num_file);
                strcpy(blade_tree_fileC,blade_tree_file);
                strcat(blade_tree_fileC,FIL);

                openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n\n",ith);
                        if (progress)
                                printf("\nData set # %ld:\n\n",ith);
                }
                fitch_getinput();
                for (jumb = 1; jumb <= njumble; jumb++){
                        if (jumb == 1) {
                                inputdata(replicates, printdata, lower, upper, x, reps);
                                setuptree(&curtree, nonodes2);
                                setuptree(&priortree, nonodes2);
                                setuptree(&bestree, nonodes2);
                                if (njumble > 1)
                                        setuptree(&bestree2, nonodes2);
                        }
                        maketree();
                }

                firstset = false;
                if (eoln(infile) && (ith < ensembles de donnees))
                        scan_eoln(infile);

                free(blade_out_file);
                free(blade_tree_file);
                free(blade_out_fileC);
                free(blade_tree_fileC);
                FClose(outfile);
                FClose(outtree);
        }
        else{
                openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);
                fitch_getinput();
                inputdata(replicates, printdata, lower, upper, x, reps);
                firstset = false;
                if (eoln(infile) && (ith < ensembles de donnees))
                        scan_eoln(infile);
                FClose(outfile);
                FClose(outtree);
        }
    }
    // Attente des autres processeurs
    MPI_Barrier(MPI_COMM_WORLD);
} /* END inferencePdata */
```

```
/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres a inferer                    *
 *                                                                             *
 * param:                                                                      *
 *      param    : Fichier des parametres de la fonction                       *
 *=============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){
        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;
        int     max_procs;
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));

        /* Tableau de sauvegarde des meilleurs likelihood trouves par chaque processeur */
        TBestllhood=(double *) malloc((numprocs+1)*sizeof(double));

        sprintf(num_blade, "%d", myid);

        fitch_getinput();
        if(njumble > numprocs)
                max_procs =numprocs;
        else
                max_procs=njumble;

        if(myid < max_procs){
                inputdata(replicates, printdata, lower, upper, x, reps);
                setuptree(&curtree, nonodes2);
                setuptree(&priortree, nonodes2);
                setuptree(&bestree, nonodes2);
                if (njumble > 1)
                        setuptree(&bestree2, nonodes2);
        }
        FClose(outfile);
        if (trout)
                FClose(outtree);


        for (jumb = 1; jumb <= njumble; jumb++){
                if((jumb-1)%max_procs==myid){
                        sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                        blade_out_fileC  = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC = (char*)malloc(100*sizeof(char));
                        blade_out_file  = (char*)malloc(100*sizeof(char));
                        blade_tree_file = (char*)malloc(100*sizeof(char));

                        // Nom du fichier outfile sur la lame
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);
```

```
                        // Nom du fichier d'arbre sur la lame
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);

                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        maketree2();
                        tours++;
                        firstset = false;

                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        if(trout)
                                FClose(outtree);
                }
        }
        if(max_procs>1){
                if(myid>0 && myid<max_procs){
                        MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                        if(myid==leproc)
                                MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                }
                else{
                        if(myid==0){
                                TBestllhood[0]=bestlk;
                                leproc=myid;
                                for(i=1; i<max_procs;i++)
                                        MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                                MPI_COMM_WORLD, &status );
                                for(i=1; i<max_procs;i++){
                                        if( TBestllhood[i]> bestlk){
                                                leproc = i;
                                                bestlk =  TBestllhood[i];
                                        }
                                }
                                for(i=1; i<max_procs;i++)
                                        MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );
                                if(leproc!=myid)
                                        MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                                MPI_COMM_WORLD, &status );
                        }
                }
        }
} /* END inferencePtree */
```

```
/*=============================================================================*
 * FONCTION MAIN                                                               *
 *                                                                             *
 *=============================================================================*/
int main(int argc, Char *argv[])
{
        /* Variables necessaires a la parallélisation*/
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param;
        int i;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine = (char *)malloc(400*sizeof(char));

#ifdef MAC
        argc = 1;           /* macsetup("Fitch","");       */
        argv[0]="Fitch";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc,argv);
        progname = argv[0];                                     // Nom du programme
        paramName =  argv[1];
        param = fopen(paramName,"r");       // Fichier de paramètres

        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(outt,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(outt,OUTTREE);

        /* Ouverture du fihier d'entree et de sortie */
        openfile(&infile,inf,"input file","r",argv[0],infilename);
        openfile(&outfile,outf,"output file","w",argv[0],outfilename);


        ibmpc = IBMCRT;
        ansi = ANSICRT;
        mulsets = false;
        ensembles de donnees = 1;
        firstset = true;
        doinit(param);


        if (trout && myid==0)
                openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);
```

```
/* La valeur du seed doit etre modifier pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;

for (i=0;i<spp;++i){
        enterorder[i]=0;
}
/* Cas d'une lame*/
if( numprocs == 1 ){
        inference();
}

else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        if(ensembles de donnees==1 && njumble==1){
                inference();
        }
        }
        else{
                /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                progress = false;

                /* Cas d'un ensemble de donnees*/
                if(ensembles de donnees==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
                                sprintf(nb, " %d ", leproc); // nb est le processeur ayant le meilleur
                                                             // arbre
                                sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                                   // été trouve (numero de l'arbre)

                                // Formation ligne de commande pour appele le scripts de
                                // concatenation des fichiers
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);
                        }
                }

                /* Cas de plusieurs ensembles de donnees */
                if(ensembles de donnees > 1 ){
                        FClose(outfile);
                        if (trout)
                                FClose(outtree);

                        sprintf(nb, " %d ", (int)ensembles de donnees);
                        inferencePdata(argv, chemin);
                        openfile(&outfile,outf,"output file","a",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,outt,"output tree file","a",argv[0],outtreename);
                        if(myid==0){
                                sprintf(nbtours, " %d ", (int)ensembles de donnees/numprocs);

                                //Formation ligne de commande pour appele le scripts de
                                // concatenation des fichiers
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);
                        }
                }
        }
        if (trout)
                FClose(outtree);
        FClose(outfile);
        FClose(infile);
#ifdef MAC
        fixmacfile(outfilename);
        fixmacfile(outtreename);
#endif
        MPI_Finalize();
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif
        return 0;
}/* END main*/
```

Méthode de DNAML:

Nous présenterons seulementdes parties du fichier dnamlclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "seq.h"
#include "limits.h"

/*Inclusion de la librairie de MPI*/
#include "mpi.h"


/* version 3.6. (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Akiko Fuseki, Sean Lamont, Andrew Keeffe,
Dan Fineman, and Patrick Colacurcio.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

typedef struct valrec {
        double rat, ratxi, ratxv, orig_zz, z1, yl, z1zz, z1yy, xiz1, xiy1xv;
        double *ww, *zz, *wwzz, *vvzz;
} valrec;

typedef long vall[maxcategs];
```

```
typedef double contribarr[maxcatego];


/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================
 * Version sequentielle de l'inference de l'arbre phylogenetique              *
 *                                                                            *
 * param:                                                                     *
 *      param    : Fichier des parametres de la fonction                      *
 =============================================================================*/
void inference(FILE *param){

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n", ith);
                        printf("\nData set # %ld:\n", ith);
                }
                ttratio = ttratio0;
                getinput();

                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);
        }
} /* inference */


/*=============================================================================
 * Parallelisation en fonction du nombre d'ensembles de donnees               *
 *                                                                            *
 * param:                                                                     *
 *      param    : Fichier des parametres de la fonction                      *
 =============================================================================*/
void inferencePdata(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
```

```
/* Distribution des ensembles de donnees a traiter aux lames */
if((ith-1)%numprocs==myid){

        /* Creation des fichiers intermediaires */
        blade_out_fileC  = (char*)malloc(100*sizeof(char));
        blade_tree_fileC = (char*)malloc(100*sizeof(char));
        blade_out_file  = (char*)malloc(100*sizeof(char));
        blade_tree_file = (char*)malloc(100*sizeof(char));
        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

        /* Nom du fichier outfile sur la lame */
        strcpy(blade_out_file,chemin);
        strcat(blade_out_file,OUTFILEBLADE);
        strcat(blade_out_file, num_blade);
        strcat(blade_out_file,"_");
        strcat(blade_out_file,num_file);
        strcpy(blade_out_fileC,blade_out_file);
        strcat(blade_out_fileC,FIL);

        /* Nom du fichier d'arbre sur la lame */
        strcpy(blade_tree_file,chemin);
        strcat(blade_tree_file,OUTTREEBLADE);
        strcat(blade_tree_file, num_blade);
        strcat(blade_tree_file,"_");
        strcat(blade_tree_file,num_file);
        strcpy(blade_tree_fileC,blade_tree_file);
        strcat(blade_tree_fileC,FIL);
        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
        if (trout)
                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);


        if (ensembles de donnees > 1) {
                fprintf(outfile, "Data set # %ld:\n", ith);
        }
        ttratio = ttratio0;

        getinput();
        if (ith == 1)
                firstset = false;

        /* Boucle en fonction du nombre d'arbres */
        for (jumb = 1; jumb <= njumble; jumb++)
                maketree(param);

        /* Fermeture des fichiers intermediaires */
        free(blade_out_file);
        free(blade_tree_file);
        free(blade_out_fileC);
        free(blade_tree_fileC);
        FClose(outfile);
        FClose(outtree);
}
else{
        openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
        if (trout)
                openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

        getinput();
        if (ith == 1)
                firstset = false;
        FClose(outfile);
```

```
                FClose(outtree);
            }
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* inferencePdata */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres a inferer                    *
 *                                                                             *
 * param:                                                                      *
 *         param    : Fichier des parametres de la fonction                    *
 *============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;              // Indice des boucles utilisees
        int     max_procs;      // Nombre maximum de processeurs a utiliser


        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;
        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        TBestllhood     =(double *) malloc((numprocs+1)*sizeof(double));

        sprintf(num_blade, "%d", myid);
        ttratio = ttratio0;

        /* Le fichier outfile est deja ouvert */
        getinput();

        /* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile */
        FClose(outfile);
        if (trout && myid ==0)
                FClose(outtree);

        firstset = false;

        /* Determination du nombre de processeurs necessaires */
        if(njumble > numprocs)
                max_procs =numprocs;
        else
                max_procs=njumble;

        /* Boucle en fonction du nombre d'arbres */
        for (jumb = 1; jumb <= njumble; jumb++){

                /* Distribution des arbes a traiter aux lames*/
                if((jumb-1)%max_procs==myid){

                        /* Creation des fichiers intermediaires */
                        sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                        blade_out_fileC         = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                        blade_out_file          = (char*)malloc(100*sizeof(char));
                        blade_tree_file         = (char*)malloc(100*sizeof(char));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        /* Traitement de l'arbre */
                        maketree2(param);
                        tours++;

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        if(trout)
                                FClose(outtree);
                }
        }

        /* Synchronisation des resultats */
        if(max_procs>1){
                /* Cas des lames autres que le premier */
                if(myid>0 && myid<max_procs){
                        MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                        if(myid==leproc)
                                MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                }
                /* Cas de la premiere lame*/
                else{
                        if(myid==0){
                                TBestllhood[0]=bestlk;
                                leproc=myid;
                                for(i=1; i<max_procs;i++)
                                        MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                                MPI_COMM_WORLD, &status );

                                /* Recherche du meilleur resultat */
                                for(i=1; i<max_procs;i++){
                                        if( TBestllhood[i]> bestlk){
                                                leproc = i;
                                                bestlk =  TBestllhood[i];
                                        }
                                }
```

```
                        for(i=1; i<max_procs;i++)
                                MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                        if(leproc!=myid)
                                MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                        MPI_COMM_WORLD, &status );
                    }
                }
        }
} /* inferencePtree */

/*=============================================================================
 * FONCTION MAIN                                                              *
 *                                                                            *
 =============================================================================*/
int main(int argc, Char *argv[]){  /* DNA Maximum Likelihood */

        /* Variables necessaires a la parallélisation */
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *catf = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine = (char *)malloc(400*sizeof(char));

#ifdef MAC
        argc = 1;           /* macsetup("DnaML","");*/
        argv[0] = "DnaML";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc,argv);
        progname = argv[0];                                  // Nom du programme
        paramName = argv[1];                          // Nom du fichier de paramètres
        param = fopen(paramName,"r");       // Fichier de paramètres

        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(catf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(catf,CATFILE);strcat(weif,WEIGHTFILE);
        strcat(outt,OUTTREE);


        /* Ouverture du fihier d'entree et de sortie */
        openfile(&infile,inf,"input file","r",argv[0],infilename);
        openfile(&outfile,outf,"output file","w",argv[0],outfilename);

        /* Initialisation des variables */
        mulsets = false;
        ensembles de donnees = 1;
        firstset = true;
        ibmpc = IBMCRT;
        ansi = ANSICRT;
        grbg = NULL;
        doinit(param);
        ttratio0 = ttratio;

        /* Ouverture des autres fichiers necessaires */
        if (ctgry)
                openfile(&catfile,catf,"categories file","r",argv[0],catfilename);
        if (weights || justwts)
                openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
        if (trout && myid==0)
                openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);
        if (!usertree) nonodes2--;

        /* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
        if((myid%2)==0)
                seed[0]=seed[0]*(myid+1);
        else
                seed[0]=seed[0]*(myid+1)+1;


        /* Cas d'une lame*/
        if( numprocs == 1 ){
                inference(param);
        }
        else{
                bestlk = (double)LONG_MIN;
                sprintf(nbprocs, " %d ", numprocs);

                /* Cas d'un ensemble de donnees et  d'un arbre */
                if(ensembles de donnees==1 && njumble==1){
                        if(myid==0){
                                inference(param);
                        }
                }
                else{
                        /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                        progress  = false;
                        if (myid == 0)
                                startwtime = MPI_Wtime();
                        /* Cas d'un ensemble de donnees et  plusieurs arbes */
                        if(ensembles de donnees==1){
                                inferencePtree(param, argv, chemin);
                                if(myid==0){
                                        sprintf(nb, " %d ", leproc);   // nb est le processeur ayant le meilleur
                                                                       // arbre
                                        sprintf(nbtours, " %d ", bestour);// nbtours est le tour auquel
                                                                //l'arbre a été trouve (numero de l'arbre)

                                        /* Formation ligne de commande pour appele le scripts de
                                        concatenation des fichiers */
                                        sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                        strcat(comLine, nb);
                                        strcat(comLine, nbtours);
                                        strcat(comLine, outf );
```

```
                                strcat(comLine, " " );

                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);// Execution du script de la formation des fichiers
                                                // resultats
                        }
                }

                /* Cas de plusieurs ensembles de donnees */
                if(ensembles de donnees > 1 ){
                        /* Fermeture des fichiers de resultat */
                        FClose(outfile);
                        if (trout && myid==0)
                                FClose(outtree);
                        inferencePdata(param, argv, chemin);
                        /* Script sur le premier blade */
                        if(myid==0){
                                sprintf(nb, " %d ", (int)ensembles de donnees);
                                sprintf(nbtours, " %d ", (int)ensembles de donnees/numprocs);
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la
                                                  // formation des fichiers resultats
                        }
                }
        }

        /* Nettoyage */
        clean_up();
        MPI_Finalize();
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif
        return 0;
} /* DNA Maximum Likelihood */
```

Méthode de DNAMLK:

Nous présenterons seulementdes parties du fichier dnamlkclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "seq.h"
#include "limits.h"
```

```
/*Inclusion de la librairie de MPI*/
#include "mpi.h"

/* version 3.6. (c) Copyright 1986-2004 by the University of Washington
and by Joseph Felsenstein.  Written by Joseph Felsenstein.  Permission is
granted to copy and use this program provided no fee is charged for it
and provided that this copyright notice is not removed. */

#define epsilon       0.0001   /* used in makenewv, getthree, update */
#define over          60


/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================*
 * Version sequentielle de l'inference de l'arbre phylogenetique               *
 *                                                                             *
 * param:                                                                      *
 *       param    : Fichier des parametres de la fonction                      *
 *=============================================================================*/
void inference(FILE *param){

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
                ttratio = ttratio0;
                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n\n", ith);
                        if (progress)
                                printf("\nData set # %ld:\n", ith);
                }
                getinput();
                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);
        }
} /* inference */
```

```
/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres a inferer                    *
 *                                                                             *
 * param:                                                                      *
 *       param    : Fichier des parametres de la fonction                      *
 *=============================================================================*/
void  inferencePtree(FILE *param, char **argv, char *chemin){
        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
```

```
char	*num_blade;
char	*num_file;
num_blade		= (char *)malloc(5*sizeof(char));
num_file		= (char *)malloc(5*sizeof(char));
TBestllhood		= (double *) malloc((numprocs+1)*sizeof(double));


sprintf(num_blade, "%d", myid);
ttratio = ttratio0;

/* Le fichier outfile est deja ouvert */
getinput();

/* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile7 */
FClose(outfile);
if (trout)
	FClose(outtree);

firstset = false;

/* Determination du nombre de processeurs necessaires */
if(njumble > numprocs)
	max_procs =numprocs;
else
	max_procs=njumble;

/* Boucle en fonction du nombre d'arbres */
for (jumb = 1; jumb <= njumble; jumb++){
	if((jumb-1)%max_procs==myid){

		/* Creation des fichiers intermediaires */
		sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
		blade_out_fileC		= (char*)malloc(100*sizeof(char));
		blade_tree_fileC	= (char*)malloc(100*sizeof(char));
		blade_out_file		= (char*)malloc(100*sizeof(char));
		blade_tree_file		= (char*)malloc(100*sizeof(char));

		/* Nom du fichier outfile sur la lame */
		strcpy(blade_out_file,chemin);
		strcat(blade_out_file,OUTFILEBLADE);
		strcat(blade_out_file, num_blade);
		strcat(blade_out_file,"_");
		strcat(blade_out_file,num_file);
		strcpy(blade_out_fileC,blade_out_file);
		strcat(blade_out_fileC,FIL);

		/* Nom du fichier d'arbre sur la lame */
		strcpy(blade_tree_file,chemin);
		strcat(blade_tree_file,OUTTREEBLADE);
		strcat(blade_tree_file, num_blade);
		strcat(blade_tree_file,"_");
		strcat(blade_tree_file,num_file);
		strcpy(blade_tree_fileC,blade_tree_file);
		strcat(blade_tree_fileC,FIL);
		openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
		if (trout)
			openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

		maketree2(param);
		tours++;

		/* Fermeture des fichiers intermediaires */
		free(blade_out_file);
		free(blade_tree_file);
		free(blade_out_fileC);
		free(blade_tree_fileC);
		FClose(outfile);
		if(trout)
			FClose(outtree);
	}
}

/* Synchronisation des resultats */
if(max_procs>1){
	/* Cas des lames autres que le premier */
	if(myid>0 && myid<max_procs){
		MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
		MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
		if(myid==leproc)
			MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
	}
	/* Cas de la premiere lame*/
	else{
		if(myid==0){
			TBestllhood[0]=bestlk;
			leproc=myid;
			for(i=1; i<max_procs;i++)
				MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
					MPI_COMM_WORLD, &status );

			/* Recherche du meilleur resultat */
			for(i=1; i<max_procs;i++){
				if( TBestllhood[i]> bestlk){
					leproc = i;
					bestlk =  TBestllhood[i];
				}
			}
			for(i=1; i<max_procs;i++)
				MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

			if(leproc!=myid)
				MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
					MPI_COMM_WORLD, &status );
		}
	}
}
} /* inferencePtree */


/*=============================================================================
 * Parallelisation en fonction du nombre d'ensembles de donnees               *
 *                                                                            *
 * param:                                                                     *
 *	param	: Fichier des parametres de la fonction                       *
 =============================================================================*/
void	inferencePdata(FILE *param, char **argv, char *chemin){

	/* Variables necessaires a la formations des fichiers sur chaque lame*/
	char		*blade_out_file;
	char		*blade_tree_file;
	char		*blade_out_fileC;
	char		*blade_tree_fileC;
	char		*num_blade;
```

```
char            *num_file;
num_blade       = (char *)malloc(5*sizeof(char));
num_file        = (char *)malloc(5*sizeof(char));
sprintf(num_blade, "%d", myid);
/* Boucle en fonction du nombre d'ensembles de donnees */
for (ith = 1; ith <= ensembles de donnees; ith++) {

        /* Distribution des ensembles de donnees a traiter aux lames */
        if((ith-1)%numprocs==myid){
                /* Creation des fichiers intermediaires */
                blade_out_fileC = (char*)malloc(100*sizeof(char));
                blade_tree_fileC = (char*)malloc(100*sizeof(char));
                blade_out_file  = (char*)malloc(100*sizeof(char));
                blade_tree_file = (char*)malloc(100*sizeof(char));
                sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                /* Nom du fichier outfile sur la lame */
                strcpy(blade_out_file,chemin);
                strcat(blade_out_file,OUTFILEBLADE);
                strcat(blade_out_file, num_blade);
                strcat(blade_out_file,"_");
                strcat(blade_out_file,num_file);
                strcpy(blade_out_fileC,blade_out_file);
                strcat(blade_out_fileC,FIL);

                /* Nom du fichier d'arbre sur la lame */
                strcpy(blade_tree_file,chemin);
                strcat(blade_tree_file,OUTTREEBLADE);
                strcat(blade_tree_file, num_blade);
                strcat(blade_tree_file,"_");
                strcat(blade_tree_file,num_file);
                strcpy(blade_tree_fileC,blade_tree_file);
                strcat(blade_tree_fileC,FIL);
                openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n", ith);
                }
                ttratio = ttratio0;

                getinput();
                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);

                /* Fermeture des fichiers intermediaires */
                free(blade_out_file);
                free(blade_tree_file);
                free(blade_out_fileC);
                free(blade_tree_fileC);
                FClose(outfile);
                FClose(outtree);
        }
        else{
                openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);
                getinput();

                if (ith == 1)
                        firstset = false;
                FClose(outfile);
                FClose(outtree);
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* InferencePdata */


/*=============================================================================
 * FONCTION MAIN                                                               *
 *                                                                             *
 =============================================================================*/
int main(int argc, Char *argv[])
{ /* DNA Maximum Likelihood with molecular clock */

        /* Variables necessaires a la parallélisation */
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param;
        char *paramName;
        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *catf = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine  = (char *)malloc(400*sizeof(char));

#ifdef MAC
        argc = 1;           /* macsetup("Dnamlk", "Dnamlk");      */
        argv[0] = "Dnamlk";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc,argv);
        progname = argv[0];                 // Nom du programme
        paramName =  argv[1];               // Non du fichier de paramètres
        param = fopen(paramName,"r");       // Fichier de paramètres


        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(catf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(catf,CATFILE);strcat(weif,WEIGHTFILE);
        strcat(outt,OUTTREE);
```

```
/* Ouverture du fihier d'entree et de sortie */
openfile(&infile,inf,"input file","r",argv[0],infilename);
openfile(&outfile,outf,"output file","w",argv[0],outfilename);
/* Initialisation des variables */
ibmpc = IBMCRT;
ansi = ANSICRT;
ensembles de donnees = 1;
mulsets = false;
firstset = true;
doinit(param);
ttratio0   = ttratio;

/* Ouverture des autres fichiers necessaires */
if (ctgry)
        openfile(&catfile,catf,"categories file","r",argv[0],catfilename);
if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout)
        openfile(&outtree,out,"output tree file","w",argv[0],outtreename);

/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;

/* Cas d'une lame*/
if( numprocs == 1 ){
        inference(param);
}
else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(ensembles de donnees==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
        }
        else{
                /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                progress  = false;

                /* Cas d'un ensemble de donnees et  de plusieurs arbres*/
                if(ensembles de donnees==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
                                sprintf(nb, " %d ", leproc);   // nb est le processeur ayant le meilleur
                                                               // arbre
                                sprintf(nbtours, " %d ", bestour);// nbtours est le tour auquel l'arbre a
                                                                  // été trouve (numero de l'arbre)
                        /* Formation ligne de commande pour appele le scripts de concatenation des
                        fichiers */
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);
                        }
                }
                /* Cas de plusieurs ensembles de donnees */
                if(ensembles de donnees > 1 ){
                        /* Fermeture des fichiers de resultat */
                        FClose(outfile);
                        if (trout)
                                FClose(outtree);
                        inferencePdata(param, argv, chemin);
                        /* Script sur le premier blade */
                        if(myid==0){
                                sprintf(nb, " %d ", (int)ensembles de donnees);
                                sprintf(nbtours, " %d ", (int)ensembles de donnees/numprocs);

                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);
                        }
                }
        }

        /* Fermeture des fichiers */
        FClose(infile);
        FClose(catfile);
        FClose(weightfile);
#ifdef MAC
        fixmacfile(outfilename);
        fixmacfile(outtreename);
#endif
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif

        MPI_Finalize();
        return 0;
} /* DNA Maximum Likelihood with molecular clock */
```

Méthode de PROML:

Nous présenterons seulementdes parties du fichier promlclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "seq.h"
#include "limits.h"
```

```
/*Inclusion de la librairie de MPI*/
#include "mpi.h"


/* version 3.6. (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Lucas Mix, Akiko Fuseki, Sean Lamont,
Andrew Keeffe, Dan Fineman, and Patrick Colacurcio.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */


typedef long vall[maxcategs];
typedef double contribarr[maxcategs];

/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================
 * Version sequentielle de l'inference de l'arbre phylogenetique              *
 *                                                                            *
 * param:                                                                     *
 *         param    : Fichiers des parametres de la fonction                  *
 =============================================================================*/
void inference(FILE *param){
        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n", ith);
                        printf("\nData set # %ld:\n", ith);
                }
                getinput();

                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++){
                        max_num_sibs = 0;
                        maketree(param);
                }
        }
} /* inference */


/*=============================================================================
 * Parallelisation en fonction du nombre d'ensembles de donnees               *
 *                                                                            *
 * param:                                                                     *
 *         param    : Fichier des parametres de la fonction                   *
 =============================================================================*/
void inferencePdata(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {

                /* Distribution des ensembles de donnees a traiter aux lames */
                if((ith-1)%numprocs==myid){

                        /* Creation des fichiers intermediaires */
                        blade_out_fileC  = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC = (char*)malloc(100*sizeof(char));
                        blade_out_file   = (char*)malloc(100*sizeof(char));
                        blade_tree_file  = (char*)malloc(100*sizeof(char));
                        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);


                        if (ensembles de donnees > 1) {
                                fprintf(outfile, "Data set # %ld:\n", ith);
                        }

                        getinput();
                        if (ith == 1)
                                firstset = false;

                        /* Boucle en fonction du nombre d'arbres */
                        for (jumb = 1; jumb <= njumble; jumb++){
                                max_num_sibs = 0;
                                maketree(param);
                        }

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
```

```
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        FClose(outtree);
                }
                else{
                        openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

                        getinput();
                        if (ith == 1)
                                firstset = false;
                        FClose(outfile);
                        FClose(outtree);
                }
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* inferencePdata */


/*=============================================================================
 * Parallelisation en fonction du nombre d'arbres                             *
 *                                                                            *
 * param:                                                                     *
 *      param    : Fichier des parametres de la fonction                      *
 =============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;
        int     max_procs;
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;
        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        TBestllhood     = (double *) malloc((numprocs+1)*sizeof(double));


        sprintf(num_blade, "%d", myid);

        /* Le fichier outfile est deja ouvert */
        getinput();

        /* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile */
        FClose(outfile);
        if (trout)
                FClose(outtree);

        firstset = false;

        /* Determination du nombre de processeurs necessairess */
        if(njumble > numprocs)
                max_procs =numprocs;
        else
                max_procs=njumble;

        /* Boucle en fonction du nombre d'arbres */
        for (jumb = 1; jumb <= njumble; jumb++){
                max_num_sibs = 0;
                /* Distribution des arbes a traiter aux lames*/
                if((jumb-1)%max_procs==myid){

                        /* Creation des fichiers intermediaires */
                        sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                        blade_out_fileC         = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                        blade_out_file          = (char*)malloc(100*sizeof(char));
                        blade_tree_file         = (char*)malloc(100*sizeof(char));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        /* Traitement de l'arbre */
                        maketree2(param);
                        tours++;

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        if(trout)
                                FClose(outtree);
                }
        }

        /* Synchronisation des resultats */
        if(max_procs>1){
                /* Cas des lames autres que le premier */
                if(myid>0 && myid<max_procs){
                        MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                        if(myid==leproc)
                                MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                }
                /* Cas de la premiere lame*/
                else{
                        if(myid==0){
                                TBestllhood[0]=bestlk;
```

```
                                leproc=myid;
                                for(i=1; i<max_procs;i++)
                                        MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                                MPI_COMM_WORLD, &status );

                                /* Recherche du meilleur resultat */
                                for(i=1; i<max_procs;i++){
                                        if( TBestllhood[i]> bestlk){
                                                leproc = i;
                                                bestlk =  TBestllhood[i];
                                        }
                                }
                                for(i=1; i<max_procs;i++)
                                        MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                                if(leproc!=myid)
                                        MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                                MPI_COMM_WORLD, &status );
                        }
                }
        }
} /* inferencePtree */


/*-------------------------------------------------------------------------*
 * FONCTION MAIN                                                           *
 *                                                                         *
 *-------------------------------------------------------------------------*/
int main(int argc, Char *argv[])
{  /* Protein Maximum Likelihood */

        /* Variables necessaires a la parallélisation */
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param;
        char *paramName;
        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *catf = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine = (char *)malloc(400*sizeof(char));


#ifdef MAC
        argc = 1;           /* macsetup("ProML","");       */
        argv[0] = "ProML";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc,argv);
        progname = argv[0];
        paramName =  argv[1];                           // Nom du fichier de paramètres

param = fopen(paramName,"r");       // Fichier de paramètres

// Lecture du path et creation des noms des fichiers
getstryng1(chemin,param);

/* Creation du nom des différents fichiers avec le path */
strcpy(inf,chemin);strcpy(outf,chemin);strcpy(catf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(catf,CATFILE);strcat(weif,WEIGHTFILE);
strcat(outt,OUTTREE);


/* Ouverture du fihier d'entree et de sortie */
openfile(&infile,inf,"input file","r",argv[0],infilename);
openfile(&outfile,outf,"output file","w",argv[0],outfilename);

/* Initialisation des variables */
mulsets = false;
ensembles de donnees = 1;
firstset = true;
ibmpc = IBMCRT;
ansi = ANSICRT;
grbg = NULL;


doinit(param);

/* Ouverture des autres fichiers necessaires */
if (ctgry)
        openfile(&catfile,catf,"categories file","r",argv[0],catfilename);
if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout && myid==0)
        openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);

/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;

/* Cas d'une lame*/
if( numprocs == 1 ){
        inference(param);
}
else{
        bestlk = (double)LONG_MAX;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(ensembles de donnees==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
        }
        else{
                /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                progress = false;

                /* Cas d'un ensemble de donnees et  plusieurs arbes */
                if(ensembles de donnees==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
```

```
                                sprintf(nb, " %d ", leproc); // nb est le processeur ayant le meilleur
                                                        // arbre
                                sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                        // été trouve (numero de l'arbre)
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la
                                                // formation des fichiers resultats
                        }
                }

                /* Cas de plusieurs ensembles de donnees */
                if(ensembles de donnees > 1 ){

                        /* Fermeture des fichiers de resultat */
                        FClose(outfile);
                        if (trout)
                                FClose(outtree);
                        inferencePdata(param, argv, chemin);

                        /* Script sur le premier blade */
                        if(myid==0){
                                sprintf(nb, " %d ", (int)ensembles de donnees);
                                sprintf(nbtours, " %d ", (int)ensembles de donnees/numprocs);
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la
                                                // formation des fichiers resultats
                        }
                }
        }
        clean_up();
        MPI_Finalize();

#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif
        return 0;
} /* Protein Maximum Likelihood */
```

Méthode de PROMLK:

Nous présenterons seulementdes parties du fichier promlkclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "pbylip.h"
#include "seq.h"
#include "limits.h"


/*Inclusion de la librairie de MPI*/
#include "mpi.h"

/* version 3.6. (c) Copyright 1986-2004 by the University of Washington
and by Joseph Felsenstein.  Written by Joseph Felsenstein and Lucas Mix.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

#define epsilon      0.0001  /* used in makenewv, getthree, update */
#define over         60


/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================*
 * Version sequentielle de l'inference de l'arbre phylogenetique               *
 *                                                                             *
 * param:                                                                      *
 *         param    : Fichier des parametres de la fonction                    *
 *============================================================================*/
void inference(FILE *param){

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {
                if (ensembles de donnees > 1) {
                        fprintf(outfile, "Data set # %ld:\n\n", ith);
                        if (progress)
                                printf("\nData set # %ld:\n", ith);
                }
                getinput();
                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++){
                        max_num_sibs = 0;
                        maketree(param);
                }
        }
} /* inference */
```

```
/*=============================================================================*
 * Parallelisation en fonction du nombre d'ensembles de donnees                *
 *                                                                             *
 * param:                                                                      *
 *        param    : Fichier des parametres de la fonction                     *
 *============================================================================*/
void  inferencePtree(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int         i;
        int         max_procs;
        char        *blade_out_file;
        char        *blade_tree_file;
        char        *blade_out_fileC;
        char        *blade_tree_fileC;
        char        *num_blade;
        char        *num_file;
        num_blade           = (char *)malloc(5*sizeof(char));
        num_file            = (char *)malloc(5*sizeof(char));
        TBestllhood         =(double *) malloc((numprocs+1)*sizeof(double));


        sprintf(num_blade, "%d", myid);
        getinput();

        /* Fermeture des fichiers resultats */
        FClose(outfile);
        if (trout)
                FClose(outtree);

        firstset = false;

        /* Determination du nombre de processeurs necessairess */
        if(njumble > numprocs)
                max_procs =numprocs;
        else
                max_procs=njumble;

        /* Boucle en fonction du nombre d'arbres */
        for (jumb = 1; jumb <= njumble; jumb++){
                if((jumb-1)%max_procs==myid){

                        /* Creation des fichiers intermediaires */
                        sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                        blade_out_fileC             = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                        blade_out_file              = (char*)malloc(100*sizeof(char));
                        blade_tree_file             = (char*)malloc(100*sizeof(char));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        /* Traitement de l'arbre */
                        max_num_sibs = 0;
                        maketree2(param);
                        tours++;

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        if(trout)
                                FClose(outtree);
                }
        }

        /* Synchronisation des resultats */
        if(max_procs>1){
                /* Cas des lames autres que le premier */
                if(myid>0 && myid<max_procs){
                        MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                        if(myid==leproc)
                                MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                }
                /* Cas de la premiere lame*/
                else{
                        if(myid==0){
                                TBestllhood[0]=bestlk;
                                leproc=myid;
                                for(i=1; i<max_procs;i++)
                                        MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                                MPI_COMM_WORLD, &status );

                                /* Recherche du meilleur resultat */
                                for(i=1; i<max_procs;i++){
                                        if( TBestllhood[i]> bestlk){
                                                leproc = i;
                                                bestlk =  TBestllhood[i];
                                        }
                                }
                                for(i=1; i<max_procs;i++)
                                        MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                                if(leproc!=myid)
                                        MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                                MPI_COMM_WORLD, &status );
                        }
                }
        }
} /* inferencePtree */
```

```
/*=============================================================================*
 * Parallelisation en fonction du nombre d'ensembles de donnees                *
 *                                                                             *
 * param:                                                                      *
 *                                                                             *
 *       param    : Fichiers des parametres de la fonction                     *
 *============================================================================*/
void  inferencePdata(FILE *param, char **argv, char *chemin){


        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;
        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        /* Boucle en fonction du nombre d'ensembles de donnees */
        for (ith = 1; ith <= ensembles de donnees; ith++) {

                /* Distribution des ensembles de donnees a traiter aux lames */
                if((ith-1)%numprocs==myid){

                        /* Creation des fichiers intermediaires */
                        blade_out_fileC = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC = (char*)malloc(100*sizeof(char));
                        blade_out_file = (char*)malloc(100*sizeof(char));
                        blade_tree_file = (char*)malloc(100*sizeof(char));
                        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        if (ensembles de donnees > 1) {
                                fprintf(outfile, "Data set # %ld:\n", ith);
                        }

                        getinput();
                        if (ith == 1)
                                firstset = false;

                        /* Boucle en fonction du nombre d'arbres */
                        for (jumb = 1; jumb <= njumble; jumb++){
                                max_num_sibs = 0;
                                maketree(param);
                        }

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        FClose(outtree);
                }
                else{
                        openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

                        getinput();
                        if (ith == 1)
                                firstset = false;
                        FClose(outfile);
                        FClose(outtree);
                }
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* InferencePdata */


/*=============================================================================*
 * FONCTION MAIN                                                               *
 *                                                                             *
 *============================================================================*/
int main(int argc, Char *argv[])
{  /* Protein Maximum Likelihood with molecular clock */

        /* Variables necessaires a la parallélisation*/
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *catf = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine = (char *)malloc(400*sizeof(char));

#ifdef MAC
        argc = 1;       /* macsetup("Promlk", "");      */
        argv[0] = "Promlk";
#endif
        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
MPI_Get_processor_name(processor_name,&namelen);

/* Identification des paramètres de la ligne de commande */
init(argc,argv);
progname = argv[0];                                     // Nom du programme
paramName =  argv[1];                           // Non du fichier de paramètres
param = fopen(paramName,"r");       // Fichier de paramètres

// Lecture du path et creation des noms des fichiers
getstryng1(chemin,param);

/* Creation du nom des differents fichiers avec le path */
strcpy(inf,chemin);strcpy(outf,chemin);strcpy(catf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(catf,CATFILE);strcat(weif,WEIGHTFILE);
strcat(outt,OUTTREE);

/* Ouverture du fihier d'entree et de sortie */
openfile(&infile,inf,"input file","r",argv[0],infilename);
openfile(&outfile,outf,"output file","w",argv[0],outfilename);

ibmpc = IBMCRT;
ansi = ANSICRT;
ensembles de donnees = 1;
mulsets = false;
firstset = true;
doinit(param);

/* Ouverture des autres fichiers necessaires */
if (ctgry)
        openfile(&catfile,catf,"categories file","r",argv[0],catfilename);
if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout)
        openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);


/* Modification de la valeur du seed (au nivcau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;


/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;

/* Cas d'une lame*/
if( numprocs == 1 ){
        inference(param);
}
else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(ensembles de donnees==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
```

```
}
else{
        /* Pas d'ecriture par les programmes en parallele car c'est un melange */
        progress  = false;

        /* Cas d'un ensemble de donnees*/
        if(ensembles de donnees==1){
                inferencePtree(param, argv, chemin);

                if(myid==0){
                        sprintf(nb, " %d ", leproc); // nb est le processeur ayant le meilleur
                                                     // arbre
                        sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                           // été trouve (numero de l'arbre)

                        /* Formation ligne de commande pour appele le scripts de
                        concatenation des fichiers */
                        sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                        strcat(comLine, nb);
                        strcat(comLine, nbtours);
                        strcat(comLine, outf );
                        strcat(comLine, " " );
                        if (trout)
                                strcat(comLine, outt );
                        printf("\n Script : %s", comLine);
                        system(comLine);
                }
        }

        /* Cas de plusieurs ensembles de donnees */
        if(ensembles de donnees > 1 ){

                /* Fermeture des fichiers de resultat */
                FClose(outfile);
                if (trout)
                        FClose(outtree);

                inferencePdata(param, argv, chemin);

                /* Reouverture des fichiers de resultat */
                openfile(&outfile,outf,"output file","a",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,outt,"output tree file","a",argv[0],outtreename);

                /* Script sur le premier blade */
                if(myid==0){
                        sprintf(nb, " %d ", (int)ensembles de donnees);
                        sprintf(nbtours, " %d ", (int)ensembles de donnees/numprocs);

                        /* Formation ligne de commande pour appele le scripts de
                        concatenation des fichiers */
                        sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                        strcat(comLine, nbprocs);
                        strcat(comLine, nb);
                        strcat(comLine, nbtours);
                        strcat(comLine, outf );
                        strcat(comLine, " " );
                        if (trout)
                                strcat(comLine, outt );
                        printf("\n Script : %s", comLine);
                        system(comLine);
                }
```

```
                }
            }
        }
        clean_up();

#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif
        MPI_Finalize();
        return 0;
} /* Protein Maximum Likelihood with molecular clock */
```

Méthode de DNAPARS:

Nous présenterons seulementdes parties du fichier dnaparsclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "seq.h"
#include "limits.h"

/*Inclusion de la librairie de MPI*/
#include "mpi.h"


/* version 3.6 (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Akiko Fuseki, Sean Lamont, and Andrew Keeffe.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

#define MAXNUMTREES     1000000 /* bigger than number of user trees can be */

/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================*
 * Version sequentielle de l'inference de l'arbre phylogenetique               *
 *                                                                             *
 * param:                                                                      *
 *         param    : Fichier des parametres de la fonction                    *
 *============================================================================*/
void inference(FILE *param){

        /* Boucle en fonction du nombre de msets */
        for (ith = 1; ith <= msets; ith++) {
                if (!(justwts && !firstset))
                        allocrest();
                if (msets > 1 && !justwts) {
                        fprintf(outfile, "\nData set # %ld:\n\n", ith);
                        if (progress)
                                printf("\nData set # %ld:\n\n", ith);
                }
                doinput();
                if (ith == 1)
                        firstset = false;
                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);
                if (!justwts)
                        freerest();
        }
} /* inference */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'ensembles de donnees                *
 *                                                                             *
 * param:                                                                      *
 *         param    : Fichier des parametres de la fonction                    *
 *============================================================================*/
void inferencePdata(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        /* Boucle en fonction du nombre de msets */
        for (ith = 1; ith <= msets; ith++) {

                /* Distribution des msets a traiter aux lames */
                if((ith-1)%numprocs==myid){

                        /* Creation des fichiers intermediaires */
                        blade_out_fileC  = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC =(char*)malloc(100*sizeof(char));
                        blade_out_file  = (char*)malloc(100*sizeof(char));
                        blade_tree_file = (char*)malloc(100*sizeof(char));
                        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
```

```
                strcat(blade_tree_file,num_file);
                strcpy(blade_tree_fileC,blade_tree_file);
                strcat(blade_tree_fileC,FIL);
                openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);

                if (trout)
                        openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                if (!(justwts && !firstset))
                        allocrest();
                if (msets > 1 && !justwts) {
                        fprintf(outfile, "\nData set # %ld:\n\n", ith);
                }

                doinput();

                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);
                if (!justwts)
                        freerest();

                /* Fermeture des fichiers intermediaires */
                free(blade_out_file);
                free(blade_tree_file);
                free(blade_out_fileC);
                free(blade_tree_fileC);
                FClose(outfile);
                FClose(outtree);
        }
        else{

                if (!(justwts && !firstset))
                        allocrest();

                openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

                doinput();
                if (ith == 1)
                        firstset = false;
                FClose(outfile);
                FClose(outtree);
        }
    }
    /* Attente des autres processeurs */
    MPI_Barrier(MPI_COMM_WORLD);
} /* inferencePdata */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres                              *
 *                                                                             *
 * param:                                                                      *
 *      param    : Fichier des parametres de la fonction                       *
 *============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){
```

```
/* Variables necessaires a la formations des fichiers sur chaque lame*/
int     i;
int     max_procs;
char    *blade_out_file;
char    *blade_tree_file;
char    *blade_out_fileC;
char    *blade_tree_fileC;
char    *num_blade;
char    *num_file;
num_blade       = (char *)malloc(5*sizeof(char));
num_file        = (char *)malloc(5*sizeof(char));
TBestllhood     =(double *) malloc((numprocs+1)*sizeof(double));


sprintf(num_blade, "%d", myid);

if (!(justwts && !firstset))
        allocrest();

/* Le fichier outfile est deja ouvert */
doinput();

/* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile */
FClose(outfile);
if (trout && myid ==0)
        FClose(outtree);

firstset = false;

/* Determination du nombre de processeurs necessaires */
if(njumble > numprocs)
        max_procs =numprocs;
else
        max_procs=njumble;

/* Boucle en fonction du nombre d'arbres */
for (jumb = 1; jumb <= njumble; jumb++) {


        /* Distribution des arbes a traiter aux lames*/
        if(jumb-1)%max_procs==myid){
                /* Creation des fichiers intermediaires */
                sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                blade_out_fileC         = (char*)malloc(100*sizeof(char));
                blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                blade_out_file          = (char*)malloc(100*sizeof(char));
                blade_tree_file         = (char*)malloc(100*sizeof(char));

                /* Nom du fichier outfile sur la lame */
                strcpy(blade_out_file,chemin);
                strcat(blade_out_file,OUTFILEBLADE);
                strcat(blade_out_file, num_blade);
                strcat(blade_out_file,"_");
                strcat(blade_out_file,num_file);
                strcpy(blade_out_fileC,blade_out_file);
                strcat(blade_out_fileC,FIL);

                /* Nom du fichier d'arbre sur la lame */
                strcpy(blade_tree_file,chemin);
                strcat(blade_tree_file,OUTTREEBLADE);
                strcat(blade_tree_file, num_blade);
                strcat(blade_tree_file,"_");
```

```
                    strcat(blade_tree_file,num_file);
                    strcpy(blade_tree_fileC,blade_tree_file);
                    strcat(blade_tree_fileC,FIL);
                    openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                    if (trout)
                            openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                    /* Traitement de l'arbre */
                    maketree2(param);
                    tours++;

                    /* Fermeture des fichiers intermediaires */
                    free(blade_out_file);
                    free(blade_tree_file);
                    free(blade_out_fileC);
                    free(blade_tree_fileC);
                    FClose(outfile);
                    if(trout)
                            FClose(outtree);
            }
    }

    /* Synchronisation des resultats */
    if (!justwts)
            freerest();
    if(max_procs>1){
            /* Cas des lames autres que le premier */
            if(myid>0 && myid<max_procs){
                    MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                    MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                    if(myid==leproc)
                            MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
            }
            /* Cas de la premiere lame*/
            else{
                    if(myid==0){
                            TBestllhood[0]=bestlk;
                            leproc=myid;
                            for(i=1; i<max_procs;i++)
                                    MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                            MPI_COMM_WORLD, &status );

                            /* Recherche du meilleur resultat */
                            for(i=1; i<max_procs;i++){
                                    if( TBestllhood[i]> bestlk){
                                            leproc = i;
                                            bestlk =  TBestllhood[i];
                                    }
                            }
                            for(i=1; i<max_procs;i++)
                                    MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                            if(leproc!=myid)
                                    MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                            MPI_COMM_WORLD, &status );
                    }
            }
    }
} /* inferencePtree */
```

```
/*=============================================================================*
 * FONCTION MAIN                                                               *
 *                                                                             *
 *=============================================================================*/

int main(int argc, Char *argv[])
{ /* DNA parsimony by uphill search */

        /* Variables necessaires a la parallélisation*/
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param=NULL;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine  = (char *)malloc(400*sizeof(char));

        /* reads in spp, chars, and the data. Then calls maketree to
        construct the tree */
#ifdef MAC
        argc = 1;       /* macsetup("Dnapars","");      */
        argv[0] = "Dnapars";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc, argv);
        progname = argv[0];                                     // Nom du programme
        paramName =  argv[1];
        param = fopen(paramName,"r");       // Fichier de paramètres */


        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(weif,WEIGHTFILE);strcat(outt,OUTTREE);

        /* Ouverture du fihier d'entree et de sortie */
        openfile(&infile,inf,"input file","r",argv[0],infilename);
        openfile(&outfile,outf,"output file","w",argv[0],outfilename);

        ibmpc = IBMCRT;
        ansi = ANSICRT;
        msets = 1;
        firstset = true;
        garbage = NULL;
        grbg = NULL;
        doinit(param);
```

```
if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout && myid ==0)
        openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);

/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;


/* Cas d'une lame*/
if( numprocs == 1){
        inference(param);
}
else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(msets==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
        }
        else{
                /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                progress  = false;

                /* Cas d'un ensemble de donnees et  plusieurs arbes */
                if(msets==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
                                sprintf(nb, " %d ", leproc);   // nb est le processeur ayant le meilleur
                                                               // arbre
                                sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                                   // été trouve (numero de l'arbre)
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la formation des fichiers
                                                  // resultats
                        }
                }

                /* Cas de plusieurs msets */
                if(msets > 1 ){
                        /* Fermeture des fichiers de resultat */
                        FClose(outfile);
                        if (trout && myid==0)
                                FClose(outtree);

                        inferencePdata(param, argv, chemin);
                        /* Script sur le premier blade */
                        if(myid==0){
                                sprintf(nb, " %d ", (int)msets);
                                sprintf(nbtours, " %d ", (int)msets/numprocs);
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la formation des fichiers
                                                  // resultats
                        }
                }
        }
        freetree(nonodes, treenode);
        FClose(infile);
        FClose(outfile);
        if (weights || justwts)
                FClose(weightfile);
        if (trout)
                FClose(outtree);
        if (usertree)
                FClose(intree);
#ifdef MAC
        fixmacfile(outfilename);
        fixmacfile(outtreename);
#endif
        if (progress)
                printf("Done.\n\n");
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif

        MPI_Finalize();
        return 0;
} /* DNA parsimony by uphill search */
```

Méthode de PARS:

Nous présenterons seulementdes parties du fichier parsclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "discrété.h"
#include "limits.h"

/*Inclusion de la librairie de MPI*/
#include "mpi.h"
```

```
/* version 3.6 (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Akiko Fuseki, Sean Lamont, and Andrew Keeffe.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

#define MAXNUMTREES     1000000  /* bigger than number of user trees can be */

/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...


/*=============================================================================
 * Version sequentielle de l'inference de l'arbre phylogenetique              *
 *                                                                            *
 * param:                                                                     *
 *         param    : Fichier des parametres de la fonction                   *
 =============================================================================*/
void inference(FILE *param){
        /* Boucle en fonction du nombre de msets */
        for (ith = 1; ith <= msets; ith++) {
                if (msets > 1 && !justwts) {
                        fprintf(outfile, "\nData set # %ld:\n\n", ith);
                        if (progress)
                                printf("\nData set # %ld:\n\n", ith);
                }
                doinput();
                if (ith == 1)
                        firstset = false;
                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);
                freerest();
        }
} /* inference */


/*=============================================================================
 * Parallelisation en fonction du nombre d'ensembles de donnees               *
 *                                                                            *
 * param:                                                                     *
 *         param    : Fichier des parametres de la fonction                   *
 =============================================================================*/
void inferencePdata(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);
```

```
/* Boucle en fonction du nombre de msets */
for (ith = 1; ith <= msets; ith++) {

        /* Distribution des msets a traiter aux lames */
        if((ith-1)%numprocs==myid){

                /* Creation des fichiers intermediaires */
                blade_out_fileC  = (char*)malloc(100*sizeof(char));
                blade_tree_fileC = (char*)malloc(100*sizeof(char));
                blade_out_file   = (char*)malloc(100*sizeof(char));
                blade_tree_file  = (char*)malloc(100*sizeof(char));
                sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                /* Nom du fichier outfile sur la lame */
                strcpy(blade_out_file,chemin);
                strcat(blade_out_file,OUTFILEBLADE);
                strcat(blade_out_file, num_blade);
                strcat(blade_out_file,"_");
                strcat(blade_out_file,num_file);
                strcpy(blade_out_fileC,blade_out_file);
                strcat(blade_out_fileC,FIL);

                /* Nom du fichier d'arbre sur la lame */
                strcpy(blade_tree_file,chemin);
                strcat(blade_tree_file,OUTTREEBLADE);
                strcat(blade_tree_file, num_blade);
                strcat(blade_tree_file,"_");
                strcat(blade_tree_file,num_file);
                strcpy(blade_tree_fileC,blade_tree_file);
                strcat(blade_tree_fileC,FIL);
                openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);


                if (msets > 1) {
                        fprintf(outfile, "Data set # %ld:\n", ith);
                }
                doinput();
                if (ith == 1)
                        firstset = false;

                /* Boucle en fonction du nombre d'arbres */
                for (jumb = 1; jumb <= njumble; jumb++)
                        maketree(param);

                freerest();

                /* Fermeture des fichiers intermediaires */
                free(blade_out_file);
                free(blade_tree_file);
                free(blade_out_fileC);
                free(blade_tree_fileC);
                FClose(outfile);
                FClose(outtree);
        }
        else{
                openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

                doinput();
```

```
                        if (ith == 1)
                                firstset = false;
                        FClose(outfile);
                        FClose(outtree);
                }
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* inferencePdata */

/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres                              *
 *                                                                             *
 * param:                                                                      *
 *       param    : Fichier des parametres de la fonction                      *
 *============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;
        int     max_procs;
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;
        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        TBestllhood     =(double *) malloc((numprocs+1)*sizeof(double));


        sprintf(num_blade, "%d", myid);

        /* Le fichier outfile est deja ouvert */
        doinput();

        /* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile */
        FClose(outfile);
        if (trout && myid ==0)
                FClose(outtree);

        firstset = false;

        /* Determination du nombre de processeurs necessaires */
        if(njumble > numprocs)
                max_procs =numprocs;
        else
                max_procs=njumble;

        /* Boucle en fonction du nombre d'arbres */
        for (jumb = 1; jumb <= njumble; jumb++){


                /* Distribution des arbes a traiter aux lames*/
                if((jumb-1)%max_procs==myid){
                        /* Creation des fichiers intermediaires */
                        sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                        blade_out_fileC         = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                        blade_out_file          = (char*)malloc(100*sizeof(char));
                        blade_tree_file         = (char*)malloc(100*sizeof(char));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        /* Traitement de l'arbre */
                        maketree2(param);
                        tours++;

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        if(trout)
                                FClose(outtree);
                }
        }

        /* Synchronisation des resultats */
        freerest();

        if(max_procs>1){
                /* Cas des lames autres que le premier */
                if(myid>0 && myid<max_procs){
                        MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                        MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                        if(myid==leproc)
                                MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
                }
                /* Cas de la premiere lame*/
                else{
                        if(myid==0){
                                TBestllhood[0]=bestlk;
                                leproc=myid;
                                for(i=1; i<max_procs;i++)
                                        MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                                MPI_COMM_WORLD, &status );

                                /* Recherche du meilleur resultat */
                                for(i=1; i<max_procs;i++){
                                        if( TBestllhood[i]> bestlk){
                                                leproc = i;
```

```
                                        bestlk =  TBestllhood[i];
                                }
                        }
                        for(i=1; i<max_procs;i++)
    {                           MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                        if(leproc!=myid)
                                MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                        MPI_COMM_WORLD, &status );
                }
        }
    }
} /* inferencePtree */


/*=============================================================================
 * FONCTION MAIN                                                              *
 *                                                                            *
 =============================================================================*/
int main(int argc, Char *argv[])
{  /* Discrété character parsimony by uphill search */

        /* Variables necessaires a la parallélisation*/
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param=NULL;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine = (char *)malloc(400*sizeof(char));

        /* reads in spp, chars, and the data. Then calls maketree to
        construct the tree */
#ifdef MAC
        argc = 1;               /* macsetup("Pars","");            */
        argv[0]="Pars";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des paramètres de la ligne de commande */
        init(argc,argv);
        progname = argv[0];                                     // Nom du programme
        paramName =  argv[1];
        param = fopen(paramName,"r");       // Fichier de paramétres */

        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(weif,WEIGHTFILE);strcat(outt,OUTTREE);
```

```
/* Ouverture du fihier d'entree et de sortie */
openfile(&infile,inf,"input file","r",argv[0],infilename);
openfile(&outfile,outf,"output file","w",argv[0],outfilename);

ibmpc = IBMCRT;
ansi = ANSICRT;
msets = 1;
firstset = true;
garbage = NULL;
grbg = NULL;
doinit(param);

if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout && myid ==0)
        openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);


/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;


/* Cas d'une lame*/
if( numprocs == 1 ){
        inference(param);
}
else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(msets==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
        }
        else{
                /* Pas d'ecriture par les programmes en parallele car c'est un melange */
                progress  = false;

                /* Cas d'un ensemble de donnees et  plusieurs arbes */
                if(msets==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
                                sprintf(nb, " %d ", leproc);    // nb est le processeur ayant le meilleur
                                                                // arbre
                                sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                                   // été trouve (numero de l'arbre)
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
```

```
                        system(comLine);  // Execution du script de la formation des fichiers
                                          // resultats
                    }
                }

                /* Cas de plusieurs msets */
                if(msets > 1 ){
                    /* Fermeture des fichiers de resultat */
                    FClose(outfile);
                    if (trout && myid==0)
                        FClose(outtree);
                    inferencePdata(param, argv, chemin);
                    /* Script sur le premier blade */
                    if(myid==0){
                        sprintf(nb, " %d ", (int)msets);
                        sprintf(nbtours, " %d ", (int)msets/numprocs);
                        /* Formation ligne de commande pour appele le scripts de
                        concatenation des fichiers */
                        sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                        strcat(comLine, nbprocs);
                        strcat(comLine, nb);
                        strcat(comLine, nbtours);
                        strcat(comLine, outf );
                        strcat(comLine, " " );
                        if (trout)
                            strcat(comLine, outt );
                        printf("\n Script : %s", comLine);
                        system(comLine);  // Execution du script de la formation des fichiers
                                          // resultats
                    }
                }
            }
        }

        FClose(infile);
        FClose(outfile);

        if (weights || justwts)
            FClose(weightfile);
        if (trout)
            FClose(outtree);
        if (usertree)
            FClose(intree);

#ifdef MAC
        fixmacfile(outfilename);
        fixmacfile(outtreename);
#endif
        if (progress)
            printf("Done.\n\n");
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif

        MPI_Finalize();
        return 0;
} /* Discrété character parsimony by uphill search */
```

Méthode de DOLLOP:

Nous présenterons seulementdes parties du fichier dollopclean.c qui contient la méthode parallélisée.

```
/* Inclusion des librairies necessaires */
#include "phylip.h"
#include "disc.h"
#include "dollo.h"
#include "limits.h"

/*Inclusion de la librairie de MPI*/
#include "mpi.h"


/* version 3.6. (c) Copyright 1993-2004 by the University of Washington.
Written by Joseph Felsenstein, Akiko Fuseki, Sean Lamont, and Andrew Keeffe.
Permission is granted to copy and use this program provided no fee is
charged for it and provided that this copyright notice is not removed. */

#define maxtrees        100  /* maximum number of tied trees stored    */

/* Definition des constantes utilisees pour le nom des fichiers */
#define OUTFILEBLADE "outfile"
#define DATA " data"
#define FIL " file"
#define OUTTREEBLADE "outtree"


...

/*=============================================================================
 * Version sequentielle de l'inference de l'arbre phylogenetique              *
 *                                                                            *
 * param:                                                                     *
 *         param    : Fichier des parametres de la fonction                   *
 =============================================================================*/
void inference(FILE *param){

        /* Boucle en fonction du nombre de msets */
        for (ith = 1; ith <= msets; ith++) {
            if (msets > 1 && !justwts) {
                fprintf(outfile, "Data set # %ld:\n\n",ith);
                if (progress)
                    printf("\nData set # %ld:\n",ith);
            }
            if (justwts){
                fprintf(outfile, "Weights set # %ld:\n\n", ith);
                if (progress)
                    printf("\nWeights set # %ld:\n\n", ith);
            }
            if (printdata && !justwts)
                fprintf(outfile, "%2ld species, %3ld  characters\n\n", spp, chars);
            doinput();
            if (ith == 1)
                firstset = false;
            /* Boucle en fonction du nombre d'arbres */
            for (jumb = 1; jumb <= njumble; jumb++)
```

```
                        maketree(param);
        }
} /* inference */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'ensembles de donnees                *
 *                                                                             *
 * param:                                                                      *
 *      param    : Fichier des parametres de la fonction                       *
 *============================================================================*/
void inferencePdata(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;

        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
        sprintf(num_blade, "%d", myid);

        /* Boucle en fonction du nombre de msets */
        for (ith = 1; ith <= msets; ith++) {

                /* Distribution des msets a traiter aux lames */
                if((ith-1)%numprocs==myid){

                        /* Creation des fichiers intermediaires */
                        blade_out_fileC  = (char*)malloc(100*sizeof(char));
                        blade_tree_fileC = (char*)malloc(100*sizeof(char));
                        blade_out_file  = (char*)malloc(100*sizeof(char));
                        blade_tree_file = (char*)malloc(100*sizeof(char));
                        sprintf(num_file,"%d", (int)((ith-1)/numprocs));

                        /* Nom du fichier outfile sur la lame */
                        strcpy(blade_out_file,chemin);
                        strcat(blade_out_file,OUTFILEBLADE);
                        strcat(blade_out_file, num_blade);
                        strcat(blade_out_file,"_");
                        strcat(blade_out_file,num_file);
                        strcpy(blade_out_fileC,blade_out_file);
                        strcat(blade_out_fileC,FIL);

                        /* Nom du fichier d'arbre sur la lame */
                        strcpy(blade_tree_file,chemin);
                        strcat(blade_tree_file,OUTTREEBLADE);
                        strcat(blade_tree_file, num_blade);
                        strcat(blade_tree_file,"_");
                        strcat(blade_tree_file,num_file);
                        strcpy(blade_tree_fileC,blade_tree_file);
                        strcat(blade_tree_fileC,FIL);
                        openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                        if (msets > 1 && !justwts) {
                                fprintf(outfile, "Data set # %ld:\n\n",ith);
                                if (progress)
                                        printf("\nData set # %ld:\n",ith);
                        }
                        if (justwts){
                                fprintf(outfile, "Weights set # %ld:\n\n", ith);
                                if (progress)
                                        printf("\nWeights set # %ld:\n\n", ith);
                        }
                        if (printdata && !justwts)
                                fprintf(outfile, "%2ld species, %3ld  characters\n\n", spp, chars);

                        doinput();
                        if (ith == 1)
                                firstset = false;

                        /* Boucle en fonction du nombre d'arbres */
                        for (jumb = 1; jumb <= njumble; jumb++)
                                maketree(param);

                        /* Fermeture des fichiers intermediaires */
                        free(blade_out_file);
                        free(blade_tree_file);
                        free(blade_out_fileC);
                        free(blade_tree_fileC);
                        FClose(outfile);
                        FClose(outtree);
                }
                else{
                        openfile(&outfile,"bidonoutfile","bidon","a",argv[0],outfilename);
                        if (trout)
                                openfile(&outtree,"bidontreefile","bidon","a",argv[0],outtreename);

                        doinput();
                        if (ith == 1)
                                firstset = false;
                        FClose(outfile);
                        FClose(outtree);
                }
        }
        /* Attente des autres processeurs */
        MPI_Barrier(MPI_COMM_WORLD);
} /* inferencePdata */


/*=============================================================================*
 * Parallelisation en fonction du nombre d'arbres                              *
 *                                                                             *
 * param:                                                                      *
 *      param    : Fichier des parametres de la fonction                       *
 *============================================================================*/
void inferencePtree(FILE *param, char **argv, char *chemin){

        /* Variables necessaires a la formations des fichiers sur chaque lame*/
        int     i;
        int     max_procs;
        char    *blade_out_file;
        char    *blade_tree_file;
        char    *blade_out_fileC;
        char    *blade_tree_fileC;
        char    *num_blade;
        char    *num_file;
        num_blade       = (char *)malloc(5*sizeof(char));
        num_file        = (char *)malloc(5*sizeof(char));
```

```
TBestllhood        =(double *) malloc((numprocs+1)*sizeof(double));

sprintf(num_blade, "%d", myid);

if (msets > 1 && !justwts) {
        fprintf(outfile, "Data set # %ld:\n\n",ith);
        if (progress)
                printf("\nData set # %ld:\n",ith);
}
if (justwts){
        fprintf(outfile, "Weights set # %ld:\n\n", ith);
        if (progress)
                printf("\nWeights set # %ld:\n\n", ith);
}
if (printdata && !justwts)
        fprintf(outfile, "%2ld species, %3ld  characters\n\n", spp, chars);

/* Le fichier outfile est deja ouvert */
doinput();

/* Fermeture des fichiers resultats a cette place car getinput utilise le fichier outfile */
FClose(outfile);
if (trout && myid ==0)
        FClose(outtree);

firstset = false;

/* Determination du nombre de processeurs necessaires */
if(njumble > numprocs)
        max_procs =numprocs;
else
        max_procs=njumble;

/* Boucle en fonction du nombre d'arbres */
for (jumb = 1; jumb <= njumble; jumb++){


        /* Distribution des arbes a traiter aux lames*/
        if((jumb-1)%max_procs==myid){
                /* Creation des fichiers intermediaires */
                sprintf(num_file,"%d", (int)((jumb-1)/max_procs));
                blade_out_fileC         = (char*)malloc(100*sizeof(char));
                blade_tree_fileC    = (char*)malloc(100*sizeof(char));
                blade_out_file          = (char*)malloc(100*sizeof(char));
                blade_tree_file         = (char*)malloc(100*sizeof(char));

                /* Nom du fichier outfile sur la lame */
                strcpy(blade_out_file,chemin);
                strcat(blade_out_file,OUTFILEBLADE);
                strcat(blade_out_file, num_blade);
                strcat(blade_out_file,"_");
                strcat(blade_out_file,num_file);
                strcpy(blade_out_fileC,blade_out_file);
                strcat(blade_out_fileC,FIL);

                /* Nom du fichier d'arbre sur la lame */
                strcpy(blade_tree_file,chemin);
                strcat(blade_tree_file,OUTTREEBLADE);
                strcat(blade_tree_file, num_blade);
                strcat(blade_tree_file,"_");
                strcat(blade_tree_file,num_file);
                strcpy(blade_tree_fileC,blade_tree_file);
                strcat(blade_tree_fileC,FIL);
                openfile(&outfile,blade_out_file,blade_out_fileC,"w",argv[0],outfilename);
                if (trout)
                        openfile(&outtree,blade_tree_file,blade_tree_fileC,"w",argv[0],outtreename);

                /* Traitement de l'arbre */
                maketree2(param);
                tours++;

                /* Fermeture des fichiers intermediaires */
                free(blade_out_file);
                free(blade_tree_file);
                free(blade_out_fileC);
                free(blade_tree_fileC);
                FClose(outfile);
                if(trout)
                        FClose(outtree);
        }
}

/* Synchronisation des resultats */
if(max_procs>1){
        /* Cas des lames autres que le premier */
        if(myid>0 && myid<max_procs){
                MPI_Send( &bestlk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
                MPI_Recv( &leproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
                if(myid==leproc)
                        MPI_Send( &bestour, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
        }
        /* Cas de la premiere lame*/
        else{
                if(myid==0){
                        TBestllhood[0]=bestlk;
                        leproc=myid;
                        for(i=1; i<max_procs;i++)
                                MPI_Recv( &TBestllhood[i], 1, MPI_DOUBLE, i, 0,
                                        MPI_COMM_WORLD, &status );

                        /* Recherche du meilleur resultat */
                        for(i=1; i<max_procs;i++){
                                if( TBestllhood[i]> bestlk){
                                        leproc = i;
                                        bestlk =  TBestllhood[i];
                                }
                        }

                        printf("\n Best lk = %lf et myid = %d", bestlk, leproc);
                        for(i=1; i<max_procs;i++)
                                MPI_Send( &leproc, 1, MPI_INT,i, 0, MPI_COMM_WORLD );

                        if(leproc!=myid)
                                MPI_Recv( &bestour, 1, MPI_INT, leproc, 0,
                                        MPI_COMM_WORLD, &status );
                }
        }
} /* inferencePtree */
```

```
/*=============================================================================
 * FONCTION MAIN                                                              *
 *                                                                            *
 =============================================================================*/
int main(int argc, Char *argv[])
{  /* Dollo or polymorphism parsimony by uphill search */

        /* Variables necessaires a la parallelisation*/
        char *nb, *nbprocs, *nbtours, *comLine ;
        FILE *param=NULL;
        char *paramName;

        char *chemin = (char*)malloc(50*sizeof(char));
        char *inf = (char*)malloc(100*sizeof(char));
        char *outf = (char*)malloc(100*sizeof(char));
        char *outt = (char*)malloc(100*sizeof(char));
        char *weif = (char*)malloc(100*sizeof(char));
        char *ancf = (char*)malloc(100*sizeof(char));

        nbprocs = (char *)malloc(5*sizeof(char));
        nbtours = (char *)malloc(5*sizeof(char));
        nb = (char *)malloc(5*sizeof(char));
        comLine  = (char *)malloc(400*sizeof(char));


#ifdef MAC
        argc = 1;       /* macsetup("Dollop",""); */
        argv[0] = "Dollop";
#endif

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        /* Identification des parametres de la ligne de commande */
        init(argc, argv);
        progname = argv[0];                                     // Nom du programme
        paramName =  argv[1];
        param = fopen(paramName,"r");                           // Fichier de paramètres *
        /* reads in spp, chars, and the data. Then calls maketree to
        construct the tree */

        // Lecture du path et creation des noms des fichiers
        getstryng1(chemin,param);

        /* Creation du nom des differents fichiers avec le path */
        strcpy(inf,chemin);strcpy(outf,chemin);strcpy(weif,chemin);strcpy(outt,chemin);strcpy(ancf,chemin);
        strcat(inf,INFILE);strcat(outf,OUTFILE);strcat(weif,WEIGHTFILE);strcat(outt,OUTTREE);
        strcat(ancf,ANCFILE);

        /* Ouverture du fihier d'entree et de sortie */
        openfile(&infile,inf,"input file","r",argv[0],infilename);
        openfile(&outfile,outf,"output file","w",argv[0],outfilename);

        ibmpc = IBMCRT;
        ansi = ANSICRT;
        garbage = NULL;
        mulsets = false;
        msets = 1;
        firstset = true;

bits = 8*sizeof(long) - 1;
doinit(param);


if (weights || justwts)
        openfile(&weightfile,weif,"weights file","r",argv[0],weightfilename);
if (trout && myid ==0)
        openfile(&outtree,outt,"output tree file","w",argv[0],outtreename);
if(ancvar)
        openfile(&ancfile,ancf,"ancestors file", "r",argv[0],ancfilename);

if (dollo)
        fprintf(outfile, "Dollo");
else
        fprintf(outfile, "Polymorphism");
fprintf(outfile, " parsimony method\n\n");
if (printdata && justwts)
        fprintf(outfile, "%2ld species, %3ld  characters\n\n", spp, chars);

/* Modification de la valeur du seed (au niveau de chaque lame) pour une meilleure parallélisation */
if((myid%2)==0)
        seed[0]=seed[0]*(myid+1);
else
        seed[0]=seed[0]*(myid+1)+1;


/* Cas d'une lame*/
if( numprocs == 1){
        inference(param);
}
else{
        bestlk = (double)LONG_MIN;
        sprintf(nbprocs, " %d ", numprocs);

        /* Cas d'un ensemble de donnees et  d'un arbre */
        if(msets==1 && njumble==1){
                if(myid==0){
                        inference(param);
                }
        }
        else{

                progress  = false;

                /* Cas d'un ensemble de donnees et plusieurs arbes */
                if(msets==1){
                        inferencePtree(param, argv, chemin);
                        if(myid==0){
                                sprintf(nb, " %d ", leproc);    // nb est le processeur ayant le meilleur
                                                                // arbre
                                sprintf(nbtours, " %d ", bestour); // nbtours est le tour auquel l'arbre a
                                                                   // été trouve (numero de l'arbre)
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPtreeML.sh ");
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
```

```
                                system(comLine);  // Execution du script de la formation des fichiers
                                                  // resultats
                        }
                }

                /* Cas de plusieurs msets */
                if(msets > 1 ){
                        /* Fermeture des fichiers de resultat */
                        FClose(outfile);
                        if (trout && myid==0)
                                FClose(outtree);
                        inferencePdata(param, argv, chemin);
                        /* Script sur le premier blade */
                        if(myid==0){
                                sprintf(nb, " %d ", (int)msets);
                                sprintf(nbtours, " %d ", (int)msets/numprocs);
                                /* Formation ligne de commande pour appele le scripts de
                                concatenation des fichiers */
                                sprintf(comLine, "%s ", "./synchroPdataML.sh ");
                                strcat(comLine, nbprocs);
                                strcat(comLine, nb);
                                strcat(comLine, nbtours);
                                strcat(comLine, outf );
                                strcat(comLine, " " );
                                if (trout)
                                        strcat(comLine, outt );
                                printf("\n Script : %s", comLine);
                                system(comLine);  // Execution du script de la formation des fichiers
                                                  // resultats
                        }
                }
        }
        FClose(infile);
        FClose(outfile);
        FClose(outtree);
#ifdef MAC
        fixmacfile(outfilename);
        fixmacfile(outtreename);
#endif
#ifdef WIN32
        phyRestoreConsoleAttributes();
#endif

        MPI_Finalize();
        return 0;
}  /* Dollo or polymorphism parsimony by uphill search */
```

Script de synchronisation en fonction des ensembles de données:

```
#!/bin/bash
# parm1 = input
# parm2 = nombre de lames utilisees
# parm3 = nombre d'ensembles de donnees
# parm4 = Nombre de tour de boucle (par lame) -1

ZERO=0                                          # Constante 0
declare -i numblade;                            # Numero de la lame
declare -i a;                                   # Compteur de boucle
declare -i nbdata;                              # Compteur du nombre d'ensembles de donnees
limite=0;                                       # Limite du compteur de boucle
numblade=0;a=0;nbdata=0;

# Identification du nombre de tours
if [ $2 -gt $3 ]
then
        limite=$3
else
        limite=$2
fi

# Boucle de copie des fichiers
while [[ "$a" -le $4 ]]
do
        numblade=-1
        mpdtrace | while read ligne; do
                numblade=$numblade+1

                if [ $numblade -gt $limite ] || [ $numblade -eq $limite ]
                then
                        continue
                else
                        nbdata=$a*$2+$numblade+1
                        if [ $nbdata -le $3 ]
                        then
                                if [ $numblade -gt $ZERO ]
                                then
                                        echo " Copies des fichiers du $ligne";
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_stat$numblade"_"$a.txt $1_phyml_stat$numblade"_"$a.txt
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_tree$numblade"_"$a.txt $1_phyml_tree$numblade"_"$a.txt
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_lk$numblade"_"$a.txt $1_phyml_lk$numblade"_"$a.txt
                                        if [ "$a" -eq $4 ]
                                        then
                                                ssh $ligne rm  Parallele/PhymlNew/*.txt &
                                                echo "Effacement des fichiers sur le $ligne"
                                        fi

                                fi
                                cat $1_phyml_stat$numblade"_"$a.txt >> $1_phyml_stat.txt
                                cat $1_phyml_lk$numblade"_"$a.txt  >> $1_phyml_lk.txt
                                cat $1_phyml_tree$numblade"_"$a.txt >> $1_phyml_tree.txt
                        fi
                fi
        done
        a=$a+1
done
```

Script de syncrhonisation en fonction des arbres à inférer:

```
#!/bin/bash
# parm1 = input
# parm2 = nombre de lames utilisees
# parm3 = nombre d'arbres inferes
# parm4 = Nombre de tour de boucle (par lame) -1
# parm5 = lame sur laquelle se trouve le meilleur arbre

ZERO=0                                          # Constante 0
declare -i numblade;                            # Numero de la lame
```

```
declare -i a;                                   # Compteur de boucle
declare -i nbtree;                              # Compteur du nombre d'arbres
limite=0;                                       # Limite du compteur de boucle
numblade=0;a=0;nbtree=0;

# Identification du nombre de tours
if [ $2 -gt $3 ]
then
        limite=$3
else
        limite=$2
fi

# Boucle de copie des fichiers
while  [[ "$a" -le $4 ]]
do
        numblade=-1
        mpdtrace | while read ligne; do
                numblade=$numblade+1

                if [ $numblade -gt $limite ] || [ $numblade -eq $limite ]
                then
                        continue
                else
                        nbtree=$a*$2+$numblade+1
                        if [ $nbtree -le $3 ]
                        then
                                if [ $numblade -gt $ZERO ]
                                then
                                        echo " Copies des fichiers du $ligne";
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_stat$numblade"_"$a.txt $1_phyml_stat$numblade"_"$a.txt
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_tree$numblade"_"$a.txt $1_phyml_tree$numblade"_"$a.txt
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_lk$numblade"_"$a.txt $1_phyml_lk$numblade"_"$a.txt



                                        if [ "$a" -eq $4 ] && [ $numblade -eq $5 ]
                                        then
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_best_tree.txt $1_phyml_best_tree.txt
scp mpiuser@$ligne:Parallele/PhymlNew/$1_phyml_best_stat.txt $1_phyml_best_stat.txt
                                                ssh $ligne rm  Parallele/PhymlNew/*.txt &
                                                echo "Effacement des fichiers sur le $ligne"
                                        fi

                                fi

                                #echo " Concatenation des fichiers sur le premier blade"
                                cat $1_phyml_stat$numblade"_"$a.txt >> $1_phyml_stat.txt
                                cat $1_phyml_lk$numblade"_"$a.txt  >> $1_phyml_lk.txt
                                cat $1_phyml_tree$numblade"_"$a.txt >> $1_phyml_tree.txt
                        fi
                fi
        done
        a=$a+1
done
```

PROGRAMME PHYML

Le code source du programme PHYML nous a été fourni par Monsieur S. Guindon qui est un membre de l'équipe qui a developpé le programme. Ma participation porte sur la parallélisation de la méthode d'inférence d'arbres. Les scripts de synchronisation developpés initialement pour les programmes du package PHYLIP ont été utilisés pour la mise à jour des fichiers résultats produits par les différentes lames.

Nous présenterons seulement le fichier main.c qui contient la partie de la parallélisation.

```
/*
PHYML :  a program that  computes maximum likelihood  phylogenies from
DNA or AA homologous sequences
Copyright (C) Stephane Guindon. Oct 2003 onward
All parts of  the source except where indicated  are distributed under
the GNU public licence.  See http://www.opensource.org for details.
*/

#include "utilities.h"
#include "ml.h"
#include "optimiz.h"
#include "bionj.h"
#include "models.h"
#include "free.h"
#include "options.h"
#include "simu.h"
#include <stdio.h>

/*Inclusion de la librairie de MPI*/
#include "mpi.h"

#ifdef PHYML

int T_MAX_FILE;

/*Variables reliees a MPI*/
int   myid, numprocs;
int   namelen;
char  processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;

int mainS();
int main1data1Tree();
int mainPTree();
int mainPdata();

/* Autres variables de PHYML */
seq **data;
allseq *alldata;
option *input;

/* Variables necessaires a la parallelisation*/
char *s_tree, *s_any, *nbtree, *nbdata, *nbprocs, *nbtours, *numblade, *numbestproc, *name;
FILE *fp_phyml_tree,*fp_phyml_stats,*fp_phyml_lk,*fp_best_tree,*fp_best_tree_stats;
```

```
arbre *tree;
int n_otu, n_data_sets;
matrix *mat;
model *mod;
time_t t_beg,t_end;
div_t hour,min;
int num_tree,tree_line_number,i;
double best_loglk;
double startwtime = 0.0, endwtime;
int unproc = 0, indice, nbtour, bestproc;
char *comLine;

/*=============================================================================
 * FONCTION MAIN                                                              *
 *                                                                            *
 =============================================================================*/
int main(int argc, char **argv){

        /* Initialisation des variables MPI */
        MPI_Init(&argc,&argv);

        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);
        srand(time(NULL));
        tree = NULL;
        mod  = NULL;

        /* Initialisation */
        Init_ConstantML(numprocs, myid);
        Init_Constant(numprocs, myid);
        s_any = (char *)mCalloc(T_MAX_FILE,sizeof(char));
        nbdata = (char *)mCalloc(5,sizeof(char));
        name = (char *)mCalloc(T_MAX_FILE,sizeof(char));
        nbtree = (char *)mCalloc(5,sizeof(char));
        numblade = (char *)mCalloc(5,sizeof(char));
        nbprocs = (char *)mCalloc(5,sizeof(char));
        nbtours = (char *)mCalloc(5,sizeof(char));
        numbestproc = (char *)mCalloc(5,sizeof(char));
        comLine  = (char *)mCalloc(400,sizeof(char));


        fflush(stdout);
        input = (option *)Get_Input(argc,argv, myid);
        Make_Model_Complete(input->mod);
        mod = input->mod;


        /* Creation des fichiers resultat au niveau de la premiere lame*/
        if(myid==0){
                fp_phyml_stats = Openfile(input->phyml_stat_file,input->phyml_stat_file_open_mode);
                fprintf(fp_phyml_stats,"\n- PHYML %s -\n\n", VERSION);
                fp_phyml_tree = Openfile(input->phyml_tree_file,input->phyml_tree_file_open_mode);
                fp_phyml_lk = fopen(input->phyml_lk_file,"w");
        }

        if (myid == 0)
                        startwtime = MPI_Wtime();

        n_data_sets = 0;

        if(input->inputtree){
                Test_Multiple_Data_Set_Format(input);
        }
        else {
                input->n_trees =1;
        }

        best_loglk = UNLIKELY;
        tree_line_number = 0;


        if(numprocs<2)
                mainS();
        else{
                sprintf(nbtree, " %d ", input->n_trees);
                sprintf(nbdata, " %d ", input->n_data_sets);
                sprintf(nbprocs, " %d ", numprocs);
                sprintf(name, "%s", input->seqfile);

                if(input->n_data_sets == 1 && input->n_trees==1)
                        main1data1Tree();

                if(input->n_data_sets == 1 && input->n_trees >1){
                        mainPTree();

                        nbtour = input->n_trees/numprocs;
                        bestproc = indice % numprocs;

                        sprintf(nbtours, " %d ", nbtour);
                        sprintf(numbestproc, " %d ", bestproc);

                        /* Formation de la chaine */
                        sprintf(comLine, "%s ", "./synchroPtree.sh ");
                        strcat(comLine, name);
                        strcat(comLine, nbprocs);
                        strcat(comLine, nbtree);
                        strcat(comLine, nbtours);
                        strcat(comLine, numbestproc);
                        if(myid==0){
                                system (comLine);
                        }
                }
                if(input->n_data_sets > 1){
                        input->n_trees =1;
                        mainPdata();
                        nbtour = input->n_data_sets/numprocs;
                        sprintf(nbtours, " %d ", nbtour);

                        /* Formation de la chaine */
                        sprintf(comLine, "%s ", "./synchroPdata.sh ");
                        strcat(comLine, name);
                        strcat(comLine, nbprocs);
                        strcat(comLine, nbdata);
                        strcat(comLine, nbtours);
                        if(myid==0){
                                system(comLine);
                        }
                }
        }
        MPI_Finalize();
        return 0;
} /* END main */
```

```
/*=============================================================================*
 * Version sequentielle de l'inference de l'arbre phylogenetique               *
 *                                                                             *
 *=============================================================================*/
int mainS(){
        /* Traitement de tous les ensembles de donnees */
        do{
                n_data_sets++;
                time(&t_beg);
                n_otu = 0;
                if(n_data_sets > input->n_data_sets)
                        data = NULL;
                else
                        data = Get_Seq(input,0);

                if(data){
                        alldata = Compact_Seq(data,input);
                        Free_Seq(data,alldata->n_otu);
                        Init_Model(alldata,mod);
                        Check_Ambiguities(alldata,input->mod->datatype,input->mod->stepsize);

                        /* Nombre d'arbres */
                        For(num_tree,input->n_trees){

                                /* Inference de l'arbre */
                                if(!input->inputtree){
                                        mat = ML_Dist(alldata,mod,1);
                                        mat->tree = Make_Tree(alldata);
                                        Bionj(mat);
                                        tree = mat->tree;
                                        Free_Mat(mat);
                                }
                                /* Lecture de l'arbre */
                                else {
                                        if(input->n_trees > 1)
                                                printf("\n. Reading user tree [#%d]\n",tree_line_number+1);
                                        else
                                                printf("\n. Reading user tree...\n");

                                        if(input->n_trees == 1) {
                                                rewind(input->fp_input_tree);
                                                tree_line_number = 0;
                                        }
                                        tree = Read_Tree_File(input->fp_input_tree);
                                        tree_line_number++;
                                        if(!tree)  {
                                                printf("\n. Missing tree for data set #%d\n",n_data_sets);
                                                printf("  This data set is not analyzed.\n");
                                                data = NULL;
                                        }
                                        if(!tree->has_branch_lengths){
                                                printf("\n. Computing branch length estimates...\n");
                                                Order_Tree_CSeq(tree,alldata);
                                                mat = ML_Dist(alldata,mod,1);
                                                mat->tree = tree;
                                                mat->method = 0;
                                                Bionj_Br_Length(mat);
                                                Free_Mat(mat);
                                        }
                                }
                                if(!tree)
                                        continue;

                                tree->mod       = mod;
                                tree->input     = input;
                                tree->data      = alldata;
                                tree->both_sides = 1;
                                tree->n_pattern  = tree->data->crunch_len/tree->mod->stepsize;

                                Order_Tree_CSeq(tree,alldata);
                                Make_Tree_4_Lk(tree,alldata,alldata->init_len);

                                if(tree->mod->s_opt->opt_topo)
                                        Simu(tree,1000);
                                else{
                                        if(tree->mod->s_opt->opt_free_param)
                                                Round_Optimize(tree,tree->data);
                                        else{
                                                Lk(tree,tree->data);
                                                printf("\n. Log(lk) :              * -> %15.6f ",tree->tot_loglk);
                                        }
                                }

                                if(tree->mod->bootstrap)
                                        Bootstrap(tree);

                                Update_BrLen_Invar(tree);
                                s_tree = Write_Tree(tree);
                                fprintf(fp_phyml_tree,"%s\n",s_tree);
                                Free(s_tree);
                                Unconstraint_Lk(tree);
                                time(&t_end);
                                hour = div(t_end-t_beg,3600);
                                min  = div(t_end-t_beg,60  );
                                min.quot -= hour.quot*60;

                                /* Cas d'un ensemble de donnees */
                                if (input->n_data_sets==1)
                                        Print_Fp_Out(fp_phyml_stats, t_beg, t_end, tree, input, n_data_sets);
                                else
                                        Print_Fp_Out_Lines(fp_phyml_stats, t_beg, t_end, tree, input,
                                                           n_data_sets);

                                Print_Site_Lk(tree,fp_phyml_lk);

                                /* Cas d'un ensemble de donnees avec plusieurs arbres (ecriture du meilleur
                                arbre trouve) */
                                if((input->n_data_sets == 1) &&(input->n_trees     > 1) &&(tree->tot_loglk >
                                  best_loglk)){
                                        best_loglk = tree->tot_loglk;
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree = fopen(s_any =
                                                strcat(s_any,"_phyml_best_tree.txt"),"w");
                                        s_tree = Write_Tree(tree);
                                        fprintf(fp_best_tree,"%s\n",s_tree);
                                        Free(s_tree);
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree_stats = fopen(s_any =
                                                strcat(s_any,"_phyml_best_stat.txt"),"w");
                                        Print_Fp_Out(fp_best_tree_stats, t_beg,  t_end, tree, input,
                                                     n_data_sets);
                                        fclose(fp_best_tree);
                                        fclose(fp_best_tree_stats);
                                }
```

```
                                Free_Tree_Lk(tree);
                                Free_Tree(tree);
                                if(input->n_data_sets > 1) break;
                        }
                        Free_Cseq(alldata);
                }
        }while(data);

        Free_Model(mod);

        if(input->fp_seq ) fclose(input->fp_seq );
        if(input->fp_input_tree) fclose(input->fp_input_tree);

        fclose(fp_phyml_lk);
        fclose(fp_phyml_tree);
        fclose(fp_phyml_stats);
        Free_Input(input);
        Free(s_any);
        return 0;
} /* END mainS */


/*=============================================================================*
 * Version parallele pour de l'inference de l'arbre pbylogenetique avec un     *
 * ensemble de donnees                                                         *
 *                                                                             *
 =============================================================================*/
int main1data1Tree(){
        /* Traitement de tous les ensembles de donnees */
        if(numprocs>1)
                unproc = 0;
        else
                unproc = 1;
        do{
                n_data_sets++;
                time(&t_beg);
                n_otu = 0;
                if(n_data_sets > input->n_data_sets)
                        data = NULL;
                else
                        data = Get_Seq(input,0);

                if(data){
                        if(n_data_sets > 1 && myid==0)
                                printf("\n. Data set [#%d]\n",n_data_sets);

                        alldata = Compact_Seq(data,input);
                        Free_Seq(data,alldata->n_otu);
                        Init_Model(alldata,mod);
                        Check_Ambiguities(alldata,input->mod->datatype,input->mod->stepsize);

                        /* Nombre d'arbres */
                        For(num_tree,input->n_trees){

                                /* Inference de l'arbre */
                                if(!input->inputtree){
                                        mat = ML_Dist(alldata,mod,unproc);
                                        mat->tree = Make_Tree(alldata);
                                        Bionj(mat);
                                        tree = mat->tree;
                                        Free_Mat(mat);
                                }
```

```
                /* Lecture de l'arbre */
                else {
                        if(input->n_trees == 1) {
                                rewind(input->fp_input_tree);
                                tree_line_number = 0;
                        }
                        tree = Read_Tree_File(input->fp_input_tree);
                        tree_line_number++;
                        if(!tree && myid==0) {
                                printf("\n. Missing tree for data set #%d\n",n_data_sets);
                                printf("  This data set is not analyzed.\n");
                                data = NULL;
                        }
                        if(!tree->has_branch_lengths){
                                Order_Tree_CSeq(tree,alldata);
                                mat = ML_Dist(alldata,mod,unproc);
                                mat->tree = tree;
                                mat->method = 0;
                                Bionj_Br_Length(mat);
                                Free_Mat(mat);
                        }
                }

                if(!tree)
                        continue;

                tree->mod       = mod;
                tree->input     = input;
                tree->data      = alldata;
                tree->both_sides = 1;
                tree->n_pattern  = tree->data->crunch_len/tree->mod->stepsize;

                Order_Tree_CSeq(tree,alldata);
                Make_Tree_4_Lk(tree,alldata,alldata->init_len);

                if(tree->mod->s_opt->opt_topo)
                        Simu(tree,1000);
                else{
                        if(tree->mod->s_opt->opt_free_param)
                                Round_Optimize(tree,tree->data);
                        else{
                                Lk(tree,tree->data);
                                if(myid==0)
                                        printf("\n. Log(lk) :          * -> %15.6f ",
                                                        tree->tot_loglk);
                        }
                }

                if(tree->mod->bootstrap)
                        Bootstrap(tree);

                if(myid==0){
                        Update_BrLen_Invar(tree);
                        s_tree = Write_Tree(tree);
                        fprintf(fp_phyml_tree,"%s\n",s_tree);
                        Free(s_tree);
                        Unconstraint_Lk(tree);
                        time(&t_end);
                        hour = div(t_end-t_beg,3600);
                        min  = div(t_end-t_beg,60 );
                        min.quot -= hour.quot*60;
```

```
                                /* Cas d'un ensemble de données*/
                                if (input->n_data_sets==1)
                                        Print_Fp_Out(fp_phyml_stats, t_beg, t_end, tree, input,
                                                n_data_sets);
                                else
                                        Print_Fp_Out_Lines(fp_phyml_stats, t_beg, t_end, tree,
                                                input, n_data_sets);


                                Print_Site_Lk(tree,fp_phyml_lk);

                                /* Cas d'un ensemble de donnees avec plusieurs arbres (ecriture du
                                meilleur arbre trouve) */
                                if((input->n_data_sets == 1) &&(input->n_trees    > 1) &&
                                  (tree->tot_loglk > best_loglk)){
                                        best_loglk = tree->tot_loglk;
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree = fopen(s_any =
                                                strcat(s_any,"_phyml_best_tree.txt"),"w");
                                        s_tree = Write_Tree(tree);
                                        fprintf(fp_best_tree,"%s\n",s_tree);
                                        Free(s_tree);
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree_stats = fopen(s_any =
                                                strcat(s_any,"_phyml_best_stat.txt"),"w");
                                        Print_Fp_Out(fp_best_tree_stats, t_beg,  t_end, tree, input,
                                                n_data_sets);
                                        fclose(fp_best_tree);
                                        fclose(fp_best_tree_stats);
                                }
                        }
                        Free_Tree_Lk(tree);
                        Free_Tree(tree);
                        if(input->n_data_sets > 1) break;
                }
                Free_Cseq(alldata);
        }
        }while(data);

        Free_Model(mod);
        if(myid==0){
                if(input->fp_seq ) fclose(input->fp_seq );
                if(input->fp_input_tree) fclose(input->fp_input_tree);
                fclose(fp_phyml_lk);
                fclose(fp_phyml_tree);
                fclose(fp_phyml_stats);
        }
        Free_Input(input);
        Free(s_any);

        return 0;
} /* END main1data1Tree */

/*=============================================================================
 * Parallelisation en fonction du nombre d'arbres                              *
 *                                                                             *
 =============================================================================*/
int mainPTree(){

        double  *tab_best_loglk;
        int     *Tindice;

        char    *blade_phyml;
        char    *blade_phyml_stat_file;
        char    *blade_phyml_tree_file;
        char    *blade_phyml_lk_file;
        char    *num_blade;
        char    *num_file;

        /* Allocation de memoire*/
        Tindice          = (int *) malloc((numprocs+1)*sizeof(int));
        tab_best_loglk   = (double *) malloc((numprocs+1)*sizeof(double));
        blade_phyml      = (char *)mCalloc(100,sizeof(char));
        num_blade        = (char *)mCalloc(5,sizeof(char));
        num_file         = (char *)mCalloc(5,sizeof(char));

        /* Fermeture des tampons ouverts precedemment*/
        if(myid==0){
                fclose(fp_phyml_lk);
                fclose(fp_phyml_tree);
                fclose(fp_phyml_stats);
        }

        sprintf(num_blade, "%d", myid);
        strcpy(blade_phyml,input->seqfile);
        strcat(blade_phyml,"_phyml_");

        /* Traitement de tous les ensembles de donnees */
        do{
                n_data_sets++;
                time(&t_beg);
                n_otu = 0;
                if(n_data_sets > input->n_data_sets)
                        data = NULL;
                else
                        data = Get_Seq(input,0);

                if(data){
                        if(n_data_sets > 1)
                                printf("\n. Data set [#%d]\n",n_data_sets);

                        alldata = Compact_Seq(data,input);
                        Free_Seq(data,alldata->n_otu);
                        Init_Model(alldata,mod);
                        Check_Ambiguities(alldata,input->mod->datatype,input->mod->stepsize);

                        /* Nombre d'arbres */
                        For(num_tree,input->n_trees){
                                if(num_tree%numprocs==myid){

                                        /* Ouverture des fichiers d'ecriture */
                                        blade_phyml_stat_file      =
                                                (char *)mCalloc(T_MAX_FILE,sizeof(char));
                                        blade_phyml_tree_file      =
                                                (char *)mCalloc(T_MAX_FILE,sizeof(char));
                                        blade_phyml_lk_file        =
                                                (char *)mCalloc(T_MAX_FILE,sizeof(char));

                                        sprintf(num_file, "%d", num_tree/numprocs);

                                        strcpy(blade_phyml_stat_file, blade_phyml);
                                        strcat(blade_phyml_stat_file,"stat");
                                        strcat(blade_phyml_stat_file, num_blade);
                                        strcat(blade_phyml_stat_file,"_");
```

```
strcat(blade_phyml_stat_file, num_file);
strcat(blade_phyml_stat_file,".txt");

strcpy(blade_phyml_tree_file, blade_phyml);
strcat(blade_phyml_tree_file,"tree");
strcat(blade_phyml_tree_file, num_blade);
strcat(blade_phyml_tree_file,"_");
strcat(blade_phyml_tree_file, num_file);
strcat(blade_phyml_tree_file,".txt");

strcpy(blade_phyml_lk_file, blade_phyml);
strcat(blade_phyml_lk_file,"lk");
strcat(blade_phyml_lk_file, num_blade);
strcat(blade_phyml_lk_file,"_");
strcat(blade_phyml_lk_file, num_file);
strcat(blade_phyml_lk_file,".txt");

fp_phyml_stats = Openfile(blade_phyml_stat_file,
                          input->phyml_stat_file_open_mode);
fp_phyml_tree = Openfile(blade_phyml_tree_file,
                         input->phyml_tree_file_open_mode);
fp_phyml_lk = fopen(blade_phyml_lk_file,"w");

Free (blade_phyml_stat_file);
Free (blade_phyml_tree_file);
Free (blade_phyml_lk_file);


/* Inference de l'arbre */
if(!input->inputtree){
        mat = ML_Dist(alldata,mod,1);

        mat->tree = Make_Tree(alldata);
        Bionj(mat);
        tree = mat->tree;
        Free_Mat(mat);
}
/* Lecture de l'arbre */
else {
        if(myid==0){
                if(input->n_trees > 1)
                        printf("\n. Reading user tree [#%d]\n",
                               tree_line_number+1);
                else
                        printf("\n. Reading user tree...\n");

                if(input->n_trees == 1) {
                        rewind(input->fp_input_tree);
                        tree_line_number = 0;
                }
        }
        tree = Read_Tree_File(input->fp_input_tree);
        tree_line_number++;
        if(!tree && myid==0) {
                printf("\n. Missing tree for data set #%d\n",
                       n_data_sets);
                printf("  This data set is not analyzed.\n");
                data = NULL;
        }
        if(!tree->has_branch_lengths){
                if(myid==0)
                        printf("\n. Computing branch length
                                estimates...\n");
                Order_Tree_CSeq(tree,alldata);
                mat = ML_Dist(alldata,mod,1);
                mat->tree = tree;
                mat->method = 0;
                Bionj_Br_Length(mat);
                Free_Mat(mat);
        }
}

if(!tree){
        fclose(fp_phyml_lk);
        fclose(fp_phyml_tree);
        fclose(fp_phyml_stats);
        continue;
}

tree->mod       = mod;
tree->input     = input;
tree->data      = alldata;
tree->both_sides = 1;
tree->n_pattern  = tree->data->crunch_len/tree->mod->stepsize;

Order_Tree_CSeq(tree,alldata);
Make_Tree_4_Lk(tree,alldata,alldata->init_len);

if(tree->mod->s_opt->opt_topo)
        Simu(tree,1000);
else{
        if(tree->mod->s_opt->opt_free_param)
                Round_Optimize(tree,tree->data);
        else{
                Lk(tree,tree->data);
        }
}

if(tree->mod->bootstrap)
        Bootstrap(tree);

Update_BrLen_Invar(tree);
s_tree = Write_Tree(tree);
fprintf(fp_phyml_tree,"%s\n",s_tree);
Free(s_tree);
Unconstraint_Lk(tree);

/* Cas d'un ensemble de donnees */
if (input->n_data_sets==1)
        Print_Fp_Out(fp_phyml_stats, t_beg, t_end, tree, input,
                     n_data_sets);
else
        Print_Fp_Out_Lines(fp_phyml_stats, t_beg, t_end, tree,
                           input, n_data_sets);


Print_Site_Lk(tree,fp_phyml_lk);
```

```
                                /* Cas d'un ensemble de donnees avec plusieurs arbres (ecriture du
                                meilleur arbre trouve) */
                                if((input->n_data_sets == 1) &&(input->n_trees    > 1) &&
                                  (tree->tot_loglk > best_loglk)){
                                        indice = num_tree;
                                        best_loglk = tree->tot_loglk;
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree = fopen(s_any =
                                                strcat(s_any,"_phyml_best_tree.txt"),"w");
                                        s_tree = Write_Tree(tree);
                                        fprintf(fp_best_tree,"%s\n",s_tree);
                                        Free(s_tree);
                                        strcpy(s_any,input->seqfile);
                                        fp_best_tree_stats = fopen(s_any =
                                                strcat(s_any,"_phyml_best_stat.txt"),"w");
                                        Print_Fp_Out(fp_best_tree_stats, t_beg,  t_end, tree, input,
                                                n_data_sets);
                                        fclose(fp_best_tree);
                                        fclose(fp_best_tree_stats);
                                }
                                fclose(fp_phyml_lk);
                                fclose(fp_phyml_tree);
                                fclose(fp_phyml_stats);
                                Free_Tree_Lk(tree);
                                Free_Tree(tree);
                                if(input->n_data_sets > 1)
                                        break;
                        }
                }
                Free_Cseq(alldata);
        }
}while(data);

//Envoie du meilleur lk au premier blade
if(myid>0){
        MPI_Send( &best_loglk, 1, MPI_DOUBLE,0, 0, MPI_COMM_WORLD );
        MPI_Send( &indice, 1, MPI_INT,0, 0, MPI_COMM_WORLD );
}
else{
        for(i=1;i<numprocs; i++){
                MPI_Recv( &tab_best_loglk[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
                        &status );
                MPI_Recv( &Tindice[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status );
        }
}
for(i=1;i<numprocs; i++){
        if( best_loglk >= tab_best_loglk[i]){
                if(best_loglk > tab_best_loglk[i]){
                        indice=Tindice[i];best_loglk = tab_best_loglk[i];
                }
                else{
                        if(indice>Tindice[i])
                                indice=Tindice[i];
                }
        }
}

Free_Model(mod);
if(input->fp_seq )
        fclose(input->fp_seq );
        if(input->fp_input_tree)
                fclose(input->fp_input_tree);

        Free_Input(input);
        Free(s_any);
        return 0;
} /* END mainPTree */


/*=============================================================================
 * Parallelisation en fonction du nombre d'ensembles de données               *
 *                                                                            *
=============================================================================*/
int mainPdata(){
        char    *blade_phyml;
        char    *blade_phyml_stat_file;
        char    *blade_phyml_tree_file;
        char    *blade_phyml_lk_file;
        char    *num_blade;             // Numero du blade en chaine de caractere
        char    *num_tour;              // Numero de tour de la boucle sur un blade en chaine de caractere

        /* Allocation de memoire*/
        blade_phyml     = (char *)mCalloc(100,sizeof(char));
        num_blade       = (char *)mCalloc(5,sizeof(char));
        num_tour        = (char *)mCalloc(5,sizeof(char));

        /* Fermeture des tampons ouverts precedemment dans le main*/
        if(myid==0){
                fclose(fp_phyml_lk);
                fclose(fp_phyml_tree);
                fclose(fp_phyml_stats);
        }

        sprintf(num_blade, "%d", myid);
        strcpy(blade_phyml,input->seqfile);
        strcat(blade_phyml,"_phyml_");

        For(n_data_sets,input->n_data_sets){
                sprintf(num_tour, "%d", n_data_sets/numprocs);
                data = Get_Seq(input,0);
                n_otu = 0;

                if(n_data_sets%numprocs==myid && data){

                        /* Creation des fichiers d'ecriture */
                        blade_phyml_stat_file   = (char *)mCalloc(T_MAX_FILE,sizeof(char));
                        blade_phyml_tree_file   = (char *)mCalloc(T_MAX_FILE,sizeof(char));
                        blade_phyml_lk_file     = (char *)mCalloc(T_MAX_FILE,sizeof(char));

                        strcpy(blade_phyml_stat_file, blade_phyml);

                        strcat(blade_phyml_stat_file,"stat");
                        strcat(blade_phyml_stat_file, num_blade);
                        strcat(blade_phyml_stat_file,"_");
                        strcat(blade_phyml_stat_file,num_tour);
                        strcat(blade_phyml_stat_file,".txt");

                        strcpy(blade_phyml_tree_file, blade_phyml);
                        strcat(blade_phyml_tree_file,"tree");
                        strcat(blade_phyml_tree_file, num_blade);
                        strcat(blade_phyml_tree_file,"_");
                        strcat(blade_phyml_tree_file,num_tour);
```

```
strcat(blade_phyml_tree_file,".txt");

strcpy(blade_phyml_lk_file, blade_phyml);
strcat(blade_phyml_lk_file,"lk");
strcat(blade_phyml_lk_file, num_blade);
strcat(blade_phyml_lk_file,"_");
strcat(blade_phyml_lk_file,num_tour);
strcat(blade_phyml_lk_file,".txt");

fp_phyml_stats = Openfile(blade_phyml_stat_file,input->phyml_stat_file_open_mode);
fp_phyml_tree = Openfile(blade_phyml_tree_file,input->phyml_tree_file_open_mode);
fp_phyml_lk = fopen(blade_phyml_lk_file,"w");

Free (blade_phyml_stat_file);
Free (blade_phyml_tree_file);
Free (blade_phyml_lk_file);

alldata = Compact_Seq(data,input);
Free_Seq(data,alldata->n_otu);
Init_Model(alldata,mod);
Check_Ambiguities(alldata,input->mod->datatype,input->mod->stepsize);

/* Nombre d'arbres */
For(num_tree,input->n_trees){

        /* Inference de l'arbre */
        if(!input->inputtree){
                mat = ML_Dist(alldata,mod,1);
                mat->tree = Make_Tree(alldata);
                Bionj(mat);
                tree = mat->tree;
                Free_Mat(mat);
        }
        /* Lecture de l'arbre */
        else {
                if(input->n_trees > 1)
                        printf("\n. Reading user tree [#%d]\n",tree_line_number+1);
                else
                        printf("\n. Reading user tree...\n");

                if(input->n_trees == 1) {
                        rewind(input->fp_input_tree);
                        tree_line_number = 0;
                }
                tree = Read_Tree_File(input->fp_input_tree);
                tree_line_number++;
                if(!tree) {
                        printf("\n. Missing tree for data set #%d\n",n_data_sets);
                        printf("  This data set is not analyzed.\n");
                        data = NULL;
                }
                if(!tree->has_branch_lengths){
                        printf("\n. Computing branch length estimates...\n");
                        Order_Tree_CSeq(tree,alldata);
                        mat = ML_Dist(alldata,mod,1);
                        mat->tree = tree;
                        mat->method = 0;
                        Bionj_Br_Length(mat);
                        Free_Mat(mat);
                }
        }
                if(!tree)
                        continue;

                tree->mod      = mod;
                tree->input    = input;
                tree->data     = alldata;
                tree->both_sides = 1;
                tree->n_pattern  = tree->data->crunch_len/tree->mod->stepsize;

                Order_Tree_CSeq(tree,alldata);
                Make_Tree_4_Lk(tree,alldata,alldata->init_len);

                if(tree->mod->s_opt->opt_topo)
                        Simu(tree,1000);
                else{
                        if(tree->mod->s_opt->opt_free_param)
                                Round_Optimize(tree,tree->data);
                        else{
                                Lk(tree,tree->data);
                                printf("\n. Log(lk) :          * -> %15.6f ",tree->tot_loglk);
                        }
                }

                if(tree->mod->bootstrap)
                        Bootstrap(tree);

                Update_BrLen_Invar(tree);
                s_tree = Write_Tree(tree);
                fprintf(fp_phyml_tree,"%s\n",s_tree);
                Free(s_tree);
                Unconstraint_Lk(tree);
                time(&t_end);
                hour = div(t_end-t_beg,3600);
                min  = div(t_end-t_beg,60 );
                min.quot -= hour.quot*60;

                /* Cas d'un ensemble de donnees */
                if (input->n_data_sets==1)
                        Print_Fp_Out(fp_phyml_stats, t_beg, t_end, tree, input, n_data_sets);
                else
                        Print_Fp_Out_Lines(fp_phyml_stats, t_beg, t_end, tree, input,
                                          n_data_sets);

                Print_Site_Lk(tree,fp_phyml_lk);

                Free_Tree_Lk(tree);
                Free_Tree(tree);
                if(input->n_data_sets > 1) break;
        }
        Free_Cseq(alldata);

        fclose(fp_phyml_lk);
        fclose(fp_phyml_tree);
        fclose(fp_phyml_stats);
    }
}
Free_Model(mod);
if(input->fp_seq ) fclose(input->fp_seq );
if(input->fp_input_tree) fclose(input->fp_input_tree);


Free_Input(input);
```

```
        Free(s_any);
        return 0;
} /* END mainPdata */
#endif
/*************************************************************/
```