

# Département Génie Mathématique

## Modèle de parallélisme en îles pour des métaheuristiques

En relation avec l'équipe  
**DOLPHIN - Inria Lille**



**Rapport de projet semestriel**  
Septembre 2012 - Janvier 2013

Alexandre Quemy & Thibault Lasnier

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	Objectifs . . . . .	4
<b>2</b>	<b>Contexte</b>	<b>5</b>
2.1	Contexte . . . . .	5
2.1.1	ParadisEO . . . . .	5
2.1.2	Algorithme génétique à base de population . . . . .	6
2.2	Introduction au parallélisme . . . . .	7
2.2.1	Taxonomie de Flynn . . . . .	7
2.3	Les architectures parallèles . . . . .	8
2.3.1	Mémoire partagée . . . . .	8
2.3.2	Mémoire distribuée . . . . .	8
2.4	Modèle d'îles . . . . .	9
<b>3</b>	<b>Modélisation</b>	<b>11</b>
3.1	Modélisation . . . . .	11
3.1.1	Identification des briques logicielles . . . . .	11
3.1.2	Cas d'utilisation . . . . .	11
3.1.3	Diagramme de classes . . . . .	12
3.1.4	Topologie . . . . .	13
3.1.5	Iles . . . . .	15
3.1.6	Politique d'intégration . . . . .	15
3.1.7	Politique de migration . . . . .	15
3.1.8	Modèle en îles . . . . .	16
3.1.9	Communications . . . . .	17
3.1.10	Tolérance aux pannes . . . . .	18
<b>4</b>	<b>Implémentation</b>	<b>19</b>
4.1	Préambule . . . . .	19
4.2	Modèle . . . . .	19
4.2.1	Table de correspondance . . . . .	19
4.3	Mécanismes d'îles . . . . .	19
4.3.1	Notificateur d'îles . . . . .	19
4.3.2	Processus de création de l'île . . . . .	20
4.4	Réalisation du modèle hétérogène . . . . .	21
4.5	Optimisation . . . . .	22
4.5.1	Optimisation des communications . . . . .	22
4.5.2	Optimisation des copies . . . . .	23
<b>5</b>	<b>Le livrable</b>	<b>24</b>
5.1	Etat du livrable . . . . .	24
5.2	Améliorations . . . . .	24
5.3	Utilisation . . . . .	25
5.3.1	Modèle homogène . . . . .	25
5.3.2	Modèle hétérogène . . . . .	26
5.3.3	Mécanismes avancés . . . . .	27

<b>6</b>	<b>Tests</b>	<b>31</b>
6.1	Tests . . . . .	31
6.2	Tests unitaires . . . . .	31
6.3	Tests de couverture . . . . .	32
6.4	Tests de performances . . . . .	33
6.4.1	Modèle homogène vs Sérialisé . . . . .	33
6.4.2	Influence du facteur communication . . . . .	34
<b>7</b>	<b>Conclusion</b>	<b>38</b>
7.1	Conclusion . . . . .	38
<b>8</b>	<b>Annexes</b>	<b>40</b>
8.1	Diagramme de classes . . . . .	40

# Chapitre 1

## Introduction

### 1.1 Introduction

Lorsque l'on parle de parallélisme, on pensera la majorité du temps à un gain de performance se traduisant par un temps de calcul moins long. Il existe deux grands types de parallélisme qui sont le parallélisme de données et le parallélisme de tâches.

Le premier vise à effectuer en parallèle un traitement unique et coûteux en temps sur des données. Il s'agit essentiellement de modifier un algorithme déjà existant pour l'adapter au parallélisme. Le second type préfère paralléliser des tâches qui ne dépendent pas l'une de l'autre, permettant également un gain de temps. Outre la modification d'algorithmes déjà existants, ce type de parallélisme fait naître de nouveaux algorithmes dont l'avantage n'est pas nécessairement la réduction du temps de calcul mais une meilleure stabilité numérique, une plus grande convergence ou d'autres propriétés mathématiques intéressantes. C'est le cas par exemple du modèle en îles, qui par nature, n'existe que sur des architectures parallèles. Ces algorithmes sont évidemment plus spécifiques et ne s'appliquent que dans des domaines précis.

Dans le cadre du projet semestriel de GM4, nous avons eu l'occasion de travailler sur un projet en relation avec l'équipe DOLPHIN d'Inria Lille. Il s'agissait de développer un modèle de parallélisme en îles pour le module de parallélisme à mémoire partagée du logiciel ParadisEO, dédié à la conception et l'implémentation de méthodes stochastiques d'optimisation.

Dans un premier temps, nous présenterons ce qu'est un modèle en îles et les avantages théoriques qu'il peut avoir et développerons le contexte du projet (présentation de ParadisEO, rappels de parallélisme), puis dans un second temps nous détaillerons le travail effectué en termes de modélisation et d'implémentation.

La dernière partie du rapport présentera les tests de performances effectués et leur analyse, ainsi que la manière d'utiliser le modèle en îles tel qu'il est implémenté et d'en étendre les possibilités pour des modèles complexes.

### 1.2 Objectifs

Les objectifs du projet sont les suivants :

- Développer un modèle de parallélisme d'îles coopératives avec des îles homogènes (même types d'algorithmes) et une topologie en anneau.
- Généraliser le modèle d'îles pour qu'il fonctionne avec des îles hétérogènes.
- Implémenter différentes topologies (cube, étoile, dynamique...).
- Hybrider les deux niveaux de parallélisme master / slave et modèle d'îles.

Il nous était demandé de nous focaliser sur le premier point, avec une modélisation de qualité permettant progressivement et facilement d'étendre le modèle pour lui ajouter des fonctionnalités : modèle hétérogène, hybridation, nouvelles topologies...

En amont de la modélisation, il nous a fallu appréhender rapidement le sujet, à la fois vis à vis du modèle en îles en tant que tel mais également et surtout au regard du contexte. En effet, il fallait prendre en compte le framework EO sur lequel allait reposer notre travail, notamment en comprenant son design global, son fonctionnement et ses composantes.

# Chapitre 2

## Contexte

### 2.1 Contexte

#### 2.1.1 ParadisEO

ParadisEO est un framework open-source développé en C++ et dédié au développement de méthodes d'optimisation approchées classiques, multi-objectifs, parallèles et hybrides. Le logiciel est composé de plusieurs modules dont voici une brève description.

##### **Evolving Objects**

Evolving Objects était un framework indépendant de ParadisEO, destiné à l'élaboration de métaheuristiques à base de population comme les algorithmes génétiques.

Développé initialement par l'équipe TAO de Inria Saclay, menée par Marc Schoenauer, le framework fut par la suite maintenu par Thales. Enfin, fin 2012, les projets ParadisEO et Evolving Objects ont fusionnés pour laisser lieu au seul logiciel ParadisEO, EO devenant un module à part entière de ce dernier.

La totalité des autres modules de ParadisEO reposent sur Evolving Objects.

##### **MO : Recherche locale**

Le module MO s'articule autour des algorithmes de recherche locale et des métaheuristiques à base de voisinage. On parle également de méthodes de trajectoire dans le sens où ces algorithmes itératifs vont construire des trajectoires dans l'espace des solutions en tentant de se diriger vers une solution optimale.

On retrouve dans cette catégorie des algorithmes classiques d'optimisation combinatoire comme le recuit simulé ou la recherche tabou.

##### **MOEO : Optimisation multi-objectifs**

Le module MOEO permet la création d'algorithmes d'optimisation multiobjectifs. L'un des outils principaux dans l'optimisation multiobjectifs est le front de Pareto qui permet de donner un ensemble de solutions dont chacune est optimale pour un ou plusieurs critères par rapport aux autres solutions de cet ensemble. D'abord utilisés en finance et en économie, les fronts de Pareto permettent également la résolution de problèmes liés à l'industrie.

##### **PEO**

Le module PEO est un module transversal permettant la parallélisation des métaheuristiques des autres modules ainsi que l'élaboration du modèle de parallélisme en îles. Le module propose du parallélisme à mémoire partagée grâce aux pThreads et du parallélisme à mémoire distribuée

Pour des raisons de design et de technologies, le module PEO est devenu obsolète et durant l'été 2012, un nouveau module de parallélisme à mémoire partagée, nommé SMP, a été développé, intégrant uniquement un modèle maître / esclave.

En remplacement de la partie distribuée, un module de EO utilisant MPI est en cours de développement.

## SMP

SMP pour Shared Memory Parallelism est le nouveau module de parallélisme à mémoire partagée de ParadisEO. Développé en C++11, il se contente de fournir un modèle Maître / Esclave d'où la nécessité de fournir PEO le temps d'ajouter à SMP les fonctionnalités manquantes.

## Autres modules

Il existe d'autres modules plus spécifiques comme EPMpi, le penchant à mémoire distribuée de SMP, ainsi qu'un module GPU utilisant CUDA.

Enfin, EDO est un module pour la création d'algorithmes à estimation de distributions.

### 2.1.2 Algorithme génétique à base de population

#### Principe des algorithmes génétiques

Les algorithmes génétiques font partie des méthodes dites d'intelligence calculatoire. Ils se basent sur la théorie de l'évolution darwiniste pour résoudre un problème en faisant évoluer un ensemble de solutions à un problème donné, dans l'optique de trouver les meilleurs résultats. Ces algorithmes sont principalement stochastiques, car ils utilisent itérativement des processus aléatoires.

La majorité de ces algorithmes servent à résoudre des problèmes d'optimisation. On parle donc de métaheuristiques.

Parmi les grandes méthodes présentes dans ParadisEO, on note le recuit simulé, la recherche tabou, la recherche itérative locale ou encore des méthodes MCMC (Markov Chain, Monte Carlo) comme l'algorithme de Metropolis-Hastings, et des méthodes plus spécifiques comme le NSGA-II (Non-Dominated Sorting Genetic Algorithms) et IBEA (Indicator-Based Evolutionary Algorithm) pour l'optimisation multi-objectifs.

#### Fonctionnement d'un algorithme génétique

Il existe une multitude de métaheuristiques et les algorithmes génétiques à base de population en font partie. Étant donné que le modèle en îles dont fait l'objet ce projet présuppose des algorithmes à base de population, nous ne présenterons que ce type de méthodes.

Nous partons d'une population initiale où chaque individu représente une solution potentielle au problème. Cette population est générée aléatoirement.

Ensuite, la population est évaluée, c'est à dire que l'on attribue un indicateur qualitatif à chaque individu. La manière d'évaluer est très spécifique au problème et pas toujours évidente à mettre au point. L'évaluation est généralement l'opération la plus coûteuse.

Les meilleurs individus sont alors sélectionnés. Ils sont croisés. C'est à dire que l'on va prendre les meilleures parties de chaque individu pour en former un nouveau. On effectue alors des mutations avec une probabilité assez faible. Les mutations permettent d'éviter de tomber dans des optimums locaux, mais doivent être assez rares pour conserver une convergence de l'algorithme. Enfin, les nouveaux individus sont replacés dans la population initiale, les moins bons sont supprimés pour garder une population de taille constante.

Le processus est itéré le nombre de générations voulues ou selon des paramètres divers : temps, nombre d'évaluations, etc.

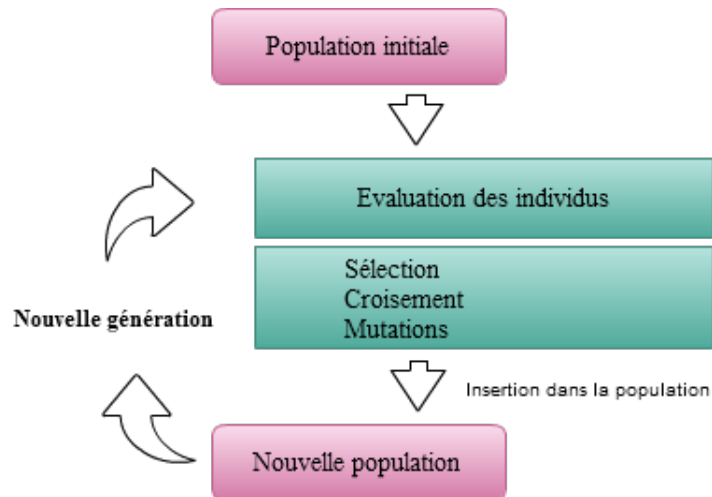


FIGURE 2.1 – Principe des algorithmes génétiques à base de population

## 2.2 Introduction au parallélisme

Le parallélisme consiste à pouvoir effectuer du traitement d'informations de manière simultanée, s'opposant ainsi au traitement de l'information séquentiel. Cela repose d'une part sur des architectures matérielles spécifiques ainsi que sur l'implémentation spécifique d'algorithmes pour ces architectures. Il faut alors que le problème se prête au parallélisme ce qui est le cas de nombreux problèmes récurrents dans la modélisation et la simulation en générale, le traitement de l'information et de l'image, etc.

### 2.2.1 Taxonomie de Flynn

La taxonomie de Flynn, proposée en 1966 par Michael J. Flynn permet de classer les différents types d'architectures en fonction de leur traitement des données et des instructions. Même si cette classification est aujourd'hui mise en défaut par des architectures hybrides, elle demeure simple à comprendre et permet toujours d'englober la majorité des architectures modernes.

#### SISD (Single Instruction, Single Data)

Cette architecture correspond à un ordinateur séquentiel qui n'effectue aucun parallélisme, ni de donnée, ni de tâche. C'est typiquement le cas des processeurs monocore et premiers ordinateurs en règle générale.

#### SIMD (Single Instruction, Multiple Data)

Le SIMD permet d'appliquer une même instruction à un jeu de données multiples. Ce genre d'architecture est donc très utile dans le cas de calculs matriciels et c'est pourquoi beaucoup de processeurs modernes intègrent des instructions SIMD dont tirent pleinement profit des bibliothèques comme BLAS par exemple.

Les processeurs vectoriels font partie de cette catégorie et sont surtout utilisés dans le calcul intensif, loin du grand public. Pour l'anecdote, la Playstation 3, console de jeu de Sony, embarque un processeur Cell qui fait partie de cette catégorie. C'est pour cette particularité que de nombreuses entités ont construit des clusters entièrement basés sur des Playstation 3 : des instituts de recherche au département de la défense des USA. Notons également que les GPU reposent en règle générale sur ce genre d'instructions. On peut citer par exemple le framework CUDA de NVidia.

#### MISD (Multiple Instruction, Single Data)

Ici une même donnée sera traitée par plusieurs processeurs en parallèle. Ce genre d'architecture est peu exploité et donc plutôt exotique.

## MIMD (Multiple Instructions, Multiple Data)

Ce dernier mode de fonctionnement désigne les machines multi-processeurs qui permettent de traiter des données différentes, avec des instructions différentes de manière parallèle. Chaque processeur exécute son code de manière asynchrone et indépendante. La cohérence des données est assurée par la synchronisation des processeurs qui dépend de l'organisation de la mémoire. C'est le mode le plus courant avec l'avènement des processeurs multicores et c'est également celui qui va nous intéresser plus particulièrement dans ce projet.

## 2.3 Les architectures parallèles

Parmi les architectures MIMD, on peut distinguer deux grands types d'architecture parallèle, en fonction de l'organisation de la mémoire. Le choix de l'architecture est dictée par des raisons économiques : le matériel à mémoire partagée est plus cher que celui du distribuée.

### 2.3.1 Mémoire partagée

Dans ce cas, les processus lancés, aussi appelés processus légers ou thread, accèdent à une mémoire commune. C'est typiquement le cas des processeurs multicœurs qui partagent la mémoire et des caches assurent à bas niveau une cohérence des accès. Ici, la synchronisation se fait à l'aide de sémaphores ou de mutex (on parle d'exclusion mutuelle mais il s'agit pratiquement d'une sémaphore à deux états). Une technique consiste également à utiliser des types dit atomic puisque la lecture ou l'écriture de variables atomic se fait en cycle d'horloge, garantissant un accès thread-safe.

L'avantages de cette architecture est que l'overhead, c'est à dire le coût supplémentaire lié aux communications, est faible voire inexistant puisqu'il n'y a pas de passage par le réseau. De plus, il n'y a pas besoin (ou pas souvent) de s'assurer de la cohérence de la mémoire.

Enfin, contrairement au parallélisme à mémoire distribuée, il s'utilise sur un seul noeud.

### Technologies multithread

Au niveau C++, les technologies pour faire du multithread sont variées. Historiquement, c'est la librairie pthread, pour POSIX thread qui est apparue la première, pour le langage C. A défaut d'autre bibliothèque aussi complète, elle fut souvent utilisée également en C++ malgré une approche très C-like à base de pointeurs de fonctions. Evidemment, les pthreads ne sont pas portables.

La librairie OpenMP est également très populaire puisque portable mais aussi très facile à utiliser. Elle est assez scalable dans le sens où elle permet une parallélisation intuitive (on délèguera tout à la bibliothèque) ou une parallélisation plus fine. Malgré le fait qu'elle ne fasse pas partie du standard du C++ et n'est pas disponible nativement, sa popularité a fait que la plupart des compilateurs intègrent une implémentation d'OpenMP.

La bibliothèque Boost propose également des threads. L'avantage est qu'ils sont portables puisqu'ils wrappent tantôt les pthreads, tantôt les winthread selon l'environnement sur lequel le programme est exécuté. Boost étant une bibliothèque éprouvée et réputée, elle est très proche du standard et c'est donc une dépendance couramment admise.

Enfin, avec la nouvelle norme du C++, le standard définit un modèle de parallélisme proposant, entre autres, des threads dont l'objectif étant évidemment d'être portable. En l'état actuel des choses, il s'agit principalement d'une surcouche des pthreads avec du sucre syntaxique pour coller avec un C++ moderne.

### 2.3.2 Mémoire distribuée

Les processeurs ont ici chacun leur mémoire et n'ont pas de moyen d'accéder directement à la mémoire des autres processeurs. Les informations sont envoyés sous la forme de messages, ce qui est une opération coûteuse. Les noeuds sont des machines indépendantes pouvant être regroupés dans le cadre de clusters ou de grilles. Les noeuds communiquent entre eux via le réseau grâce à des middleware comme MPI ou des sockets.

On retrouve différents types de topologie réseau veillant à transmettre l'information de noeud en noeud. Parmi la plus utilisée dans le monde du calcul intensif on retrouve l'hypercube qui possède les caractéristiques d'arc-transitivité et de distance-transitivité, ce qui implique que toutes ses arêtes jouent le même rôle et que tous ses



sommets ont les mêmes propriétés de distance.

Comme les noeuds n'ont généralement pas de mémoire commune, cela pose des problèmes notamment pour la cohérence de la mémoire puisque différents processus sont lancés sur chaque noeud.

## MPI

MPI pour The Message Passing Interface, est une norme décrivant une bibliothèque C, C++ et FORTRAN (même si d'autres langages ont vu naître leur propre implémentation) de communication par envoi de messages, et donc utile pour le parallélisme à mémoire distribuée. Cette norme décrit les communicateurs, groupes de processus capables de communiquer entre eux, et des méthodes de communications. Les deux principales sont le point à point qui permet à deux processus d'un même communicateur d'échanger une donnée, et le broadcast qui permet d'impliquer tous les processus d'un communicateur dans l'échange de données.

On retrouve plusieurs implémentations de MPI ayant chacune leurs spécificités, notamment au niveau du support matériel mais aussi de part des détails d'implémentation. L'implémentation officielle est MPICH2, mais on retrouve également OpenMPI qui est très populaire et d'ailleurs utilisée dans le nouveau module de parallélisme à mémoire distribuée de EO.

## 2.4 Modèle d'îles

Le théorème dit du *no free lunch* démontre qu'il n'existe pas de métaheuristique meilleure qu'une autre et que la qualité d'une métaheuristique ne peut-être établi qu'au regard de certaines classes de problèmes et dépend également des paramètres de l'algorithme voire de son implémentation. Ainsi, il est difficile d'anti-

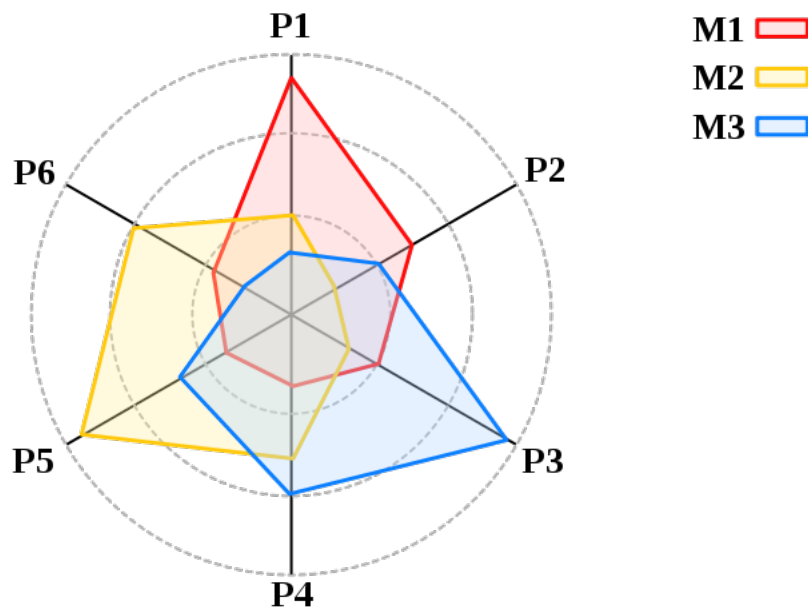


FIGURE 2.2 – Illustration du théorème no free lunch

ciper l'adéquation d'une métaheuristique sur une instance quelconque sans expérimentation ou une phase d'apprentissage. Un facteur primordial de l'efficacité d'une méthode à base de population sur une instance donnée réside dans l'équilibre entre l'intensification (c'est à dire une sélection élitiste des individus ce qui a pour conséquence de forcer la convergence) et la diversification (c'est à dire élargir le champs de recherche des solutions).

On comprendra aisément la difficulté à allier à la fois ces deux concepts antagonistes.

Le modèle en îles consiste à lancer plusieurs algorithmes génétiques à base de population, les îles, qui vont évoluer de manière indépendante mais interagir par moments. Cela permet des convergences locales d'algorithmes tout en conservant une diversité globale.

L'organisation des îles au sein du modèle est régie par une topologie qui peut influencer la tolérance à la convergence locale. En effet, une île isolée, avec peu de communications vers d'autres îles, impactera moins la convergence prématurée ou non, du modèle tandis qu'une île centrale, comme il y en a une dans une topologie en étoile, pourra avoir des effets bénéfiques ou néfastes selon la politique de l'île.

Typiquement, le facteur principal influençant la diversification / intensification du modèle (c'est à dire à un niveau d'abstraction supérieur à celui du simple algorithme) est la politique propre à chaque île. Les interactions entre îles sont définies par certaines règles : périodicité dans le nombre de générations, périodicité temporelle, mais également des considérations sur la qualité d'une solution, etc.

Dans les modèles classiques, la politique et la topologie sont des attributs statiques ne permettant pas d'adap-

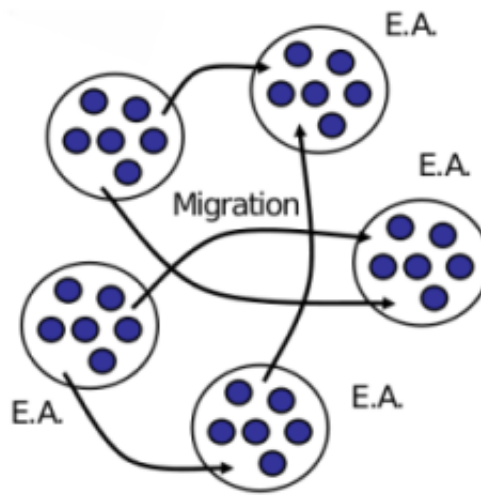


FIGURE 2.3 – Modèle en îles d'après une topologie quelconque.

ter dynamiquement les étapes de diversification ou d'intensification.

Des travaux actuels s'intéressent à des topologies dynamiques, changeantes au cours de l'évolution des algorithmes selon certains signaux (des informations sur la convergence des individus, l'intégration des population migrantes,...).

De la même manière, la topologie peut être stochastique. Chaque lien inter-île a une certaine probabilité d'être établi au moment d'une migration. Ces probabilités peuvent être statiques, mais également dynamique, permettant, par exemple, d'isoler des îles n'apportant rien en terme de diversité ou de qualité des solutions.

En résumé, les apports théoriques du modèle en îles résident dans une vitesse de convergence plus rapide, un domaine de recherche plus vaste et une meilleure gestion des étapes de diversification / intensification qui se traduit par la sortie potentielle de certains optimums locaux.

# Chapitre 3

## Modélisation

### 3.1 Modélisation

Il est à noter que les éléments présents dans cette section sont ceux de la modélisation telle qu'elle a été prévue puis validée lors de la réunion avec Clive Canape le 5 Novembre 2012. Lors de l'implémentation, certaines parties ont été revues et raffinées pour diverses raisons : problèmes techniques mais aussi et surtout meilleure solution et scalabilité.

Ces changements ont surtout eu lieu au regard des diagrammes de classes et de séquences.

Nos connaissances sur le sujet et le contexte du projet n'étaient pas homogène. Ainsi nous avons décidé de réaliser l'intégralité de la modélisation ensemble, quitte passer plus de temps sur cette première étape primordiale.

#### 3.1.1 Identification des briques logicielles

La première étape a été de définir les briques logicielles qui vont composer le modèle en îles. Cette étape s'est faite naturellement à la lecture de divers documents sur le sujet. On peut citer « Metaheuristics : from design to implementation » de E-G. Talbi ainsi que la thèse « Une théorie asymptotique des algorithmes génétiques » de R. Cerf.

Les briques ainsi identifiées sont :

- Le modèle d'île, collection d'îles organisées selon une topologie.
- L'île, algorithme génétique étendu par des mécanismes de communications.
- La topologie, structure décrivant l'agencement des îles au sein d'un modèle.
- La politique de migration, définissant les critères d'envoi des individus.
- La politique d'intégration, définissant la manière dont une île intègre de nouveaux individus.

Ce découpage a fait apparaître 4 axes de travail, tant pour la modélisation que pour l'implémentation, à savoir :

- Les conteneurs et wrappers nécessaires au modèle : le modèle d'île et l'île.
- L'organisation des îles : la topologie.
- La gestion des individus : les politiques de migration et d'intégration.
- Les communications entre les îles à l'aide des 3 axes précédents.

#### 3.1.2 Cas d'utilisation

Nous avons décomposés les cas d'utilisation par rapport à l'utilisateur en 3 catégories :

- Ceux concernant l'île.
- Ceux concernant le modèle d'îles.
- Ceux concernant la topologie.

#### Cas d'utilisation relatifs à l'île

La majorité des cas d'utilisation de l'île sont ceux de l'algorithme évolutionniste qu'elle va envelopper et donc ne les avons pas détaillés (de plus, selon l'algorithme, il sont susceptibles de changer).

Pour l'utilisateur final, l'île n'est autre qu'un algorithme avec des mécanismes étendus, dont l'utilisateur lambda n'est pas dans l'obligation de connaître. Les cas d'utilisation sont donc les suivants :

- Changer la population sur laquelle l'algorithme va travailler.
- Changer la politique d'intégration.
- Changer la politique de migration.

Ces besoins fonctionnels étant suffisamment simples, il n'a pas été jugé nécessaire de les développer plus.

#### Cas d'utilisation relatifs aux modèle d'îles

De la même manière, les cas d'utilisations pour le modèle d'îles sont les suivants :

- Exécuter le modèle.
- Obtenir les individus de toutes les îles.
- Ajouter des îles.
- Changer la topologie.

Le premier point sera largement détaillé par la suite et le second est implicite puisque comme nous le verrons, seuls des références sont manipulés par le modèle et les îles.

Enfin, les deux derniers ne nécessitaient pas un développement plus grand.

#### Cas d'utilisation relatifs à la topologie

Enfin, pour la topologie, nous avons identifiés le cas d'utilisation suivant :

- Création d'une topologie personnalisée.

Les autres sont implicites : création de la topologie souhaitée qui se traduit par la simple instanciation de l'objet `Topology` comme nous le verrons.

Nous devons tout de même développer le cas d'utilisation de la topologie personnalisée. L'utilisateur devra écrire sa topologie sous forme matricielle dans un fichier texte. Il était prévu d'utiliser le parser JSON de EO mais celui-ci étant trop restrictif et alourdissant considérablement la syntaxe pour l'utilisateur final, il a été décidé qu'un simple fichier texte serait une meilleure solution. L'utilisateur pourra alors faire parser son fichier pour obtenir un objet topologie ayant les caractéristiques désirées.

Dans le cas où le nombre d'îles du modèle ne correspond pas à la dimension de la matrice, une exception sera jetée et l'exécution du modèle avortée. De même, une exception sera lancée dans le cas où l'utilisateur ne respecte pas le format d'entrée spécifié.

#### Scénario principal

Dans le but d'illustrer l'utilisation finale du modèle d'îles et l'utilité de chacun des besoins fonctionnels identifiés, voici le scénario principal :

- L'utilisateur crée les données communes aux îles.
- L'utilisateur crée ses îles, avec pour chaque îles les étapes suivantes :
  - Création d'une population.
  - Création des parties propres à l'algorithme (exemple : critère d'arrêts).
  - Création de la politique de migration.
  - Création de la politique d'intégration.
- L'utilisateur crée sa topologie.
- L'utilisateur crée son modèle en l'initialisant avec sa topologie.
- L'utilisateur ajoute les îles au modèle.
- L'utilisateur lance l'exécution du modèle.
- (L'utilisateur effectue des traitements sur les populations : tri, etc.)
- L'utilisateur récupère et exploite les résultats.

On notera que certaines étapes peuvent s'effectuer dans n'importe quel ordre. Ainsi, seule la première étape doit vraiment précéder la création des îles, et évidemment, le modèle doit être créé et rempli avec des îles pour pouvoir être lancé.

Cette flexibilité au niveau de la création du modèle pose cependant un problème dont il faudra tenir compte lors de la création des diagrammes de séquences : la topologie est créée sans nécessairement connaître le nombre d'îles sur lequel elle va devoir travailler.

### 3.1.3 Diagramme de classes

Le diagramme général des classes étant trop grand pour tenir sur la page, celui-ci se trouve en annexe.

### 3.1.4 Topologie

#### Rôle de monteur

Le Monteur est un patron de conception destiné à séparer le processus de création de sa représentation finale. Dans le cas d'une topologie, le processus de création est la partie la plus complexe puisqu'il s'agit d'initialiser correctement la représentation matricielle du graphe associée à telle ou telle topologie. Ainsi, la classe Topology sera simplement composée d'une matrice et d'une manière d'initialiser cette topologie. Cette manière d'initialiser est le Monteur et il est passé en template à la création de l'objet.

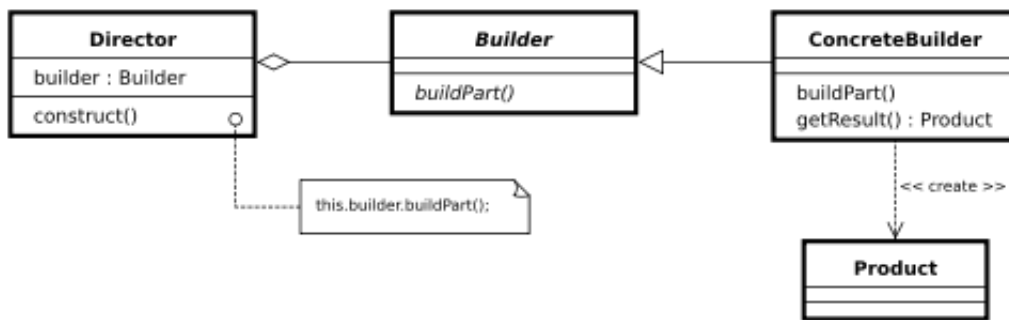


FIGURE 3.1 – Illustration du DP Builder.

**Application du monteur** Dans notre cas, comme l'initialisation est la partie complexe de la topologie, nous avons choisie de la déléguer à un builder, où chaque spécialisation de la classe abstraite TopologyBuilder va initialiser la matrice suivant un algorithme particulier. Toutes les topologies qui ont une structure mathématique, et qui peuvent être construites par un algorithme doivent passer par ce processus de construction. De manière générale, pour une topologie, nous avons besoin de la construire, de connaître les voisins d'un nœud, et d'isoler un nœud (ie : couper les communications depuis et vers ce nœud).

Pour certaines topologies, des paramètres autres que le nombre de nœuds interviennent lors de la création.

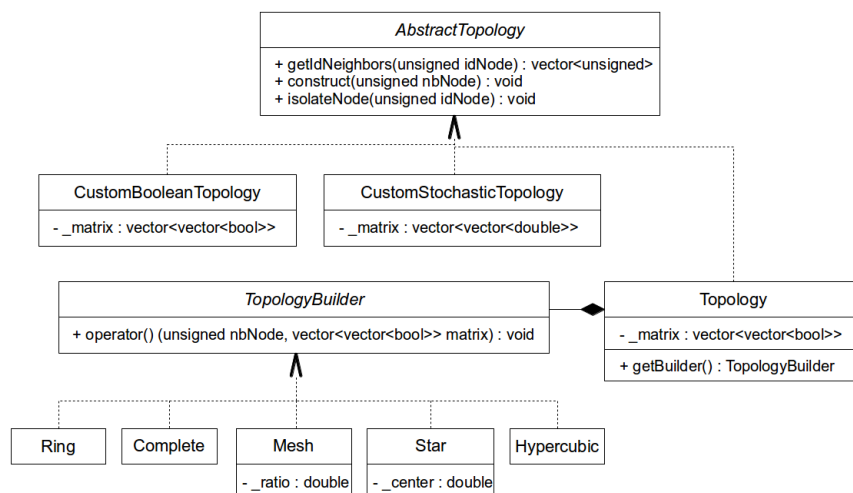


FIGURE 3.2 – DP Builder appliqué à la topologie.

Par exemple, la topologie en étoile doit savoir quel sommet constitue le centre, ou encore la topologie maillée est décrite par un ratio qui représente l'allongement de la structure. Ces paramètres sont en fait des attributs du builder en question, qui ont la possibilité d'être changé à tout moment, ce qui a pour effet de reconstruire la topologie en fonction de ce paramètre.

**Topologies personnalisées** Dans les cas très particuliers d'une topologie sans véritable structure, la possibilité est laissée de charger un fichier contenant directement la matrice des connexions entre les nœuds. On n'a alors plus besoin de monteur, car l'initialisation se fait uniquement par le fichier, et l'instance de topologie ne peut être reconstruite différemment.

Il existe deux types de topologies pour cette situation. Les topologies dites booléennes, qui pour un nœud donné, renverront toujours les mêmes nœuds voisins ; et les topologies dites stochastiques, où la communication entre le nœud  $i$  et le nœud  $j$  se fait avec une probabilité  $p_{i,j}$ , probabilité située à la  $i$ ème ligne et  $j$ ème colonne de la matrice. A chaque fois que l'on demande les voisins d'un certain nœud, on n'aura pas toujours la même réponse.

**Algorithmie** Dans la construction des topologies, certaines sont assez faciles à aborder, comme la topologie en anneau, et d'autres le sont moins, comme l'hypercubique. Pour cette étude, nous montrerons des matrices constituées de 0 et de 1 pour plus de commodité.

La topologie d'hypercubique dispose d'une contrainte de taille : elle ne peut être constituée que de  $2^n$  nœuds, où  $n$  est la dimension de l'espace vectoriel le plus petit contenant l'hypercube. Si l'on choisit de représenter l'hypercube par un cube unité de dimension  $n$ , alors la coordonnée de chaque sommet est un  $2^n$ -upplet composé de  $n$  fois 1 et  $(2^n - n)$  fois 0.

Tous ces  $2^n$ -upplets sont différents, et on peut obtenir une bijection entre cet ensemble de points et les entiers

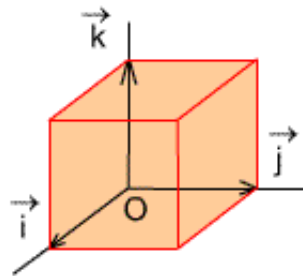


FIGURE 3.3 – Cube unité.

de 0 à  $2^n - 1$  par simple conversion binaire. Ceci nous donne une façon de numérotter les sommets de l'hypercube.

Deux sommets adjacents de l'hypercube sont reliés par une arête, parallèle à un vecteur unité. Donc pour passer d'un sommet à l'autre, on ne peut uniquement changer qu'une coordonnée du  $2^n$ -upplet. Ce qui signifie que deux sommets adjacents ont des numéros qui ne diffèrent que d'un chiffre dans la base 2, ce qu'il faut expliquer aux personnes souhaitant maîtriser les nœuds qui communiquent entre eux.

Pour construire la matrice correspondante, on peut s'inspirer de la géométrie. En effet, on peut remarquer une relation de récurrence sur la construction des hypercubes. De la même manière qu'un cube est construit par deux carrés (dimension 2) relié par 4 arêtes (22), un hypercube de dimension  $n$ , est construit avec deux hypercubes de dimension  $n-1$ , reliés par  $2^{n-1}$  arêtes.

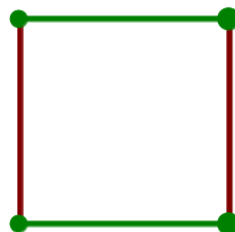


FIGURE 3.4 – Le carré est la réunion de 2 hypercube de dimension 1 (vert) et de 2 arêtes (rouge).

On sait que pour passer d'un hypercube à celui de dimension supérieure, on double la taille de la matrice ( $H_n$ ), et chaque nœud dispose d'un nouveau voisin. Intuitivement, avec la propriété citée ci-dessus, on com-

prend que l'on peut construire la matrice par bloc : deux fois la matrice  $H_{n-1}$  et en plus une communication par ligne et par colonne :

$$H_n = \begin{pmatrix} H_{n-1} & I_{n-1} \\ I_{n-1} & H_{n-1} \end{pmatrix}$$

En commençant par  $H_0 = (0)$ , on obtient :

$$H_3 = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

### 3.1.5 Îles

L'île est un wrapper d'algorithmes génétiques à base de population auxquels elle rajoute des mécanismes de communications. Il faut donc absolument que l'île dérive de eoAlgo, la classe de base des algorithmes à base de population.

Pour savoir vers qui envoyer ses messages, un design pattern Observer est mis en place entre l'île, qui sert de notificateur, et le modèle, qui sert d'observateur.

Comme le modèle est asynchrone, il faut stocker les individus qui arrivent en attendant que l'île ne soit plus occupée. En effet, le seul moment où elle peut techniquement vérifier si de nouveaux individus sont arrivés et les intégrer est entre chaque génération effectuée (car nous ne pouvons maîtriser ce qui se passe entre le début et la fin d'une génération).

### 3.1.6 Politique d'intégration

La politique d'intégration définit la manière dont une île qui reçoit des individus doit l'intégrer au sein de sa propre population. Il s'agit ici du même mécanisme que le mécanisme de remplacement des anciens individus par les nouveaux individus générés par un algorithme génétique lors d'une génération.

Cela signifie que les mécanismes de la politique d'intégration sont déjà implémentés par ParadisEO et qu'il suffira de manipuler ces objets déjà existants.

### 3.1.7 Politique de migration

#### Élément de politique

Pour une gestion fine de la politique de migration, la politique de migration ne peut pas être modélisé comme un eoCheckPoint classique, à savoir un design pattern décorateur qui ne permettrait pas de dissocier les différentes conditions de migration et les différents processus de sélection.

Pour réaliser ceci, la politique de migration est modélisée par un conteneur d'éléments de politique où chaque élément est constitué d'un quand et d'un qui.

Le quand est le critère de migration, implémenté par un eoContinue, tandis que qui représente le processus de sélection des individus, implémenté par un eoSelect.

De cette manière il est possible d'avoir par exemple une telle politique :

Quand	Qui
Toutes les 10 générations	5% de la population totale
A partir de 30 secondes, toutes les 5 secondes	Le meilleur individu

La condition logique ET du second élément de politique peut être réalisée grâce à un eoContinue déjà existant dans EO : eoCombinedContinue. Le OU logique peut évidemment être réalisé en utilisant un eoCheckPoint comme critère de migration.



### 3.1.8 Modèle en îles

#### Rôle d'algorithme

Le premier des rôles du modèle est celui d'algorithme. En effet, il va hériter de la classe `eoAlgo` qui est la classe de base des algorithmes à base de population. Cela permettra des modèles hybrides où une île est elle-même un modèle en îles.

Outre cette perspective d'hybridation, cet héritage a tout de même une sémantique puisque le modèle est un vrai algorithme et ses rôles sont multiples.

#### Rôle de conteneur

Le second rôle identifié du modèle est celui de conteneur. Dans un premier temps il était envisagé que chaque île ait sa propre population en son sein et que seuls les éléments nécessaires à son initialisation soient passés au constructeur de l'île. Ainsi, le modèle aurait dû proposer une API permettant de manipuler et accéder aux populations ainsi qu'aux îles.

Finalement, la solution retenue, qui est également la plus simple et limitant la complexité de l'API est celle de passer les populations par référence. Ainsi le rôle de conteneur du modèle en îles est réduit aux besoins internes du modèle : les îles et les populations étant passées par références et donc accessibles depuis l'extérieur.

#### Rôle de médiateur / routeur

Un des rôles primordial du modèle est celui de médiateur. En effet, pour des raisons de scalabilité, la topologie ne connaît pas les îles et les îles ne connaissent ni la topologie ni les autres îles.

Ainsi, le modèle d'île possèdera une table associative qui réalisera une bijection entre les sommets de la topologie et les îles. Ainsi, pour la migration, chaque île doit passer par le modèle afin de savoir vers quelle île envoyer sa population. Le modèle centralise donc l'intégralité des communications et joue le rôle de routeur ou médiateur.

Tout comme l'île, le modèle doit avoir un conteneur contenant les populations migrantes. Cette population doit également s'assortir d'un pointeur sur l'île envoyant la population afin que le modèle puisse trouver les voisins grâce à sa table de correspondance.

#### Algorithme

Lorsque l'utilisateur demande au modèle de se lancer, l'algorithme principal peut être découpé en 3 étapes :

- Initialisation du modèle.
- Ordonnancement des requêtes de migration.
- Arrêt de l'algorithme (synchronisation des threads tant au niveau des îles que des requêtes).

La phase d'initialisation consiste à créer la topologie et construire la table de correspondance. En effet, le modèle a véritablement besoin de connaître la topologie et d'avoir une table de correspondance qu'au moment où l'algorithme est lancé.

Pour éviter de reconstruire la topologie et la table pendant la phase d'ajout des îles par l'utilisateur (cela poserait également problème puisque certaines topologies ne permettent pas d'être construites avec n'importe quel nombre d'îles) cela n'est fait qu'une fois l'algorithme lancé.

Ensuite, des threads sont créés pour lancer les îles en parallèles.

Commence alors la seconde phase. Tant qu'il reste des îles actives, nous vérifions périodiquement s'il y a des requêtes à satisfaire. De plus, nous regardons si une île est inactive pour la première fois, auquel cas nous devons l'isoler dans la topologie afin de ne plus envoyer d'individu vers elle.

La dernière phase permet d'éviter à l'implémentation divers problèmes de synchronisme. Dans un premier temps, nous attendons la fin de l'exécution du thread de chaque île (lui-même attendant la fin de l'exécution de ses requêtes de migration). Ensuite, nous attendons la fin de l'exécution de chacune des dernières requêtes du modèle (c'est à dire l'ajout dans les queues de populations à intégrer pour certaines îles).

Enfin, comme les îles sont arrêtées, il reste potentiellement des individus à intégrer dans leur queue de travail. Ces individus peuvent être intéressants pour la solution finale retenue et donc nous forçons une dernière intégration sur chacune des îles.



### Critère d'arrêt

Le modèle doit s'arrêter lorsque plus aucune île n'est en activité. Pour savoir si une île est arrêtée, il ne faut pas regarder le statut du thread sur lequel elle tourne mais simplement le statut de l'algorithme qu'elle wrappe. En effet, une île va effectuer son algorithme, avec les envois éventuels de population. Comme chaque message requiert la création d'un thread et que l'île est la seule en mesure de connaître ses messages créés (et donc les threads), elle doit, une fois son algorithme terminé, attendre l'exécution complète de ses migrations, c'est à dire attendre la fin de vie des threads. Dans le cas contraire, il serait possible que le programme entier termine avant qu'une requête ne soit exécutée, entraînant ainsi une erreur fatale.

Ainsi, c'est un booléen supplémentaire indiquant le statut de l'algorithme d'une île que le modèle va vérifier. Cela permet d'isoler plus rapidement une île au sein de la topologie et donc de diminuer le nombre de requêtes inutiles.

### Changement de topologie pendant l'exécution

Il est envisageable de changer de topologie pendant l'exécution. Cela implique de reconstruire la topologie mais également d'isoler à nouveau toutes les îles qui ne sont plus actives, en se référant au booléen de statut dont il est question dans le paragraphe sur le critère d'arrêt.

Il faudra également faire attention à bien protéger l'accès à la topologie durant sa modification et sa reconstruction. Cela bloquera l'intégration des requêtes vers et depuis le modèle, mais normalement le changement de topologie est assez rare pour que cela ait un effet notable sur la vitesse et l'engorgement du modèle.

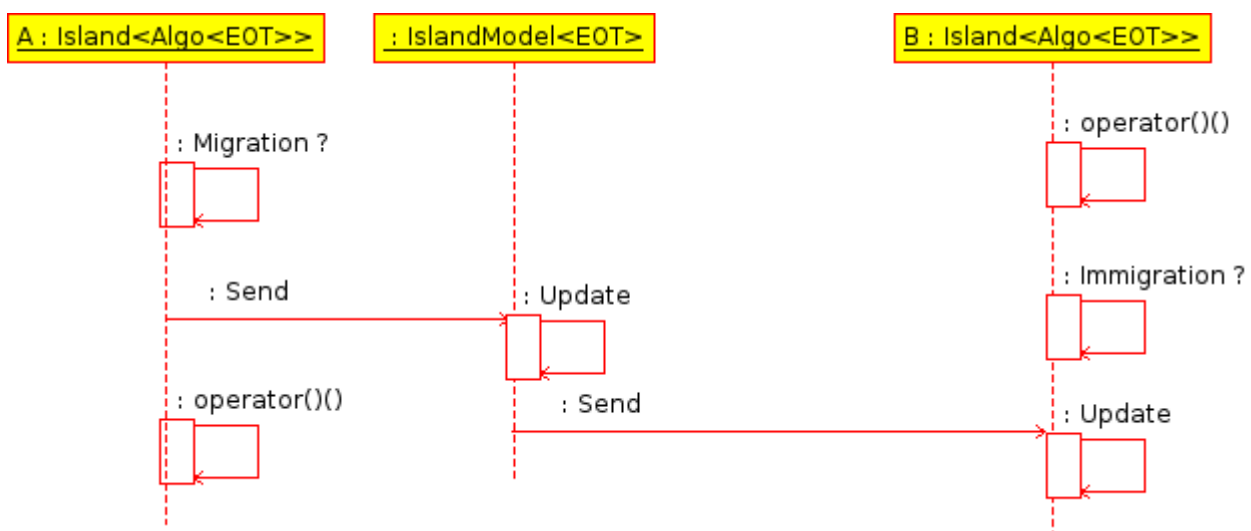
### 3.1.9 Communications

Les communications sont le point crucial du modèle. Le modèle étant asynchrone, il faut qu'une île qui fait migrer des individus ne soit pas bloquée en attendant que ceux-ci arrivent à destination. De même, durant la réception, le modèle ne doit pas être bloqué.

Pour cela, comme nous l'avons vu, l'île et le modèle disposent d'un conteneur pour stocker les populations arrivantes. Ces deux classes disposent également d'une méthode Update et d'une méthode Send permettant respectivement l'ajout dans le conteneur et l'envoi d'un élément du conteneur. Ce conteneur étant une queue, c'est sur le principe FIFO (First In / First Out) que se fait l'envoi.

### Séquence de migration

FIGURE 3.5 – Diagramme de séquence schématique d'une migration.



A la fin de l'exécution d'un génération, l'île A vérifie sa politique de migration. Si un critère est rempli, elle déclenche sa méthode Send. Cette méthode sélectionne les individus à migrer en fonction du critère atteint, et va créer un nouveau thread bindé avec la méthode Update du modèle qui observe l'île. Cela permet de libérer l'île migrante et ne pas bloquer le modèle qui lui est potentiellement en train d'effectuer une migration. Lorsque le modèle n'est pas occupé, il vérifie périodiquement sa queue afin de déclencher, si elle n'est pas

vide, sa méthode Send. Cette méthode Send va demander la liste des voisins de l'île dont est issue la population migrante considérée. Elle va ensuite créer un thread par voisin, chacun étant chargé d'exécuter la méthode Update de l'île en question.

Finalement, l'île B qui a reçu des individus va intégrer ces individus en fonction de sa politique d'intégration. Le test sur la présence d'individus à intégrer est réalisé au même moment que celui sur la migration, c'est à dire entre deux générations.

Il s'agira principalement à l'implémentation de faire attention aux accès concurrents sur les queues de travail, notamment celle du modèle qui est très sollicitée et d'éviter les goulots d'étranglements (cf. Optimisation des communications).

### 3.1.10 Tolérance aux pannes

Dans le cadre du parallélisme à mémoire partagée, la panne qui n'entraîne pas la chute de l'ensemble du système est très rare.

A contrario, dans le cadre du parallélisme à mémoire distribuée, la tolérance à la panne est primordiale puisqu'un noeud est plus susceptible de tomber en panne ou de ne plus être accessible par le réseau.

Etant donné que notre travail portait uniquement sur du mémoire partagée, notre travail sur la tolérance aux pannes a été limitée. Notons que par le design de ParadisEO, les populations d'un algorithme ne peut croître ou diminuer sous peine de lancer une exception. Cela implique donc que les individus sont migrés avec une sémantique de valeur (ils sont copiés pour chaque île voisine mais une copie reste dans l'île d'origine), et non pas avec une sémantique d'entité (l'individu quitterait définitivement son île et ne pourrait migrer que vers une seule île).

De fait, la tolérance à la perte subite d'un processeur ne résulterait qu'en la perte des individus des îles fonctionnant sur ce processeur, ce qui est un moindre mal. De même, les migrants qui auraient dû être envoyés vers une île qui ne répond plus ne seraient pas perdus puisque copiés.

Quand à la répartition de la charge sur les processeurs, il est difficile de faire quelque chose de précis en mémoire partagée et donc cela est laissé à l'ordonnanceur de l'OS. C'est par contre une problématique importante en mémoire distribuée (avec par exemple une répartition intelligente des îles sur les différents noeuds selon la topologie des îles et la topologie réseau).

# Chapitre 4

## Implémentation

### 4.1 Préambule

Ce chapitre dédié à l'implémentation est un chapitre technique qui traite des difficultés et des techniques d'implémentations mises en oeuvre. Sa lecture n'est pas obligatoire pour la suite et n'intéressera que d'éventuels développeurs ou des personnes avec une certaine affinité pour les casse-têtes en C++...

### 4.2 Modèle

#### 4.2.1 Table de correspondance

La table de correspondance doit réaliser une bijection entre les îles et les sommets du graphe de la topologie. Le C++ ne disposant pas d'un conteneur bimap, il a fallu développer le noter.

Il s'agit d'un conteneur basique réalisant une bijection et permettant d'accéder aux éléments soit par la droite soit par la gauche de l'association.

Au niveau interne, la bijection est stockée par deux `std::map`, l'une codant l'association droite et l'autre l'association gauche. L'information stockée est directement du type des templates spécifiés à la création de l'objet. Ainsi, il faut faire attention à la copie des objets et il vaut mieux templater avec des pointeurs sur les entités à "stocker". Il ne s'agit donc pas réellement d'un conteneur mais vraiment d'un mécanisme permettant de stocker une association bidirectionnelle.

```
1 template<class A, class B>
2 class Bimap
3 {
4 protected :
5     std::map<A,B> rightAssociation ;
6     std::map<B,A> leftAssociation ;
7 };
```

Listing 4.1– Classe Bimap pour la représentation de bijection.

### 4.3 Mécanismes d'îles

#### 4.3.1 Notificateur d'îles

Étant donné qu'une fois l'algorithme lancé, le seul moment contrôlable pour permettre la vérification à la fois de la politique de migration et la politique d'intégration se situe à chaque génération. L'algorithme va alors tester chacun de ses `eoContinue`, qui forment un ensemble de critère d'arrêt pour l'algorithme.

L'objectif était donc de pouvoir injecter un objet en tant que `eoContinue` qui permettrait de vérifier les politiques à chaque génération. La méthode la plus simple aurait été de demander explicitement à l'utilisateur d'ajouter la politique de migration et d'intégration aux `eoContinue` de l'algorithme, au moment où il le crée. Sémantiquement, il apparaît que ces mécanismes de notifications n'ont rien à voir avec les `eoContinue` et donc nous avons décidé que leur création et leur gestion relevaient de la gestion interne de l'île et que l'utilisateur

final devrait avoir à passer sa politique de migration, sa politique d'intégration et les `eoContinue` de l'algorithme à wrapper par l'île de manière séparée.

La première approche a été d'inclure, comme prévue par la modélisation initiale, la vérification de la politique de migration et d'intégration directement dans l'algorithme, par l'intermédiaire de l'injection de la politique dans ses `eoContinue`.

Le principal défaut de cette approche est qu'une fois injectée, l'île n'a plus de contrôle pour modifier sa politique, si, un jour, l'API offre des stratégies dynamiques d'évolution de politique au court de l'algorithme du modèle d'îles.

Pour pallier à ce problème, un `eoContinue` nommé `IslandNotifier` a été créé. Il prend en paramètre un pointeur sur l'île à notifier, ainsi qu'une tâche bindée, sans paramètre et de type de retour `void`. Cette tâche est nécessaire, par construction, une méthode de la classe d'île et sera lancée par l'île à notifier, dans l'opérateur `()` de l'`IslandNotifier`. Ainsi, une fois l'`IslandNotifier` ajouté aux `eoContinue` de l'algorithme, à chaque génération, lorsque celui-ci vérifiera la validité de ses `eoContinue`, il lancera malgré lui une méthode de l'île à qui il appartient.

Cet `IslandNotifier` est bien entendu assez générique grâce aux `std::function`, mais dans notre cas, on fera appel à une méthode `check` qui vérifiera les politiques de l'île. Ainsi, l'île peut contrôler et modifier sa politique puisque c'est elle même qui vérifie sa politique.

Cela a également une seconde conséquence : l'île ne peut lancer l'algorithme sur un thread séparé pour des raisons évidentes. L'algorithme wrappé par l'île et l'île communiquent en permanence de manière synchrone et il ne peut en être autrement. En effet, la validité d'un élément de migration se fait au regard de l'état de la population à un instant  $t$  et si, pendant la vérification par l'île l'algorithme continue, la population se retrouvera à un état à l'instant  $t + \Delta$ .

Ainsi, c'est l'île tout entière qui doit être lancée sur un thread, ou elle même un thread, selon les considérations de conception.

A noter que le concept de Notifier a été étendu permettant à chacun d'utiliser le mécanisme de callback pour effectuer une tâche quelconque. C'est particulièrement utile, comme expliqué dans la partie utilisation du livrable, pour la gestion dynamique de la politique ou de la topologie.

### 4.3.2 Processus de création de l'île

Pour les raisons évoquées dans le paragraphe précédant, la politique de migration et d'intégration est passée en marge des autres arguments donnés pour la création de l'algorithme évolutionniste. Il faut donc pouvoir injecter notre `IslandNotifier` bindé sur la méthode `check` de l'île lors de la création de l'île.

Le déroulement de cette injection est le suivant :

- Récupération des `eoContinue` dans le `parameter pack`.
- Injection du notificateur d'île dans les `eoContinue` récupérés.
- Reconstruction du `parameter pack`.
- Création de l'algorithme wrappé par l'île.

Le `parameter pack` est un objet contenant un nombre quelconque d'éléments de type quelconque. De part sa généralité, il permet de grandement factoriser le code pour permettre de wrapper un très grand nombre d'objets différents au sein d'un même wrapper. En l'occurrence, notre modèle d'île doit pouvoir faire tourner n'importe quel type d'algorithme à base de population, quel qu'en soit l'API.

Il est cependant assez difficile de manipuler des `parameter pack` sans passer par des techniques de métaprogrammation à base de template.

Dans notre cas, nous avons opté par une expansion de `parameter pack` grâce à une technique récursive aliant template et `type_trait`.

Le coût de l'expansion du `parameter pack` et de la récupération de la variable du type souhaité est relativement léger : deux appels de méthodes lors de la création de chaque île, et un temps de compilation légèrement plus long.

Cependant, une fois la référence sur les `eoContinue` récupérée, il n'est pas possible de directement ajouter les notificateurs d'îles puisque la création de l'algorithme doit se faire dans la liste d'initialisation. En effet, le coût de recopie des différents objets nécessaires à la création de l'algorithme peut-être relativement important. Ainsi, la classe `Island` hérite de manière privée d'une classe `ContWrapper` uniquement destinée à accueillir un `eoCheckPoint`, conteneur de `eoContinue`. A la construction d'un `ContWrapper`, celui-ci ajoute les `eoContinue` récupérés dans le `parameter pack` et crée un `IslandNotifier` bindé sur la méthode `check` de l'île en question.

Enfin, par la technique du `tag dispatching`, le `parameter pack` est reconstruit et celui-ci passé à la liste d'initialisation pour la création de l'algorithme.

L'avantage de wrapper les continuators de l'utilisateur dans un autre conteneur avec le Notifier plutôt que d'ajouter le notifier dans les continuators de l'utilisateur réside dans l'absence d'effet de bord. En effet, l'utilisateur n'est pas nécessairement au courant des mécanismes internes de création de l'île et ne s'attend pas à ce que ses continuators soient modifiés (comme tout est passé par référence). Ainsi, si l'on imagine qu'il puisse réutiliser ses `eoContinue` pour une autre expérience à la suite, ceux-ci pourraient contenir le Notifier bindé sur une instance d'île qui n'existe plus et ainsi provoquer un comportement indéfini très difficile à localiser.

## 4.4 Réalisation du modèle hétérogène

La difficulté de la réalisation du modèle hétérogène a été double : proposer une API inchangée pour l'utilisateur et avoir une abstraction suffisante pour supporter n'importe quel type d'île.

Dans un premier temps nous pensions que des conversions explicites fonctionneraient, par exemple entre une particule d'un PSO et un individu d'un EA classique. En effet, la particule est simplement un individu avec des propriétés supplémentaires. Ces propriétés auraient simplement été perdues lors de la conversion. Cela n'a pas été possible pour des raisons techniques. En effet, si l'on ne prend que l'exemple du PSO et de ses particules, une particule va être templâtée sur le type d'individu dont elle étend le comportement et la description. Ainsi, il étant impossible pour l'île de connaître le type d'individu de la particule puisque l'île ne possède pas de `Template Parameter` sur l'individu. De même, l'île doit absolument connaître le type de la population manipuler par le modèle auquel elle est rattachée et inversement (du fait du `pattern observer` entre l'île et le modèle et part le prototype des mécanismes de communication tant au niveau de l'île que du modèle).

De fait, un nouveau concept a été introduit. Il s'agit de celui de population de base, c'est à dire la population de base sur laquelle doit travailler le modèle et le type des individus qui vont transiter par tous les mécanismes de communication.

L'île et le modèle se voit affubler d'un nouveau template, nommé `bEOT`. L'île concrète hérite d'une île abstraite dont le seul template est le `bEOT` et l'interface celle des mécanismes de communication (manipulant uniquement des `bEOT`).

Il restait à définir deux choses : d'une part il faut pouvoir convertir tous les types utilisés par les îles vers le `bEOT` (et inversement, convertir les types manipulés par les îles vers des `bEOT`), et d'autre part il fallait prévoir un comportement par défaut qui soit transparent pour l'utilisateur qui ne veut utiliser que des mécanismes homogènes.

La solution technique à la première interrogation a été de demander à l'utilisateur de créer ses propres fonctions de conversions. Ce système est plus souple et élégant qu'une tentative de `transtypage` violent et surtout non connu de l'utilisateur qui n'a aucun contrôle sur la manière dont ce `transtypage` va s'effectuer. Au contraire ici, c'est lui qui donne la sémantique voulue à la conversion et il peut ainsi faire fonctionner des modèles avec des îles sur des individus qui n'ont à priori rien à voir au niveau de l'implémentation (ce qui n'était pas possible avec un simple `cast` qui n'aurait pu deviner comment effectuer la conversion). Au niveau de l'implémentation, cela se traduit par deux arguments en plus dans le constructeur de l'île qui sont des `std::functions`.

Pour la seconde interrogation, la solution résidait dans le fait qu'il est possible d'utiliser un template par défaut qui soit lui même un template ! Ainsi nous avons pu écrire :

```
1 template<template <class> class EOAlgo, class EOT, class bEOT = EOT>
2 class Island : private ContWrapper<EOT, bEOT>, public AIsland<bEOT>
3 {};
```

Listing 4.2– Template par défaut.

Ceci posait tout même le problème d'avoir une fonction de conversion par défaut. Pour cela, nous avons utilisé une lambda function qui se comporte comme la fonction identité puisqu'elle renvoie l'individu qu'elle a prit en paramètre. De plus, grâce au C++11 elle effectue un perfect forwarding permettant de n'avoir aucune empreinte si ce n'est l'appel à la fonction depuis la vtable.

Un second constructeur se charge donc d'initialiser comme il faut grâce aux lambdas :

```
1 template<template <class> class EOAlgo, class EOT, class bEOT>
2 template<class ... Args>
3 paradiseo::smp::Island<EOAlgo,EOT,bEOT>::Island(eoPop<EOT>& _pop, IntPolicy<EOT>&
4 _intPolicy, MigPolicy<EOT>& _migPolicy, Args&... args) :
5     Island(
6         // Default conversion functions for homogeneous islands
7         [](bEOT& i) -> EOT { return std::forward<EOT>(i); },
8         [](EOT& i) -> bEOT { return std::forward<bEOT>(i); },
9         _pop, _intPolicy, _migPolicy, args...)
10 {} }
```

Listing 4.3– Fonction de conversions par défaut.

Ainsi, les mécanismes des îles sont tels qu'elles convertissent à chaque fois les individus à migrer vers le type de base et convertissent de fait tous les individus qu'elles reçoivent vers leur type. Elles peuvent le faire indépendamment du fait de savoir si elles manipulent la même population que la population de base ou non.

Typiquement :

```
1 eoPop<EOT> migPop; // Population migrante
2 _select(pop, migPop); // Selectionnee en fonction de la politique
3
4 // Convert pop to base pop
5 eoPop<bEOT> baseMigPop; // Population migrante convertie
6 for(auto& indi : migPop) // On ajoute dans cette nouvelle population les individus
7     // convertis
8     baseMigPop.push_back(convertToBase(indi));
```

Listing 4.4– Mécanisme de conversion avant envoi.

## 4.5 Optimisation

### 4.5.1 Optimisation des communications

Lors des premiers tests d'intégration effectués sur le modèle homogène, nous nous sommes aperçu d'un phénomène d'engorgement qui finissait par ralentir de manière très significative le modèle.

Dans un premier temps, le modèle, lorsqu'il exécutait sa méthode Send, accédait à toute les requêtes, c'est à dire qu'il allait créer un nouveau thread pour chaque population arrivée dans sa queue de travail. De fait, l'accès à la queue de travail est bloqué pour éviter les accès concurrents. Ainsi, les requêtes venant d'îles étaient mises en attente jusqu'à la fin de l'accès à la queue par le modèle. Le modèle refait un tour de sa boucle principale, et pendant ce temps toutes les requêtes qui étaient en attente déposent leur population dans la queue de travail. Le modèle réexécute sa méthode Send sur une queue de travail contenant bien plus de populations migrantes que la première fois, impliquant un temps de traitement bien plus important et donc le monopôle de l'accès à la queue durant un laps de temps plus important. Le nombre de requêtes en attente augmente donc de manière très significative.

Pour pallier à ce problème, la solution adoptée a été de simplement accéder à une seule requête plutôt que de

toute les satisfaire. Nous avons donc un mécanisme bien plus fluide ce qui se ressent sur l'exécution générale du modèle.

```

1  template<class EOT>
2  void paradiseo::smp::IslandModel<EOT>::send(void)
3  {
4      std::lock_guard<std::mutex> lock(m);
5      if (!listEmigrants.empty())
6      {
7          // Get the neighbors
8          unsigned idFrom = table.getLeft()[listEmigrants.front().second];
9          std::vector<unsigned> neighbors = topo.getIdNeighbors(idFrom);
10
11         // Send elements to neighbors
12         eoPop<EOT> migPop = std::move(listEmigrants.front().first);
13         for (unsigned idTo : neighbors)
14             sentMessages.push_back(std::thread(&AIland<EOT>::update, table.getRight()[
15                 idTo], std::move(migPop)));
16
17         listEmigrants.pop();
18     }
19 }
```

Listing 4.5– Méthode Send du modèle.

### 4.5.2 Optimisation des copies

On remarquera qu'un certain nombre de copies d'objets peuvent être effectuées, notamment lors de l'envoi sur un thread mais également lorsque la population est converti dans le type de base ou depuis le type de base. Ainsi, il a fallu regarder précisément quelles étaient les copies d'objets qui pouvaient être supprimées pour optimiser la complexité spatiale. Les principales copies à supprimer sont les suivantes :

- Lors de l'envoi de la population migrante vers le modèle, une seule copie est nécessaire.
- Lors de la conversion des migrants vers le type de base, aucune copie n'est nécessaire.

Pour supprimer ces copies, nous avons utilisé un mécanisme du C++11 à savoir le déplacement sémantique via l'opération `std::move`.

Ainsi si l'on prend l'exemple de l'envoi de la population d'une île vers le modèle :

```

1  sentMessages.push_back(std::thread(&IslandModel<bEOT>::update, model, std::move(
    baseMigPop), this));
```

Listing 4.6– Envoi du message avec déplacement de population.

Ici, la population migrante ne sert plus localement à la fonction `Send` de l'île. Il est donc inutile de créer une nouvelle copie pour la passer au thread, surtout que la population peut être importante. Ainsi, le déplacement sémantique va éviter cette copie en "cassant" le lien entre l'identifiant de la variable `baseMigPop` et le contenu mémoire, à savoir les individus, pour remettre ce lien sur un nouvel indentifiant (en fonction du prototype de la méthode `Update` lancée par le thread).

Il faut tout de même rester prudent sur ce genre d'optimisation puisque le `std::move` a un coût en temps qui n'est pas négligeable et c'est donc une histoire de compromis entre complexité spatiale et temporelle. Dans le cas présent, la taille des individus et à plus forte raison d'une population justifie amplement l'utilisation d'un tel mécanisme.



# Chapitre 5

## Le livrable

### 5.1 Etat du livrable

Le livrable est composé du modèle d'îles homogène et hétérogène. Il propose des mécanismes supplémentaires pour la réalisation de politiques ou d'évènements complexes en accord avec les recherches actuelles sur le sujet.

Un certain nombre de topologies usuelles sont également codées : mesh, hypercubique, en anneau, en étoile. Concernant la topologie, une topologie stochastique et une topologie personnalisée, avec chargement via un fichier texte sont également proposées.

La conception a été faite de sorte que tous les éléments constitutifs du modèle puissent être étendus et personnalisés par simple héritage.

Autour du livrable, des tests de couverture concluant ont été réalisés, ainsi que des tests unitaires et d'intégration permettant d'assurer la non-régression future. Une batterie de tests a également validé les apports théoriques du modèle en îles tel qu'il est implémenté.

Enfin, une série de tutoriels a été réalisée pour guider les utilisateurs dans la création des modèles d'îles, à la fois homogène mais également hétérogène ainsi que dans l'utilisation de mécanismes avancés. La documentation technique rédigée permettra la maintenance et l'évolution du modèle au sein du module SMP.

### 5.2 Améliorations

Un certain nombre d'améliorations peuvent être envisagées. Dans un premier temps, l'objectif d'hybridation n'a pas été atteint. Il semble assez facile de l'intégrer au module SMP, mais par faute de temps, nous avons préféré nous concentrer sur les tests pour valider les bases du modèle en îles.

Certains travaux indiquent qu'il est intéressant d'avoir des migrations avec des individus envoyés avec une sémantique d'entité (c'est à dire pas par copie, les individus quittent leur île). Couplé à une bonne politique (voir une politique dynamique) cela permet de vider les îles inutiles de leur population et ainsi de repérer naturellement les îles importantes ou non du modèle. C'est une idée d'amélioration qui n'est pour le moment pas réalisable à cause du design de EO qui est ne permet pas une variation de la taille de la population d'un algorithme.

D'un point de vue implémentation, quelques petites améliorations pourraient être envisagées. Notamment la réalisation d'un conteneur STL-compliant pour la Bimap ou l'utilisation des `std::cond_var` au lieu d'un `nano sleep` pour la gestion du modèle. En effet, la durée du `nano sleep` est choisi arbitrairement de manière à limiter l'utilisation CPU de la boucle



principale du modèle. A contrario le `std::cond_var` va permettre d'attendre explicitement la notification d'une île indiquant qu'elle a déposé de la population à faire migrer dans la queue de travail du modèle.

## 5.3 Utilisation

Cette section est une traduction partielle et une adaptation des tutoriels rédigés pour le modèle en îles et disponibles sur le site de ParadisEO.

### 5.3.1 Modèle homogène

Le modèle homogène est le plus simple à mettre en place.

Pour cela il faut définir les parties communes à toutes les îles (les opérateurs de mutation, de croisements, notamment).

```

1 #include <smp>
2
3 //Common part to all islands
4 IndiEvalFunc plainEval;
5 IndiInit chromInit;
6 eoDetTournamentSelect<Indi> selectOne(param.tSize);
7 eoSelectPerc<Indi> select(selectOne); // by default rate==1
8 IndiXover Xover; // CROSSOVER
9 IndiSwapMutation mutationSwap; // MUTATION
10 eoSGATransform<Indi> transform(Xover, param.pCross, mutationSwap, param.pMut);
11 eoPlusReplacement<Indi> replace;
```

Listing 5.1– Parties communes à toutes les îles.

Ensuite nous pouvons créer la topologie ainsi que le modèle qui accueillera nos îles. Le modèle est initialisé avec la topologie :

```

1 // MODEL
2 Topology<Complete> topo;
3 IslandModel<Indi> model(topo);
```

Listing 5.2– Initialisation de la topologie et du modèle

On notera que le template de l'objet `Topology` représente son type d'organisation, ici en graphe complet. De même, le template du modèle représente le type d'individu sur lesquels les îles vont travailler.

Il faut ensuite créer ses îles. Dans un premier temps, il faut créer les parties spécifiques à chaque algorithme : critères de continuation, et la population. Enfin, il faut définir la politique de migration et la politique d'intégration :

```

1 // ISLAND 1
2 // // Algorithm part
3 eoGenContinue<Indi> genCont(param.maxGen+100);
4 eoPop<Indi> pop(param.popSize, chromInit);
5 // // Emigration policy
6 // // // Element 1
7 eoPeriodicContinue<Indi> criteria(5);
8 eoDetTournamentSelect<Indi> selectOne(20);
9 eoSelectNumber<Indi> who(selectOne, 3);
10
11 MigPolicy<Indi> migPolicy;
12 migPolicy.push_back(PolicyElement<Indi>(who, criteria));
13
14 // // Integration policy
15 eoPlusReplacement<Indi> intPolicy;
16
17 Island<eoEasyEA,Indi> island1(pop, intPolicy, migPolicy, genCont, plainEval, select,
    transform, replace);
```

Listing 5.3– Création d’une île.

Enfin, il suffit d’ajouter les îles au modèle et de lancer le modèle :

```
1 model.add(island1);
2 model.add(island2);
3 ...
4 model.add(islandn);
5
6 model();
```

Listing 5.4– Ajout des îles et lancement du modèle.

### 5.3.2 Modèle hétérogène

La création d’un modèle hétérogène, c’est à dire fonctionnant avec des algorithmes de types différents (comme un PSO et un SGA par exemple) requiert la notion d’individu de base.

Ces individus de bases sont les individus connus par le modèle sont les seuls qui vont pouvoir être manipuler lors des communications.

Généralement il s’agira du type qui est le plus manipulé au sein des îles mais il pourra s’agir du type solution du problème à résoudre, même si la majorité des îles manipulent d’autres type d’individus.

La seule chose que cela implique par rapport au modèle homogène est qu’il faut définir un moyen de convertir un individu de son type vers le type de base et du type de base vers les autres types manipulés par le modèle. Cela peut être aussi bien une fonction libre qu’une méthode de classe dont une instance précise va exécuter la conversion, ou bien une lambda fonction.

Dans l’exemple suivant nous avons une population de d’individu de base de type Indi et une population secondaire de type Indi2.

Dans un premier temps nous définissons nos fonctions de conversions :

```
1 // Conversion functions
2 Indi2 fromBase(Indi& i, unsigned size)
3 {}
4
5 Indi toBase(Indi2& i)
6 {}
```

Listing 5.5– Fonctions de conversions.

Le prototype attendu par l’île est le suivant :

```
Indi fromBase(Indi2&)
```

Idem pour la conversion inverse. Cependant, il est tout à fait possible, comme c’est le cas dans notre exemple, d’avoir plus de paramètres dans ses fonctions grâce aux mécanismes de `std::bind`.

L’étape suivant est d’encapsuler les fonctions de conversions dans des `std::function` qui pourront être passer à l’île lors de sa création :

```
1 auto frombase = std::bind(fromBase, std::placeholders::_1, VEC_SIZE);
2 auto tobase = std::bind(toBase, std::placeholders::_1);
```

Listing 5.6– Création des `std::function`.

L’inférence de type grâce au mot clef `auto` permet de ne pas se soucier de la syntaxe du `std::function`. L’argument supplémentaire, à savoir la taille, de notre première fonction a été remplacé par la valeur effective pour notre instance (la constante `VEC_SIZE`).

A contrario, pour respecter le prototype attendu par l’île, le premier argument a été laissé libre grâce à l’objet

std::placeholders::\_1.

La seule chose restante est de déclarer, comme nous l'avons vu pour le modèle homogène, les îles puis le modèle.

```
1 Island<eoSGA,Indi2,Indi> gga(frombase, tobase, pop, intPolicy, migPolicy, select, xover,
    CROSS_RATE, mutation, MUT_RATE, eval, continuator);
2 Island<eoEasyEA,Indi> ea(pop2, intPolicy2, migPolicy2, genCont, plainEval, select2,
    transform, replace);
3
4 IslandModel<Indi> model(topo);
5 model.add(ea);
6 model.add(gga);
7
8 model();
```

Listing 5.7– Fonctions de conversions.

La petite différence avec le modèle homogène est que l'île a un troisième template qui est celui de l'individu de base. Dans le cas d'une île dont le type manipulé est le même que le type de base du modèle, le troisième template est inutile puisque initialisé par défaut sur ce type de base. C'était le cas implicitement avec le modèle homogène. Enfin, notons qu'on peut évidemment avoir plus de deux types différents au sein d'un même modèle. Il suffit simplement de créer les fonctions de conversions nécessaires.

### 5.3.3 Mécanismes avancés

#### Topologies classiques, personnalisées et stochastiques

Afin de maîtriser les topologies, on doit garder à l'esprit que les communications ne vont pas dans les deux sens. C'est-à-dire qu'une communication de l'île A à l'île B n'implique pas une communication dans l'autre sens.

##### Topologies classiques

Voici une brève description des topologies classiques.

- La topologie complète est la plus simple. Chaque noeud est aux autres, excepté lui même. Pour ce cas, les communications se font dans les deux sens.
- Voyons la topologie en anneau. Chaque noeud est connecté à un unique autre noeud, qui est son successeur dans la liste de noeuds. Le dernier noeud est relié au premier.
- Maintenant, la topologie en étoile est un peu plus complexe. Le centre de l'étoile communique avec tous les autres noeuds, mais ne reçoit aucune communication. Par défaut, le noeud central est le premier, mais on peut le changer à n'importe quel moment, au même titre que l'on peut changer de topologies :

```
1 Topology<Star> topo_star;
2 /*
3  Building island model, and you want to change the center :
4  */
5 topo_star.getBuilder().setCenter(2); //This has for effect to set the center of the
    star to the third node.
```

Listing 5.8– Réalisation d'un OU logique.

- Maintenant, la topologie maillée. Cette topologie représente une grille rectangulaire, et à chaque noeud se trouve une île. Donc, chaque noeud est connecté à 1, 2, 3 ou 4 autres noeuds. Les communications se font dans les deux sens. Par défaut, la grille approche au maximum la forme d'un carré. Cependant, comme pour la topologie en étoile, il est possible d'influencer la forme de la grille via un paramètre appelé ratio. Par exemple, pour 18 noeuds :
  - 1\*18 ratio = 1/18
  - 2\*9 ratio = 2/9
  - 3\*6 ratio = 1/2

Par défaut, la grille aura une forme de 3 par 6, mais si le ratio est changé à 0, la grille aura une forme de ligne (1 par 18). Si le ratio est mis à 1/3, la forme choisie sera 2 par 9, car son ratio est le plus proche.

```
1 Topology<Mesh> topo_mesh;
2 /*
3  Building island model, and you want to change the shape :
4  */
5 topo_mesh.getBuilder().setRatio(0); //This has for effect to set shape of the grid
   to a line.
```

Listing 5.9– Changement du ratio d’une topologie maillée.

- Finalement, vous pouvez utiliser une topologie hypercubique. La seule contrainte est la taille : le nombre d’îles doit être une puissance de 2. Les communications vont dans les deux sens. Si vous voulez organiser vos îles pour contrôler lesquelles sont connectées, vous devez savoir que les indices de deux îles connectées ont le même codage binaire, excepté pour un chiffre. Par exemple, les îles 5(1010) et 7(1110) sont connectées.

#### Utiliser des topologies personnalisées.

Il y a deux types de topologies personnalisées : booléenne et stochastique. Les deux sont construites à partir d’un fichier contenant la matrice. Le fichier va être parsé, chaque ligne doit contenir le même nombre de valeurs, qui est aussi le nombre de lignes (matrice carrée).

La CustomBooleanTopology contient des entiers (normalement 0 ou 1). La valeur à la ligne  $i$  et colonne  $j$  est la probabilité d’avoir une communication entre l’île  $i$  et l’île  $j$ . Par conséquent, les valeurs négatives sont considérées nulles, et les valeurs au dessus de 1 sont considérées comme étant 1.

Exemple pour la topologie booléenne :

```
file : data_boolean
0 1 0
1 0 1
0 1 0
```

```
1 CustomBooleanTopology topo_bool("data_boolean");
2 //getting the neighbors of island 1:
3 std::vector<unsigned> neighbors = topo_bool.getIdNeighbors(1); //return a vector
   containing 0 and 2
```

Listing 5.10– Chargement d’une topologie personnalisée booléenne.

Exemple pour la topologie stochastique :

```
file : data_stochastic
0 1 0
1 0 .75
0 1 0
```

```
1 CustomBooleanTopology topo_stoch("data_stochastic");
2 //getting the neighbors of island 1:
3 std::vector<unsigned> neighbors = topo_bool.getIdNeighbors(1); //return a vector
   containing 0, and the probability 0.75 to contain the value 2.
```

Listing 5.11– Chargement d’une topologie stochastique.

## Gestion de politique avancée

Bien que formellement, un élément de politique se matérialise par un critère quand et un critère qui, il est possible de réaliser des conditions logiques OU et ET grâce à des mécanismes déjà présents dans Evolving Objects.

Ainsi pour réaliser une condition OU, on peut utiliser un `eoCheckPoint` comme critère, qui est un conteneur de `eoContinue` (et renverra vrai si au moins un des `eoContinue` est vrai). Par exemple :

```
1 eoSelect<EOT> who;
2 eoPeriodicGenContinue<EOT> criteria_1(50);
3 eoFitContinue<EOT> criteria_2(5000);
4 eoCheckPoint<EOT> criterion(criteria_1);
5 criterion.add(criteria_2);
6
7 PolicyElement<EOT> rule_1(criteria, who);
```

Listing 5.12– Réalisation d'un OU logique.

La migration sera effectuée si l'algorithme effectue une génération 0 modulo 50 ou si un individu au moins à une fitness de 5000. Il peut évidemment y avoir plus que deux critères.

Pour créer une condition ET on peut utiliser un `eoCombinedContinue` qui va retourner vrai uniquement si tous les `eoContinue` contenus dedans retournent vrai. Par exemple :

```
1 eoSelect<EOT> who;
2 eoPeriodicGenContinue<EOT> criteria_1(50);
3 eoFitContinue<EOT> criteria_2(5000);
4 eoCombinedContinue<EOT> criterion(criteria_1);
5 criterion.add(criteria_2);
6
7 PolicyElement<EOT> rule_1(criteria, who);
```

Listing 5.13– Réalisation d'un ET logique.

Un migration sera effectuée uniquement à partir du moment où un individu au moins à une fitness d'au moins 5000 et si la génération courante est 0 modulo 50.

Ceci est particulièrement utile lorsque l'on veut créer des politiques dynamiques qui vont pouvoir évoluer entre diversification et intensification. On peut par exemple imaginer une politique très laxiste sur la sélection d'individu pendant les 30 premières minutes et qui va devenir plus élitiste par la suite :

```
1 First, we need to define the diversification step :
2 eoSelect<EOT> who; // Selection without condition on the fitness, for instance
3 eoPeriodicGenContinue<EOT> criteria_1(50);
4 eoSecondsElapsedContinue<EOT> criteria_2(60*30);
5 eoCombinedContinue<EOT> criterion(criteria_1);
6 criterion.add(criteria_2);
7
8 PolicyElement<EOT> diversification_step(criteria, who);
```

Listing 5.14– Diversification.

Pour l'étape d'intensification nous devons utiliser un nouveau `eoContinue` créée pour l'occasion : `eoInvertedContinue` qui est un foncteur renvoyant l'opposé du `eoContinue` qu'il contient.

```
1 eoSelect<EOT> who; // Elitist selection
2 eoPeriodicGenContinue<EOT> criteria_1(50);
3 eoInvertedContinue<EOT> criteria_2((eoSecondsElapsedContinue<EOT>(60*30)));
4 eoCombinedContinue<EOT> criterion(criteria_1);
5 criterion.add(criteria_2);
6
7 PolicyElement<EOT> intensification_step(criteria, who);
```

Listing 5.15– Intensification.

### Création d'évènements par callback

Comme évoqué dans la présentation du modèle en îles, les recherches actuelles s'intéressent à des évolutions dynamiques des îles et du modèle. Nous avons vu qu'il était possible d'avoir une politique flexible dans le paragraphe précédent.

Ici nous présentons un système permettant d'effectuer n'importe quelle tâche par une île. Ceci implique donc une modification éventuelle de sa politique, mais également une modification de la topologie, ou tout autre chose (récupération de statistiques intermédiaires par exemple).

Pour cela, il faut utiliser la classe Notifier qui dérive de eoUpdater. Il s'agit d'un foncteur qui va prendre comme paramètre à la construction, une `std::function` (là encore il peut s'agir de lambdas, de fonctions libres ou de méthodes de classes exécutées par une instance précise). Ce foncteur s'ajoute à la liste des eoContinue d'une île et va être exécuté chaque génération, exécutant lui même la tâche ainsi bindée. De fait, lors de l'écriture de la tâche à effectuer, il faut tenir compte des critères d'exécution de cette tâche (sinon elle est effectuée chaque génération ce qui n'est sûrement pas l'objectif).

Voici un exemple permettant de changer la topologie après 10 minutes :

```

1 First, we need to write a function to change the topology after 10 minutes :
2 void changeTopo(IslandModel<Indi>* _model, AbstractTopology& _topo)
3 {
4     static auto start = std::chrono::system_clock::now();
5     auto now = std::chrono::system_clock::now();
6     auto elapsed = now - start;
7     static bool first = false; // We would like to change only one time !
8     if(std::chrono::duration_cast<std::chrono::minutes>(elapsed).count() > 10 && !first)
9     {
10         _model->setTopology(_topo);
11         first = true;
12     }
13 }
```

Listing 5.16– Fonction de changement de topologie après 10 minutes.

Il reste à binder notre fonction avec les arguments voulus et l'inclure dans l'île :

```

1 Topology<Ring> topo2; // New topology
2 std::function<void(void)> task = std::bind(changeTopo, &model, topo2); // We bind our
   function with our new topology
3 Notifier topoChanger(task); // We create the notifier
4 checkPoint.add(topoChanger); // We add the notifier, assuming that checkPoint is the
   eoCheckPoint used by an island as continuators
5
6 Island<eoEasyEA,Indi> test(pop, intPolicy, migPolicy, checkPoint, plainEval, select,
   transform, replace);
```

Listing 5.17– Ajout d'un notifier dans une île.

# Chapitre 6

## Tests

### 6.1 Tests

Les tests unitaires ont porté principalement sur l'île et la topologie alors que les tests d'intégration se chargeaient de tester un modèle homogène, un modèle hétérogène et le wrapper permettant de créer facilement un modèle homogène.

Étant donné la simplicité de politique de migration et d'intégration il a été décidé de ne pas les tester directement.

### 6.2 Tests unitaires

De par sa création, l'île peut être lancée indépendamment d'un modèle d'île. Cela ne présente aucun intérêt si ce n'est celui de permettre de réaliser des tests unitaires plus facilement.

Ainsi, le test effectué consiste à vérifier que la qualité de la meilleure solution obtenue par l'île est la même que celle de l'algorithme lancé séquentiellement et non wrappé dans une île (avec une graine fixée pour le générateur aléatoire). Cela a permis de mettre en avant un problème qui ne peut pas, dans l'état actuel du framework, être résolu. En effet, une fois la graine fixée, le générateur se transforme en simple suite numérique. Lorsque plusieurs îles sont lancées en parallèle, elles vont accéder au même générateur aléatoire et donc à la même série. Le problème est double. D'une part, comme les îles vont accéder à la même série, les processus qui dépendent des nombres à récupérer vont changer, ce qui implique que les algorithmes ne vont pas donner le même résultat que leur homologue sérialisé.

De plus, selon la charge du CPU, une île va mettre plus ou moins de temps à accéder à nouveau à la série (par exemple l'évaluation a mis plus de temps à une certaine génération qu'à la précédente), permettant potentiellement à une autre île d'accéder à la valeur destinée à la première. Ainsi, le résultat, même avec une graine fixée, n'est plus déterministe.

La solution serait d'avoir un générateur aléatoire par algorithme, mais ce n'est malheureusement pas possible avec le design actuel de EO.

Nous avons cependant effectué des tests qualitatifs sur les résultats afin de s'assurer que les résultats obtenus ne sont pas aberrants.

Le protocole consistait à lancer 150 fois 2 îles identiques, en parallèles sur une instance du TSP à 535 villes, avec une population de 1000 individus par îles et à comparer les résultats avec une île sérialisée tout en essayant de trouver une structure dans les résultats obtenus.

Seul le meilleur individu a été gardé. Nous obtenons une moyenne de -35343 pour l'algorithme sérialisé contre -35021.94 pour les îles en parallèle. La différence semble minime, de l'ordre de 1%. Même si les tests effectués donnent en moyenne un meilleur résultat en moyenne, cela est trop dépendant du matériel pour que l'on puisse en conclure sur un apport bénéfique de la présence d'un unique générateur aléatoire.

Voici les résultats obtenus :

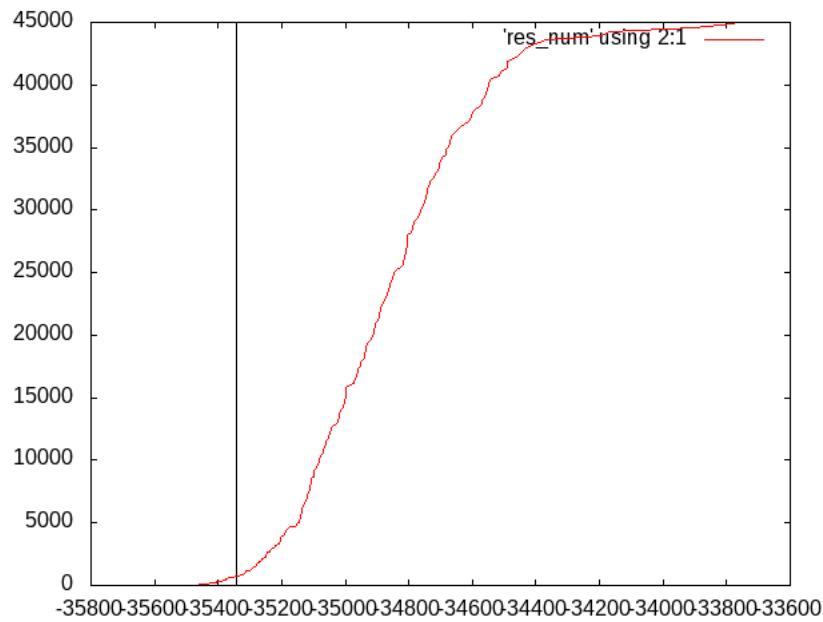


FIGURE 6.1 – Fréquence cumulée des individus

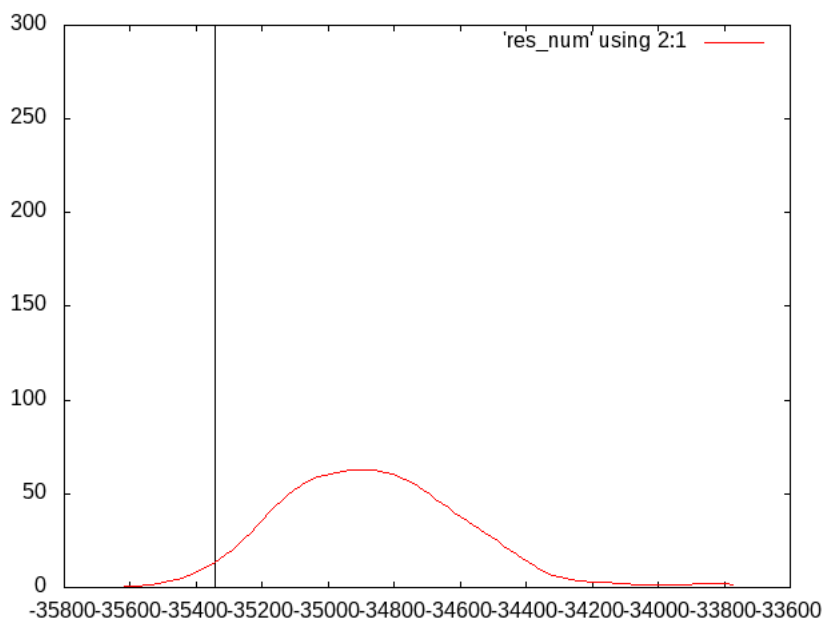


FIGURE 6.2 – Estimation par noyau de la distribution

### 6.3 Tests de couverture

Les tests de couverture permettent de quantifier la proportion de code testé et de détecter les éventuelles parties de code mort, c'est à dire les parties de code qui ne sont plus utilisées nul part. Ces tests sont particulièrement importants pour garantir la qualité d'un code.

Les tests de couverture obtenus sur le module SMP, incluant également le pattern Maître / Esclaves sont très satisfaisants puisqu'ils sont proches de 100%.



	Hit	Total	Coverage
<b>Lines:</b>	325	327	99.4 %
<b>Functions:</b>	167	177	94.4 %
<b>Branches:</b>	394	658	59.9 %

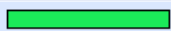

	Line Coverage ↕		Functions ↕		Branches ↕	
<a href="#">src</a>		99.5 %	220 / 221	95.7 %	135 / 141	57.0 %
<a href="#">src/topology</a>		99.1 %	105 / 106	88.9 %	32 / 36	70.4 %

FIGURE 6.3 – Résultats des tests de couverture du module SMP.

## 6.4 Tests de performances

Les tests de performances ont un double rôle à jouer ici puisqu'ils doivent quantifier à la fois la qualité de la programmation parallèle et vérifier les apports théoriques du modèle en île dans le but de valider l'implémentation.

Les tests de performance ont été effectués sur un problème du voyageur de commerce, avec des instances de différentes tailles (approximativement 100 villes à 13 000 villes). Il s'agit d'instances réelles pour mettre en situation le modèle en îles réalisés.

La première série de tests est purement descriptive puisqu'elle ne fait que donner des indicateurs sur des instances déterministes, c'est à dire avec une graine fixée (modulo le fait qu'il n'y ait qu'un générateur aléatoire). La seconde partie des tests est plus intéressante parce qu'elle donne une réponse statistique quand à l'influence de certains facteurs.

### 6.4.1 Modèle homogène vs Sérialisé

#### Test 1

##### Objectifs :

Mettre en évidence une rapidité de convergence accrue du modèle.

##### Protocole :

Graine fixée

Population de 1000 individus

Modèles d'îles homogènes

Politique élitiste :

1. eoPeriodicContinue;Route; criteria\_2(25);
2. eoDetTournamentSelect;Route; selectOne\_2(100);
3. eoSelectNumber;Route; who\_2(selectOne\_2, 100);

##### Résultats : Feuille 1 du tableur

##### Analyse des résultats :

Dans le cas où la graine est fixée la variance de l'échantillon diminue vraiment, tant en passant de l'algorithme sérialisé à 3 îles qu'en passant de 3 à 4 îles. Cependant, en ne considérant que les 1000 meilleurs individus, le passage de 3 à 4 îles ne semblent plus significatifs.

L'algorithme semble avoir convergé, aussi bien pour 3 et pour 4 alors qu'il n'a vraisemblablement pas convergé pour l'algorithme seul.

On peut également se demander si la politique très élitiste mise en place n'a pas fait converger les algorithmes vers un minimum local, ainsi si l'algorithme converge à partir de 3 îles et pour les critères d'arrêts spécifiés, alors il convergera vers le même optimum pour un nombre d'îles supérieur (sauf si par chance ces îles supplémentaires sont générés avec des individus meilleurs à la base).

Les autres métriques sont unanimes : la SCR est divisé par 10 à 100 voire par un facteur  $10^6$  sur des popu-

lations constantes par exemple, montrant fitness bien meilleurs sur le modèle d'îles.

Le graphique nous montre clairement que les modèles d'îles ont convergés alors que la fitness des individus de l'algorithme seul semble continuer de s'améliorer (cf. dernier tiers).

Le test s'avère ici plus qualitatif que quantitatif car il ne répond pas à la question suivante : quelle est le gain sur la vitesse de convergence.

Accessoirement on peut également se demander si l'algorithme sérialisé aurait convergé vers un meilleur optimum ou non, et ce, malgré une politique très élitiste.

## Test 2

### Objectifs :

Quantifier le gain sur la rapidité de convergence du modèle d'île.

### Protocole :

Même protocole que ci-dessus. On changera les critères des continuators pour un `eoFitContinue` que l'on fixera à -1350 qui correspond à peu près à la solution obtenue pour les modèles d'îles.

On ajoutera un `eoGenCounter` pour compter le nombre de générations avant d'atteindre cette fitness.

### Résultats :

Sérialisé : 3361 générations

3 îles : 1261 générations\*

4 îles : 968 générations\*

\* Obtenus comme moyenne sur 100 tests (non déterministe malgré la graine, à cause de l'unique générateur aléatoire)

Notons également que la valeur maximale pour 4 îles est atteinte pour 1401, soit plus que la moyenne pour 3 îles (l'influence de l'unique générateur peut donc être handicapante). A l'instar le meilleur résultat est obtenu pour 904.

### Analyse des résultats :

L'algorithme s'arrête au moment où l'algorithme est prêt à converger (première apparition de la meilleure fitness observée). Le résultat montre bien que les modèles d'îles convergent plus rapidement.

On obtient un ratio de 2,6 pour le modèle à 3 îles et un ratio de 3,47 pour le modèle à 4 îles.

## 6.4.2 Influence du facteur communication

### Test déterministe

#### Objectifs :

Mettre en avant l'intérêt des communications dans la convergence de l'algorithme et la qualité de solutions.

#### Protocole :

Graine fixée.

3 îles sans communication, 3 îles avec communications (topologie complète et politique élitiste).

Partant d'une même population, on lance les deux modèles pour 1000 générations par île et on regarde la qualité finale obtenue sur l'ensemble de la population.

#### Résultats : Feuille 2 du tableur

### Analyse des résultats :

Tous les indicateurs sont favorables au modèle avec communication. On note une amélioration moyenne de 30% des solutions et une somme des carrés des résidus qui a diminué d'un facteur 70.

Le graphique montre que le modèle sans communication n'a pas encore convergé alors que celui avec communication est en passe d'avoir complètement convergé.

### Test non déterministe : ANOVA 1

Dans le cas général où la graine n'est pas fixée, nous voulons déterminer si les communications ont un facteur sur la qualité de la solution obtenue. Pour cela nous allons essayer de nous servir de l'ANOVA 1 : l'analyse de variance à un facteur.

Nous avons ici un facteur, les communications, avec deux modes : avec ou sans communication.

Nos observations seront notées  $y_{ij}$  avec  $i \in \{1, 2\}$  respectivement sans et avec communication, et  $j \in \{1 \dots 100\}$  puisque nous avons réalisé 100 expériences avec et sans communication. Le modèle d'ANOVA 1 se base sur des hypothèses qui ne sont pas triviales dans notre cas :

- Les données  $y_{ij}$  obtenus sont réalisations d'une variable aléatoire  $Y_{ij}$  de loi normale  $N(\mu_i, \sigma^2)$
- Les variables aléatoires ( $Y_{ij}$ ) sont globalement indépendantes.

Notre modèle serait donc :

$$Y_{ij} = \mu_i + \epsilon_{ij}$$

Avec  $\epsilon_{ij}$  indépendants et identiquement distribués de loi normale  $N(0, \sigma^2)$ .

Les hypothèses du modèle n'étant pas trivialement vérifiées, il nous faudra dans un premier temps les vérifier avant de réaliser l'analyse de la variance à proprement parler.

Pour cela nous allons adopter le plan qui suit :

- Une estimation par noyau nous dira si la fitness des meilleurs individus suit une loi normale.
- Un test de Barlett nous dira si il y a homoscedasticité (condition du modèle).
- Nous dresserons alors un tableau d'analyse de variance.
- Un test de comparaison par approche de modèle nous donnera une réponse au risque de 5% quand à l'influence du facteur communication sur la différence des moyennes des deux groupes.

Les tests de normalité nous permettent de conclure à la normalité des deux échantillons. Les tests effectués sont le test de Lilliefors et le test de Shapiro-Wilk. Ils ont été réalisés sous R grâce au paquet normtest.

```
> lillie.test(t(X1))
```

Lilliefors (Kolmogorov-Smirnov) normality test

```
data: t(X1)
D = 0.0599, p-value = 0.5072
```

```
> shapiro.test(t(X1))
```

Shapiro-Wilk normality test

```
data: t(X1)
W = 0.9859, p-value = 0.3694
```

```
> lillie.test(t(X2))
```

Lilliefors (Kolmogorov-Smirnov) normality test

```
data: t(X2)
D = 0.0547, p-value = 0.6533
```

```
> shapiro.test(t(X2))
```

Shapiro-Wilk normality test

```
data: t(X2)
W = 0.992, p-value = 0.8184
```

Le seuil fixé est de 5%. L'hypothèse du test est que les individus suivent une loi normale. La p-value étant supérieure au seuil, on ne rejette pas l'hypothèse et on conclue donc à la normalité de l'échantillon.

```
fligner.test(list(t(X1),t(X2)))
```

Fligner-Killeen test of homogeneity of variances

```
data: list(t(X1), t(X2))
```

```
Fligner-Killeen:med chi-squared = 15.5555, df = 1, p-value = 8.012e-05
```

Le test de variance indique une variance non-commune aux deux groupes. Cependant, cette variance non homogène est moins problématique qu'une distribution non normale des individus, surtout lorsque les groupes comportent autant d'individus. Ainsi, le test de Fisher effectué lors de l'ANOVA sera potentiellement faussé en rejetant plus facilement l'influence significative du facteur étudié.

Analyse de variance: un facteur

#### RAPPORT DÉTAILLÉ

<i>Groupes</i>	<i>Nb d'échantillons</i>	<i>Somme</i>	<i>Moyenne</i>	<i>Variance</i>
Colonne 1	100	-143361	-1433,61	4378,09889
Colonne 2	100	-173363	-1733,63	10445,0435

#### ANALYSE DE VARIANCE

<i>Source</i>	<i>Somme Carrés</i>	<i>DDL</i>	<i>Moy. Carrés</i>	<i>F</i>	<i>Probabilité</i>
Entre Groupes	4500600,02	1	4500600,02	607,239665	3,1397818E-062
Residuelle	1467491,1	198	7411,57121212		
Total	5968091,12	199			

# Chapitre 7

## Conclusion

### 7.1 Conclusion

Ce projet en partenariat avec Inria a été pour nous l’occasion d’avoir un contact privilégié avec un laboratoire de recherche et d’être dans des conditions plus proches des conditions d’un projet non-académique. Ce fut l’occasion d’aborder de nombreux domaines tels que l’optimisation en règle générale, mais aussi plus spécifiquement les algorithmes génétiques, le parallélisme et le génie logiciel. Durant ce projet nous avons pu mettre en pratique un grand nombre de compétences acquises durant le cursus GM, notamment la partie modélisation, mais également le partie développement. Ce fut l’occasion d’approfondir l’ensemble de ces connaissances et de les renforcer avec des notions indispensables en entreprise : bonnes pratiques, utilisation de gestionnaires de version, politique de tests unitaires, d’intégration, de couverture, etc. mais aussi de manière plus générale la méthodologie, la communication et le reporting régulier auprès des encadrants.

De manière plus étonnante, la batterie de tests réalisés sur le livrable a été l’occasion de mettre en pratique les cours de statistiques au travers de GnuPlot mais également R et Matlab tandis que l’initialisation des topologies relevait de la théorie des graphes.

La rédaction des tutoriaux et des tests nous a obligé à réellement comprendre les mécanismes des algorithmes génétiques et de diverses techniques d’optimisation combinatoire.

Enfin, le livrable est de notre point de vue tout à fait satisfaisant. Les tests ont montré l’efficacité de l’implémentation et validé les apports théoriques d’un tel modèle. L’évolutivité du modèle semble tout à fait correcte et la documentation livrée permettra une utilisation facilitée.

Pour conclure, ce projet a été enrichissant en tout point. D’une part pour la connaissance des attentes d’un laboratoire assimilable ici à un client et de la gestion de projet, d’autre part pour la connaissance théorique d’un certain nombre de domaines : génie logiciel, parallélisme, algorithmes génétiques et optimisation combinatoire. Enfin, les apports techniques ne sont pas non plus négligeables : C++11, métaprogrammation, R, utilisation de ParadisEO, CMake, Git, etc.

# Bibliographie

- [1] El-Ghazali Talbi, *Metaheuristics : From Design to Implementation*.  
ISBN : 978-0-470-27858-1
- [2] Raphaël Cerf, *Une théorie asymptotique des algorithmes génétiques*.  
[http ://www.math.u-psud.fr/~cerf/](http://www.math.u-psud.fr/~cerf/)
- [3] Anthony Williams, *C++ Concurrency : Practical Multithreading*.  
ISBN-13 : 978-1933988771
- [4] Arnaud Liefoghe, *Cours d'optimisation - LIFL / Lille 1*.

## Chapitre 8

# Annexes

### 8.1 Diagramme de classes



On peut vérifier que Algo<EOT> est un algorithme à base de population par type\_traits

