Génie logiciel sur un framework de meta-heuristiques
Rapport de Stage

Alexandre QUEMY
Département Génie Mathématique de l'INSA de Rouen
Stage réalisé du 25 Juin au 31 août 2012

Inria Lille
40 avenue Halley, Bât. A, Park Plaza
59650 Villeneuve d'Ascq

INSA de Rouen
Avenue de l'université
76 000 Rouen

Remerciements

J'aimerais d'abord remercier mon maître de stage Clive Canape pour ses nombreux conseils et sa grande disponibilité tout au long de ces dix semaines de stage.

Je remercie également l'ensemble de l'equipe DOLPHIN qui m'a très bien accueilli et a eu la patience de répondre à toutes mes questions.

Merci à Nadia d'avoir partagé son bureau avec moi malgré le bazar que je pouvais y mettre. Merci également à Mathieu et Mattieu pour leurs avis et leur patience vis à vis de mes questions. Merci à Yacine pour sa bonne humeur et nos discussions musicales.

Enfin, je voudrais remercier les anciens ou actuels GM qui se trouvaient dans le coin pour leurs avis éclairés sur l'INSA, l'Inria, la recherche en générale, le monde profesionnel, etc : Clive, Sophie, Manon et Mattieu.

Table des matières

Chapitre 1

Introduction

ParadisEO est un framework open-source développé en C++ et dédié au développement de méthodes d'optimisation approchées (métaheuristiques stochastiques) classiques, multi-objectifs, parallèles et hybrides.

Le framework repose lui-même sur le framework **EO** pour **Evolving Objects** qui fournit, entre autres, des templates pour la création d'algorithmes évolutionnistes.

Développé par l'équipe **DOLPHIN**, le framework souffre d'un système d'installation vieillissant rendant **ParadisEO** très difficilement instalable, voire impossible à installer. La volonté de redynamiser le projet et simplifier l'installation et la distribution du logiciel appelait donc à une restructuration importante du logiciel et de la ligne directrice à adopter en terme d'intégration continue, de support et de contributions ainsi que le passage à des technologies plus récentes (ou une mise à niveau).
Notons également que la stabilisation d'une version de **ParadisEO** s'inscrit dans perspective de la fusion à court terme avec le framework **Evolving Objects**, actuellement développé par une équipe extérieure à **Inria**, travaillant pour **Thales**. **Evolving Objects** a été développé à la base par **Inria**, plus particulièrement par l'équipe **TAO** et Marc Schoenauer.

Le second point important est la parallélisation, fer de lance de **ParadisEO**, qui souffre également d'un vieillissement à cause de sa dépendance à un ancien module, et parce qu'il repose sur des technologies ayant quelque peu évoluées. Il s'agissait de couper les dépendances à cet ancien module, refactoriser certaines parties du code, et adopter un nouveau design, séparant parallélisation distribuée et parallélisation partagée.

Le stage s'est donc déroulé en deux parties, entrecoupées d'une réunion importante avec l'équipe de **Thales** afin de préparer la fusion et qui a permis d'affiner et de corriger les objectifs de la seconde partie.

La formation d'ingénieur débouche majoritairement dans le milieu de l'entreprise, mais aussi dans le domaine de la recherche. Ce dernier m'intéresse particulièrement et ayant pu travailler à la fois au sein d'une grosse industrie lors de mon stage ouvrier et dans une société de services, ce stage est l'occasion de découvrir un nouveau secteur et de confirmer et affiner mon projet professionnel.
C'est de plus, l'opportunité parfaite pour découvrir et approfondir des domaines qui m'intéressent particulièrement, à savoir l'optimisation combinatoire et les algorithmes évolutionnistes en général, ainsi que la programmation parallèle et distribuée.

Chapitre 2

Contexte du stage

2.1 L'Inria, laboratoire de recherche

2.1.1 Présentation et organisation

L'**Inria**, (anciennement connu sous l'acronyme **INRIA** pour Institut National de Recherche en Informatique et Automatique) est un établissement à caractère scientifique crée en 1967, suite au **Plan Calcul** [**?**] initié par le général De Gaulle sur l'impulsion de Michel Debré et d'un groupe de hauts fonctionnaires et d'industriels, destiné à assurer l'indépendance du pays en matière d'informatique et plus particulièrement de **HPC** (High Performance Computing).

Sous tutelle des ministères de la Recherche et de l'Industrie, **Inria** mène une double activité : le développement d'une recherche de pointe et la création de valeur, grâce au transfert des résultats de ses travaux vers le monde économique et la société.

La recherche s'axe autour d'équipes-projet. Il s'agit d'une équipe de taille limitée ayant des objectifs scientifiques et un programme de recherche clairement établi. Elle bénéficie d'une autonomie financière par un budget distribué par l'institut et provenant de contrats de recherche. Ces équipes sont formées après évaluation du projet par une commission et sont révisées tous les 4 ans par des experts extérieurs à l'équipe. Une équipe projet dure en moyenne **7** ans et au maximum **12** ans.

Table 2.1 – Quelques chiffres

4 290 personnes
252 millions de budget en 2010
730 stagiaires de fin d'étude chaque année
800 contrats de recherche en cours
70 équipes associées
105 start-up créées

2.1.2 La place de l'Inria en France et à l'étranger

L'**Inria** représente un des principaux laboratoires de recherches en informatique et mathématiques de France, avec 8 centres sur tout le territoire.
En outre, l'**Inria** bénéficie d'une excellente intégration à la fois dans le domaine académique

par une grande proximité avec les universités (notamment Paris Diderot, et Paris Pierre et Marie Curie, en tout plus de 45 universités et grandes écoles) et à la fois avec l'entreprise puisqu'on dénote de nombreux partenariats avec de grands industriels comme Total, EDF, Alcatel, Microsoft, Thales ou Bull.

Au niveau européen, l'institut est un membre fondateur du Consortium européen de recherche en sciences informatiques et mathématiques. Il coordonne également la partie française du réseau **EIT ICT Labs** dédié à l'« internet du futur ».
On notera également une bonne visibilité à l'étranger par de nombreux partenariat aux États-Unis, en Amérique Latine, en Afrique, au Moyen-Orient ou en Asie, et notamment en Chine grâce au Laboratoire franco-chinois de recherche en informatique, automatique et mathématiques appliquées (**LIAMA**) implanté à Beijing dont l'**Inria** est un des fondateurs.

2.1.3 Quelques projets Inria

Parmi les projets menés par l'**Inria**, nous pouvons citer :

- Les langages de programmation **Caml** et dérivés, très utilisés dans l'enseignement.
- L'assistant de preuves **Coq** (preuve de theorèmes).
- La bibliothèque de calcul flottant multiprécision **MPFR**.
- **Scilab**, un logiciel pour le calcul numérique et scientifique, similaire à **MATLAB**.
- La licence de logiciel libre **CeCILL**, co-écrite avec le **CEA** et le **CNRS**.
- ...

2.2 L'Inria Lille - Nord Europe

En 2002 est créé le centre **Inria Futurs**, incubateur du centre de recherche de Lille qui voit le jour en 2008. Le centre représente quelques **300** personnes, dont **200** chercheurs pour un total de **36** nationalités représentées, **14** équipes de recherches et plus de **80** contrats de partenariat industriels. Le centre de Lille oriente ses recherches vers trois principales thématiques :

- Infrastructures logicielles pour l'intelligence ambiante
- Interaction et modélisation du vivant
- Modélisation et simulation des systèmes complexes

2.3 L'équipe DOLPHIN

L'équipe qui m'a accueillie est l'équipe **DOLPHIN** [**?**] (pour Discrete multi-objective Optimization for Large scale Problems with Hybrid dIstributed techNiques.) spécialisée dans les thèmes de l'optimisation, l'apprentissage et les méthodes statistiques. Elle est dirigée par El-Ghazali Talbi.

L'équipe **DOLPHIN** a pour objectif la modélisation et la résolution parallèle de problèmes d'optimisation combinatoire (multi-objectifs) de grande taille. Des méthodes parallèles coopératives efficaces sont développées à partir de l'analyse de la structure du problème traité. Les problèmes traités font partie de la classe des problèmes génériques (ordonnancement flow-shop, élaboration de tournées, etc.) ou bien des problèmes industriels issus de la logistique, transport, énergie et de la bioinformatique.

2.4 ParadisEO

2.4.1 Présentation

Le code de **ParadisEO** a été développé par Sébastien Cahon entre 2004 et 2006, lors de sa thèse, sous la direction de E-G Talbi et N. Melab.
Tout au long de son existence, des doctorants et des chercheurs ont développé et intégré de nouveaux algorithmes dans **ParadisEO**.
Le framework s'articule autour de 4 modules [**?**], reposant tous sur **EO** :

- **MO**, pour l'optimisation mono-objectif.
- **MOEO**, pour l'optimisation multi-objectifs.
- **PEO**, pour la parallélisation distribuée (sur grille et sur cluster) et à mémoire partagée.
- **GPU**, pour le calcul GPGPU, sur GPU Nvidia, via CUDA.

2006	1 ingénieur
2007	2 ingénieurs
2008	3 ingénieurs
2009	2 ingénieurs
2010	1 ingénieur
2011	1 ingénieur

TABLE 2.2 – Ressources humaines affectées

2.4.2 Algorithme génétique metaheuristiques

Principe des algorithmes génétiques

Les **algorithmes génétiques** font partie des méthodes dites d'**intelligence calculatoire**. Ils se basent 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. Le framework **EO** s'occupe de fournir l'architecture de base et les méthodes communes à l'élaboration de n'importe quel algorithme génétique, tandis que

ParadisEO exploite ces fonctionnalités dans le cadre plus spécifique de problèmes d'optimisation de la classe **NP-difficile**.

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 Evolutionnary Algorithm) pour l'optimisation multi-objectifs.

Fonctionnement d'un algorithme génétique

Même s'il existe une grande variété d'algorithmes génétiques, le fonctionnement général reste souvent le même. 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 attribut 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.

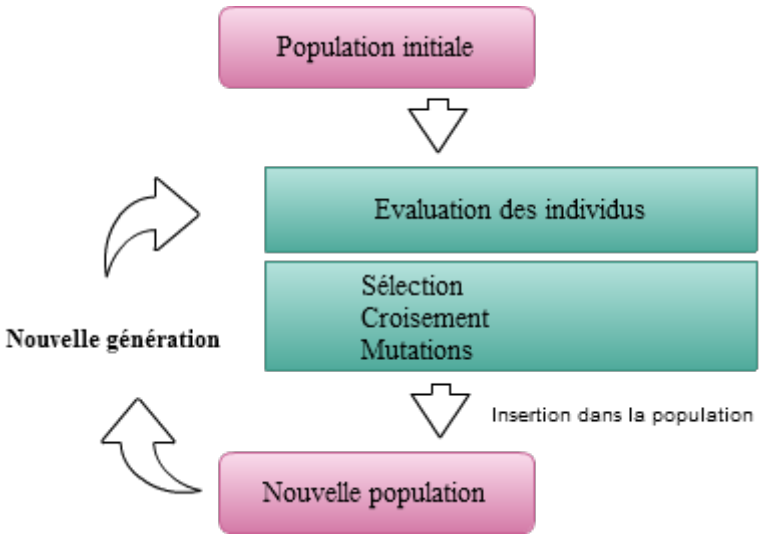

FIGURE 2.1 – Principe des algorithmes génétiques

Chapitre 3

Bonne pratique du développement logiciel

3.1 Le management du code

3.1.1 Gestionnaire de version et les forges

Lors du développement d'un logiciel, le gestionnaire de version apporte de nombreux avantages. Evidemment, il permet de centraliser les sources du projet dans un seul endroit, et évite les pertes accidentelles dû à un problème matériel par exemple, mais de permettre à chacun d'accéder et de travailler sur les versions les plus récentes des fichiers.
Il permet également la création de **branches** qui s'articulent autour d'un « tronc » ou d'une branche dite « master » (celui le logiciel utilisé) qui autorise un développeur à ajouter des fonctionnalités et travailler sur le projet sans impacter les autres développeurs, puis d'intégrer ses modifications (ou de les supprimer) avec la branche principale par la suite. Avec ce principe de branches, et l'historique de toutes les modifications effectuées par chacun des développeurs, le gestionnaire de version est indispensable pour éviter les **régressions** du code, et tout simplement pour mettre en place un travail collaboratif.
On peut également ajouter la possibilité de « tagger » une révision, c'est à dire l'estampiller d'une certaine version afin de donner des repères temporels clairs dans l'évolution du logiciel, notamment pour les utilisateurs.

Les **forges** sont des systèmes de gestion de développement collaboratif, intégrant, bien évidemment, un gestionnaire de version et d'autres outils de suivis et administration d'un projet. La forge **Inria** [1] par exemple, propose des rapports d'activités, la gestion de listes de diffusions (pour les développeurs, l'aide aux utilisateurs, les annonces, etc.), un forum, des outils de suivis (bug reports, support requests, feature requests), un calendrier, etc.

3.1.2 Les systèmes de build

La construction logicielle ou **build**, a plusieurs tâches, exécutées de manière séquentielle qui vont mener à l'obtention d'un logiciel installé et utilisable.

1. https ://gforge.inria.fr/

FIGURE 3.1 – Déroulement d'une construction logicielle

Brièvement, il consiste à s'assurer que les **dépendances** sont installées (compilateur, librairies, etc.), voire même tester la **compatibilité** de la machine (architecture 32/64 bits, environnement UNIX, POSIX, etc.), **compiler** les sources, exécuter les **tests** pour s'assurer que le logiciel fonctionnera correctement sur la machine. On peut de manière optionnelle mais recommandée tester la qualité du code par des **tests de couverture** ou de **fuites mémoire** par exemple (voir outils complémentaires).
Ensuite vient la mise en **paquets** du logiciel. Typiquement des binaires pré-compilés pour un OS particulier (.deb pour Debian-like, .rpm pour RedHat-like, .exe pour Windows, etc.).
Enfin, l'étape de **déploiement** permet d'installer le logiciel de manière correcte et flexible pour l'utilisateur final.

En effectuant toutes ces tâches de manière automatisée, on s'assure qu'à terme, il sera distribuable sur toutes les plateformes supportées. C'est d'autant plus important lorsque les contraintes de portabilité sont assez importantes comme sous **ParadisEO** puisque le framework doit être disponible sous **Linux**, **Mac OS X** et **Windows**, en supportant les architectures classiques (32 et 64 bits) et en fournissant des paquets pour la majorité des distributions **Linux**.

Image modifée depuis :
http ://blog.soat.fr/2010/03/industrialisation-des-developpements-jee-integration-continue/

3.1.3 Présentation de CMake

CMake est un système de construction logicielle **libre**, **multi-plateforme** et **multi-langage**. L'un des avantages de **CMake** est qu'il permet de générer des fichiers projets pour un grand nombre d'**IDE** comme Eclipse ou CodeBlocks. Ainsi, chaque utilisateur ou développeur peut utiliser son propre **IDE** sans que cela n'impacte le projet ou ne restreigne les autres utilisateurs.

CMake s'utilise de manière très peu intrusive en ajoutant simplement des fichiers **CMakeLists.txt** dans chacun des répertoires du projets, contenant les directives de construction : création d'une bibliothèque, cible personnalisée pour la création d'une documentation, résolution des dépendances, directives d'installation, etc. On notera que CMake gère le **build** dit « out-of-sources », c'est à dire la construction du logiciel dans un répertoire séparé des sources ce qui présente l'avantage de laisser l'arborescence du projet propre (notamment pour faciliter l'utilisation du gestionnaire de version).

CMake est aussi livré avec d'autres outils très pratique qui complète le processus de construction. **CTest** permet de gérer les tests. **CPack** permet de faire de mettre en paquet un **build**, par exemple un .deb pour les sytèmes Debian, .rpm pour les systèmes RedHat, .exe pour Windows, etc. Ces paquets sont réalisés par des générateurs externes, comme **rpmbuild** sous RedHat ou encore **NSIS** sous Windows. **CPack** s'occupe simplement de générer, à partir des **CMakeLists**, les scripts utilisés par ces générateurs.
Enfin, **CDash** est un dashboard permettant de visualiser différentes informations sur les **builds** : erreurs et avertissements à la compilation, information sur le système qui a lancé le **build** (OS, architecture CPU, RAM disponible, etc.), les **tests** effectués, réussis ou échoués, les **tests de couverture** et des **tests mémoire**.

3.2 Les tests

3.2.1 Boites blanches et noires

Un système informatique est qualifié de **boite blanche** si son fonctionnement interne est parfaitement connu et le résultat de celui-ci **déductible**. C'est donc l'inverse d'une **boite noire** qui va masquer les détails internes d'un logiciel, module ou partie de code.
Ainsi, des tests sur une **boite blanche** consistent à tester les mécanismes internes du système, en choisissant des données en entrée et en validant les résultats en sortie. Les résultats en sortie peuvent être validés à partir de la connaissance du fonctionnement de la **boite blanche**.
Ces tests s'opposent aux tests sur **boites noires** qui s'effectuent non pas sur les mécanismes internes mais sur la **fonctionnalité** du système. En l'occurrence, on s'assure le système produise des résultats conforme à la fonctionnalité qu'il doit effectuer, sans avoir de détails sur ce qu'il se passe en interne.
Les **boites noires** sont évidemment problématiques pour garantir la fiabilité d'un système. **ParadisEO**, étant open-source, est une **boite blanche**.

3.2.2 Test unitaire

Les **tests unitaires** ont pour but de tester un module ou une fonctionnalité spécifique d'un programme pour s'assurer de son bon comportement, indépendamment du reste du code. Ainsi on teste la partie de code avec des données « classiques », puis des **données limites** pour observer le comportement du module.

Il existe des framework de **test unitaires** qui permettent d'écrire ces tests de manière très rapide ou de les générer automatiquement dans bien des cas. On peut citer **Google Test** ou **cppunit**.
Sur **ParadisEO**, les **tests unitaires** sont réalisés à la main et peuvent être exécutés via **CTest**. On remarque qu'il est assez difficile de tester certaines fonctionnalitées de **ParadisEO** puisqu'elles utilisent des opérateurs stochastiques.

3.2.3 Test d'intégration

A contrario des **tests unitaires**, permettant de tester des parties ou modules spécifiques, les **tests d'intégration** s'assurent de la cohérence des parties développées indépendamment les unes des autres. Les **tests d'intégration** interviennent donc après les **tests unitaires**.
Le couple « **tests unitaires + tests d'intégration** » permettent d'établir une composante essentielle de l'intégration continue.

3.2.4 Test de couverture

Les **tests de couverture** permettent de quantifier la proportion de code testé et donc d'avoir une vue sur les parties du code qui mériteraient d'être testées pour s'assurer de leur qualité. L'autre indicateur important de ces tests est le « code mort », c'est à dire les parties de code qui ne sont plus utilisées et que l'on peut supprimer.
L'outil le plus simple pour ce genre de test est **gcov** qui fait partie des outils **GNU**. Les rapports générés étant assez illisibles, on utilisera **lcov** qui génère des statistiques sous forme de **HTML**. Son intérêt est qu'il est utilisable nativement avec **CMake** et qu'il peut envoyer ses rapports de tests directement au dashboard **CDash**.

3.2.5 Test de non-régression

Les **tests de non-régression** ont pour but de détecter l'introduction ou la réintroduction de bugs dans le code d'un logiciel déjà testé, suite à une modification de celui-ci. Comme ces défaut peuvent survenir dans des parties non affectée par les modifications, c'est l'ensemble du logiciel qui soit être retesté.
Dans **ParadisEO**, la **non-régression** est effectuée par les **tests unitaires** qui peuvent être relancés très facilement par **CTest**.

3.3 L'intégration continue

3.3.1 Concept

Le rôle de l'**intégration continue** est de permettre de valider à chaque étape de modification d'un logiciel (ajout de fonctionnalité, mais aussi refactoring par exemple) que ces modifications n'entraînent pas de **régression** ou n'impactent pas la **stabilité** du logiciel.

Il s'agit en réalité d'un ensemble de **bonnes pratiques** permettant de participer à la qualité du logiciel tout au long de son développement, passant également par la mise en place d'outils.
Ces **bonnes pratiques** permettent notamment d'éviter l'introduction de nouveaux bugs, les problèmes de dernières minutes, permet de prévenir en cas d'incompatibilité (dépendances non satisfaites par exemple) et de tester immédiatement un logiciel et son installation.

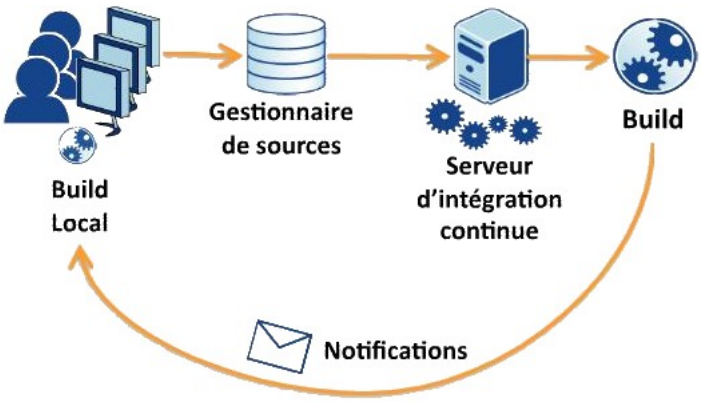

FIGURE 3.2 – Intégration continue

Sur cette image nous pouvons voir les différentes étapes de l'intégration continue :
- Lors d'une modification, le développeur s'assure que le **build** local se déroule sans problème.
- Si aucun problème n'est rencontré, la modification est intégrée dans les dépôts de l'**outil de versionnage** : git, svn, etc.
- Le **serveur d'intégration** récupère les sources et va lancer les **builds** de manière régulière (par exemple chaque jour, durant la nuit).
- Il envoie alors des rapports aux développeurs qui peuvent rapidement réagir en cas de problème.

ParadisEO n'utilise pas de serveur d'intégration continue, cependant, avoir un système de **build** robuste est essentiel pour aider à maintenir la qualité du logiciel et également faciliter l'accès à celui-ci.

Une solution complémentaire ou alternative au serveur d'intégration continue consiste à produire des **Nightly Build**, qui correspondent à des **builds** automatiques, souvent effectués la nuit à partir des dernières sources disponibles sur la **forge**. Comme pour le **serveur d'intégration continue**, une batterie de tests est effectuée et des notifications sont envoyées aux développeurs pour parer aux problèmes le plus tôt possible. Le principe est particulièrement pertinent sur des projets de recherche comme **ParadisEO** (qui fonctionnait avec des **Nightly Build** il y a quelques années).

3.3.2 Méthodes agiles

De manière générale, les **méthodes agiles** s'opposent aux méthodes monolothiques telles que le **cycle en V** ou le **modèle en cascade**. Là où ces dernières reposent sur un développement de bout en bout, avec l'effet « levé de rideau » pour l'utilisateur final ou le client, les **méthodes agiles** proposent un développement **incrémental** basé sur des **itérations courtes**. L'avantage des **méthodes agiles** sont multiples : le client peut mieux piloter le projet, la complexité du développement est réduit à celle de l'itération (plus courte donc

Image modifée depuis :
http ://blog.soat.fr/2010/03/industrialisation-des-developpements-jee-integration-continue/

moindre par rapport à l'ensemble du projet), un **système d'intégration continue** est bien plus facile à mettre en place et efficace du fait que chaque itération fourni un **produit fonctionnel** et donc testable.

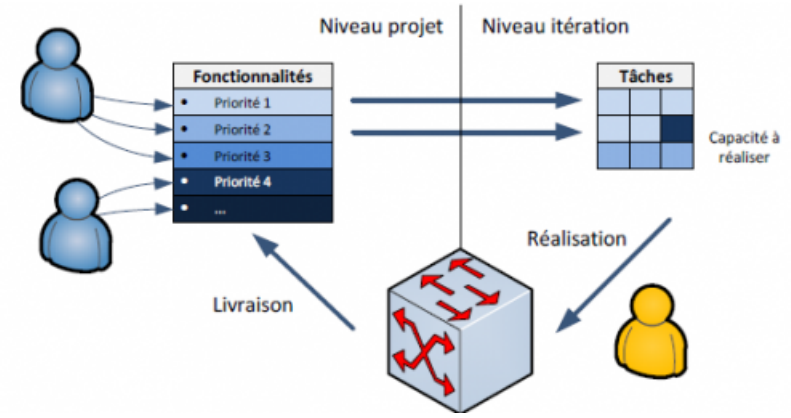

FIGURE 3.3 – Principe d'une méthode agile

Il existe un grand nombre de **méthodes agiles** (XP, Scrum, etc.) possédant chacune ses caractéristiques. Sur le schéma suivant on peut voir une itération d'une méthode agile. L'**intégration continue** s'intercalle à la fois sur le partie **Développement**, grâce aux builds automatiques, mais également et surtout sur toute la partie **Mise en production**.

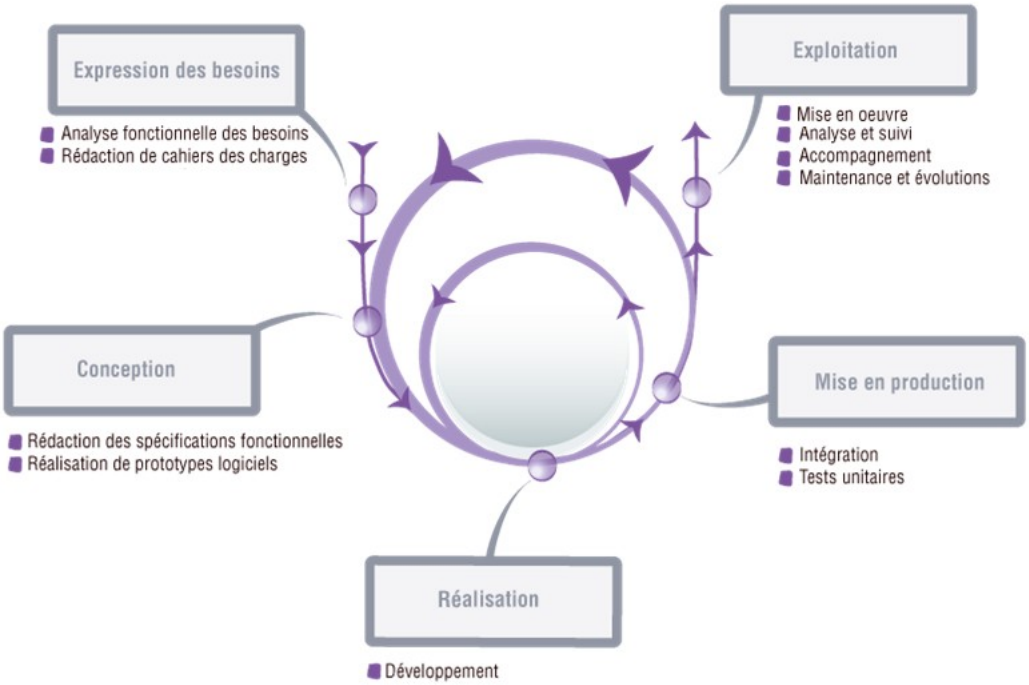

FIGURE 3.4 – Illustration d'une méthode agile

ParadisEO adopte une méthode agile de développement, notamment parce que ce modèle est particulièrement adapté aux équipes-projets comme elles sont définies à **Inria**. De plus,

Crédit illustration :
http ://fleid.net/2012/01/10/gestion-de-projet-decisionnel-methodes-agiles-ou-cycle-en-v/
Avec l'aimable autorisation de Florian Eiden.

Image modifée depuis :
http ://www.itsaxir.com/fr/engagements/methodologies/developpement

sur **ParadisEO**, plusieurs modules coexistent, répondant à des besoins différents, et n'étant couplé qu'à une base commune, le framework **EO**. Ils peuvent donc évoluer de manière indépendante en fonction des nouveaux besoins fonctionnels apparaissant.

Chapitre 4

Stabilisation de ParadisEO

4.1 Problématique

Depuis quelques années, aucune version stable de **ParadisEO** n'est sortie, le support Windows a été délaissé et l'installation est devenue très difficile. En effet, elle repose sur des scripts bash longs et complexes, alors que cette technologie n'est clairement pas adaptée pour générer des installations portables, propres et stables de logiciels.
De même, récemment des modifications dans l'interprétation de la norme C++ par gcc, le compilateur gnu, ont fait apparaître des erreurs à la compilation, rendant le programme non-instalable sans toucher au code source.

Il s'agissait alors de stabiliser **ParadisEO** pour sortir une nouvelle version, intégrant la dernière version de **EO**, ainsi que revoir l'intégralité du système de build et d'intégration continue pour permettre une manière plus souple de distribuer, installer et tester **ParadisEO**, tout en facilitant le développement des futures versions.
Un des axes principaux de travail était la facilité de l'utilisateur à installer puis utiliser **ParadisEO** pour ses projets.

4.2 Présentation de l'existant et limites

La version de **ParadisEO** sur laquelle j'ai basé mon travail était la version **1.4b**. datant de septembre 2011. Le script bash sur laquelle était basée l'installation faisait plus de 1000 lignes alors que tout ce qu'il faisait était aujourd'hui réalisable par **CMake** , de manière beaucoup plus courte et sûre.
Parmi les défauts de ce script on note :

- Bash n'est pas portable (n'existe pas sous Windows et un utilisateur UNIX peut préférer utiliser zsh, sh, csh, etc.)
- Bash n'est pas adapté pour générer un projet d'une telle envergure (nombreux modules, compilation conditionnelle, dépendances variées, etc.)
- Le script était très intrusif, notamment en envoyant par défaut des logs sur les serveurs **Inria** ou en écrivant dans le bashrc de l'utilisateur (ce qui dénote encore une fois de la non portabilité de l'application)

De plus, la hiérarchie des builds générés était peu pratique et ne reflétait pas l'unité du logiciel. En effet, chaque module du logiciel disposait de son fichier de build, alors que l'avantage de **CMake** , comme dit précédemment, est de permettre d'avoir un build complètement extérieur à la hiérarchie du projet.

FIGURE 4.1 – Répertoires du projet avant et après

Un autre problème, lié à la politique de distribution du logiciel, est que **ParadisEO** était fourni avec ses dépendances, notamment **libXML** et **MPICH2**, ce qui faisait accroître considérablement la taille du paquet distribué (+20 Mo contre 3.5 Mo pour la version finale livrée, hors documentation). Cela posait également un problème en terme de mise à jour du logiciel puisqu'il fallait (ou aurait fallu) « repackager » **ParadisEO** à chaque nouvelle version de ses dépendances si l'on voulait les maintenir à jour.

Le **CMake** présent dans la version 1.4b était assez restreint, et possédait pas mal de défauts, notamment une redondance de code, une flexibilité très faible, quelques mauvaises utilisations de **CMake** (sûrement dû à l'évolution de l'outil ces dernières années), l'impossibilité de déployer proprement **ParadisEO** et par là même de l'utiliser simplement pour ses projets. On notera également la présence de pas mal de commandes dépréciées, et l'impossibilité de faire un package.
Enfin, les tutoriels n'étaient pas à jour, de même que le système de contributions / création d'un projet utilisant **ParadisEO**, encore plus après la mise en place du nouveau système de build.

Pour finir, cette version de **ParadisEO**, embarquait toujours le module Old-MO. Ce module concerne l'optimisation mono-objectif et a été totalement réécrit pour un meilleur design il y a quelques années. Cependant, il n'a jamais été retiré à cause des dépendances d'autres modules à celui-ci, notamment MOEO pour l'optimisation multi-objectifs, et **PEO**, pour le parallélisme.

4.3 Besoins fonctionnels & solutions adoptées

Après la compréhension globale de l'architecture du framework et une formation à **CMake** , l'une des première tâches a été de lister les besoins fonctionnels, et les différents travaux à réaliser.

Voici la liste des besoins fonctionnels validés ainsi que leur utilité pour le projet.

4.3.1 Portabilité

L'installation devait être portable à la fois sous les UNIX, comprenant MAC OS X, et sous Windows. Le support sous Windows a été restreint à MinGW pour des raisons de facilité du support. Les versions de Visual Studio n'assurent pas nécessairement la rétrocompabilité, et les différences à prendre en compte dans le **CMake** sont assez importantes.

4.3.2 Build type

Les cibles principales sont Debug et Release. La cible Debug ajoute des flags pour pouvoir obtenir des informations au debbuger et un niveau de verbosité plus important alors que la cible Release s'occupe de faire de l'optimisation.

4.3.3 Dashboard

L'envoie sur le Dashboard via **CDart** de rapports d'installation permet d'assurer un meilleur support de ParadisEO. Les rapports sont envoyés uniquement sur demande, contrairement au script bash qui les envoyaient, sauf contre-indication.

4.3.4 Verbosité

Permet, suivant la cible, d'avoir plus ou moins de messages relatifs au build et à la compilation. Avoir un niveau de verbosité faible est plus accueillant voire rassurant pour quelqu'un qui ne connait pas le logiciel (bien que les warnings cachés soient bénins : variables non utilisées sur certains OS par exemple).

Les pragmad sont des directives inclues par préprocesseur et qui vont être interprétées différemment selon les compilateurs. Le pragma message permet d'afficher un message à chaque passage du compilateur sur la directive. Comme **EO** et **ParadisEO** fonctionnent sur la base de template, le compilateur effectue beaucoup de passes pour résoudre les templates et affiche donc énormément de fois le même pragma. Dans une optique plus « user-friendly », il a été décidé de les désactiver.

4.3.5 Package et installation

Génération de packages (.deb, .rpm, .exe, etc.), uniquement en mode Release.
L'architecture doit-être testée pour permettre une bonne gestion des chemins d'installation et en déduire la manière de construire certains paquets (notamment les .deb).
Dans le cas d'OS X la version doit être testée et **CMake** doit lever une erreur si la version est trop ancienne.

4.3.6 Module

Un module **CMake** doit permettre à l'utilisateur de tester la dépendance à **ParadisEO**, et donc d'utiliser **ParadisEO** dans son projet facilement.
C'est une alternative à l'approche intrusive du script bash qui modifiait le bashrc de l'utilisateur.

4.3.7 Divers

Documentation

Génération de la documentation avec Doxygen. Mise à jour des fichiers de configuration de Doxygen.

MrProper

Une simple cible pour supprimer l'ensemble du répertoire de build. Très utile en développement, notamment lorsqu'on travaille justement sur les builds.

Tests

Ils ne sont pas générés et exécutés par défaut. Envoie possible sur le dashboard.

Couverture

Commande personnalisée pour lancer des tests de couverture avec lcov.

Compilateur

Permettre à l'utilisateur de spécifier son compilateur. Implique de toucher un petit peu au code de **EO** pour supprimer la dépendance à **OpenMP** et ainsi supporter **Clang** (qui n'a pas d'implémentation à ce jour d'**OpenMP**).

Permettre à **CMake** de sortir rapidemment lorsque le compilateur n'est pas trouvé. Le comportement par défaut lève l'erreur bien plus tard qu'au moment de la détection du compilateur.

Tutoriaux

Définir une macro pour ajouter facilement des tutoriaux (le code actuel étant très redondant).

4.4 Mise en place d'un serveur Git

Comme il a été décidé de passer à **Git** et comme nous ne savions pas s'il était possible de conserver l'historique des commits entre **SVN** et **Git**, j'ai décidé de travailler sur un serveur **Git** privé indépendant de la forge **Inria**.

Après quelques tests, il s'est avéré que git permettait d'importer sans perte d'historique des dépôts **SVN** [**?**]. C'est donc en fin de stage, entre les tests et la sortie de version que j'ai effectué la migration.

4.5 Intégration des dernières versions

La première chose à faire pour travailler était d'avoir une version installable avec l'existant. En effet, quelques erreurs apparaissaient dans lors de la compilation. La première était l'oubli de l'inclusion de certains headers standards. Voici un exemple :

```
erreur: ‘size_t’ was not declared in this scope
```

La seconde erreur provenait d'un changement de comportement de gcc vis à vis de la norme. En effet, depuis gcc **4.7**, si une méthode est utilisée dans une classe templatée, et que cette méthode n'utilise à aucun moment le template de la classe dont elle fait partie, il faut la préfixer par this->. Cela permet notamment à gcc de déduire plus rapidement la portée des méthodes, et d'accélérer la compilation en présence de templates [**?**].
La majorité des classes de **EO** et **ParadisEO** sont templatées, et on retrouve quelques classes dans ce cas de figure, avec plus ou moins d'opérations à préfixer.

```
note: declarations in dependent base ‘eoEasyEA<Indi>’ are not found by
 unqualified lookup
note: use ‘this->operator()’ instead
```

Ces quelques erreurs corrigées, l'installation s'effectuait correctement.
J'ai ensuite cloné le dépôt git de **EO** pour obtenir la dernière version de **EO** disponible et l'intégrer au projet pour la version stable.
Quelques soucis sont apparus, notamment des petites changements dans les noms de certaines classes et des noms de fichiers, ainsi que des modifications mineures dans le nom des variables du **CMake** existant.

4.6 Découplage de OldMO et PEO

Dans un premier temps, j'ai purement et simplement supprimé le dossier contenant toutes les sources de **PEO**. Comme **PEO** n'était une dépendance pour aucun des autres modules, sa suppression n'a aucunemment impactée l'installation et le fonctionnement des autres fichiers.

OldMO posa plus de problèmes dans le sens où il restait certaines dépendances à ce module dans **MOEO**. En réalité, il ne s'agissait que de headers qui n'avaient pas été supprimés et de quelques classes dont la seule modification entre **MO** et **MOEO** fut le nom.
Par contre, il subsistait un certain nombre de tests qui requierait la présence de **OldMO**. Certains ne testaient que des parties de **OldMO** et pouvaient donc être retirés sans soucis. Pour les autres, j'ai du réécrire quelques parties de tests pour qu'ils fonctionnent à nouveau.

On note par exemple qu'il existe des équivalents, en tout cas dans les concepts, entre les classes suivantes [**?**] :
moNeighbor ⇔ moMove + informations à propos de l'individu
moNeighborhood ⇔ moMoveInit + moMoveNext + comment évaluer le voisinage
moEval ⇔ moIncrEval + évaluation complète de la population

Il était donc assez facile de réécrire des tests en se basant sur l'API, les tutoriaux et les autres tests existants.

4.7 Réecriture du CMake

4.7.1 Méthodologie et déroulement

La méthode que j'ai adoptée pour réécrire le **CMake** était une méthode incrémentale. Dans un premier temps, j'ai écris un **CMakeLists.txt** principal, à la racine du projet, minimal, afin de construire un build « out-of-source » fonctionnel, et ce, module par module (**EO**, puis **MO**, puis **MOEO**, pour les dépendances.)
Les itérations suivantes étaient focalisées sur la réalisation du **CMake** pour **MO**. Celui de **MOEO** pourrait être parfaitement calqué sur celui de **MO**. La contrainte était de ne pas toucher au **CMake** de **EO**, ou le moins possible pour pouvoir intégrer facilement de futures versions de **EO**, sans avoir à modifier le **CMake** , jusqu'à la fusion des projets.

Une fois ceci fait, j'ai pu factoriser certains comportements, notamment pour l'écriture des tutoriaux, puis établir des cibles globales : profiling, documentation, rapport sur CDash, etc.

Enfin, il restait à ajouter les directives d'installation, de génération de paquets et le fichier FindParadiseo. Durant ces phases, je me suis aperçu qu'il fallait obligatoirement toucher au **CMake** de **EO** pour plusieurs raisons. Dans un premier temps, quelques messages d'avertissements subsistaient dans le fichier de configuration de **Doxygen** de **EO**, à cause de directives dépréciées. Dans un second temps, l'ensemble des chemins d'installation était à corriger afin d'obtenir une hiérarchie homogène et agréable pour l'utilisateur. Toujours dans un soucis d'homogénéité, il fallait modifier le nom des cibles personnalisées de **EO**, notamment celle nommée doc en la préfixant par -eo afin d'avoir une cible générale doc pour le logiciel qui appelerait doc-module pour chacun des modules.

Un autre soucis, était que le **CMake** de **EO** utilisait souvent les variables ${CMAKE_BIN_DIR} ou ${CMAKE_SOURCE_DIR} pour désigner la base de ses chemins. Or, ces variables designent respectivement la racine du répertoire de build et du répertoire des sources du projet. Seulement, comme **EO** devient un module de **ParadisEO**, ces variables empêchaient d'avoir un build « out-of-sources » propre.
Il a donc fallu toutes les remplacer par des ${CMAKE_CURRENT_BIN_DIR} et ${CMAKE_CURRENT_SOURCE_DIR} qui utilise cette fois le chemin relatif.

Enfin, plusieurs modifications ont dû être effectuées dans les directives d'installation.

4.7.2 Organisation du build

Le nombre de directives pour la construction de **ParadisEO** étant assez importantes, j'ai décidé, comme cela se fait sur des projets d'une certaine envergure, de créer un répertoire **CMake** et d'y déposer différents fichiers thématiques relatifs à la construction du logiciel.

On retrouve donc :

- **Module** : répertoire contenant le module de recherche de ParadisEO.
- **Config.cmake** : fichier regroupant les directives relatives aux cibles, flags, modes d'installation et aux différents paramètres configurables par l'utilisateur.
- **Macro.cmake** : comme son nom l'indique, il regroupe les différentes macro utilisées dans tout le logiciel, notamment l'ajout (presque) automatique de nouveaux tutoriels.
- **Package.cmake** : fichier relatif à la mise en paquet du logiciel quelque soit le système d'exploitation.
- **Target.cmake** : fichier regroupant les cibles personnalisées globales telles que le profiling, la cible de documentation globale, ou encore le nettoyage du répertoire de build.

Cette organisation permet de faciliter l'évolutivité du système de construction et la maintenance de celui-ci.

4.8 Ecriture du findParadiseo

4.8.1 Présentation & intérêts

Lorsqu'un utilisateur se procure **ParadisEO**, plusieurs choix s'offrent à lui. S'il est administrateur de la machine, il peut installer des paquets, ou compiler les sources puis utiliser la cible d'installation créée par **CMake** . S'il n'a pas les droits d'administration, il peut simplement compiler.
Cependant, dans les deux cas, comment peut-il utiliser **ParadisEO** dans un projet personnel qu'il a besoin de redistribuer ? Cela pose plusieurs problèmes : où sont situées les bibliothèques de **ParadisEO** ? Où sont situés les headers à inclure ?

Dans le cas où l'utilisateur utilise **CMake** pour son projet, le but était de fournir un fichier **findParadisEO.cmake**, nommé module dans la nomenclature de **CMake** . Il a pour but de permettre au build du projet de l'utilisateur de trouver **ParadisEO**, quelque soit l'OS ou la machine, ainsi que de spécifier quelles bibliothèques fournies par **ParadisEO** il veut utiliser et donc charger (par exemple libeo, libmoeo, etc.).
Le fichier **FindParadiEO.cmake** peut donc simplement être redistribué avec n'importe quel projet utilisant **ParadisEO**. Au moment de créer le build, **CMake** incluera le module qui cherchera dans des chemins spécifiques, suivants certains paramètres, les bibliothèques et headers. S'il ne les trouve pas, il renverra une erreur indiquant que la dépendance n'est pas satisfaite. A l'utilisateur final de faire le nécessaire pour installer ParadisEO.

Une autre utilisation de ce fichier consiste à le placer, par un administrateur, dans le sous-répertoire module du répertoire d'installation de **CMake** . Ainsi, les utilisateurs n'ont qu'une directive à rajouter à leur fichier **CMakeLists.txt** principal pour tester la dépendance à **ParadisEO** (alors normalement installé).

4.8.2 Déroulement du module

Le déroulement du module est purement séquentiel. Après les différentes instructions, une étape de résolution est effectuée, permettant soit de lever une erreur lors de la construction du projet utilisateur, soit de retourner des variables utiles à l'utilisateur pour construire son logiciel : chemins des en-têtes, librairies, etc.

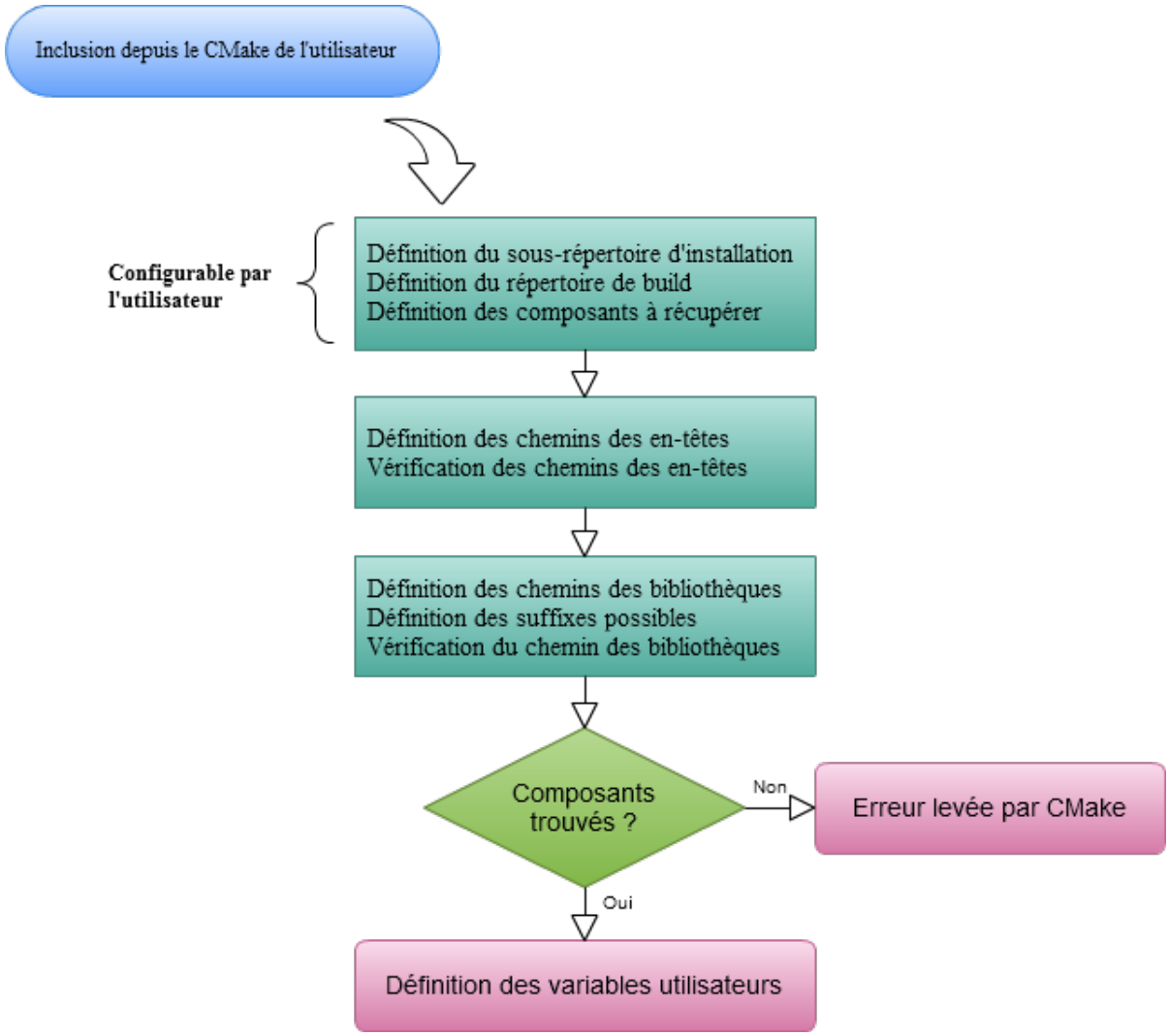

FIGURE 4.2 – Déroulement du module findParadiseo.cmake

Utilisation du module

Pour utiliser **ParadisEO** dans leur projet, les utilisateurs n'ont qu'à rajouter les quelques lignes suivantes :

```
1 set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/
    module")
2 find_package(Paradiseo COMPONENTS moeo eoutils eo)
3
4 include_directories(${PARADISEO_INCLUDE_DIR})
```

Listing 4.1– Utilisation de FindParadiseo.cmake

Avec évidemment la liste des composants qu'ils souhaitent utiliser.

Ensuite, ils peuvent « linker » leurs exécutables avec les bibliothèques ainsi trouvées :

```
1 target_link_libraries(ZDT_NSGAII ${PARADISEO_LIBRARIES} lZDT)
```

Listing 4.2– Edition de liens en utilisant les bibliothèques souhaitées

4.8.3 Système de contributions

Un système de contribution existait déjà. Il s'agissait d'une archive contenant un répertoire d'exemple de contribution utilisant **ParadisEO** et d'un répertoire proposant un squelette de contribution.
J'ai mis à jour le **CMake** de l'archive, notamment en intégrant le module **FindParadiseo.cmake**, et complétement réécrit le guide explicatif. Le guide présente notamment l'utilisation basique de **CMake** et quelques bonnes pratiques du développement avec les outils tournant autour de **CMake** . Les utilisateurs n'ont alors qu'à envoyer l'archive pour qu'elle soit intégrée à **ParadisEO** si la contribution est jugée pertinente.

Je suis d'avis qu'il s'agit plus d'une aide à la création d'un projet utilisant **ParadisEO** plutôt qu'un système de contributions. La réunion à **Thalès** est d'ailleurs allée dans ce sens. En effet, avec la décision de passer à **Git**, il devient très facile pour l'utilisateur voulant contribuer de le faire en clonant le dépôt, produisant son code puis en demander un push sur le serveur. Evidemment, il reste à déterminer une politique quand aux contributions, aux utilisateurs habilités à contribuer, etc.

4.8.4 Tests pour la release

Les tests ont été effectués en fin de stage, les deux dernières semaines. Ils consistent à tester de manière exhaustive l'installation de **ParadisEO** sur le plus grand nombre de plateformes possible.

Dans un premier temps il a fallu écrire une politique de tests afin de coordonner au mieux, les différentes installations.

Nous avons choisi de tester l'installation de **ParadisEO** sur les OS suivants, qui représentent à peu près l'ensemble des types de paquets différents que l'on peut trouver :

- Fedora
- Ubuntu
- OS X Lion + Snow Leopard
- Windows 7

Dans chacun des cas, il fallait tester pour une version 32 et 64 bits.

A partir d'iso vierge, voici la marche à suivre que a été établie pour tester correctement une plateforme :

- Constuire **ParadisEO** en :
  - Min
  - Full
  - Release
  - Debug
  - Profiling
- Lors du build profiling, tester la cible de couverture via lcov
- Pour chaque build lancer les tests avec envoie sur le dashboard
- Tester les cibles de documentation
- Générer les paquets (avec cross-packaging : générer des paquets .deb sous Fedora et .rpm sous Ubuntu)
- Tester un projet minimal avant installation, sans root, en spécifiant le chemin via PARADISEO_ROOT

- Tester après installation via paquet, sans le chemin, toujours sans root
- Tester avec un IDE quelconque un linkage avec et sans installation
- Tester la désinstallation du paquet
- Tester la cible install

Chapitre 5

Le parallélisme

5.1 Présentation

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 prete au parallélisme ce qui est le cas de nombreux problèmes récurrents comme la **modélisation** et la **simulation** en générale, le **traitement de l'information** et de l'image, etc.

On distingue deux types de parallélismes : le parallélisme de **données** et de **tâches**. Le premier vise à effectuer en parallèle le traitement de données, réduisant ainsi le temps de calcul. Il s'agit essentiellement de modifier un algorithme déjà existant pour l'adapter au parallélisme. Le second type préfère paralléliser des tâches différentes. 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** décrit plus loin, qui par nature, n'existe que sur des architectures parallèles.

5.1.1 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 intégrent des instructions **SIMD** dont tirent pleinement profit des bibliothèques comme **BLAS** par exemple.

Les processeurs **vectoriels** font partis 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 avec **ParadiseEO**.

5.2 Les architectures parallèles

Parmi les architectures **MIMD**, on peut distinguer deux grands types d'architecture paralléle, 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 du distribuée.

5.2.1 Mémoire partagée

Dans ce cas, les processeurs lancés, aussi appellés **processus légers** ou « **thread** », accèdent à une mémoire commune. C'est typiquement le cas des processeurs multicores 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 déléguera 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.

5.2.2 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 caractéritiques 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 envoie 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.

http ://openmp.org/wp/openmp-compilers/

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**.

5.2.3 Modèle hybride

Il existe évidemment des modèles hybrides, alliant mémoire partagée et mémoire distribuée. Un cluster peut être composé de différents noeuds, donc à mémoire distribuée, et à l'intérieur de chaque noeud, on peut retrouver des processeurs à mémoire partagée.
Ce genre de modèle permet d'avoir un niveau de granularité supplémentaire pour peu que l'on sache en tirer profit (le développement pourra être très dépendant des contraintes matérielles).
Notons que d'autres modèles hybrides existent (notamment avec l'utilisation de **GPU** par exemple).

5.3 Performances

5.3.1 Loi d'Amdahl

La **loi d'Amdahl** permet de quantifier le gain supposé de performance que l'on peut attendre en améliorant une composante de sa performance (en général le nombre de noeuds ou de processeurs).
Elle s'énonce ainsi :

$$T_n = (1 - s)T_p + \frac{sT_p}{A}$$

Avec T_n le temps d'exécution obtenu sur le nouveau système, T_a le temps sur l'ancien système, s la fraction du temps T_a concerné par l'amélioration et A l'accélération obtenue par l'amélioration.

On peut ensuite en déduire le **speedup** ou amélioration globale :

$$S = \frac{1}{(1 - s)} + \frac{s}{A}$$

L'étude de la limite révèle que :

$$S_{max} = \frac{1}{1 - s}$$

On peut donc en déduire que le gain de performance est conditionné par la partie non-concernée par l'amélioration ce qui semble tout à fait logique.

Cependant, la **loi d'Amdahl** est optimiste puisqu'elle ne tient pas compte de contraintes matérielles, de temps de communication, etc. Si on s'attend, en suivant la loi, à avoir une augmentation linéaire du **speedup** en fonction du nombre de coeurs disponibles, en réalité très vite l'augmentation de coeurs n'apporte qu'un gain très faible comme témoigne le graphique suivant :

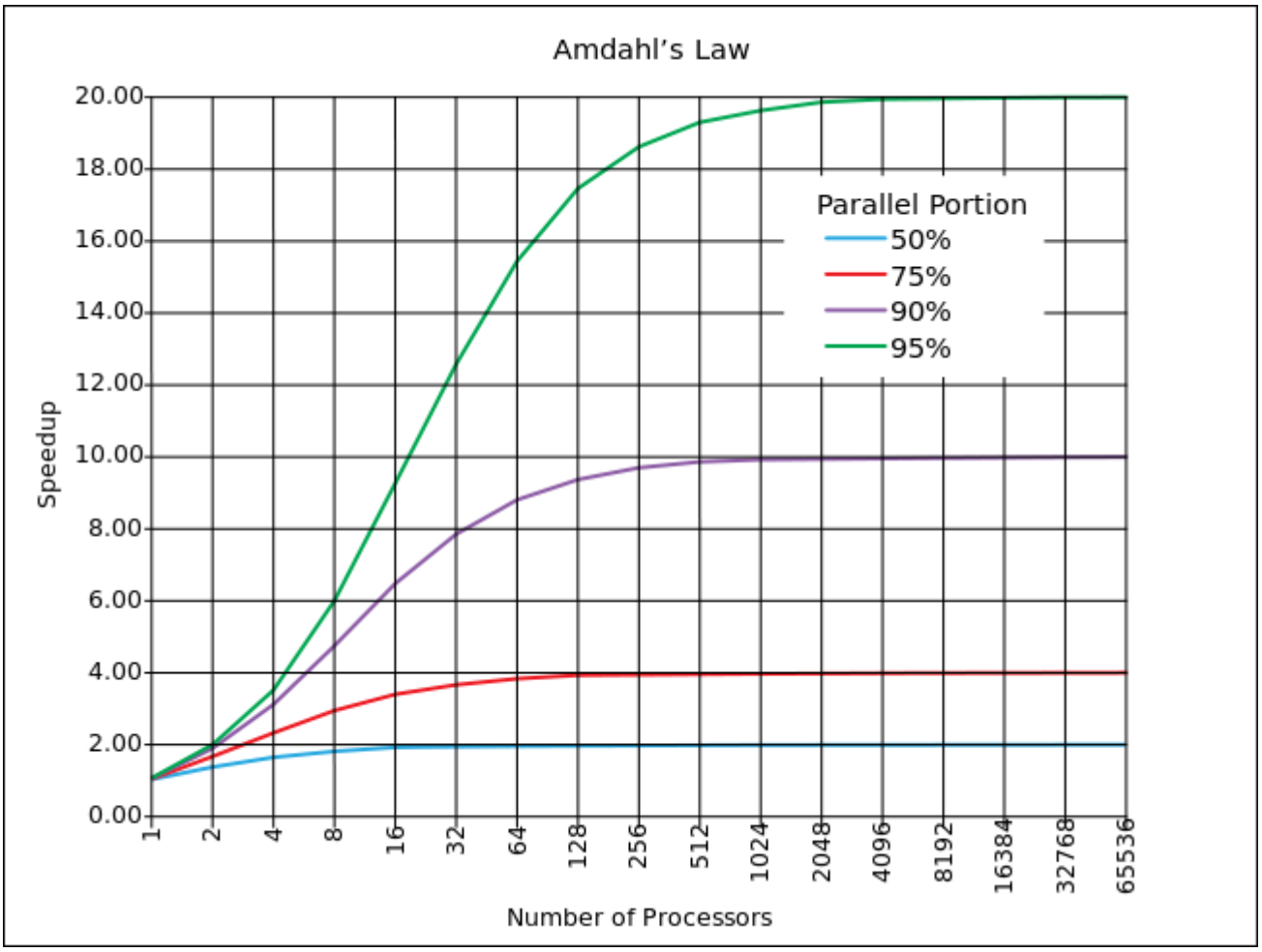

FIGURE 5.1 – Illustration de la loi d'Amdahl

Dans le cas de parallélisme à mémoire partagée, un autre phénomène analogue intervient. Si l'on fixe le nombre de processeurs du système et que l'on augmente le nombre de processus légers créés par le programme, alors on observera un **speedup** pratiquement linéaire jusqu'au nombre de processeurs, puis un palier avec un seuil dit critique, et enfin une chute brutale des performances.
C'est ce phénomène que j'ai essayé de montrer dans les tests de **speedup** effectués en fin de stage.

Notons qu'il existe plusieurs autres métriques de performances comme la **loi de Gustafson** ou la **métrique de Karp-Flapp**.

5.4 ParadisEO et le parallélisme

Différents modèles de parallélisation existent, plus ou moins adaptés en fonction du problème à traiter. Le module de parallélisme de **ParadisEO**, nommé **PEO**, en intégre plusieurs : le modèle **maitre / esclaves**, le **multistart** et le **modèle en îles**.

5.4.1 Modèle Maitre / Esclave

Le modèle **maitre-esclave** est le pattern le plus simple et le plus couramment utilisé en programmation concurrente. Un thread dit **maître** va envoyer des données à traiter à un nombre quelconque de threads appelés **esclave**. Le rôle du **maitre** est souvent associé à celui de **scheduler** qui va répartir les charges sur les **esclaves** et les ordonner ainsi que gérer les communications.

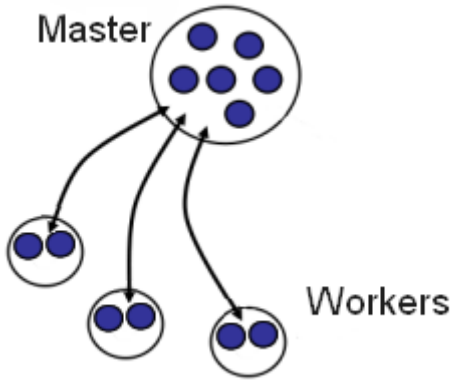

FIGURE 5.2 – Modèle Maitre / Esclave

5.4.2 Multistart

Le modèle **multistart** lance plusieurs fois le même algorithme, mais avec des données initiales différentes. Dans le cas d'algorithmes génétiques, c'est donc la population initiale qui change. Les algorithmes sont lancés en parallèles et à la fin, la meilleure solution parmi les solutions données par chacun des algorithmes est retenue.

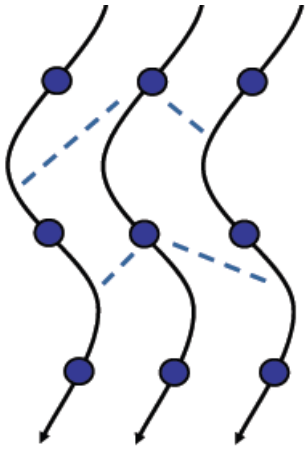

FIGURE 5.3 – Modèle Multistart

5.4.3 Modèle en îles

Un autre modèle de parallélisation, cette fois plus spécifique aux algorithmes évolutionnistes est le **modèle en îles**. Il consiste à lancer plusieurs algorithmes en même temps sur différents processus (ou noeuds pour du distribué) sur des populations différentes. Au bout d'un certain temps certains individus vont **migrer** vers un autre processus, de manière synchrone ou non. Cela permet de brasser un plus grand échantillon d'individus et d'éviter au maximum de tomber dans un **minimum local**.

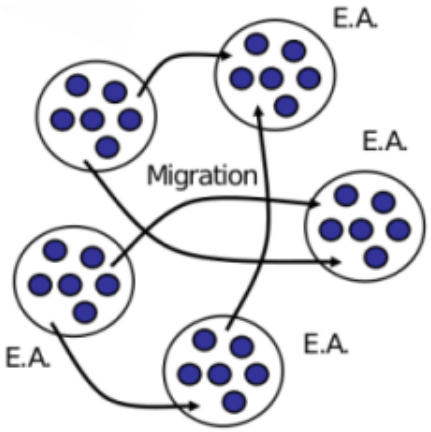

FIGURE 5.4 – Modèle en îles

Chapitre 6

Développement d'un module de parallélisme

6.1 Problématique

L'actuel module de parallélisation, **PEO**, regroupe à la fois la parallélisation distribuée, via **MPI** et son implémentation **MPICH2**, et à la fois la parallélisation à mémoire partagée via les threads POSIX.

L'objectif dans un premier temps était de faire du **refactoring** sur **PEO**, notamment en intégrant les threads standards, en réglant le problème de synchronisme, et en le découplant de **Old-MO**. La réunion à **Thalès** a complètement changé l'objectif de la mission. En effet, l'équipe de **EO** a développé de son côté un module de parallélisation distribuée avec **OpenMPI**, basique mais destiné à évoluer pour remplacer à terme la partie distribuée de **PEO**. Etant inutile de développer deux fois la même chose et le code de **PEO** vieillissant, nous nous sommes orienté vers un découplage de la partie distribuée de **PEO** de sa partie partagée. Partant du constat que peu d'utilisateurs utilisent la parallélisation distribuée tandis que tous les PC modernes sont capables de faire de la parallélisation à mémoire partagée, cela semble une bonne idée de proposer deux modules distincts.

6.2 Etude de PEO

6.2.1 Intégration de PEO comme module

Pour faire fonctionner **PEO** avec la version stabilisée de **ParadisEO**, il a fallu que je remette à jour son **CMake** ce qui fut rapide grâce au travail effectué auparavant.

6.2.2 Tentative de découplage

Dans un premier temps, j'ai essayé de me baser sur les tutoriaux existants de **PEO**, pour isoler les parties utilisant des threads des parties utilisant **MPI**. En supprimant de manière empirique certaines fonctionnalités je recontruisais **PEO** en faisant en sorte qu'il passe les tests et que les tutoriaux marchent.

N'arrivant pas à obtenir de résultat satisfaisant, et le code interne de **PEO** n'étant pas commenté (seul les quelques classes de l'API l'étaient), je me suis tourné vers le logiciel de modélisation **UML** BoUML qui permet, entre autres, de faire de la **rétroingénierie** en créant des diagrammes **UML** à partir des sources.
Grace à l'étude de ces diagrammes **UML**, nous nous sommes rendu compte que le design actuel de **PEO** ne permettait pas de manière aisée de découpler la partie distribuée de la partie partagée. Nous avons donc décidé qu'il était plus intéressant de créer un nouveau module de parallélisation à mémoire partagée depuis zéro, avec l'objectif de pouvoir l'intégrer à la release prévue pour la fin de stage.
En complément, **PEO** serait toujours distribué, cette fois à part, en attendant que le module **SMP** implémente toutes les fonctionnalités actuelles de **PEO**, en matière de mémoire partagée.

6.3 Création du nouveau module SMP

6.3.1 Objectifs

L'objectif principal lors de ce stage était de fournir une base fonctionnelle pour un module de parallélisation à mémoire partagée afin de l'intégrer lors de la sortie de la nouvelle version de **ParadisEO** en fin de stage.

Le modèle **Maitre / Esclave** est le premier modèle de parallélisation qu'il a été décidé d'implémenter dans le nouveau module de **ParadisEO** nommé **SMP** pour **Slave/Master ParadisEO** mais aussi (et surtout à terme) pour **Symmetric MultiProcessing** ou encore **Shared Memory Parallelism**. Le modèle doit être capable de paralléliser les algorithmes plus utilisés de **EO** et permettre l'ajout de nouveaux algorithmes le plus facilement possible.

Dans un second temps, le **modèle en îles** sera implémenté (mais pas dans le cadre de ce stage, le temps imparti étant trop court).

6.3.2 Les contraintes

La principale contrainte à respecter était celle de l'utilisateur. En effet, là où **PEO** proposait une API différente à l'utilisateur, nous voulions ici encapsuler complètement et de manière transparente les différents algorithmes qui lui sont proposés par les autres modules de **ParadisEO** et EO.
Cela impliquait donc un maximum de **généricité** et de **factorisation** de code, en passant notamment par de la **méta-programmation**. Les outils apportés par le C++ 2011 étaient

Le SMP est une architecture parallèle qui consiste à augmenter le nombre de CPU pour accroitre la puissance de calcul.

alors indispensables pour arriver à cette fin, ne serait-ce que par la présence des « templates variadic », du développement des « type-traits »ou par l'ajout des `std::function` ou `std::bind` hérités de la librairie **Boost**.
La récente standardisation des threads (impliquant leur portabilité à terme) et des types « atomic » ajoutait encore un poids à l'adoption de cette norme pour le module **SMP**.

Cela a tout de même une conséquence sur la distribution du module. Bien que livré avec **ParadisEO** (contrairement à **PEO** qui reste ajoutable par la suite), le module ne sera pas installé par défaut puisqu'il requiert des compilateurs récents (gcc **4.7** par exemple) qui ne se trouvent pas sur toutes les machines, en particulier sous Mac OS X qui ne dispose que de gcc **4.2** pour sa version la plus récente.

Enfin, la seconde contrainte était d'être le moins dépendant possible, si ce n'est pas du tout, de l'implémentation autres modules afin de faciliter la maintenance et l'évolution du module. Comme expliqué plus loin, cette contrainte n'a pu être que partiellement respectée au vue de difficultés techniques liées au design des autres modules.

Enfin, une contrainte de qualité était fixée, avec notamment un taux de couverture ne descendant pas en dessous de **70** à **75**%, impliquant des jeux de tests exhaustifs.

6.4 Approche générique

6.4.1 Différents types d'algorithmes

Un des premiers problèmes apparu est la **diversité** des types d'algorithmes. Outre la hiérarchie des classes propres à l'implémentation dans **EO** ou **ParadisEO**, de manière théorique nous avons pu isoler deux types d'algorithmes différents par leur traitement sur la population.

La première catégorie va effectuer l'ensemble du traitement de l'algorithme sur un individu et boucler ce traitement sur tous les individus de la population. Ici, il est clair qu'il est intéressant de paralléliser la boucle principale qui comporte tout le traitement relatif à un individu.

Le second type d'algorithme va cette fois appliquer une suite de traitements à l'ensemble de la population à la fois, donnant lieu à une nouvelle génération d'individus. Le nombre de générations provient des « continators » et, les bornes sont données par un fichier en paramètre.

L'objectif étant que tout soit transparent pour l'utilisateur, il fallait donc déterminer le type d'algorithme pour ensuite lui affecter la bonne stratégie de parallélisation.

Voici un type de solution proposée :

```
1 if(typeid(EOAlgo<EOT>) == typeid(eoEasyEA<EOT>)
2     || typeid(EOAlgo<EOT>) == typeid(eoSGA<EOT>))
3     paraType = EVAL_LOOP;
4 else if(typeid(EOAlgo<EOT>) == typeid(eoEasyPSO<EOT>)
5     || typeid(EOAlgo<EOT>) == typeid(eoSyncEasyPSO<EOT>))
6     paraType = ALGORITHM;
7 else
8     throw std::runtime_error("Unknow algorithm");
```

Listing 6.1– Selection d'algorithmes en RT avec typeid

EOAlgo est un template du wrapper qui représente l'algorithme dont le wrapper hérite. **EOT** est le template représentant l'individu composant les populations.

Cette méthode possède plusieurs inconvénients. Tout d'abord elle est peu maintenable : elle oblige la personne voulant rajouter un algorithme à modifier une partie critique du module, en l'occurrence le **maitre**. Ajoutons également qu'elle est peu lisible et accroit donc d'autant la difficulté de maintenabilité. Enfin, le principal inconvénient est qu'elle ne respecte pas le principe de substitution de Liskov (LSP)
Une solution plus élégante sera proposée par la suite pour un problème similaire.

6.4.2 Modifier le comportement de l'algorithme

L'objectif de transparence pour l'utilisateur doit lui permettre d'utiliser ses algorithmes de manière naturelle, en se référant à l'API spécifique à l'algorithme, et non pas une API supplémentaire fournie par le module de parallélisation.

Ainsi, après avoir récupéré la stratégie de parallélisation à effectuer en fonction du type d'algorithme dont notre wrapper hérite, il faut redéfinir les mécanismes entrant en jeu dans ce type de parallélisation.

Le concept est simple. Dans le premier cas, où l'on doit paralléliser l'algorithme appliqué à un individu, il faut redéfinir l'**opérateur ()** (c'est lui qui lance l'algorithme, quelque soit l'algorithme).
Dans le second cas, où l'on doit uniquement paralléliser l'évaluation qui est faite d'un coup sur la population, à chaque génération, on doit « intercepter » le foncteur qui se charge d'évaluer l'ensemble de la population, et le wrapper pour qu'il soit utilisé de manière transparente par l'algorithme (le wrapper intégrant évidemment le **scheduler** pour la répartition entre les différents theads). Ce second cas, comme nous le verrons, a été bien plus problématique.

LSP : Si $q(x)$ est une propriété démontrable pour tout objet x de type T, alors $q(y)$ est vraie pour tout objet y de type S tel que S est un sous-type de T.

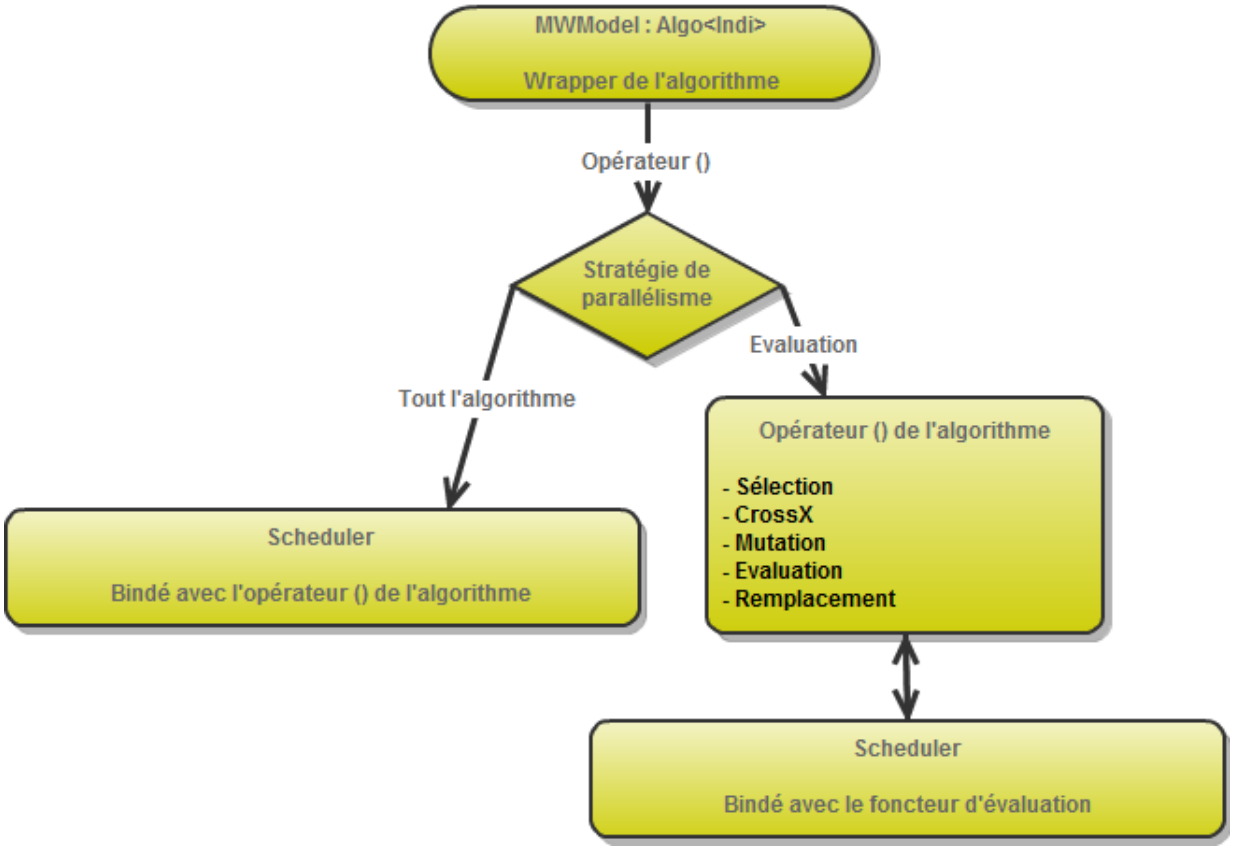

FIGURE 6.1 – Flowchart du déroulement du wrapper de l'algorithme

Voici un exemple schématique de l'**operateur ()**, qui permettrait de lancer l'algorithme wrappé :

```
1  template<template <class> class EOAlgo, class EOT>
2  void paradiseo::smp::MWModel<EOAlgo,EOT>::operator()
3  {
4      if(paraType == EVAL_LOOP)
5      {
6          // Appel de l'operateur () de la classe parente
7          EOAlgo<EOT>::operator()(pop);
8      }
9      else // Le foncteur d'evaluation est deja wrappe
10     {
11         // Appel de l'operateur () redefini
12         this->operator()(pop);
13     }
14 }
```

Listing 6.2– Sélection de l'opérateur () en fonction du type d'algorithme

6.4.3 Difficultés techniques et abandon de l'approche

Le principal foncteur à intercepter était donc le foncteur d'**évaluation globale** de la population. Or, là où le foncteur d'**évaluation d'un individu** est passé au constructeur de l'algorithme, le foncteur d'**évaluation globale** est construit à partir du foncteur d'**évaluation d'un invividu**, à l'intérieur du constructeur de l'algorithme.

Il est donc impossible de l'intercepter comme on pouvait le faire avec le foncteur d'évaluation pour utiliser un foncteur wrappé.

Illustrons le problème par le code suivant :

```
1  template<template <class> class EOAlgo, class EOT>
2  class MWModel : public EOAlgo<EOT>
3
4  ...
5
6  template<template <class> class EOAlgo, class EOT>
7  template<class... Args>
8  paradiseo::smp::MWModel<EOAlgo,EOT>::MWModel (unsigned workersNb, Args
       &... args ) :
9      EOAlgo<EOT>(args...),
10     scheduler(workers)
11 {
12     this->popLoopEval = wrappedPopLoopEval;
13 }
```

Listing 6.3– Constructeur du wrapper d'algorithmes

Notre wrapper hérite publiquement de l'algorithme. Nous pouvons voir que le constructeur prend en paramètre un nombre quelconque d'arguments via un « template variadic ». Cela permet de factoriser l'ensemble des déclinaisons des constructeurs de l'algorithme.
Le constructeur du wrapper fait tout de suite appel au constructeur de la classe parente en lui passant le « parameters pack », pour constuire la partie relative à l'algorithme. Il est indispensable de le faire à ce moment là sans quoi on ne pourrait de toute manière pas construire l'objet.

Le problème provient d'affectation dans le constructeur. En effet, notre foncteur wrappé dérive nécessairement du type de **popLoopEval** pour lui rajoute des traitements spécifiques (appel au **scheduler**). Seulement, la sémantique de copie n'est pas compatible avec l'héritage public et par transtypage implicite puis affectation (via l'opérateur d'affectation par défaut), notre foncteur wrappé perd ses spécificités pour redevenir un foncteur du type de base.
Ne pouvant pas intercepter le foncteur à la création, nous nous retrouvions dans une impasse qui nous a fait renoncer à une approche générique de l'implémentation de l'algorithme. Notons que tous les passages de foncteurs se font par référence et que pour pouvoir passer des pointeurs il faudrait modifier en profondeur **EO**, ce qui n'est évidemment pas possible.

6.5 Approche spécifique

6.5.1 Structure générale

L'approche qui a été retenue est celle qui consiste à réécrire l'algorithme en modifiant les parties à paralléliser. On perd l'abstraction que l'on pouvait avoir avec l'approche précédente et cette solution n'est que partiellement satisfaisante. Cependant, aucune autre solution n'a été trouvée ou n'était permise par le design de **EO**.

6.5.2 Tag Dispatching

Pour choisir le bon algorithme en fonction du type dont hérite notre wrapper, nous pouvons tester son type en **runtime** avec la solution proposée plus haut, à base de conditionnelles et de « typeid ». Comme évoquée, cette solution est peu élégante et c'est pourquoi j'ai mis en place un système de « Tag Dispatching [**?**] », qui va permettre en « compile time » de résoudre les fonctions qui seront utilisées, tout en supprimant les conditionnelles.

Le « Tag Dispatching » consiste à créer un tag par algorithme, qui n'est autre qu'une structure totalement vide, comme suit :

```
1 struct error_tag {};
2 struct eoEasyEA_tag {};
3 etc.
```

Listing 6.4– Exemples de structures vides servant de tag

On définit ensuite nos classes de traits, sans oublier une classe non spécialisée dans le cas où l'algorithme utilisé n'est pas répertorié :

```
1  template<template <class> class A, class B>
2  struct traits {
3      typedef error_tag type;
4  };
5
6  template<class B>
7  struct traits<eoEasyEA,B> {
8      typedef eoEasyEA_tag type;
9  };
10 etc.
```

Listing 6.5– Définitions des classes de traits

Enfin, dans notre wrapper, nous définissons les différentes implémentations des algorithmes et une méthode publique permettant d'être appelée par l'utilisateur :

```
1  public:
2      /**
3       * Run the algorithm on population
4       */
5      void operator()(eoPop<EOT>& _pop);
6
7  protected:
8      /**
9       * Specific algorithm for eoEasyEA
10      */
11     void operator()(eoPop<EOT>& _pop, const eoEasyEA_tag&);
12
13     /**
14      * If we don't know the algorithm type
15      */
16     void operator()(eoPop<EOT>& _pop,const error_tag&);
```

Listing 6.6– Prototypes des différents opérateurs ()

L'astuce réside dans le tag passé en paramètre qui permet en fonction de la spécialisation du wrapper d'appeler la bonne méthode. Voici la méthode publique :

```
1 void MWModel<EOAlgo,EOT>::operator()(eoPop<EOT>& _pop)
2 {
3     // Call the tag dispatcher
4     operator()(_pop,typename traits<EOAlgo,EOT>::type());
5 }
```

Listing 6.7– Opérateur () qui appelle le bon opérateur () privé en fonction du template

Le code est donc d'autant plus simple que la méthode publique n'est constituée que d'une instruction. Le coût de l'opération est nulle puisque les **tags** sont des structures vides et ne sont jamais utilisées. Elles seront donc supprimées du binaire par le compilateur qui effectue ce genre d'optimisations automatiquement.

Le problème inhérent au traitement spécifique des algorithmes, c'est qu'avec l'ajout d'algorithmes, c'est le wrapper qui s'allonge inéxorablement. En effet, la fonction spécifique à un algorithme tourne autour des **40** à **50** lignes, et l'on retrouve une bonne vingtaine d'algorithmes entre ceux de **EO**, **MO** et **MOEO**.

Pour rendre le code plus lisible et maintenable, j'ai effectué une découpe peu orthodoxe qui consiste à avoir un sous-répertoire **MWAlgo** proposant un fichier par fonction spécifique à un algorithme. Ces fichiers sont regroupé dans un **header** lui même inclu depuis le wrapper (c'est le wrapper qui inclu puisque toutes ces fonctions utilisent des templates).

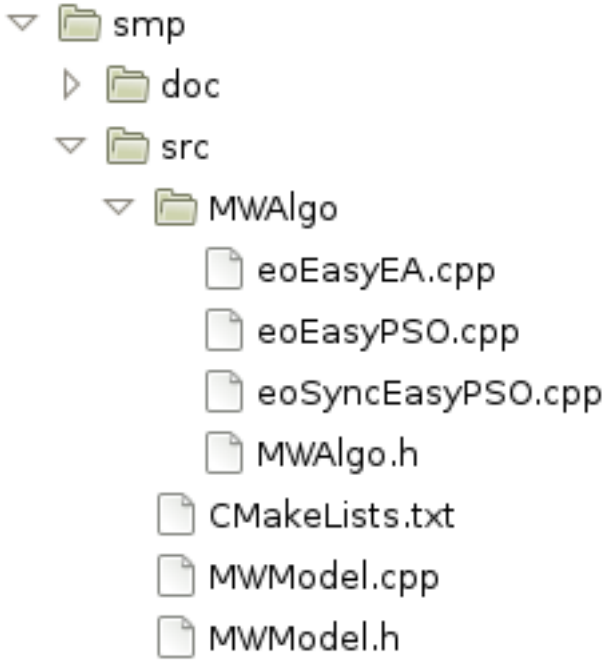

FIGURE 6.2 – Fichiers contenant les algorithmes

6.5.3 Le maitre

Le **maitre** correspond au wrapper de l'algorithme. Il s'occupe de créer l'algorithme à partir des paramètres fournis à son instanciation. Il va également grâce au « tag dispatching » choisir la méthode à lancer.
Il possède un **scheduler** qu'il va instancier avec le nombre de **workers** choisi par l'utilisateur, ou déterminé par la fonction « hardware_concurrency » qui permet de retourner le nombre de coeurs du processeur. Le **maitre** s'occupe simplement d'appeler le **scheduler** pour paralléliser une instruction particulière depuis n'importe où. Concrètement, il suffit de passer un foncteur et la population sur laquelle il va s'appliquer afin de lancer le **scheduler**.

6.5.4 Le scheduler

Le principe

Le **scheduler** est la partie la plus importante puisque c'est lui qui va coordonner le travail des différents threads.

Lors de l'appel à l'**opérateur ()**, les threads sont lancés, avec pour tâche à exécuter, la méthode **apply** du **scheduler** à qui sont passés en paramètre le foncteur et la population sur laquelle l'appliquer.
Ensuite, en fonction de la politique de découpe, des sous-paquets d'individus sont formés et envoyés sur les threads qui s'occupent de les traiter. Enfin, le **scheduler** attend la fin du travail des threads. En effet, si l'on prend le cas d'un algorithme classique de **EO**, les opérations dépendent des précédentes. Ainsi, il faut attendre la fin de l'évaluation de toute la population pour pourvoir entamer la sélection d'individus, sans quoi une exception est levée.

Pour déterminer la politique à adopter, une simple conditionnelle aurait pu suffire. Seulement, étant donné le nombre potentiellement très important de générations et donc d'appels au **scheduler** (sachant qu'il peut y en avoir plusieurs par génération) cela aurait rajouté un coût non-négligeable au traitement. Ainsi, comme pour le **maitre**, un système de « tag dispatching » permet de choisir sa **politique** grâce aux « type traits » en **compile time**. Cela permet de n'avoir aucun coût supplémentaire.

La politique par défaut, au vue des benchmarks, est la politique **linéaire** et l'utilisateur peut changer au besoin.

Politique linéaire

Une des politiques applicable à la gestion des threads est une découpe linéaire de la population. A la réception par le **scheduler** de la population, celui-ci la divise en autant de **paquets égaux** qu'il y a de threads, et les envoie pour être traités.
C'est la politique la plus simple à mettre en place mais elle possède le gros désavantage de ne pas tenir compte la **charge des processeurs** utilisés. On peut imaginer qu'un des processeurs est surchargé. Ainsi, les threads travaillant dessus mettent beaucoup de temps à terminer leur paquet, bien plus que les autres. Sachant qu'il faut attendre la fin de tous les threads, le thread le plus lent détermine les performances globales de l'algorithme et un processeur surchargé peut ainsi faire baisser terriblement l'intérêt de cette découpe.

Politique progressive

Pour pallier au problème précédent, on peut envisager une politique qui va charger petit à petit et de plus en plus les threads, de manière individuelle. Lorsqu'un thread a terminé un paquet, le **scheduler** lui renvoie un paquet plus gros à traiter. En cas de ralentissement d'un processeur on limite ainsi considérablement l'effet sur les performances puisque les threads tournant sur d'autres coeurs assurent le reste du traitement.

Concrètement, la boucle principale tourne avec un compteur d'individu qui indique le nombre d'individus envoyés à l'évaluation. Lorsque ce nombre atteint la taille de la population, on sort de la boucle et on **synchronise** les threads pour les arrêter. A chaque tour de boucle, le **scheduler** regarde, pour chaque worker, le nombre d'invidus qu'il lui reste à traiter. Si ce nombre est nul, le **scheduler** acquière un « lock », remet des individus en accord avec un **planning** interne dans le « vector » de travail du thread, modifie la valeur du **planning** et relache ensuite le « lock ».

La fonction apply

La fonction **apply** du **scheduler** représente la fonction exécutée par le thread. Elle dépend donc de la **politique** adoptée.

Le fonctionnement **apply** est très simple et repose sur un même principe. Une boucle principale tourne jusqu'à l'évaluation de toute la population. A l'intérieur, plusieurs conditions se succèdent. Si le « vector »d'évaluation est vide, si la politique est linéaire, on sort de la fonction, si c'est elle est progressive, on attend que le **scheduler** remette des individus à traiter.

Améliorations

Le point commun de tous les foncteurs de **EO** est de s'appliquer à une population. Les autres arguments, dans le cas de foncteurs non-unaires, peuvent être déterminés à l'endroit même où l'on appelle le **scheduler** et sont donc indépendants de celui-ci. Le **scheduler** appliquant juste une découpe de la population pour appliquer le foncteur sur des sous-groupes. Une idée intéressante pour la généricité du scheduler aurait été d'utiliser les fonctionnalités du C++ 2011 héritées de **Boost** que sont les `std::function` et `std::bind` (avec les `std::placeholders`). Les `std::function` permettent de répésenter n'importe quel type de fonction. On lui donnera alors à la volée une fonction particulière avec `std::bind` (dont les arguments peuvent être eux-même fixés ou laissés libres, avec les `std::placeholders`). Ainsi, nous aurions pu avoir un unique **opérateur ()** pour le scheduler qui aura prit en paramètre un `std::function` avec pour signature `void (eoPop)`, et une population **eoPop**.
Dans l'algorithme, si un foncteur était autre que unaire, il aurait pu binder le ou les arguments manquants avant d'être passé au **scheduler** en ayant alors une signature conforme.

Le soucis vient du fait que les foncteurs de **EO** dérivent tous d'une classe abstraite et qu'il est donc impossible de les binder. De plus, ces foncteurs sont templatés, ce qui fait qu'ils ne peuvent être bindés pour cette raison.

6.6 Tests

6.6.1 Tests unitaires

Etant donné le peu de classes disponibles, les **tests** furent assez rapides à mettre en place pour la base du module. Le test de la classe de **Thread** s'assure que le wrapper respecte toujours les **barrières** et lance quelques fonctionnalités basiques. Le test du **Scheduler** construit une population dont les individus ont la particularité d'avoir un compteur du nombre d'évaluation dont ils ont été l'objet. Ainsi, le test lance une évaluation complète de la population et une assertion vérifie que chaque individu a été évalué **une et une seule fois**.

Etant donnée l'approche spécifique à chaque algorithme, il nous fallait tester chaque algorithme sur un exemple concret. Dans chacun des cas il fallait donc créer un algorithme, ce qui s'avère assez long et fastidieux, surtout lorsque l'on ne connait pas nécessairement le principe de l'algorithme et qu'aucun document autre que l'API n'aide à sa compréhension. Pour ces tests, la graine de la fonction aléatoire était fixée afin d'obtenir toujours la même population initiale, et, normalement, toujours la même population finale. Une assertion est présente après le lancement de l'algorithme pour vérifier que le résultat obtenu par l'algorithme parallélisé correspond à la valeur obtenue, pour la même graine, par l'algorithme sérialisé.

```
Test project /home/aquemy/dev/paradiseo/git/build/smp
    Start 1: t-smpThread
1/5 Test #1: t-smpThread .....................   Passed    0.02 sec
    Start 2: t-smpScheduler
2/5 Test #2: t-smpScheduler ..................   Passed    0.03 sec
    Start 3: t-smpMW_eoEasyEA
3/5 Test #3: t-smpMW_eoEasyEA ................   Passed    0.08 sec
    Start 4: t-smpMW_eoEasyPSO
4/5 Test #4: t-smpMW_eoEasyPSO ...............   Passed    0.04 sec
    Start 5: t-smpMW_eoSyncEasyPSO
5/5 Test #5: t-smpMW_eoSyncEasyPSO ...........   Passed    0.01 sec

100% tests passed, 0 tests failed out of 5
```

FIGURE 6.3 – Résultats des tests unitaires

6.6.2 Tests de couverture

La contrainte de couverture a largement été respectée comme en témoigne ce rapport de lcov :

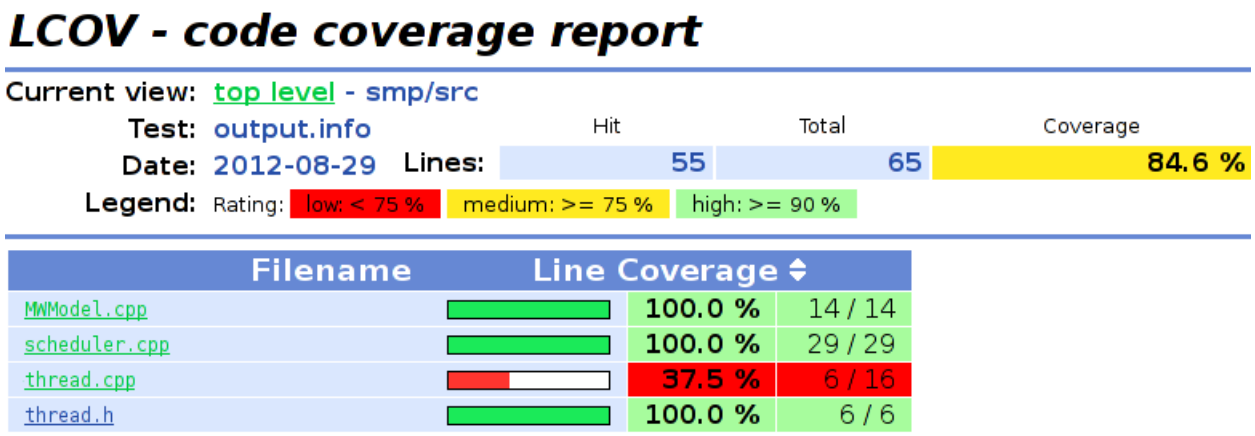

FIGURE 6.4 – Résultats des tests de couverture

Comme nous pouvons le voir, la classe **Thread** n'est testé qu'à **35%** ce qui provient de la volonté de ne pas créer de tests inutiles pour des choses très simples. En l'occurrence, cette classe est juste un wrapper pour s'abtraire de la technologie choisie : les `std::thread`.
A contrario, le taux de couverture est optimal pour tous les autres fichiers du module, ce qui compense largement en faisant rester le taux de couverture au dessus des exigences.

6.7 Documentation et exemples

Comme pour les autres modules, la documentation a été générée à partir de **Doxygen** puis mise en ligne sur la forge **Inria**.
Outre la mise à jour des procédures d'installation, il a fallu rédiger des articles sur l'utilisation du module **SMP**.
Un premier présente le module de manière générale, le second présente l'installation du module, et enfin, une leçon vient expliquer le fonctionnement de modèle **Maitre / Esclaves** implémenté, de manière générique puis de manière pratique sur le problème de l'affectation quadratique.

Les articles sont disponibles aux adresses suivantes :
- http ://paradiseo.gforge.inria.fr/index.php ?n=Main.SMP
- http ://paradiseo.gforge.inria.fr/index.php ?n=Doc.TutoSMPLesson1
- http ://paradiseo.gforge.inria.fr/index.php ?n=Doc.InstallSMP

6.8 Benchmark et speedup

6.8.1 Méthodologie

Pour réaliser des tests de performances et constater le **speedup**, j'ai créé un exemple minimal d'algorithme du type **eoEasyEA**. Différents paramètres sont à prendre en compte : nombre d'individus dans la population, nombre de générations effectuées par l'algorithme, nombre de threads utilisés par le programme, politique de découpe de la population, nombre de coeurs physiques de la machine, et temps d'évaluation d'un individu.

Pour tenir compte de tous ces paramètres, j'ai créé quelques **scripts bash** permettant de récupérer un grand nombre de mesures. Il a ensuite fallu que je l'effectue sur des machines à l'architecture différente, notamment par l'utilisation de **Grid 5000**.

A partir des différents temps d'exécution et speedup calculés, il est possible de tracer des courbes et des map grâce à **Gnuplot** en fonction du temps et des paramètres choisis pour mettre en évidence leur implication dans la performance de l'algorithme.

6.8.2 Grid 5000

Présentation

Grid 5000 est une grille de calculs intensifs française dédiée à la recherche.
La particularité des grilles de calculs est que leurs ressources sont **hétérogènes**, c'est à dire de nature différente (OS et matériel), **distribuées** sur différents sites géographiques.
Grid 5000 est composé de clusters répartis sur **13** sites : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia, Toulouse, Reims, Porto-Alegre (Brésil) et Luxembourg, pour un total de plus de **7900** coeurs.

Utilisation

L'accès à **Grid 5000** se fait par **SSH**, à une machine dite frontend, qui permet ensuite de se connecter au site souhaité, toujours par **SSH**. Un gestionnaire de tâches et de ressources nommé **OAR** permet de faire des réservations de noeuds suivant différents paramètres (caractéristiques des noeuds, temps de réservation, etc.).
Un mode déploiement permet de rebooter l'ensemble des noeuds réservés sous une image personnalisée, permettant d'avoir un environnement de travail personnalisé et surtout, avec les droits administrateurs.

Pour les besoins des **benchmarks** j'ai donc créée une image personnalisée de **Debian Squeeze**, en utilisant les dépôts de la version béta **Debian Wheezy** pour backporter gcc et g++ en version **4.7**, en installant également **ParadisEO** et en ajoutant des scripts pour les **benchmarks**.
J'ai ensuite pu déployer l'image sur des machines bi-processeur équipés de **AMD Opteron 6164 HE**, pour un total de **24** coeurs physiques.

6.8.3 Résultats et commentaires

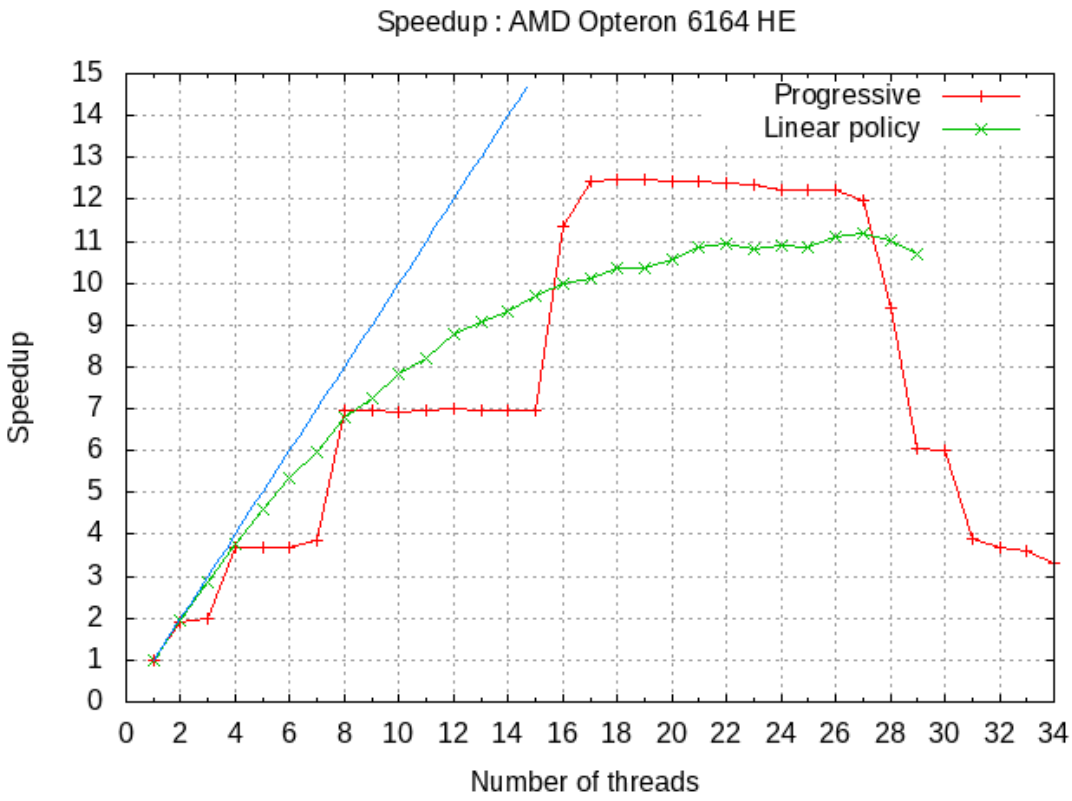

FIGURE 6.5 – Benchmark sur Grid 5000

Sur ce graphique nous pouvons observer l'évolution du **speedup** en fonction du nombre de workers et de la politique de **scheduling**. La droite en bleue représente l'identité, la limite théorique possible pour le **speedup**.
La principale différence que l'on peut observer entre les deux politiques et que la politique progressive fonctionne par **palliers** alors que la linéaire fait état d'une courbe plus **régulière**. Même si le linéaire semble souvent au dessus du progressif, ce dernier permet en **speedup** supérieur, avec environ **12.5** comparé à **11** du linéaire. L'efficacité observée pour le **speedup** le plus élevé est d'environ **72%** obtenu en progressif avec **17** threads.

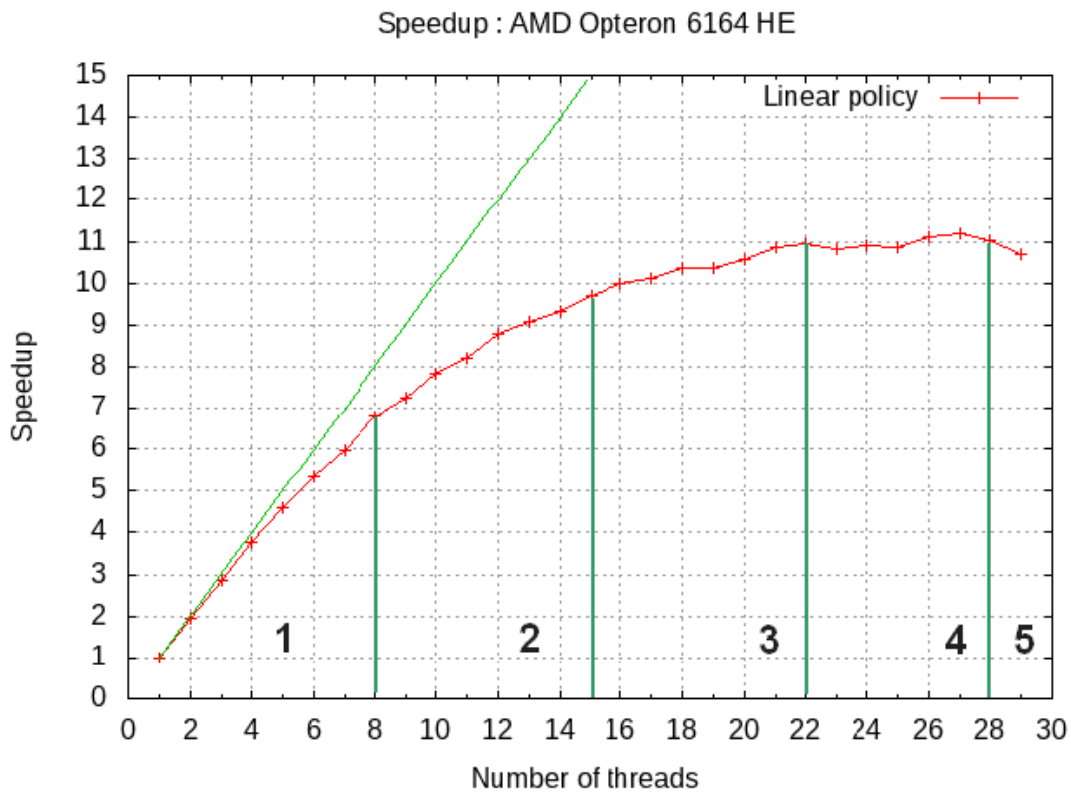

FIGURE 6.6 – Benchmark sur Grid 5000

Ce second graphique est le même que le premier, si ce n'est qu'il n'a d'affiché que la politique linéaire. On peut distinguer plusieurs phases.

1. Speedup linéaire et donc idéal
2. Dégradation du speedup lié à la portion de code parallélisé
3. Dégradation supplémentaire, probablement liée à la cohérence du cache (car la machine est bi-processeur)
4. Saturation lié au nombre de threads disponibles, et donc stagnation du speedup
5. Dégradation du speedup lié au nombre de threads disponibles et au surcoût des communications

Ces tests ne sont pas nécessairement très représentatifs puisque pas assez nombreux et exhautifs. De même, le matériel testé n'est pas forcément très courant et peut influer sur la manière dont l'OS le gère et gère les communications entre les CPU.
Il aurait été intéressant de pouvoir tester sur des machines au matériel plus commun, comme des **i7** ou **i5**. Ceci n'a pu être fait faute de temps, mais espérons que le retour des utilisateurs de **ParadisEO** permettra de quantifier le **speedup** de manière plus précise.

Chapitre 7

Conclusion et bilan du stage

7.1 Bilan du stage

Le sujet de stage tel qu'il était formulé sur la convention de stage abordait uniquement la première partie du stage, à savoir la mise en place d'outils d'ingénierie logicielle pour **ParadisEO**, établir un nouveau système de construction logicielle et d'intégration continue. De ce point de vue là, le travail apporté a été satisfaisant puisque **ParadisEO** a pu passer à une version stable, en proposant une installation facilité pour tous les utilisateurs, quelque soit le système d'exploitation. Un travail de documentation a également été fait pour répondre aux besoins des utilisateurs en terme d'installation mais également d'utilisation de **ParadisEO**. Le travail sur l'évolutivité du système de **build** et d'intégration continue permettra facilement la fusion entre **ParadisEO** et **EO**.

La seconde partie, concernant le parallélisme fut, à mon sens la plus intéressante. Le module **SMP** a pu être intégré dans la version stable de **ParadisEO** et sera très certainement utilisé rapidemment puisque l'accent a été mis sur la simplicité d'utilisation.
On peut regretter le manque de temps, dû à deux à trois semaines de flottement entre les deux parties (pour les raisons évoquées dans le rapport) qui n'a permis d'intégrer qu'un modèle de parallélisme : le modèle maitre / esclave. Cependant, les tests de performances effectués sont plutôt corrects même s'ils gagneraient à être plus exhaustifs.

Il me semble donc que les objectifs du stage sont pleinement atteints et que l'intérêt du travail effectué sur **ParadisEO** est indéniable.

7.2 Apports techniques

Les apports techniques ont été très nombreux. Tout d'abord en génie logiciel, ce fut l'occasion d'approfondir mes connaissances tout en maitrisant des outils très utilisés. L'intérêt essentiel du stage est qu'il m'a permis d'appliquer des méthodologies et des démarches sur un projet d'envergure, en étant confronté aux difficultés liées à ce genre de projet : d'une part des problèmes liés au code lorsque l'on développe en passant derrière des équipes qui se sont succédées pendant plusieurs années, et d'autre part l'importance et la difficulté de la mise en place d'une politique de qualité du code en passant par l'intégration continue, un système de build performant, etc.

Parmi les outils utilisés et je pense maitrisés, on peut citer la suite d'outils gravitant autour de **CMake** (CMake, CPack, CDash et CTest), ainsi que dans une moindre mesure

Valgrind et **lcov**.

Ce fut également l'occasion d'être sensibilisé aux problématiques de portabilité et de support, notamment aux vues des difficultés rencontrées sous Windows ou Mac OS X, concernant le packaging, la compilation et l'utilisation de normes récentes. Il s'agissait souvent de compromis entre l'utilisation de technologies récentes mais non supportées par tous les OS et les parts d'utilisation non négligeables sous ces OS : Windows ou Mac OS X.

De manière marginale on peut noter des apports dans le domaine des système d'exploitation. En effet, n'ayant jamais (ou presque) touché de Windows ou de Mac OS X, ce fut l'occasion d'en apprendre un peu plus sur ces OS et leurs spécificités.

Enfin et non des moindres, la partie développement sur le module de parallélisation m'a enseignée beaucoup de choses sur des pratiques avancées du C++ [**?**], que ce soit idiomes [**?**] ou design pattern [**?**][**?**], la metaprogrammation [**?**] et évidemment, concernant la nouvelle norme [**?**][**?**] très riche en nouveautés. Ce module m'a également inité à la programmation paralléle à mémoire partagée via l'utilisation de processus légers [**?**]. Les nouveaux concepts à assimiler et les nombreux pièges de la concurrentes furent l'occasion d'expérimenter pas mal de prototypes afin d'arriver à un résultat satisfaisant, me laisser ainsi l'occasion de faire un tour assez exhaustif de ce que propose le C++ standard en matière de gestion des threads [**?**]. Ce fut l'occasion de pouvoir travailler sur une grille de calcul d'envergure grâce à **Grid 5000**, de mieux comprendre le fonctionnement d'un cluster et les problématiques liées au calcul intensif. Enfin, **Grid 5000**, m'a permis d'apprendre les rudiments de MPI et ses communications points à points (même si cela n'a pas servi pour le stage).

Pour finir, ce stage fut également, un peu à part de la mission, l'occasion d'en apprendre plus sur des domaines qui m'intéressent fortement : algorithmique et optimisation (recuit simulé, recherche tabou, etc.) ainsi que sur les algorithmes génétiques et évolutionnistes.

7.3 Impact sur mon projet profesionnel

Je recherchais particulièrement un stage dans le milieu de la recherche puisque c'est le domaine qui semblait m'attirer. Les discussions très intéressantes que j'ai pu avoir avec, à la fois des thésards, des ingénieurs recherches, des ingénieurs à cheval avec l'entreprise, et des chercheurs ont permis, à défaut de confirmer complètement mon projet professionnel, de faire ressortir un certain nombre de réflexions à mûrir. Notamment, la barrière qui semble se dresser devant un jeune docteur qui veut retourner dans le monde de l'entreprise me semble assez rebutante pour entreprendre moi même une thèse.

Ceci dit, le contexte général du monde la recherche m'a semblé intéressant et j'ai pu découvrir que ce milieu ne se limitait pas aux thésards et enseignants-chercheurs mais que les ingénieurs y avaient également leur place.

Finalement, je vais attendre de voir ce qui se fait à l'étranger, notamment avec un potentiel départ en 5ème année, afin d'arrêter mon choix sur une poursuite avec le diplôme d'ingénieur ou non.

Annexes

Grille de déroulement du stage

Semaine	Description des activités
Phase 1 : Stabilitation de ParadisEO	
Semaines 1	Découverte de ParadisEO Lecture des tutoriaux Ecriture des besoins fonctionnels Mise en place des outils (git, svn, etc.) Correction de code (this, include cstlib, etc.)
Semaines 2	Découplage de PEO, Old-MO Mise à plat / mise à niveau du CMake Intégration de la documentation Intégration de CDash
Semaines 3	Création de l'installation Intégration du Packaging Création du findParadiseo
Semaines 4	Correction des bugs Prise en main de Grid 5000 Rédaction du QuickGuide Mise à jour des tutoriaux sur le site Etablissement de la procédure de tests
20 juillet	Réunion à Palaiseau
Phase 2 : Parallélisme	
Semaines 5	Mise à jour du CMake de PEO Découplage PEO / OldMO Tentative de découplage distribuée / partagée Rétro-ingenierie sur PEO
Semaines 6	Décision de créer SMP Autoformation sur les threads standards Création du wrapper
Semaines 7	Création du scheduler Création de la politique linéaire Création du système de build de SMP
Semaines 8	Création de la politique progressive Benchmark sur Grid 5000
Semaines 9	Tests Création des paquets Mise à jour des pages du site Rédaction de la documentation SMP
Semaines 10	Correction de bugs mineurs Migration sous Git Tag de la version 2.0 Publication de la nouveau version de ParadisEO

Bibliographie

[1] Wikipedia, *Plan Calcul*.
http ://fr.wikipedia.org/wiki/Plan_Calcul
Valide au 16/08/12

[2] DOLPHIN Team, *Site dédié à l'équipe*.
http ://dolphin.lille.inria.fr
Valide au 16/08/12

[3] ParadiseEO, *Design concepts*.
http ://paradiseo.gforge.inria.fr/index.php ?n=Doc.Design
Valide au 16/08/12

[4] Kernel.org, *Manuel de git-svn*.
http ://www.kernel.org/pub/software/scm/git/docs/git-svn.html
Valide au 16/08/12

[5] GCC, *Vers gcc 4.7*.
http ://gcc.gnu.org/gcc-4.7/porting_to.html
Valide au 16/08/12

[6] Module MO, *Nouveau design*.
http ://paradiseo.gforge.inria.fr/addon/paradiseo-mo/concepts/paradiseo-newmo-design.pdf
Valide au 16/08/12

[7] NC State University, *Annonce d'un cluster PS3*.
http ://moss.csc.ncsu.edu/ mueller/cluster/ps3/coe.html
Valide au 16/08/12

[8] HPC Wire, *Air Force's Condor Cluster*.
http ://www.hpcwire.com/hpcwire/2010-12-03/air_forces_ps3_condor_cluster_takes_flight.html
Valide au 16/08/12

[9] Quemy Alexandre, *Tag Dispatching en C++*.
http ://aquemy.com/articles-tagdispatching.html
Valide au 28/08/12

[10] Stepanov Alexandre, *Notes.*
http ://www.stepanovpapers.com/notes.pdf
Valide au 16/08/12

[11] Wikibooks, *More C++ idioms.*
http ://en.wikibooks.org/wiki/More_C%2B%2B_Idioms
Valide au 16/08/12

[12] Wikibooks, *Design Patterns.*
http ://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns
Valide au 16/08/12

[13] Source Making, *Design Patterns, UML, Refactoring.*
http ://sourcemaking.com
Valide au 16/08/12

[14] Andrei Alexandrescu,
Modern C++ Design : Generic Programming and Design Patterns Applied.

[15] Herb Sutter, *Elements of modern C++ style.*
http ://herbsutter.com/elements-of-modern-c-style
Valide au 16/08/12

[16] Bjarne Stroustrup, *C++11 FAQ.*
http ://www.stroustrup.com/C++11FAQ.html
Valide au 16/08/12

[17] Just, *Multithreading in C++0x.*
http ://www.justsoftwaresolutions.co.uk/threading/
Valide au 16/08/12

[18] C++ documentation, *Multithreading in C++0x.*
http ://en.cppreference.com/w/cpp/thread
http ://en.cppreference.com/w/cpp/atomic
Valide au 16/08/12