

Université
de Toulouse

THÈSE

En vue de l'obtention du

Doctorat de l'Université de Toulouse

Délivré par l'Institut National des Sciences Appliquées de Toulouse
Spécialité : Systèmes Informatiques

Présentée et soutenue par Bernard Miegemolle

Le 11 septembre 2008

Prédiction de comportement d'applications parallèles et placement à l'aide de modèles économiques sur une grille de calcul

Jury

M. Emmanuel JEANNOT	Rapporteur
M. Jean-Louis PAZAT	Rapporteur
M. Thierry MONTEIL	Directeur de thèse
M. Christophe CHASSOT	Président
M. Jean-Marie GARCIA	Invité

Ecole Doctorale : Systèmes
Unité de recherche : LAAS - CNRS
Directeur de thèse : Thierry MONTEIL

Résumé

Une des solutions les plus prometteuses actuellement à la course à la puissance de traitement consiste à créer des grilles. Néanmoins, leur utilisation optimale n'est pas encore atteinte dû notamment à la complexité que ce support d'exécution amène pour les administrateurs et utilisateurs. Cette thèse se concentre sur la gestion des ressources composant une grille de calcul. Nous montrons comment traiter ce problème à l'aide de paradigmes économiques.

Nous définissons un modèle économique permettant de gérer les ressources d'une grille. Ce modèle propose d'associer un coût à chacune des machines de la grille. Le placement d'une application est assimilé à un problème d'optimisation non-linéaire sous contraintes et à variables entières, pour lequel le choix des machines à utiliser doit minimiser un compromis entre le temps d'exécution de l'application et son coût. Une implémentation de ce modèle à l'aide d'un algorithme génétique est proposée, de même que son intégration au sein de l'ordonnanceur OAR utilisé sur Grid'5000.

Dans une seconde partie de la thèse, des travaux ont été effectués dans le domaine de la prédiction du temps d'exécution d'une application. Nous définissons une méthode hybride de prédiction basée à la fois sur le profil des applications ainsi que sur un historique d'exécutions passées, combinant une analyse de la structure du programme à une méthode d'apprentissage basé sur des instances. Nous montrons notamment que la prise en compte du profil des applications améliore les prédictions réalisées au moyen de méthodes classiques basées seulement sur des historiques d'exécutions passées.

Mots-clés : Grilles de calcul, gestion de ressources, modèles économiques, prédiction de temps d'exécution, modèle hybride.

Abstract

One of the most promising solution to the challenge of processing power increase consists in creating Grids. Nevertheless, their use in an optimal way has not been reached because of complexity of such a support. This thesis focuses on one of the issues inherent in the grid concept, which is the grid resource management. We demonstrate that this problem can be dealt by applying economic paradigms from real-life to the resource management issue.

We define an economic model that enables to manage processors that belong to a computational grid. The purpose of this model is to associate a cost with each host. Scheduling can thus be assimilated to a non-linear optimization problem under constraints, with integer variables. The choice of executing hosts has to minimize a compromise between the application execution time and its cost. We propose an implementation of this model that uses a genetic algorithm. This implementation is finally integrated into the OAR scheduler that is used upon the French experimental research grid : Grid'5000.

In a second part of the thesis, we achieved some work in the domain of application execution time prediction. Indeed, an estimation of this time is necessary to the economic model. We define an hybrid method of time prediction that is both profile-based and historic-based. This prediction is achieved by combinating a program structure analysis to an instance-based learning method. We demonstrate that taking in account the application profile improves predictions that are carried out through classical historic-based prediction methods.

Keywords : Grid computing, resource management, economic models, execution time prediction, hybrid model.

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés au sein du Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS), une unité propre de recherche du Centre National de la Recherche Scientifique (CNRS). Je remercie Messieurs Malik Ghallab et Raja Chatila, directeurs successifs du LAAS-CNRS pour m'avoir accueilli au sein de ce laboratoire et mis à ma disposition les moyens nécessaires à la réalisation de mes travaux.

J'exprime ma très sincère reconnaissance à Monsieur Thierry Monteil pour m'avoir encadré et soutenu tout au long de cette thèse. Je le remercie vivement pour ses précieux conseils, ainsi que de la confiance dont il m'a toujours témoigné.

Je remercie tout particulièrement Messieurs Jean-Louis Pazat et Emmanuel Jeannot pour avoir accepté d'être les rapporteurs de cette thèse. J'exprime également ma profonde gratitude à Monsieur Christophe Chassot pour avoir accepté de présider mon jury de thèse, ainsi qu'à Monsieur Jean-Marie Garcia pour avoir accepté mon invitation à participer à ce jury.

Au cours de mes travaux, j'ai été amené à collaborer avec de nombreuses personnes. Je tiens à leur exprimer ici toute ma reconnaissance. Je remercie en particulier Samuel Richard, pour ses compétences techniques et sa disponibilité. Son aide m'a été d'un grand soutien. Je tiens également à remercier Rémi Sharrock, Fadi Khalil, Guillaume Evrard et Olivier Dusacq avec qui j'ai eu le plus grand plaisir à collaborer.

Mes remerciements vont également à l'ensemble des personnes que j'ai côtoyé au sein du LAAS, parmi lesquelles Cédric Fortuny, Sébastien Harasse, François Gaillard, Olivier Brun, David Gauchard, Charles Bockstal, Wael Suleiman, Cyril Figeac... Je ne peux malheureusement pas toutes les citer ici, et je m'excuse par avance auprès des absents.

Je tiens à exprimer mes plus profonds remerciements à toute ma famille qui m'a toujours été d'un grand soutien au cours de ces années de thèse, et de toutes celles qui les ont précédées. Je remercie en particulier mes parents pour leur soutien sans faille, dans les moments faciles comme dans les moments difficiles. Je remercie également ma sœur pour toute l'aide qu'elle m'a apporté, notamment lors de la relecture de ce mémoire.

Enfin, mes derniers remerciements vont à Erika, qui a su être présente à mes côtés jour après jour. *Gracias por estar conmigo, por la alegría que me das cada día, y por tu apoyo en mis momentos de duda.*

Table des matières

1	Introduction	21
1.1	Les grilles de calcul	21
1.1.1	Définition d'une grille de calcul	21
1.1.2	Caractéristiques d'une grille et de ses ressources	22
1.2	La plate-forme de recherche Grid'5000	23
1.2.1	Présentation générale	23
1.2.2	Architecture et réseau	23
1.2.3	Accès aux ressources	24
1.2.4	Principe d'utilisation de la grille	25
1.3	Problématique liée à la gestion des ressources	27
1.4	Plan de la thèse	28
2	Gestion des ressources d'une grille de calcul	29
2.1	Objectifs et challenges	29
2.1.1	Caractéristiques de l'ordonnancement sur grille	29
2.1.2	Missions d'un gestionnaire de ressources	30
2.2	Ordonnancement sur une grille de calcul	31
2.2.1	Applications	32
2.2.2	Ressources	32
2.2.3	Fonctions objectifs	33
2.2.4	Catégories d'ordonnancement	34
2.2.4.1	Placement statique et placement dynamique	34
2.2.4.2	Placement centralisé, décentralisé ou hiérarchique	34
2.2.4.3	Modèle coopératif et modèle non-coopératif	35
2.2.5	Stratégies d'ordonnancement	35
2.2.6	Introduction aux modèles économiques	38
2.2.6.1	Historique	38
2.2.6.2	Quelques exemples d'utilisation de modèles économiques .	39
2.2.6.3	Modèles économiques existants	40
2.3	Outils d'observation et de prédiction	41
2.3.1	MDS (Meta Directory Service)	41
2.3.2	NWS (Network Weather Service)	42
2.3.3	Ganglia	43
2.3.4	Monika et DrawOARGantt	43
2.4	Outils de placement et de gestion des ressources	45
2.4.1	Globus	45
2.4.2	AppLeS	45
2.4.3	Nimrod/G et le framework GRACE	47
2.4.4	Le gestionnaire de ressources OAR	48

2.5	Conclusion	50
3	Conception et implémentation d'un modèle économique	53
3.1	Introduction	53
3.1.1	Objectifs et contexte	53
3.1.2	Caractéristiques du modèle proposé	54
3.2	Modèle mathématique	55
3.2.1	Définition des variables utilisées	55
3.2.1.1	Aspects temporels	55
3.2.1.2	Données du problème	56
3.2.1.3	Inconnues recherchées	58
3.2.2	Modélisation sous forme de problème d'optimisation	59
3.2.3	Equations régissant le modèle	60
3.2.3.1	Relations entre les inconnues recherchées et contraintes	60
3.2.3.2	Quantité de calculs d'une tâche	60
3.2.3.3	Calcul du temps d'exécution de l'application	62
3.2.3.4	Calcul du coût de l'application	66
3.2.3.5	Influence des applications soumises sur la charge de la grille	68
3.3	Implémentation du modèle économique proposé par un algorithme génétique	71
3.3.1	Choix d'implémentation	72
3.3.2	Les algorithmes génétiques	72
3.3.3	Implémentation	74
3.3.3.1	Représentation des individus	74
3.3.3.2	Opérateurs de reproduction	75
3.3.4	Validation de l'algorithme mis en place	76
3.3.4.1	Etude comparative des résultats obtenus en mode statique	76
3.3.4.2	Etude des résultats obtenus en mode dynamique	83
3.3.5	Performances de l'algorithme	86
3.3.5.1	Convergence de l'algorithme	86
3.3.5.2	Temps de résolution	88
3.4	Intégration du modèle économique au sein d'OAR	89
3.4.1	Etude préliminaire	89
3.4.1.1	Spécifications	89
3.4.1.2	Incompatibilités entre le fonctionnement actuel d'OAR et le modèle économique	90
3.4.2	Politiques d'ordonnancement	91
3.4.3	Soumission d'applications	92
3.4.3.1	Mise en place d'un nouveau client	92
3.4.3.2	Traitement des paramètres du modèle	93
3.4.4	Mécanismes de placement des tâches	94
3.4.5	Vue d'ensemble du processus de soumission	96
3.4.6	Tests et validation	97
3.5	Conclusion	100
4	Etude de l'estimation du temps d'exécution d'une application	103
4.1	Introduction	103
4.2	Travaux portant sur l'estimation du WCET d'une application	104
4.2.1	Objectifs et contexte	104
4.2.2	Méthodes dynamiques d'analyse du WCET	105
4.2.3	Méthodes statiques d'analyse du WCET	106

4.2.3.1	Analyse de flot	106
4.2.3.2	Analyse de bas niveau	109
4.2.3.3	Calcul du WCET	110
4.3	Prédiction basée sur un historique d'exécutions passées	112
4.3.1	Principe général	112
4.3.2	Etat de l'art	112
4.3.3	Apprentissage basé sur des instances	114
4.3.3.1	Principe	114
4.3.3.2	Terminologie utilisée	114
4.3.3.3	Notion de distance entre deux expériences	115
4.3.3.4	Calcul de l'estimation du temps d'exécution	116
4.4	Approche hybride de prédiction de temps d'exécution d'une application	118
4.4.1	Principe général	118
4.4.2	Définitions et hypothèses	119
4.4.2.1	Hypothèses sur le temps d'exécution des blocs de base	119
4.4.2.2	Hypothèses sur le temps d'exécution des fonctions	120
4.4.3	Modèle mathématique	120
4.4.4	Mise en œuvre du modèle	121
4.5	Conclusion	122
5	Détermination du temps d'exécution des blocs de base d'un programme	123
5.1	Introduction	123
5.1.1	Objectifs	123
5.1.2	Principe	124
5.2	Exemple traité et faisabilité	124
5.2.1	Evolution du temps d'exécution du programme en fonction des entrées	124
5.2.2	Cohérence des temps obtenus et reproductibilité	125
5.2.3	Influence de l'utilisation de gprof et gcov sur le temps d'exécution du programme	126
5.3	Résolution du système d'équations à l'aide d'outils et de méthodes standards	129
5.3.1	Introduction	129
5.3.2	Calcul d'une estimation initiale de la solution	129
5.3.3	Résolution du problème sous la forme d'un système linéaire	130
5.3.3.1	Présentation	130
5.3.3.2	Résultats obtenus	131
5.3.4	Résolution du problème sous la forme d'un système non-linéaire	132
5.3.4.1	Présentation	132
5.3.4.2	Résultats obtenus	133
5.3.5	Conclusion	134
5.4	Résolution itérative du système d'équations	134
5.4.1	Introduction	134
5.4.1.1	Positionnement du problème	134
5.4.1.2	Caractéristiques du système d'équations à résoudre	135
5.4.2	Reformulation du système	135
5.4.3	Résolution du système par itérations	137
5.4.3.1	Définition de la suite	137
5.4.3.2	Convergence de la suite	137
5.4.3.3	Lien entre le conditionnement du système initial et la convergence de la suite	138

5.4.4	Résultats obtenus	140
5.5	Amélioration de la convergence de la résolution itérative du système d'équations	142
5.5.1	Introduction d'une erreur ε dans le système à résoudre	142
5.5.2	Reformulation du système à résoudre en prenant en compte l'erreur introduite	143
5.5.3	Résolution du nouveau système par itérations	144
5.5.3.1	Définition de la suite	144
5.5.3.2	Convergence de la suite	144
5.5.3.3	Choix de la matrice d'erreur	144
5.5.4	Résultats obtenus	145
5.6	Conclusion	147
6	Etude du comportement d'un programme	149
6.1	Introduction	149
6.2	Méthode d'estimation du nombre d'exécutions des blocs de base d'un programme	150
6.2.1	Modèle mathématique	150
6.2.2	Implémentation	152
6.2.2.1	Principe de fonctionnement	152
6.2.2.2	Phase d'apprentissage	153
6.2.2.3	Prédiction	154
6.2.3	Exemple traité	154
6.2.3.1	Présentation générale de l'application	154
6.2.3.2	Structure du programme	156
6.2.3.3	Entrées du programme	157
6.2.4	Résultats obtenus	159
6.2.4.1	Estimation du comportement du programme pour une requête donnée	159
6.2.4.2	Extension de l'estimation à un ensemble de requêtes	162
6.2.4.3	Influence du contenu de la base de connaissances	164
6.3	Prise en compte de l'impact relatif des entrées du programme sur le nombre d'exécutions des blocs de base	167
6.3.1	Positionnement du problème	167
6.3.2	Reformulation du modèle de prédiction	167
6.3.3	Annotations du code source d'un programme	168
6.3.3.1	Objectifs et principe	168
6.3.3.2	Mise en œuvre	168
6.3.4	Résultats obtenus	171
6.4	Modèle complet de prédiction hybride de temps d'exécution	172
6.4.1	Préambule	172
6.4.2	Prédiction du temps d'exécution d'un ensemble de requêtes	173
6.4.3	Variation du nombre d'expériences contenues dans la base de connaissances	173
6.5	Conclusion	174
7	Conclusion	177
7.1	Prédiction de comportement d'applications parallèles	177
7.2	Placement à l'aide de modèles économiques sur une grille de calcul	178
7.3	Vue d'ensemble	179

7.4	Perspectives	181
Annexes		185
A	Utilisation de l'adaptateur du client de soumission d'applications d'OAR	185
B	Stockage des paramètres nécessaires au modèle économique sous OAR	187
B.1	Paramètres stockés dans la base de données d'OAR	187
B.2	Paramètres stockés dans le fichier de configuration du module d'ordonnancement	188
C	Processus de soumission d'une application dans OAR et synchronisation du module d'ordonnancement	191
C.1	Processus de soumission	191
C.2	Synchronisation entre le module d'ordonnancement créé et le serveur OAR .	192
D	Principe de fonctionnement de l'outil de profiling gprof	197
D.1	L'instrumentation	197
D.2	L'échantillonnage	197
E	Exemple de l'élévation d'une matrice carrée à la puissance désirée	199
E.1	Code source original	199
E.2	Fichiers obtenus avec gprof et gcov	201
E.2.1	Résultat produit par gprof	201
E.2.2	Résultat produit par gcov	202
E.3	Analyse des résultats	203
E.3.1	Découpage en blocs de base	203
E.3.2	Expression du nombre d'exécutions des blocs de base en fonction des entrées	203
E.3.3	Dépendance entre les nombres d'exécutions des différents blocs de base	204
F	Notions mathématiques pour la résolution itérative d'un système d'équations linéaires mal conditionné	205
F.1	Valeurs propres et valeurs singulières d'une matrice	205
F.1.1	Valeurs propres	205
F.1.2	Valeurs singulières	205
F.2	Conditionnement d'une matrice	206
F.3	Décomposition en valeurs singulières	206
F.3.1	Propriétés des matrices U et V	207
F.3.2	Propriétés de la matrice Σ	207
Bibliographie		209

Table des figures

1.1	Architecture de Grid'5000	24
1.2	Conventions de nommage et accès aux ressources	25
1.3	Exécution de l'application de simulation électromagnétique de micro-systèmes complexes sur Grid'5000	27
2.1	Architecture d'un gestionnaire de grille	31
2.2	Le problème de l'ordonnancement	32
2.3	Catégories d'ordonnancements adaptés aux grilles de calcul	34
2.4	Modèle de files d'attente de l'ordonnancement avec qualité de service garantie	37
2.5	Architecture du <i>Meta Directory Service</i> de <i>Globus</i>	41
2.6	Structure du <i>Network Weather Service</i>	42
2.7	Architecture de <i>Ganglia</i>	43
2.8	Visualisation de l'état des ressources de Grid'5000 à l'aide de <i>Monika</i>	44
2.9	Diagramme de Gantt représentant les réservations de ressources dans le temps	44
2.10	Fonctionnement du gestionnaire de ressources de <i>Globus</i>	46
2.11	Structure des agents d'ordonnancement du système <i>AppLeS</i>	47
2.12	Hiérarchie modulaire d'OAR	49
2.13	Etapes réalisées lors de la soumission d'une application sur OAR	50
3.1	Quantité de calculs devant être effectuée par l'application	61
3.2	Détermination du temps d'exécution d'une tâche sur un processeur de puissance donnée et de charge nulle	61
3.3	Détermination du temps d'exécution d'une tâche sur un processeur de puissance donnée de charge non-nulle	62
3.4	Intervalles de temps nécessaires à l'exécution d'une tâche sur un processeur donné	63
3.5	Mise en séquence de trois tâches parallèles placées sur un même processeur	64
3.6	Enchaînement des dates de début et fin de trois tâches placées sur un même processeur	64
3.7	Ordonnancement choisi en mode statique pour minimiser le temps d'exécution de l'application sur des processeurs non-chargés	78
3.8	Charge appliquée aux processeurs	78
3.9	Ordonnancements choisis en mode statique pour minimiser le temps d'exécution de l'application sur des processeurs préalablement chargés	79
3.10	Ordonnancements choisis en mode statique pour minimiser le coût de l'application avec des coefficients locaux de coût neutres	80
3.11	Ordonnancement choisi en mode statique pour minimiser le coût de l'application avec prise en compte des coefficients locaux de coût	81

3.12	Ordonnancements choisis en mode statique pour minimiser un compromis entre le coût de l'application et sa date de terminaison (sans charge, coefficients locaux de coût neutres)	82
3.13	Ordonnancements choisis en mode statique pour minimiser un compromis entre le coût de l'application et sa date de terminaison (avec prise en compte de la charge et des coefficients locaux de coût)	83
3.14	Ordonnancement choisi en mode dynamique pour minimiser le coût de l'application	85
3.15	Ordonnancements choisis en mode dynamique pour minimiser le coût de l'application et sa date de terminaison	85
3.16	Convergence de l'algorithme génétique pour le calcul du placement d'une application de 20 tâches sur 30 processeurs	87
3.17	Convergence de l'algorithme génétique pour le calcul du placement d'une application de 40 tâches sur 1000 processeurs	88
3.18	Système de files d'attente géré par OAR	91
3.19	Structures des tables de la base de données d'OAR pour la gestion des files d'attente	92
3.20	Transmission des différents paramètres au modèle économique	93
3.21	Tables créées pour la transmission de paramètres à l'algorithme génétique .	94
3.22	Tables d'OAR contenant le placement des jobs et leur date de début	95
3.23	Soumission d'une application via l'adaptateur du client <code>oarsub</code>	96
3.24	Placement d'une application, avec $W^C = 100\%, W^T = 0\%$ (Exp. 1) et $W^C = 0\%, W^T = 100\%$ (Exp. 2)	98
3.25	Possibilités de placement de l'application entre Toulouse et Sydney	99
3.26	Choix de placement de l'application entre la grappe de Toulouse et celle de Sydney	99
4.1	Outils graphiques pour l'analyse statique du WCET	107
4.2	Fonctions de pondération	118
5.1	Temps d'exécution du programme <code>matrixPower</code> en fonction de ses entrées (vue 3D)	125
5.2	Temps d'exécution du programme <code>matrixPower</code> en fonction de ses entrées (vue 2D)	126
5.3	Ecart-type relatif à la moyenne des temps d'exécution du programme <code>matrixPower</code> en fonction de ses entrées	126
5.4	Temps d'exécution de l'application en fonction de ses entrées pour un Intel Xeon 5148 LV	127
5.5	Déférence relative entre les temps d'exécution avec et sans profiling pour un Intel Xeon 5148 LV	128
5.6	Comparaison entre le temps d'exécution estimé par <code>linprog</code> et le temps d'exécution réel du programme	132
5.7	Comparaison entre le temps d'exécution estimé par <code>fsolve</code> et le temps d'exécution réel du programme	133
5.8	Evolution de l'erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative non-améliorée	141
5.9	Comparaison entre le temps d'exécution estimé par la résolution itérative non-améliorée et le temps d'exécution réel du programme	141

5.10	Erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative améliorée en fonction du rayon spectral de la matrice d'itération choisi	146
5.11	Evolution de l'erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative améliorée pour différentes valeurs de $\rho(G')$	146
5.12	Comparaison entre le temps d'exécution estimé par la résolution itérative améliorée et le temps d'exécution réel du programme	147
6.1	Influence de la constante k sur la fonction de pondération	152
6.2	Structure de la base de données	152
6.3	Principe de fonctionnement du module de prédiction du comportement d'un programme	153
6.4	Evolution des particules au cours des phases de prédiction et de correction .	156
6.5	Etapes de l'algorithme de filtrage particulaire	156
6.6	Structure de l'application de filtrage particulaire	157
6.7	Temps d'exécution de chaque tâche esclave en fonction du nombre de particules et du nombre d'intervalles de temps considérés	158
6.8	Temps d'exécution de chaque tâche esclave en fonction du nombre total de tâches	159
6.9	Nombre d'exécutions de blocs de base en fonction du nombre de particules et du nombre d'intervalles de temps considérés	160
6.10	Exemple de nombre d'exécutions d'un bloc de base en fonction du nombre total de tâches	161
6.11	Distance séparant la requête de chaque expérience, et fonction de pondération associée	161
6.12	Comparaison entre le nombre d'exécutions des blocs de base réel et celui estimé par le modèle de prédiction	162
6.13	Nombre d'exécutions des blocs de base du programme pour les 500 jeux d'entrées considérés	163
6.14	Erreur relative sur l'ensemble des blocs de base du programme pour chaque requête	164
6.15	Erreur relative sur chaque série de 500 requêtes en fonction du nombre d'expériences présentes dans la base et du coefficient k	165
6.16	Coupes de la vue 3D	165
6.17	Domaine de valeurs de k offrant une précision correcte sur chaque série de requêtes	166
6.18	Représentation du creux d'erreur obtenu	166
6.19	Comparaison des erreurs obtenues sans et avec amélioration du modèle de prédiction	171
6.20	Erreur relative sur chaque série de 500 requêtes en fonction du nombre d'expériences présentes dans la base et du coefficient k pour la version améliorée du modèle de prédiction	172
6.21	Mise en œuvre de l'approche hybride de prédiction de temps d'exécution .	173
6.22	Temps d'exécution réel et prédit du programme pour différents jeux d'entrées	174
6.23	Erreur relative sur l'ensemble des prédictions (250 requêtes) en fonction du nombre d'expériences contenues dans la base de connaissances	175
7.1	Vue d'ensemble des réalisations	180
C.1	Machine à états du processus de soumission d'une application	191

Liste des tableaux

3.1	Opérateurs de reproduction utilisés	76
3.2	Modifications de la table <i>Queues</i>	92
3.3	Modifications de la table <i>Admission_rules</i>	92
4.1	Découpage de la fonction en blocs de base	107
5.1	Différences relatives entre les temps d'exécution obtenus avec et sans profiling pour diverses architectures	128
5.2	Données numériques relevées au cours de l'évaluation de la méthode itérative non-améliorée	140
6.1	Comparaison des temps d'exécution de la tâche maître et des tâches esclaves	157
6.2	Erreur relative sur l'ensemble des prédictions (250 requêtes) en fonction du nombre d'expériences contenues dans la base de connaissances	174
C.1	Actions à réaliser en fonction des valeurs contenues dans les tables <i>Jobs</i> et <i>Cost_scheduling</i>	193

Chapitre 1

Introduction

1.1 Les grilles de calcul

L'augmentation constante des besoins en termes de puissance de calcul informatique a toujours été un défi auquel la communauté informatique s'est confrontée. Le calcul scientifique ou financier, la modélisation ou bien la réalité virtuelle sont autant d'exemples pour lesquels il est nécessaire de disposer d'une grande puissance de calcul.

Même si les évolutions technologiques de ces dernières années ont permis d'aboutir à la création de machines de plus en plus puissantes, celles-ci ne parviennent toujours pas à fournir suffisamment de puissance pour effectuer des calculs d'une complexité trop élevée, ou traitant un trop grand nombre de données. Le calcul parallèle, ou distribué, est une réponse à ce problème [1]. L'idée est de répartir le calcul sur un ensemble de machines, reliées entre elles par des réseaux rapides, afin d'augmenter les performances pour l'effectuer.

La notion de grappe de calcul (ou *cluster*) a ainsi vu le jour. Il s'agit de regrouper un ensemble d'ordinateurs, qualifiés de noeuds de calcul, et de les connecter en vue de partager leurs ressources, et d'en permettre une gestion globale. Une grappe est généralement composée de machines homogènes, tant en termes d'architecture que de système d'exploitation, géographiquement proches (souvent situées dans la même pièce, voire dans une même armoire que l'on qualifie de *rack*), et reliées par des réseaux très rapides pouvant atteindre 10 Gb/s.

Depuis, des réseaux longues distances à haut débit ont été mis en place, avec notamment l'Internet grand public. Ce facteur, lié à l'explosion de la puissance des stations de travail à moindre coût pour le grand public, a permis l'émergence d'un nouveau concept : l'interconnexion des *clusters* aboutissant aux grilles de calcul.

1.1.1 Définition d'une grille de calcul

Une grille de calcul est une infrastructure matérielle et logicielle qui fournit un accès fiable, cohérent, pervasif¹ et peu onéreux à des ressources informatiques [2]. Le but est ainsi de fédérer des ressources provenant de diverses organisations désirant collaborer en vue de faire bénéficier les utilisateurs d'une capacité de calcul et de stockage qu'une seule

¹Un environnement pervasif est un environnement dans lequel les objets communicants se reconnaissent et se localisent automatiquement entre eux, sans action particulière de l'utilisateur.

machine ne peut fournir. Cependant, tout système informatique distribué ne peut posséder l'appellation de grille de calcul [3]. En effet, une grille est un système qui coordonne des ressources non soumises à un contrôle centralisé, qui utilise des protocoles et interfaces standards dans le but de délivrer une certaine qualité de service (en termes de temps de réponse ou bien de fiabilité par exemple).

Une analogie forte existe entre le développement des grilles de calcul et celui des grilles d'électricité (*power grids*) au début du XXème siècle [2, 4]. A cette époque, la révolution ne résidait pas en l'électricité elle-même, mais plutôt en la constitution d'un réseau électrique fournissant aux individus un accès fiable et peu onéreux à l'électricité, au travers d'une interface standard : la prise de courant. Les composants formant le réseau électrique sont hétérogènes, et la complexité induite est totalement masquée à l'utilisateur final. Ainsi, une grille de calcul possède les mêmes propriétés d'hétérogénéité des ressources la constituant et de transparence vis-à-vis de l'utilisateur final.

En outre, le calcul sur grille possède les objectifs suivants [5] :

- fournir une importante capacité de calcul parallèle,
- gérer des applications avec *deadline* proche,
- mieux répartir l'utilisation des ressources,
- exploiter les ressources sous-utilisées,
- accéder à des ressources additionnelles,
- assurer une tolérance aux fautes pour un coût moindre.

1.1.2 Caractéristiques d'une grille et de ses ressources

Le calcul sur grille consiste à fédérer des ressources de calcul et de stockage géographiquement réparties, en vue de permettre leur utilisation de manière transparente pour tout client de la grille. Nous allons maintenant examiner les caractéristiques principales d'une grille de calcul et des ressources qui la composent [6, 1] :

- **Hétérogénéité** : Une grille héberge des ressources matérielles et logicielles très différentes les unes des autres. Cette diversité doit être masquée à l'utilisateur de la grille, qui doit pouvoir accéder à ces ressources de manière totalement transparente. Ceci suppose la mise en place de protocoles de communication standardisés, et impose des contraintes de portabilité de code.
- **Partage de ressources** : Les ressources d'une grille peuvent appartenir à différentes organisations. Celles-ci peuvent choisir de partager l'ensemble des ressources de la grille entre elles, et également d'autoriser d'autres organisations à y accéder. Cela permet de mieux répartir la charge de travail, et d'exploiter les ressources sous-utilisées, promouvant ainsi l'efficacité et la réduction des coûts d'exploitation.
- **Multiplicité des domaines d'administration** : Chaque organisation peut mettre en place des politiques de gestion et de sécurité différentes, en termes d'accès au réseau, d'authentification ou bien encore de confidentialité. Les ressources doivent néanmoins être accessibles et utilisables par tous les clients de la grille. De plus, il est indispensable de fournir des méthodes d'administration particulières aux personnes en charge de la gestion globale de la grille.

- **Passage à l'échelle** : Le nombre de ressources constituant une grille peut varier d'une dizaine à plusieurs milliers. Ce problème de dimensionnement pose de nouvelles contraintes sur les applications et les algorithmes de gestion des ressources. Notons que le passage à l'échelle est également désigné par le terme de scalabilité (anglicisme issu du terme *scalability*).
- **Accès transparent, consistant et pervasif** : La grille doit être vue comme un unique ordinateur virtuel. La complexité de la plate-forme sous-jacente est masquée à l'utilisateur. De fait, la grille doit gérer les défaillances éventuelles des ressources, et s'adapter en permanence à l'environnement dynamique qui la constitue. L'accès aux ressources disponibles doit être garanti à l'utilisateur à tout instant.

1.2 La plate-forme de recherche Grid'5000

1.2.1 Présentation générale

Le projet national de recherche Grid'5000 [7, 8] propose la mise en place d'une grille de calcul expérimentale destinée aux chercheurs. Dix-sept laboratoires et unités de recherche participent à ce projet. Les machines constituant la grille sont réparties sur neuf sites en France : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse.

L'objectif de Grid'5000 est de fournir une plate-forme expérimentale pour tous les projets nationaux de recherche liés au domaine du calcul sur grille. Ainsi, les dernières innovations en termes d'intergiciels², d'applications massivement parallèles, ou bien encore de techniques de programmation adaptées aux grilles, peuvent être testées sur une plate-forme réelle.

La grille mise en place se doit donc d'être entièrement configurable et contrôlable par ses utilisateurs : les machines peuvent exécuter des systèmes d'exploitation et des environnements entièrement personnalisés, et peuvent même être redémarrées à distance. Des outils de réservation des noeuds de calcul ainsi que d'observation des ressources (*monitoring*) sont également disponibles.

1.2.2 Architecture et réseau

Les ressources de Grid'5000 peuvent être représentées sous la forme d'une structure hiérarchique, comme le montre la figure 1.1.

Ainsi, la plate-forme Grid'5000 est constituée de plusieurs sites. Chaque site possède un ou plusieurs *clusters* : il s'agit de groupes de machines ayant les mêmes caractéristiques matérielles, en termes de processeurs, de mémoire, et de capacité de stockage. Chaque machine peut contenir plusieurs processeurs (*CPUs*), eux-mêmes potentiellement constitués de plusieurs coeurs (ou *cores*), la plus petite unité de calcul indivisible disponible.

²Un intericiel (*middlewares*) est une classe de logiciels qui assure l'intermédiaire entre les applications et les systèmes d'exploitation présents sur les machines de la grille.

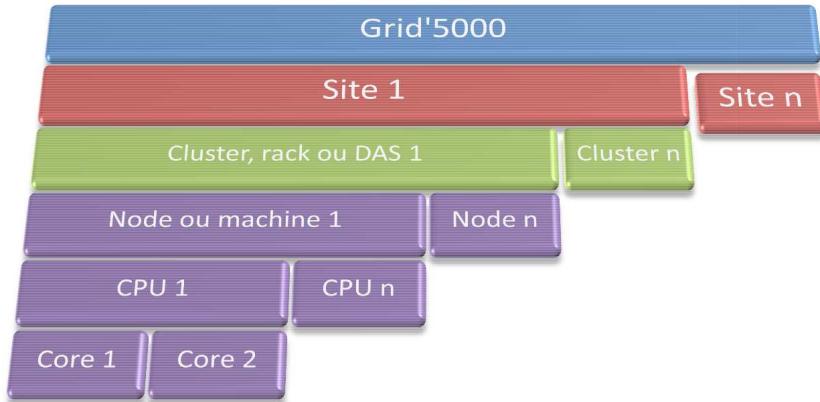


FIG. 1.1 – Architecture de Grid'5000

Les noeuds utilisés sont principalement de type *IBM eServer*, *HP ProLiant*, *Sun Fire*, *Dell PowerEdge* ou *Apple Xserve*. Ils sont en général dotés d'au moins 2 processeurs chacun. Trois grandes familles de processeurs équipent les noeuds : *AMD Opteron*, *Intel Xeon* et *PowerPC*.

Les différents sites de Grid'5000 sont interconnectés par le réseau RENATER, le Réseau National de télécommunications pour la Technologie, l'Enseignement et la Recherche [9]. Les liens connectant la plupart d'entre eux supportent des débits de 10 Gb/s. Si certains sites sont équipés d'un réseau interne multi-gigabit de type Myrinet 2G, Myrinet 10G, ou Infiniband, d'autres en revanche sont parfois limités au gigabit Ethernet.

Dans un souci de sécurité, et pour ne pas utiliser la puissance de calcul offerte par Grid'5000 à des fins d'attaque réseau ou de déni de service, les noeuds de calcul sont confinés dans un réseau totalement clos et non-routable (adressage IP privé), indépendant du réseau Internet. Seules certaines machines spécifiques sont autorisées à recevoir des connexions extérieures.

1.2.3 Accès aux ressources

Afin de fournir un accès consistant aux ressources de la grille, certaines conventions ont été décidées quant à la manière de fonctionner de chaque site, au nommage des machines, ou bien encore aux rôles de certaines machines spécifiques. Le but est ainsi d'uniformiser l'accès aux ressources de la grille, et de cacher par conséquent l'hétérogénéité présente entre les différents sites.

La figure 1.2 met en évidence la présence des machines suivantes sur chacun des sites de Grid'5000 :

- **La machine “ACCES”** : Son rôle est de donner un accès à la grille depuis l'extérieur. Du fait du confinement de la grille, il s'agit de la seule possibilité d'accès depuis l'extérieur. Cette machine n'autorise que des connections entrantes authentifiées et sécurisées via *ssh* (shell sécurisé, port tcp 22). Aucune connexion sortante n'est possible, garantissant ainsi l'herméticité de la grille en sortie.

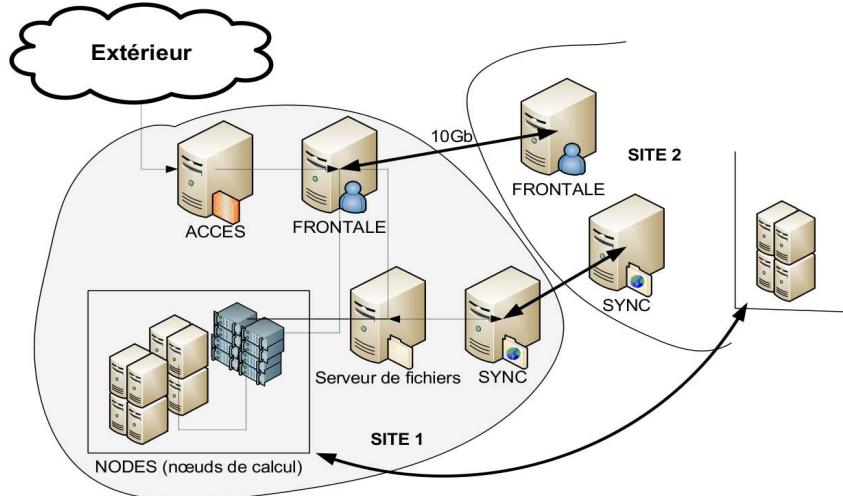


FIG. 1.2 – Conventions de nommage et accès aux ressources

- **La machine “FRONTALE” :** Cette machine permet les accès entre les sites via le réseau privé interne, afin de pouvoir se connecter d’un site à un autre. Ceci n’est en effet pas possible avec la machine “ACCES”, cette dernière n’acceptant que les connexions extérieures. De plus, la machine “FRONTALE” fournit les outils de réservation des noeuds de calcul.
- **La machine “SYNC” :** Elle est responsable de la synchronisation des données entre les différents sites. En effet, si toutes les machines d’un même site (frontale et noeuds de calcul) possèdent un espace de stockage partagé par *NFS* (*Network File System*), il n’y a pas de synchronisation automatique inter-site. Pour synchroniser ses données sur l’ensemble des sites de la grille, l’utilisateur emploie les machines “SYNC” dédiées avec les commandes appropriées.
- **Les machines de type “NODES” :** Il s’agit des noeuds de calcul exécutant les applications des utilisateurs de la grille. Ils sont regroupés en grappes de machines homogènes, et peuvent être réservés par l’intermédiaire des outils fournis par la frontale. Ces machines sont entièrement configurables et contrôlables par les différents utilisateurs de la grille.

1.2.4 Principe d'utilisation de la grille

Dans le cadre des recherches menées au cours de cette thèse, l’efficacité des grilles de calcul dans le domaine de la simulation électromagnétique de micro-systèmes complexes a pu notamment être démontrée [10, 11, 12]. Nous avons mis par ailleurs en évidence une procédure d’utilisation type de la plate-forme Grid’5000. Elle se décompose en deux phases principales : la mise au point des expériences, puis l’exécution de ces dernières.

Mise au point des expériences

Il s’agit ici de développer la ou les applications à exécuter sur la grille. Dans la plupart des cas, ce sont des applications massivement parallèles qui sont mises au point. Un nombre

important de tâches sont exécutées en parallèle, utilisant ainsi pleinement les capacités offertes par la grille.

La grille peut également permettre de lancer des applications paramétriques. Dans ce cas, la même application est exécutée un grand nombre de fois. Seules les entrées qui lui sont transmises changent d'une exécution à l'autre. La grille permet alors de gagner du temps en mettant ces exécutions en parallèle sur un grand nombre de nœuds de calcul. Les simulations électromagnétiques de micro-systèmes complexes mettent en jeu ce type d'applications.

Enfin, l'utilisateur a la possibilité de créer un environnement de travail personnalisé. Il s'agit d'un système d'exploitation (distribution *Linux Debian* par exemple) personnalisé, contenant les divers outils, librairies et applications dont il a besoin pour ses expériences. Cet environnement pourra ensuite être déployé de manière automatisée sur l'ensemble des nœuds de calcul utilisés.

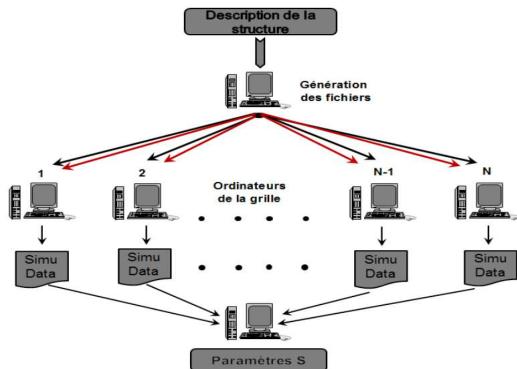
Durant cette phase, la grille n'est utilisée que très ponctuellement, la plupart du temps à des fins de tests.

Exécution des expériences

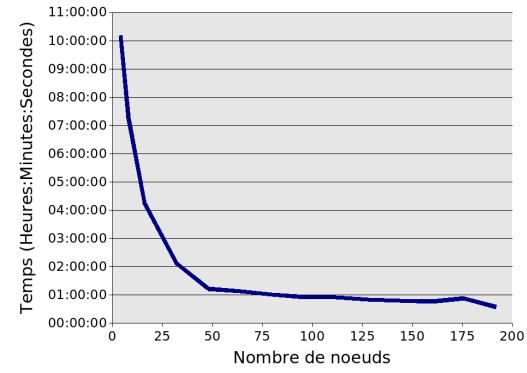
Il s'agit de la phase qui nous intéresse plus particulièrement. Elle met en jeu les étapes suivantes :

1. *Réservation des ressources* : L'utilisateur réserve, pour une durée déterminée, autant de ressources qu'il a besoin. Ces ressources peuvent être situées sur un seul *cluster*, de même qu'elles peuvent être distribuées sur plusieurs sites de la grille.
2. *Exécution de l'application* : L'application est ensuite exécutée au cours de la période réservée, en suivant les étapes suivantes :
 - (a) *Phase d'initialisation* : Cette phase regroupe toutes les opérations devant être exécutées avant les expériences proprement dites. Il peut s'agir, par exemple, de la génération de fichiers d'entrées utilisés par les différentes instances de l'application qui seront lancées (cf. figure 1.3(a)).
 - (b) *Déploiement du système d'exploitation personnalisé* : Le système d'exploitation mis au point lors de la préparation de l'expérience est installé automatiquement sur chaque nœud de calcul réservé. Ces derniers sont alors redémarrés pour utiliser ce système.
 - (c) *Déploiement des fichiers d'entrées* : Les données nécessaires à l'exécution des expériences, comme par exemple les fichiers générés à l'étape (a), sont transférées sur les nœuds de calcul.
 - (d) *Exécution de l'application* : L'application peut enfin être exécutée sur chacun des nœuds réservés. Dans le cas de l'application de simulation électromagnétique de micro-systèmes complexes, les expériences réalisées dans [10] mettent en avant l'impact significatif de l'utilisation d'un grand nombre de machines sur les temps d'exécution mesurés (cf. figure 1.3(b)).

- (e) *Récupération des fichiers générés* : Les données résultant de l'exécution de l'application sont collectées et mises à disposition de l'utilisateur. Elles sont alors stockées sur le serveur NFS des sites concernés. Les données présentes localement sur chaque noeud sont effacées lors du redémarrage des machines.



(a) Distribution des données d'entrées et collectes des données générées



(b) Temps d'exécution de l'expérience en fonction du nombre de machines utilisées

FIG. 1.3 – Exécution de l'application de simulation électromagnétique de micro-systèmes complexes sur Grid'5000

1.3 Problématique liée à la gestion des ressources

Différents axes de recherche majeurs concernant le calcul sur grille peuvent être identifiés [2], parmi lesquels figurent la sécurité au sein des grilles, l'instrumentation et l'analyse de performance, la gestion des ressources, les outils et modèles de programmation adaptés, etc. Cette thèse se concentre sur les aspects liés à la gestion des ressources des grilles de calcul, en prenant comme exemple la grille de recherche Grid'5000.

La section précédente montre que, sur cette plate-forme, les utilisateurs peuvent réserver autant de machines qu'ils le désirent, et ce pour une durée qu'ils spécifient eux-mêmes. Si certaines utilisations de la grille apparaissent comme raisonnables (quelques centaines de machines durant quelques heures), d'autres en revanche doivent être considérées comme exceptionnelles afin de pouvoir servir de manière équitable l'ensemble de la communauté utilisatrice de la grille. C'est le cas d'expériences nécessitant l'utilisation de plusieurs milliers de noeuds de calcul durant plusieurs jours.

Cette thèse traite le problème de l'ordonnancement dans un tel contexte. Elle propose de définir un modèle d'utilisation de la grille fondé sur des valeurs humaines telles que le partage des ressources dans le but de satisfaire au mieux l'ensemble de la communauté. L'objectif est ainsi de parvenir à réguler la demande sur les ressources, d'une part en incitant les utilisateurs à adopter un comportement responsable quant à l'utilisation qu'ils ont de la grille, et d'autre part en les aidant à mieux cerner les besoins en ressources de leurs applications.

1.4 Plan de la thèse

L'objectif principal de cette thèse est double. Dans un premier temps, il s'agit de mettre au point un modèle d'ordonnancement basé sur des critères économiques. L'idée est ainsi de parvenir à réguler, de manière naturelle, la demande sur les ressources de la grille. Il est possible d'imaginer que chaque utilisateur, ou groupe d'utilisateurs, de Grid'5000 soit doté d'un budget virtuel utilisable pour réserver des ressources, offrant ainsi à chacun un accès équitable à ces dernières.

Le second objectif de cette thèse concerne la prédiction du temps d'exécution d'applications. Le but est de parvenir à estimer de manière fiable le temps d'exécution d'une application, de sorte que cette estimation puisse être utilisée dans le processus de réservation de ressources. En effet, des études ont démontré que les utilisateurs qui spécifient eux-mêmes les temps d'exécution prévus pour leurs applications ont tendance à surestimer ces temps, parfois même de manière importante [13]. Il en résulte alors un gaspillage des ressources, pouvant créer un phénomène de pénurie.

Le chapitre 2 propose un état de l'art concernant la gestion des ressources des grilles de calcul. Tout d'abord, nous détaillons les différentes missions d'un gestionnaire de ressources, telles que l'ordonnancement ou l'observation de leur état. Ensuite, différents gestionnaires de ressources existants sont étudiés. Une section de ce chapitre aborde les principales techniques d'ordonnancement utilisées par ces systèmes. Enfin, nous montrons comment des paradigmes reposant sur l'utilisation de modèles économiques peuvent contribuer à l'ordonnancement sur les grilles de calcul.

Nous proposons dans le chapitre 3 un modèle économique permettant la gestion des ressources d'une grille de calcul. Le modèle mathématique mis au point y est détaillé. Une implémentation en est ensuite proposée, et nous montrons comment l'intégrer au sein de l'ordonnanceur actuellement utilisé sur Grid'5000. Une analyse des résultats du modèle et des performances de son implémentation est effectuée.

Le modèle économique proposé nécessite de connaître à l'avance la durée d'exécution des applications à placer. Ainsi, dans la seconde partie de la thèse, nous nous intéressons à la prédiction du temps d'exécution d'applications parallèles, destinées à être exécutées sur une grille de calcul.

Le chapitre 4 dresse un état de l'art des techniques pouvant être utilisées pour estimer le temps d'exécution d'une application. Nous montrons que deux principales techniques existent pour cela : la prédiction basée sur un historique d'exécutions passées, et celle basée sur le profil de l'application. Dans ce chapitre, nous introduisons également la méthode développée dans le cadre de cette thèse, à savoir une approche hybride combinant le principe des deux techniques décrites.

Enfin, les chapitres 5 et 6 décrivent l'approche de prédiction choisie, chacun traitant un aspect particulier de la méthode hybride. Ainsi, nous étudions comment, à partir du profil d'une application, nous pouvons tout d'abord déterminer des informations temporelles sur celle-ci, puis comment déterminer son comportement. La connaissance de ces deux données permet d'estimer le temps d'exécution de l'application.

Chapitre 2

Gestion des ressources d'une grille de calcul

La gestion de ressources distribuées est un domaine dans lequel de nombreuses recherches ont été menées. Qu'il s'agisse de machines massivement parallèles de type supercalculateur, ou bien de *clusters*, des outils de gestion efficaces ont été mis au point. Cependant, de tels outils ne sont pas adaptés à des supports tels que les grilles de calcul.

Ce chapitre présente le problème de la gestion des ressources d'une grille de calcul. Il introduit les différentes caractéristiques propres à ce domaine et détaille le fonctionnement de différents gestionnaires de ressources existants. Une attention toute particulière est portée au gestionnaire *OAR*, puisqu'il s'agit de celui utilisé sur la plate-forme Grid'5000. D'autre part, ce chapitre permet de positionner le modèle d'ordonnancement proposé dans le cadre de cette thèse parmi l'offre déjà existante, et ainsi de mettre en avant les contributions qu'il apporte.

2.1 Objectifs et challenges

2.1.1 Caractéristiques de l'ordonnancement sur grille

Les ressources présentes sur une grille de calcul possèdent des caractéristiques devant être prises en charge par les systèmes responsables de leur gestion. Le type de grille que l'on considère (grille de calcul, d'information, de stockage, etc.) conditionne le type de ressources à gérer. Concernant une grille de calcul, il s'agit principalement des processeurs fournissant la puissance de calcul, et du réseau reliant les différents noeuds [14, 15, 16].

Les systèmes d'ordonnancement doivent prendre en considération les caractéristiques suivantes [14, 17] :

- **Hétérogénéité des ressources** : La grille est une interconnexion de ressources appartenant à différents domaines. Si les ressources peuvent être homogènes au sein d'un même domaine, il en est tout autrement lorsque l'ensemble de la grille est considéré. De fortes différences peuvent être constatées en termes d'architectures matérielles des machines partagées, de vitesse des processeurs, de systèmes d'exploitation utilisés, de bande-passante reliant les machines au réseau, etc. Ceci résulte en une différenciation des capacités de calcul et d'accès aux données pour chaque ressource.

- **Autonomie des sites** : Les sites constituant la grilles appartiennent à différents domaines d'administration, chacun possédant ses propres politiques d'accès, de sécurité et de gestion des ressources. Le gestionnaire de ressources doit savoir s'adapter à ces spécificités, et laisser une totale autonomie aux différents sites.
- **Diversité des applications** : Une grande diversité d'utilisateurs venant de différents horizons peuvent être amenés à utiliser la grille. Il en résulte une grande palette d'applications pouvant être exécutées, et donc devant être prises en charge par les algorithmes d'ordonnancement. Il peut s'agir d'applications parallèles communicantes qui possèdent un nombre important de tâches échangeant une quantité variable de données, de même que des applications paramétriques, pour lesquelles différentes instances indépendantes de l'application sont lancées pour des paramètres différents.
- **Ressources non-dédiées** : Les ressources présentes sur la grille ne sont généralement pas dédiées à celle-ci. Les utilisateurs des divers domaines peuvent utiliser les ressources leur appartenant sans passer par le gestionnaire de ressources de la grille, ce qui implique une diversité de nouveaux paramètres à prendre en compte. Cependant, certaines grilles de calcul possèdent des ressources dédiées, comme les grilles orientées vers la prestation de services applicatifs (ASP pour *Application Service Provider*) [18].
- **Aspects dynamiques** : Une des caractéristiques des grilles de calcul est le comportement dynamique des ressources qui la composent. D'une part, la disponibilité de ces dernières peut varier fortement au cours du temps : une ressource peut rejoindre ou quitter la grille inopinément. De plus, les performances des ressources peuvent également fluctuer en fonction de l'importance de leur utilisation.

Les caractéristiques ainsi mises en avant sont très différentes de celles que l'on retrouve dans les systèmes traditionnels. Par exemple, les ressources d'un *cluster* sont généralement homogènes, et peuvent être considérées comme stables. Il est par conséquent nécessaire d'adapter les gestionnaires de ressources afin de prendre en compte l'ensemble de ces caractéristiques [2, 19].

2.1.2 Missions d'un gestionnaire de ressources

Un gestionnaire de ressources regroupe un ensemble de composants, dont le rôle principal est de fournir les services suivants [17, 20] :

- un service d'ordonnancement,
- un service d'information,
- un service d'exécution et d'observation.

En pratique, la figure 2.1 montre les composants mis en jeu ainsi que leurs interactions [15, 17]. Le rôle du gestionnaire de ressources de la grille est de s'interfacer avec les gestionnaires locaux, d'une part pour soumettre des applications aux ressources des domaines dont ceux-ci ont la responsabilité, et d'autre part pour collecter les informations provenant de ces même ressources. Ceci garantit ainsi l'autonomie des différents domaines de la grille. De plus, au sein d'un domaine, les utilisateurs peuvent confier des tâches au gestionnaire de ressources local, mettant ainsi en avant l'aspect potentiellement non-dédié de la grille.

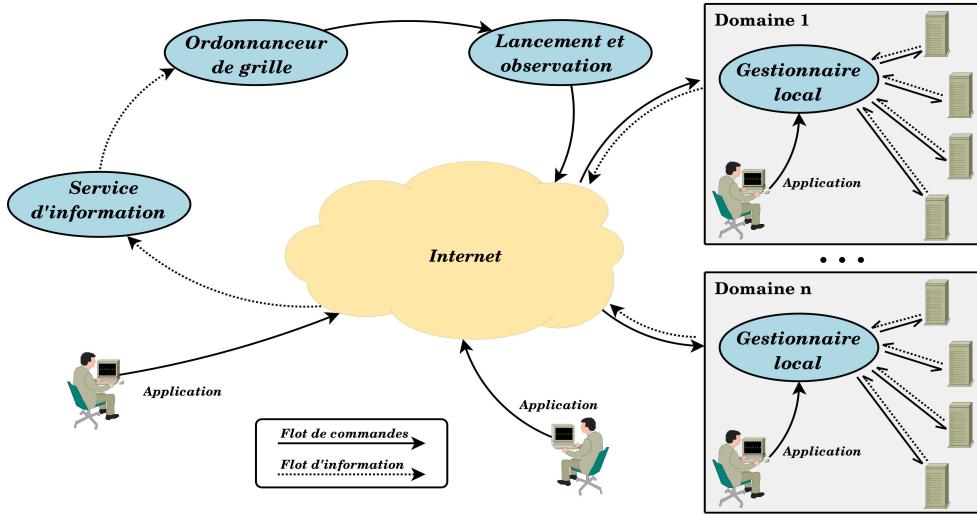


FIG. 2.1 – Architecture d'un gestionnaire de grille

L'ordonnancement sur une grille de calcul met en œuvre les trois étapes suivantes [21] :

1. **Découverte des ressources** : L'utilisateur fournit au gestionnaire la description de l'application à exécuter. Le système doit alors déterminer quelles sont les ressources disponibles pour cet utilisateur. Pour cela, le service d'information est utilisé afin d'identifier les ressources présentes sur la grille. Un premier filtrage est alors opéré afin d'éliminer celles que l'utilisateur n'a pas le droit d'utiliser. De plus, le gestionnaire calcule les besoins de l'application en ressources (leur nombre, leur puissance, le temps nécessaire, etc.). Un second filtrage peut alors être effectué pour ne retenir que les ressources pouvant satisfaire ces besoins.
2. **Sélection des ressources** : Il s'agit de l'étape d'ordonnancement. Etant donné un ensemble de ressources possibles pour exécuter l'application de l'utilisateur, le gestionnaire doit choisir celles qui seront réellement utilisées. Il s'appuie pour cela sur une estimation des besoins en ressources de l'application, ainsi que sur une prédition des performances des ressources. Divers algorithmes sont alors disponibles, permettant de choisir un ordonnancement en fonction des objectifs fixés (équilibrage de charge, temps d'exécution minimal, etc.).
3. **Exécution de l'application** : Cette dernière étape permet au gestionnaire de grille de se connecter à un ou plusieurs gestionnaires locaux afin de leur confier l'application à exécuter. Un suivi de la progression de cette exécution et de l'état des ressources employées peut éventuellement être mis en place.

2.2 Ordonnancement sur une grille de calcul

Le problème de l'ordonnancement sur une grille de calcul peut être représenté par la figure 2.2. Il s'agit d'attribuer les tâches de l'application à exécuter aux différentes ressources de la grille. Pour cela, l'ordonnanceur considère d'une part les caractéristiques de l'application, et d'autre part celles des ressources disponibles. L'ordonnancement est alors calculé de manière à satisfaire un objectif fixé [22].

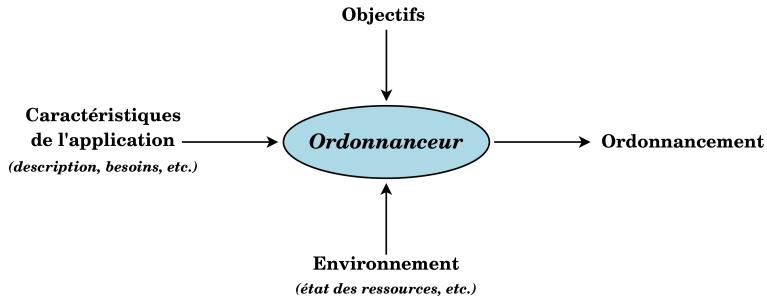


FIG. 2.2 – Le problème de l’ordonnancement

2.2.1 Applications

Differentes caractéristiques des applications candidates à l'exécution sur un support tel qu'une grille de calcul doivent être prises en compte par le gestionnaire de ressources de la grille [14, 23].

La première de ces caractéristiques correspond aux exigences des applications liées aux calculs. Il s'agit ici de fournir à l'ordonnanceur des données telles que le temps processeur nécessaire pour les exécuter, ou encore l'espace mémoire demandé. D'autres facteurs, tels que la date de début requise par l'application ou son coût maximal, peuvent être pris en considération.

De plus, il est également possible de distinguer le cas des applications *batch* de celui des applications interactives. Les premières ne nécessitent aucune intervention de la part de l'utilisateur au cours de leur exécution. Elles sont ainsi parfaitement adaptées à une exécution sur grille, et leur exécution peut être préemptée et différée si besoin. En revanche, les applications interactives dialoguent régulièrement avec l'utilisateur. Dans ce cas, de telles pratiques sont difficilement envisageables.

Les besoins des applications en termes de réseau doivent également être pris en considération. Par exemple, s'il s'agit d'applications parallèles, les besoins en termes de bande-passante dépendent du volume de données devant être transférées entre les tâches. Les applications paramétriques n'ont quant à elles aucune exigence particulière sur ce point. Ces caractéristiques influenceront l'ordonnancement afin de ne pas pénaliser ces applications par un mauvais choix de ressources.

Enfin, il est possible de distinguer deux derniers cas : celui des applications temps réel, pour lesquelles les tâches sont soumises à des contraintes de *deadlines* (dates butoir), et les applications auxquelles un niveau de qualité de service est garanti. Pour exécuter ce type d'applications dans les délais impartis, le gestionnaire de ressources doit sélectionner les ressources pour lesquelles les performances prédictes sont fiables.

2.2.2 Ressources

Les ressources gérées par les systèmes étudiés dans ce chapitre peuvent être fortement hétérogènes. Ainsi, divers paramètres les concernant doivent être pris en compte lors de la phase d'ordonnancement [14].

La première caractéristique que nous distinguons concerne les performances des ressources. Pour des ressources de calcul (processeurs), il s'agit par exemple du nombre d'opérations pouvant être réalisées à la seconde. Dans le cas des réseaux, c'est la bande-passante qui est considérée. Dans tous les cas, les capacités des ressources constituent un paramètre important de l'ordonnancement.

Les capacités des ressources sont de plus modulées par leur charge. Le gestionnaire de ressources est alors responsable de la prédition de performances des ressources, en se basant sur une estimation de cette charge. Plusieurs méthodes peuvent être utilisées pour ceci : l'établissement d'un modèle théorique, l'estimation basée sur un historique, etc.

Une autre caractéristique importante à retenir est l'aspect dédié des ressources. La majorité des grilles fonctionne avec des ressources non-dédiées aux activités générées par la grille. En d'autres termes, les applications soumises se partagent l'accès aux ressources avec des applications lancées par les utilisateurs locaux de ces ressources. Dans de telles conditions, il est difficile de prédire leurs performances, ce qui peut avoir un impact significatif sur l'ordonnancement.

Enfin, de nombreux autres paramètres pourront être pris en compte, parmi lesquels le fonctionnement en temps partagé, ou bien encore la possibilité de préempter les applications présentes sur les ressources.

2.2.3 Fonctions objectifs

L'ordonnanceur choisit les ressources auxquelles les tâches des applications soumises seront associées de manière à atteindre un objectif. Deux types d'objectifs sont principalement considérés [14, 17] :

- **Objectifs centrés sur les applications** : Ces objectifs tendent à satisfaire au mieux les exigences de chacune des applications soumises. La fonction objectif à minimiser pour trouver un ordonnancement peut inclure les paramètres suivants :
 - *Temps d'exécution* : le temps mis pour exécuter l'application.
 - *Temps d'attente* : le temps passé par l'application dans une file d'attente, avant son exécution d'une part, et durant son exécution si elle subit des interruptions.
 - *Temps de réponse* : le temps total, correspondant à la somme du temps d'exécution et du temps d'attente.
 - *Ralentissement de l'application* : le rapport entre le temps d'attente de réponse et le temps d'exécution réel de l'application.
 - *Coût de l'application* : si la gestion des ressources est réalisée à l'aide de paradigmes économiques.
- **Objectifs centrés sur les ressources** : Les objectifs appartenant à cette catégorie favorisent un jeu de métriques liées au système, et non aux applications. Les propriétés suivantes du système peuvent alors être considérées :
 - *Capacité de traitement* : le nombre d'applications qui se terminent par unité de temps (par heure, par jour, etc.).
 - *Taux d'utilisation* : le pourcentage de temps durant lequel la ressource est occupée.

- *Performance moyenne des applications.*
- *Profits générés* : si la gestion des ressources est réalisée à l'aide de paradigmes économiques.

2.2.4 Catégories d'ordonnancement

Des études ont été menées afin de parvenir à définir des catégories d'algorithmes d'ordonnancement [24, 15, 22, 17]. Nous choisissons de retenir les catégories présentées en figure 2.3.

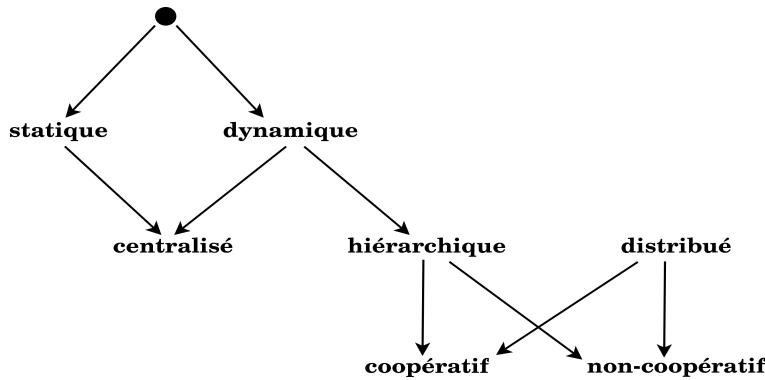


FIG. 2.3 – Catégories d'ordonnancements adaptés aux grilles de calcul

2.2.4.1 Placement statique et placement dynamique

Dans le cas du placement statique, la décision de placement est prise hors-ligne, c'est-à-dire avant l'exécution du programme. Les informations concernant l'état des ressources ainsi que de l'application soumise sont supposées disponibles au moment où l'ordonnancement est calculé. Ce type de placement est bien adapté aux applications déterministes dont le comportement est bien maîtrisé d'une part, et d'autre part aux grilles pour lesquelles l'état des ressources peut être prédit de façon fiable, comme c'est le cas des grilles dont les ressources sont dédiées.

Le placement dynamique repose sur le principe d'une allocation des tâches à la volée, lorsque l'application s'exécute. Il s'agit ainsi d'un placement en-ligne. L'algorithme peut éventuellement s'adapter au cours de l'exécution de l'application à l'état des ressources, notamment à la charge des processeurs, en choisissant de migrer certaines tâches d'une ressource à une autre. Le placement dynamique est utile lorsqu'il est impossible, par exemple, de déterminer le temps d'exécution de l'application à exécuter.

2.2.4.2 Placement centralisé, décentralisé ou hiérarchique

Une stratégie d'ordonnancement centralisée concentre toutes les prises de décisions à un seul endroit : un ordonnanceur unique est responsable de calculer un placement des applications soumises sur l'ensemble des ressources disponibles de la grille.

Cette approche est optimale, puisque l'ordonnanceur a une connaissance de l'état de toutes les ressources. Cependant trois points négatifs se posent :

- le passage à l'échelle n'est pas facilité par une telle approche,
- ce système n'admet pas de tolérance aux fautes,
- le goulot d'étranglement ainsi créé peut engendrer une baisse importante des performances du système.

Le modèle décentralisé, ou distribué, confie la responsabilité de l'ordonnancement à différents ordonnanceurs, agissant de manière coopérative ou non. Enfin, la dernière stratégie d'ordonnancement repose sur un modèle hiérarchique, pour lequel les ordonnanceurs gèrent des entités de plus ou moins haut niveau selon leur position dans la hiérarchie. Ce dernier modèle constitue une combinaison de l'approche statique et de l'approche dynamique.

2.2.4.3 Modèle coopératif et modèle non-coopératif

Dans le cas d'une stratégie d'ordonnancement dynamique ou hiérarchique, les différents ordonnanceurs peuvent collaborer entre eux pour déterminer un placement, ou bien ils peuvent travailler de manière totalement indépendante. Le modèle coopératif permet à chaque ordonnanceur de calculer une partie du placement global d'une application, en communiquant avec les autres ordonnanceurs dans le but d'évoluer vers un objectif commun.

Au contraire, les ordonnanceurs évoluant en mode non-coopératif sont des entités autonomes qui prennent des décisions dans le but d'optimiser uniquement leurs propres objectifs, sans se soucier de la vue d'ensemble du système.

2.2.5 Stratégies d'ordonnancement

Différentes stratégies d'ordonnancement, également qualifiées d'algorithmes de placement, peuvent être définies d'après les points cités précédemment. Quelques exemples peuvent être cités :

- **La politique de type FIFO (*First In, First Out*)** [25] : La stratégie la plus simple consiste à exécuter les applications dans l'ordre où elles sont soumises. Dans sa version la plus stricte, les ressources ne fonctionnent pas en temps partagé. Ce type de politique peut ainsi générer des problèmes dus au fractionnement des ressources. En effet, si le nombre de ressources libres n'est pas suffisant pour exécuter l'application en tête de la file d'attente, celle-ci bloquera les autres applications en attente.
- **Le *backfilling*** [26, 27] : Il s'agit d'une amélioration de la politique FIFO. Lorsqu'une ou plusieurs applications sont bloquées en tête de file d'attente pour cause de manque de ressources, les autres travaux en attente peuvent passer en tête de file s'ils n'interfèrent pas avec celles-ci. On distingue alors :
 - Le *conservative backfilling* : les applications peuvent passer en tête de file si elles ne retardent aucun travail arrivé avant.
 - Le *easy backfilling* : les applications peuvent passer en tête de file si elles ne retardent pas le travail arrivé en premier (en tête de file).

- **Le *gang scheduling*** [28, 29] : Les applications évoluent en temps partagé sur les ressources. Chacune d'entre elles dispose d'un accès aux ressources pendant une quantité de temps fixe (on emploie le terme de *quantum*), puis elle est interrompue afin de libérer les ressources pour les autres applications, et reprendra son exécution lors du prochain *quantum* qui lui sera alloué. Notons que la préemptibilité est un prérequis à cette approche.
- **La politique MET (*Minimum Execution Time*)** : Cette stratégie permet d'assigner chacune des tâches au processeur qui minimise son temps d'exécution, sans se préoccuper de sa disponibilité. Chaque tâche aura ainsi la ressource qui lui est en théorie la mieux adaptée. Cependant, cette approche résulte en pratique à un fort déséquilibre de charge entre les ressources, et si le temps d'exécution des tâches est minimal, leur temps de réponse (attente et exécution) n'est en revanche pas optimal.
- **La politique MCT (*Minimum Completion Time*)** : Cette politique est identique à la politique MET, excepté que le temps pris en compte est le temps de réponse, c'est-à-dire le cumul du temps d'attente et du temps d'exécution. Cette stratégie permet alors un meilleur équilibrage des charges sur l'ensemble des ressources considérées.
- **La politique MECT (*Minimum Execution and Completion Time*)** : En pratique, les politiques MET et MCT peuvent être utilisées conjointement [30, 31]. L'algorithme de placement admet alors plusieurs entrées : l'ensemble des tâches T_i de l'application, et le temps d'exécution $e_{i,j}$ de chaque tâche i sur un processeur P_j . Pour chacune des tâches T_i de l'application, les étapes suivantes sont nécessaires afin de déterminer le processeur m qui l'exécutera :
 - Calcul du temps d'attente maximal : $b_{max} = \max_{P_j \in P} b_{i,j}$
où $b_{i,j}$ désigne le temps d'attente de la tâche T_i sur le processeur P_j .
 - Détermination de l'ensemble de ressources $P' = \{P_k \mid c_{i,k} < b_{max}\}$
Il s'agit ainsi de déterminer les ressources pour lesquelles le temps de réponse sera inférieur au temps d'attente maximal.
 - Si de telles ressources existent ($|P'| > 0$), alors on choisit m tel que : $e_{i,m} = \min_{P'_j \in P} e_{i,j}$
c'est-à-dire que le processeur élu est celui qui, parmi les ressources sélectionnées, minimise le temps d'exécution de l'application.
 - Dans le cas contraire, on choisit le processeur m minimisant le temps de réponse parmi toutes les ressources disponibles : $c_{i,m} = \min_{P_j \in P} c_{i,j}$
 Il s'agit ainsi de faire une première sélection de ressources pour lesquelles le temps d'attente de la tâche n'est pas pénalisante. Dans ce cas, la politique MET peut être appliquée. Si de telles ressources n'existent pas, on emploie alors la politique MCT.
- **Le *rescheduling*** [32] : Au cours de l'exécution d'une application, il est possible que l'ordonnancement initialement calculé perde son optimalité en fonction des performances du système ou bien en raison du changement des besoins de l'application. Il peut ainsi être intéressant de modifier l'ordonnancement afin de migrer les tâches vers des ressources plus à même de les exécuter. Cette stratégie peut également être utilisée pour mettre en place un équilibrage de charge dynamique des ressources.

- **Le placement avec qualité de service garantie [33]** : Cet algorithme permet de placer des applications parallèles sur un support orienté vers la prestation de services applicatifs : les ressources sont dédiées aux activités liées à la grille, de telle sorte que leur charge est connue et prévisible à tout moment. Quatre classes d'applications sont définies, par ordre de priorité décroissante (cf. figure 2.4) :

- Applications avec *deadline* : une date butoir est définie pour chaque application, avant laquelle celle-ci doit se terminer. L'exécution peut être immédiate ou différée.
- Applications prioritaires : cette classe garantit que l'exécution sera immédiate.
- Applications à ressources non-partagées : chaque application sera seule à utiliser les ressources qui lui sont attribuées durant son exécution.
- Applications *best effort* : aucune contrainte n'est exprimée sur ces applications. Elles sont exécutées le plus tôt possible avec les ressources disponibles.

Ce type de placement soulève un problème d'optimisation, pour lequel le critère retenu permet d'optimiser l'utilisation des ressources du fournisseur. Ainsi, pour une application A donnée, l'ordonnancement s est choisi parmi tous les ordonnancements $S(A)$ possibles d'après le critère suivant :

$$\min_{s \in S(A)} \left(\max_m \left(t_f(m, s) + t_f(m, s) \times M_f \times \frac{M_u(m)}{M_t(m)} \right) \right)$$

La date de libération des ressources $t_f(m, s)$ est optimisée. De plus, un second critère consiste à intégrer l'espace mémoire utilisé, de manière à pénaliser les machines dont la mémoire est déjà saturée. Ainsi, le rapport $\frac{M_u(m)}{M_t(m)}$ représente la quantité de mémoire utilisée sur la machine m par rapport à la mémoire totale. Le coefficient M_f permet d'ajuster la pondération de cette partie du critère, tandis qu'on multiplie par $t_f(m, s)$ afin de se ramener dans le même ordre de grandeur que la première partie du critère.

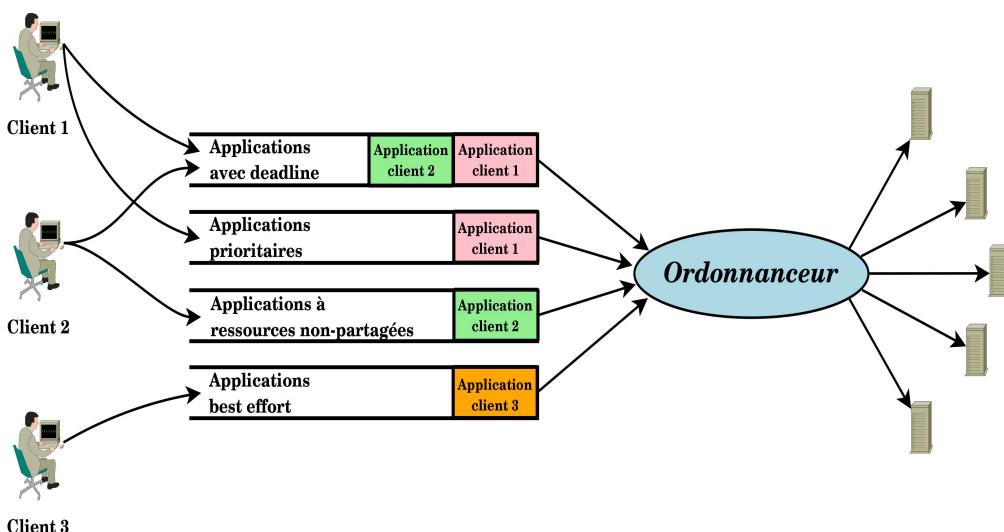


FIG. 2.4 – Modèle de files d'attente de l'ordonnancement avec qualité de service garantie

2.2.6 Introduction aux modèles économiques

La grille de calcul est un lieu d'échange de ressources : leurs propriétaires les mettent à disposition des utilisateurs de la grille pour que ces derniers puissent y exécuter leurs applications. Dans un tel contexte, les fournisseurs de ressources et leurs consommateurs auront chacun leurs propres objectifs et stratégies [34]. Une approche basée sur les modèles économiques du marché réel peut ainsi être employée dans le cadre du calcul sur grille.

Les modèles économiques constituent une métaphore pour la gestion des ressources et le placement d'applications, permettant ainsi de réguler l'offre et la demande au niveau des ressources de la grille tout en garantissant une certaine qualité de service aux clients. Ils permettent ainsi d'inciter les producteurs à partager leurs ressources, en leur fournissant une contrepartie, et également inciter les consommateurs à réfléchir au compromis entre temps de calcul et coût selon la qualité de service désirée.

2.2.6.1 Historique

L'expérience d'Harvard en 1968

Le premier exemple d'utilisation de paradigmes économiques pour gérer des ressources informatiques remonte à la fin des années 60. Sutherland publie en 1968 un rapport détaillant l'utilisation d'un système d'enchères mis en place au sein de l'université d'Harvard pour réguler l'accès à un ordinateur, le *PDP-1* [35].

Cet ordinateur était utilisé par l'ensemble des étudiants de l'université, ainsi que par les membres enseignants et chercheurs, afin d'exécuter leurs expériences. Celles-ci pouvaient durer de quelques minutes à plusieurs heures, au cours desquelles les utilisateurs devaient rester devant la machine. En effet, les expériences étaient pour la plupart interactives, et nécessitaient ainsi la présence des utilisateurs. De plus, le PDP-1 était mono-utilisateur. Par conséquent, les utilisateurs étaient en compétition pour y accéder, et une demande très importante était constatée en journée, à des heures que l'on pourrait qualifier de "pratiques" pour les utilisateurs.

Un système d'enchères a alors été mis en place. Chaque personne se voyait remettre un certain montant d'une monnaie fictive, le *yen*, en fonction de la priorité de ses expériences. Le temps était divisé en tranches de 15 minutes, et chaque utilisateur pouvait proposer une certaine somme, en *yens* par heure, pour obtenir l'accès à la machine durant l'intervalle de temps désiré. Chaque utilisateur pouvait alors surenchérir en fonction de ses besoins, et de ses capacités à le faire. La coordination des enchères était réalisée en centralisant toutes les offres sur un tableau d'affichage unique.

Sutherland montre ainsi qu'un tel système était parfaitement fonctionnel. Il était en outre exempt de tout phénomène de famine, et était auto-suffisant : les *yens* étaient automatiquement reversés aux utilisateurs aussitôt après les avoir consommés.

De la fin des années 60 au milieu des années 80

Dès lors, l'emploi de paradigmes économiques a été repris pour réguler l'utilisation de ressources informatiques [36]. Par exemple, Nielsen présente dans [37] un modèle de tarification destiné à réguler la demande au sein d'une communauté de 200 physiciens du *Stanford Linear Accelerator Center*. Ces derniers exécutaient la plupart du temps des applications *batch*. L'objectif était alors de les inciter à étaler les soumissions au cours du temps afin de ne pas saturer le système.

La facturation de l'utilisation des ressources pouvait également avoir d'autres objectifs [38, 39], tels que l'amortissement du coût du matériel. Les stratégies d'établissement des prix s'appuyaient alors sur l'optimisation du profit des fournisseurs de ressources.

L'arrivée des réseaux et du calcul distribué

A partir du milieu des années 80, la démocratisation des réseaux informatiques a changé la philosophie de l'utilisation de modèles économiques. La multiplicité des fournisseurs de ressources a compliqué la situation [40] : désormais, ces derniers sont en compétition pour vendre l'utilisation de leurs ressources. On se rapproche alors d'une réelle économie de marché [41, 42, 43].

L'utilisation de modèles économiques permet alors de décentraliser le processus d'allocation de ressources [41], limitant ainsi sa complexité. Ceci implique une plus grande efficacité dans la prise de décision, une meilleure tolérance aux fautes dans le système et un passage à l'échelle plus aisé.

2.2.6.2 Quelques exemples d'utilisation de modèles économiques

Ferguson et al. ont mis en place un algorithme d'équilibrage de charge basé sur des concepts provenant de la micro-économie [44]. Celui-ci est appliqué aux systèmes distribués dans le but d'allouer et de partager les ressources de calcul et de communication. Ainsi, les processeurs vendent des ressources libres (lien réseau ou temps CPU), en utilisant un modèle de vente aux enchères. Chaque application détermine alors ses préférences en termes de fournisseur : elle choisit ceux qui pourront lui fournir la meilleure prestation (selon ses besoins) en fonction du budget dont elle est dotée, et peut ainsi leur soumettre une offre. Les auteurs montrent que les performances de cette approche sont au moins aussi bonnes que celles des systèmes traditionnels d'équilibrage de charge.

Les auteurs de [45] décrivent le système appelé *Spawn*. Il s'agit d'un système de gestion de ressources permettant de gérer un réseau de stations de travail hautes-performances. Il récupère les ressources non-utilisées, et permet aux utilisateurs de les employer, évitant ainsi un gaspillage inutile. Un système de vente aux enchères est alors mis en place pour vendre des *quantums* de temps CPU aux applications. Celles-ci peuvent ainsi disposer d'un accès dédié à la ressource durant les *quantums* acquis. Les applications considérées par *Spawn* sont des applications parallèles à gros grain, ou bien des applications paramétriques. Celles-ci ont la possibilité de soumettre des offres aux fournisseurs, une offre étant composée du montant proposé pour une durée de temps spécifiée. Il est à noter que *Spawn* ne

tolère pas la migration, ni la préemption. Ainsi une application, qui ne se termine pas durant la période de temps qu'elle a achetée, a la priorité sur l'achat des *quantums* suivants, si toutefois son budget lui permet de suivre les prix du marché.

De la même manière, *POPCORN* [46] est un logiciel Java permettant de vendre aux enchères les cycles CPU d'ordinateurs reliés à l'Internet. La monnaie utilisée est acquise en partageant ses ressources, et est dépensée lors de la consommation de celles disponibles.

Un exemple de gestion des ressources d'une grille composée d'appareils mobiles est présenté dans [47]. Dans ce cas, les fournisseurs de ressources sont les appareils mobiles, tandis que les consommateurs sont les serveurs WAP qui leur confient des tâches à exécuter. L'objectif des auteurs a été de définir un modèle de calcul de coût s'appuyant sur un jeu de négociation non-coopératif, qui soit juste et stable. Ce modèle a alors été utilisé pour mettre en place un système d'ordonnancement décentralisé robuste à équilibrage de charge.

Le dernier exemple que nous citerons est celui de *Mariposa* [40, 48]. Il s'agit d'une base de données distribuée utilisant une approche basée sur les modèles économiques pour gérer le stockage ainsi que l'exécution des requêtes. Chaque site contient un ensemble d'objets stockés. Lorsqu'un client soumet une requête, celle-ci est décomposée en sous-requêtes, et un appel d'offres est opéré auprès des sites. En fonction du budget alloué à la requête, le client choisit les sites les plus adaptés, et leur soumet alors un ensemble de sous-requêtes. Lorsque celles-ci ont été traitées, les sites renvoient d'une part leur résultat, et d'autre part la facture correspondante. Il est à noter que si les sites mettent plus de temps à exécuter une requête que prévu, des pénalités leur sont imputées.

2.2.6.3 Modèles économiques existants

Différents modèles économiques peuvent être identifiés :

- Le **modèle de marché** [34, 49, 50] permet aux fournisseurs de ressources de fixer un prix auquel les utilisateurs seront facturés, proportionnellement à la quantité de ressources consommées. Le prix décidé par les propriétaires de ressources peut être fixe, ou bien varier au cours du temps selon la loi de l'offre et de la demande.
- Le **modèle des négociations** [51] permet à l'utilisateur, contrairement au précédent, d'agir sur le prix des ressources qu'il désire consommer. Ici, producteurs et consommateurs auront leurs propres objectifs, et ils devront négocier afin de les atteindre.
- Le **modèle de l'appel d'offres** [40] est inspiré des mécanismes mis en place dans le monde des affaires pour gérer les échanges de biens et de services. Il permet au client de trouver le fournisseur de service approprié pour travailler sur une tâche donnée. Dans ce contexte, une nomenclature particulière est adoptée. Le client (ou le courtier qui le représente) est appelé "manager", tandis que chaque fournisseur est un "contractant" potentiel.
- Le **modèle de la vente aux enchères** [44, 45] établit une négociation entre un fournisseur et plusieurs consommateurs potentiels, ce qui permet d'aboutir à l'établissement d'un prix et au choix d'un consommateur. Une troisième entité entre également en jeu. Il s'agit du commissaire-priseur dont le rôle est de contrôler le bon déroulement de l'enchère.

- Le **modèle du partage proportionnel des ressources** [52] permet de partager une ressource entre plusieurs utilisateurs. La part de ressource allouée à chaque utilisateur par rapport aux autres est alors proportionnelle à son offre. Ceci permet à tous les utilisateurs d'avoir accès aux ressources, même s'ils n'ont pas un budget important par rapport aux autres utilisateurs.

2.3 Outils d'observation et de prédition

L'observation de l'état des ressources est une des missions fondamentales des gestionnaires de ressources des grilles de calcul. Cette section décrit quelques exemples d'outils d'observation et de prédition existants.

2.3.1 MDS (Meta Directory Service)

Le *MDS* [53] est le service d'information intégré à la *Globus Toolkit*¹. Il s'agit d'un annuaire LDAP (*Lightweight Directory Access Protocol*) dont le rôle est de collecter l'état des ressources de la grille. Les informations collectées comprennent :

- la configuration des ressources, c'est-à-dire les informations statiques les concernant (fréquence du processeur, nombre de processeurs, quantité de mémoire, etc.),
- l'état instantané des ressources, c'est-à-dire les informations dynamiques les concernant (charge du processeur, nombre de processeurs utilisés, quantité de mémoire occupée, etc.),
- les informations sur les applications (besoins en termes de processeur, de mémoire, etc.).

Ce service possède une architecture hiérarchique (cf. figure 2.5) au sein de laquelle chaque ressource est associée à un *GRIS* (*Grid Resource Information Service*), eux même étant à la base d'une structure arborescente où chaque *GIIS* (*Grid Index Information Service*) collecte les informations provenant de plusieurs *GRIS* sous-jacents.

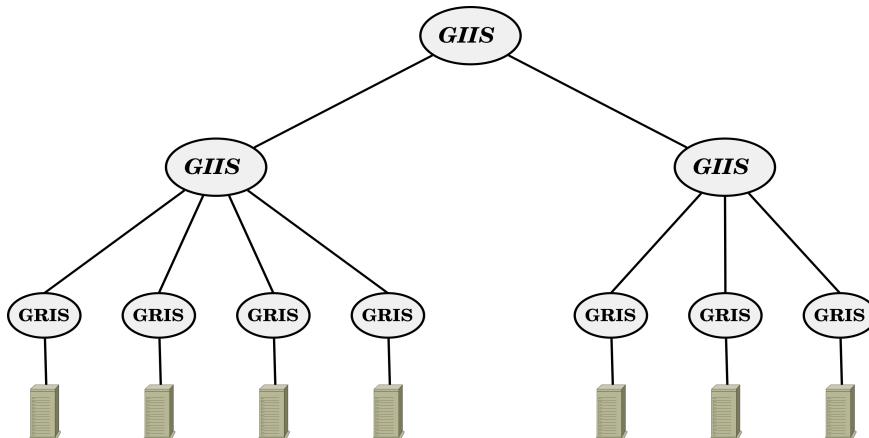


FIG. 2.5 – Architecture du *Meta Directory Service* de *Globus*

¹ *Globus* [54] est un intergiciel regroupant un ensemble d'outils qui offrent les services nécessaires à la mise en place d'une grille de calcul (service de gestion des ressources, service d'information, gestionnaire de sécurité, etc.).

Un *GRIS* permet la sauvegarde d'informations, statiques ou dynamiques, provenant de la ressource qui y est rattachée. Lorsque des informations sur une ressource sont demandées, la requête est transmise au travers de l'arbre, chaque *GIIS* la routant vers l'un des *GIIS* ou *GRIS* qu'il indexe. Le *GIIS* de haut niveau fournit ainsi une image cohérente de la globalité de la grille. La structure hiérarchique utilisée permet une meilleure *scalabilité* du service. Cependant, si le *MDS* permet de connaître l'état actuel de la grille, il n'implémente aucune fonction de prédition.

2.3.2 NWS (Network Weather Service)

NWS [55] est un outil d'observation fournissant la prédition de performances de ressources dynamiques dans des environnements distribués. Ainsi, *NWE* prédit les performances du réseau (latence et bande-passante) [56, 57], le pourcentage de CPU disponible sur chaque machine qu'il contrôle [58] et les performances de la mémoire.

Des mesures périodiques sont effectuées sur les performances délivrées par les ressources. Les prédictions sont ensuite réalisées au moyen de techniques statistiques en s'appuyant sur l'historique de ces mesures. Les performances prédictes peuvent être alors communiquées aux ordonnanceurs par le biais d'une interface dédiée à cet usage. Trois catégories de modèles de prédition sont disponibles :

- les modèles basés sur la moyenne utilisant une estimation de la moyenne comme prédition,
- les modèles basés sur la médiane,
- les modèles d'autorégression.

NWS choisit le modèle de prédition le plus adapté pour chacune des ressources (celui qui donne les meilleurs résultats) en comparant les erreurs de prédition avec les mesures.

Trois modules permettent le fonctionnement de *NWS*. Le premier est le système de capteurs qui collecte les informations sur les performances des ressources. Le module de prévision est responsable de la prédition des performances. Il transmet ensuite celles-ci au module de données. La figure 2.6 représente cette structure.

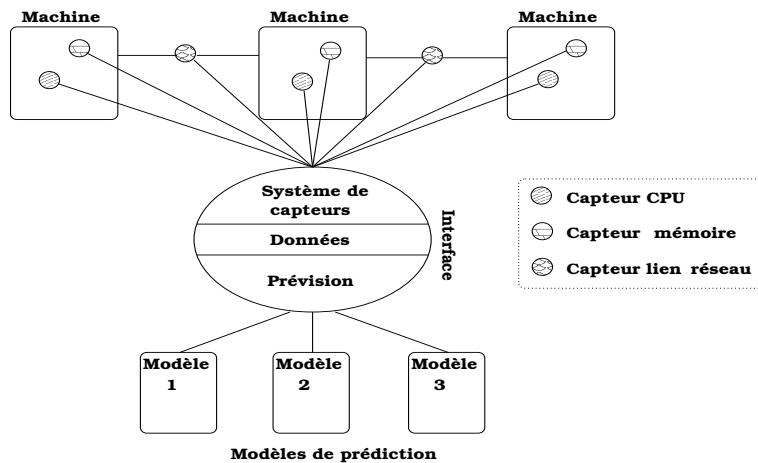


FIG. 2.6 – Structure du *Network Weather Service*

2.3.3 Ganglia

Ganglia [59] est un outil d’observation pour environnements de calcul hautes performances, tels que les grappes ou les grilles de calcul. Il utilise une représentation hiérarchique des grappes en grille afin de favoriser le passage à l’échelle (cf. figure 2.7).

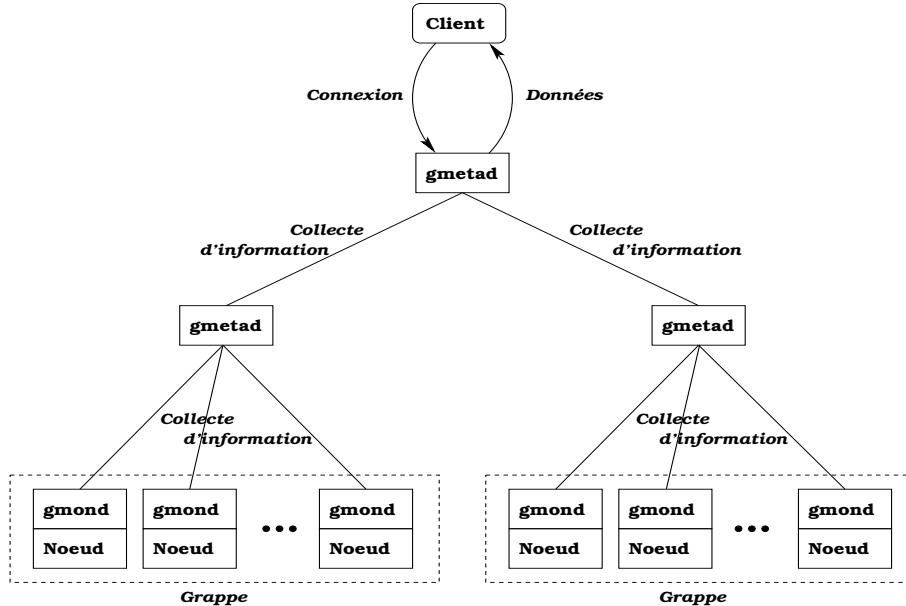


FIG. 2.7 – Architecture de *Ganglia*

Ganglia utilise des protocoles de communication *multicasts* ainsi qu’une structure hiérarchique de connexions point-à-point entre certains nœuds de chacune des grappes afin de relier ces dernières entre elles et d’agréger leurs informations d’état.

L’implémentation de *Ganglia* repose principalement sur deux *daemons* : *gmond* et *gmetad* :

- *gmond* : ce type de *daemon* a pour objectif de collecter les informations provenant de chaque grappe. Un *daemon* est présent sur chacun des nœuds du *cluster*, et communique avec les autres en utilisant plusieurs protocoles multicasts. Ces communications permettent de publier l’ajout d’un nouveau nœud, d’envoyer ses informations d’état aux autres nœuds de la grappe, ou de détecter la défaillance de l’un d’entre eux. Toutes les machines d’un même *cluster* communiquent entre elles, et ont ainsi la même vision de l’état de la grappe.
- *gmetad* : ce type de *daemon* permet de construire une représentation hiérarchique d’un ensemble de *clusters*. Chaque *daemon* collecte les informations concernant l’état de ses nœuds fils et en construit une représentation agrégée. Un nœud fils peut être une machine d’une grappe (dans ce cas, l’information agrégée est l’état de la grappe), ou bien un autre *daemon gmetad* (l’information agrégée est alors l’état d’un ensemble de *clusters*).

2.3.4 Monika et DrawOARGant

Le gestionnaire de ressources OAR, utilisé sur la plate-forme Grid’5000, s’appuie sur une base de données SQL dans laquelle il centralise toutes les informations qu’il traite.

Ces informations peuvent concerner les ressources (leurs caractéristiques, leur état, etc.) aussi bien que les applications placées sur la grille (qu'elles soient terminées, en cours d'exécution, ou planifiées pour un démarrage ultérieur).

L'utilisation d'une telle base de données offre une totale flexibilité quant au développement d'outils de visualisation. En effet, OAR fournit des tables qualifiées de "visuelles" : tout logiciel faisant un simple accès en lecture seule à ces tables peut ainsi construire une interface graphique de visualisation de l'état de grille.

L'outil *Monika* permet ainsi de visualiser, au sein d'un portail web, les différentes informations concernant l'état de la grille. Un tableau regroupe les noeuds présents sur un site (cf. figure 2.8), et affiche pour chacun d'entre eux son état actuel. Il est ainsi possible de connaître à chaque instant quelles sont les ressources disponibles (état *Free*), en panne (*Down*), ou encore utilisées (l'identifiant de l'application en cours d'exécution est alors affiché).

Grid'5000 Rennes OAR nodes

Summary:

OAR node status	Free	Busy	Total
Nodes	7	218	260
Cores	24	618	714

Reservations:

paramount-1	Free	Free	Free	Free	paramount-2	Free	Free	Free	Free
253990	253990	253990	253990	253990	paramount-4	253990	253990	253990	253990
paramount-9	253990	253990	253990	253990	paramount-10	252944	252944	252944	252944
paramount-13	252944	252944	252944	252944	paramount-14	252944	252944	252944	252944
paramount-17	252944	252944	252944	252944	paramount-18	Free	Free	Free	Free
paramount-23	253994	253994	253994	253994	paramount-24	253993	253993	253993	253993
paramount-25	Down				paramount-26	253993	253993	253993	253993
paramount-28	253992	253992	253992	253992	paramount-29	253991	253991	253991	253991
paramount-33	253991	253991	253991	253991	paramount-34	251937	251937	251937	251937
paramquad-1	249717	249717	249717	249717	paramquad-2	249717	249717	249717	249717
paramquad-8	Free	Free	Free	Free	paramquad-9	253998	253998	253998	253998
paramquad-12	251937	251937	251937	251937	paramquad-13	249717	249717	249717	249717

FIG. 2.8 – Visualisation de l'état des ressources de Grid'5000 à l'aide de *Monika*

DrawOARGantt est le deuxième outil de référence visuelle, utilisé sur Grid'5000. Il représente la distribution temporelle des applications sur chaque noeud sous la forme d'un diagramme de Gantt, comme le montre la figure 2.9.

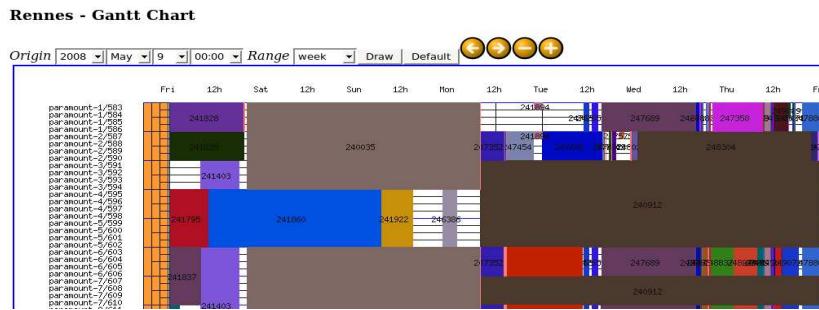


FIG. 2.9 – Diagramme de Gantt représentant les réservations de ressources dans le temps

2.4 Outils de placement et de gestion des ressources

Le gestionnaire de ressources joue un rôle central dans le fonctionnement d'une grille de calcul, puisqu'il est responsable de l'allocation des ressources aux différentes applications soumises par les utilisateurs. Cette section présente les principaux outils de ce domaine existants.

2.4.1 Globus

La *Globus Toolkit* est un ensemble d'outils et de logiciels permettant de concevoir et de mettre en œuvre des grilles de calcul et les applications qui leur sont destinées. L'objectif principal de *Globus* est de fournir aux différents utilisateurs d'une grille une API leur permettant d'accéder de façon transparente aux ressources qui leur sont offertes. Trois composants essentiels de cet intergiciel peuvent être distingués [54] :

- le gestionnaire de ressources,
- le service d'information (cf. section 2.3.1),
- le gestionnaire de données.

Le gestionnaire de ressources de *Globus* implémente un modèle d'ordonnancement décentralisé. Il est composé des trois entités suivantes (cf. figure 2.10) :

- Le **resource broker** reçoit les requêtes provenant des clients. Celles-ci sont écrites en RSL (*Resource Specification Language*), qualifié à ce stade de RSL de haut niveau. Le *broker* est alors responsable de la phase de découverte des ressources. Il utilise pour cela le *Meta Directory Service*. Le processus de spécialisation de la requête RSL peut alors commencer : il s'agit de la transformation de la requête originale en une nouvelle requête RSL pour laquelle l'endroit où se trouvent les ressources adéquates est spécifié.
- Le **resource co-allocator**² gère les requêtes multi-sites provenant du *resource broker*. Il transmet alors ces requêtes aux domaines appropriés, et permet de voir l'ensemble de ces derniers comme une seule et même entité.
- Un **resource manager** (appelé également **GRAM** pour *Grid Resource Allocation Manager*) est présent sur chacun des sites de la grille. Il s'appuie sur un système de gestion de ressources local pour placer et exécuter l'application.

Globus fournit l'infrastructure et les services permettant la gestion des ressources de la grille, tandis que l'ordonnancement est réalisé par des *resource brokers* individuels, tels que *AppLeS* ou *Nimrod/G*.

2.4.2 AppLeS

AppLeS [60] est un système d'ordonnancement centré sur les applications, utilisable dans un environnement de grille, et visant à permettre à chacune des applications d'atteindre ses objectifs en termes de performances.

Pour cela, *AppLeS* utilise une politique d'ordonnancement décentralisé et non-coopératif, basée sur l'utilisation d'agents individuels. Chaque application soumise possède son propre

²Le procédé de *co-allocation* consiste à allouer des ressources provenant de différents sites à une seule et même application.

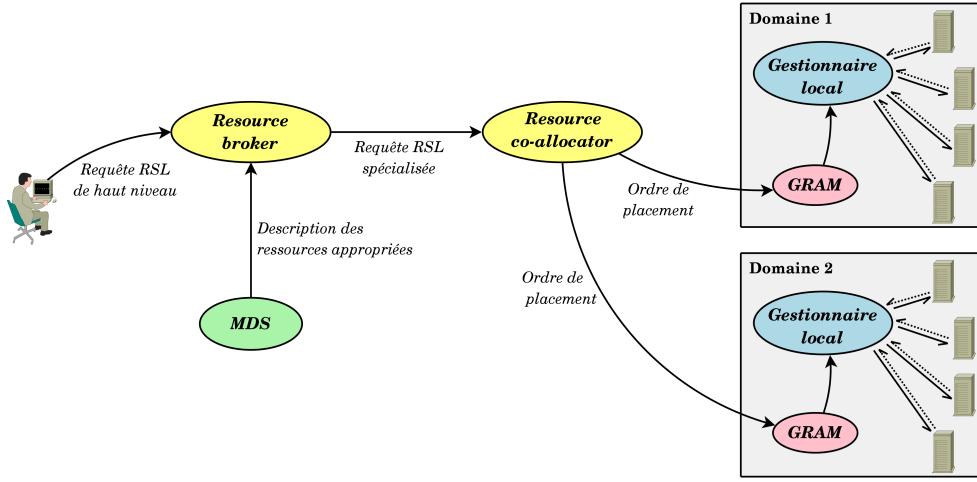


FIG. 2.10 – Fonctionnement du gestionnaire de ressources de *Globus*

agent, qui est alors chargé de trouver l’ordonnancement optimal pour les différentes tâches de l’application. Pour cela, il choisit un ensemble de ressources à partir d’informations sur l’application et le système (caractéristiques de l’application, objectifs de performances, ressources disponibles, etc.), et évalue les différentes possibilités d’ordonnancement en fonction de leur impact sur l’application.

AppLeS n’étant qu’un système d’ordonnancement, il utilise les services d’autres systèmes de gestion de ressources, comme *Globus* par exemple, afin de permettre l’exécution effective des tâches de l’application.

La figure 2.11 présente la structure des agents d’ordonnancement. On peut ainsi distinguer les cinq principaux composants de ces derniers :

- Le **resource selector** choisit différentes combinaisons de ressources pour l’exécution de l’application.
- Le **planner** (planificateur) génère un ordonnancement spécifique à chacune des combinaisons de ressources sélectionnées.
- Le **performance estimator** estime les performances de l’application pour chacun des ordonnancements déterminés, selon des métriques fournies par l’utilisateur.
- Le **coordinator** gère les différentes étapes de l’ordonnancement réalisées par les trois composants précédents, et détermine finalement l’ordonnancement optimal.
- L’**actuator** s’interface finalement avec le gestionnaire de ressources sous-jacent afin de réaliser le placement correspondant au meilleur ordonnancement trouvé.

L’outil d’observation *NWS* (cf. section 2.3.2) est utilisé pour recueillir les informations dynamiques sur les ressources, et pour en prédire la charge sur la période de temps concernée par le placement de l’application soumise.

De plus, l’utilisateur fournit des informations spécifiques sur la structure de l’application, ainsi que différentes caractéristiques la concernant. Il doit en outre indiquer quels sont les critères de performances retenus, de même que les éventuelles contraintes et préférences sur l’exécution.

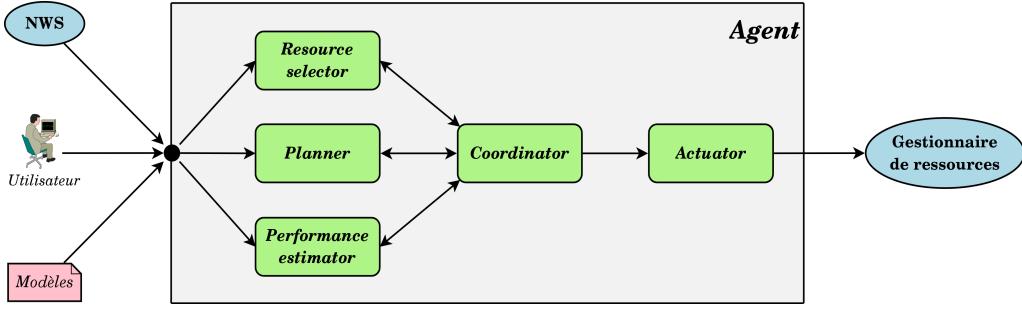


FIG. 2.11 – Structure des agents d’ordonnancement du système *AppLeS*

Enfin, des modèles sont utilisés pour estimer les performances des applications. Ces modèles spécifient la structure des applications, permettant ainsi de les regrouper en différentes catégories, telles que les applications paramétriques ou encore les applications maître/esclave.

2.4.3 Nimrod/G et le framework GRACE

Nimrod/G [61] est un système d’ordonnancement utilisant des paradigmes économiques pour placer des applications sur une grille de calcul. Les applications considérées ici sont uniquement des applications paramétriques.

Tout comme *AppLeS*, *Nimrod/G* est associé à un intergiciel tel que *Globus* pour réaliser des tâches comme la découverte des ressources, ou bien le placement effectif des applications. De plus, *Nimrod/G* fait partie du framework *GRACE* (*Grid Architecture for Computational Economy*), qui implémente l’infrastructure économique nécessaire pour l’échange de ressources. *GRACE* est ainsi considéré par les auteurs comme un intergiciel pouvant co-exister avec d’autres intergiciels de type *Globus*.

L’ordonnancement est réalisé selon deux politiques distinctes [62]. La première consiste à minimiser le temps d’exécution de l’application, tout en respectant des contraintes de budget et de *deadline*. La seconde consiste à minimiser le coût de l’application, en respectant ces deux mêmes contraintes.

L’utilisateur spécifie à la fois la date à laquelle il désire que l’application se termine au plus tard (*deadline*), ainsi que son coût maximal (budget). Ces deux paramètres peuvent s’exprimer de la manière suivante :

- $budget = C_{MIN} + B_{FACTOR} \times (C_{MAX} - C_{MIN})$
- $deadline = T_{MIN} + D_{FACTOR} \times (T_{MAX} - T_{MIN})$

Dans ces expressions, C_{MIN} et C_{MAX} indiquent respectivement le coût associé à la ressource la moins chère et à la ressource la plus chère pouvant exécuter l’ensemble des tâches de l’application. De même, T_{MIN} et T_{MAX} représentent le temps nécessaire pour exécuter l’ensemble de l’application sur le processeur le plus lent et sur le plus rapide.

L’utilisateur choisit le type de placement qu’il désire (minimisation du temps d’exécution ou du coût). Pour cela, il peut ajuster la valeur des coefficients B_{FACTOR} et D_{FACTOR} :

des valeurs de B_{FACTOR} proches de 1 indiquent qu'il est prêt à payer autant que nécessaire pour exécuter son application, de même que si D_{FACTOR} tend vers 1, cela signifie alors qu'il n'accorde que très peu d'importance à la durée de l'application. Il est possible de noter que si l'un des coefficients devient négatif, l'application ne pourra pas s'exécuter du fait des contraintes qu'elle imposerait dans ce cas.

Enfin, chaque ressource est associée à un prix, duquel l'utilisateur doit s'acquitter afin de pouvoir l'employer. Le modèle économique utilisé est ainsi un modèle de marché.

2.4.4 Le gestionnaire de ressources OAR

OAR [63] est le gestionnaire de ressources employé sur la plate-forme nationale de recherche Grid'5000. Il implémente diverses fonctionnalités telles que la gestion des priorités des applications, la réservation de ressources ou encore le *backfilling*.

Les priorités sont gérées à l'aide d'un système de files d'attente. Chaque file possède ses propres règles d'admission d'applications, sa propre politique d'ordonnancement, et est associée à une priorité. Un module, le meta-ordonnanceur, gère alors l'ordre de priorité des applications en fonction de la file où elles se trouvent.

La politique d'ordonnancement utilisée pour placer des applications sur Grid'5000 est de type FIFO avec *backfilling*, pour laquelle les réservations de ressources sont autorisées. Lors de la soumission d'une application, l'utilisateur peut exprimer des besoins tels que le nombre de ressources nécessaires à l'application, le temps d'exécution prévu, ou les caractéristiques physiques des machines. Il peut également spécifier une date de démarrage pour l'application : si suffisamment de ressources sont nécessaires à ce moment-là pour la durée désirée, l'application est effectivement placée. Si aucune date n'est spécifiée, OAR cherche le premier créneau horaire où l'application pourra s'exécuter.

Deux types d'applications sont considérées par OAR : les applications interactives et les applications *batch*. Dans le premier cas, l'utilisateur a la possibilité d'interagir avec le système grâce à une connexion de type *ssh*. Il peut alors lancer les commandes qu'il désire sur chacun des nœuds réservés.

OAR fonctionne en mode non-partagé : chaque nœud de calcul ne peut être réservé que par un seul utilisateur à la fois. Ceci est dû à la philosophie de Grid'5000 : mettre à disposition des chercheurs un environnement stable, qu'ils peuvent entièrement contrôler, pour exécuter leurs expériences. L'accès exclusif aux ressources permet de s'assurer que les expériences menées ne seront pas perturbées par des éléments extérieurs.

Architecture d'OAR

OAR possède une architecture modulaire (figure 2.12), où chaque module est responsable de tâches particulières. Ces modules s'appuient sur l'utilisation d'une base de donnée relationnelle, telle que *MySQL*, qui leur permet de communiquer entre eux. Même si OAR est écrit en langage *Perl*, l'utilisation d'une telle base de données ne contraint pas le développement de modules additionnels dans ce langage.

Les différents modules qui composent OAR sont les suivants :

- **Almighty** : Ce module constitue le serveur OAR. Il décide des actions qui doivent être effectuées, en fonction des informations ou commandes provenant des utilisateurs, de leurs programmes, ou bien des autres modules d’OAR. Lorsqu’une action doit être entreprise, Almighty fait appel au module concerné pour la réaliser.
- **Sarko** : Ce module est périodiquement exécuté par Almighty dans le but de surveiller les applications en cours d’exécution, et d’en demander la terminaison si leur date de fin a expirée (dans le cas où l’utilisateur n’a pas réservé suffisamment de temps pour permettre la fin de son application).
- **Judas** : Le rôle de ce module est de réaliser les tâches de journalisation (*loggin* en anglais), en affichant ou consignant les messages de débogage (*debug*), d’avertissement (*warning*) ou d’erreur (*error*).
- **Leon** : Ce module est invoqué lorsqu’un job doit être tué, que ce soit à la demande de l’utilisateur ou bien d’un autre module.
- **NodeChangeState** : Il est responsable du changement de l’état des ressources, ainsi que de la vérification de la présence de jobs sur celles-ci.
- **Scheduler** : Le rôle de ce module est de réaliser les tâches d’ordonnancement d’OAR, et de réserver les ressources demandées par les applications soumises. Notre objectif est ainsi d’ajouter une nouvelle politique d’ordonnancement s’appuyant sur des critères économiques à celles déjà implémentées.
- **Runner** : Ce module est responsable du lancement effectif des applications sur les ressources réservées, au moment où elles doivent être exécutées.

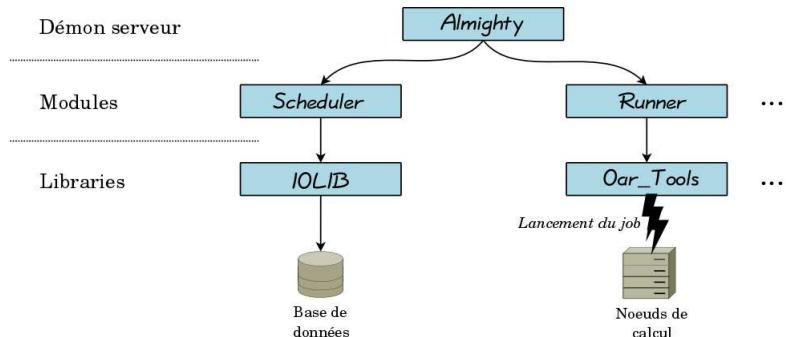


FIG. 2.12 – Hiérarchie modulaire d’OAR

Fonctionnement d’OAR

Examinons le fonctionnement d’OAR, en ce qui concerne la soumission d’une application (figure 2.13). Dans un premier temps, l’utilisateur se connecte à la machine possédant les utilitaires clients de OAR (tels que l’outil de soumission `oarsub` ou celui de monitoring `oarstat`). Cette machine est qualifiée de “frontale”.

Dans un second temps, la commande `oarsub` est invoquée afin de soumettre l’application à OAR. L’utilisateur demande à cette occasion un nombre de ressources fixe, une durée fixe et une date de début optionnelle dans le cadre d’une soumission à réservation explicite.

Si la réservation est implicite, OAR choisit lui-même la date de début de l'application, correspondant au premier créneau horaire où les ressources demandées sont disponibles.

L'outil `oarsub` valide alors syntaxiquement la requête, puis se connecte au serveur de base de données pour vérifier la disponibilité des ressources demandées. Il informe éventuellement l'utilisateur de leur indisponibilité (pour une réservation explicite) en indiquant le prochain créneau horaire libre.

Enfin, si la requête est valide, `oarsub` envoie une commande de soumission au serveur OAR, qui se chargera du choix des noeuds, de leur réservation et de l'exécution de l'application. Notons que cette instance d'exécution est appelée "job".

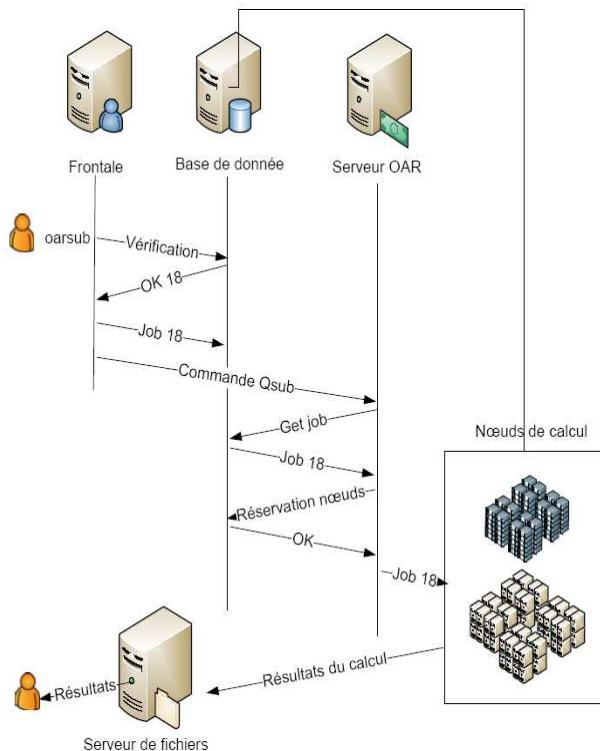


FIG. 2.13 – Etapes réalisées lors de la soumission d'une application sur OAR

2.5 Conclusion

La gestion des ressources dans un système distribué tel que les grilles de calcul prend plusieurs aspects. Elle doit assurer d'une part un service d'information permettant de recueillir des données sur les ressources présentes, et d'autre part un service d'ordonnancement ainsi qu'un service d'exécution.

Ce chapitre a mis en avant certains outils utilisés dans ce contexte. Des outils, tels que *NWS* et *Ganglia*, permettent d'observer l'état des ressources de la grille. Des systèmes

d'ordonnancement tels que *AppLeS* ou *Nimrod/G* peuvent être intégrés à des intergiciels comme *Globus* pour obtenir un système de gestion de ressources complet.

L'utilisation de paradigmes économiques pour gérer les ressources d'une grille a également été introduite. *Nimrod/G* est un exemple récent d'ordonnanceur s'appuyant sur de tels paradigmes. Il implémente un modèle de marché, pour lequel le prix des ressources est fixe. Le but est alors de trouver un ordonnancement qui minimise soit le coût de l'application, soit son temps d'exécution. D'autres systèmes basés sur des modèles économiques existent, tels que *Spawn* ou *POPCORN*. Ces modèles s'appuient sur une vente aux enchères des ressources aux différentes applications.

Un des objectifs de cette thèse consiste à créer un modèle d'ordonnancement basé sur des considérations économiques, et de l'intégrer au gestionnaire de ressources OAR. Il en résulte alors un gestionnaire de ressources complet et entièrement fonctionnel, utilisant une économie de marché pour attribuer des ressources aux applications soumises.

L'originalité des travaux menés réside dans la combinaison des caractéristiques suivantes :

- La grille fonctionne en mode dédié : les ressources sont dédiées aux activités de la grille, comme cela peut être le cas pour les grilles utilisées en mode ASP ou pour Grid'5000.
- Les ressources fonctionnent en mode partagé : plusieurs applications provenant de différents clients pourront utiliser une même ressource au même moment.
- Un modèle de prédiction des performances des ressources est intégré à l'ordonnanceur.
- Le modèle économique utilisé est un modèle de marché, dont le coût des ressources peut varier au cours du temps.
- Un compromis entre la date de terminaison de l'application et son coût est minimisé. L'importance relative de ces deux paramètres est spécifiée par l'utilisateur.
- L'ordonnancement détermine à la fois les ressources à utiliser, ainsi que la date à laquelle l'application devra être démarrée.

Chapitre 3

Conception et implémentation d'un modèle économique

Ce chapitre propose un modèle économique pour la gestion des ressources d'une grille de calcul. Il sera présenté sous la forme d'un problème d'optimisation, pour lequel nous définirons toutes les variables entrant en jeu, ainsi que les équations les mettant en relation.

Ce modèle mathématique sera implanté sous la forme d'un algorithme génétique, puis intégré dans un environnement opérationnel, afin d'inclure cet aspect économique à un ordonnanceur existant. Nous pourrons alors valider les résultats fournis par le modèle économique défini dans ce chapitre, et évaluer les performances d'un tel système.

3.1 Introduction

Avant de définir le modèle mathématique sur lequel repose le modèle économique proposé, et d'en fournir une implémentation, nous allons en premier lieu présenter ses objectifs, ainsi que le contexte auquel il s'applique. Ensuite, nous détaillerons l'ensemble de ses caractéristiques.

3.1.1 Objectifs et contexte

L'objectif principal du modèle économique décrit dans cette thèse est d'offrir un cadre pour la gestion des ressources d'une grille de calcul. Il s'agit ainsi de proposer un modèle permettant de mettre en relation les fournisseurs des ressources de la grille avec leurs consommateurs (les utilisateurs de la grille) [41], et de réguler ainsi l'offre et la demande sur ces ressources.

Même si cela n'est pas obligatoire, il est possible que les différents fournisseurs de ressources soient également les consommateurs de ces ressources. Dans un tel cas, le modèle économique pourra être assimilé à un modèle d'échange entre les différents intervenants. La monnaie utilisée peut être, au choix, une monnaie réelle ou bien une monnaie virtuelle (jetons permettant d'acheter l'accès aux ressources). Dans tous les cas, fournisseurs et consommateurs ont chacun leurs propres objectifs qu'il convient de satisfaire au mieux. Ainsi, les premiers chercheront à maximiser leurs profits, tandis que les seconds chercheront à trouver le meilleur compromis entre la qualité de service espérée et le coût que cela représente [64].

Nous supposons que la grille est totalement dédiée aux applications placées par le biais d'un ordonnanceur basé sur le modèle économique défini ici. Il ne devra donc y avoir aucun autre moyen de lancer une quelconque application. Cela interdit l'utilisation de notre modèle sur des grilles, telles que celle du *SETI@home* [65], pour lesquelles l'utilisation des ressources est partagée entre les applications lancées par le biais de la grille (calcul scientifique par exemple) et d'autres applications souvent lancées par leurs propriétaires (applications bureautiques, par exemple, si la machine est mise à disposition sur la grille par un particulier). Les grilles auxquelles le modèle économique proposé peut s'appliquer sont donc entièrement dédiées, comme cela peut être le cas dans le cadre des grilles orientées vers la prestation de services applicatifs, ou ASP pour *Application Service Provider* [18]. Cela permet d'avoir un parfait contrôle sur l'utilisation des ressources de la grille, et sur les applications qu'elles exécuteront. Enfin, s'agissant d'une grille de calcul, les ressources considérées peuvent bien sûr être hétérogènes.

Le modèle économique issu des travaux présentés dans cette thèse ne prend en compte que les ressources de type processeur dans les critères d'ordonnancement. Il constitue une extension du modèle que nous avons proposé lors du 7ème Congrès des Doctorants EDSYS [50], puisque, en plus de l'état des ressources de la grille au moment du calcul d'un ordonnancement, il prend en compte leur état futur pour effectuer ce calcul. Ceci introduit en outre le besoin de pouvoir estimer les évolutions futures auxquelles seront soumises les ressources.

3.1.2 Caractéristiques du modèle proposé

Le modèle économique proposé permet le placement d'applications parallèles sur une grille de calcul. Ces applications sont constituées de plusieurs tâches pouvant s'exécuter en parallèle sur différents processeurs. Nous supposerons que toutes les tâches d'une même application sont identiques et synchrones. Elles débuteront leur exécution simultanément, et communiqueront par le biais de l'échange de messages via un réseau, et ce de manière synchrone. De plus, seules des applications à gros grain¹ seront considérées [66].

Le rôle du modèle économique est donc, à l'instar de tout modèle d'ordonnancement, de placer de telles applications sur une grille de calcul, c'est-à-dire d'associer chacune des tâches de ces applications aux processeurs de la grille. L'originalité de l'approche économique tient au fait qu'aux critères de rapidité d'exécution souvent utilisés par les ordonnanceurs existants, s'ajouteront des critères de coût.

Une des principales caractéristiques du modèle proposé est la prise en compte du temps. Ainsi, les choix effectués par l'ordonnanceur ne dépendent pas uniquement de l'état actuel de la grille, mais également des variations futures de cet état, qui sont connues au moment du calcul de l'ordonnancement. Ceci permet, en plus de déterminer un placement pour les tâches d'une application à exécuter, de calculer le meilleur moment où exécuter cette application.

¹La granularité d'une application parallèle représente le rapport entre la quantité de calculs effectués et la quantité de communications. Une application à gros grain est donc une application pour laquelle des messages sont échangés de manière sporadique au terme de longues périodes de calcul. Ainsi, dans ce cas, le temps passé en communication peut être négligé devant celui passé en calcul.

Le modèle économique défini dans ce chapitre est basé sur un modèle de marché [34, 67, 61, 49, 50]. Le prix des ressources n'est pas fixe, puisqu'il peut varier au cours du temps, en fonction de l'offre et la demande par exemple. Cependant, l'évolution du prix au cours du temps est connue au moment de l'ordonnancement, afin d'inclure ces variations au résultat produit par notre modèle. Le modèle de prix inclura divers paramètres, tels que :

- le nombre de ressources utilisées,
- les caractéristiques de ces ressources,
- leur durée d'utilisation,
- le moment d'utilisation.

Le modèle prend en compte les besoins des fournisseurs de ressources et des consommateurs de plusieurs manières. Tout d'abord, il permet aux fournisseurs de ressources d'agir sur les prix des ressources. Ils peuvent ainsi augmenter ou diminuer, au choix, le prix de n'importe quel processeur, en fonction de caractéristiques des machines qui ne sont pas prises en compte dans notre modèle (la présence de périphériques ou de logiciels spécifiques par exemple). Ils peuvent également, pour un processeur donné, définir des variations de son coût au cours du temps, leur offrant ainsi un moyen de mieux contrôler l'offre et la demande. Cela leur permettra de mettre en place, par exemple, un système d'heures pleines le jour et d'heures creuses la nuit.

Les utilisateurs des ressources pourront choisir de privilégier, dans le calcul de l'ordonnancement, une qualité de service basée sur l'économie (en optant pour les ressources les moins chères), ou bien basée sur la rapidité d'exécution pour avoir un résultat au plus tôt, tel que cela a été fait dans les travaux de Rajkumar Buyya [68, 61]. Le modèle économique présenté dans ce chapitre calculera des ordonnancements offrant aux utilisateurs un compromis entre le coût de leurs applications et la rapidité d'exécution de ces dernières, compromis dont ils auront spécifié à l'avance les proportions.

Notons enfin que chaque tâche d'une application soumise sera, à l'issue du calcul de l'ordonnancement, affectée à un et un seul processeur. Il n'y aura ensuite pas de migration possible. De plus, les ressources fonctionnent en mode partagé. Cela signifie que plusieurs tâches, provenant d'applications différentes, peuvent être exécutées en parallèle au même moment sur un processeur. Ainsi, à l'inverse d'ordonnanceurs existants, il n'y a aucun mécanisme de réservation de ressources. Seule la charge des ressources peut constituer un frein à leur utilisation. Cette charge doit donc être mise à jour à chaque fois qu'une application est placée sur la grille, afin de prendre en compte, lors des soumissions suivantes, les ressources qu'elle utilisera.

3.2 Modèle mathématique

Le modèle économique présenté dans ce chapitre repose sur un modèle mathématique mettant en relation les caractéristiques intrinsèques des ressources de la grille avec les besoins requis par les applications à placer sur celle-ci [69].

3.2.1 Définition des variables utilisées

3.2.1.1 Aspects temporels

Le modèle économique défini dans ce chapitre permet de déterminer un ordonnancement pour les diverses tâches constituant l'application à placer. Le calcul de cet ordonnancement

prend en compte l'état de la grille au moment où il est effectué, mais il tient compte également de l'état futur de la grille. Ainsi, il est nécessaire d'inclure la gestion du temps au sein de notre modèle.

Nous choisissons de discréteriser le temps en intervalles de durées égales. Les données qui varient au cours du temps sont considérées comme constantes sur chacun de ces intervalles.

Les variables suivantes sont ainsi définies :

- N^K = Nombre d'intervalles de temps considérés.

$$\hookrightarrow N^K \in \mathbb{N}^*$$

- D_k = Date de début de l'intervalle k , avec $k \in [1; N^K]$.

$$\hookrightarrow \forall k \in \{1, \dots, N^K\} \quad D_k \in \mathbb{R}_+$$

- ΔT = Durée de chacun des intervalles de temps.

$$\hookrightarrow \Delta T \in \mathbb{R}_+^*$$

Ainsi, le temps est découpé en N^K intervalles $[D_k; D_k + \Delta T]_{1 \leq k \leq N^K}$. On a :

$$\begin{aligned} \forall k \in \{1, \dots, N^K\} \quad D_k &= (k - 1) \cdot \Delta T \\ \implies &\left\{ \begin{array}{l} D_1 = 0 \\ \forall k \in \{2, \dots, N^K\} \quad D_k = D_{k-1} + \Delta T \end{array} \right. \end{aligned}$$

La date $D_1 = 0$ correspond au moment où l'ordonnancement est calculé, ce qui constitue le point de départ de la période sur laquelle celui-ci est cherché.

La période de temps couverte pour le calcul d'un ordonnancement est choisie par les administrateurs de la grille. Elle peut être de différents ordres de grandeur, allant d'une journée à une semaine, voire plus.

3.2.1.2 Données du problème

Les variables définies ici sont toutes connues au début de l'ordonnancement des différentes tâches de l'application à placer sur la grille. Elles peuvent être constantes, ou bien elles peuvent varier au cours du temps.

Dans le cas de données variant au cours du temps, on considère que leurs variations futures sont connues ou peuvent être estimées, et qu'ainsi on connaît la valeur de ces données sur chacun des N^K intervalles de temps. Ceci suppose une parfaite maîtrise de l'usage des ressources de la grille, ce qui est permis grâce à l'hypothèse d'une grille dédiée formulée précédemment.

Caractéristiques et état de la grille

Une partie des données du problème permet de décrire la grille de calcul utilisée :

- N_A^P = Nombre de processeurs disponibles.

$$\hookrightarrow \quad N_A^P \in \mathbb{N}^*$$

- P_j = Puissance de chaque processeur disponible, avec $j \in [1; N_A^P]$. La puissance d'un processeur peut être obtenue à l'aide de *benchmarks*² tels que *SPECint*, *SPECfp* ou bien *MFLOPS*.

$$\hookrightarrow \quad \forall j \in \{1, \dots, N_A^P\} \quad P_j \in \mathbb{R}_+^*$$

- $L_{j,k}$ = Charge des processeurs, avec $j \in [1; N_A^P]$ et $k \in [1; N^K]$. $L_{j,k}$ représente ainsi la charge du processeur j durant l'intervalle de temps $[D_k; D_k + \Delta T[$. Nous pouvons noter que cette charge est entièrement prévisible, la grille étant exclusivement dédiée aux applications placées par le biais d'un ordonnanceur basé sur notre modèle économique. Elle ne dépend que des applications déjà placées au moment de l'ordonnancement (applications en cours d'exécution et applications planifiées pour une exécution future), et peut ainsi être recalculée à chaque fois qu'une nouvelle application est placée sur la grille.

$$\hookrightarrow \quad \forall j \in \{1, \dots, N_A^P\}, \forall k \in \{1, \dots, N^K\} \quad L_{j,k} \in [0; 1] \subset \mathbb{R}$$

Caractéristiques de l'application à placer

Les propriétés de l'application à placer sont exprimées à l'aide des variables suivantes :

- A = Nombre de tâches de l'application.

$$\hookrightarrow \quad A \in \mathbb{N}^*$$

- P^R = Puissance du processeur de référence. De la même manière que ce qui est décrit dans [44], lorsque l'utilisateur effectue une requête de placement d'application, il doit fournir une estimation du temps d'exécution de celle-ci, en prenant pour référence cette puissance. Cette puissance est estimée de la même manière que la puissance des processeurs constituant la grille de calcul.

$$\hookrightarrow \quad P^R \in \mathbb{R}_+^*$$

- T^R = Temps d'exécution global de l'application demandé par l'utilisateur. Ce temps correspond au temps d'exécution cumulé de chacune des tâches de l'application, et est exprimé par rapport à la puissance de référence P^R .

$$\hookrightarrow \quad T^R \in \mathbb{R}_+^*$$

- T^C = Estimation du temps moyen passé par chaque tâche en communication. Cette valeur dépend du nombre de messages échangés et de leur taille, et est donnée pour une bande-passante du réseau nominale.

$$\hookrightarrow \quad T^C \in \mathbb{R}_+^*$$

²Un test de performance, ou *benchmark*, est un test dont l'objectif est de déterminer les performances d'un système informatique.

- N_{min}^P et N_{max}^P = Bornes imposées par l'utilisateur sur le nombre de processeurs à utiliser.

$$\hookrightarrow \quad N_{min}^P \in \mathbb{N}^* \quad \text{et} \quad N_{max}^P \in \mathbb{N}^* \quad \text{avec} \quad N_{min}^P \leq N_{max}^P$$

Paramètres de coût

Les ressources que le modèle économique prend en compte sont celles qui sont liées aux processeurs, c'est-à-dire :

- le temps processeur consommé,
- le nombre de processeurs utilisés,
- la puissance des processeurs utilisés.

Selon le mode de fonctionnement de la grille, leurs coûts sont fixés par les propriétaires des ressources ou par les administrateurs de la grille. Ces paramètres de coût sont représentés par les variables suivantes :

- C^T = Coefficient permettant de pondérer le coût correspondant au temps processeur consommé.

$$\hookrightarrow \quad C^T \in \mathbb{R}_+$$

- C^N = Coefficient permettant de pondérer le coût correspondant au nombre de processeurs utilisés.

$$\hookrightarrow \quad C^N \in \mathbb{R}_+$$

- C^P = Coefficient permettant de pondérer le coût correspondant à la puissance des processeurs utilisés.

$$\hookrightarrow \quad C^P \in \mathbb{R}_+$$

- $C_{j,k}^P$ = Coefficient appliqué sur le coût de chaque processeur, avec $j \in [1; N_A^P]$ et $k \in [1; N^K]$. Ainsi, $C_{j,k}^P$ représente le coefficient appliqué au coût lié à la puissance du processeur j durant l'intervalle $[D_k; D_k + \Delta T[$. Il permet ainsi d'ajuster localement le coût de chaque processeur, dans le but d'offrir aux administrateurs un moyen de contrôler l'offre et la demande sur ces processeurs.

$$\hookrightarrow \quad \forall j \in \{1, \dots, N_A^P\}, \forall k \in \{1, \dots, N^K\} \quad C_{j,k}^P \in \mathbb{R}_+$$

3.2.1.3 Inconnues recherchées

Les variables décrites dans cette section sont calculées à partir des données du problème, dans le but de déterminer un ordonnancement pour l'application à placer sur la grille. Ces variables fournissent toutes les informations nécessaires au placement de l'application :

- u_j = Nombre de tâches de l'application à lancer sur le processeur j , avec $j \in [1; N_A^P]$. Les tâches de l'application à placer étant supposées identiques, cette inconnue fournit ainsi l'ordonnancement calculé par notre modèle.

$$\hookrightarrow \quad \forall j \in \{1, \dots, N_A^P\} \quad u_j \in \mathbb{N}$$

- N_U^P = Nombre de processeurs utilisés. Cette variable est déduite à partir du vecteur u , et doit être bornée par les variables N_{min}^P et N_{max}^P .

$$\hookrightarrow N_U^P \in \mathbb{N}^*$$

- k^S = Indice de l'intervalle de temps au cours duquel l'application sera exécutée ; l'intervalle ainsi désigné est $[D_{ks}; D_{ks} + \Delta T[$. Cette variable complète ainsi l'ordonnancement fourni par le vecteur u , en précisant le moment où l'application devra être lancée sur les processeurs contenus dans ce vecteur.

$$\hookrightarrow k^S \in [1; N^K] \subset \mathbb{N}$$

- t^S = Date de début de l'application. Cette inconnue représente la même entité que k^S , à ceci près qu'elle exprime une date, c'est-à-dire un temps à partir de la date $D_1 = 0$, et non un intervalle.

$$\hookrightarrow t^S \in \mathbb{R}_+$$

- C_{App} = Coût de l'application. Il dépendra à la fois des processeurs choisis, ainsi que de la date de début de l'application.

$$\hookrightarrow C_{App} \in \mathbb{R}_+$$

- T_{App} = Date de fin de l'application. Il s'agit de la date à laquelle la dernière tâche de l'application se terminera. Cette inconnue dépend également des processeurs choisis ainsi que de la date de début de l'application.

$$\hookrightarrow T_{App} \in \mathbb{R}_+^*$$

3.2.2 Modélisation sous forme de problème d'optimisation

L'objectif du modèle présenté dans ce chapitre est de calculer un placement complet pour l'application, c'est-à-dire à la fois un placement que l'on peut qualifier de spatial (choix des processeurs) et un placement temporel (date de début d'exécution). Ce placement doit permettre un compromis, tel que spécifié par l'utilisateur, entre le coût de l'application et sa date de terminaison.

Ainsi, le modèle économique peut être assimilé à un problème d'optimisation, où les inconnues cherchées seront calculées de manière à minimiser un tel compromis.

La fonction objectif du modèle est donc une somme pondérée entre le coût et la date de fin de l'application. Les poids sur ces deux paramètres, que nous notons respectivement W^C et W^T , sont exprimés par l'utilisateur, en fonction de ce qu'il préfère favoriser lors du calcul du placement.

La fonction objectif peut donc s'exprimer de la manière suivante :

$$Target = W^C \cdot C_{App} + W^T \cdot T_{App} \quad (3.1)$$

avec : $W^C \in \mathbb{R}_+$ et $W^T \in \mathbb{R}_+$

Le vecteur de placement u et la date de début de l'application t^S seront ainsi calculés de manière à minimiser cette fonction objectif.

Notons enfin que le temps d'exécution de l'application n'est pas nécessairement optimisé. Seule la date de fin de l'application le sera, puisqu'il s'agit de ce qui intéresse réellement l'utilisateur.

3.2.3 Equations régissant le modèle

3.2.3.1 Relations entre les inconnues recherchées et contraintes

Lors du calcul du placement d'une application sur la grille de calcul, la minimisation de la fonction objectif est soumise à diverses contraintes sur les inconnues recherchées.

Ainsi, il est tout d'abord évident que le nombre de processeurs utilisés dépend directement du vecteur de placement u . On peut définir le nombre de processeurs utilisés comme étant le nombre de processeurs pour lesquels le nombre de tâches qui leur sont affectées est non-nul :

$$N_U^P = \text{card} (\{j \in [1; N_A^P] / u_j \neq 0\})$$

De plus, le nombre de processeurs utilisés doit être compris entre les bornes spécifiées par l'utilisateur, et ne doit bien entendu pas dépasser les capacités de la grille de calcul :

$$N_{min}^P \leq N_U^P \leq N_{max}^P \leq N_A^P$$

On peut également noter que le cumul du nombre de tâches lancées sur les différents processeurs doit correspondre au nombre total de tâches de l'application à placer :

$$\sum_{j=1}^{N_A^P} u_j = A$$

Cette contrainte garantit de fait que nous n'utiliserons pas plus de processeurs qu'il n'y a de tâches à exécuter :

$$N_U^P \leq A$$

Enfin, nous pouvons exprimer la date de début t^S de l'application en fonction de l'intervalle k^S au cours duquel elle sera exécutée :

$$t^S = D_{k^S} \iff t^S = (k^S - 1) \cdot \Delta T$$

3.2.3.2 Quantité de calculs d'une tâche

Avant de détailler les équations permettant le calcul du coût de l'application à placer ainsi que de sa date de terminaison, il est utile de définir la quantité de calculs d'une tâche.

On définit la quantité de calculs Q_{App} devant être effectuée par l'application à l'aide de la relation suivante :

$$Q_{App} = T^R \cdot P^R$$

Cette quantité est ainsi calculée grâce au temps d'exécution T^R de l'application sur un processeur de référence de puissance P^R et de charge nulle. Graphiquement, la figure 3.1 montre que Q_{App} correspond à l'aire du rectangle tracé.



FIG. 3.1 – Quantité de calculs devant être effectuée par l'application

Toutes les tâches de l'application à placer étant identiques, on peut ainsi exprimer la quantité de calculs Q devant être effectuée par chacune de ses tâches :

$$Q = \frac{T^R \cdot P^R}{A} \quad (3.2)$$

Cette entité permet de calculer le temps d'exécution t de chaque tâche sur un processeur de puissance donnée P_j et de charge nulle :

$$Q = t \cdot P_j \quad \Rightarrow \quad t = \frac{Q}{P_j} \quad \Rightarrow \quad t = \frac{1}{A} \cdot \frac{T^R \cdot P^R}{P_j} \quad (3.3)$$

La figure 3.2 montre la relation précédente sous la forme d'une simple loi de conservation de surface.

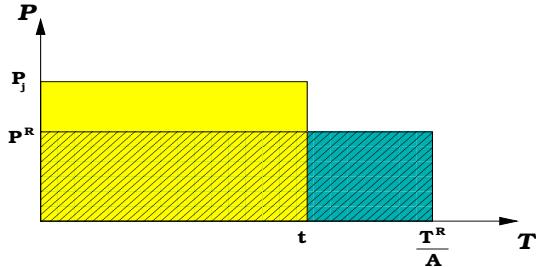


FIG. 3.2 – Détermination du temps d'exécution d'une tâche sur un processeur de puissance donnée et de charge nulle

Enfin, il est possible de calculer le temps d'exécution t' de chaque tâche sur un processeur de puissance donnée P_j , mais ayant une charge L_j non-nulle :

$$Q = t' \cdot P_j \cdot (1 - L_j)$$

$$t' = \frac{Q}{P_j \cdot (1 - L_j)} = \frac{t}{(1 - L_j)} \quad \Rightarrow \quad t' = \frac{1}{A} \cdot \frac{T^R \cdot P^R}{P_j \cdot (1 - L_j)}$$

La figure 3.3 illustre une telle relation. On constate sur cette figure que la quantité de calculs est effectuée sur le processeur en un temps t' , et sur la partie non-chargée du processeur (visuellement, sur le premier schéma, il s'agit de la bande située entre $P_j \cdot L_j$ et P_j). La charge étant un nombre réel compris entre 0 et 1, on peut se ramener au second schéma où la quantité de calculs s'exprime en fonction de $P_j \cdot (1 - L_j)$.

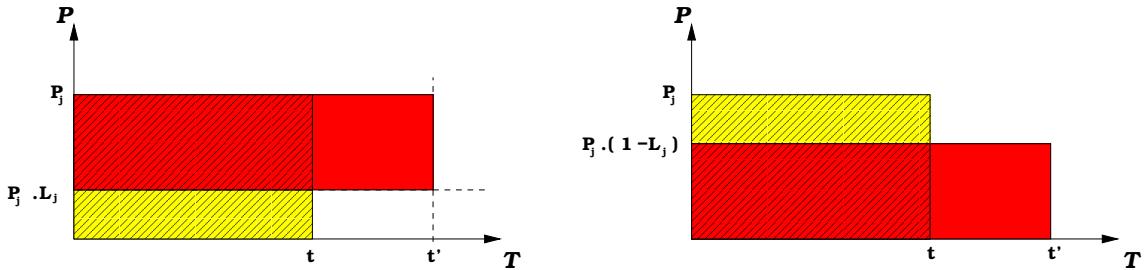


FIG. 3.3 – Détermination du temps d'exécution d'une tâche sur un processeur de puissance donnée de charge non-nulle

3.2.3.3 Calcul du temps d'exécution de l'application

Dans cette section, nous montrons comment le temps d'exécution réel d'une application à placer sur la grille peut être obtenu en fonction de l'ordonnancement choisi d'une part, et de la date de début de l'application d'autre part.

Calcul du nombre d'intervalles de temps nécessaires pour exécuter une tâche sur un processeur donné

Intéressons-nous tout d'abord à la quantité de calculs maximale pouvant être effectuée sur un processeur j durant un intervalle de temps $[D_k; D_k + \Delta T[$. On note cette quantité $q_{j,k}$. Cette valeur correspond donc à la quantité de calculs effectuée au cours de toute la durée ΔT d'un intervalle de temps, sur la partie non-chargée du processeur j . La durée de l'intervalle, la puissance du processeur, ainsi que la charge du processeur durant cet intervalle étant des données connues du problème, on a :

$$\forall j \in \{1, \dots, N_A^P\}, \forall k \in \{1, \dots, N^K\} \quad q_{j,k} = \Delta T \cdot P_j \cdot (1 - L_{j,k}) \quad (3.4)$$

Connaissant la quantité totale de calculs que chaque tâche doit effectuer (équation (3.2)), nous pouvons déterminer le nombre minimal d'intervalles de temps nécessaires à un processeur quelconque pour exécuter une tâche entière. Notons K_{j,k^S} le nombre minimal d'intervalles de temps nécessaires au processeur j pour exécuter une tâche de l'application à placer à partir de l'intervalle $[D_{k^S}; D_{k^S} + \Delta T[$. On a :

$$\forall j \in \{1, \dots, N_A^P\} \quad K_{j,k^S} = \min \left\{ K \mid \sum_{k=k^S}^{k^S+K-1} q_{j,k} \geq Q \right\}$$

Ainsi, K_{j,k^S} correspond au nombre d'intervalles de temps requis à partir de $[D_{k^S}; D_{k^S} + \Delta T[$ pour effectuer une quantité de calculs suffisante pour exécuter une tâche complète. En substituant $q_{j,k}$ et Q par leurs valeurs (équations (3.2) et (3.4)), on obtient :

$$\forall j \in \{1, \dots, N_A^P\} \quad K_{j,k^S} = \min \left\{ K \quad / \quad \sum_{k=k^S}^{k^S+K-1} \Delta T \cdot P_j \cdot (1 - L_{j,k}) \geq \frac{T^R \cdot P^R}{A} \right\} \quad (3.5)$$

Dans le cas où le processeur j ne doit exécuter qu'une seule tâche, celle-ci débutera au cours de l'intervalle $[D_{k^S}; D_{k^S} + \Delta T[$ tandis qu'elle se terminera durant l'intervalle $[D_{k^S+K_{j,k^S}-1}; D_{k^S+K_{j,k^S}-1} + \Delta T[$. On considère alors que sa date de début sera $t^S = (k^S - 1) \cdot \Delta T$ (correspondant à la date de début de l'intervalle où elle commence) et que sa date de fin sera $t_j^e = (k^S + K_{j,k^S}) \cdot \Delta T$ (date de fin de l'intervalle où elle se termine).

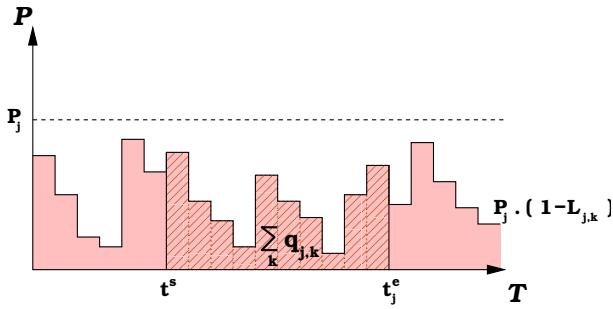


FIG. 3.4 – Intervalles de temps nécessaires à l'exécution d'une tâche sur un processeur donné

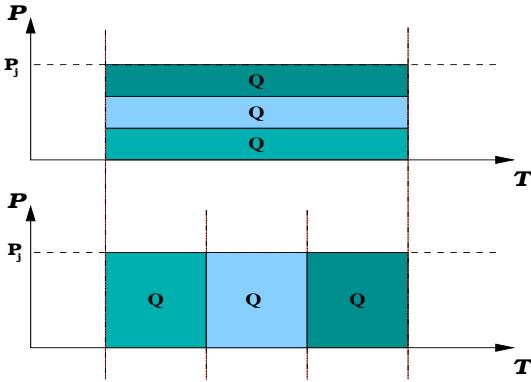
Il est enfin à noter que la date de début de la tâche t^S est indépendante du processeur j qui l'exécute, puisque toutes les tâches de l'application démarrent simultanément à cette date. Cependant, la date de fin t_j^e de la tâche dépendra du processeur j choisi pour l'exécuter (la puissance et la charge de ce dernier ayant une influence sur la rapidité d'exécution de la tâche).

Calcul du nombre d'intervalles de temps nécessaires pour exécuter un ensemble de tâches sur un seul processeur

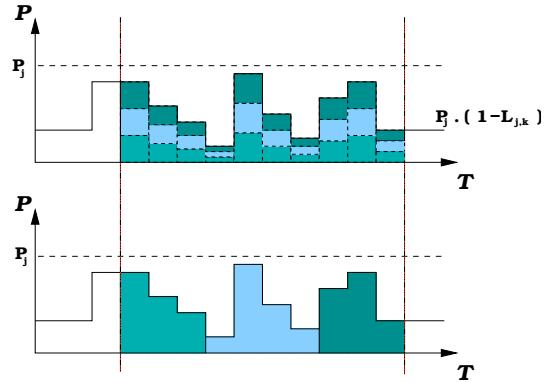
Supposons maintenant que plusieurs tâches doivent être exécutées sur un même processeur. Ceci peut arriver dans le cas où l'ordonnanceur choisit d'attribuer plusieurs tâches à un processeur j donné ($u_j > 1$). Dans la réalité, ces tâches seront lancées simultanément à la date de début de l'application, et elles s'exécuteront en parallèle. Cependant, une telle approche ne permet pas de calculer facilement le nombre d'intervalles de temps nécessaires à leur exécution.

Nous allons donc supposer que ces tâches s'exécutent séquentiellement (figure 3.5), de sorte que l'on puisse utiliser la formule donnée par l'équation (3.5). Même si cela ne correspond pas à la réalité, une telle hypothèse est justifiée dès lors que les tâches ne

communiquent pas entre elles. L'hypothèse émise sur l'utilisation d'applications à gros grain permet alors une telle approximation.



(a) Processeur possédant une charge initiale nulle



(b) Processeur possédant une charge initiale non-nulle

FIG. 3.5 – Mise en séquence de trois tâches parallèles placées sur un même processeur

Pour chaque processeur j utilisé (c'est-à-dire pour lequel $u_j \neq 0$), les tâches qu'il exécute sont numérotées de 1 à u_j . Notons $k_{i,j}^s$ l'indice de l'intervalle de début de la tâche i sur le processeur j , et $k_{i,j}^e$ celui de l'intervalle où la tâche i se termine. La première tâche ($i = 0$) commencera au début de l'intervalle $[D_{k^S}; D_{k^S} + \Delta T[$, tandis que les suivantes commenceront au début de l'intervalle suivant celui au cours duquel la tâche précédente se termine. La figure 3.6 montre ceci.

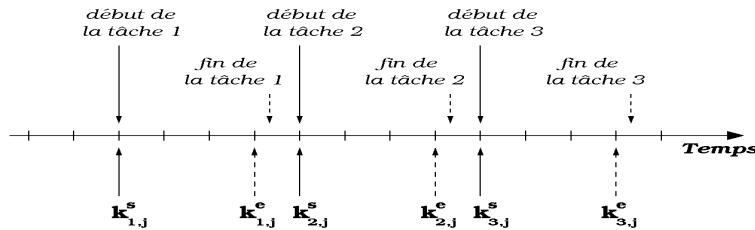


FIG. 3.6 – Enchaînement des dates de début et fin de trois tâches placées sur un même processeur

Ainsi, on peut écrire les relations suivantes :

$$\left\{ \begin{array}{ll} k_{1,j}^s = k^S & \\ \forall i \in \{1, \dots, u_j\} & k_{i,j}^e = k_{i,j}^s + K_{j,k_{i,j}^s} - 1 \\ \forall i \in \{2, \dots, u_j\} & k_{i,j}^s = k_{i-1,j}^e + 1 = k_{i-1,j}^s + K_{j,k_{i-1,j}^s} \end{array} \right.$$

On peut en déduire que la période d'activité d'un processeur j , en rapport avec l'application à placer, est comprise dans l'intervalle de temps $[D_{k^S}; D_{k_{u_j,j}^e + 1} + \Delta T[$ (cf. figure 3.6).

Calcul de la date de fin de l'application

L'intervalle de temps au cours duquel l'application se termine correspond au dernier intervalle où une de ses tâches termine son exécution (en d'autres termes, il s'agit de celui où la plus tardive des tâches, tous processeurs confondus, prend fin). Notons k^E l'indice de cet intervalle de temps ; celui-ci sera donc l'intervalle $[D_{k^E}; D_{k^E} + \Delta T[$. L'expression de k^E est la suivante :

$$k^E = \max_{j \in [1; N_A^P]} \left\{ k_{u_j, j}^e \right\} = \max_{j \in [1; N_A^P]} \left\{ k_{u_j, j}^s + K_{j, k_{u_j, j}^s} - 1 \right\}$$

En réalité, les tâches de l'application communiquent entre elles. Cependant, l'hypothèse formulée sur l'utilisation d'applications à gros grain permet de supposer que le temps de communication entre les tâches n'aura que peu d'impact sur le calcul de k^E .

Déterminons maintenant la date t^E à laquelle l'application se termine. Cette date correspond ainsi à la date de fin de l'intervalle $[D_{k^E}; D_{k^E} + \Delta T[$. On obtient donc :

$$t^E = D_{k^E} + \Delta T = (k^E - 1) \cdot \Delta T + \Delta T$$

$$t^E = k^E \cdot \Delta T$$

Cette date de fin ne considère que le temps de calcul de l'application. Nous pouvons désormais y inclure le temps passé en communication par chacune des tâches. Ceci permet d'avoir une valeur plus précise de la date de fin de l'application, et peut éventuellement permettre de tenir compte du débit des liens entre les processeurs dans le calcul de l'ordonnancement (par exemple en indexant ce temps de communication sur le débit réseau le plus faible parmi les processeurs sélectionnés). Notons t^{EC} la date de fin de l'application incluant le temps de communication des tâches qui la composent. Sa valeur est donnée par la relation suivante :

$$t^{EC} = t^E + T^C = k^E \cdot \Delta T + T^C$$

L'intervalle contenant cette date est l'intervalle $[D_{k^{EC}}; D_{k^{EC}} + \Delta T[$, où k^{EC} est défini comme suit :

$$k^{EC} = \frac{t^{EC}}{\Delta T} + 1$$

La fonction objectif du modèle économique décrit dans ce chapitre est un compromis entre le coût de l'application C_{App} et sa date de terminaison T_{App} (équation (3.1)). Nous assimilerons donc la date de fin T_{App} utilisée dans la fonction objectif du modèle avec la date de fin de l'application incluant les communications entre les tâches. On a donc :

$$T_{App} = t^{EC}$$

Les calculs effectués jusqu'ici permettent donc de déduire que la date de fin T_{App} considérée dans le critère de minimisation dépend, comme l'on pouvait s'y attendre, des différents paramètres énumérés ci-dessous :

- les caractéristiques de l'application (T^R, P^R, A, T^C),
- les caractéristiques ainsi que l'état de la grille (P_j et $L_{j,k}$, avec $j \in [1; N_A^P]$ et $k \in [1; N^K]$),
- l'ordonnancement choisi (inconnues u_j , avec $j \in [1; N_A^P]$),
- la date de début choisie (inconnue k^S).

Temps d'exécution de l'application

A titre informatif, il est intéressant de calculer le temps d'exécution de l'application. Même s'il ne rentre pas directement dans le calcul du critère à minimiser, le temps d'exécution est utilisé dans la section suivante pour calculer le coût de l'application.

Connaissant la date de fin de l'application, ainsi que sa date de début, la formulation de son temps d'exécution est très simple. Nous distinguons cependant deux temps d'exécution différents : le premier comprend les temps de communication des tâches, tandis que le second n'en tient pas compte et exprime ainsi le temps passé par l'application en calculs purs.

Le temps d'exécution de l'application comprenant les temps d'exécution des tâches est exprimé à l'aide de la relation suivante :

$$t_{exec}^{EC} = t^{EC} - t^S$$

Le temps passé par l'application en calculs purs est le suivant :

$$t_{exec}^E = t^E - t^S$$

La relation suivante est bien entendu vérifiée :

$$t_{exec}^{EC} = t_{exec}^E + T^C$$

3.2.3.4 Calcul du coût de l'application

Le coût d'une application placée sur la grille dépend de trois paramètres différents :

- le temps processeur consommé,
- le nombre de processeurs utilisés,
- la puissance des processeurs utilisés.

Le coût lié à ces trois paramètres sera par la suite pondéré par les coefficients C^T , C^N et C^P .

Coût lié au temps processeur consommé

Pour calculer ce coût, il est indispensable de connaître le temps processeur consommé par l'application. La section précédente nous a permis de déterminer le temps d'exécution de l'application. Malheureusement, même si l'on ne prend pas en compte les temps de communication entre les différentes tâches, le temps d'exécution ne peut pas être utilisé ici car il ne reflète pas le temps processeur réellement consommé. En effet, il dépend de la charge des processeurs, et il n'est pas envisageable d'inclure cette dernière dans le calcul du coût lié au temps processeur.

Ainsi, nous allons calculer le temps processeur réellement consommé par l'application, indépendamment de la charge des processeurs utilisés. Soit t_j^{CPU} , le temps CPU consommé par l'application sur le processeur j . Il correspond au cumul des temps CPU consommés par chaque tâche de l'application exécutée sur ce processeur. D'après l'équation (3.3), son expression est donnée par la relation suivante :

$$t_j^{CPU} = u_j \cdot \frac{Q}{P_j} = \frac{T^R \cdot P^R}{A} \cdot \frac{u_j}{P_j}$$

Le temps CPU ainsi consommé ne dépendra, pour un processeur donné, que du nombre de tâches placées sur celui-ci ainsi que de la quantité de calculs devant être effectuée par chacune de ces tâches.

Le temps processeur consommé par la totalité de l'application correspond au cumul des temps consommés sur chaque processeur de la grille :

$$t^{CPU} = \sum_{j=1}^{N_A^P} t_j^{CPU} = \frac{T^R \cdot P^R}{A} \cdot \sum_{j=1}^{N_A^P} \frac{u_j}{P_j}$$

Coût lié au nombre de processeurs utilisés

La composante du coût de l'application liée au nombre de processeurs utilisés est directement assimilée au nombre de processeurs utilisés par l'application, c'est-à-dire au nombre de processeurs j pour lesquels $u_j \neq 0$: N_U^P

Coût lié à la puissance des processeurs utilisés

Le coût lié à la puissance des processeurs utilisés dépend de deux paramètres principaux :

- la puissance des processeurs utilisés,
- les coefficients permettant d'ajuster localement le coût de chaque processeur. Ces coefficients permettent de mettre en place des variations sur le coût des processeurs, en augmentant ou en diminuant le prix de certains d'entre eux sur des périodes de temps déterminées.

On note C_{App}^P la composante du coût C_{App} de l'application liée à la puissance des processeurs utilisés. Elle est définie par la relation suivante :

$$C_{App}^P = \frac{t_{exec}^E}{t_{exec}^{EC}} \cdot \sum_{j=1}^{N_A^P} \left[u_j \cdot P_j \cdot \left(\frac{1}{t_{exec}^{EC}} \cdot \sum_{k=k^S}^{k^{EC}} C_{j,k}^P \cdot \Delta T \right) \right] \quad (3.6)$$

Dans cette formule, la somme $\sum_{j=1}^{N_A^P} u_j \cdot P_j$ permet de facturer la puissance du processeur utilisé pour chacune des tâches. Si plusieurs tâches sont présentes sur le même processeur, alors la puissance de ce dernier est facturée plusieurs fois afin de ne pas favoriser le regroupement des tâches sur un seul processeur pour économiser (ce rôle est tenu par le coût lié au nombre de processeurs utilisés).

Cette somme est pondérée par le coefficient d'ajustement des coûts $C_{j,k}^P$. Notons que la moyenne des coefficients $C_{j,k}^P$ est calculée sur toute la durée de l'application, communications entre tâches incluses, c'est-à-dire sur l'intervalle de temps $[D_{k^S}, D_{k^{EC}} + \Delta T]$ (et non $[D_{k^S}, D_{k^E} + \Delta T]$). En effet, les communications sont réparties tout le long de l'exécution de l'application, et donc tous les intervalles inclus dans l'intervalle $[D_{k^S}, D_{k^{EC}} + \Delta T]$ sont concernés par les activités de calcul du processeur.

Cependant, le résultat obtenu alors est multiplié par $\frac{t_{exec}^E}{t_{exec}^{EC}}$ afin de ne facturer que les temps de calcul, et ainsi d'exclure les temps de communication du coût lié à la puissance des processeurs.

Coût global de l'application

Le coût global de l'application, pris en compte dans le critère à minimiser, correspond à une somme pondérée de chacun des coûts calculés précédemment :

$$C_{App} = C^T \cdot t^{CPU} + C^N \cdot N_U^P + C^P \cdot C_{App}^P$$

3.2.3.5 Influence des applications soumises sur la charge de la grille

L'ensemble des tâches des applications soumises est réparti sur différents processeurs de puissances hétérogènes, influençant ainsi la charge de la grille. Cette charge étant prise en compte dans les calculs d'ordonnancements, elle doit être mise à jour lorsqu'une application est placée afin d'en tenir compte dans les ordonnancements des applications soumises par la suite. Nous montrons dans cette section comment estimer la nouvelle charge des processeurs résultant du placement d'une nouvelle application sur la grille.

Préambule

Toutes les tâches de l'application démarrent simultanément. Leur vitesse d'exécution dépend à la fois de la puissance et de la charge des processeurs qui les exécutent. Cependant, les tâches communiquent entre elles de manière synchrone, de sorte qu'elles doivent s'attendre lors de la transmission de messages si elles ne s'exécutent pas à la même vitesse.

Nous distinguons ainsi trois états dans lesquels les tâches peuvent se trouver :

- calcul (exécution sur le processeur),
- communication (transmission de message),
- attente.

De plus, nous prenons en considération uniquement les applications à gros grain, ce qui signifie que les temps de communication peuvent être négligés devant les temps de calcul.

Processeurs limitants et tâches limitantes

Soit t_j^{EC} la date de fin de l'application sur le processeur j (avec $j \in [1; N_A^P]$), c'est-à-dire la date à laquelle se termine l'ensemble des tâches de l'application s'exécutant sur ce processeur. On a :

$$\forall j \in \{1, \dots, N_A^P\} \quad t_j^{EC} = D_{k_{u_j,j}} + \Delta T = k_{u_j,j}^e \Delta T$$

La date de fin globale t^{EC} de l'application, calculée précédemment, correspond à la date à laquelle les dernières tâches de l'application se terminent. Elle s'exprime ainsi en fonction de t_j^{EC} de la manière suivante :

$$t^{EC} = \max_{j \in [1; N_A^P]} \{t_j^{EC}\} \quad \Rightarrow \quad \forall j \in \{1, \dots, N_A^P\} \quad t_j^{EC} \leq t^{EC}$$

Ainsi, les tâches s'exécutant en réalité en parallèle sur tous les processeurs, ce sont les processeurs mettant le plus de temps à exécuter les tâches qui leur sont allouées qui synchronisent la totalité de l'application par le biais des messages transmis. Les autres processeurs seront eux en attente de ces messages tant que les processeurs les moins rapides ne les auront pas transmis. Les processeurs mettant le plus de temps à exécuter les tâches qui leur sont attribuées (ceux pour lesquels $t_j^{EC} = t^{EC}$) seront qualifiés de "processeurs limitants", et les tâches qu'ils exécutent seront appelées de la même façon "tâches limitantes".

Charge induite sur les processeurs limitants

Soit $L_{j,k}$ la charge initialement prévue sur les processeurs de la grille, avec $j \in [1; N_A^P]$ et $k \in [1; N^K]$. Soit $L_{j,k}^{NEW}$ la charge mise à jour en tenant compte du placement des tâches de l'application traitée, avec également $j \in [1; N_A^P]$ et $k \in [1; N^K]$. Nous allons déterminer cette nouvelle charge dans le cas des processeurs limitants.

L'exécution des tâches limitantes de l'application nécessitera la durée totale attribuée à l'application, à savoir la période comprise entre la date de début t^S de l'application et la date de fin t^{EC} déterminée dans la section précédente. Ainsi, ces tâches ne seront jamais en attente, et de fait, ne pourront se trouver que dans deux états possibles : calcul ou communication.

Ne considérant que les applications à gros grain, les processeurs limitants exécuteront en permanence les tâches de l'application qui leur sont attribuées, et ce de t^S à t^{EC} . Ainsi, le processeur ne sera jamais libéré par l'application pendant toute la durée de son exécution. On peut en déduire que la nouvelle charge du processeur $L_{j,k}^{NEW}$ est maximale, donc égale à 1, durant toute l'exécution de l'application.

On en déduit ainsi les relations suivantes :

$$\forall j \in [1; N_A^P] / t_j^{EC} = t^{EC}, \forall k \in [k^S; k^{EC}] \quad L_{j,k}^{NEW} = 1$$

$$\forall j \in [1; N_A^P] / t_j^{EC} = t^{EC}, \forall k \notin [k^S; k^{EC}] \quad L_{j,k}^{NEW} = L_{j,k}$$

Charge induite sur les processeurs non-limitants

Soit $L_{j,k}$ la charge initialement prévue sur les processeurs de la grille, avec $j \in [1; N_A^P]$ et $k \in [1; N^K]$. Soit $L_{j,k}^{NEW}$ la charge mise à jour en tenant compte du placement des tâches de l'application traitée, avec également $j \in [1; N_A^P]$ et $k \in [1; N^K]$. Nous allons déterminer cette nouvelle charge dans le cas des processeurs non-limitants, pour lesquels $t_j^{EC} < t^{EC}$.

Les tâches s'exécutant plus rapidement que sur les processeurs limitants, elles pourront se trouver dans les trois états existants : calcul, communication ou attente des tâches plus lentes. Lorsqu'elles s'exécutent, ces tâches chargent les processeurs sur lesquels elles se trouvent à 100%. En revanche, lorsqu'elles sont dans un autre état (communication ou attente), elles libèrent les processeurs, permettant aux tâches des autres applications de s'exécuter à leur tour.

Ainsi, au cours de la durée d'exécution de l'application (de t^S à t^{EC}), les tâches non-limitantes alterneront des phases de calcul avec des phases d'attente ou de communication. Ne connaissant ni l'enchaînement ni la durée de ces phases, nous ne pouvons pas déterminer de manière précise la nouvelle charge des processeurs à chaque instant de l'exécution de l'application. Cependant, il est possible de répartir la quantité de calculs devant être effectuée par les tâches sur toute la durée d'exécution de l'application, et ainsi répartir de manière homogène la charge supplémentaire ajoutée par l'application nouvellement placée.

Soit P_j^* la puissance nécessaire pour effectuer les calculs d'une tâche sur le processeur j (avec $j \in [1; N_A^P]$) en répartissant ces derniers de manière uniforme entre t^S et t^{EC} . On a :

$$\forall j \in [1; N_A^P] / t_j^{EC} \neq t^{EC} \quad P_j^* = \frac{Q}{t^{EC}} = \frac{1}{A} \cdot \frac{T^R \cdot P^R}{t^{EC}}$$

La charge additionnelle induite par l'application sur le processeur j sera ainsi de $\frac{P_j^*}{P_j}$ durant toute la durée d'exécution de l'application. Cette valeur est une moyenne calculée sur l'ensemble de la durée d'exécution de l'application. Il est toutefois possible qu'un processeur puisse être ponctuellement trop chargé pour assumer la totalité de cette charge additionnelle. Dans ce cas, les calculs devront être reportés jusqu'à ce que la charge initiale du processeur diminue suffisamment pour équilibrer l'addition de la charge induite.

Ainsi, pour chaque intervalle de temps inclus dans la durée d'exécution de l'application (c'est-à-dire $\forall [D_k; D_k + \Delta T[\subset [t^S; t^{EC}] \Leftrightarrow \forall k \in [k^S, k^{EC}]$), la nouvelle charge qui prend en compte les tâches de l'application est égale à la somme de la charge initiale et de la charge additionnelle induite, si cette somme reste inférieure à 1 ; dans le cas contraire, la nouvelle charge sera de 1, et le surplus de charge induite sera reporté sur l'intervalle de temps suivant.

Soit L_j^{ADD} la charge additionnelle induite en moyenne sur toute la durée d'exécution de l'application sur le processeur j , avec $j \in [1; N_A^P]$. Elle ne dépend pas de l'intervalle de temps k , puisqu'il s'agit d'une moyenne. On a :

$$L_j^{ADD} = \frac{P_j^*}{P_j}$$

Soit $L_{j,k}^{REP}$ la charge reportée de l'intervalle $[D_k; D_k + \Delta T[$ vers l'intervalle $[D_{k+1}; D_{k+1} + \Delta T[$, avec $j \in [1; N_A^P]$ et $k \in [k^S; k^{EC} - 1]$. Elle est nulle sauf si le cumul de toutes les charges du processeur j durant l'intervalle $[D_k; D_k + \Delta T[$ dépasse 1.

La charge reportée peut être obtenue à l'aide d'une formule de récurrence :

$$\forall j \in [1; N_A^P] \quad \begin{cases} L_{j,k^S}^{REP} = \max \left\{ 0, L_{j,k^S} + L_j^{ADD} - 1 \right\} \\ \forall k \in [k^S + 1; k^{EC} - 1] \quad L_{j,k}^{REP} = \max \left\{ 0, L_{j,k} + L_j^{ADD} + L_{j,k-1}^{REP} - 1 \right\} \end{cases}$$

Il est ainsi désormais possible de déterminer la nouvelle charge qui prend en compte les tâches de la nouvelle application placée :

$$\forall j \in [1; N_A^P] / t_j^{EC} \neq t^{EC} \quad k = k^S \quad \Rightarrow \quad L_{j,k}^{NEW} = \min \left\{ L_{j,k} + L_j^{ADD}, 1 \right\}$$

$$\forall j \in [1; N_A^P] / t_j^{EC} \neq t^{EC}, \forall k \in [k^S + 1; k^{EC}] \quad L_{j,k}^{NEW} = \min \left\{ L_{j,k} + L_j^{ADD} + L_{j,k-1}^{REP}, 1 \right\}$$

$$\forall j \in [1; N_A^P] / t_j^{EC} = t^{EC}, \forall k \notin [k^S; k^{EC}] \quad L_{j,k}^{NEW} = L_{j,k}$$

3.3 Implémentation du modèle économique proposé par un algorithme génétique

Le modèle mathématique étant défini, cette section en propose une implémentation. Dans un premier temps, nous montrons comment résoudre ce problème d'optimisation à l'aide d'un algorithme génétique. Ensuite, une étude des résultats obtenus ainsi que de ses performances est réalisée, étape préalable nécessaire à l'intégration de cet algorithme au sein d'un environnement opérationnel.

3.3.1 Choix d'implémentation

Le modèle économique présenté dans cette thèse repose sur un problème d'optimisation non-linéaire, en variables entières, et sous contraintes. Les outils standards d'optimisation ne se prêtent que très mal à de telles caractéristiques.

Par exemple, sous *Matlab*, l'optimisation en variables discrètes est assez peu efficace. Il est possible, par exemple, de contraindre l'optimisation en imposant une contrainte non-linéaire exprimant la nature discrète des variables. Il est également possible, comme cela est proposé dans le manuel d'optimisation de *Matlab*, de progresser par optimisations successives, en fixant, après chaque itération, une inconnue supplémentaire à sa valeur arrondie. Cependant, cette manière d'opérer n'est pas acceptable, l'ordre de grandeur des inconnues (le nombre de tâches par processeur) étant la plupart du temps trop faible pour que cette méthode soit suffisamment précise.

Un autre exemple de logiciel d'optimisation est le logiciel *Xpress*. Celui-ci ne permet pas de résoudre le problème, car bien qu'il traite les résolutions en variables entières, il accepte uniquement les non-linéarités de 1er ordre.

Ainsi, les méthodes standards d'optimisation ne se prêtant pas à la nature du problème, le modèle a été implémenté à l'aide d'un algorithme génétique. En effet, les algorithmes de cette catégorie permettent d'obtenir une solution approchée satisfaisante à un problème d'optimisation, qu'il soit linéaire ou non, qu'il soit formulé en variables continues ou discrètes, et ce en un temps correct.

3.3.2 Les algorithmes génétiques

Les algorithmes génétiques appartiennent à la catégorie des algorithmes évolutionnistes, ces derniers s'inspirant directement du processus de l'évolution naturelle pour résoudre différents problèmes liés à l'optimisation, à la planification, à la modélisation, à l'identification de systèmes, à la classification, ou bien encore à la simulation de systèmes [70].

Algorithmes évolutionnistes

Les algorithmes évolutionnistes sont fondés sur la théorie de l'évolution énoncée par Charles Darwin [71], et selon laquelle seuls les individus les plus adaptés à leur environnement parviennent, au terme d'une lutte pour la vie, à survivre et à se reproduire. Appliquée au domaine informatique, cette théorie propose de faire évoluer un ensemble de solutions à un problème donné afin de trouver les meilleurs résultats. Comme cela est le cas dans le cadre de l'évolution naturelle, ce procédé résulte d'un processus stochastique, utilisant à chaque itération des phénomènes aléatoires pour faire évoluer les solutions.

Dans un tel contexte, les solutions possibles à un problème donné sont appelées les "individus". Un ensemble d'individus est alors qualifié de "population". L'objectif des algorithmes évolutionnistes, étant donné une population initiale constituée de divers individus choisis aléatoirement, est de les faire évoluer, génération après génération, de façon à tendre vers la solution optimale cherchée.

Un algorithme évolutionniste est défini par les éléments suivants [72] :

- la représentation (le codage) des individus,
- la fonction d'évaluation des individus, qui permet d'estimer l'adaptation de la solution au problème donné,
- les opérateurs de sélection, permettant de sélectionner des individus d'une population à partir de leur évaluation,
- les opérateurs de variation, permettant de faire évoluer les individus par le biais de croisements ou de mutations.

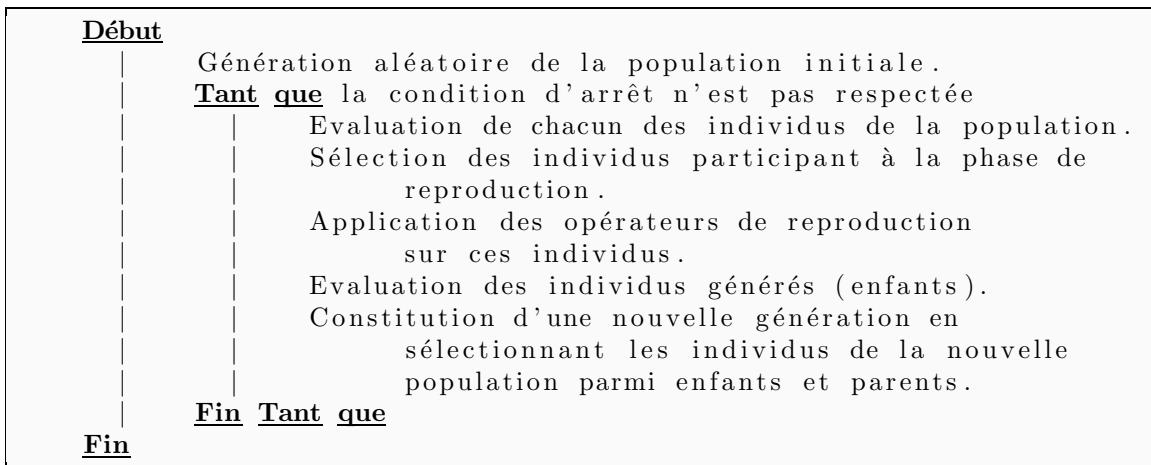
Notons enfin que ces algorithmes sont divisés en trois familles principales : les stratégies d'évolution [73], la programmation évolutionnaire [74] et les algorithmes génétiques [75, 76].

Algorithmes génétiques

L'origine des algorithmes génétiques remonte à 1975, lorsque John Holland propose les premiers d'entre eux pour résoudre des problèmes d'optimisation combinatoire [77]. Il introduit ainsi pour la première fois l'opérateur de croisement, en complément des mutations utilisées jusqu'alors.

Même s'ils sont souvent utilisés pour résoudre des problèmes d'optimisation, les applications des algorithmes génétiques ne sont pas limitées à cela [78, 79]. Ils peuvent ainsi être employés pour la reconnaissance de motifs, ou bien dans des domaines aussi diversifiés que la robotique, la biologie et la médecine [80].

Un algorithme génétique se déroule, de manière générale, selon les étapes suivantes :



Deux mécanismes de reproduction entre en jeu lors de l'élaboration des individus enfants :

- **la mutation** : elle permet de modifier aléatoirement, et de manière localisée, les caractéristiques d'un individu,
- **le croisement** : appelé également *cross-over*, il permet à deux individus de s'échanger une partie de leurs caractéristiques.

3.3.3 Implémentation

Les deux aspects importants à considérer lors de l'implémentation d'un algorithme génétique sont la représentation des individus d'une part, et d'autre part la façon dont les opérations de reproduction sont réalisées. En pratique, le choix d'une structure de données pour les solutions du problème imposera la manière dont ces opérations seront effectuées.

3.3.3.1 Représentation des individus

Rappelons la nature du problème à résoudre. Il s'agit d'un problème d'optimisation, pour lequel la fonction objectif à minimiser est (équation (3.1)) :

$$Target = W^C \cdot C_{App} + W^T \cdot T_{App}$$

Cette équation dépend de deux inconnues que nous cherchons à déterminer à l'aide de l'algorithme génétique :

- u_j = Nombre de tâches de l'application à lancer sur le processeur j , avec $j \in [1; N_A^P]$. Il s'agit ainsi de l'ordonnancement de l'application sur les différents processeurs.
- k^S = Indice de l'intervalle de temps au cours duquel l'application sera exécutée ; l'intervalle ainsi désigné est $[D_{ks}; D_{ks} + \Delta T]$. Cela correspond à la date de début de l'application.

Il apparaît ainsi qu'une manière évidente de représenter les individus consiste à considérer ces deux informations, telles que nous les avons définies dans le modèle mathématique. En conséquence, une solution peut se résumer à un vecteur représentant l'utilisation des différents processeurs associé à une date de début.

Par exemple, soient cinq tâches à placer sur dix processeurs. Deux tâches sont attribuées au processeur 2, tandis que les trois autres sont attribuées respectivement aux processeurs 3, 5 et 7. L'individu S associé (sans tenir compte de la date de début) est alors :

$$S = [0, 2, 1, 0, 1, 0, 1, 0, 0, 0]$$

L'indice du vecteur correspond aux processeurs, tandis que le contenu correspond au nombre de tâches associées à chacun d'entre eux.

Ce choix n'est pas satisfaisant, car aucune solution naturelle pour effectuer un *cross-over* n'apparaît. Il est en effet indispensable de conserver la contrainte du nombre de tâches à placer (dans l'exemple, toujours cinq tâches), une propriété que l'on peut perdre lors des brassages successifs.

Considérons, pour illustrer ceci, deux individus S_1 et S_2 , représentant deux placements différents de cinq tâches sur dix processeurs :

$$\begin{cases} S_1 = [0, 2, 1, 0, 1, 0, 1, 0, 0, 0] \\ S_2 = [0, 1, 3, 0, 1, 0, 0, 0, 0, 0] \end{cases}$$

Un croisement est réalisé entre les processeurs 3 et 4. On obtient ainsi deux vecteurs S_1^* et S_2^* :

$$\begin{cases} S_1^* = [0, 2, 1, 0, 1, 0, 0, 0, 0, 0] \\ S_2^* = [0, 1, 3, 0, 1, 0, 1, 0, 0, 0] \end{cases}$$

Le placement donné par le vecteur S_1^* n'associe que quatre tâches aux dix processeurs, tandis que le vecteur S_2^* en attribue six. Cette représentation des individus n'est donc pas adaptée à notre problème.

Une seconde représentation des individus consiste à se placer du point de vue des tâches, celles-ci cherchant à s'attribuer un processeur. L'exemple précédent de cinq tâches placées sur les processeurs 2,2,3,5 et 7 devient alors :

$$S = [2, 2, 3, 5, 7]$$

L'indice du vecteur correspond maintenant aux tâches, tandis que le contenu correspond aux numéros des processeurs qui leur sont attribués.

Il est maintenant possible de réaliser simplement un cross-over entre deux individus, tout en conservant les propriétés des placements d'origine. Par exemple, soient deux individus S_1 et S_2 , représentant deux placements différents de cinq tâches sur dix processeurs :

$$\begin{cases} S_1 = [2, 2, 3, 5, 7] \\ S_2 = [1, 2, 3, 6, 7] \end{cases}$$

Un croisement est réalisé entre les tâches 2 et 3. On obtient ainsi deux vecteurs S_1^* et S_2^* :

$$\begin{cases} S_1^* = [1, 2, 3, 5, 7] \\ S_2^* = [2, 2, 3, 6, 7] \end{cases}$$

Nous pouvons remarquer l'importance de l'ordre des tâches dans cette structure de données. En effet, supposons que les tâches ne soient pas ordonnées : les bonnes propriétés des placements seraient perdues. Par exemple, les placements précédents (non-ordonnés) $[2, 2, 3, 5, 7]$ et $[1, 7, 3, 6, 2]$ donneraient, avec le même point de croisement, les solutions filles $[2, 2, 3, 6, 2]$ et $[1, 7, 3, 5, 7]$.

3.3.3.2 Opérateurs de reproduction

Opérateurs de reproduction appliqués au choix d'un ordonnancement

L'opérateur de croisement qui s'applique à l'ordonnancement des tâches sur les processeurs est décrit ci-dessus. Il permet à deux solutions de s'échanger le placement de quelques unes de leurs tâches. Le point de croisement est choisi de manière totalement aléatoire.

L'opérateur de mutation, quant à lui, sélectionne une tâche au hasard, et la déplace sur un processeur quelconque. De plus, le déplacement de tâches des processeurs maximisant le temps d'exécution est favorisé grâce à une probabilité de tirage aléatoire plus forte.

Opérateurs de reproduction appliqués au choix d'une date de début

Dans le cas de l'optimisation de la date de début des applications, l'opérateur de *cross-over* calcule la date de début des activités processeurs, pour les solutions filles, en réalisant une moyenne aléatoirement pondérée des dates de début des solutions parents. Afin de favoriser la recherche de l'optimum, en cas d'égalité de ces dates chez les parents, on applique une "légère perturbation" à la date de début, afin de favoriser la convergence locale.

Enfin, les mutations sont réalisées en appliquant de petits déplacements autour de la valeur de la date de début des activités processeurs.

Le tableau 3.1 présente une synthèse des opérateurs de reproduction que nous utilisons.

	<i>Influence sur les tâches</i>	<i>Influence sur les dates de début</i>
Opérateurs de croisement	Brassage des placements des parents	Moyenne pondérée de la date de début
Opérateurs de mutation	Déplacement d'une tâche au hasard, en particulier celles des processeurs les plus lents	Recherche locale de meilleures solutions par ajout de perturbations

TAB. 3.1 – Opérateurs de reproduction utilisés

3.3.4 Validation de l'algorithme mis en place

3.3.4.1 Etude comparative des résultats obtenus en mode statique

Afin de vérifier de façon précise les solutions données par l'algorithme génétique, nous allons les comparer avec celles que fournit le logiciel *Xpress*. Cependant, celui-ci ne permettant pas de résoudre notre problème tel que formulé précédemment, nous devons simplifier le modèle pour réaliser cette étude.

Pour cela, nous ne prendrons pas en compte la dynamique du système. Par conséquent, nous considérerons que le coût des processeurs ainsi que leur charge sont tous deux constants au cours du temps. Cette simplification permet de se ramener à un problème d'optimisation linéaire à variables entières que nous pouvons modéliser sous *Xpress*. Ce problème, que nous qualifions ici de statique, est décrit en détail dans [50].

Nous fixons les valeurs des données du problème pour réaliser cette expérience :

- $\Delta T = 5s$
- $N^K = 720$ La période couverte comprend donc 720 intervalles de 5 secondes chacun, soit une durée de 1 heure.
- $N_A^P = 30$
- $\forall j \in [1; N_A^P] \quad P_j = 100 \times j$ La distribution des puissances est linéaire, et augmente proportionnellement avec le rang j des processeurs.

- $\forall j \in [1; N_A^P], \forall k \in [1; N^K] \quad L_{j,k} = L_j$ La charge de chaque processeur est considérée constante au cours du temps. Nous étudierons les résultats obtenus pour une charge nulle ($L_j = 0 \quad \forall j \in [1; N_A^P]$), ainsi que pour une charge non-nulle aléatoire.
- $A = 20$
- $P^R = 100$
- $T^R = 200000s$ soit environ 56h au total sur un processeur de puissance P^R , ce qui correspond à des tâches d'environ 2h45 chacune.
- $T^C = 0s$
- $N_{min}^P = 4 \quad N_{max}^P = 20$
- $C^T = 1 \quad C^N = 1 \quad C^P = 50$
- $\forall j \in [1; N_A^P] \quad \forall k \in [1; N^K] \quad C_{j,k}^P = C_j^P$ Le coefficient local de coût de chaque processeur est considéré constant au cours du temps. Nous étudierons les résultats obtenus pour des coefficients neutres ($C_j^P = 1 \quad \forall j \in [1; N_A^P]$), ainsi que pour des coefficients permettant de bénéficier de la puissance de certains processeurs à très bas prix ($\exists j \in [1; N_A^P] \quad C_j^P = \frac{1}{P_j}$).
- W^C et W^T dépendront de l'expérience réalisée (minimisation du temps d'exécution seul, du coût seul, ou d'un compromis des deux).

Minimisation du temps d'exécution sur des processeurs non-chargés

Cette expérience vise à minimiser uniquement la date de fin de l'application ($W^C = 0$ et $W^T = 1$), sur des processeurs de charge nulle ($L_j = 0 \quad \forall j \in [1; N_A^P]$).

La figure 3.7 montre l'ordonnancement choisi par l'algorithme génétique dans 90 % des cas, pour lequel on a :

$$\begin{cases} C_{App} = 2314194 \\ T_{App} = 720s \end{cases} \implies Target = 720$$

Les ordonnancements trouvés dans les 10 % des cas restants ne présentent que de faibles variations par rapport à celui-ci (inférieures à 8 % de la valeur du critère).

Ainsi, l'algorithme génétique attribue à l'application les processeurs les plus rapides, afin qu'elle se termine au plus tôt. De plus, un grand nombre de processeurs est choisi, et ce afin de ne pas ralentir les tâches en surchargeant les machines (seuls les processeurs les plus puissants peuvent se permettre d'exécuter deux tâches sans que cela ne ralente l'application).

La même solution est trouvée par *Xpress*. La seule différence que nous pouvons noter est que le temps d'exécution ne subit pas d'approximation sous *Xpress* : $T_{app} = 714.286s$, contrairement à notre algorithme, où ce temps est échantillonné en pas de 5s correspondant à ΔT . De plus, la résolution effectuée par ce logiciel étant entièrement déterministe, un seul ordonnancement est trouvé.

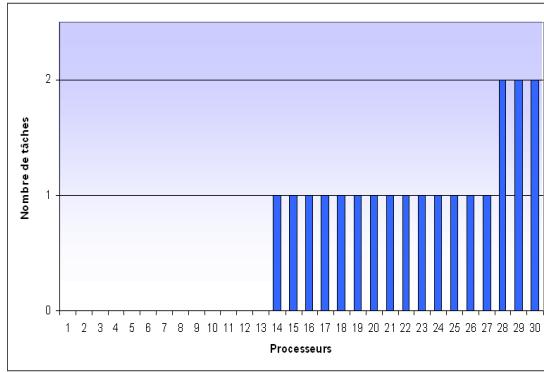


FIG. 3.7 – Ordonnancement choisi en mode statique pour minimiser le temps d'exécution de l'application sur des processeurs non-chargés

Minimisation du temps d'exécution sur des processeurs possédant une charge non-nulle

La charge des processeurs est maintenant introduite ($0 \leq L_j \leq 1 \quad \forall j \in [1; N_A^P]$). Elle est représentée en figure 3.8. Seule la date de terminaison de l'application est minimisée ($W^C = 0$ et $WT = 1$).

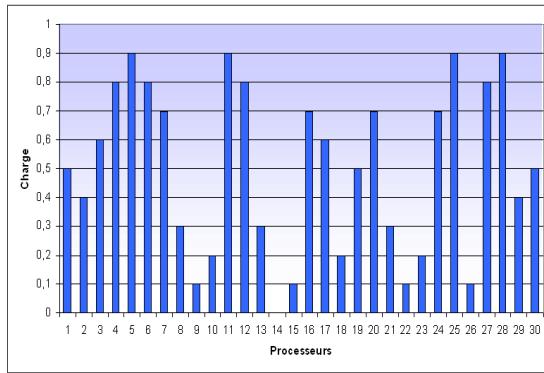
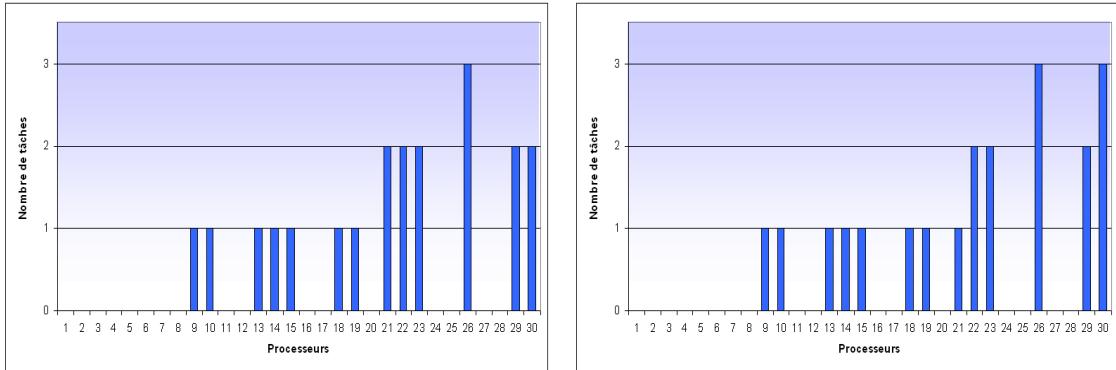


FIG. 3.8 – Charge appliquée aux processeurs

De même que précédemment, plusieurs ordonnancements très proches sont trouvés par l'algorithme génétique. L'un d'entre eux est obtenu dans plus de 90 % des cas (cf. figure 3.9(a)). Il fournit les valeurs suivantes :

$$\begin{cases} C_{App} = 2140592 \\ T_{App} = 1370s \end{cases} \implies Target = 1370$$

Les tâches ont été déplacées des processeurs les plus chargés vers d'autres moins chargés. Ainsi, on constate d'une part une augmentation du temps d'exécution de l'application (due à la fois à l'utilisation de processeurs chargés et moins puissants), ainsi qu'une diminution du coût de l'application (les processeurs moins puissants utilisés à présent étant moins chers) même si le coût n'entre pas dans le critère à minimiser.



(a) Ordonnancement choisi par l'algorithme génétique

(b) Ordonnancement choisi par Xpress

FIG. 3.9 – Ordonnancements choisis en mode statique pour minimiser le temps d'exécution de l'application sur des processeurs préalablement chargés

Il est à noter que la solution donnée par *Xpress* est légèrement différente. Elle donne un meilleur temps d'exécution que celui calculé la plupart des fois par l'algorithme génétique : $T_{App} = 1282.05s$. L'algorithme développé s'éloigne ainsi de la solution optimale à hauteur de 6,86 % de la valeur optimale du critère.

Minimisation du coût de l'application avec des coefficients locaux de coût neutres

Cette expérience vise cette fois à minimiser uniquement le coût de l'application ($W^C = 1$ et $W^T = 0$), en appliquant aux processeurs des coefficients locaux de coût neutres ($C_j = 1 \quad \forall j \in [1; N_A^P]$). Ainsi, le prix des processeurs est proportionnel à leur puissance.

Plusieurs solutions sont indifféremment obtenues par l'algorithme génétique. L'une d'entre elles est représentée par la figure 3.10(a). Elles obéissent à la loi :

$$\begin{cases} u_1 = x \\ u_2 = y \\ u_3 = 1 \\ u_4 = 1 \\ u_j = 0 \quad \forall j \in [5; N_A^P] \end{cases}$$

Toutes ces solutions offrent un coût optimal de $C_{App} = 310837$, tandis que le temps d'exécution varie (l'ordonnancement de la figure 3.10(a) produit un temps d'exécution de $T^{App} = 60000s$ sur des processeurs non-chargés).

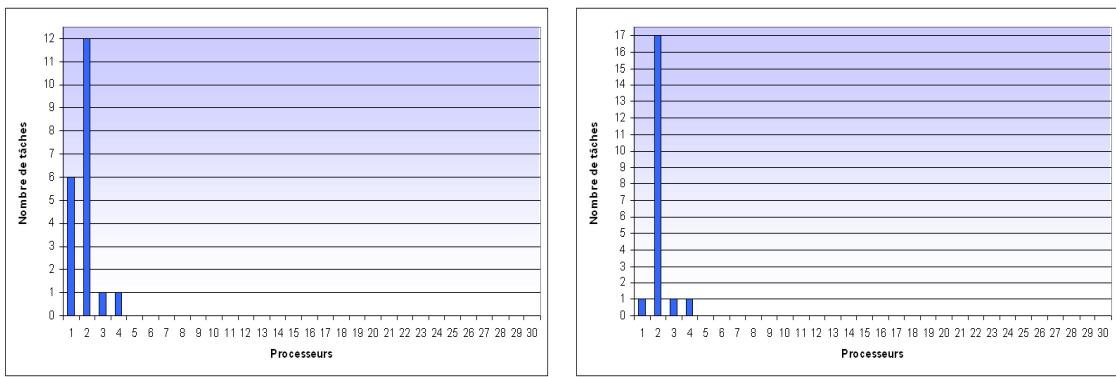
Nous pouvons remarquer que l'algorithme minimise le nombre de processeurs utilisés, puisque ce nombre est payant ($C^N \neq 0$). Cependant, il est obligé d'en utiliser au moins quatre, car cela est spécifié dans les caractéristiques de l'expérience ($N_{min}^P = 4$). On constate que l'on a bien : $N_U^P = N_{min}^P = 4$.

Notons donc que toutes les solutions trouvées ici sont semblables du point de vue du critère de l'algorithme, même si elles induisent des temps d'exécution plus ou moins élevés.

En effet, ce facteur n'ayant pas été pris en compte dans les paramètres de l'algorithme ($W^T = 0$), ces solutions lui sont toutes équivalentes. On aura donc toujours intérêt, en pratique, à mettre un coefficient non-nul sur le facteur temps, pour éviter d'obtenir, à coût égal, des temps d'exécution trop longs.

Enfin, *Xpress* trouve également une solution appartenant à la famille de solutions trouvées par l'algorithme (figure 3.10(b)). Cette solution est unique, puisque le logiciel est déterministe, et elle est de la forme :

$$\begin{cases} u_1 = 1 \\ u_2 = 17 \\ u_3 = 1 \\ u_4 = 1 \\ u_j = 0 \quad \forall j \in [5; N_A^P] \end{cases}$$



(a) Ordonnancement choisi par l'algorithme génétique

(b) Ordonnancement choisi par *Xpress*

FIG. 3.10 – Ordonnancements choisis en mode statique pour minimiser le coût de l'application avec des coefficients locaux de coût neutres

Minimisation du coût de l'application avec prise en compte de coefficients locaux

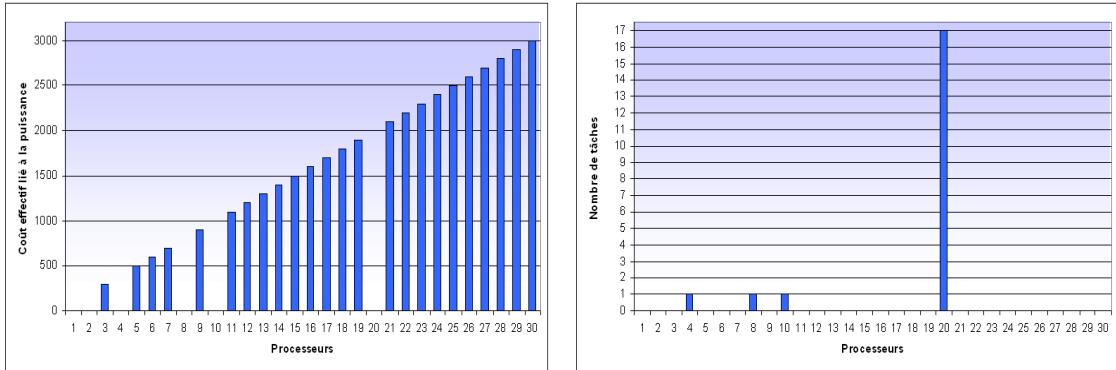
Nous introduisons maintenant des coefficients locaux de coût associés aux processeurs (cf. figure 3.11(a)), tels que :

$$\begin{cases} C_j^P = \frac{1}{P_j} & \text{si le processeur est en tarif promotion} \\ C_j^P = 1 & \text{sinon} \end{cases}$$

Tout comme l'expérience précédente, seul le coût de l'application sera minimisé ($W^C = 1$ et $W^T = 0$).

Dans le cadre de cette expérience, l'algorithme met en évidence la solution représentée en figure 3.11(b). Les valeurs suivantes sont obtenues :

$$\begin{cases} C_{App} = 23254 \\ T_{App} = 8500s \end{cases} \implies Target = 23254$$



(a) Coût effectif des processeurs (avec prise en compte des coefficients locaux)

(b) Ordonnancement optimal trouvé

FIG. 3.11 – Ordonnancement choisi en mode statique pour minimiser le coût de l’application avec prise en compte des coefficients locaux de coût

Cinq processeurs sont au tarif promotionnel, et possèdent ainsi le même coût lié à la puissance. L’algorithme choisit de placer la quasi-totalité des tâches sur le processeur le plus rapide, non pas pour minimiser la date de fin de l’application, mais pour minimiser le temps CPU consommé, ce dernier étant payant ($C^T \neq 0$). Trois tâches sont tout-de-même placées sur trois autres processeurs, afin de respecter la contrainte du nombre de processeurs minimum à utiliser ($N_U^P \geq N_{min}^P$).

Notons enfin que *Xpress* trouve exactement les mêmes résultats que l’algorithme génétique.

Minimisation d’un compromis entre le coût de l’application et son temps d’exécution sur des processeurs non-chargés et avec des coefficients locaux de coût neutres

Afin de donner aux deux facteurs, temps et coût, une influence comparable dans la recherche du meilleur compromis, les coefficients suivants sont choisis :

- $W^C = 1$
- $W^T = 500$

Ces deux valeurs sont choisies à partir du coût optimal (lorsque le temps d’exécution n’est pas pris en compte) et de la date de fin optimale (lorsque le coût de l’application est ignoré). Il est bien entendu tout-à-fait possible d’automatiser un tel calcul.

Le meilleur ordonnancement découvert par l’algorithme figure en 3.12(a). Les résultats obtenus dans ce cas sont les suivants :

$$\begin{cases} C_{App} = 1321879 \\ T_{App} = 1670s \end{cases} \implies Target = 2156879$$

Les solutions choisies mettent en évidence un placement sur des processeurs compris dans un domaine de puissances intermédiaires, permettant d’avoir une certaine rapidité d’exécution pour un coût assez faible. En conséquence, ni le coût de l’application, ni sa durée d’exécution ne sont optimaux. Ceci est conforme aux résultats attendus, puisque

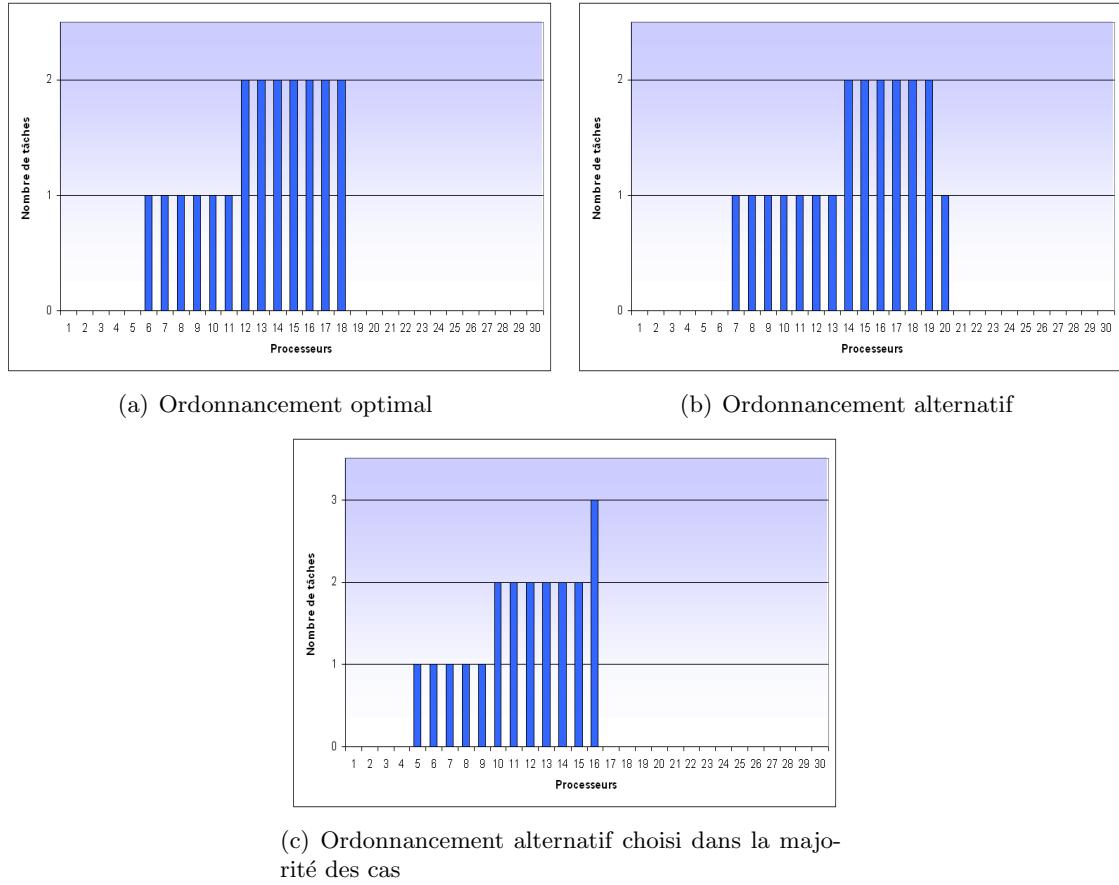


FIG. 3.12 – Ordonnancements choisis en mode statique pour minimiser un compromis entre le coût de l’application et sa date de terminaison (sans charge, coefficients locaux de coût neutres)

nous voulions trouver un bon compromis entre les deux. De plus, le besoin en rapidité d’exécution force l’algorithme à utiliser plus de processeurs, tout en essayant de ne pas faire trop augmenter le coût.

L’algorithme donne d’autres solutions proches en termes de critère, telle que celle représentée sur la figure 3.12(b), qui est un minimum local auquel correspondent les valeurs suivantes :

$$\begin{cases} C_{App} = 1455164 \\ T_{App} = 1430s \end{cases} \implies Target = 2170164$$

Dans ce cas, le coût est bien supérieur à celui de la solution optimale, mais la diminution significative du temps d’exécution compense le critère.

A l’inverse, le dernier placement choisi par l’algorithme (figure 3.12(c)) fait croître le temps d’exécution et réduit le coût, pour proposer une valeur du critère proche de l’optimal.

Ces solutions “alternatives” restent assez proches de la solution optimale en termes de critère. On obtient ainsi, à 95 %, un de ces trois placements, qui sont à environ 1 % de

la solution optimale. Notons que le placement le plus fréquemment obtenu est celui de la figure 3.12(c), choisi dans environ 50 % des cas.

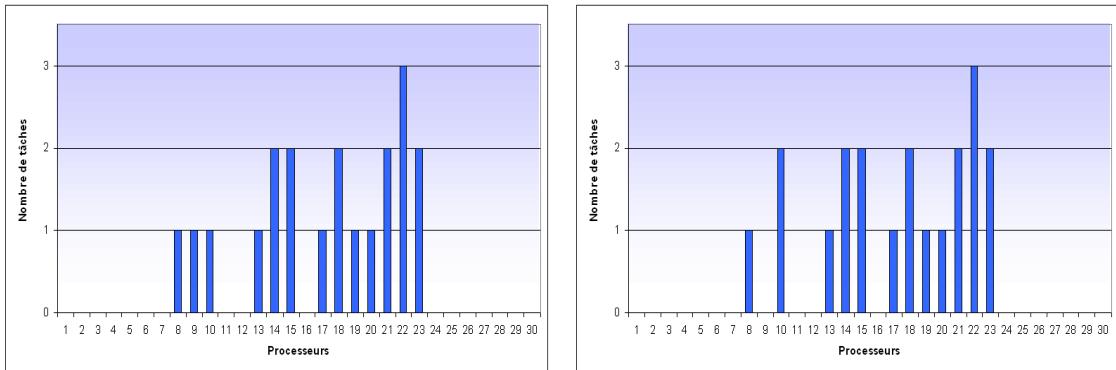
Notons enfin qu'*Xpress* découvre la solution optimale donnée par l'algorithme génétique en figure 3.12(a). Les valeurs obtenues par ce logiciel de manière exacte (sans l'échantillonnage de $\Delta T = 5s$) sont :

$$\begin{cases} C_{App} = 1321883.076 \\ T_{App} = 1666.67s \end{cases} \implies Target = 2155218.07$$

Minimisation d'un compromis entre le coût de l'application et son temps d'exécution sur des processeurs chargés et avec des coefficients locaux de coût significatifs

Les paramètres de l'expérience sont utilisés ($W^C = 1$ et $WT = 500$). De plus, les processeurs possèderont une charge représentée par la figure 3.8, ainsi que le profil de coût donné par la figure 3.11(a). Dans de telles conditions, le meilleur ordonnancement trouvé par l'algorithme génétique (figure 3.13(a)) donne les résultats suivants :

$$\begin{cases} C_{App} = 1544313 \\ T_{App} = 1790s \end{cases} \implies Target = 2439313$$



(a) Meilleur ordonnancement trouvé par l'algorithme génétique

(b) Ordonnancement donné par *Xpress*

FIG. 3.13 – Ordonnancements choisis en mode statique pour minimiser un compromis entre le coût de l'application et sa date de terminaison (avec prise en compte de la charge et des coefficients locaux de coût)

L'ordonnancement déterminé par *Xpress* (figure 3.13(b)) diffère légèrement. Il n'améliore le critère obtenu avec l'algorithme génétique que d'environ 0,1 % :

$$\begin{cases} C_{App} = 1544318 \\ T_{App} = 1785.71s \end{cases} \implies Target = 2437173$$

3.3.4.2 Etude des résultats obtenus en mode dynamique

Nous allons à présent vérifier la cohérence du comportement de l'algorithme génétique dans un contexte dynamique, où la charge des processeurs ainsi que les coefficients de coût

qui leur sont appliqués varient en fonction du temps. Il doit donc désormais optimiser le critère en fonction d'une variable supplémentaire : la date de début des tâches.

Les valeurs des données du problème sont conservées (une application de 20 tâches à placer sur 30 processeurs disponibles). Seul le cadre temporel est changé :

- $\Delta T = 30s$
- $N^K = 600$ La période couverte comprend maintenant 600 intervalles de 30 secondes chacun, soit une durée de 5 heures.

Minimisation du coût de l'application

Les expériences précédentes ont montré que, lorsque le coût de l'application doit être minimisé, les tâches occupent les processeurs les plus économiques (ceux dont la puissance est faible, ou ceux bénéficiant d'un tarif promotionnel).

Dans cette expérience, nous fixons un profil de coefficients de coût ne dépendant plus du processeur utilisé, mais variant au cours du temps (figure 3.14(a)). Nous le définissons comme suit :

$$\begin{cases} \forall j \in [1; N_A^P] \quad \forall k \in [1; 99] & C_{j,k}^p = 1 \\ \forall j \in [1; N_A^P] \quad \forall k \in [100; 199] & C_{j,k}^p = 0,1 \\ \forall j \in [1; N_A^P] \quad \forall k \in [200; 600] & C_{j,k}^p = 1 \end{cases}$$

L'ensemble des processeurs bénéficie ainsi d'un tarif promotionnel durant 100 intervalles de temps.

Lorsque seul le coût de l'application est minimisé ($W^C = 1$ et $W^T = 0$), l'algorithme génétique propose d'attendre le début de l'offre proportionnelle afin de profiter des coûts les plus faibles :

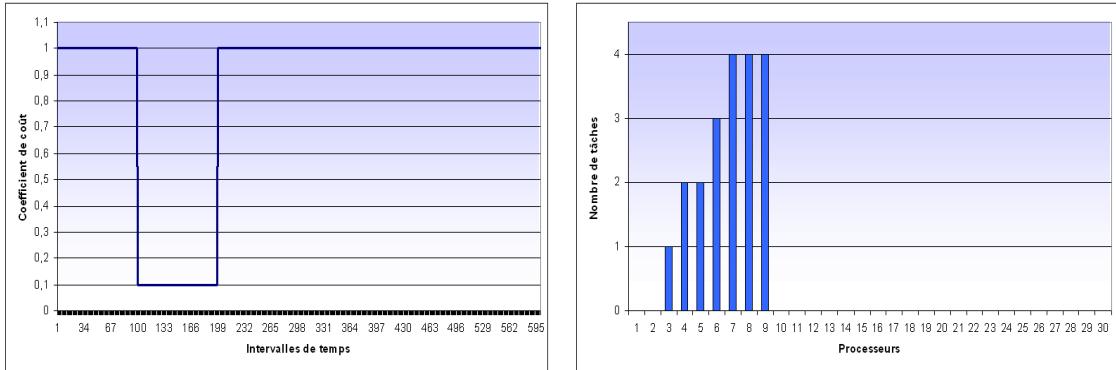
$$k^S = 100$$

La figure 3.14(b) montre le placement choisi. Les processeurs les moins puissants ne sont pas utilisés, dans le but de terminer l'exécution de l'application avant que ne finisse l'intervalle économiquement avantageux. De cette manière, la durée d'exécution de l'application est de 93 intervalles.

Minimisation d'un compromis entre le coût de l'application et son temps d'exécution

Tentons maintenant de minimiser un compromis entre la date de fin de l'application et son coût, en appliquant la pondération suivante pour le calcul du critère :

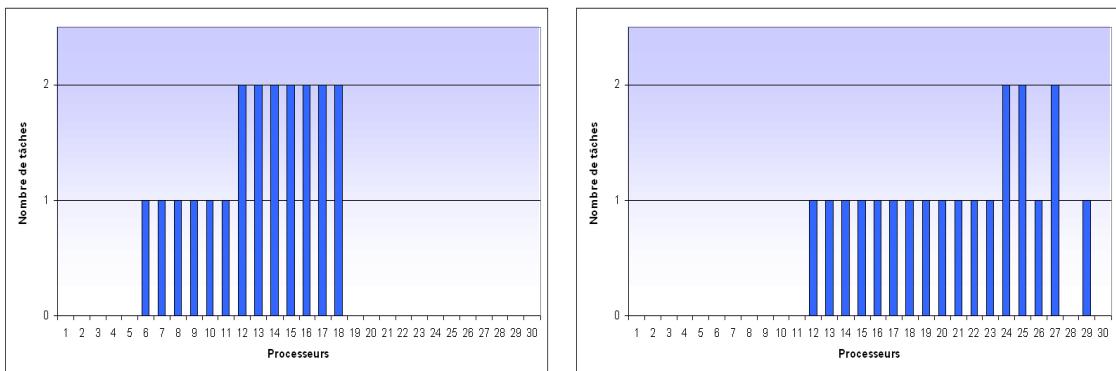
- $W^C = 1$
- $W^T = 500$



(a) Evolution des coefficients locaux de coût au cours du temps (b) Ordonnancement trouvé par l'algorithme génétique

FIG. 3.14 – Ordonnancement choisi en mode dynamique pour minimiser le coût de l’application

En utilisant le même profil de coefficients de coût que précédemment (3.14(a)), l’algorithme opte pour un début immédiat des tâches ($k^S = 0$). En effet, le coût économisé durant l’intervalle soldé ne peut compenser le temps perdu à l’attendre. Le placement optimal est alors identique à celui trouvé dans un contexte statique. Il est à nouveau présenté dans la figure 3.15(a). Notons que le critère correspondant à cette solution est $Target = 2156879$.



(a) Ordonnancement choisi si la période promotionnelle commence au 100ème intervalle de temps (b) Ordonnancement choisi si la période promotionnelle commence au 80ème intervalle de temps

FIG. 3.15 – Ordonnancements choisis en mode dynamique pour minimiser le coût de l’application et sa date de terminaison

Par contre, si les processeurs sont soldés à partir du 80ème intervalle de temps, le temps d’attente plus faible est alors compensé par l’économie due aux faibles coefficients de coût. L’algorithme trouve ainsi une solution de critère inférieur, représentée par la figure 3.15(b), avec une date de départ différée : $k^S = 80$.

Les processeurs étant moins onéreux, et le temps d’attente étant tout de même important, la répartition des tâches est globalement déplacée vers les processeurs les plus puissants.

3.3.5 Performances de l'algorithme

L'objectif final de l'algorithme génétique mis en place est d'être intégré au sein d'un ordonnanceur existant, afin d'être utilisé dans un environnement de grille. Ses performances constituent donc un point critique, le temps nécessaire à la détermination d'une solution devant être suffisamment faible pour que l'algorithme soit utilisable.

Nous allons donc étudier les performances de l'algorithme, tout d'abord en termes de convergence, puis en termes de temps de calcul nécessaire à l'obtention d'une solution.

Cette étude permettra notamment de vérifier que l'algorithme reste performant malgré un passage à l'échelle du problème. En effet, le temps de résolution croît avec la taille du problème, les structures de données plus volumineuses devenant plus longues à traiter, et la convergence se trouve ralentie par un espace de solutions à explorer plus étendu. L'algorithme doit supporter un tel passage à l'échelle, afin d'être utilisable dans un environnement de grille où la notion de passage à l'échelle joue un rôle essentiel.

3.3.5.1 Convergence de l'algorithme

Dans un premier temps, nous allons étudier la convergence de l'algorithme pour le calcul du placement d'une application de 20 tâches sur 30 processeurs, avec les mêmes caractéristiques que dans les expériences précédentes. Une charge variable aléatoire sera appliquée sur les processeurs, tandis qu'un profil de coefficients locaux de coût neutres sera appliqué, ceci afin de ne pas avoir de solutions évidentes.

La figure 3.16 montre la convergence de l'algorithme vers le critère optimal dans différents cas :

- lorsque seul le coût de l'application à placer est minimisé ($W^C = 1$ et $W^T = 0$),
- lorsque seule la date de terminaison de l'application est minimisée ($W^C = 0$ et $W^T = 1$),
- lorsque un compromis des deux est minimisé ($W^C = 1$ et $W^T = 500$),

Il est possible de constater que la convergence de l'algorithme est assez rapide lorsque seul le coût de l'application doit être minimisé (figure 3.16(a)). En effet, l'algorithme met moins de 400 itérations pour trouver la solution optimale. Le profil de coût étant constant au cours du temps, la recherche des machines les moins coûteuses ne dépendra pas de la date de début de l'application. L'espace de recherche est ainsi fortement réduit, et la solution optimale est rapidement trouvée.

La minimisation de la date de fin de l'application nécessite plus d'itérations, environ 650, pour trouver la solution optimale (figure 3.16(b)). Dans ce cas, la date de début joue un rôle important dans le critère à minimiser, car ce dernier dépend de la charge des machines variant au cours du temps. Des optimums locaux sont régulièrement trouvés. L'algorithme génétique a subi quelques modifications afin d'éviter de stagner dans ces derniers. Par exemple, à critère égal, l'algorithme privilégie les solutions pour lesquelles un nombre minimal de processeurs maximise le temps d'exécution.

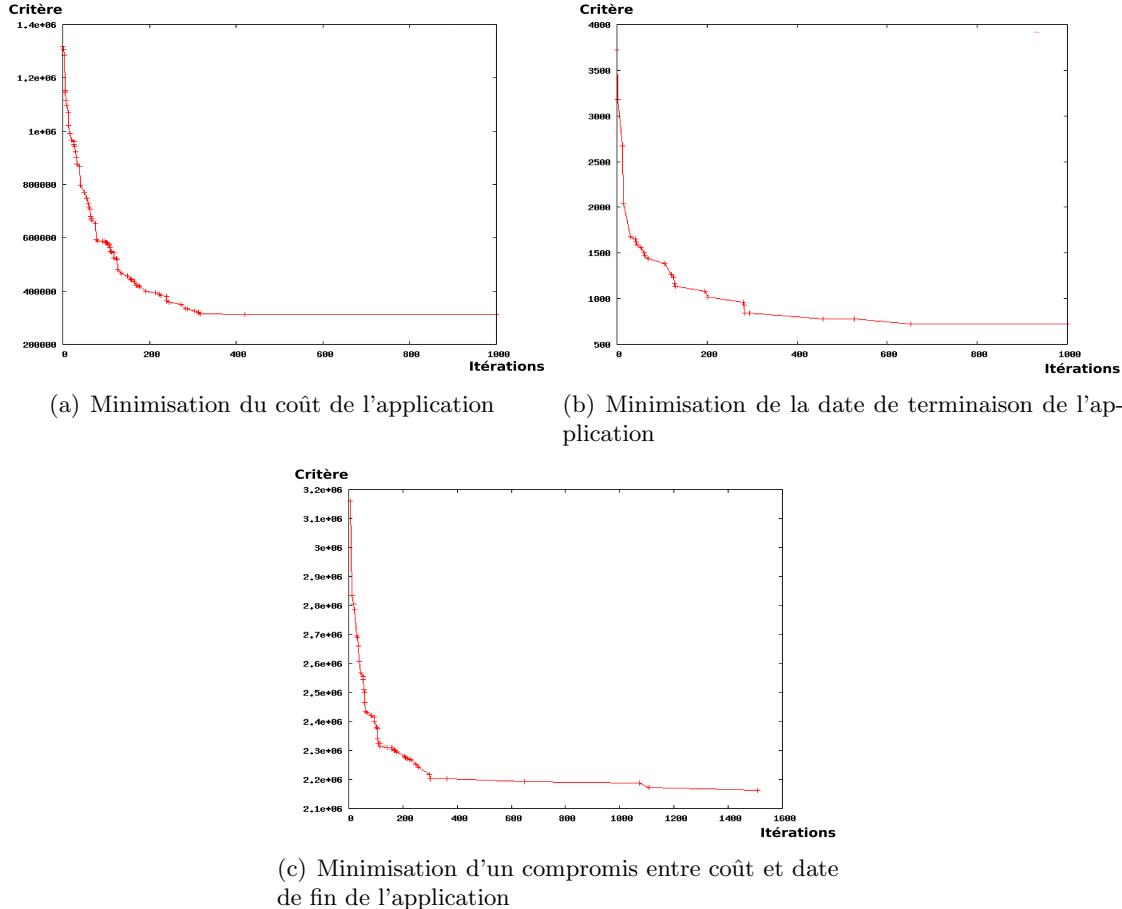


FIG. 3.16 – Convergence de l’algorithme génétique pour le calcul du placement d’une application de 20 tâches sur 30 processeurs

La minimisation d’un compromis entre le coût et la date de fin de l’application converge beaucoup moins rapidement que la minimisation d’un seul critère (figure 3.16(c)). On constate que, au terme de 1400 itérations, l’algorithme parvient toujours à améliorer la solution qu’il trouve. Cependant, la valeur du critère ne varie que très peu entre deux optimums locaux successifs rencontrés, de sorte qu’un résultat acceptable est très vite trouvé.

Passage à l’échelle

La figure 3.17 montre maintenant la convergence de l’algorithme pour une application de 40 tâches à placer sur une grille de 1000 processeurs.

Plusieurs points peuvent être notés. Tout d’abord, la convergence de l’algorithme génétique est extrêmement rapide, de l’ordre de 1000 itérations, lors de la minimisation du seul coût de l’application (figure 3.17(a)), et ce pour les mêmes raisons que celles mentionnées précédemment.

La minimisation du temps d’exécution nécessite, quant à elle, environ 3200 itérations pour trouver une solution proche de l’optimale (figure 3.17(b)).

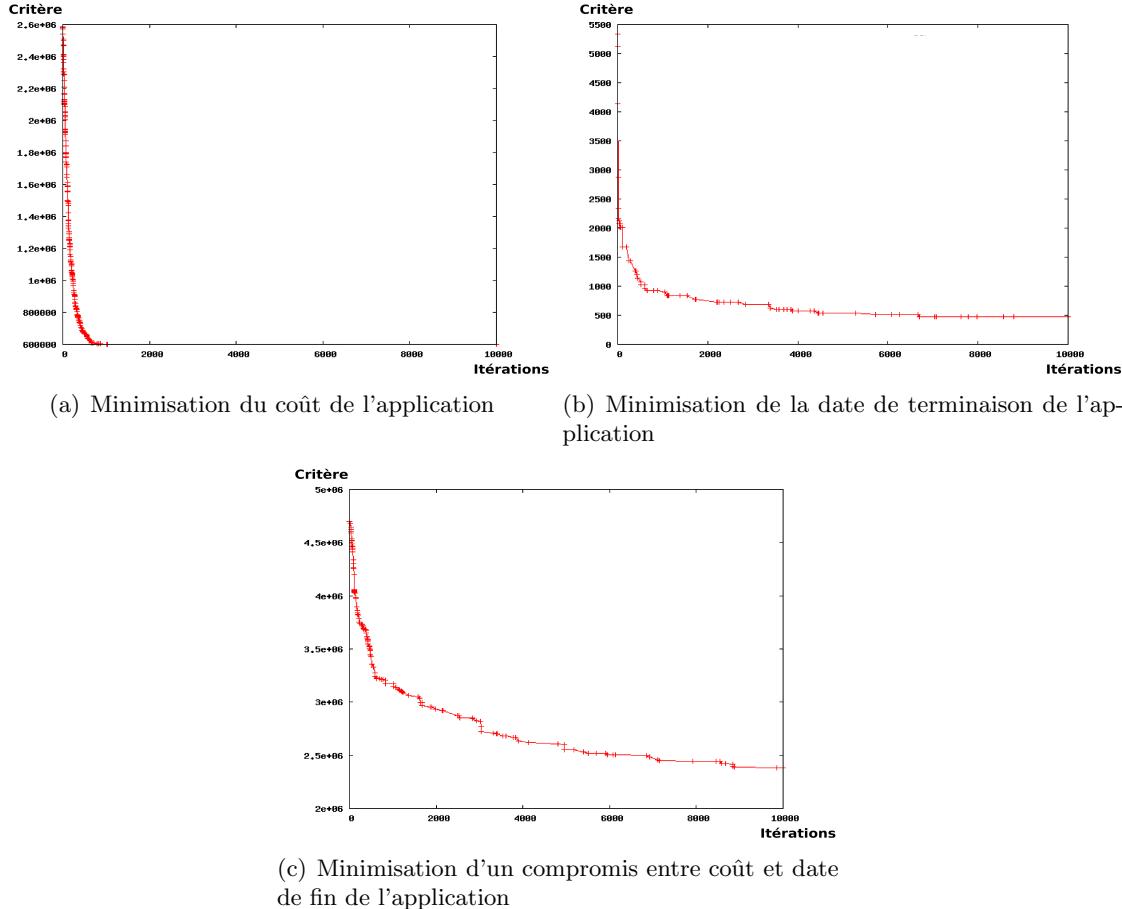


FIG. 3.17 – Convergence de l'algorithme génétique pour le calcul du placement d'une application de 40 tâches sur 1000 processeurs

Enfin, le compromis de ces deux critères nécessite un nombre important d'itérations, environ 9000, pour parvenir à se stabiliser (figure 3.17(c)). Cependant, un résultat correct est approché autour des 5000 itérations.

3.3.5.2 Temps de résolution

Afin d'obtenir un temps de résolution minimal, de nombreuses contraintes ont encadré la phase de développement de l'algorithme génétique implémentant le modèle économique.

- L'algorithme a été codé en utilisant le langage C, offrant une exécution des plus rapides.
- De nombreuses optimisations ont été apportées au code :
 - optimisation du temps passé dans chaque fonction du programme,
 - minimisation des allocations mémoires,
 - utilisation de pointeurs pour le passage de paramètres,
 - utilisation du parallélisme (*multi-threading*) : le calcul des critères des différents individus est divisé entre plusieurs *threads*, dans le but de profiter des caractéristiques des machines multi-processeurs).
- L'algorithme a été modifié à plusieurs reprises afin d'accélérer sa convergence, et donc son temps d'exécution.

Ainsi, le placement d'une application de 40 tâches sur une grille de 5000 processeurs prend 18 secondes, lorsque l'algorithme génétique est exécuté sur une machine bi-processeur Intel cadencée à 3,2 GHz.

En réalité, les 5000 processeurs pourront être divisés en différents domaines³ qui seront en compétition pour exécuter l'application. Dans ce cas, l'algorithme pourra être exécuté en parallèle sur chaque domaine, en considérant ainsi un nombre restreint de processeurs, et ce sera le domaine obtenant la meilleure solution de placement qui sera choisi.

Notons que le temps d'exécution de l'algorithme diminue bien avec le nombre de processeurs considérés, puisque le placement d'une application de 40 tâches sur 1000 processeurs est réalisé en moins de 5 secondes, toujours sur une machine Intel $2 \times 3,2$ GHz.

Ce temps de 5 secondes est très faible au regard du temps requis pour l'exécution des applications nécessitant un support tel qu'une grille de calcul. Ainsi, cet algorithme convient parfaitement à l'ordonnancement d'applications sur grilles, et il est tout-à-fait envisageable de l'utiliser dans un environnement fonctionnel. Ce point est abordé dans la section suivante.

3.4 Intégration du modèle économique au sein d'OAR

Dans cette section, nous montrons comment intégrer le modèle économique défini précédemment au sein d'un environnement opérationnel. Nous choisissons comme support pour nos travaux la grille expérimentale de recherche *Grid'5000*, et son gestionnaire de ressources *OAR* [63]. Après une étude préliminaire détaillant notamment les spécifications de la tâche à accomplir, les modifications apportées à OAR, tant au niveau de la soumission d'applications que du placement de celles-ci, seront abordées.

3.4.1 Etude préliminaire

Avant d'aborder l'intégration de l'algorithme génétique implémentant le modèle économique au sein d'OAR, une étude préliminaire est réalisée afin de détailler les spécifications de cette intégration, ainsi que les difficultés induites par l'architecture et le fonctionnement d'OAR.

3.4.1.1 Spécifications

L'intégration de l'algorithme génétique au sein d'OAR vise à mettre en œuvre le modèle économique défini précédemment dans un environnement opérationnel, afin d'en permettre une utilisation dans un contexte entièrement réel.

Le choix de l'ordonnanceur OAR a notamment été motivé par son architecture modulaire (cf. section 2.4.4), permettant d'insérer aisément un nouveau modèle d'ordonnancement, comme cela est expliqué par la suite. Ainsi, l'objectif de cette étude est de montrer comment faire appel à l'algorithme génétique développé dans le cadre de cette thèse lorsqu'une demande de placement est soumise à OAR.

³Un domaine est un ensemble de machines, généralement proches géographiquement, appartenant à la même organisation. Elles sont dans la plupart des cas reliées entre elles par un réseau offrant une meilleure bande-passante que celle qui les connecte au reste de la grille.

De plus, cette intégration doit minimiser les modifications apportées à OAR, afin d'en accroître la compatibilité avec les futures versions de l'ordonnanceur.

3.4.1.2 Incompatibilités entre le fonctionnement actuel d'OAR et le modèle économique

Une première mise en confrontation du modèle économique proposé dans cette thèse et d'OAR montre les différences majeures qui les opposent.

La première différence concerne la réservation d'une ressource. Dans OAR, une ressource ne peut, par défaut, être utilisée que par un utilisateur à la fois. Ainsi, sur la plate-forme Grid'5000, l'utilisateur réserve des machines qui lui sont alors totalement dédiées. Il peut donc déployer, s'il le désire, un système d'exploitation personnalisé, et être sûr que ses expériences ne seront pas perturbées par d'autres utilisateurs. Il est cependant possible, via une option explicite, d'autoriser de partager les ressources réservées avec d'autres utilisateurs (*timesharing*). Dans ce cas, OAR permet d'attribuer plusieurs jobs aux mêmes ressources, sans se préoccuper des répercussions sur leur temps d'exécution. Ainsi, l'utilisateur doit tenir compte, lorsqu'il soumet une application, des jobs déjà placés sur les ressources qu'il utilisera ainsi que des jobs potentiellement à venir pour fixer la durée de la réservation, sous peine de ne pas voir son application se terminer au cours de la période réservée.

Nous avons vu dans le modèle économique, que les utilisateurs soumettent des applications composées de plusieurs tâches, en spécifiant un compromis rapidité / coût. Ces tâches entrent alors en compétition sur des ressources de type processeur. Ainsi, plusieurs tâches, qui ne sont pas nécessairement issues de la même application et donc du même utilisateur, peuvent s'exécuter simultanément sur un même processeur, autorisant alors par défaut la notion de temps partagé entre différents utilisateurs sur une même ressource.

La solution retenue consiste donc à modifier OAR afin que plusieurs personnes puissent réserver simultanément, par défaut, une même machine. Pour ce faire, la commande de soumission `oarsub` doit changer sa manière de vérifier la disponibilité des ressources, et le serveur OAR ne doit pas être perturbé par l'exécution de plusieurs jobs sur une même ressource. De plus, la prédiction de la charge des machines doit être gérée par OAR afin de ne plus autoriser de réservations sur des ressources saturées.

La deuxième différence majeure opposant le modèle économique proposé et OAR concerne la représentation d'une ressource. Le modèle prend en compte uniquement des ressources de type CPU (processeur) pour l'ordonnancement, alors qu'OAR introduit une représentation sous forme hiérarchique. Ainsi, l'architecture choisie sur Grid'5000 par exemple montre que la grille est divisée en plusieurs sites géographiquement distincts. Chaque site contient une ou plusieurs grappes de machines (*clusters*), eux-mêmes pouvant regrouper plusieurs centaines de machines. Enfin, la plupart de ces machines contiennent plusieurs processeurs. Cependant, la plus petite unité réservable par OAR est donc la machine, tandis que le modèle considère uniquement les processeurs.

La solution consiste, dans ce cas, à utiliser la notion de *CPUSED*⁴. Un *CPUSED* s'apparente à une autorisation d'utilisation de un ou plusieurs processeurs et d'une quantité de mémoire système bien définie. Ainsi, une machine disposant de deux CPUs et de 2Go de mémoire pourra, par exemple, accueillir deux *CPUSED*s (composés d'un CPU et d'1Go de mémoire chacun). Ainsi, la plus petite unité que l'on pourra réservé grâce à OAR devra donc être un *CPUSED*.

3.4.2 Politiques d'ordonnancement

OAR s'appuie sur la notion de files d'attente pour la soumission d'applications (figure 3.18). Chaque utilisateur peut choisir une file d'attente différente, qui peut également être fonction de son niveau de priviléges, lors de la soumission de son application. Un aiguilleur vérifie d'abord si l'utilisateur a le droit de soumettre un job dans la file d'attente demandée (d'après des règles d'admission écrites dans la base de données) avant de l'insérer dans la file correspondante. Chaque file a une politique d'ordonnancement différente (*scheduling policy*), tandis que leurs priorités relatives sont gérées par le module "Meta-Ordonnanceur" (*Meta-Scheduler*).

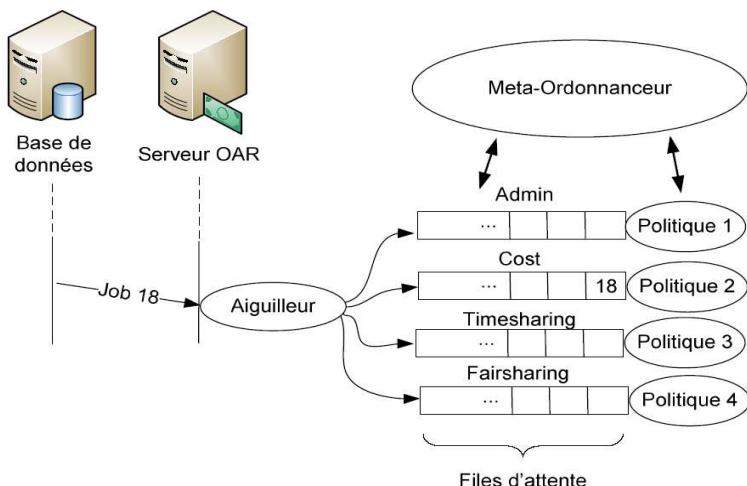


FIG. 3.18 – Système de files d'attente géré par OAR

Prenons l'exemple de deux utilisateurs, l'un ayant demandé l'insertion de son job dans la file d'attente *Timesharing* de priorité faible, et l'autre dans la file *Admin* de priorité plus élevée. L'aiguilleur vérifie d'abord les autorisations, puis insère les jobs dans les files. Le meta-ordonnanceur garantira que les jobs dans la file de priorité la plus élevée seront traités en premier, et lancera la politique d'ordonnancement de la file. La politique est locale à chaque file et ne traite donc que les jobs dans sa file. On peut donc mettre en place une file *Cost*, comme indiqué sur la figure, soumise à une politique d'ordonnancement correspondant au modèle économique.

⁴Les *CPUSED*s sont des objets du noyau Linux permettant aux utilisateurs de partitionner leurs machines multi-processeurs pour créer des zones d'exécutions distinctes, chacune possédant son ou ses propres processeurs et sa propre zone de mémoire. Une application lancée sur un *CPUSED* sera confinée à l'intérieur de celui-ci, et ne pourra en aucun cas sortir de cette zone pour s'exécuter sur d'autres processeurs de la même machine que ceux qui lui ont été attribués.

La définition des files d'attente est effectuée dans la base de données, grâce aux tables *Queues* (files) et *Admission_rules* (règles d'admission) représentées en figure 3.19.

Queues	Admission_rules
Queue_name Priority Scheduler_policy State	Id Rule

FIG. 3.19 – Structures des tables de la base de données d'OAR pour la gestion des files d'attente

Nous modifions ainsi la base de données pour ajouter à OAR la file d'attente correspondant au modèle économique. Notons que ces modifications limitées seront faciles à mettre à nouveau en œuvre dans le cas d'un changement de version d'OAR, puisqu'il s'agit de rajouter l'appel à notre ordonnanceur dans la table *Queues* (tableau 3.2), et de modifier la table *Admission_rules* (tableau 3.3) pour faire appel à notre ordonnanceur par défaut (si l'utilisateur ne spécifie pas de file d'attente lors de la soumission de son application).

Queues			
Queue_name	Priority	Scheduler_policy	State
Cost	5	Cost_scheduler	Active

TAB. 3.2 – Modifications de la table *Queues*

Admission_rules	
Id	Rule
1	<code>if (not defined(\$queue_name)) { \$queue_name = "Cost" ; }</code>

TAB. 3.3 – Modifications de la table *Admission_rules*

Nous associons ainsi la file *Cost* avec la politique d'ordonnancement basée sur le modèle économique défini. Celle-ci est implémentée par le programme **Cost_scheduler** qui sera lancé par une interface d'appel écrite en langage *Perl* à la demande du serveur OAR (depuis la librairie *Oar_Tools*).

3.4.3 Soumission d'applications

3.4.3.1 Mise en place d'un nouveau client

Afin d'ajouter les fonctionnalités du modèle économique à l'outil de soumission d'OAR, nous avons développé un adaptateur (*wrapper*) du client **oarsub**. L'avantage d'une telle solution est de ne pas modifier le client existant mais de l'utiliser, en y ajoutant une couche implémentant les fonctionnalités désirées. Cela garantit une séparation entre le développement d'OAR et celui de notre client : le code d'**oarsub** n'est pas modifié et les fonctionnalités de notre client s'intègreront donc sans effort dans les nouvelles versions d'OAR.

Ce nouveau client propose les fonctionnalités nécessaires à l'utilisation du modèle économique pour le placement des applications qui lui sont soumises. Ainsi, il permet à l'utilisateur de spécifier les données nécessaires au modèle, telles que :

- le nombre de tâches de l'application (A),
- la durée d'exécution de l'application sur un processeur de référence (T^R),
- la puissance de ce processeur de référence (P^R),
- le temps moyen de communication des tâches (T^C),
- les bornes sur le nombre de processeurs à utiliser (N_{min}^P et N_{max}^P).

L'adaptateur créé permet également de spécifier certaines options usuelles d'OAR, telles que l'exécutable à lancer, en redirigeant simplement ces options vers la commande `oarsub`. Cependant, un filtrage de ces options est effectué, certaines d'entre elles étant incompatibles avec l'utilisation du modèle économique. Ainsi, l'utilisateur ne peut plus spécifier de date de début fixe, car celle-ci est choisie par l'algorithme génétique. De plus, l'utilisateur n'a plus le choix de la file d'attente vers laquelle diriger son application. Seule la file *Cost* peut être utilisée. Enfin, il n'est plus possible de réserver des machines pour une utilisation interactive, puisque seules des applications parallèles en placement *batch* sont considérées dans notre modèle.

Plus d'informations quant à l'utilisation de l'adaptateur du client `oarsub` sont disponibles dans l'annexe A.

3.4.3.2 Traitement des paramètres du modèle

L'exécution de l'algorithme génétique requiert trois types de paramètres :

- ceux fournis par l'utilisateur, correspondant aux caractéristiques de l'application soumise,
- ceux qui sont fixés par l'administrateur de la grille ou par les propriétaires des ressources, incluant notamment les paramètres de coût,
- ceux dépendant de l'environnement, tels que le nombre de processeurs disponibles, leur puissance et leur charge.

Ces paramètres seront passés à l'algorithme génétique soit par la base de données d'OAR, soit par des fichiers de configuration. La figure 3.20 résume les cas d'utilisation de ces deux moyens.

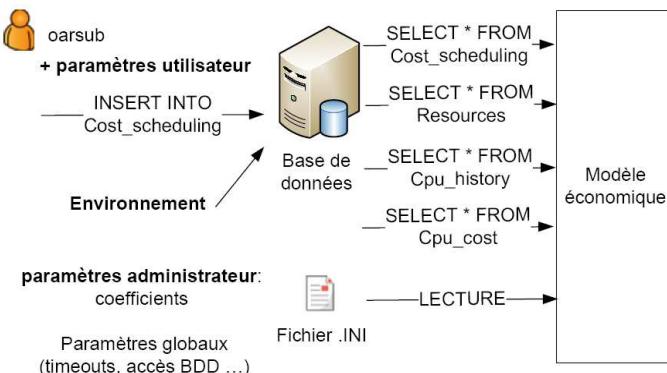


FIG. 3.20 – Transmission des différents paramètres au modèle économique

OAR étant un logiciel en perpétuelle évolution, la modification des tables qui lui sont propres n'est pas envisageable si l'on désire découpler OAR de l'implémentation de notre module d'ordonnancement. Ainsi, pour passer les paramètres à l'algorithme génétique, nous créons nos propres tables (figure 3.21).

Cost_scheduling	Cpu_load	Cpu_cost
Job_id Nb_proc_min Nb_proc_max Nb_tasks Puissance_ref Temps_ref Temps_com Coef_time Coef_cost Etat PID_cost Start_date	Cpu Date_start Date_stop Value	Cpu Date_start Date_stop Value

FIG. 3.21 – Tables créées pour la transmission de paramètres à l'algorithme génétique

La table *Cost_scheduling* contient tous les paramètres fournis par l'utilisateur concernant l'application qu'il soumet. Notons qu'ils sont insérés dans la table par le client d'*oarsub* adapté.

Les tables *Cpu_load* et *Cpu_cost* stockent respectivement la charge et les coefficients locaux de coût des processeurs, ces deux données dépendant du temps. Les valeurs contenues dans la table *Cpu_cost* seront renseignées par l'administrateur de la grille, tandis que la table *Cpu.load* sera mise à jour par le module d'ordonnancement en fonction des placements des applications soumises.

Enfin, un fichier de configuration est mis en place permettant à l'administrateur de renseigner les coefficients de coût des différentes ressources à appliquer (C^T , C^N et C^P). Ce fichier contient également les paramètres nécessaires au fonctionnement de l'algorithme génétique (nombre d'individus constituant les populations, taux de croisement, taux de mutation, etc.).

L'annexe B décrit en détail le contenu des tables et du fichier qui stockent les paramètres du modèle économique.

3.4.4 Mécanismes de placement des tâches

Lorsqu'une application est soumise au module d'ordonnancement, l'algorithme génétique calcule un placement en fonction des paramètres d'entrée qui lui sont transmis de deux manières différentes :

- par le biais de la base de données, la table *Cost_scheduling* contenant les données fournies par l'utilisateur concernant l'application soumise, et les tables *Cpu_cost* et *Cpu_load* contenant respectivement les variations futures du coût et de la charge des processeurs,

- grâce au fichier de configuration décrit dans l'annexe B, contenant les coefficients de tarification fixés par l'administrateur, ainsi que les paramètres propres à l'exécution de l'algorithme génétique (nombre de *threads* à utiliser, nombre d'individus considérés, nombre d'itérations, probabilité d'apparition de mutations, etc.).

Dans un premier temps, le module d'ordonnancement entame une phase d'initialisation, au cours de laquelle il stocke dans la base de données la date courante ainsi que son identifiant de processus, respectivement dans les champs `Start_date` et `PID_cost` de la table `Cost_scheduling`.

L'étape de synchronisation peut alors commencer en appliquant, selon les cas, les actions décrites dans l'annexe C. Notons que ces traitements sont réalisés dans un *thread* séparé, afin de ne pas ralentir le calcul du placement qui s'effectue en parallèle. En effet, certains d'entre eux traitent des cas d'incohérence et peuvent parfois prendre du temps si plusieurs vérifications avec des *timeouts* sont entreprises.

Lorsque l'exécution de l'algorithme génétique prend fin, les résultats du placement sont écrits dans les tables `Gantt-jobs-predictions` et `Gantt-jobs-resources` (figure 3.22).

Gantt_jobs_predictions	Gantt_jobs_resources
Moldable_job_id Start_time	Moldable_job_id Resource_id

FIG. 3.22 – Tables d'OAR contenant le placement des jobs et leur date de début

Dans la table `Gantt-jobs-predictions`, nous écrivons l'identifiant du job (champ `Moldable_job_id`) ainsi que la date de début calculée (`Start_time`). Notre module d'ordonnancement calcule la date en secondes par rapport au référentiel 0 (date de début de l'ordonnancement considéré dans notre modèle économique) : il donne le nombre de secondes à attendre avant le lancement de l'application. Il faut convertir ce référentiel en rajoutant ce nombre de secondes à la date actuelle au format *SQL* puis écrire le résultat dans la base.

Dans la table `Gantt-jobs-resources`, nous devons insérer le vecteur de placement. Il faut pour cela le convertir en autant de lignes correspondant au couple {identifiant, ressource utilisée}. Par exemple si le modèle a placé 2 tâches sur une ressource, on ajoutera 2 lignes identiques.

L'écriture du résultat de l'algorithme génétique dans ces tables de la base de données permettra à OAR de se charger de la réservation des noeuds, et de lancer l'application au moment convenu sur les ressources choisies.

Notons que lorsque la phase d'ordonnancement est terminée, l'utilisateur peut accéder à la liste des ressources qui lui ont été attribuées au travers des outils de visualisation d'OAR. Lorsque l'application est lancée, l'utilisateur peut se connecter aux noeuds comme à l'habitude avec la commande “`oarsub -C job_id`”. L'application quant à elle peut récupérer la liste des noeuds réservés à partir du fichier `$OAR_NODEFILE`, comme cela peut être réalisé lorsque l'application est soumise avec le client `oarsub` classique.

3.4.5 Vue d'ensemble du processus de soumission

En pratique, la soumission d'une application est effectuée en six étapes (figure 3.23) :

1. L'utilisateur invoque l'adaptateur que nous fournissons en lui donnant les caractéristiques de l'application à placer.
2. L'adaptateur vérifie la validité des paramètres fournis par l'utilisateur et les insère dans la base de données en vue de leur utilisation par le module d'ordonnancement. Il fait ensuite appel au client `oarsub` en lui passant les paramètres adéquats, et en spécifiant notamment que la file choisie pour ce job est la file *Cost*.
3. Le client `oarsub` reçoit le job, et, après validation de la requête, il l'insère dans la file *Cost*.
4. Le module d'ordonnancement gérant la file *Cost* est invoqué. Il fait appel à l'algorithme génétique pour déterminer un placement. Celui-ci calcule donc, en fonction des données fournies par l'utilisateur et présentes dans la base de données, la date de début de l'application ainsi que les machines à utiliser.
5. L'ordonnancement calculé est inséré dans la base de données. La charge des machines déterminées par l'algorithme génétique est alors mise à jour afin de prendre en compte la nouvelle application.
6. L'application se trouvant dans la base de données d'OAR, elle est lancée par le module d'exécution d'OAR lorsque la date de début déterminée par l'algorithme génétique arrive.

L'annexe C donne plus de détails quant au processus de soumission d'applications.

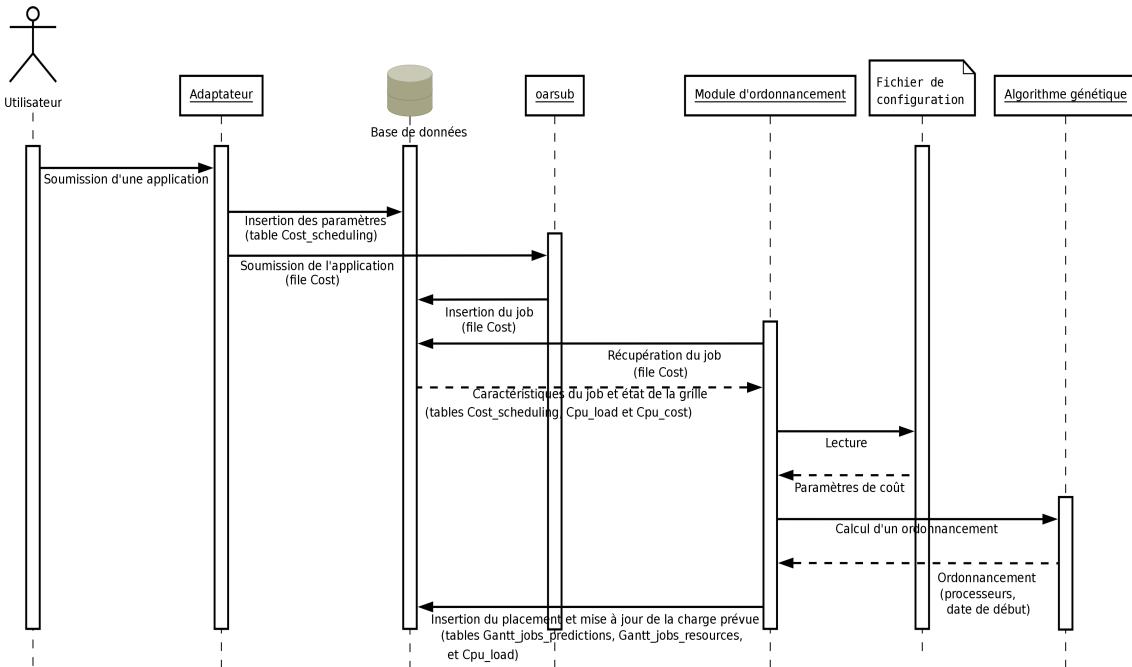


FIG. 3.23 – Soumission d'une application via l'adaptateur du client `oarsub`

3.4.6 Tests et validation

L'objectif de cette section est de présenter les tests réalisés afin de valider l'intégration de l'algorithme génétique implémentant le modèle économique proposé au sein de l'ordonnanceur OAR. Pour cela, nous utilisons OAR couplé à notre module d'ordonnancement en mode simulation : l'infrastructure fournie par OAR est utilisée, mais les applications soumises ne sont pas exécutées. Ceci permet de définir une grille virtuelle, utilisable pour soumettre des applications et calculer les ordonnancements associés.

Le contexte de simulation mis en place consiste en trois *clusters* constitués de cent processeurs chacun. Ces trois clusters sont géographiquement répartis de la manière suivante :

- un cluster à Toulouse, France (fuseau horaire GMT + 1),
- un cluster à Paris, France (fuseau horaire GMT + 1),
- un cluster à Sydney, Australie (fuseau horaire GMT + 10).

Les processeurs utilisés sont les suivants :

- *Clusters de Toulouse et de Sydney* : Intel Xeon cadencé à 3,8 GHz,
- *Cluster de Paris* : AMD Athlon cadencé à 1,2 GHz.

Ainsi, les machines présentes sur les grappes de Toulouse et de Sydney présentent les mêmes caractéristiques, tandis que les ressources parisiennes se montrent environ deux fois moins puissantes (en tenant compte du type des processeurs, de leur vitesse et de leur cache). Les administrateurs de ces différentes grappes ont mis en place une politique de prix identique, basée sur la puissance des ressources. Ainsi, l'utilisation du cluster parisien est plus intéressante en termes financiers, mais pénalise fortement le temps d'exécution des applications qui lui sont soumises.

La gestion des “heures creuses / heures pleines” est réalisée afin de mieux répartir l’offre et la demande au cours de la journée. Ainsi, une période d’heures creuses est mise en place quotidiennement, de 18h à 6h du matin (heure locale), afin de favoriser une utilisation nocturne des ressources. En choisissant comme référentiel le fuseau horaire dans lequel se trouve la France (GMT + 1), la période d’heures creuses australienne se situe entre 9h du matin et 21h.

Deux simulations ont été réalisées. Pour chacune d’elles, l’utilisateur soumet une application composée de 60 tâches, chacune d’elles requérant 12 heures de temps CPU sur un processeur de référence de type AMD Athlon cadencé à 1,2 GHz (ou 6 heures sur un Intel Xeon cadencé à 3,8 GHz), ce qui représente un total d’un mois de temps processeur. Pour la première simulation, la soumission est réalisée à 16h, heure française, soit deux heures avant que les grappes françaises n’entrent en période d’heures creuses, et cinq heures avant que le cluster australien ne passe en heures pleines. La seconde simulation met en scène une soumission dont la date varie de 14h à 18h. Pour des questions de simplification, la charge des processeurs est initialement nulle, et n’entre donc pas en compte dans le calcul du placement.

Simulation 1

Dans un premier temps, l'utilisateur ne prend en compte que le coût de l'application, en spécifiant un coefficient de 100% pour le paramètre de coût. Cela revient à minimiser le critère suivant pour déterminer un placement :

$$T_{\text{target}} = C_{\text{App}}$$

L'ordonnancement obtenu, représenté dans la figure 3.24 (expérience 1), montre que toutes les tâches sont dirigées vers le cluster parisien, ce qui constitue le choix le moins onéreux. L'application durant 12 heures sur cette grappe, la date de début sera choisie afin de profiter au maximum de la période d'heures creuses. Ainsi, l'application est programmée pour être exécutée de 18h à 6h.

Si l'utilisateur souhaite que son application se termine au plus tôt, quel qu'en soit le prix à payer, seule la date de terminaison ne sera considérée dans la minimisation du critère :

$$T_{\text{target}} = T_{\text{App}}$$

Dans ce cas (figure 3.24, expérience 2), la grappe se trouvant à Paris n'est jamais sollicitée. En revanche, les clusters de Toulouse ou de Sydney sont indifféremment choisis, car ils ont les mêmes performances en termes de capacité de calcul. De plus, le prix à payer n'étant pas pris en compte dans le choix du placement, l'application est démarrée immédiatement sur l'un de ces deux clusters, et elle se termine seulement 6 heures plus tard, soit à 22h (heure française).

Notons cependant que, suivant la grappe choisie, le coût de l'application ne sera pas le même. En effet, sur les 6 heures que représentent sa durée, 5 heures seront passées en heures creuses sur le cluster de Sydney, contre deux heures seulement sur le cluster toulousain. En pratique, l'utilisateur aura intérêt à inclure le coût de l'application dans le critère à optimiser, dans une proportion certes faible, mais suffisante pour proscrire, dans ce cas, l'utilisation de la grappe toulousaine.

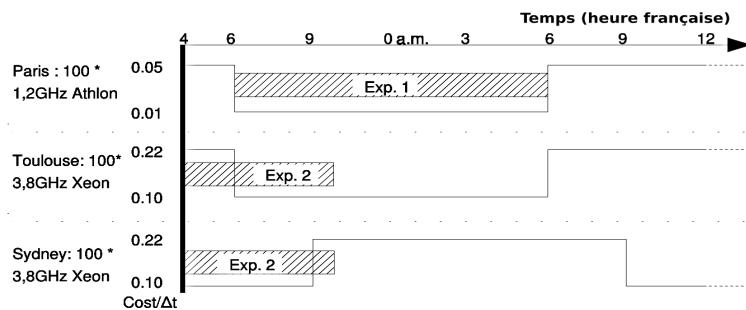


FIG. 3.24 – Placement d'une application, avec $W^C = 100\%$, $W^T = 0\%$ (Exp. 1) et $W^C = 0\%$, $W^T = 100\%$ (Exp. 2)

Simulation 2

Dans cette série d'expériences, la date de soumission de l'application varie de 14h à 18h, heures françaises. L'utilisateur applique un coefficient de 10% sur la prise en compte de la date de terminaison de l'application dans le critère à minimiser. Le coefficient de coût reste donc prépondérant (90%) dans le choix du placement. La grappe parisienne ne pourra toutefois jamais être choisie à cause de l'importance, certes faible, donnée au temps d'exécution de l'application.

La figure 3.25 montre que le module d'ordonnancement d'OAR choisit alternativement Sydney ou Toulouse, en démarrant l'exécution de l'application tout de suite après la soumission pour le cluster de Sydney, tandis qu'un délai est appliqué à la grappe toulousaine.

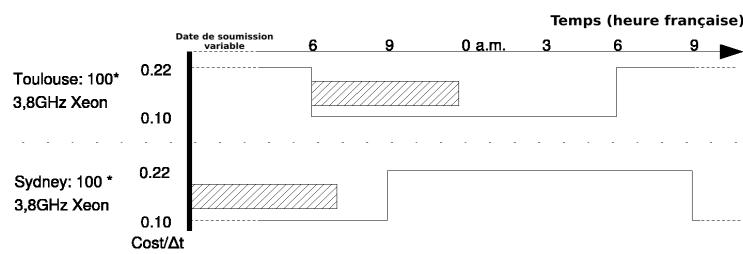
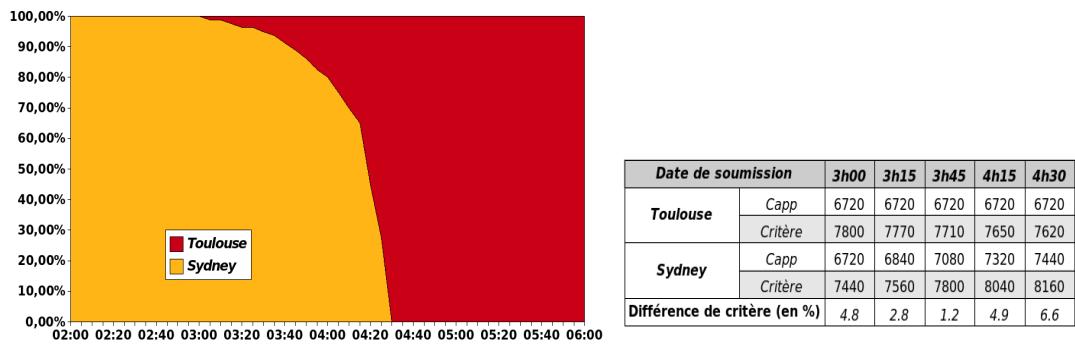


FIG. 3.25 – Possibilités de placement de l'application entre Toulouse et Sydney

Le choix de ces clusters dépend de la date à laquelle l'application est soumise, comme le montre la figure 3.26. En effet, si l'application est soumise suffisamment tôt, la grappe de Sydney est plus intéressante que celle de Toulouse, car elle bénéficie de la période d'heures creuses. L'application peut alors être exécutée tout de suite pour minimiser également la date de fin de l'application. Cependant, plus l'application est soumise tardivement, plus le choix du cluster toulousain devient judicieux. En effet, la période d'heures creuses de Sydney se réduit, et il devient plus intéressant de placer l'application à Toulouse afin de bénéficier de la baisse de tarif survenant à 18h. La date de début est d'ailleurs, dans ce cas, repoussée à 18h, afin de bénéficier du tarif réduit sur toute la durée de l'application.



(a) Proportion du choix entre la grappe de calcul de Toulouse et celle de Sydney

(b) Valeur du critère correspondant aux clusters de Toulouse et de Sydney pour différentes dates

FIG. 3.26 – Choix de placement de l'application entre la grappe de Toulouse et celle de Sydney

Il est toutefois possible de remarquer que la transition entre Sydney et Toulouse se fait progressivement, de 15h à 16h30, heures françaises. Lorsque la soumission est effectuée avant 15h, le cluster de Sydney est choisi systématiquement, de même qu'après 16h30, le choix est toujours reporté sur la grappe toulousaine. Ceci coïncide avec le fait que le critère de Sydney augmente au cours du temps, tandis que celui de Toulouse diminue. Toutefois, entre 15h et 16h30, l'écart entre les critères calculés pour les deux clusters n'est pas suffisamment élevé pour permettre au module d'ordonnancement de formuler un choix tranché.

A 15h35, les deux grappes de calcul présentent la même valeur de critère, les rendant donc, en théorie, strictement équivalentes pour le choix de placement. Cependant, la figure 3.26 montre que le module d'ordonnancement choisit encore Sydney à 95% à cette date, au lieu des 50% auxquels on pourrait s'attendre. Ce décalage est dû au codage de l'algorithme génétique, qui privilégie la rapidité d'exécution au détriment d'une légère imprécision sur le critère calculé quand la différence de critères entre les solutions de placement potentielles est très faible. En effet, dans le cadre de cette expérience, la différence calculée entre les valeurs du critère pour les solutions de placement sur Sydney ou Toulouse n'excède pas 6,6%. Ainsi, durant la phase transitoire allant de 15h à 16h30, l'algorithme génétique ne trouve peut-être pas toujours la solution optimale, mais il en reste toutefois très proche.

3.5 Conclusion

Dans ce chapitre, nous avons présenté un modèle économique permettant de gérer les ressources d'une grille de calcul. Plus précisément, ce modèle a pour objectif de calculer un placement lorsque des applications lui sont soumises. Les applications considérées ici sont des applications parallèles à gros grain, constituées de plusieurs tâches identiques et synchrones.

Divers paramètres sont pris en compte lors du calcul du placement de telles applications :

- les caractéristiques de l'application, incluant le nombre de tâches qui la composent et la quantité de calculs devant être effectuée par celles-ci,
- le prix des ressources, pouvant varier au cours du temps,
- la puissance des ressources,
- la charge des ressources, dont l'évolution au cours du temps est estimée en fonction des applications déjà placées sur la grille,
- les paramètres de coût définis par les administrateurs de la grille,
- les besoins de l'utilisateur,
- etc.

Le placement choisi reflète un compromis entre le coût de l'application et la rapidité avec laquelle elle se terminera, tel que spécifié par l'utilisateur. Ce compromis permet ainsi d'aboutir au choix des ressources qui exécuteront l'application soumise, mais également de la date à laquelle l'application devra être exécutée, une exécution immédiate ne s'avérant pas forcément être le choix le plus judicieux.

Le modèle mathématique défini dans ce chapitre soulève un problème d'optimisation non-linéaire, à variables entières et sous contraintes. Les outils classiques d'optimisation

se prêtant mal à la résolution d'un tel problème, nous avons développé un algorithme génétique permettant de répondre à nos besoins.

Cet algorithme montre des résultats tout-à-fait encourageants, tant en termes de précision des résultats obtenus, qu'en termes de performances. En effet, le calcul du placement d'une application de 40 tâches sur 5000 processeurs ne prend que quelques secondes, ce qui est satisfaisant au regard de la durée des applications placées sur des supports tels que les grilles de calcul, durée qui atteint souvent plusieurs heures.

Enfin, l'algorithme génétique développé a été intégré au sein d'un ordonnanceur existant, afin de pouvoir être utilisé dans un environnement opérationnel. Nous avons pour cela choisi l'ordonnanceur OAR, utilisé sur la plate-forme nationale de recherche *Grid'5000*.

Le modèle économique développé nécessite de connaître à l'avance une estimation du temps d'exécution de l'application, dans le but d'évaluer la quantité de calculs devant être effectuée, et de l'inclure ainsi dans le choix du placement. Si ce temps peut être fourni par l'utilisateur, comme c'est le cas actuellement sous OAR, cette solution n'est pas nécessairement la meilleure car la plupart des utilisateurs ont tendance à surestimer, parfois de manière importante, le temps d'exécution de leurs applications. Les chapitres suivants proposent ainsi une méthode permettant d'estimer, de façon fiable, le temps d'exécution d'une application en fonction des entrées qui lui sont appliquées.

Chapitre 4

Etude de l'estimation du temps d'exécution d'une application

L'objectif de ce chapitre est de définir les bases de l'étude de l'estimation du temps d'exécution d'une application. Dans une première partie de ce chapitre, un ensemble de techniques de prédiction est abordé. Ces dernières se divisent en deux familles : les techniques de prédiction basées sur un historique, et celles basées sur le profil des applications. Dans une seconde partie, l'approche adoptée dans cette thèse est introduite : il s'agit d'une méthode hybride combinant ces deux types de prédiction.

4.1 Introduction

De nombreuses recherches ont été effectuées dans le domaine de la prédiction du temps d'exécution d'applications, afin de déterminer comment relier le temps d'exécution d'une application avec le contexte dans lequel elle est lancée (entrées de l'application, plate-forme d'exécution, etc.). L'objectif final est ainsi de pouvoir estimer le temps d'exécution d'une application avant même qu'elle ne soit lancée.

Dans le domaine du temps réel par exemple, l'utilité d'une telle donnée est cruciale au bon fonctionnement des systèmes, qu'ils soient critiques ou non [81]. En effet, les applications temps réel sont soumises à des contraintes temporelles (*deadlines*) qu'il convient de respecter impérativement pour les systèmes temps réel strict, ou au mieux pour les systèmes temps réel souple. Dans tous les cas, la connaissance du temps d'exécution des applications est nécessaire aux ordonnanceurs temps réel pour gérer l'ordre d'exécution des applications qui leur sont soumises [82, 83, 84]. Plus qu'une simple estimation, ils utilisent le temps d'exécution des applications obtenu dans le pire des cas (*WCET* pour *Worst-Case Execution Time*).

Les mécanismes d'ordonnancement appliqués aux domaines des grappes et des grilles de calcul nécessitent également de connaître, à priori, une estimation de la durée des applications à placer [27, 85, 86, 13, 87]. C'est notamment le cas pour le modèle économique proposé dans cette thèse [69].

Concernant le domaine des grilles de calcul, cette donnée peut être fournie par les utilisateurs au moment où ils soumettent leurs applications. Les performances du placement les incitent généralement à donner une estimation relativement correcte du temps

d'exécution de leurs applications [88, 89]. Par exemple, un temps sous-estimé entraîne un arrêt prématuré de l'application au terme de la période de temps réservée, tandis qu'un temps surestimé entraînera un démarrage tardif de l'application si aucun créneau libre suffisamment important n'est trouvé. Ce type de problématique peut notamment être rencontré sur la plate-forme de recherche *Grid'5000* [7], avec l'outil de réservation OAR.

Cependant, il peut être intéressant d'automatiser l'estimation du temps d'exécution des applications soumises, d'une part pour éviter les erreurs dues aux mauvaises estimations effectuées par les utilisateurs, et d'autre part par soucis d'ergonomie et de facilité d'emploi des outils d'ordonnancement.

Deux grandes familles de techniques de prédition sont examinées dans ce chapitre :

- La prédition basée sur un historique d'exécutions passées (*historic-based prediction*) : cette technique est notamment utilisée pour prédire le temps d'applications séquentielles ou parallèles, en vue de les placer sur un support d'exécution tel qu'un *cluster* ou une grille de calcul.
- La prédition basée sur le profil des applications (*profile-based prediction*) : ce type d'analyse est particulièrement utilisé dans le domaine du temps réel en vue de déterminer le temps d'exécution d'une application dans le pire des cas.

4.2 Travaux portant sur l'estimation du WCET d'une application

4.2.1 Objectifs et contexte

Le temps d'exécution dans le pire des cas, ou WCET, est le temps maximal que mettra l'application pour s'exécuter, et ce pour l'ensemble des jeux d'entrées possibles [90, 91]. Il s'agit donc du temps mis par l'application pour s'exécuter dans le cas où ses entrées déterminent un chemin d'exécution maximisant le temps d'exécution. Il est à noter que le WCET d'une application est spécifique à une architecture matérielle donnée.

L'estimation du WCET d'une application est particulièrement utile dans le domaine du temps réel, pour lequel des contraintes temporelles fortes doivent être respectées, sous peine d'avoir des conséquences catastrophiques sur le système. La connaissance du temps d'exécution maximal d'une application temps réel peut permettre de :

- dimensionner un système temps réel, c'est-à-dire déterminer par exemple la puissance du matériel nécessaire à l'exécution d'un ensemble défini de tâches, ou bien calculer le nombre de tâches supportées par un système matériel donné, afin de s'assurer que l'échéance de chacune des tâches soit respectée,
- déterminer un ordonnancement hors-ligne, pour lequel la durée maximale de chacune des tâches à ordonner doit être connue à l'avance [92],
- optimiser le code d'une application, grâce à l'obtention des chemins d'exécution critiques.

Cependant, pour être exploitable, le WCET déterminé doit satisfaire deux critères principaux [91]. Le premier, totalement impératif, est un critère de sûreté. En effet, le WCET

calculé doit absolument être supérieur au temps d'exécution maximal réel de l'application. La non-satisfaction de cette contrainte pourrait entraîner le non-respect d'échéances critiques du système temps réel.

Le second critère à satisfaire est un critère de précision. Celui-ci peut être traité de manière plus souple que le précédent. La précision du WCET permet d'exprimer l'écart entre la valeur calculée et la valeur réelle. Pour être utile, l'estimation du temps d'exécution maximal d'une application ne doit pas être trop pessimiste, sous peine d'entraîner un surdimensionnement du système.

4.2.2 Méthodes dynamiques d'analyse du WCET

Dans cette classe de méthodes, le temps d'exécution du programme est mesuré, soit sur un système réel ou bien à l'aide d'un simulateur. L'application est ainsi exécutée sur le matériel cible, et une mesure de son temps d'exécution est effectuée [93]. Lorsqu'une telle exécution est impossible, un simulateur logiciel peut être utilisé pour simuler le système matériel.

Quel que soit le moyen utilisé pour mesurer le WCET, un jeu d'entrées est utilisé pour exécuter le programme. La principale difficulté des méthodes dynamiques réside ainsi dans la détermination d'un jeu d'entrées conduisant au temps d'exécution le plus long. Pour cela, il est possible d'avoir recours à des jeux de test explicites, ou bien à des jeux de test symboliques.

Les jeux de test explicites correspondent à des entrées réelles du programme. Ces entrées seront utilisées tour à tour pour exécuter le programme, et le temps d'exécution le plus long obtenu sera considéré comme étant le WCET de l'application. Dès lors, la principale difficulté qui se pose est la confiance qui peut être accordée à cette méthode, dans la mesure où les jeux de test sélectionnés doivent comporter le jeu d'entrées amenant au temps d'exécution dans le pire des cas. Par conséquent, l'importance du choix des jeux de test est mise en avant. Pour résoudre ce problème, plusieurs solutions sont envisageables :

- **Mesurer le temps d'exécution d'un programme pour l'ensemble de ses jeux d'entrées possibles.** Cette solution n'est possible que si le nombre de jeux d'entrées de l'application est raisonnable. De plus, le temps mis pour exécuter tous les jeux de test est dans la plupart des cas rédhibitoire.
- **Effectuer les mesures à partir des jeux d'entrées définis par l'utilisateur.** Cette solution s'appuie sur la connaissance que possède ce dernier du programme, et ainsi sur son aptitude à identifier le jeu de test amenant au WCET. Cependant, cette méthode n'est pas entièrement fiable, et son utilisation est limitée à des applications simples.
- **Générer automatiquement un ensemble de jeux de test** qui doit amener au temps d'exécution maximal de l'application. Cette méthode utilise divers algorithmes, tels que les algorithmes évolutionnistes [94], et peut également s'appuyer sur l'utilisation d'un cluster lorsque le nombre de jeux de test générés est très élevé [95].

Une seconde classe de jeux de test envisageables pour obtenir une mesure du WCET d'une application est la classe des jeux de test symboliques [96]. Dans ce cas, les entrées du

programme sont considérées comme étant inconnues, et le jeu d'instruction du processeur cible est étendu afin de prendre en compte les calculs sur des opérandes à valeurs inconnues. Les jeux de test symboliques ne peuvent être utilisés que dans le cadre d'une exécution de l'application sur un simulateur logiciel.

4.2.3 Méthodes statiques d'analyse du WCET

L'analyse statique d'un programme consiste en une analyse de la structure du programme à partir de son code source ou de son code objet, dans le but d'en déduire son WCET [81]. Elle s'effectue en trois étapes :

1. **L'analyse de flot** : cette phase permet de déterminer l'ensemble des chemins d'exécution possibles dans le programme.
2. **L'analyse de bas niveau** : elle permet d'évaluer l'impact de l'architecture matérielle sur le WCET.
3. **Le calcul du WCET** : la valeur du WCET est déterminée à partir du résultat des deux phases précédentes.

4.2.3.1 Analyse de flot

L'analyse de flot a pour but de déterminer l'ensemble des chemins d'exécution possibles d'un programme. Pour cela, la première étape consiste à découper le code de ce programme en blocs de base.

Un bloc de base est une séquence maximale d'instructions possédant un et un seul point d'entrée ainsi qu'un et un seul point de sortie dans le flot de contrôle du programme. Un bloc de base ne contient ainsi que des instructions simples qui excluent toute instruction de branchement (structures de contrôle, appels de fonctions, etc.) [97].

Graphe de flot de contrôle et arbre syntaxique

Un graphe de flot de contrôle peut être utilisé afin de décrire graphiquement tous les enchaînements possibles entre les différents blocs de base. Les noeuds de ce graphe représentent les blocs de base de l'application, tandis que les arcs expriment les relations de précédence entre ces blocs.

Il est également possible de s'appuyer sur une représentation du programme à l'aide d'un arbre syntaxique. Ce dernier est construit à partir du code source de l'application, dans le but d'en décrire la structure syntaxique. Les noeuds de cet arbre représentent les structures du langage de haut niveau, tandis que les feuilles représentent les blocs de base du programme.

Le code suivant montre l'exemple d'une fonction réalisant le produit de deux matrices carrées de même dimension. Le tableau 4.2.3.1 montre comment découper ce programme en blocs de base. La figure 4.1 montre le graphe de flot de contrôle ainsi que l'arbre syntaxique qui lui sont associés.

```

1 // Multiplicates two matrixes
2 void matrixMultiplication(int dimensions, unsigned long leftMatrix [dimensions][dimensions],
3                           unsigned long rightMatrix [dimensions][dimensions],
4                           unsigned long resultMatrix [dimensions][dimensions]) {
5
6     for ( int i = 0 ; i < dimensions ; i++ ) {
7         for ( int j = 0 ; j < dimensions ; j++ ) {
8             resultMatrix [i][j] = 0;
9             for ( int k = 0 ; k < dimensions ; k++ ) {
10                 resultMatrix [i][j] += leftMatrix [i][k] * rightMatrix [k][j];
11             }
12         }
13     }
14 }
15 }
```

Blocs de base	Code associé
BB_1	<code>int i = 0;</code>
BB_2	<code>i < dimensions</code>
BB_3	<code>int j = 0;</code>
BB_4	<code>j < dimensions</code>
BB_5	<code>resultMatrix[i][j] = 0; int k = 0;</code>
BB_6	<code>k < dimensions</code>
BB_7	<code>resultMatrix[i][j] += leftMatrix[i][k] * rightMatrix[k][j]; k++;</code>
BB_8	<code>Fin de la fonction</code>
BB_9	<code>i++;</code>
BB_{10}	<code>j++;</code>

TAB. 4.1 – Découpage de la fonction en blocs de base

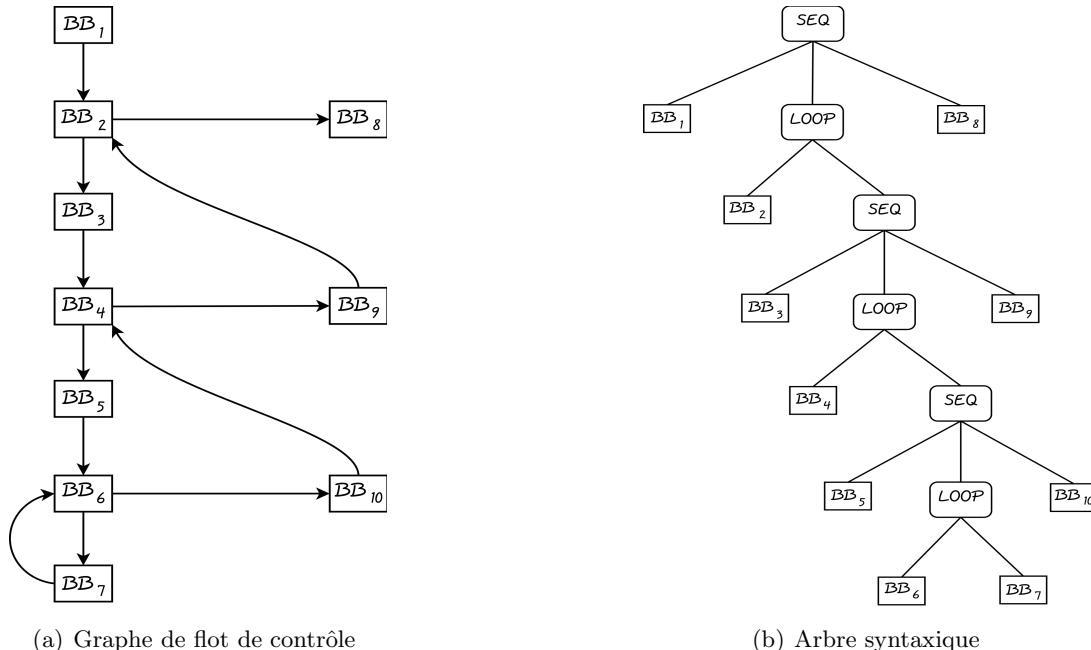


FIG. 4.1 – Outils graphiques pour l’analyse statique du WCET

Informations complémentaires sur le flot de contrôle

Dans le cas général, l'analyse de flot n'est pas un problème décidable [97]. En effet, certaines propriétés du programme peuvent impliquer que certains chemins d'exécution ne soient pas finis. Ainsi, trouver des chemins d'exécution finis et exploitables revient à un problème de l'arrêt des algorithmes de parcours de graphes.

Des études ont démontré que l'analyse de flot est un problème décidable si les conditions suivantes sont respectées [98, 90] :

- absence de structures dynamiques telles que des pointeurs ou des tableaux dynamiques,
- absence de récursivité,
- utilisation exclusive de boucles bornées.

Ces restrictions à apporter aux programmes ne sont parfois pas directement exprimables par leur structure. Ainsi, le graphe de flot de contrôle et l'arbre syntaxique ne reflètent pas ces conditions, même si elles existent. Il est donc nécessaire d'obtenir des informations complémentaires. Ces dernières ont pour objectif de restreindre le nombre de chemins d'exécution possibles, et ainsi d'améliorer la précision du WCET estimé. En pratique, elles expriment des contraintes sur le programme, telles que des bornes sur le nombre d'itérations des boucles ou bien l'identification de chemins infaisables.

Ces informations complémentaires sur le flot de contrôle peuvent être obtenues de plusieurs manières. Elles peuvent être demandées à l'utilisateur, qui a une connaissance suffisante de l'application pour pouvoir les fournir. Il peut pour cela utiliser un langage spécifique pour exprimer ces informations, ou bien écrire des annotations dans le programme ou dans un fichier séparé [99, 100, 101, 102, 90].

Il existe également des méthodes permettant de déduire automatiquement ces informations. Ainsi, dans [103, 104, 105, 106], les auteurs proposent différentes méthodes mettant en œuvre une analyse du flot de données du programme, dans le but de connaître la valeur de certaines variables en différents points du programme. Ceci leur permet ensuite de déduire de manière automatique les informations concernant les chemins infaisables ainsi que les bornes sur le nombre d'itérations des boucles. Ces informations peuvent également être déterminées en combinant des techniques d'énumération de chemins avec une exécution symbolique [107, 108].

Enfin, une autre approche pour maîtriser le flot de contrôle est également envisageable. Il s'agit d'apporter des modifications au code du programme afin de faciliter la détermination du WCET. Ainsi, dans [109], l'auteur propose une méthode permettant d'éliminer de manière automatique les boucles non-structurées de l'analyse de flot. Une autre solution consiste à contraindre l'exécution du programme à un seul chemin [110, 111]. Si d'ordinaire, différents jeux d'entrées conduisent à des chemins différents avec des temps d'exécution différents, il est en effet possible de modifier le code du programme afin de rendre le chemin d'exécution indépendant des entrées, facilitant ainsi la détermination du WCET.

Code utilisé

Lors de la phase d'analyse de flot se pose la question du code à utiliser pour mener à bien cette phase.

Le code source permet de travailler au niveau du langage de programmation. L'analyse qui en résulte est très indépendante de l'architecture du processeur cible. L'utilisation du code source est nécessaire pour obtenir l'arbre syntaxique, mais l'utilisation exclusive de ce code entraîne des difficultés de prédiction des bornes temporelles, car celle-ci ne peut pas se faire indépendamment du contexte, du compilateur et du processeur cible.

Le code objet permet de travailler au niveau du langage d'assemblage, ce qui permet de prendre en compte tous les effets de l'architecture matérielle donnée. Même si elle offre cet avantage, une analyse exclusivement de bas niveau n'est pas simple, notamment pour les phases faisant intervenir l'utilisateur.

La solution généralement adoptée pour effectuer l'analyse de flot d'un programme est d'utiliser conjointement code source et code objet. Le code source est utilisé pour obtenir des informations fonctionnelles sur le programme, par le biais d'annotations. Ensuite, le code objet permettra d'analyser le programme et d'en déduire son WCET grâce aux informations précédentes.

4.2.3.2 Analyse de bas niveau

La seconde phase de la déduction du temps d'exécution maximal d'un programme par son analyse statique est la phase d'analyse de bas niveau. Elle consiste à estimer le temps d'exécution maximal de chacun des blocs de base sur une architecture matérielle donnée.

Cette analyse, effectuée à partir du code objet, repose essentiellement sur la précision des modèles matériels utilisés. Les systèmes matériels intègrent des mécanismes permettant d'accélérer le temps d'exécution des programmes, tels que :

- des pipelines,
- des unités de prédiction de branchement,
- des unités d'exécution multiple,
- des caches.

Les modèles matériels dont on dispose n'en tiennent pas toujours compte. En effet, l'obtention de telles informations est souvent rendue difficile à cause des documentations constructeurs qui sont incomplètes, et car la modélisation de tels mécanismes n'est pas aisée et entraîne souvent l'introduction d'erreurs de modélisation. Cependant, de nombreux travaux ont été menés afin d'analyser l'impact de mécanismes tels que les caches d'instructions [112, 113, 114], les caches de données [115, 116], les prédictions de branchement [117, 118], ou encore les pipelines [112, 113, 119, 120], et de leur répercuter sur l'estimation du WCET.

La prise en compte de ces mécanismes d'accélération de code n'est pas obligatoire puisque si le temps d'exécution des blocs de base est calculé sans tenir compte, une borne maximale de leurs temps d'exécution est obtenue. Cependant, l'imprécision des modèles

d'architectures matérielles implique une estimation du WCET qui en est d'autant plus pessimiste.

Une approche mêlant analyse statique et analyse dynamique peut être utilisée [121, 122, 123, 124]. En effet, l'analyse de bas niveau est limitée par la précision des modèles utilisés pour représenter l'architecture matérielle. Dans ce cas, l'analyse de flot est couplée à une analyse de bas niveau reposant sur des mesures de temps d'exécution pour chacun des blocs de base du programme. Ces mesures peuvent être directement prises à partir d'exécutions du programme sur l'architecture matérielle cible, ou bien en utilisant des simulateurs logiciels.

4.2.3.3 Calcul du WCET

L'ensemble des chemins possibles au sein du programmes sont maintenant connus grâce à l'analyse statique. De même, l'analyse de bas niveau a permis de déterminer le temps d'exécution maximal des blocs de base constituant le programme. Le calcul du WCET peut alors être réalisé à partir de ces deux informations.

Techniques basées sur les chemins d'exécution

La première méthode de calcul du WCET s'appuie sur l'algorithme des graphes. Il s'agit d'associer la valeur du WCET de chacun des blocs de base du programme aux nœuds du graphe de flot de contrôle. Ainsi, un graphe valué est obtenu. Il est alors possible d'appliquer les techniques traditionnelles de recherche du pire chemin dans un graphe issues de la théorie des graphes [113, 125, 126].

Lorsqu'un chemin maximisant le temps d'exécution est ainsi trouvé, une vérification est réalisée afin de déterminer s'il s'agit bien d'un chemin d'exécution possible. Si ce n'est pas le cas, il est exclu du graphe, et une nouvelle recherche est commencée.

Ces techniques, qualifiées de *path-based techniques*, permettent ainsi d'obtenir le chemin d'exécution aboutissant au pire temps d'exécution. Cette connaissance peut, en outre, être utilisée pour optimiser un programme dans le but d'en diminuer le WCET.

Techniques basées sur l'arbre syntaxique

Une autre classe de méthodes s'appuie sur l'arbre syntaxique du programme. Le WCET est, dans ce cas, calculé de manière récursive. Un parcours de l'arbre est effectué de bas en haut, en partant des feuilles qui correspondent aux blocs de base, et qui contiennent donc l'information de WCET obtenue au cours de l'analyse de bas niveau. Cette information est ensuite remontée jusqu'à la racine en fonction des noeuds rencontrés [127, 117, 128].

Ainsi, le WCET associé à une séquence (nœud de type *SEQ*) correspond à la somme des WCET des structures qui la composent (nœuds fils). Au contraire, le WCET d'une structure conditionnelle s'appuiera sur l'utilisation de l'opérateur *max* permettant de choisir la branche conduisant au pire temps d'exécution.

Ce type de technique, qualifiée de *tree-based*, aboutit à l'obtention d'un arbre temporel associant les WCET calculés à chaque nœud de l'arbre syntaxique.

Techniques d'énumération implicite des chemins

La technique qui rencontre le plus de succès dans la littérature est celle d'énumération implicite des chemins (*IPET* pour *Implicit Path Enumeration Technique*) [129, 130]. Cette technique exploite l'inutilité d'énumérer explicitement les chemins d'exécution menant au pire scénario, le but étant de calculer simplement le WCET sans se préoccuper du chemin permettant d'y aboutir.

L'objectif de cette méthode est de transformer le graphe de flot de contrôle en un ensemble de contraintes devant être respectées, afin de se ramener à un problème d'optimisation linéaire à variables entières [97, 131]. Deux types de contraintes sont à distinguer :

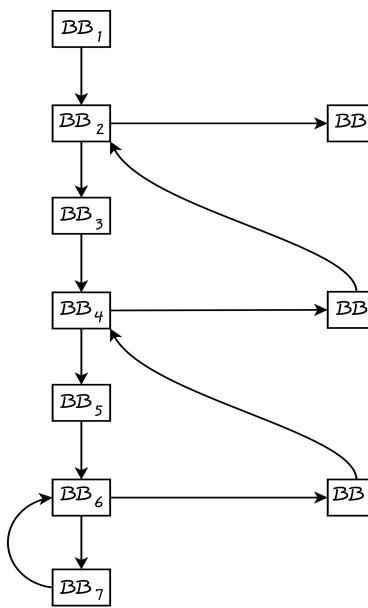
- **contraintes structurelles** : ces contraintes décrivent la structure du graphe,
- **contraintes fonctionnelles** : il s'agit des contraintes exprimant les informations complémentaires sur le flot de contrôle.

L'analyse de flot permet de connaître le découpage d'un programme en blocs de base. L'analyse de bas niveau, quant à elle, fournit le majorant des temps d'exécution pour chacun de ces blocs. On note ainsi T_b^{BB} le WCET du bloc de base correspondant au nœud b du graphe de flot de contrôle. L'analyse IPET cherche à déterminer le nombre d'occurrences de chacun des nœuds du graphe dans le pire chemin d'exécution, noté N_b^{BB} .

Les contraintes structurelles permettent d'exprimer le fait que, pour chaque nœud du graphe de flot de contrôle, la somme du nombre d'occurrences du bloc de base considéré doit être égale à la fois à la somme des occurrences de ses blocs prédécesseurs et de celles de ses blocs successeurs. Quant aux contraintes fonctionnelles, elles permettent de limiter le nombre de chemins possibles, en limitant le nombre d'occurrences de certains nœuds (par exemple pour exprimer le nombre maximum d'itérations de boucles), ou bien en définissant des contraintes d'exclusion mutuelle.

L'exemple de la fonction `matrixMultiplication` décrit précédemment permet de définir les contraintes suivantes :

Contraintes structurelles :



$$\left\{ \begin{array}{l} N_1^{BB} + N_9^{BB} = N_2^{BB} \\ N_2^{BB} = N_3^{BB} + N_8^{BB} \\ N_3^{BB} + N_{10}^{BB} = N_4^{BB} \\ N_4^{BB} = N_5^{BB} + N_9^{BB} \\ N_5^{BB} + N_7^{BB} = N_6^{BB} \\ N_6^{BB} = N_7^{BB} + N_{10}^{BB} \end{array} \right.$$

Contraintes fonctionnelles :

$$\left\{ \begin{array}{l} N_1^{BB} = 1 \\ N_3^{BB} = \text{dimensions} \\ N_3^{BB} = \text{dimensions}^2 \\ N_3^{BB} = \text{dimensions}^3 \end{array} \right.$$

Le calcul du WCET se ramène ainsi à un problème d'optimisation linéaire à variables entières sous contraintes. Les inconnues N_b^{BB} sont déterminées en minimisant la fonction objectif :

$$WCET = \sum_b N_b^{BB} \cdot T_b^{BB} \quad (4.1)$$

Le chemin emprunté dans le pire des cas n'est pas fourni par cette méthode. Seul le nombre d'occurrences de chacun des blocs de base dans le chemin d'exécution est déterminé.

4.3 Prédiction basée sur un historique d'exécutions passées

4.3.1 Principe général

Dans cette approche, l'estimation du temps d'exécution d'une application est effectuée d'après les temps d'exécution de cette même application obtenus par le passé.

Cette approche considère que le temps d'exécution d'une application dépend du contexte dans lequel elle est lancée, et que deux exécutions effectuées dans des contextes relativement proches produiront des temps d'exécution voisins [87, 132, 88, 86]. Le problème qui se pose alors est de définir et de quantifier cette notion de proximité.

Le contexte d'exécution d'une application comprend tous les éléments ayant une influence sur son temps d'exécution, tels que l'architecture du matériel servant à l'exécution et les entrées du programme. Par la suite, pour une plus grande clarté, nous réduirons ce contexte aux seules entrées du programme. L'architecture cible pourra être prise en compte de plusieurs manières :

- en l'incluant au modèle présenté par la suite, et en la traitant de la même manière que les entrées du programme,
- en conservant le modèle, mais en ayant un historique différent par type d'architecture cible.

4.3.2 Etat de l'art

Approche par catégorisation des applications

L'approche choisie par Gibbons [87, 133], puis par Smith et al. [88] et Krishnaswamy et al. [134], est de classifier les applications en différentes catégories. Ces dernières sont définies à l'aide de *templates* identifiant un ensemble de caractéristiques des applications, permettant d'affecter chaque job à une ou plusieurs catégories d'applications ainsi définies.

Différents *templates* sont définis, chacun regroupant une combinaison différente de caractéristiques parmi les suivantes :

- le type de l'application (batch ou interactive, séquentielle ou parallèle),
- la file de soumission utilisée,
- l'utilisateur,
- l'exéutable,
- les arguments,
- le nombre de noeuds, etc.

Par exemple, un *template* regroupant la file dans laquelle l'application est soumise et l'identifiant de l'utilisateur ayant effectué cette soumission génère un ensemble de catégories telles que : {file **cost**, utilisateur **bmiegemo**}, {file **admin**, utilisateur **monteill**}, etc.

Ces *templates* permettent donc de regrouper les différentes applications et de calculer, pour chacun des groupes obtenus, le temps d'exécution moyen des applications qui le composent. Ces valeurs calculées sont ensuite utilisées pour effectuer la prédiction du temps d'exécution des applications futures. Notons que les catégories définies ne sont pas nécessairement disjointes, un même job pouvant apparaître dans plusieurs catégories.

Ainsi, deux applications appartenant à une même catégorie seront qualifiées de "similaires", tandis que si elles n'apparaissent ensemble dans aucune des catégories créées, elles seront au contraire jugées comme étant "différentes". Lorsqu'une application est soumise, on peut alors estimer son temps d'exécution à partir des applications qui lui sont similaires.

L'exactitude d'une telle prédiction est soumise à la définition d'un ensemble de *templates* approprié au contexte dans lequel on se place. Si un trop grand nombre de catégories est défini, celles-ci ne contiendront pas assez de jobs pour effectuer des prédictions suffisamment précises. En revanche, si un nombre trop faible de catégories sont considérées, des applications qui n'ont pas suffisamment de liens entre elles seront regroupées, aboutissant également à des prédictions peu fiables.

Gibbons définit dans [87, 133] un ensemble fixe de six *templates* combinant les caractéristiques suivantes : la file de soumission utilisée, l'utilisateur effectuant la soumission, le nom de l'exécutable et le nombre de noeuds utilisés. Il est cependant possible, pour améliorer la qualité des prédictions, de définir l'ensemble des *templates* à utiliser grâce à des techniques qui déterminent les caractéristiques des applications les plus à même de fournir une meilleure définition de la similarité. Smith et al. ont pour cela mis au point deux algorithmes, un algorithme glouton et un algorithme génétique, qui fournissent automatiquement un ensemble de *templates* améliorant la qualité des prédictions par rapport à celles fournies par la technique de Gibbons [88]. De la même manière, Krishnaswamy et al. utilisent un algorithme ensembliste d'approximation pour sélectionner les *templates* définissant au mieux la similarité [134].

Approche par apprentissage basé sur des instances

Cette approche s'appuie sur une base de connaissances contenant toutes les exécutions d'applications passées [13, 86]. Chaque job passé constitue ainsi une expérience, possédant à la fois :

- des **entrées** : ce sont les conditions sous lesquelles l'expérience est observée,
- des **sorties** : ce qui peut être observé sous ces conditions.

Des techniques d'apprentissage à partir d'instances [135, 136] sont alors mises en place pour trouver les jobs similaires à celui faisant l'objet de l'estimation de son temps d'exécution. Ainsi, lorsqu'une requête de prédiction est formulée, ses entrées sont présentées à la base de connaissances. Les expériences passées sont alors examinées pour savoir lesquelles

sont les plus appropriées pour participer à la prédiction du temps d'exécution de l'application soumise. Cette sélection est réalisée en utilisant la notion de distance [137] entre la requête et les expériences de la base de connaissance.

Cette approche de prédiction servira de base aux travaux menés dans cette thèse. La section suivante lui est donc entièrement dédiée.

4.3.3 Apprentissage basé sur des instances

4.3.3.1 Principe

L'apprentissage basé sur des instances permet de trouver des instances similaires d'exécution d'applications au sein d'une base de connaissances. La phase d'apprentissage en elle-même consiste simplement à stocker dans la base de connaissances chaque expérience, constituée à la fois de ses entrées (paramètres de l'application, machines utilisées, etc.) et de ses sorties (temps d'exécution, utilisation du processeur et de la mémoire, etc.). La prédiction est ensuite effectuée en cherchant des expériences similaires à celle constituant la requête, en s'appuyant sur quatre composantes distinctes à définir [136] :

- une **distance** permettant de déterminer la similarité de deux expériences, en indiquant leur éloignement relatif,
- le **nombre de voisins les plus proches** considérés pour formuler une prédiction,
- une **fonction de pondération** déterminant l'importance relative de chaque expérience sur la prédiction en fonction de sa distance à la requête,
- un **modèle local** indiquant comment combiner les expériences retenues pour effectuer une prédiction correcte.

4.3.3.2 Terminologie utilisée

On désigne par le terme générique d'*entrée* tous les éléments susceptibles de varier d'une exécution à l'autre, et qui auront une influence sur le temps d'exécution de l'application. Il peut s'agir de paramètres du programme, de la taille des fichiers qu'il aura à traiter, du nombre de données ou de leur complexité, etc.

L'ensemble des éléments appliqués aux programmes pour une seule et même exécution sera appelé *l'entrée du programme*, notée E . Une entrée sera ensuite constituée de plusieurs *valeurs d'entrée*, ou bien attributs ou encore paramètres d'entrée, que nous notons E_v .

Ainsi, l'entrée E d'un programme correspond à un vecteur ayant autant de composantes que ce que l'entrée comporte de valeurs.

Soit N^V le nombre de valeurs constituant les entrées d'un programme donné, on note une entrée de ce programme :

$$E = \{E_v\}_{v \in [1; N^V]} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_{N^V} \end{bmatrix}$$

Chaque valeur d'entrée E_v peut être :

- un **attribut nominal**, tel qu'une chaîne de caractères,
- un **attribut linéaire**, pouvant être *continu* (nombre réel par exemple) ou discret (nombre entier).

Une valeur d'entrée linéaire E_v varie au sein d'un intervalle défini par son type de données. Les bornes de cet intervalle sont notées \min_{E_v} et \max_{E_v} . Ainsi, on a :

$$\forall v \in [1; N^V] \quad E_v \in [\min_{E_v}; \max_{E_v}]$$

4.3.3.3 Notion de distance entre deux expériences

Considérant les expériences comme des points, il est possible de calculer une distance entre elles [137]. Plus particulièrement, les distances sont calculées entre les entrées associées à chaque expérience. Elles permettent ainsi de produire une valeur numérique reflétant la proximité ou l'éloignement des valeurs de deux entrées quelconques du programme. Cette distance est d'autant plus faible que les valeurs constituant les deux entrées sont voisines.

Fonctions de distance

Considérons deux expériences, correspondant aux entrées $E^{(1)}$ et $E^{(2)}$. Plusieurs distances peuvent être appliquées à ces expériences, parmi lesquelles :

- la *distance euclidienne* :

$$D(E^{(1)}, E^{(2)}) = \sqrt{\sum_{v=1}^{N^V} d(E_v^{(1)}, E_v^{(2)})^2} \quad (4.2)$$

- la *distance de Manhattan* :

$$D(E^{(1)}, E^{(2)}) = \sum_{v=1}^{N^V} d(E_v^{(1)}, E_v^{(2)})$$

- la *distance de Minkowski* :

$$D(E^{(1)}, E^{(2)}) = \sqrt[p]{\sum_{v=1}^{N^V} d(E_v^{(1)}, E_v^{(2)})^p}$$

D'autres distances sont également répertoriées dans [137]. Par la suite, nous utiliserons la distance euclidienne, telle qu'utilisée par Smith et Wong [13] et par Senger et al. [86].

Métrique hétérogène de distance

La fonction de distance $d(E_v^{(1)}, E_v^{(2)})$ permet de calculer l'écart entre deux mêmes attributs appartenant aux entrées $E^{(1)}$ et $E^{(2)}$. Ces derniers pouvant être nominaux ou linéaires, il est nécessaire de définir une distance capable de gérer ces deux cas de figures. L'approche décrite dans [137] et reprise dans [13, 86] consiste à combiner une métrique de chevauchement pour les paramètres nominaux, et une distance euclidienne normalisée pour les valeurs linéaires.

On définit ainsi la distance entre deux valeurs d'entrée :

$$d(E_v^{(1)}, E_v^{(2)}) = \begin{cases} 1 & \text{si la valeur } E_v^{(1)} \text{ ou } E_v^{(2)} \text{ est inconnue,} \\ N(E_v^{(1)}, E_v^{(2)}) & \text{si } E_v^{(x)} \text{ est nominale,} \\ L(E_v^{(1)}, E_v^{(2)}) & \text{si } E_v^{(x)} \text{ est linéaire.} \end{cases} \quad (4.3)$$

La fonction de chevauchement est alors définie comme suit :

$$N(E_v^{(1)}, E_v^{(2)}) = \begin{cases} 0 & \text{si la valeur } E_v^{(1)} = E_v^{(2)}, \\ 1 & \text{dans le cas contraire.} \end{cases}$$

Enfin, la fonction de distance euclidienne normalisée est :

$$L(E_v^{(1)}, E_v^{(2)}) = \frac{|E_v^{(1)} - E_v^{(2)}|}{\max_{E_v^{(x)}} - \min_{E_v^{(x)}}}$$

avec :

$$L(E_v^{(1)}, E_v^{(2)}) \in [0; 1]$$

4.3.3.4 Calcul de l'estimation du temps d'exécution

Modèles locaux de combinaison d'expériences

La prédiction du temps d'exécution d'une application est réalisée en combinant les différentes expériences contenues dans la base de connaissances. Il existe pour cela trois modèles de combinaison différents [135] :

- Modèles des plus proches voisins, ou modèles k -NN pour k -nearest neighbor [138] : le temps d'exécution de la requête est calculé grâce à une moyenne non-pondérée des k plus proches expériences, au sens de la distance définie précédemment, contenues dans la base de connaissances.
- Régression polynomiale à pondération locale [138, 139] : une surface est ajustée aux points du voisinage de la requête, typiquement grâce à des modèles locaux linéaires ou quadratiques.
- Moyenne pondérée [138, 13, 86] : la prédiction du temps d'exécution d'une requête est déterminée en effectuant une moyenne pondérée des temps d'exécution des expériences voisines. Le poids de chaque instance d'exécution considérée est une fonction inversée de sa distance à la requête.

Par la suite, nous utiliserons le modèle de la moyenne pondérée.

Modèle de combinaison d'expériences basé sur leur moyenne pondérée

Soit une entrée E^* pour laquelle on souhaite estimer le temps d'exécution d'une application.

Soit $T(E)$ le temps d'exécution de l'application pour une entrée E . On estime le temps d'exécution de l'application pour l'entrée E^* grâce à l'expression suivante [135, 13] :

$$T(E^*) = \frac{\sum_E \left[K(D(E^*, E)) \cdot T(E) \right]}{\sum_E K(D(E^*, E))} \quad (4.4)$$

où K est une fonction permettant de pondérer l'influence des différentes expériences contenues dans la base de connaissances sur le résultat final, et ce en fonction de la distance D séparant deux entrées (équation 4.2). Les expériences proches de la requête seront ainsi mieux pondérées, et leur temps d'exécution aura une influence plus forte que celui des expériences plus éloignées.

Fonction de pondération

La fonction de pondération K , utilisée dans le modèle de combinaison d'expériences basé sur leur moyenne pondérée, permet d'associer un coefficient de pondération à une distance séparant deux entrées. Cette fonction doit posséder les caractéristiques suivantes [135, 136] :

- la valeur maximale de la fonction est atteinte lorsque la distance séparant les deux expériences considérées est nulle,
- la fonction doit être décroissante lorsque la distance augmente,
- la fonction de pondération doit être aussi régulière que possible afin que la fonction d'estimation le soit également ; des discontinuités de la fonction de pondération impliquent des discontinuités dans les prédictions.

L'objectif de cette fonction est ainsi de faire en sorte que les entrées de la base de connaissances dont la distance est proche de l'entrée pour laquelle l'estimation est effectuée soient celles qui auront le plus de poids dans le calcul du résultat. Pour cela, plusieurs fonctions peuvent être choisies [135, 136]. Quelques exemples sont donnés par la figure 4.2.

Dans la suite de la thèse, nous utiliserons la fonction gaussienne :

$$K(d) = e^{-(\frac{d}{k})^2} \quad (4.5)$$

avec k constante permettant de faire varier la largeur de la gaussienne. Cette constante permet de s'adapter à la densité d'expériences présentes dans la région contenant la requête.

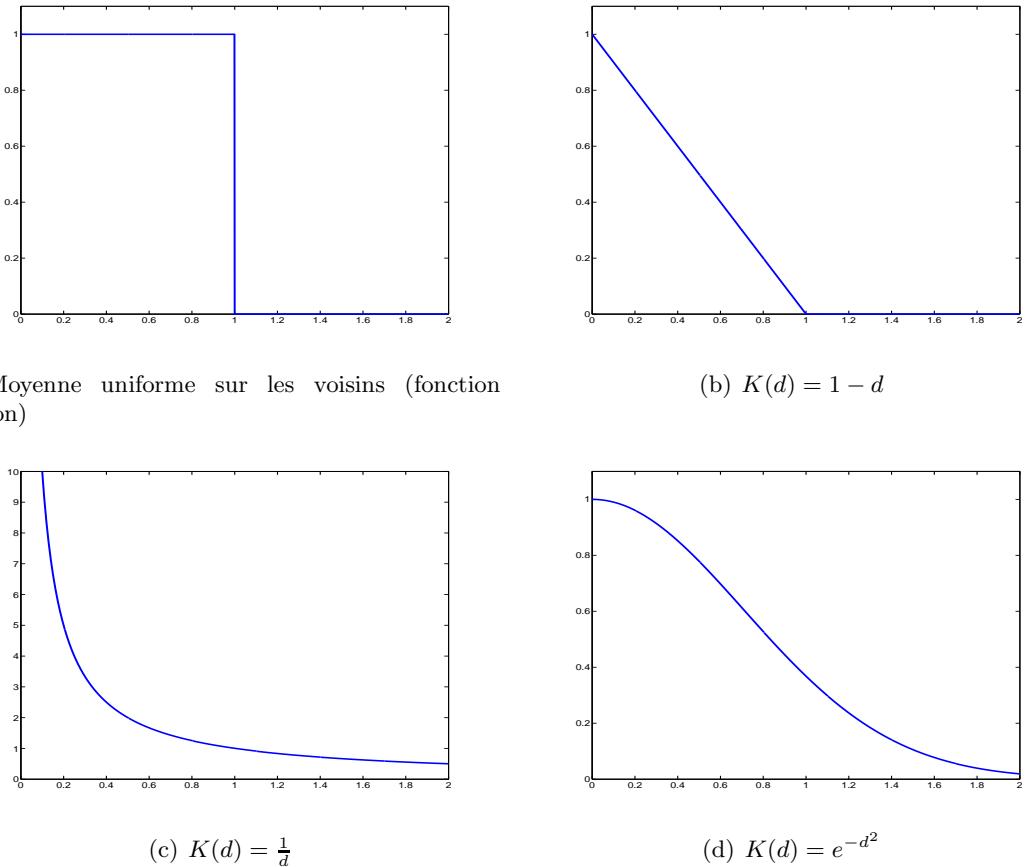


FIG. 4.2 – Fonctions de pondération

4.4 Approche hybride de prédiction de temps d'exécution d'une application

4.4.1 Principe général

Nous choisissons d'utiliser une méthode hybride de prédiction de temps d'exécution qui combine les deux approches présentées. La détermination du temps d'exécution d'une application s'appuiera sur un historique d'exécutions passées, et tiendra compte du profil de l'application pour obtenir une précision accrue.

L'approche hybride que nous avons introduite dans [140] utilise ainsi la représentation de la structure du programme introduite lors de la présentation de l'analyse statique du WCET. Le programme est donc découpé en blocs de base, fragments d'exécution possédant un seul point d'entrée et un seul point de sortie dans le graphe de flot de contrôle. Tout comme dans le cadre de l'analyse IPET, la méthode hybride de prédiction utilise une technique d'énumération implicite des chemins : seul le nombre d'exécutions des blocs de base est pris en compte, quel que soit l'ordre dans lequel ils sont exécutés.

Par ailleurs, l'énumération implicite des chemins permet d'étendre la notion de blocs de base. Nous définirons, dans la prochaine section, la notion de blocs de base étendus, sur laquelle repose l'approche hybride présentée dans le cadre de cette thèse.

Ainsi, la prédiction du temps d'exécution d'une application débute par l'étude de son profil. Une expression similaire à l'équation (4.1) est obtenue, pour laquelle le temps d'exécution des blocs de base, ainsi que leur nombre d'exécutions (nombre d'occurrences dans le chemin d'exécution) doivent être déterminés. Si la première de ces valeurs peut être déterminée à l'aide d'un simple système d'équations comme nous le montrerons ultérieurement, la seconde est, en revanche, estimée à l'aide d'une méthode d'apprentissage basé sur des instances.

4.4.2 Définitions et hypothèses

Un programme est une suite d'instructions généralement regroupées en sous-programmes, également appelés fonctions. Le nombre de passages dans chaque fonction d'un programme peut varier d'une exécution à l'autre en fonction des entrées qui sont appliquées au programme.

Toutes les instructions appartenant à un seul et même bloc de base sont exécutées un même nombre de fois, qui correspond au nombre d'occurrences de ce bloc dans le chemin d'exécution. Cette valeur peut dépendre des entrées du programme.

Nous introduisons les définitions suivantes :

Définition Un *bloc de base étendu* est un ensemble de blocs de base appartenant à une seule et même fonction, et qui sont exécutés le même nombre de fois, quelles que soient les entrées appliquées au programme. Ce nombre d'exécutions peut cependant varier d'une exécution à l'autre selon les entrées appliquées au programme. Ainsi, les blocs qui sont toujours exécutés le même nombre de fois ne sont pas individuellement distingués. On les considère au contraire comme une seule et même entité. Ceci permet de simplifier la complexité du problème, puisque le nombre de blocs considérés sera ainsi plus faible, sans pour autant perdre de la précision.

Corollaire Un bloc de base étant constitué d'un ensemble d'instructions possédant un point d'entrée et un point de sortie et ne contenant pas d'instruction de branchement, un bloc de base étendu est un ensemble d'instructions appartenant à une seule et même fonction et qui sont exécutées le même nombre de fois.

Définition Un bloc de base étendu est dit maximal s'il n'est inclus dans aucun autre bloc de base étendu.

Notons que, par abus de langage et par souci de simplification des écritures, nous assimilerons par la suite le terme de "bloc de base" à celui de "bloc de base étendu maximal". Ainsi, toutes les équations permettant de définir l'approche hybride de prédiction de temps d'exécution prennent en compte les blocs de base étendus maximaux des programmes, même si cela n'est pas précisé de manière explicite.

4.4.2.1 Hypothèses sur le temps d'exécution des blocs de base

Le temps d'exécution d'un bloc de base sera considéré comme constant, c'est-à-dire indépendant du contexte d'exécution du programme. Cette hypothèse exclut la prise en compte des mécanismes présents sur les processeurs modernes, tels que les caches.

En outre, le temps d'exécution des blocs de base est indépendant des entrées appliquées au programme, et ce en raison de l'absence d'instruction de branchement (notamment conditionnel) à l'intérieur des blocs de base.

Ces hypothèses sont également valables pour les blocs de base étendus maximaux du programme.

4.4.2.2 Hypothèses sur le temps d'exécution des fonctions

Contrairement aux blocs de base, le temps d'exécution d'une fonction n'est pas nécessairement constant. Il peut dépendre des entrées du programme, et sera influencé par le nombre d'exécutions de chacun des blocs de base constituant la fonction. Ainsi, les entrées du programme peuvent influer sur le nombre de passage à l'intérieur des blocs de base de la fonction, faisant ainsi varier son temps d'exécution, mais le temps d'exécution de chacun des blocs de base sera toujours constant.

Notons que si une fonction fait appel à une autre fonction, le temps d'exécution de la fonction appelée ne sera pas compté dans le temps d'exécution de la fonction appelante.

4.4.3 Modèle mathématique

L'équation suivante peut être utilisée pour évaluer le temps d'exécution d'une application. A noter que celui-ci dépend des entrées du programme.

$$T_{App}(E) = \sum_{f \in \mathbb{F}} N_f^F(E) \cdot T_f^F(E) \quad (4.6)$$

où :

- \mathbb{F} est l'ensemble des fonctions du programme,
- $N_f^F(E)$ est le nombre d'exécutions de la fonction f , dépendant des entrées,
- $T_f^F(E)$ est le temps d'exécution de la fonction f , dépendant également des entrées.

Cette équation montre comment calculer le temps d'exécution d'une application découpée en fonctions. Cependant, le temps d'exécution d'une fonction n'est pas nécessairement constant. Il dépend du nombre d'exécutions et du temps d'exécution des blocs de base qu'elle contient.

Ainsi, le temps d'exécution d'une fonction peut s'exprimer :

$$\forall f \in \mathbb{F} \quad T_f^F(E) = \sum_{b \in \mathbb{BB}_f^F} N_b^{BB}(E) \cdot T_b^{BB} \quad (4.7)$$

où :

- \mathbb{BB}_f^F est l'ensemble des blocs de base étendus maximaux de la fonction f ,
- $N_b^{BB}(E)$ est le nombre d'exécutions du bloc de base b , dépendant des entrées,
- T_b^{BB} est le temps d'exécution du bloc de base b , considéré comme constant.

Le temps d'exécution de l'application devient alors (équations 4.6 et 4.7) :

$$T_{App}(E) = \sum_{f \in \mathbb{F}} \left[N_f^F(E) \cdot \sum_{b \in \mathbb{BB}_f^F} N_b^{BB}(E) \cdot T_b^{BB} \right] \quad (4.8)$$

De manière plus simple, l'équation sur laquelle s'appuie la prédiction du temps d'exécution d'une application basée sur son profil est la suivante :

$$T_{App}(E) = \sum_{b \in \mathbb{BB}} N_b^{BB}(E) \cdot T_b^{BB} \quad (4.9)$$

où BB est l'ensemble des blocs de base étendus maximaux du programme :

$$\mathbb{BB} = \bigcup_{f \in \mathbb{F}} \mathbb{BB}_f^F$$

4.4.4 Mise en œuvre du modèle

Une prédiction basée sur le modèle décrit par l'équation 4.9 nécessite de pouvoir estimer à la fois le nombre d'exécutions de chacun des blocs de base d'un programme, ainsi que leur durée d'exécution.

On distingue ainsi deux catégories d'informations à déterminer :

- **informations temporelles** : ces informations permettent d'évaluer le temps d'exécution de chacun des blocs de base d'un programme. Le chapitre 5 montre comment obtenir ces temps en résolvant un système d'équations linéaires.
- **informations comportementales** : ces informations permettent d'évaluer le comportement d'un programme en fonction de ses entrées. Une approche par apprentissage basé sur des instances sera ainsi utilisée pour déterminer de telles informations, c'est-à-dire pour estimer le nombre d'occurrences de chacun des blocs de base d'un programme au sein du chemin d'exécution suivi. Le chapitre 6 présente ceci en détail.

L'étude que nous mènerons par la suite s'appuie sur deux logiciels standards de la suite GNU :

- **Gprof**, un *profiler* permettant de connaître, après l'exécution d'une application, sa durée d'exécution totale, ainsi que le temps d'exécution de chacune de ses fonctions et le nombre de fois où elles ont été appelées. Le principe de fonctionnement de cet outil est détaillé dans l'annexe D.
- **Gcov**, un logiciel permettant d'effectuer un test de couverture d'un programme. Après l'exécution du programme, *gcov* donne, pour chaque ligne de son code, le nombre de fois où elle a été exécutée.

Ces deux outils appartenant à la suite GNU, ils ont été retenus pour des raisons de stabilité et de pérennité. *Gprof* sera ainsi utilisé pour déterminer les temps d'exécution T_f^F de chacune des fonctions du programme. Quant à *gcov*, il permettra le découpage du programme en blocs de base étendus maximaux, et donnera le nombre d'occurrences N_b^{BB} de ces blocs au sein du chemin d'exécution suivi.

4.5 Conclusion

Dans ce chapitre, nous avons présenté l'approche hybride de prédition du temps d'exécution d'une application. Cette approche innove par la mise en relation des travaux menés dans le cadre de la détermination du WCET d'applications temps réel, utilisés pour l'ordonnancement temps réel, avec ceux qui concernent la prédition basée sur un historique d'exécutions passées, généralement utilisés pour l'amélioration des performances des ordonneurs de grilles ou de clusters.

La prédition du temps d'exécution basé sur un historique d'exécutions passées suppose que des applications lancées dans des contextes proches produiront des temps d'exécution proches. La méthode d'apprentissage basé sur des instances montre comment estimer un temps d'exécution à partir d'une base de connaissances, en définissant une fonction de distance permettant d'évaluer la différence entre deux entrées du programme, une fonction de pondération déterminant l'importance relative de chaque expérience contenue dans la base de connaissances sur le résultat final, et enfin un modèle local indiquant comment combiner les expériences retenues pour effectuer une prédition correcte.

Les techniques les plus répandues de détermination du WCET d'applications temps réel s'appuient sur le profil de ces applications, c'est-à-dire sur leur découpage en blocs de base. Les majorants des temps d'exécution de ces blocs sont déterminés à l'aide de modèles d'architectures matérielles, ou bien déterminés de manière dynamique, à partir de mesures directes. Le WCET est alors calculé en déterminant le nombre d'occurrences de ces blocs dans le chemin d'exécution qui mène au pire temps d'exécution.

L'approche hybride se base également sur le profil des applications. Ces dernières sont découpées en blocs de base, puis les temps d'exécution de ces derniers sont déterminés à l'aide de mesures directes. Cependant, l'outil *gprof* utilisé n'est pas conçu pour donner de telles informations. Il ne peut en effet donner que le temps d'exécution global des fonctions du programme, sans descendre au niveau des blocs. Par conséquent, le chapitre 5 montre comment déterminer ces informations, qualifiées de temporelles.

Une fois le temps d'exécution des blocs de base connu, il est possible de déterminer une estimation du temps d'exécution du programme pour un jeu d'entrées donné en déterminant le nombre d'occurrences de ces blocs dans le chemin d'exécution pour le jeu d'entrée considéré. Ceci est réalisé en utilisant une technique d'apprentissage basé sur des instances, tel que le montre le chapitre 6.

Chapitre 5

Détermination du temps d'exécution des blocs de base d'un programme

L'objectif de ce chapitre est de montrer comment déterminer les informations temporelles d'un programme, c'est-à-dire le temps d'exécution des blocs de base étendus maximaux qui le constituent. La méthode présentée s'appuie sur l'utilisation des outils de profiling standards offerts par la suite GNU : *gprof* et *gcov*.

Nous montrons dans une première partie que le temps d'exécution des blocs de base est la solution d'un système d'équations linéaires, à m équations et n inconnues. Dans une deuxième partie, les résultats obtenus à l'aide de méthodes de résolution standards seront abordés. Nous montrerons l'inefficacité de telles méthodes pour notre cas. Enfin, dans une troisième partie, nous définirons une méthode de résolution du système par itérations successives, et montrerons comment rendre celle-ci exploitable.

5.1 Introduction

5.1.1 Objectifs

Ce chapitre décrit une méthode d'obtention des informations temporelles d'un programme. Il s'agit ici de déterminer le temps d'exécution des blocs de base étendus maximaux qui le composent. Nous supposons que ceux-ci sont constants, et donc indépendants des entrées.

D'après l'équation (4.9), le temps d'exécution d'un programme s'exprime de la manière suivante :

$$T_{App}(E) = \sum_{b \in \mathbb{B}} N_b^{BB}(E) \cdot T_b^{BB}$$

L'objectif de ce chapitre est ainsi de proposer une méthode permettant de calculer le temps d'exécution T_b^{BB} de chacun des blocs de base b du programme.

5.1.2 Principe

Déterminer le temps d'exécution T_b^{BB} de chacun des blocs de base b du programme à l'aide de l'équation précédente nécessite de connaître les valeurs du temps d'exécution global $T_{App}(E)$ du programme ainsi que du nombre d'exécutions $N_b^{BB}(E)$ de chacun des blocs de base.

Ces valeurs peuvent être obtenues grâce aux deux outils que nous utilisons : *gprof* pour connaître le temps d'exécution du programme, et *gcov* pour connaître le nombre d'exécutions des blocs de base. Notons que l'outil *gcov* ne permet pas de connaître directement le nombre d'exécutions des blocs de base. Il s'agit en fait d'un utilitaire dont le rôle est de réaliser des tests de couverture. Pour cela, il indique le nombre de fois où chaque instruction du programme est exécutée. Il est ensuite possible de regrouper de manière automatique les instructions en blocs de base étendus maximaux, et ainsi d'en déduire leur nombre d'exécutions.

On obtient donc une équation possédant autant d'inconnues que ce que le programme comporte de blocs de base étendus maximaux (c'est-à-dire $Card(\mathbb{B}\mathbb{B})$ inconnues). Afin de déterminer l'ensemble des T_b^{BB} , il est nécessaire de connaître plusieurs jeux de valeurs de $T_{App}(E)$ avec les $N_b^{BB}(E)$ correspondants, dans le but de se ramener à la résolution d'un système d'équations linéaires, dont le nombre d'inconnues est égal au nombre de blocs de base constituant le programme.

En exécutant plusieurs fois le programme pour des entrées différentes, on obtient ainsi un système d'équations linéaires dont le nombre d'équations est égal au nombre d'exécutions du programme effectuées. Le calcul des temps d'exécution des blocs de base du programme revient à résoudre un tel système.

5.2 Exemple traité et faisabilité

Dans ce chapitre, nous utiliserons un exemple de programme simple, entièrement déterministe afin de valider notre méthode. Cet exemple, décrit dans l'annexe E, consiste en un programme permettant d'élèver une matrice carrée de dimension donnée à une puissance donnée.

Le programme `matrixPower` utilisé comporte deux entrées qui influencent son temps d'exécution :

- la dimension de la matrice,
- la puissance à laquelle la matrice doit être élevée.

5.2.1 Evolution du temps d'exécution du programme en fonction des entrées

Le code du programme `matrixPower` donné en annexe montre que les deux entrées (variables `dimensions` et `power`) sont susceptibles d'agir sur le temps d'exécution du programme, puisque toutes deux ont un impact sur le nombre d'itérations des boucles présentes dans le code source.

La figure 5.1 montre l'évolution du temps d'exécution du programme en fonction de ses entrées. Le matériel pris en compte pour cette étude concerne des machines *Dell PowerEdge 1950* dont le processeur est un *Intel Xeon 5148 LV* double-cœur cadencé à 2.33 Ghz, et possédant 4 Go de mémoire vive.

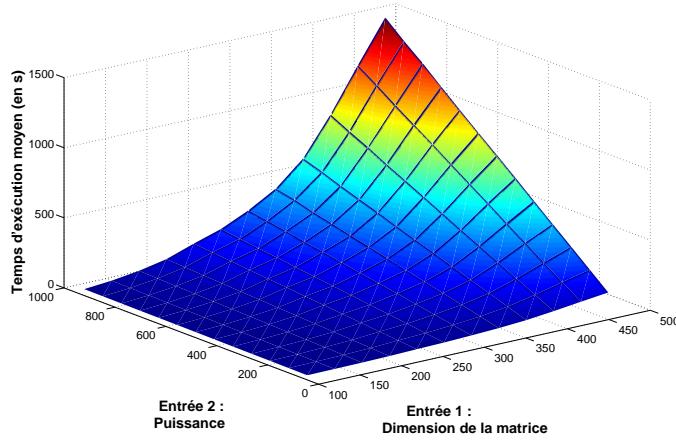


FIG. 5.1 – Temps d'exécution du programme `matrixPower` en fonction de ses entrées (vue 3D)

Cette courbe confirme l'effet des entrées sur le temps d'exécution du programme. Celui-ci augmente bien lorsque des matrices de dimension élevée sont traitées, ou bien lorsque la puissance à laquelle la matrice doit être élevée devient plus importante.

Il est également possible de remarquer que l'augmentation du temps d'exécution du programme est absolument continue, la courbe étant exempte d'irrégularité. Ceci permet de s'assurer que notre prédition pourra être effectuée dans de bonnes conditions, et qu'aucune surprise n'est attendue sur ce point.

Nous choisissons de représenter également l'évolution des temps d'exécution en fonction des entrées sous la forme d'un graphique à deux dimensions. En effet, ce type de représentation facilitera dans les sections suivantes la comparaison entre le temps d'exécution estimé et celui réellement obtenu. Dans ce cas, l'ensemble des jeux d'entrées sont regroupés en abscisse. Les variations de l'entrée `puissance` sont imbriquées à l'intérieur de celles de l'entrée `dimensions`. La figure 5.2 montre ce graphique.

5.2.2 Cohérence des temps obtenus et reproductibilité

Un aspect important qu'il convient de vérifier concerne la cohérence des temps obtenus, du point de vue de la reproductibilité des expériences. En effet, il est indispensable que deux exécutions distinctes du programme pour des entrées identiques produisent des temps d'exécution similaires.

Pour cela, nous choisissons d'exécuter le programme 100 fois pour chacun des jeux d'entrées testés. La courbe traçant la moyenne des temps obtenus sur chacune des exécutions est semblable à la figure 5.1. Nous allons donc nous intéresser plus particulièrement à la courbe des écarts-types relatifs à la moyenne obtenus (figure 5.3).

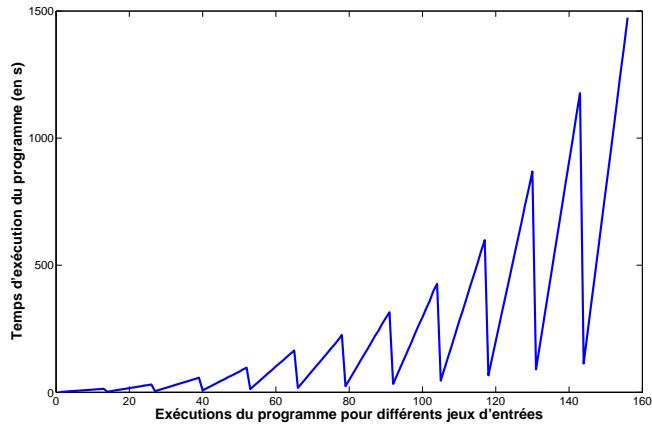


FIG. 5.2 – Temps d’exécution du programme `matrixPower` en fonction de ses entrées (vue 2D)

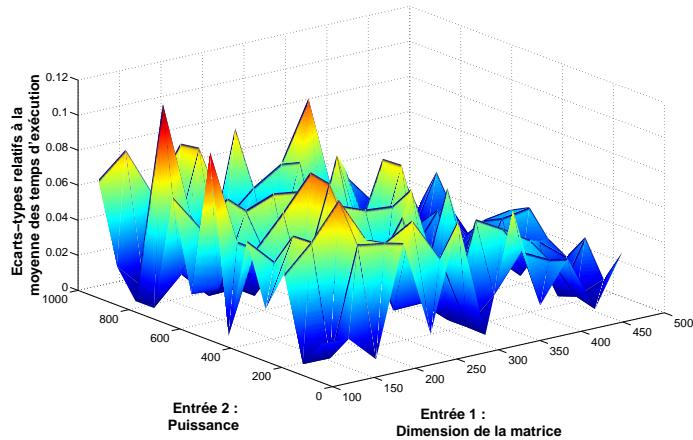


FIG. 5.3 – Ecart-type relatif à la moyenne des temps d’exécution du programme `matrixPower` en fonction de ses entrées

Il est ainsi possible d’observer que l’écart-type du temps d’exécution varie entre 0 et 10%, la moyenne de cet écart-type relatif étant de 3,45%. De plus cet écart-type a tendance à être plus élevé pour des exécutions courtes du programme. Cette tendance n’est pas gênante, puisqu’il est normal d’obtenir des erreurs d’autant plus importantes que les temps d’exécution obtenus sont faibles.

La faible valeur d’écart-type obtenue ainsi est rassurante quant à la reproductibilité des expériences. Ici aussi, cela permet de s’assurer que la prédiction se fera dans de bonnes conditions.

5.2.3 Influence de l’utilisation de `gprof` et `gcov` sur le temps d’exécution du programme

Le dernier aspect étudié concerne l’influence de l’utilisation de `gprof` et `gcov` sur le temps d’exécution du programme. Le propos de cette section est ainsi de montrer que l’utilisation de `gprof` et de `gcov` n’est pas rédhibitoire. En effet, exécuter une application en effectuant

au même temps du profiling ainsi que du test de couverture consomme des ressources, et de fait ralentit l'application. Si la différence de temps d'exécution entre une exécution normale et une exécution avec profiling est trop importante, il est évident que l'utilisation des deux outils est compromise.

Même si *gcov* n'est pas un profiler, nous assimilerons le terme d'"exécution sans profiling" avec une exécution normale, c'est-à-dire sans utiliser *gprof* ni *gcov*, et le terme d'"exécution avec profiling" avec une exécution instrumentée par les deux outils.

Cette étude sera réalisée sur différentes machines, possédant chacune sa propre architecture, afin d'avoir une vue d'ensemble des répercussions induites par l'utilisation de *gprof* et de *gcov*. Nous détaillerons les résultats pour un seul type d'architecture, les remarques étant identiques pour tous. Un tableau récapitulatif montrera l'ensemble des résultats obtenus.

La première machine pour laquelle les répercussions de l'utilisation de *gprof* et de *gcov* sont étudiées possède un processeur de type *Intel Xeon 5148 LV* double cœur cadencé à 2.33 Ghz.

La figure 5.4 montre comment varient les temps d'exécution de l'application avec et sans profiling en fonction des entrées qui lui sont appliquées.

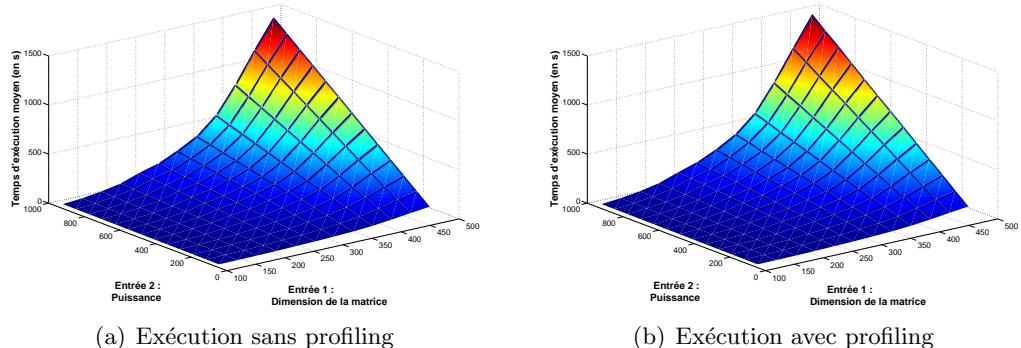


FIG. 5.4 – Temps d'exécution de l'application en fonction de ses entrées pour un Intel Xeon 5148 LV

Le profil de variation du temps d'exécution de l'application en fonction de ses entrées est identique dans le cas où *gprof* et *gcov* sont utilisés que dans le cas où aucun profiling n'est réalisé. Cependant, comme l'on peut s'y attendre, les exécutions avec profiling prennent légèrement plus de temps. La figure 5.5 met en évidence cette différence.

Cette figure montre que, quelle que soit la durée de l'application (à l'échelle de la seconde ou bien de la dizaine de minutes), la différence relative entre le temps d'exécution avec profiling et celui sans profiling est du même ordre de grandeur, même si elle est moindre pour des temps d'exécution plus importants. En moyenne, cette différence est de 4,09%.

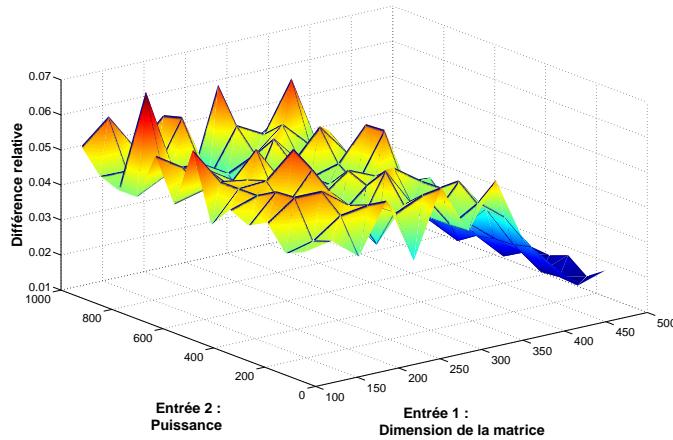


FIG. 5.5 – Différence relative entre les temps d'exécution avec et sans profiling pour un Intel Xeon 5148 LV

Notons que la décroissance observée montre que le profiling du programme entraîne un temps d'exécution d'autant plus important en rapport de la durée d'exécution que celle-ci est faible. Ceci permet de supposer qu'une phase d'initialisation du profiling entre en jeu, phase dont la durée est indépendante du temps d'exécution du programme.

Cette même étude a été réalisée sur différents types de processeurs, afin de corroborer les résultats précédents. Le tableau 5.1 regroupe les différences relatives entre les temps d'exécution avec et sans profiling, pour chacune des architectures testées. Les courbes représentant ces différences relatives de temps d'exécution sont tout-à-fait similaires au graphique de la figure 5.5.

	<i>Intel Xeon 5148 LV 2.33 GHz</i>	<i>Athlon XP-M 1800+</i>	<i>Pentium IV 2.4 GHz</i>	<i>UltraSparc IIe 650 MHz</i>
Différence relative	4,09%	11,3%	29,3%	5,7%

TAB. 5.1 – Différences relatives entre les temps d'exécution obtenus avec et sans profiling pour diverses architectures

Cette étude montre que la différence de temps observée, entre une exécution utilisant *gprof* et *gcov* et une ne les utilisant pas, dépend fortement du type d'architecture considéré. Ainsi, les processeurs *Intel Xeon* et *UltraSparc IIe* montrent peu de différences, ce qui rend tout-à-fait possible les exécutions profilées en mode production. Dans le cas de l'*Athlon XP-M*, même s'il montre une différence relative de temps plus importante, celle-ci reste raisonnable. En revanche, le *Pentium IV* affiche une différence de temps rédhibitoire, qui peut empêcher l'utilisation des outils de profiling en mode production.

5.3 Résolution du système d'équations à l'aide d'outils et de méthodes standards

5.3.1 Introduction

L'exécution du programme pour différentes valeurs d'entrées permet d'obtenir un système d'équations linéaires dont le nombre d'équations est égal au nombre d'exécutions du programme effectuées. Les inconnues de ce système correspondent aux temps d'exécution des blocs de base du programme, informations temporelles nécessaires à l'approche de prédiction hybride proposée dans cette thèse.

Soit X le nombre d'exécutions distinctes du programme réalisées, avec pour chacune d'elles des jeux d'entrées différents. On note $E^{(e)}$, avec $e \in [1, X]$, les entrées ayant servi à ces exécutions. A l'issue de ces dernières, un système d'équations est donc obtenu :

$$\forall e \in [1, X] \quad T_{App}(E^{(e)}) = \sum_{b \in \mathbb{BB}} N_b^{BB}(E^{(e)}) \cdot T_b^{BB} \quad (5.1)$$

Ce système est un système d'équations linéaires à X équations et à $Card(\mathbb{BB})$ inconnues. En outre, nous supposons que le système possède suffisamment d'équations pour être résolu ($X \geq Card(\mathbb{BB})$) ; le système est donc potentiellement surdéterminé. Nous utiliserons le logiciel *Matlab* afin de le résoudre et plus particulièrement deux méthodes de résolution différentes, la première s'appuyant sur un solveur de problèmes relevant de la programmation linéaire (commande `linprog` de Matlab), et la seconde reposant sur un solveur de systèmes d'équations non-linéaires (commande `fsoolve`).

Quelle que soit la méthode utilisée, le processus mis en jeu est un processus itératif d'optimisation, pour lequel il est nécessaire de disposer d'une approximation de la solution servant de solution de départ pour les itérations. La section suivante montre comment obtenir une telle valeur.

5.3.2 Calcul d'une estimation initiale de la solution

Afin de déterminer une approximation initiale de la solution du système d'équations cherchée, toutes les instructions du code source du programme seront supposées posséder le même temps d'exécution. Selon cette hypothèse, le temps d'exécution des blocs de base étendus maximaux ne dépendra que du nombre de lignes qui les composent.

Cette hypothèse simplificatrice est bien entendu fausse dans la réalité, mais elle permet une première estimation de la solution qui pourra ensuite être améliorée par les fonctions `linprog` et `fsoolve` de *Matlab*.

On introduit la notation suivante :

- $N_{b,b \in \mathbb{BB}}^L(E)$: nombre total d'exécutions de lignes pour le bloc de base b (cumul du nombre d'exécutions de chacune de ces lignes au cours de l'exécution du programme). Ce nombre dépend des entrées du programme.

Nous rappelons également les deux définitions suivantes :

- $T_{b,b \in \mathbb{B}}^{BB}$: temps d'exécution unitaire du bloc de base b . Il s'agit de l'entité que l'on désire estimer ici.
- $T_{App}(E)$: temps d'exécution du programme obtenu grâce à l'utilisation du profiler *gprof*.

L'estimation initiale du temps d'exécution d'un bloc de base peut ainsi s'exprimer de la manière suivante :

$$\forall b \in \mathbb{B} \quad T_b^{BB} = T_{App}(E) \cdot \frac{N_b^L(E)}{\sum_{i \in \mathbb{B}} N_i^L(E)} \quad (5.2)$$

Enfin, en considérant les X exécutions successives du programme, on peut estimer plus finement le temps d'exécution des blocs de base en effectuant une moyenne des résultats obtenus par l'équation 5.2 :

$$\forall b \in \mathbb{B} \quad T_b^{BB} = \frac{1}{X} \cdot \sum_{e=1}^X \left[T_{App}(E^{(e)}) \cdot \frac{N_b^L(E^{(e)})}{\sum_{i \in \mathbb{B}} N_i^L(E^{(e)})} \right] \quad (5.3)$$

5.3.3 Résolution du problème sous la forme d'un système linéaire

5.3.3.1 Présentation

Le système précédent est un système d'équations linéaires classique pouvant s'exprimer sous la forme matricielle suivante :

$$A \cdot x = b \quad (5.4)$$

où :

- A est la matrice du système,
- x est le vecteur inconnu,
- b est le vecteur second membre connu.

La correspondance entre les équations 5.1 et 5.4 fait apparaître les relations suivantes :

$$\bullet A = \begin{bmatrix} N_{b_1}^{BB}(E^{(1)}) & N_{b_2}^{BB}(E^{(1)}) & \dots & N_{b_{Card(\mathbb{B})}}^{BB}(E^{(1)}) \\ N_{b_1}^{BB}(E^{(2)}) & N_{b_2}^{BB}(E^{(2)}) & \dots & N_{b_{Card(\mathbb{B})}}^{BB}(E^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ N_{b_1}^{BB}(E^{(X)}) & N_{b_2}^{BB}(E^{(X)}) & \dots & N_{b_{Card(\mathbb{B})}}^{BB}(E^{(X)}) \end{bmatrix}$$

$$\bullet x = \begin{bmatrix} T_{b_1}^{BB} \\ T_{b_2}^{BB} \\ \vdots \\ T_{b_{Card(\mathbb{B})}}^{BB} \end{bmatrix}$$

$$\bullet \quad b = \begin{bmatrix} T_{App}(E^{(1)}) \\ T_{App}(E^{(2)}) \\ \vdots \\ T_{App}(E^{(X)}) \end{bmatrix}$$

Le vecteur x recherché contient bien le temps d'exécution de chacun des blocs de base étendus maximaux du programme.

Afin de résoudre ce système, nous utilisons la fonction `linprog` de Matlab. Celle-ci traite des problèmes de programmation linéaire, et permet de minimiser une équation linéaire, sous contraintes d'égalités et d'inégalités, et en exprimant des contraintes sur les solutions à obtenir. Ainsi, cette fonction de Matlab permettra la résolution du système d'équations sous sa forme linéaire.

La fonction minimisée ici est la fonction $A \cdot x - b$, le but étant de s'approcher le plus possible de 0. Nous posons également des contraintes d'égalité permettant de spécifier que chaque composante du vecteur x calculé doit être positive. Ceci permet de garantir la cohérence des résultats cherchés, ces derniers représentant les temps d'exécution des blocs de base étendus maximaux du programme.

5.3.3.2 Résultats obtenus

Le programme `matrixMultiplication` a été exécuté 156 fois, avec des jeux d'entrées différents représentant toutes les combinaisons possibles entre 12 valeurs différentes de la variable `dimensions` et 13 valeurs de la variable `power`. Le temps d'exécution obtenu pour chacune de ces exécutions a été vu précédemment (figure 5.2).

Parmi ces 156 exécutions, 46 ont été choisies aléatoirement pour être utilisées dans le but de résoudre le système. On a donc, dans ce cas, un système d'équations linéaires $A \cdot x = b$ mettant en jeu $X = 46$ équations distinctes. Nous nous servons alors de la fonction `linprog` pour résoudre ce système.

L'exactitude de la solution obtenue sera évaluée en comparant le produit $A \cdot x$, constituant le temps d'exécution estimé, avec le vecteur b représentant le temps d'exécution réel de l'application. Notons que, pour cette phase, l'ensemble des exécutions sera considéré (c'est-à-dire 156 exécutions), afin d'évaluer également l'exactitude de la solution pour des exécutions non prises en compte dans le système d'équations.

La figure 5.6 met en parallèle les temps d'exécution réel et estimé du programme.

Sur cette figure, on constate que le temps estimé possède le même profil d'évolution que le temps d'exécution réel de l'application (qui correspond à la figure 5.2). Cependant, l'ordre de grandeur n'est pas du tout respecté. En effet, le temps d'exécution réel de l'application s'étale entre 1 seconde et 1500 secondes environ, tandis que le temps estimé atteint des pics de $1\ 700\ 000$ secondes. En moyenne, le temps estimé diffère du temps d'exécution réel de $1,50 \times 10^5\%$. Il est à noter que l'axe des ordonnées de ce graphique est représenté avec une échelle logarithmique.

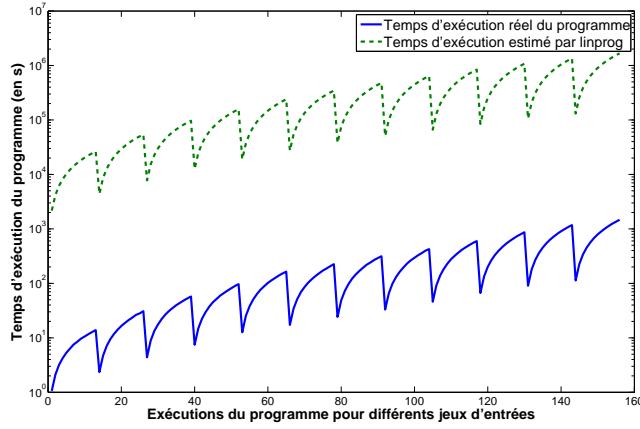


FIG. 5.6 – Comparaison entre le temps d'exécution estimé par `linprog` et le temps d'exécution réel du programme

Ainsi, même si une relation de proportionnalité est respectée entre les temps d'exécution estimés et ceux qui ont été mesurés, les résultats donnés par cette méthode ne sont absolument pas convenables.

5.3.4 Résolution du problème sous la forme d'un système non-linéaire

5.3.4.1 Présentation

Bien que de nature linéaire, nous allons maintenant tenter de résoudre le système d'équations à l'aide d'un solveur de systèmes d'équations non-linéaires. Nous utilisons, pour cela, la fonction `fsoolve` de Matlab qui, partant d'une estimation initiale de la solution, cherche à améliorer cette solution dans le but de minimiser un ensemble de fonctions non-linéaires.

Afin d'obtenir des valeurs positives pour le temps d'exécution de chacun des blocs de base étendus maximaux du programme, ce système sera transformé en un système d'équations non-linéaires, contenant d'une part X équations exprimant le système à résoudre, auxquelles s'ajoutent ensuite un ensemble d'équations permettant d'exprimer le souhait d'obtenir des solutions positives.

Le système prend alors la forme suivante :

$$\forall i \in [1, X + \text{Card}(\mathbb{B}\mathbb{B})] \quad P^{(i)}(x) = 0 \quad (5.5)$$

où :

- $P_{i \in [1, X + \text{Card}(\mathbb{B}\mathbb{B})]}^{(i)}$ est un ensemble de fonctions associant un réel à un vecteur,
- x est le vecteur inconnu.

Le rôle d'un solveur tel que *Matlab* sera alors de trouver une solution pour x minimisant l'ensemble des fonctions $P_{i \in [1, X + \text{Card}(\mathbb{B}\mathbb{B})]}^{(i)}(x)$.

Les différents éléments du système d'équations non-linéaires peuvent s'exprimer de la manière suivante :

- Solution cherchée :

$$x = \begin{bmatrix} T_{b_1}^{BB} \\ T_{b_2}^{BB} \\ \vdots \\ T_{b_{Card(\mathbb{B}\mathbb{B})}}^{BB} \end{bmatrix}$$

- Equations traduisant le système à résoudre :

$$\forall i \in [1, X] \quad P^{(i)}(x) = \sum_{b \in \mathbb{B}\mathbb{B}} \left[N_b^{BB}(E^{(i)}) \cdot T_b^{BB} \right] - T_{App}(E^{(i)})$$

- Equations contrignant l'optimisation à la recherche de solutions positives :

$$\forall i \in [X + 1, X + Card(\mathbb{B}\mathbb{B})] \quad P^{(i)}(x) = |T_{b_{(i-X)}}^{BB}| - T_{b_{(i-X)}}^{BB}$$

5.3.4.2 Résultats obtenus

Le contexte d'obtention des résultats est strictement identique à celui utilisé pour évaluer l'exactitude de la résolution réalisée par la fonction `linprog`. Nous utilisons ainsi les 46 mêmes exécutions pour constituer le système d'équations à résoudre. La fonction `fsolve` est alors utilisée pour chercher une solution x à ce système.

La figure 5.7 représente les temps d'exécution réels et estimé du programme pour les 156 jeux d'entrées considérés.

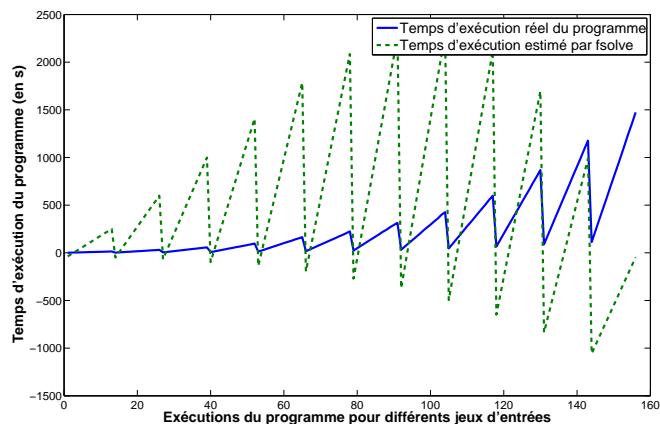


FIG. 5.7 – Comparaison entre le temps d'exécution estimé par `fsolve` et le temps d'exécution réel du programme

Si l'ordre de grandeur du temps d'exécution estimé par `fsolve` est plus raisonnable que celui obtenu à l'aide de la fonction `linprog`, celui-ci reste tout-de-même très différent du temps d'exécution mesuré. En outre, l'évolution de la courbe du temps estimé ne correspond pas du tout à la réalité. Plus grave, certains résultats sont négatifs, ce qui est

totalement incohérent puisqu'ils représentent une durée. La différence moyenne entre les temps d'exécution observés et ceux estimés est de $7,06 \times 10^2\%$.

5.3.5 Conclusion

Les méthodes standards de résolution du système testées dans cette section ne donnent pas de résultats satisfaisants. En exprimant le problème sous une forme linéaire ou non, les résultats obtenus sont très loin de la réalité.

Ceci est dû à la nature du système d'équations devant être résolu. En effet, il s'agit d'un système mal conditionné, pour lequel des dépendances entre lignes et colonnes peuvent être observées. Si les dépendances entre lignes peuvent être évitées au maximum en choisissant des jeux d'entrées suffisamment indépendants entre eux, les dépendances entre colonnes sont quant à elles inévitables.

En effet, ces dernières traduisent la structure même du programme, puisqu'elles correspondent à des dépendances entre les nombres d'exécutions des blocs de base. Par exemple, les boucles imbriquées impliquent de telles dépendances. L'annexe E montre, pour une des fonctions du programme `matrixPower`, les dépendances mises en jeu.

Les sections suivantes décrivent l'approche imaginée dans le cadre de cette thèse pour résoudre un tel système mal conditionné.

5.4 Résolution itérative du système d'équations

5.4.1 Introduction

5.4.1.1 Positionnement du problème

Dans cette section, nous présentons une méthode de résolution par itérations successives du système d'équations :

$$\forall e \in [1, X] \quad T_{App}(E^{(e)}) = \sum_{b \in \mathbb{B}\mathbb{B}} N_b^{BB}(E^{(e)}) \cdot T_b^{BB}$$

Il s'agit ainsi de transformer la résolution d'un système mal conditionné et potentiellement surdéterminé ($X \geq Card(\mathbb{B}\mathbb{B})$) en une suite d'itérations ne mettant en jeu que des calculs matriciels simples, et qui convergent vers la solution souhaitée.

Notons le système à résoudre :

$$A \cdot x = b$$

Ce système est considéré comme étant mal conditionné, c'est-à-dire que le conditionnement de la matrice A , noté $\kappa(A)$, est très grand. Ainsi, une petite variation des valeurs contenues dans le vecteur b introduira une variation importante de la solution x obtenue. L'objectif de cette section est de définir une méthode de résolution de ce système permettant de minimiser les erreurs sur la solution obtenue.

5.4.1.2 Caractéristiques du système d'équations à résoudre

Le système à résoudre est un système d'équations linaires, de la forme $A \cdot x = b$, pour lequel :

- la matrice A du système est de dimension quelconque $m \times n$, avec $m \geq n$, non-inversible et mal conditionnée,
- le nombre m d'équations n'est pas contraint (système potentiellement surdéterminé).

Les notions mathématiques mises en œuvre sont définies dans l'annexe F.

5.4.2 Reformulation du système

Décomposition en valeurs singulières

Soit le système à résoudre :

$$A \cdot x = b \quad (5.6)$$

Il est possible de décomposer la matrice A en valeurs singulières. On obtient alors :

$$A = U \cdot \Sigma \cdot V^T \quad (5.7)$$

On a en particulier (cf. annexe F) :

- U matrice carrée de dimension m avec $U^{-1} = U^T$,
- V matrice carrée de dimension n avec $V^{-1} = V^T$,
- Σ matrice rectangulaire de dimension $m \times n$.

Ainsi, d'après les équations (5.6) et (5.7), on obtient le nouveau système à résoudre :

$$U \cdot \Sigma \cdot V^T \cdot x = b$$

$$\iff \Sigma \cdot V^T \cdot x = U^{-1} \cdot b$$

$$\iff \Sigma \cdot V^T \cdot x = U^T \cdot b \quad (5.8)$$

Proposition 5.4.1 Soient α un réel quelconque et $I_{n \times m}$ la matrice identité de dimension $n \times m$. Tout système d'équations linéaires $A \cdot x = b$ à m équations et n inconnues (avec $m \geq n$) de conditionnement quelconque peut être mis sous la forme :

$$x = G \cdot x + h \quad (5.9)$$

avec :

- G matrice carrée de dimension n telle que : $G = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot D \cdot V^T$
- h vecteur de longueur n tel que : $h = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot U^T \cdot b$

Démonstration

On note D la matrice telle que :

$$\Sigma + D = \alpha \cdot I_{m \times n} \iff \Sigma = \alpha \cdot I_{m \times n} - D \quad (5.10)$$

où :

- α est un réel quelconque,
- $I_{m \times n}$ est la matrice identité de dimension $m \times n$.

Ainsi, Σ et $\alpha \cdot I_{m \times n}$ étant des matrices diagonales de dimension $m \times n$, D est également une matrice diagonale de dimension $m \times n$.

A noter que α étant un nombre quelconque, il est possible de poser en particulier :

$$\alpha = \sigma_{max}(\Sigma)$$

où $\sigma_{max}(\Sigma)$ désigne la plus grande valeur singulière de Σ .

Grâce aux équations (5.8) et (5.10), le système suivant est obtenu :

$$(\alpha \cdot I_{m \times n} - D) \cdot V^T \cdot x = U^T \cdot b$$

$$\iff \alpha \cdot I_{m \times n} \cdot V^T \cdot x = D \cdot V^T \cdot x + U^T \cdot b$$

On multiplie, de manière transparente, chaque membre de l'équation par la matrice identité $I_{n \times m}$:

$$I_{n \times m} \cdot \alpha \cdot I_{m \times n} \cdot V^T \cdot x = I_{n \times m} \cdot D \cdot V^T \cdot x + I_{n \times m} \cdot U^T \cdot b$$

De plus, si $m \geq n$, on a : $I_{n \times m} \cdot I_{m \times n} = I_n$

$$\implies \alpha \cdot V^T \cdot x = I_{n \times m} \cdot D \cdot V^T \cdot x + I_{n \times m} \cdot U^T \cdot b$$

La matrice V est inversible, et de plus :

$$V^{-1} = V^T \iff (V^T)^{-1} = V$$

On obtient alors :

$$x = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot D \cdot V^T \cdot x + \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot U^T \cdot b$$

L'équation obtenue est bien de la forme $x = G \cdot x + h$.

5.4.3 Résolution du système par itérations

5.4.3.1 Définition de la suite

Le système défini par l'équation (5.9) de la proposition (5.4.1) peut être résolu par itérations successives :

$$\begin{cases} x_0 : \text{conditions initiales} \\ x_i = G \cdot x_{i-1} + h \end{cases} \quad (5.11)$$

A chaque itération, une nouvelle valeur de la solution x du système initial est calculée. Les conditions initiales peuvent être obtenues en suivant la méthodologie définie dans la section 5.3.2.

5.4.3.2 Convergence de la suite

Théorème 5.4.2 *La suite (5.11) converge si le rayon spectral de la matrice G , noté $\rho(G)$, est strictement inférieur à 1.*

Théorème 5.4.3 *Si la suite (5.11) converge, alors elle converge vers la solution x cherchée.*

Propriété 5.4.4 *La vitesse de convergence de la suite (5.11) est d'autant plus élevée que le rayon spectral de la matrice G est faible.*

Démonstration

Soit la suite définie par l'équation (5.11). Cette suite est convergente si :

$$\lim_{i \rightarrow +\infty} x_i = x$$

On note e_i l'écart entre l'estimation x_i de la solution à l'itération i et la solution exacte x :

$$e_i = x_i - x$$

Considérons le système à résoudre donné par l'équation (5.9) et la formule de récurrence (5.11) utilisée. On a :

$$\begin{cases} x = G \cdot x + h \\ x_i = G \cdot x_{i-1} + h \end{cases}$$

En soustrayant les deux équations, on obtient :

$$x_i - x = G \cdot x_{i-1} - G \cdot x$$

$$\iff x_i - x = G \cdot (x_{i-1} - x)$$

$$\iff e_i = G \cdot e_{i-1}$$

Par définition de l'erreur e_i , on a :

$$e_i = G^i \cdot e_0$$

La suite est convergente si l'écart entre l'estimation x_i de la solution et sa valeur exacte x tend à être nul pour un nombre important d'itérations :

$$\lim_{i \rightarrow +\infty} e_i = 0$$

Il est démontré dans [141] que, pour toute matrice M carrée, et pour tout vecteur v , on a :

$$\lim_{k \rightarrow +\infty} M^k \cdot v = 0 \iff \rho(M) < 1$$

On en déduit alors :

$$\rho(G) < 1 \iff \lim_{i \rightarrow +\infty} e_i = 0 \iff \lim_{i \rightarrow +\infty} x_i = x$$

Ainsi, la suite (5.11) utilisée pour résoudre le système $A \cdot x = b$ converge si le rayon spectral de la matrice G , c'est-à-dire sa valeur propre maximale, est inférieur à 1. De plus, si la suite converge, elle converge alors vers la solution x cherchée.

5.4.3.3 Lien entre le conditionnement du système initial et la convergence de la suite

Proposition 5.4.5 *La convergence de la suite (5.11) ne dépend que de la matrice A du système initial.*

Démonstration

La suite converge si : $\rho(G) < 1$.

De plus, on a : $G = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot D \cdot V^T$

avec :

- $\alpha = \sigma_{max}(\Sigma)$
- $\Sigma + D = \alpha \cdot I_{m \times n}$

Les matrices Σ et V étant issues de la décomposition de la matrice A en valeurs singulières, seule celle-ci influence la convergence de la suite.

Théorème 5.4.6 *Si la matrice A du système à résoudre est mal conditionnée, alors la suite (5.11) utilisée pour le résoudre converge mal.*

Démonstration

On a : $G = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot D \cdot V^T$

D étant une matrice diagonale, par définition, le rayon spectral de la matrice G dépend du rapport entre les valeurs contenues sur la diagonale de la matrice D et le nombre α :

$$\rho(G) = \frac{\sigma_{\max}(D)}{\alpha}$$

On rappelle la décomposition en valeurs singulières de la matrice A donnée par l'équation (5.7) :

$$A = U \cdot \Sigma \cdot V^T$$

On sait que les valeurs singulières de A sont les mêmes que celles de Σ , et en particulier :

$$\sigma_{\min}(A) = \sigma_{\min}(\Sigma) \quad \text{et} \quad \sigma_{\max}(A) = \sigma_{\max}(\Sigma)$$

La matrice A est supposée mal conditionnée, c'est-à-dire : $\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} \gg 1$

En particulier, on a :

$$\kappa(A) \xrightarrow[\sigma_{\min}(A) \rightarrow 0]{} +\infty \iff \kappa(A) \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} +\infty$$

D'après l'équation (5.10), on a :

$$\Sigma + D = \alpha \cdot I_{m \times n}$$

Le conditionnement de la matrice diagonale $\alpha \cdot I_{m \times n}$ est égal à 1, car :

$$\sigma_{\min}(\alpha \cdot I_{m \times n}) = \sigma_{\max}(\alpha \cdot I_{m \times n}) = \alpha$$

Les matrices Σ et D étant diagonales, on a ainsi, d'après l'équation (5.10) :

$$\begin{cases} \sigma_{\max}(\Sigma) + \sigma_{\min}(D) = \alpha \\ \sigma_{\min}(\Sigma) + \sigma_{\max}(D) = \alpha \end{cases}$$

Comme $\sigma_{\max}(\Sigma) = \alpha$, on en déduit :

$$\begin{cases} \sigma_{\min}(D) = 0 \\ \sigma_{\max}(D) = \alpha - \sigma_{\min}(\Sigma) \end{cases}$$

En particulier, on a :

$$\sigma_{\max}(D) \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} \alpha \iff \frac{\sigma_{\max}(D)}{\alpha} \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} 1 \iff \rho(G) \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} 1$$

On met ainsi en avant les deux résultats suivants :

$$\begin{cases} \rho(G) \xrightarrow{\sigma_{\min}(\Sigma) \rightarrow 0} 1 \\ \kappa(A) \xrightarrow{\sigma_{\min}(\Sigma) \rightarrow 0} +\infty \end{cases}$$

Par conséquent, des valeurs très faibles de $\sigma_{\min}(\Sigma)$ impliquent à la fois un mauvais conditionnement du système, ainsi qu'une non-convergence de la méthode itérative.

5.4.4 Résultats obtenus

Nous choisissons d'implémenter cette méthode de résolution à l'aide de Matlab, et d'utiliser ainsi les outils de décomposition de matrices en valeurs singulières qui lui sont propres.

Les exécutions représentées par la figure 5.2 sont une nouvelle fois utilisées pour évaluer l'exactitude des solutions obtenues. Deux cas de figures sont traités :

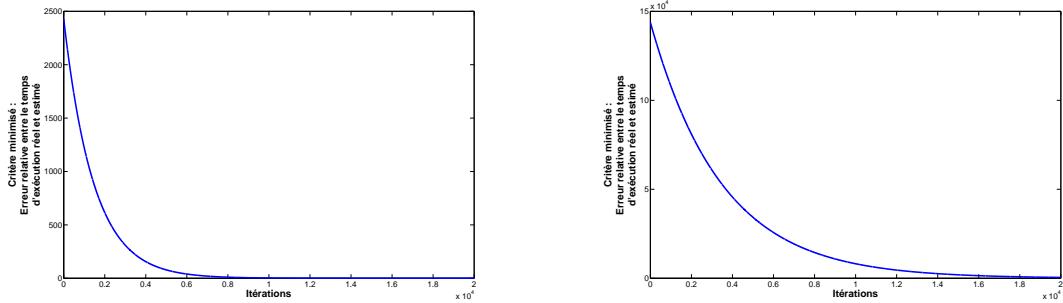
- **Cas n°1** : le système initial $A \cdot x = b$ résolu comporte 46 équations ; les résultats obtenus sont ensuite utilisés pour estimer le temps d'exécution de l'application pour 156 jeux d'entrées différents. Les temps d'exécution considérés varient entre 2 et 1500 secondes.
- **Cas n°2** : l'ensemble des expériences utilisées est restreint à celles dont le temps d'exécution est supérieur à 400 secondes.

Au cours des itérations, la solution calculée doit converger, si cela est possible, vers la solution cherchée. Connaissant le temps d'exécution réel des applications mesuré par l'outil de profiling *gprof*, il est possible de calculer l'erreur relative entre ce temps mesuré et celui estimé à l'aide de la solution du système déterminée. Cette erreur relative constitue le critère sur lequel nous basons l'évaluation de l'exactitude des solutions obtenues à chaque itération.

La figure 5.8 montre l'évolution de ce critère au cours des itérations, dans les deux cas de figures considérés. Des données numériques sont disponibles dans le tableau 5.2. Ces données confirment ce qui a été démontré précédemment. Dans les deux cas, le système est mal conditionné. Cependant, dans le cas n°1, la méthode parvient à trouver une solution générant une erreur globale sur l'ensemble des jeux d'entrées considérés inférieure à 25%, contrairement au second cas, où celle-ci est supérieure à 70%. De plus, il est possible de constater que la méthode converge plus rapidement vers une solution dans le premier cas que dans le second. Ceci s'explique par le conditionnement des matrices des systèmes résolus, très supérieur dans le cas n°2. Les propriétés de convergence de la suite vis-à-vis du conditionnement de la matrice du système sont ainsi vérifiées.

	<i>Erreur relative globale à l'ensemble des exécutions considérées</i>	<i>Nombre d'itérations nécessaires à la convergence de la méthode</i>	<i>Conditionnement de la matrice du système résolu</i>
Cas n°1	23,6%	9000	$3,45 \times 10^{27}$
Cas n°2	71,8%	18000	$3,73 \times 10^{40}$

TAB. 5.2 – Données numériques relevées au cours de l'évaluation de la méthode itérative non-améliorée

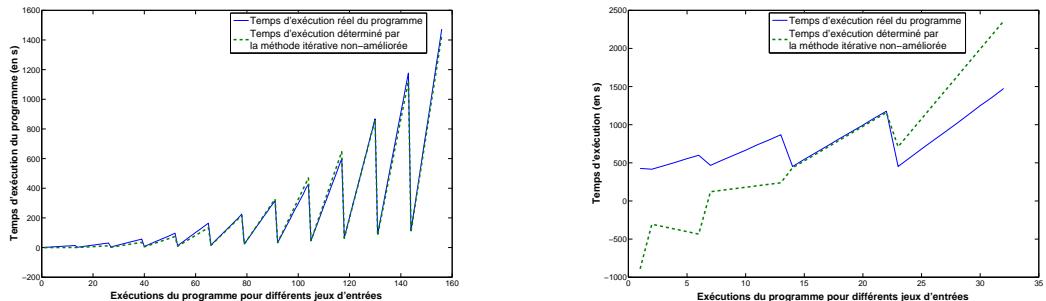


(a) Cas n°1 : estimation sur la totalité des expériences

(b) Cas n°2 : restriction des expériences utilisées

FIG. 5.8 – Evolution de l'erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative non-améliorée

Enfin, la figure 5.9 présente à la fois les temps d'exécution mesurés et ceux estimés par la méthode itérative décrite précédemment. Dans le cas n°1, les temps d'exécution estimés sont proches des temps mesurés. Une différence globale de 23,6% les sépare (cf. tableau 5.2). En majorité, les erreurs commises par la méthode itérative concernent les exécutions dont les temps d'exécution sont faibles. En revanche, le cas n°2, pour lequel le conditionnement de la matrice du système est très élevé, donne des résultats totalement inacceptables. En effet, en plus d'être éloignés de la réalité, les temps d'exécution estimés sont parfois négatifs.



(a) Cas n°1 : estimation sur la totalité des expériences

(b) Cas n°2 : restriction des expériences utilisées

FIG. 5.9 – Comparaison entre le temps d'exécution estimé par la résolution itérative non-améliorée et le temps d'exécution réel du programme

L'étude réalisée confirme les conclusions émises, à savoir que la méthode, telle qu'elle a été décrite, peut ne pas converger. La propriété de convergence dépend bien du conditionnement du système à résoudre. Il est donc indispensable de mettre au point une méthode permettant à la suite définie de converger correctement vers la bonne solution. La section suivante montre comment cela peut être réalisé.

5.5 Amélioration de la convergence de la résolution itérative du système d'équations

Cette section décrit une méthode permettant d'améliorer la convergence de la résolution itérative. Cette méthode s'appuie sur l'introduction d'une erreur ε permettant d'obtenir les conditions nécessaires au bon fonctionnement de la méthode itérative décrite précédemment. Cette erreur devra être minimale afin de ne pas fausser les résultats obtenus.

5.5.1 Introduction d'une erreur ε dans le système à résoudre

Nous avons démontré que la suite définie par l'équation (5.11) ne converge pas car :

$$\sigma_{\max}(D) \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} \alpha$$

Il est donc indispensable d'empêcher cette condition de se réaliser.

Proposition 5.5.1 Soit ε , une matrice diagonale de dimension $m \times n$, que nous qualifions de "matrice d'erreur". L'ajout de cette erreur aux valeurs singulières de A engendre un nouveau système $A' \cdot \bar{x} = b$, dont la solution tend vers la solution de $A \cdot x = b$ lorsque ε tend vers 0.

Démonstration

On a la relation suivante, issue de la décomposition de A en valeurs singulières (équation (5.7)) :

$$\sigma(A) = \sigma(\Sigma)$$

Soit Σ' , matrice diagonale contenant les valeurs singulières modifiées par ε de A . On a :

$$\sigma(\Sigma') = \sigma(A) + \sigma(\varepsilon) = \sigma(\Sigma) + \sigma(\varepsilon)$$

Les matrices Σ , Σ' et ε étant diagonales, on en déduit :

$$\Sigma' = \Sigma + \varepsilon$$

On choisit alors de substituer Σ par Σ' , afin d'introduire l'erreur ε dans le système. On définit la matrice A' résultant de cette substitution :

$$A' = U \cdot \Sigma' \cdot V^T = U \cdot (\Sigma + \varepsilon) \cdot V^T \quad (5.12)$$

$$\iff A' = A + U \cdot \varepsilon \cdot V^T$$

$$\implies A' \xrightarrow[\varepsilon \rightarrow 0]{} A$$

Soient les systèmes $A \cdot x = b$ et $A' \cdot \bar{x} = b$, on a par conséquent :

$$A' \xrightarrow[\varepsilon \rightarrow 0]{} A \implies \bar{x} \xrightarrow[\varepsilon \rightarrow 0]{} x$$

5.5.2 Reformulation du système à résoudre en prenant en compte l'erreur introduite

Nous choisissons donc de résoudre le système $A' \cdot \bar{x} = b$. Ce système dépend de l'erreur ε que l'on choisit d'introduire, celle-ci devant être déterminée de façon à améliorer la convergence du système, sans toutefois en fausser totalement les résultats.

Proposition 5.5.2 *Le système $A' \cdot \bar{x} = b$ à résoudre, après introduction de la matrice d'erreur ε , peut être mis sous la forme :*

$$\bar{x} = G' \cdot \bar{x} + h$$

avec :

- G' matrice carrée de dimension n telle que : $G' = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot (D - \varepsilon) \cdot V^T$
- h vecteur de longueur n tel que : $h = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot U^T \cdot b$

Démonstration

D'après l'équation (5.12), on a :

$$A' \cdot \bar{x} = U \cdot (\Sigma + \varepsilon) \cdot V^T \cdot \bar{x} = b$$

Or, l'équation (5.10) pose :

$$\Sigma = \alpha \cdot I_{m \times n} - D$$

D'où :

$$U \cdot [\alpha \cdot I_{m \times n} - (D - \varepsilon)] \cdot V^T \cdot \bar{x} = b$$

On en déduit alors, de la même manière que dans la section précédente :

$$[\alpha \cdot I_{m \times n} - (D - \varepsilon)] \cdot V^T \cdot \bar{x} = U^T \cdot b$$

$$\iff \alpha \cdot I_{m \times n} \cdot V^T \cdot \bar{x} = (D - \varepsilon) \cdot V^T \cdot \bar{x} + U^T \cdot b$$

$$\iff \bar{x} = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot (D - \varepsilon) \cdot V^T \cdot \bar{x} + \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot U^T \cdot b \quad (5.13)$$

5.5.3 Résolution du nouveau système par itérations

5.5.3.1 Définition de la suite

Une nouvelle suite qui prend en compte la matrice d'erreur ε peut être définie grâce à l'équation (5.13) :

$$\begin{cases} \bar{x}_0 : conditions initiales \\ \bar{x}_i = G' \cdot \bar{x}_{i-1} + h \end{cases} \quad (5.14)$$

5.5.3.2 Convergence de la suite

Théorème 5.5.3 *La suite (5.14) converge si le rayon spectral de la matrice G' , noté $\rho(G')$, est strictement inférieur à 1.*

Théorème 5.5.4 *Si la suite (5.14) converge, alors elle converge vers la solution \bar{x} cherchée.*

Propriété 5.5.5 *La vitesse de convergence de la suite (5.14) est d'autant plus élevée que le rayon spectral de la matrice G' est faible.*

Démonstration

On démontre de manière analogue à la section 5.4.3.2 que :

$$\lim_{i \rightarrow +\infty} \bar{x}_i = \bar{x}$$

Rappelons cependant que la suite converge vers une solution \bar{x} qui n'est pas exactement la solution x du système $A \cdot x = b$ recherchée. Néanmoins, il a été démontré que :

$$\bar{x} \xrightarrow[\varepsilon \rightarrow 0]{} x$$

5.5.3.3 Choix de la matrice d'erreur

Si le rayon spectral de la matrice G' peut être calculé à partir de la matrice d'erreur ε , nous allons montrer dans cette section que l'opération inverse est également possible.

Proposition 5.5.6 *Etant donnée une valeur $\rho(G')$ du rayon spectral de la matrice G' , une valeur possible de la matrice d'erreur ε pour l'obtenir est :*

$$\varepsilon = (1 - \rho(G')) \cdot D \quad (5.15)$$

Démonstration

On a : $G' = \alpha^{-1} \cdot V \cdot I_{n \times m} \cdot (D - \varepsilon) \cdot V^T$

Par définition, on a :

$$\rho(G') = \frac{\sigma_{max}(D - \varepsilon)}{\alpha}$$

$$\iff \sigma_{max}(D - \varepsilon) = \alpha \cdot \rho(G')$$

Les matrices D et ε étant des matrices diagonales, on a :

$$\begin{aligned}\sigma_{\max}(D - \varepsilon) &= \sigma_{\max}(D) - \sigma_{\max}(\varepsilon) \\ \implies \sigma_{\max}(D) - \sigma_{\max}(\varepsilon) &= \alpha \cdot \rho(G') \\ \iff \sigma_{\max}(\varepsilon) &= \sigma_{\max}(D) - \alpha \cdot \rho(G')\end{aligned}$$

Or, nous avons démontré précédemment que la relation suivante est vérifiée :

$$\sigma_{\max}(D) \xrightarrow[\sigma_{\min}(\Sigma) \rightarrow 0]{} \alpha$$

Le système étant mal conditionné ($\sigma_{\min}(\Sigma) \rightarrow 0$), on peut supposer pour le calcul d' ε que :

$$\sigma_{\max}(D) = \alpha \implies \sigma_{\max}(\varepsilon) = \sigma_{\max}(D) \cdot (1 - \rho(G'))$$

Il apparaît ainsi qu'une valeur d' ε possible pour obtenir $\rho(G')$ est :

$$\varepsilon = (1 - \rho(G')) \cdot D$$

Propriété 5.5.7 *La matrice d'erreur ε est d'autant plus faible que le rayon spectral de la matrice G' est élevé.*

Corrolaire 5.5.8 *L'erreur entre la solution \bar{x} déterminée et la solution x cherchée est d'autant plus faible que le rayon spectral de la matrice G' est élevé.*

5.5.4 Résultats obtenus

Les deux cas de figures traités dans la section précédente sont repris afin d'évaluer l'exactitude des solutions obtenues :

- **Cas n°1** : le système initial $A \cdot x = b$ résolu comporte 46 équations ; les résultats obtenus sont ensuite utilisés pour estimer le temps d'exécution de l'application pour 156 jeux d'entrées différents. Les temps d'exécution considérés varient entre 2 et 1500 secondes.
- **Cas n°2** : l'ensemble des expériences utilisées est restreint à celles dont le temps d'exécution est supérieur à 400 secondes.

La matrice d'erreur est choisie en fonction de la valeur que l'on souhaite donner au rayon spectral de la matrice d'itération G' , en appliquant la formule (5.15). Dans un premier temps, nous effectuons la résolution du système pour différentes valeurs de $\rho(G')$, allant de 0 à 1. La figure 5.10 regroupe les résultats obtenus.

Plusieurs remarques peuvent être formulées quant à ces résultats. Tout d'abord, le cas correspondant à $\rho(G') = 1$ correspond à la version itérative non-améliorée, puisque dans ce cas, la matrice d'erreur est nulle. Les pourcentages d'erreurs obtenus dans le cas n°1 et le cas n°2 (respectivement 23,6% et 71,8%) correspondent bien à ceux donnés dans la section précédente.

Dès que le rayon spectral de la matrice d'itération devient inférieur à 1, les solutions obtenues sont nettement meilleures. Dans le cas n°1, l'erreur relative moyenne constatée sur l'ensemble des exécutions est d'environ 7,5%, tandis qu'elle avoisine les 2% dans le second cas. En pratique, des solutions correctes sont obtenues pour des valeurs de $\rho(G')$ comprises entre 0,9 et 1 (1 étant exclu). Lorsque le rayon spectral de la matrice G' est inférieur à 0,9 on voit apparaître un phénomène oscillatoire pour lequel l'erreur relative varie au sein d'une plage de valeurs, et dont l'amplitude augmente lorsque $\rho(G')$ diminue. Ainsi, la confiance qui peut être accordée aux résultats obtenus diminue lorsque le rayon spectral de la matrice d'itération diminue, c'est-à-dire lorsque l'erreur ε introduite augmente.

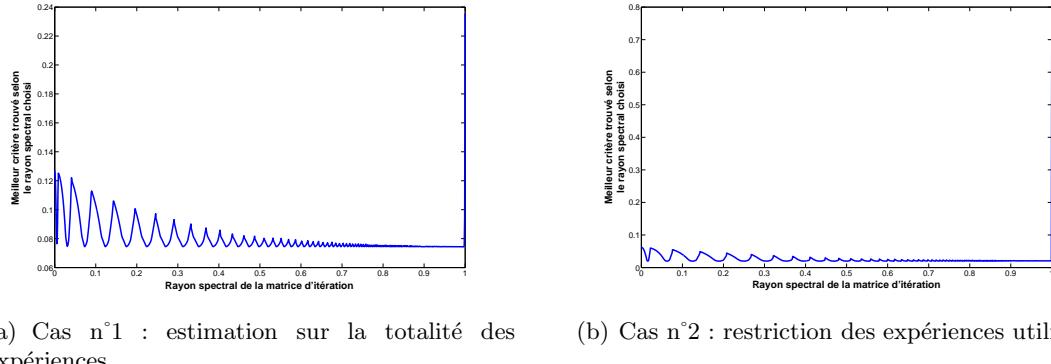


FIG. 5.10 – Erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative améliorée en fonction du rayon spectral de la matrice d'itération choisie

La figure 5.11 montre l'évolution du critère minimisé au cours des itérations pour différentes valeurs du rayon spectral de la matrice G' . On voit bien que la vitesse de convergence augmente lorsque $\rho(G')$ diminue, ce qui est conforme à nos attentes : l'erreur introduite permet bien une meilleure convergence de l'algorithme. Cependant, l'erreur ε introduite doit être ajustée de manière à trouver un compromis entre la vitesse de convergence de la méthode itérative et la fiabilité des résultats qu'elle produit.

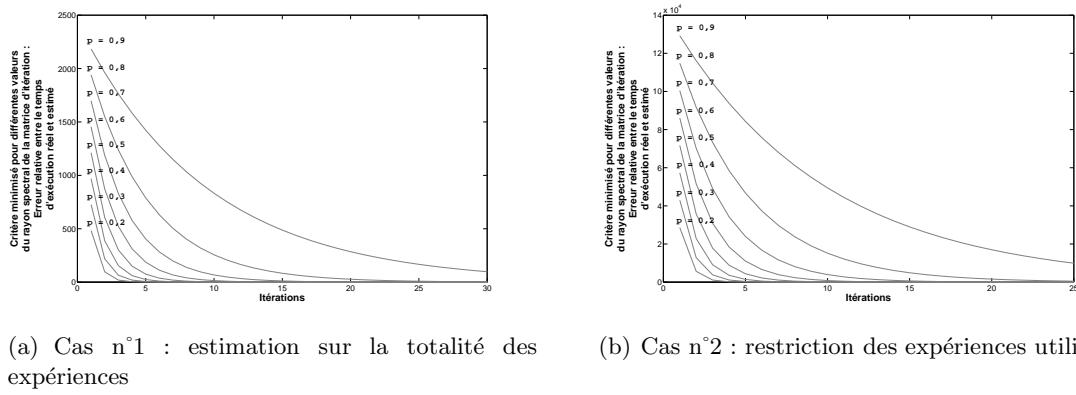


FIG. 5.11 – Evolution de l'erreur relative entre le temps d'exécution réel et celui estimé par la méthode itérative améliorée pour différentes valeurs de $\rho(G')$

Pour conclure, la figure 5.12 compare les temps d'exécution réels de l'application pour différents jeux d'entrées avec ceux estimés par la méthode améliorée de résolution itérative

de systèmes d'équations linéaires mal conditionnés. Les résultats obtenus ici sont très satisfaisants. Il est possible d'observer que la courbe des temps d'exécution estimés suit de près la courbe correspondant aux temps mesurés, et ce pour les deux cas traités. Le rayon spectral de G' choisi dans les deux cas est de 0,99.

Le cas n°1 donne une erreur relative globale de 7,43% sur l'ensemble des exécutions du programme, ce qui est nettement mieux que les 23,6% obtenus lors de la résolution initiale. Ce sont notamment les estimations correspondant aux exécutions ayant un faible temps d'exécution qui ont été améliorées, au détriment parfois de celles ayant un temps d'exécution plus important. En revanche, le cas n°2 ne regroupe que des exécutions dont les temps ne sont pas aussi éloignés les uns des autres (de 400 à 1500 secondes). Dans ce cas, l'erreur relative moyenne constatée est très raisonnable, puisqu'elle n'est que de 1,96% (contre 71,8% pour la méthode itérative non-améliorée). En outre, les deux courbes semblent parfaitement se chevaucher.

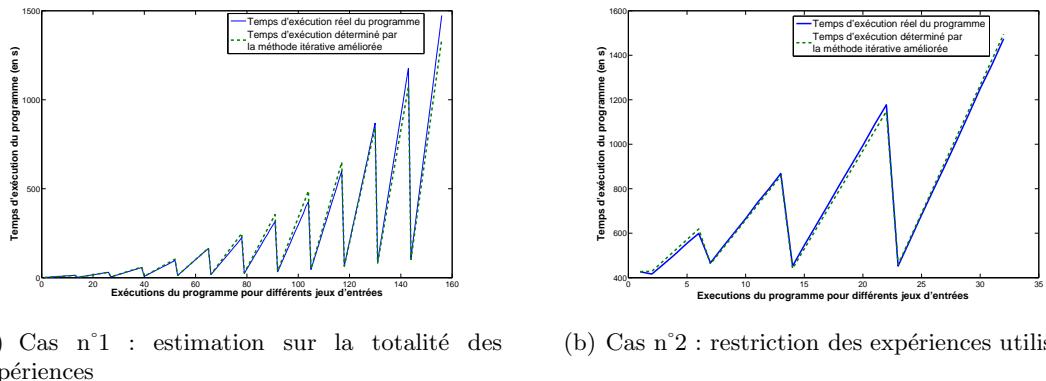


FIG. 5.12 – Comparaison entre le temps d'exécution estimé par la résolution itérative améliorée et le temps d'exécution réel du programme

5.6 Conclusion

Dans ce chapitre, nous avons traité le problème de la détermination des informations temporelles d'un programme. Il s'agit ici, étant donné un découpage du programme en blocs de base étendus maximaux, de calculer le temps d'exécution de ces derniers.

Pour cela, nous considérons que le temps d'exécution d'un bloc de base est constant, quelles que soient les entrées appliquées au programme. Cette hypothèse simplificatrice est, dans la réalité, remise en cause de par les mécanismes d'optimisation des processeurs modernes, tels que les caches et les pipelines.

Nous avons montré que l'obtention des temps d'exécution des blocs de base d'un programme se ramène à la résolution d'un système d'équations linéaires pouvant être obtenu à partir de différentes exécutions du programme pour des jeux d'entrées distincts. La matrice du système contient le nombre d'exécutions de chacun des blocs de base du programme, automatiquement calculé à partir des sorties produites par l'outil *gcov*. Le vecteur second membre contient, quant à lui, le temps d'exécution global du programme, déterminé par *gprof*.

La matrice du système d'équations à résoudre est mal conditionnée. Ceci est dû à la nature même de cette matrice, qui contient des colonnes présentant très fréquemment des dépendances entre elles. Ainsi, les méthodes de résolutions standards, issues de la programmation linéaire ou bien de l'optimisation de systèmes de fonctions non-linéaires, donnent des résultats erronés.

Nous avons mis au point une méthode de résolution du système, s'appuyant sur la définition d'une suite itérative convergeant vers la solution cherchée. Cette méthode donne de très bons résultats, dès lors que l'on introduit un faible coefficient d'erreur permettant cette convergence.

Les temps d'exécution des blocs de base ainsi calculés permettent de donner une estimation du temps global de l'application pour différents jeux d'entrées. Dans ce chapitre, ces estimations ont été effectuées en supposant connus les nombres d'exécutions des blocs de base pour chacune d'entre elles. Dans la réalité, ces informations ne seront pas connues de manière exacte, mais devront elles-aussi être estimées. C'est le propos du chapitre suivant, qui décrit une méthode de détermination des informations comportementales d'un programme à partir d'un historique d'exécutions passées.

Chapitre 6

Etude du comportement d'un programme

Ce chapitre est consacré à l'étude du comportement d'un programme, l'objectif étant d'estimer le chemin d'exécution emprunté au sein de son graphe de flot de contrôle en fonction des entrées qui lui sont appliquées. L'obtention de telles informations est réalisée par le biais d'un apprentissage basé sur des instances.

Dans une première partie de ce chapitre, nous revenons sur le modèle mathématique servant de base à la prédiction du comportement du programme. Une implémentation de ce modèle est ensuite utilisée pour prédire le comportement d'une application parallèle. Dans une seconde partie, nous proposons un moyen d'améliorer les résultats obtenus en permettant au programmeur de l'application de donner des indications sur le comportement de celle-ci par le biais d'annotations de son code source.

6.1 Introduction

L'approche hybride de prédiction du temps d'exécution d'une application repose sur l'équation (4.9) définie dans le chapitre 4 :

$$T_{App}(E) = \sum_{b \in \mathbb{BB}} N_b^{BB}(E) \cdot T_b^{BB}$$

où \mathbb{BB} représente l'ensemble des blocs de base étendus maximaux de l'application.

Le chapitre précédent a proposé une méthode d'obtention du temps d'exécution T_b^{BB} des blocs de base étendus maximaux d'un programme. L'objectif maintenant est donc de pouvoir estimer le nombre d'exécutions $N_b^{BB}(E)$ de chacun d'entre eux en fonction des entrées du programme.

Ainsi, la connaissance de ces deux entités permettra l'estimation du temps d'exécution du programme pour un jeu d'entrées donné.

6.2 Méthode d'estimation du nombre d'exécutions des blocs de base d'un programme

Le comportement d'un programme est prédit en estimant le chemin d'exécution emprunté au sein de son graphe de flot de contrôle en fonction des entrées qui lui sont appliquées. Cette estimation est réalisée de manière dite "implicite" puisque seuls les nombres d'exécutions des blocs de base seront considérés.

Une approche d'apprentissage basé sur des instances est utilisée à ce propos (cf. section 4.3.3). Il s'agit de déterminer le nombre d'occurrences $N_b^{BB}(E)$ des blocs de base dans le chemin d'exécution à partir d'un historique d'exécutions passées. L'ensemble des données collectées au cours de ces exécutions est stocké au sein d'une base de connaissances qui associe les jeux d'entrées du programme avec les nombres d'exécutions de chacun des blocs correspondants.

Notons que le terme d'*expérience* est utilisé afin de désigner une exécution passée, pour laquelle les $N_b^{BB}(E)$ ont été déterminés et stockés dans la base de connaissances. L'exécution pour laquelle ces données doivent être estimées est quant à elle qualifiée de *requête*.

6.2.1 Modèle mathématique

Le modèle défini dans cette section permet de sélectionner, au sein de la base de connaissances, un ensemble d'expériences semblables à la requête, puis de les combiner de manière à estimer le nombre d'exécutions des blocs de base d'un programme pour un vecteur d'entrées donné.

On note \mathbb{E} l'ensemble des vecteurs d'entrées admissibles du programme traité.

Soit X le nombre d'expériences contenues dans la base de connaissances. Chaque expérience est associée à un vecteur d'entrées $E^{(i)}$, avec $i \in [1; X]$, de la forme :

$$E^{(i)} = \left\{ E_v^{(i)} \right\}_{v \in [1; N^V]} = \begin{bmatrix} E_1^{(i)} \\ E_2^{(i)} \\ \vdots \\ E_{N^V}^{(i)} \end{bmatrix} \quad \text{avec } E^{(i)} \in \mathbb{E}$$

où N^V correspond au nombre de valeurs constituant les entrées du programme.

L'ensemble des vecteurs d'entrées correspondant aux expériences contenues dans la base de connaissances est un sous-ensemble de \mathbb{E} , que nous notons \mathbb{E}^{Exp} :

$$\mathbb{E}^{Exp} = \bigcup_{i \in [1; X]} \left\{ E^{(i)} \right\} \subseteq \mathbb{E} \quad \Rightarrow \quad \text{Card}(\mathbb{E}^{Exp}) = X$$

Afin de mesurer l'éloignement de deux contextes d'exécution, la distance euclidienne est utilisée [137]. Soient deux vecteurs d'entrées quelconques $E^{(1)}$ et $E^{(2)}$ du programme, celle-ci s'exprime de la manière suivante :

$$\forall E^{(1)} \in \mathbb{E} \quad , \quad \forall E^{(2)} \in \mathbb{E} \quad D\left(E^{(1)}, E^{(2)}\right) = \sqrt{\sum_{v=1}^{N^V} d\left(E_v^{(1)}, E_v^{(2)}\right)^2} \quad (6.1)$$

où $d\left(E_v^{(1)}, E_v^{(2)}\right)$ est la distance entre deux valeurs d'entrées définie à l'aide d'une métrique hétérogène de distance :

$$d\left(E_v^{(1)}, E_v^{(2)}\right) = \begin{cases} 1 & \text{si } E_v^{(1)} \text{ ou } E_v^{(2)} \text{ est inconnue,} \\ \text{si } E_v^{(x)} \text{ est nominale,} & \begin{cases} 0 & \text{si la valeur } E_v^{(1)} = E_v^{(2)}, \\ 1 & \text{dans le cas contraire.} \end{cases} \\ \text{si } E_v^{(x)} \text{ est linéaire,} & \frac{|E_v^{(1)} - E_v^{(2)}|}{\max_{E_v^{(x)}} - \min_{E_v^{(x)}}} \end{cases} \quad (6.2)$$

La distance entre deux vecteurs d'entrées pouvant être exprimée, il est maintenant possible de formuler une estimation du nombre d'occurrences des blocs de base dans le chemin d'exécution pour un vecteur d'entrées. Pour cela, une moyenne pondérée des valeurs contenues dans la base de connaissances est réalisée. La pondération sera effectuée en fonction de la distance séparant les expériences de la requête, afin de favoriser l'influence des expériences les plus proches de la requête [13, 86, 135].

Soit E^* le vecteur d'entrées correspondant à la requête, le nombre d'exécutions de chacun des blocs de base étendus maximaux b du programme est estimé de la manière suivante :

$$\forall b \in \mathbb{BB} \quad N_b^{BB}(E^*) = \frac{\sum_{E \in \mathbb{E}^S} \left[K\left(D\left(E^*, E\right)\right) \cdot N_b^{BB}(E) \right]}{\sum_{E \in \mathbb{E}^S} K\left(D\left(E^*, E\right)\right)} \quad (6.3)$$

où \mathbb{E}^S désigne un ensemble de vecteurs d'entrées correspondant aux expériences sélectionnées au sein de la base de connaissances. Plusieurs possibilités de sélection s'offrent alors :

- Toutes les expériences contenues dans la base de connaissances peuvent être utilisées pour calculer la moyenne pondérée :

$$\mathbb{E}^S = \mathbb{E}^{Exp}$$

- Seul un sous-ensemble d'expériences peut également être utilisé : $\mathbb{E}^S \subset \mathbb{E}^{Exp}$
Dans ce cas, le nombre d'expériences utilisées est restreint, soit par un nombre limite fixe d'expériences, soit par la distance de celles-ci à la requête. Par exemple :

$$\mathbb{E}^S = \{E \in \mathbb{E}^{Exp} \setminus D(E^*, E) < n\} \quad \text{avec } n \in \mathbb{R} \text{ quelconque}$$

Le calcul du nombre d'exécutions des blocs de base repose sur une fonction de pondération, permettant de modifier l'influence des expériences prises en compte en fonction de leur proximité avec la requête. Nous choisissons d'utiliser pour cela la fonction gaussienne :

$$K(d) = e^{-(\frac{d}{k})^2}$$

avec k constante permettant de faire varier la largeur de la gaussienne. Cette constante permet de s'adapter à la densité d'expériences présentes dans la région contenant la requête. La figure 6.1 montre l'influence de k sur $K(d)$.

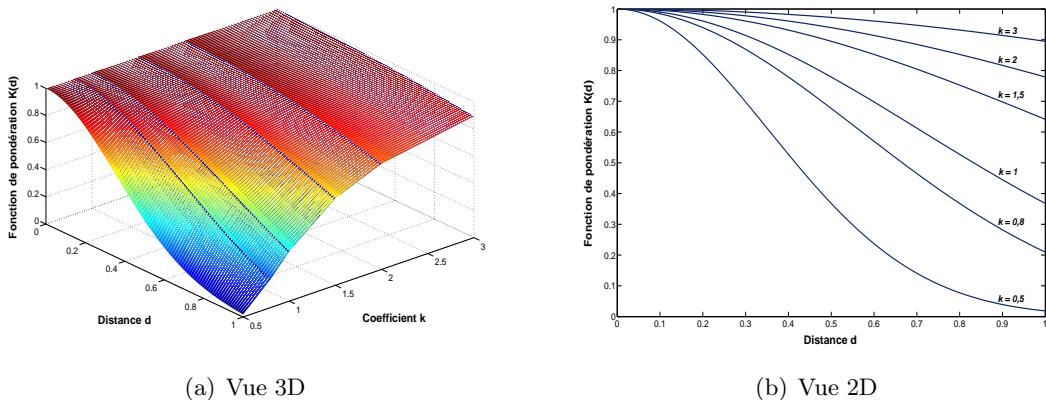


FIG. 6.1 – Influence de la constante k sur la fonction de pondération

6.2.2 Implémentation

6.2.2.1 Principe de fonctionnement

Le module de prédiction développé s'articule autour d'une base de données, dont le rôle est de stocker l'ensemble des expériences. La figure 6.2 montre la structure de cette base de données :

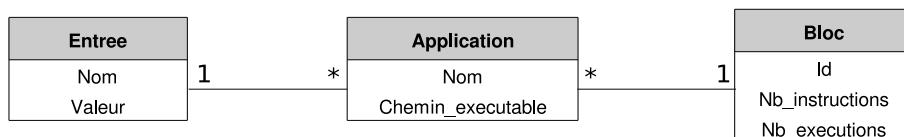


FIG. 6.2 – Structure de la base de données

Chaque expérience correspond à l'exécution d'une application pour des valeurs d'entrées données. Le nombre d'occurrences des blocs de base étendus maximaux dans le chemin d'exécution est connu pour chacune des expériences présentes.

Le principe de fonctionnement du module est donc le suivant (cf. figure 6.3) :

1. Un ensemble d'expériences est stocké au sein de la base de données. Chacune de ces expériences correspond à un jeu d'entrées unique.
2. Lorsqu'une prédiction doit être effectuée, une requête est soumise à la base de données.

- Une estimation du nombre d'exécutions des blocs de base est calculée pour la requête en fonction des expériences présentes au sein de la base de données.

La première étape constitue la phase d'apprentissage du module, tandis que les deux étapes suivantes réalisent la prédiction elle-même.

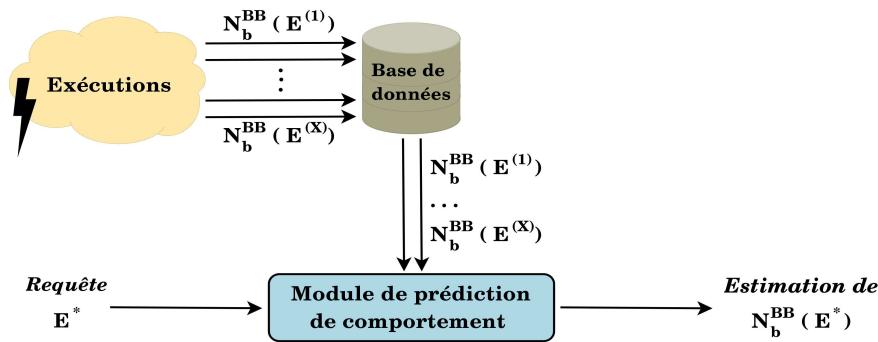


FIG. 6.3 – Principe de fonctionnement du module de prédiction du comportement d'un programme

6.2.2.2 Phase d'apprentissage

Il s'agit ici de réaliser une série d'expériences en vue de la constitution de la base de connaissances. Le programme est donc exécuté avec plusieurs jeux d'entrées. Notons que l'estimation du nombre d'exécutions des blocs de base sera d'autant plus précise que les entrées utilisées durant cette phase sont représentatives de l'ensemble des jeux d'entrées admissibles.

Il est nécessaire de pouvoir découper le programme en blocs de base, puis, à l'issue de chacune des exécutions effectuées, de pouvoir déterminer de manière exacte le nombre d'exécutions de ceux-ci. Pour cela, nous utilisons l'outil de profiling standard *gcov*, de la même manière que dans le chapitre précédent.

Grâce à cet outil, l'exécution du programme génère un fichier correspondant au fichier source du programme auquel est ajouté, en en-tête de chaque ligne, le nombre de fois où chaque instruction a été exécutée. Il est ainsi possible, en fonction de ces données, de constituer les blocs de base étendus maximaux, et de connaître leur nombre d'exécutions.

L'ensemble des données collectées au cours des différentes exécutions du programme peut être ainsi stocké dans la base de données.

En pratique, la base de données pourra être remplie au fur et à mesure. En effet, dans le cadre d'une utilisation de ce module par un ordonnanceur, la prédiction sera exploitée par l'ordonnanceur afin de déterminer un placement de l'application correspondant à la requête. Il est ensuite tout-à-fait envisageable de se servir de l'exécution du programme placé comme d'une nouvelle expérience venant enrichir la base de connaissances.

6.2.2.3 Prédiction

Cette phase s'appuie sur le modèle mathématique défini précédemment afin d'effectuer une prédiction du comportement d'une application. A ce propos, un outil a été développé, afin de permettre à l'utilisateur de soumettre une requête, constituée d'un jeu d'entrées E^* . Les différentes expériences contenues dans la base de connaissances sont alors utilisées pour déterminer le nombre d'occurrences $N_b^{BB}(E^*)$ de chacun des blocs de base b au sein du chemin d'exécution.

6.2.3 Exemple traité

Afin de vérifier les résultats obtenus par le modèle de prédiction de comportement défini dans cette section, et notamment son utilisabilité dans un contexte de grilles de calcul, nous choisissons l'exemple d'une application parallèle parfaitement adaptée à une exécution sur grille. En outre, de par sa structure, son comportement est plus difficilement prévisible que celui de l'exemple traité dans le chapitre précédent (élévation d'une matrice à une puissance donnée).

6.2.3.1 Présentation générale de l'application

L'application mise en place dans cet exemple est une application de filtrage particulaire [142, 143]. Il s'agit d'une technique d'estimation statistique de modèles basée sur des simulations, permettant de déterminer une approximation des variables d'état d'un système à un instant donné à partir de variables d'observation.

Soit un processus d'état x_k évoluant au cours du temps (l'indice k correspond aux intervalles de temps considérés). Les variables d'état du système $x_k \in \mathbb{R}^n$ sont soumises à la loi de transition suivante :

$$x_{k+1} = f_k(x_k, w_k)$$

où $w_k \in \mathbb{R}^p$ est un bruit blanc dynamique dont la densité de probabilité est connue. De plus, la densité de probabilité de l'état initial $p(x_0)$ est supposée connue (loi initiale).

A intervalles de temps k réguliers, des mesures de l'état x_k sont effectuées. On définit ainsi la série d'observations $y_k \in \mathbb{R}^p$ correspondant à ces mesures. On note Y_k l'ensemble des observations connues et disponibles à l'instant k : $Y_k = \{y_0, y_1, \dots, y_k\}$. Ces observations sont liées à l'état du système par loi d'observation suivante :

$$y_k = h_k(x_k, v_k)$$

où $v_k \in \mathbb{R}^p$ est un bruit d'observation de densité de probabilité connue.

Des entités appelées *particules* sont alors mises en œuvre afin d'explorer l'espace d'état de manière aléatoire à la recherche de la solution (la valeur des variables d'état à un instant k donné).

Une particule exprime une hypothèse sur la valeur de ces variables à l'instant cherché. Elle correspond ainsi à un élément de l'espace d'état pondéré par une densité de probabilité $p(x_k|Y_k)$, permettant de caractériser la probabilité que cet élément corresponde à l'état dans lequel le système peut se trouver d'après les observations effectuées. L'objectif est ainsi de faire évoluer les particules de façon à ce que leur ensemble approche la distribution $p(x_k|Y_k)$. Une estimation optimale de x_k est alors possible.

Soit N^P le nombre de particules utilisées. La détermination des variables d'état x_k à l'instant k met en jeu les étapes suivantes :

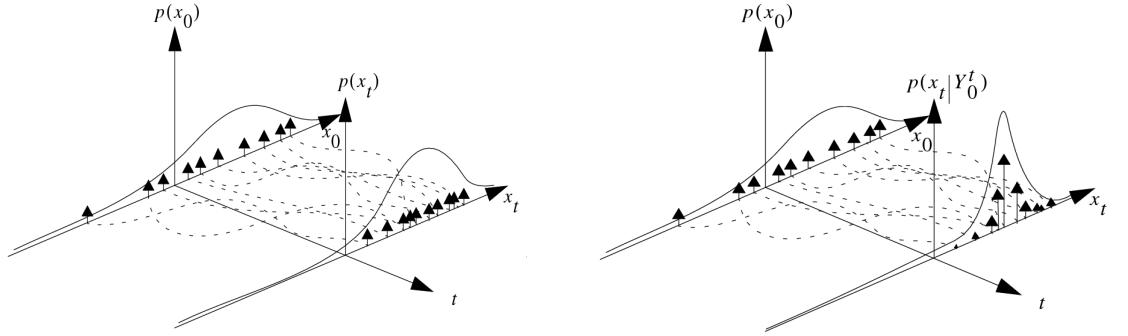
1. **Phase d'initialisation :** Une approximation de la densité de probabilité initiale $p(x_0)$ est déterminée. Les N^P particules sont alors aléatoirement distribuées sur l'espace d'état selon cette densité.
2. **Phase de prédiction :** Cette phase met en jeu l'exploration de l'espace d'état par les différentes particules. Chacune d'entre elles évolue indépendamment des autres selon la loi de transition f_k . On a alors :

$$\forall i \in \{1, \dots, N^P\} \quad x_k^i = f_k(x_{k-1}^i, w_{k-1}^i)$$

Le bruit w_{k-1}^i est issu d'un tirage aléatoire selon la densité de probabilité $p(w_{k-1})$. Cette phase permet ainsi de déterminer une approximation de $p(x_k)$, et donc de faire évoluer les particules comme le montre la figure 6.4(a).

3. **Phase de correction :** A l'instant k , lorsque l'observation y_k devient disponible, la densité de probabilité $p(x_k|Y_k)$ est mise à jour pour chacune des particules, en se basant à la fois sur la probabilité d'observation $p(y_k|x_k)$ et sur la probabilité de transition $p(x_k|x_{k-1})$. Ainsi, à l'évolution en aveugle de la phase de prédiction succède la détermination d'une probabilité conditionnelle dépendant de l'observation (cf. figure 6.4(b)).
4. **Phase de redistribution :** Au cours de cette étape, un nouvel ensemble de particules $\{x_k^i\}_{i=1,\dots,N^P}$ est obtenu en redistribuant les particules en termes de position dans l'espace d'état. Les particules sont dupliquées selon le poids qui leur est associé (le nombre de particules générées à partir d'une particule donnée est d'autant plus important que la densité de probabilité $p(x_k|Y_k)$ associée à cette dernière est élevée), tandis que les particules de faible poids sont éliminées.
5. **Phase d'estimation :** Cette dernière étape permet enfin l'estimation de la variable d'état x_k , grâce aux informations fournies par chacune des particules. Ainsi, les informations de position et de poids des différentes particules sont regroupées afin de calculer une valeur de la variable d'état.

Les étapes de *prédiction/correction/redistribution* décrites ci-dessus constituent une seule itération de l'algorithme de filtrage particulaire. En pratique, la figure 6.5 montre l'enchaînement successif de celles-ci.



(a) Evolution aveugle des particules au cours de la phase de prédition

(b) Evolution conditionnelle des particules au cours de la phase de correction

FIG. 6.4 – Evolution des particules au cours des phases de prédition et de correction

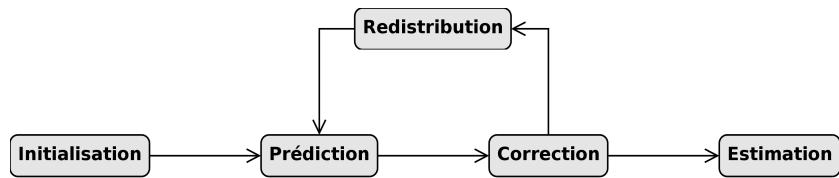


FIG. 6.5 – Etapes de l'algorithme de filtrage particulaire

6.2.3.2 Structure du programme

L'algorithme de filtrage particulaire mis en œuvre par l'application étudiée est un algorithme distribué, mettant en jeu un ensemble de tâches s'exécutant en parallèle sur différents processeurs [142, 143]. Une librairie MPI¹ [144] est utilisée pour établir des communications entre elles. Chacune des tâches est responsable de l'évolution d'un ensemble de particules vers la solution cherchée.

L'application s'appuie sur un modèle *maître/esclave* (cf. figure 6.6). L'ensemble des particules est réparti de manière égale entre N^T tâches esclaves ($\frac{N^P}{N^T}$ particules par tâche). Ces dernières sont en charge de l'initialisation puis de l'évolution des particules. Le rôle de la tâche maître est de calculer, au terme de l'exécution de l'algorithme de filtrage, une estimation de la variable d'état x_k .

L'indépendance des particules entre elles rend possible la mise en parallèle de l'algorithme de filtrage particulaire. En effet, chacune d'entre elles explore l'espace d'état de manière aléatoire et indépendante du comportement des autres particules. L'application choisie comme exemple est donc une application parallèle à gros grain, pour laquelle chaque tâche passera la plupart de son temps à effectuer des calculs indépendamment des autres tâches. Les données envoyées par chaque tâche esclave à la tâche maître ne représentent qu'un très faible volume, puisqu'il s'agit simplement de la position de chacune des particules ainsi que du poids qui lui est associé.

¹MPI (*Message Passing Interface*) propose une norme définissant une bibliothèque de fonctions, dans le but de permettre l'échange de messages entre ressources distantes. Elle est notamment devenue un standard de communication pour des noeuds exécutant des programmes parallèles.

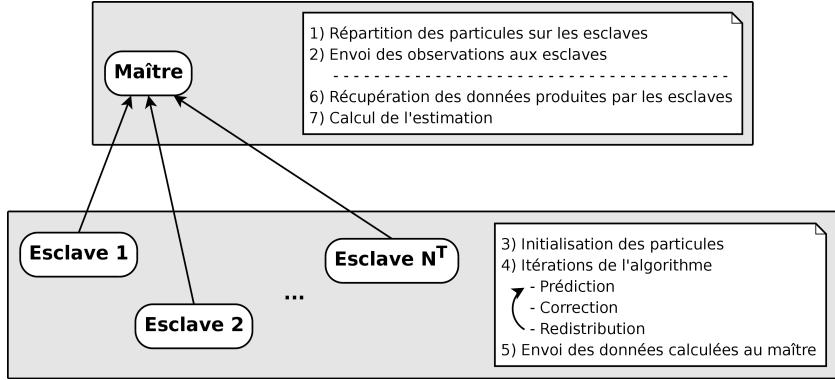


FIG. 6.6 – Structure de l’application de filtrage particulaire

Différentes exécutions de l’application montrent que le temps d’exécution de la tâche maître est très faible par rapport à celui des tâches esclaves. Le tableau 6.1 montre, pour différentes exécutions du programme, que le temps d’exécution de la tâche maître ne dépasse pas 0,2% de celui des tâches esclaves.

Par la suite, nous négligerons le temps d’exécution de la tâche maître. Nous limiterons notre étude à l’estimation du comportement des tâches esclaves de l’application, ces dernières étant de plus identiques.

Temps d’exécution de la tâche maître (en s)	Temps d’exécution de la tâche esclave (en s)	Rapport $\frac{T_{maître}^{Exec}}{T_{esclave}^{Exec}}$
0,12	89,2	0,13 %
0,23	139,8	0,16 %
0,34	197,5	0,17 %
0,41	327,4	0,13 %
0,48	683,7	0,07 %
0,56	946,5	0,06 %

TAB. 6.1 – Comparaison des temps d’exécution de la tâche maître et des tâches esclaves

6.2.3.3 Entrées du programme

Si l’application possède de nombreuses entrées permettant de contrôler le déroulement de l’algorithme, seules trois d’entre elles ont un impact significatif sur le temps d’exécution des tâches qui la composent. Nous considérerons donc uniquement ces trois entrées ($N^V = 3$) qui sont :

- le nombre de particules N^P ,
- le nombre d’intervalles de temps k considérés,
- le nombre de tâches esclaves N^T utilisées.

Le programme admet donc les vecteurs d’entrées de la forme : $E = \{E_1, E_2, E_3\}$ dont les composantes désignent ces trois valeurs respectives.

De plus, nous choisissons de limiter les intervalles de variation de ces entrées :

- $\min_{E_1} = 50000 \quad \max_{E_1} = 500000$
- $\min_{E_2} = 1000 \quad \max_{E_2} = 10000$
- $\min_{E_3} = 4 \quad \max_{E_3} = 12$

Ainsi, on définit l'ensemble \mathbb{E} des entrées admissibles du programme :

$$\mathbb{E} = \left\{ E = \begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix} \quad \text{tels que} \quad \begin{array}{l} 50000 \leq E_1 \leq 500000 \\ 1000 \leq E_2 \leq 10000 \\ 4 \leq E_3 \leq 12 \end{array} \right\}$$

Influence des entrées sur le temps d'exécution des tâches

La figure 6.7 montre l'impact du nombre de particules choisi ainsi que du nombre d'intervalles de temps considérés sur le temps d'exécution des tâches esclaves de l'application. Il apparaît que celui-ci est fortement influencé par ces deux entrées, dans le sens où des valeurs élevées de celles-ci impliquent une forte augmentation du temps d'exécution.

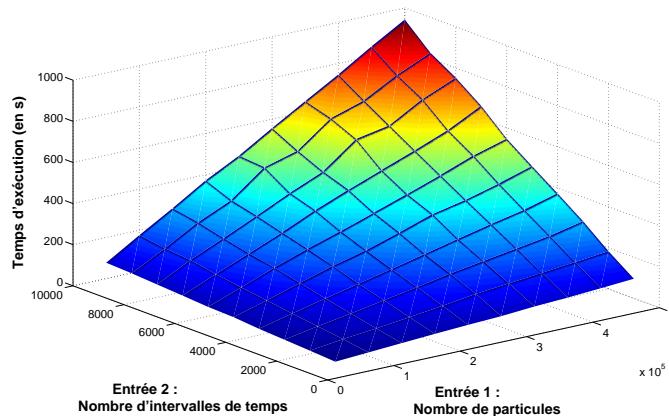


FIG. 6.7 – Temps d'exécution de chaque tâche esclave en fonction du nombre de particules et du nombre d'intervalles de temps considérés

A l'inverse, le nombre de tâches utilisées tend à diminuer le temps d'exécution de chacune d'entre elles (cf. figure 6.8). En effet, à nombre de particules égal, une augmentation du nombre de tâches implique une meilleure répartition des particules, diminuant ainsi la quantité de travail que chaque tâche doit effectuer.

Influence des entrées sur le nombre d'exécutions des blocs de base

Chaque entrée du programme influence le nombre d'occurrences d'un ensemble de blocs de base au sein du chemin d'exécution. Ainsi, la figure 6.9 montre des exemples de bloc de base dont les nombres d'exécutions dépendent de combinaisons de valeurs d'entrées différentes.

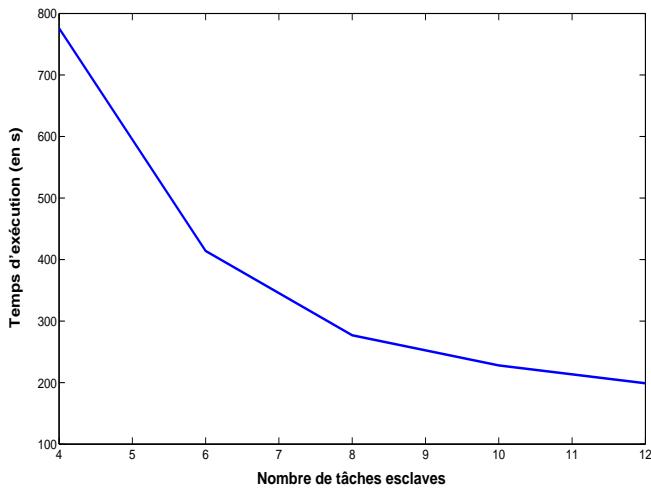


FIG. 6.8 – Temps d’exécution de chaque tâche esclave en fonction du nombre total de tâches

De la même manière que précédemment, la figure 6.10 montre un exemple de bloc dont le nombre d’exécutions dépend du nombre de tâches utilisées. Il apparaît très clairement sur cette figure qu’un nombre important de tâches implique un travail moindre pour chacune d’entre elles. Cependant, ce n’est pas le cas de tous les blocs de base, puisque certains ne dépendent pas de cette entrée. L’analyse du programme montre que les blocs dont l’exécution dépend du nombre de tâches sont ceux dont le nombre d’exécutions varie avec le nombre de particules.

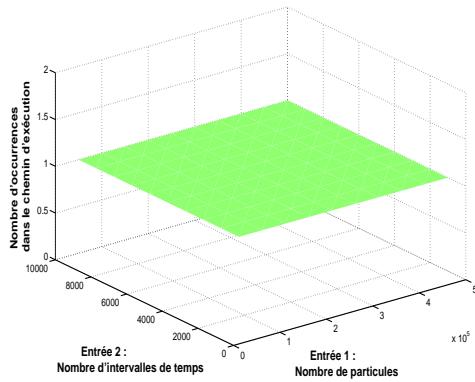
Ainsi, on peut identifier quatre types de blocs de base :

- ceux dont le nombre d’exécutions est constant quelles que soient les entrées,
- ceux dont le nombre d’exécutions dépend à la fois du nombre de particules à traiter, et du nombre de tâches utilisées pour cela (entrées E_1 et E_3),
- ceux dont le nombre d’exécutions dépend uniquement du nombre d’intervalles de temps considérés (entrée E_2),
- ceux dont le nombre d’exécutions dépend de toutes les entrées.

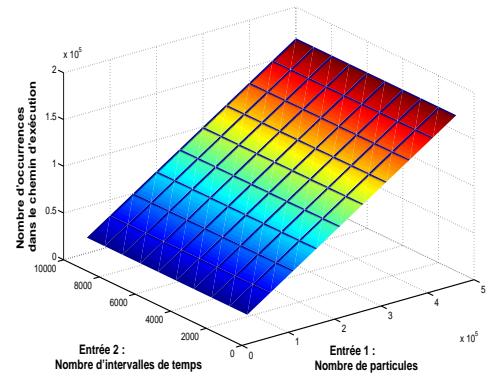
6.2.4 Résultats obtenus

6.2.4.1 Estimation du comportement du programme pour une requête donnée

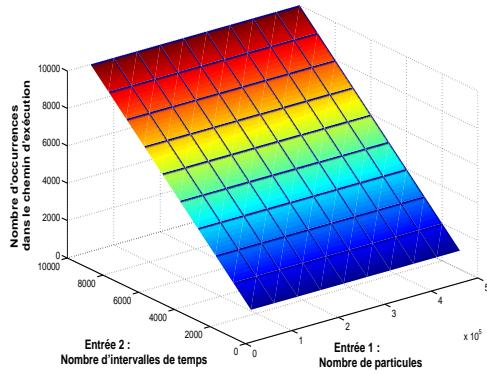
Dans un premier temps, il est indispensable de constituer une base de connaissances. Le programme est donc exécuté, en utilisant des jeux d’entrées différents, autant de fois que cela est nécessaire à l’obtention du nombre d’expériences désiré. Les résultats ci-dessous ont été déterminés grâce à une base contenant 250 expériences, générées aléatoirement, et réparties sur l’ensemble du domaine des entrées admissibles \mathbb{E} du programme.



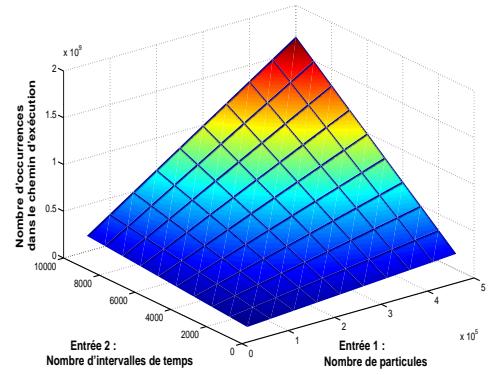
(a) Exemple de bloc de base dont le nombre d'exécutions est constant quelles que soient les entrées



(b) Exemple de bloc de base dont le nombre d'exécutions dépend du nombre de particules



(c) Exemple de bloc de base dont le nombre d'exécutions dépend du nombre d'intervalles de temps



(d) Exemple de bloc de base dont le nombre d'exécutions dépend des deux paramètres

FIG. 6.9 – Nombre d'exécutions de blocs de base en fonction du nombre de particules et du nombre d'intervalles de temps considérés

Une requête E^* est ensuite présentée à la base afin d'obtenir une prédiction sur le comportement de l'application. Nous choisissons, par exemple :

$$E^* = \begin{bmatrix} 250000 \\ 5000 \\ 8 \end{bmatrix}$$

Distance et fonction de pondération

La figure 6.11 représente la distance qui sépare la requête de chacune des expériences contenues dans la base de connaissances. Nous représentons également la valeur du poids $K(d)$ que chacune des expériences aura sur le calcul de l'estimation selon sa distance à la requête. La fonction de pondération est de plus représentée pour plusieurs valeurs du coefficient k (coefficient permettant d'ajuster la largeur de la fonction gaussienne de pondération), afin de mesurer l'impact de ce dernier sur le résultat.

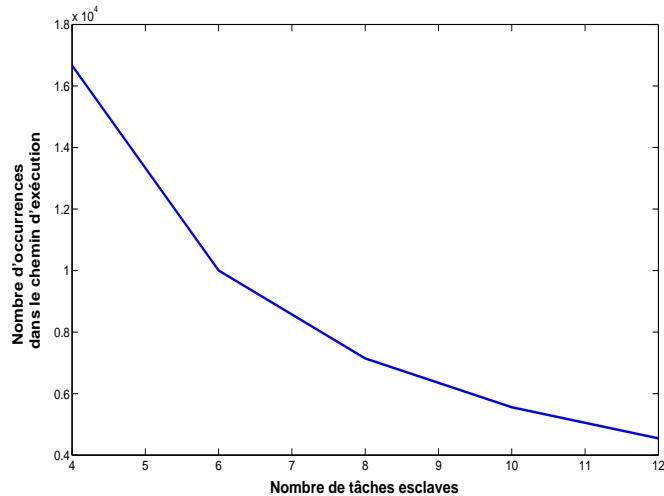


FIG. 6.10 – Exemple de nombre d'exécutions d'un bloc de base en fonction du nombre total de tâches

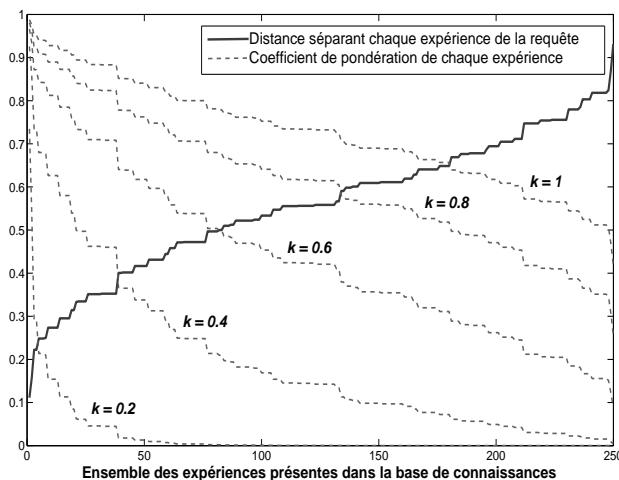


FIG. 6.11 – Distance séparant la requête de chaque expérience, et fonction de pondération associée

La distance moyenne de la requête aux expériences de la base de connaissances est de 0, 56. Elle est comprise entre une valeur minimale de 0, 11 et une valeur maximale de 0, 93. Ainsi, la requête n'est pas contenue dans la base, puisque la distance qui la sépare de la plus proche expérience n'est pas nulle.

La fonction de pondération $K(d)$ permettant de limiter l'influence des expériences éloignées sur l'estimation du nombre d'exécutions des blocs de base, et de privilégier les expériences proches de la requête, atteint son objectif. En effet, elle décroît d'autant plus que la distance entre requête et expériences augmente.

De plus, le coefficient k possède un impact réel sur la pondération. En effet, des valeurs suffisamment faibles de ce coefficient permettent de ne sélectionner que des expériences

proximes de la requête : lorsque k tend vers zéro, seul un nombre restreint d'expériences, si elles existent, possèderont un poids suffisamment fort pour avoir un impact sur le calcul de l'estimation du comportement de l'application. Au contraire, lorsque k tend vers l'infini, toutes les expériences ont la même influence, quelle que soit la distance qui les sépare de la requête.

Estimation du comportement du programme

L'objectif maintenant est d'utiliser le modèle mathématique défini dans ce chapitre pour estimer le nombre d'exécutions des blocs de base du programme pour le jeu d'entrées E^* correspondant à la requête. Pour cela, nous utilisons les 250 expériences de la base de connaissances, et nous fixons : $k = 0,01$.

La figure 6.12 montre une comparaison entre le nombre d'exécutions réel des blocs de base du programme pour le jeu d'entrées E^* et celui estimé par le modèle de prédiction défini. Il est possible de constater que les deux courbes possèdent la même forme. Le modèle de prédiction parvient ainsi à déterminer quels sont les blocs dont le nombre d'exécutions varient en fonction des entrées, de même qu'à estimer les effets des valeurs d'entrées sur ces variations. L'erreur relative moyenne, constatée sur la prédiction de l'ensemble des blocs, s'élève à 8,8% pour le jeu d'entrées considéré.

Le pourcentage d'erreur obtenu dépend de plusieurs paramètres, tels que le nombre d'expériences disponibles dans la base de connaissances, ou bien la distance moyenne séparant la requête de ces expériences. Les sections suivantes proposent d'étudier la précision des prédictions réalisées en fonction de ces paramètres.

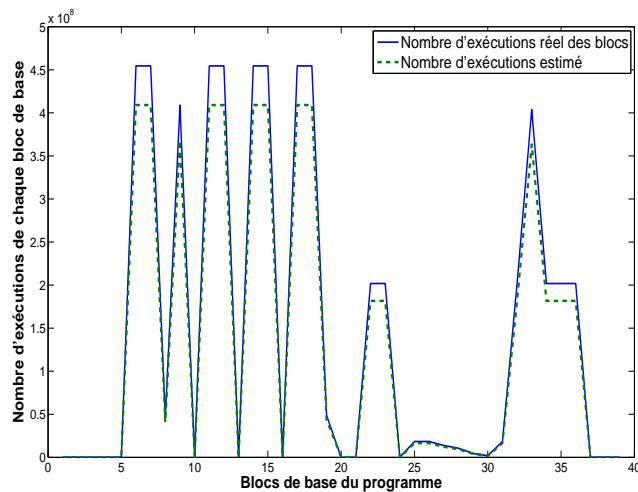


FIG. 6.12 – Comparaison entre le nombre d'exécutions des blocs de base réel et celui estimé par le modèle de prédiction

6.2.4.2 Extension de l'estimation à un ensemble de requêtes

Cette section correspond à une généralisation du cas précédent. Une prédiction est effectuée pour un ensemble de requêtes, afin d'évaluer la constance des résultats obte-

nus précédemment. Une base de connaissances constituée de 250 expériences générées aléatoirement est toujours utilisée. De même, la valeur du coefficient k est toujours fixée à 0,01.

Cette section considère un ensemble \mathbb{E}^* de 500 requêtes réparties sur l'espace d'entrées \mathbb{E} du programme. Les jeux d'entrées de ces requêtes correspondent à toutes les combinaisons des entrées E_1 , E_2 et E_3 dont les variations sont données par les lois suivantes :

- E_1 varie de $\min_{E_1} = 50000$ à $\max_{E_1} = 500000$ par pas de 50000,
- E_2 varie de $\min_{E_2} = 1000$ à $\max_{E_2} = 10000$ par pas de 1000,
- E_3 varie de $\min_{E_3} = 4$ à $\max_{E_3} = 12$ par pas de 2.

Le comportement du programme, c'est-à-dire le nombre d'exécutions de chacun de ses blocs de base, est représenté par la figure 6.13, et ce pour chacune des 500 requêtes. La figure 6.14 montre l'erreur relative obtenue sur l'ensemble des blocs de base du programme pour chaque requête.

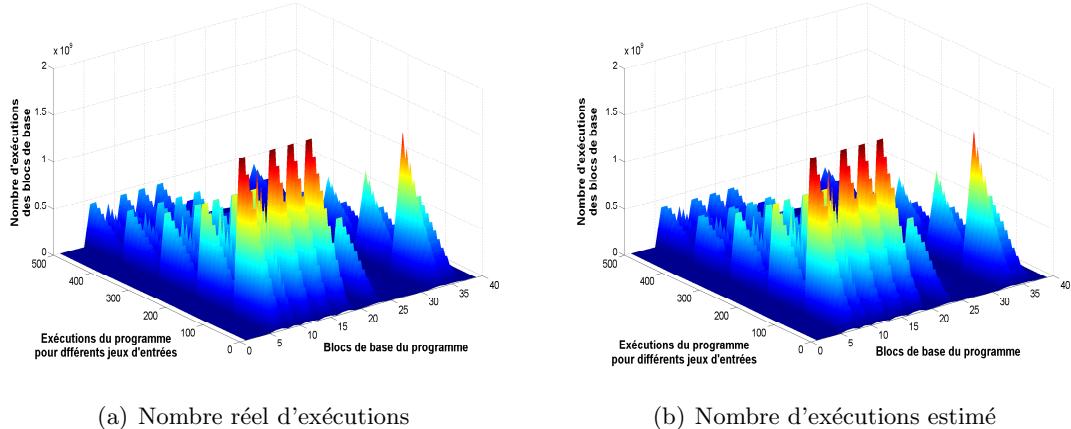


FIG. 6.13 – Nombre d'exécutions des blocs de base du programme pour les 500 jeux d'entrées considérés

L'estimation du nombre d'occurrences de chacun des blocs de base dans le chemin d'exécution semble conforme à la réalité. En effet, les courbes 6.13(a) et 6.13(b) sont tout-à-fait similaires. Cependant, la figure 6.14 met en évidence une inégalité des résultats : certaines requêtes sont mieux estimées que d'autres. L'erreur relative moyenne obtenue, toutes requêtes confondues, est de 6,61%, et s'échelonne de 0% à 48,1%. Il apparaît en outre les résultats suivants :

- 300 requêtes présentent une erreur inférieure à 1% entre le comportement réel du programme et le comportement estimé,
- 40 requêtes présentent une erreur comprise entre 1% et 5%,
- 43 requêtes présentent une erreur comprise entre 5% et 10%,
- 67 requêtes présentent une erreur comprise entre 10% et 25%,
- 50 requêtes présentent une erreur au delà de 25%, dont 19 au dessus de 40%.

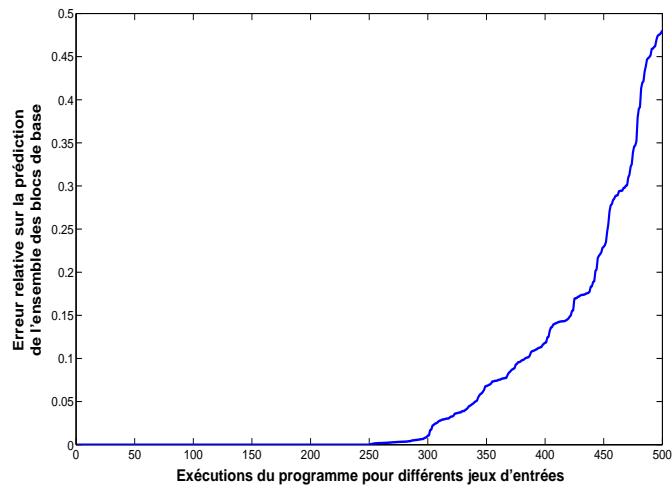


FIG. 6.14 – Erreur relative sur l’ensemble des blocs de base du programme pour chaque requête

Les erreurs obtenues dépendent de la distance séparant les requêtes effectuées des expériences contenues dans la base de connaissances. En effet, on s’aperçoit que la prédiction perd en précision lorsque peu d’expériences ayant des entrées proches de celles des requêtes sont disponibles.

6.2.4.3 Influence du contenu de la base de connaissances

L’objectif est maintenant de déterminer quelle est l’influence du contenu de la base de connaissances sur la précision de l’estimation, en se concentrant plus précisément sur l’influence du nombre d’expériences présentes au sein de celle-ci. Les expériences décrites permettent également d’étudier le lien entre le coefficient k et le degré de remplissage de la base de connaissances.

Pour cela, l’expérience précédente est reprise. Plusieurs séries d’estimations seront effectuées, en utilisant pour chacune d’elles les 500 requêtes \mathbb{E}^* décrites auparavant. Chaque série d’estimations est réalisée plusieurs fois, en faisant varier le nombre d’expériences présentes dans la base de connaissances d’une part, et le coefficient k permettant d’ajuster la largeur de la fonction gaussienne de pondération d’autre part.

La figure 6.15 représente l’erreur relative globale relevée pour chaque série de requêtes en fonction du nombre d’expériences présentes dans la base de connaissances et du coefficient k . Cette courbe met en avant une progression régulière de l’erreur observée, pouvant être nulle si les paramètres testés sont bien ajustés, mais pouvant également dépasser 200% si ce n’est pas le cas. Il apparaît ainsi que la précision du modèle défini dans ce chapitre dépend fortement du choix de ces paramètres.

Afin d’étudier de manière plus approfondie l’influence du degré de remplissage de la base de connaissances et celle du coefficient k , nous réalisons deux coupes de la vue en trois dimensions de la figure 6.15. Ces graphiques sont donnés en figure 6.16.

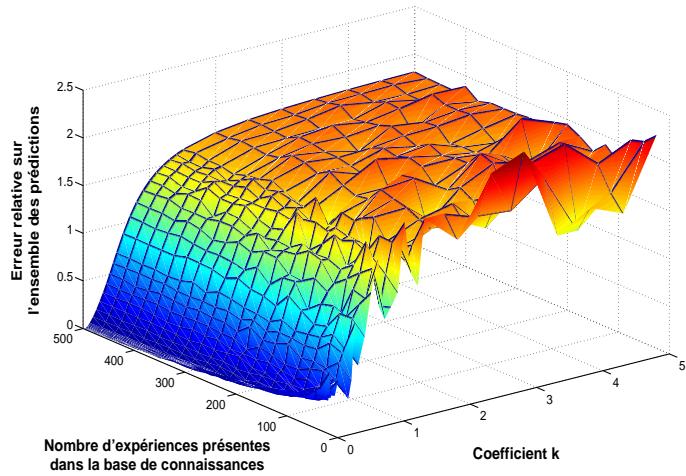
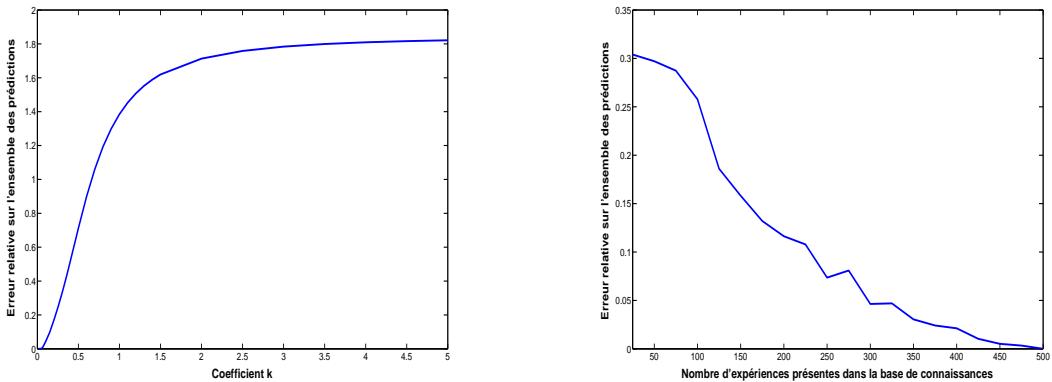


FIG. 6.15 – Erreur relative sur chaque série de 500 requêtes en fonction du nombre d’expériences présentes dans la base et du coefficient k



(a) Erreur sur les prédictions en fonction du coefficient k (500 expériences présentes dans la base de connaissances)

(b) Erreur sur les prédictions en fonction du nombre d’expériences présentes dans la base de connaissances ($k = 0.02$)

FIG. 6.16 – Coupes de la vue 3D

Pour une base de connaissances contenant suffisamment d’expériences, une valeur trop importante du coefficient k implique une erreur importante sur les prédictions (figure 6.16(a)), car elle favorise la perturbation de la prédiction par des expériences parasites. Dans la pratique, des valeurs de k comprises entre 0,01 et 0,1 offrent de bons résultats (cf. figure 6.17). Dans ce cas, même si la base est peu remplie, l’erreur globale ne dépasse pas 40%.

De plus, la courbe 6.16(b) montre que la précision de la prédiction de comportement repose essentiellement sur la base de connaissances. Un grand nombre d’expériences offre une précision accrue, puisque la probabilité de trouver une ou plusieurs expériences proches de la requête est plus élevée. En pratique, pour l’exemple d’application traité, et pour le domaine de variation des valeurs d’entrées considéré, il semble qu’un minimum de 150 expériences soit nécessaire pour obtenir des prédictions suffisamment précises.

Cependant, le manque d'expériences peut être compensé, dans la mesure du possible, par une augmentation du coefficient k , dans le but d'élargir le nombre d'expériences prises en compte dans le processus de prédition. En effet, la figure 6.18 montre que, pour de faibles valeurs de k , la courbe possède une forme de creux : l'erreur obtenue diminue tout d'abord, avant d'augmenter, lorsque la valeur k croît.

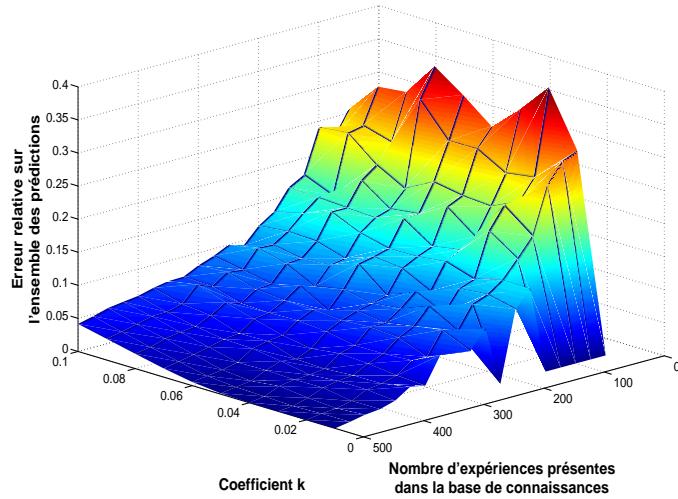
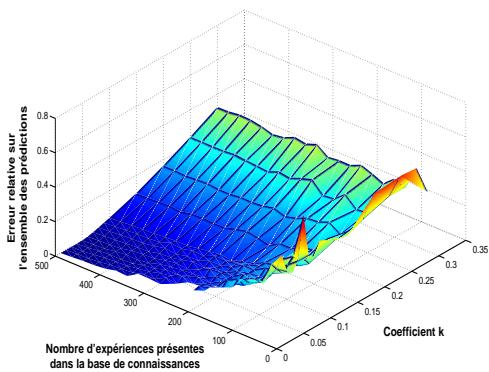
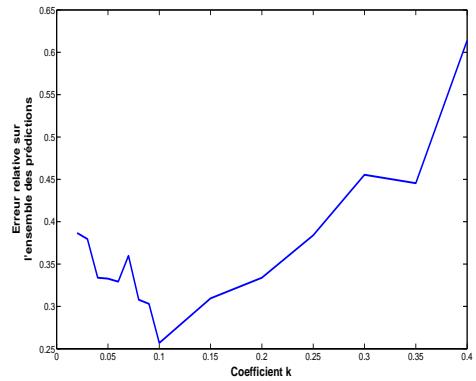


FIG. 6.17 – Domaine de valeurs de k offrant une précision correcte sur chaque série de requêtes



(a) Erreur sur les prédictions en fonction du coefficient k et du nombre d'expériences présentes dans la base



(b) Erreur sur les prédictions en fonction du coefficient k (75 expériences dans la base de connaissances)

FIG. 6.18 – Représentation du creux d'erreur obtenu

6.3 Prise en compte de l'impact relatif des entrées du programme sur le nombre d'exécutions des blocs de base

6.3.1 Positionnement du problème

Le modèle de prédiction utilisé jusqu'ici s'appuie sur l'expression (6.3). Dans ce contexte, l'estimation du nombre d'exécutions de chacun des blocs de base est conditionnée par la distance séparant la requête des expériences contenues dans la base de connaissances. De plus, cette distance est calculée à partir des différentes valeurs prises par les entrées du programme (équations (6.1) et (6.2)).

Cette formulation du modèle suppose que toutes les valeurs d'entrées du programme traité ont une influence similaire sur le nombre d'exécutions de chacun des blocs de base du programme. Cette hypothèse est bien sûr fausse, comme le montre la figure 6.9.

Considérons les blocs de base du programme de filtrage particulaire dont le nombre d'exécutions ne dépend que du nombre d'intervalles de temps considérés par l'algorithme (entrée E_2). On définit deux expériences contenues dans la base de connaissances ainsi que la requête :

$$\mathbb{E}^{Exp} = \left\{ E^{(1)} = \begin{bmatrix} 100000 \\ 10000 \\ 4 \end{bmatrix}, \quad E^{(2)} = \begin{bmatrix} 500000 \\ 1000 \\ 12 \end{bmatrix} \right\} \quad E^* = \begin{bmatrix} 100000 \\ 1000 \\ 4 \end{bmatrix}$$

D'après les valeurs numériques des entrées ci-dessus, il est possible de déduire :

$$D(E^*, E^{(1)}) < D(E^*, E^{(2)})$$

Ainsi, le modèle privilégiera l'expérience $E^{(1)}$ dans l'estimation du nombre d'exécutions de l'ensemble des blocs de base de la requête. Cependant, les blocs qui ne dépendent que de la valeur d'entrée E_2 seraient mieux estimés si l'expérience $E^{(2)}$ était utilisée à la place de $E^{(1)}$.

La précision du modèle de prédiction peut ainsi être améliorée en tenant compte de l'impact relatif des entrées du programme sur le nombre d'exécutions des blocs de base, et ce de manière individuelle pour chaque bloc et chaque entrée. L'idée est ainsi de pouvoir limiter l'influence de certaines valeurs d'entrées sur l'estimation du nombre d'exécutions d'un ensemble de blocs.

6.3.2 Reformulation du modèle de prédiction

La notion de distance séparant deux expériences doit être modifiée afin d'intégrer au modèle cet aspect de dépendance des blocs vis-à-vis des entrées. Ainsi, cette dépendance sera exprimée au sein de la fonction de distance définie.

Soit $w_{v,b}$ le poids de la valeur d'entrée v sur le calcul de la distance pour le bloc b . On pose la contrainte suivante :

$$\forall b \in \mathbb{BB} \quad , \quad \forall v \in \{1, \dots, N^V\} \quad 0 \leq w_{v,b} \leq 1$$

On définit la nouvelle fonction de distance séparant deux entrées :

$$\forall b \in \mathbb{BB} \quad , \quad \forall E^{(1)} \in \mathbb{E} \quad , \quad \forall E^{(2)} \in \mathbb{E} \quad D_b(E^{(1)}, E^{(2)}) = \sqrt{\sum_{v=1}^{N^V} w_{v,b} \cdot d(E_v^{(1)}, E_v^{(2)})^2} \quad (6.4)$$

Cette nouvelle distance peut désormais posséder des valeurs différentes selon les blocs de base du programme qui sont considérés. Ainsi, si $w_{v,b} = 0$, la distance $D_b(E^{(1)}, E^{(2)})$ est nulle, et par conséquent la valeur d'entrée v n'a aucune influence sur l'estimation du nombre d'occurrences du bloc b dans le chemin d'exécution. En revanche, pour $w_{v,b} = 1$, l'influence de l'entrée v sur le bloc b est au plus fort. Il est également possible de choisir des valeurs intermédiaires $w_{v,b}$, comprises entre 0 et 1, ce choix dépendant entièrement de la nature du programme. La métrique hétérogène de distance $d(E_v^{(1)}, E_v^{(2)})$ est quant à elle conservée à l'identique (cf. équation (6.2)).

L'équation sur laquelle se fonde la prédiction doit maintenant être modifiée, afin de prendre en compte la nouvelle distance définie. Soit E^* le vecteur d'entrées correspondant à la requête, le nombre d'exécutions de chacun des blocs de base étendus maximaux b du programme est estimé de la manière suivante :

$$\forall b \in \mathbb{BB} \quad N_b^{BB}(E^*) = \frac{\sum_{E \in \mathbb{E}^S} [K(D_b(E^*, E)) \cdot N_b^{BB}(E)]}{\sum_{E \in \mathbb{E}^S} K(D_b(E^*, E))} \quad (6.5)$$

6.3.3 Annotations du code source d'un programme

6.3.3.1 Objectifs et principe

Si la connaissance de la dépendance des blocs de base vis-à-vis des entrées du programme permet une meilleure précision du modèle, il est en revanche nécessaire de trouver un moyen de choisir les valeurs à donner au paramètre $w_{v,b}$. Le choix de ces valeurs dépend entièrement de la structure du programme. Ainsi, la personne la plus à même d'effectuer un tel choix est sans conteste le développeur de l'application.

Les annotations sont ainsi un moyen souple et facile de permettre au programmeur de renseigner les valeurs de $w_{v,b}$, sans se soucier des détails du modèle mathématique, dont il n'a pas nécessairement la connaissance. Le principe est simple : il s'agit d'exprimer, dans le code source du programme, les dépendances des instructions, ou séquences d'instructions, vis-à-vis de ses entrées.

6.3.3.2 Mise en œuvre

Spécifications

Les annotations décrites dans cette section concernent les programmes développés en C, puisqu'il s'agit du langage choisi de fait, de par l'utilisation des outils de profiling *gprof* et *gcov*.

Les spécifications de la mise en place des annotations sont simples :

- permettre à l'utilisateur de donner la dépendance du nombre d'exécutions des blocs vis-à-vis des entrées du programme,
- ne nécessiter aucune connaissance du modèle mathématique de prédiction de comportement d'applications défini dans cette thèse,
- être insérables directement dans le code source du programme,
- ne pas empêcher la compilation ni le bon fonctionnement en exécution de l'application.

Les deux derniers points sont satisfait si l'on a recours à des directives de compilation **#pragma**. Ces directives permettent au programmeur de dialoguer avec le compilateur qu'il utilise afin de personnaliser la phase de compilation. Toutes sortes de directives **#pragma** sont définies par les différents compilateurs existants. Si une directive n'est pas comprise par un compilateur, celui-ci choisit de l'ignorer.

Ainsi, l'utilisation de telles directives permet d'introduire, de manière totalement transparente, les annotations au sein du code source. Un outil permettant de lire et d'interpréter les annotations sous forme de directives **#pragma** a été développé, et les compilateurs standards, tels que *gcc*, n'en tiennent pas compte.

Les annotations définies dans cette section sont soumises à la syntaxe suivante :

```
#pragma etp annotation (paramètres)
```

où :

- **etp** signifie *execution time prediction*. Ce mot-clé caractérise donc l'ensemble des annotations que nous définissons, de manière à ce qu'elles soient immédiatement identifiables par l'outil mis en place pour les interpréter.
- **annotation** désigne le type d'annotations. Il existe quatre types différents d'annotations, décrits ci-dessous.
- **(paramètres)** est une liste de paramètres facultatifs, séparés par des virgules, et placés entre parenthèses.

Description des annotations

Nous définissons quatre types d'annotations :

- **Annotations de type “inputs”** : Ce type d'annotations permet de définir les entrées considérées dans la prédiction du temps d'exécution. De telles annotations peuvent être insérées au début du code source, au côté des traditionnelles directives **#define**.
- **Annotations de type “begin dependency”** : Ce type d'annotations débute une séquence d'instructions dont le nombre d'exécutions dépend d'entrées du programme. La liste de ces entrées, telles que définies par la directive “inputs”, est spécifiée en paramètres.

- **Annotations de type “begin independency”** : Ce type d’annotations débute une séquence d’instructions dont le nombre d’exécutions ne dépend d’aucune entrée du programme.
- **Annotations de type “end”** : Ce type d’annotations clôt toute séquence d’instructions, faisant ainsi suite aux annotations de type “begin dependency” ou “begin independency”.

Utilisation des annotations

Tout d’abord, il est nécessaire de définir les entrées du programme prises en compte par le modèle de prédiction. Ceci s’effectue par le biais des annotations de type “inputs”. Par exemple, la ligne suivante peut être ajoutée au programme de filtrage particulaire, dans le but de définir les trois entrées considérées :

```
#pragma etp inputs (particules, intervalles_temps, taches_esclaves)
```

Ensuite, à l’intérieur d’une fonction, il est possible d’indiquer qu’une séquence d’instructions dépend d’une ou plusieurs entrées. L’exemple suivant définit une boucle dont le nombre d’itérations dépend du nombre de particules :

```
#pragma etp begin dependency (particules)
for (int i = 0; i < particules; i++) {
    instructions;
}
#pragma etp end
```

Ainsi, le nombre d’exécutions des blocs de base liés à cette boucle dépend de l’entrée **particules**. Le paramètre $w_{v,b}$ associé à cette entrée et à ces blocs prend donc, grâce à cette annotation, la valeur 1.

Il est également possible d’imbriquer les blocs de dépendance et d’indépendance entre eux. Par exemple :

```
#pragma etp begin dependency (particules)
for (int i = 0; i < particules; i++) {
    instructions_1;
    #pragma etp begin dependency (intervalles_temps)
    for (int j = 0; j < intervalles_temps; j++) {
        instructions_2;
    }
    #pragma etp end
    instructions_3;
}
#pragma etp end
```

Ainsi, les blocs d’instructions 1 et 3 ne dépendent que du nombre de particules, tandis que le bloc d’instructions 2 dépend à la fois du nombre de particules et du nombre d’intervalles de temps.

Enfin, il est à noter que, par défaut, c'est-à-dire sans la présence d'annotations spécifiques au sein du programme, il est considéré que tous les blocs de base dépendent de toutes les entrées, c'est-à-dire en termes de modèle :

$$\forall b \in \mathbb{BB} \quad , \quad \forall v \in \{1, \dots, N^V\} \quad w_{v,b} = 1$$

Ceci permet aux développeurs de s'affranchir de la mise en place des annotations s'ils le désirent, tout en laissant la possibilité au module de prédiction de comportement de fonctionner de manière correcte.

6.3.4 Résultats obtenus

Les essais que nous effectuons ici possèdent les mêmes caractéristiques que précédemment, afin de comparer les résultats obtenus sans et avec l'amélioration due à la prise en compte de l'impact relatif des entrées sur le nombre d'exécutions des blocs de base.

Dans un premier temps, nous considérons une base de connaissances contenant 250 expériences générées aléatoirement, et réparties sur l'ensemble de l'espace d'entrées du programme, à laquelle 500 requêtes sont soumises. Le coefficient k conditionnant la largeur de la fonction gaussienne de pondération est fixé à 0,01.

Les résultats précédents montraient que, sans amélioration, une erreur moyenne de 6,61% était obtenue sur l'ensemble des requêtes. L'erreur associée à chaque expérience était représentée par la figure 6.14. Elle est également représentée par la figure 6.19 ci-dessous, accompagnée de la courbe associée à la prédiction utilisant le modèle amélioré.

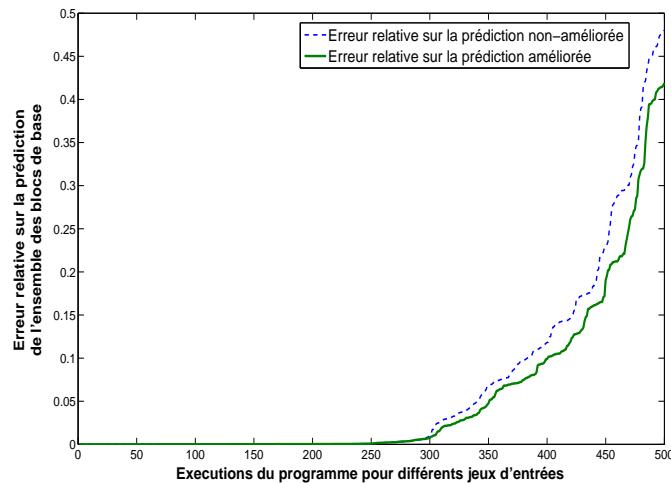


FIG. 6.19 – Comparaison des erreurs obtenues sans et avec amélioration du modèle de prédiction

Il apparaît que les modifications apportées au modèle pour prendre en compte l'impact des entrées sur les blocs ont permis d'améliorer la précision des estimations réalisées. L'erreur moyenne obtenue sur la globalité des requêtes est passée à 5,38%.

Ces résultats sont confirmés par l’expérience suivante, pour laquelle plusieurs séries de 500 requêtes sont soumises à la base de connaissances, chacune des séries correspondant à un nombre d’expériences présentes dans la base ainsi qu’à une valeur de k différente. La figure 6.20 obtenue avec le modèle amélioré peut être comparée avec les figures 6.15 et 6.17. Cette comparaison montre également une amélioration des performances du modèle de prédiction. L’erreur moyenne relevée sur la globalité des séries de requêtes est de seulement 5,90%, tandis qu’elle s’élevait à 7,47% dans la version initiale du modèle.

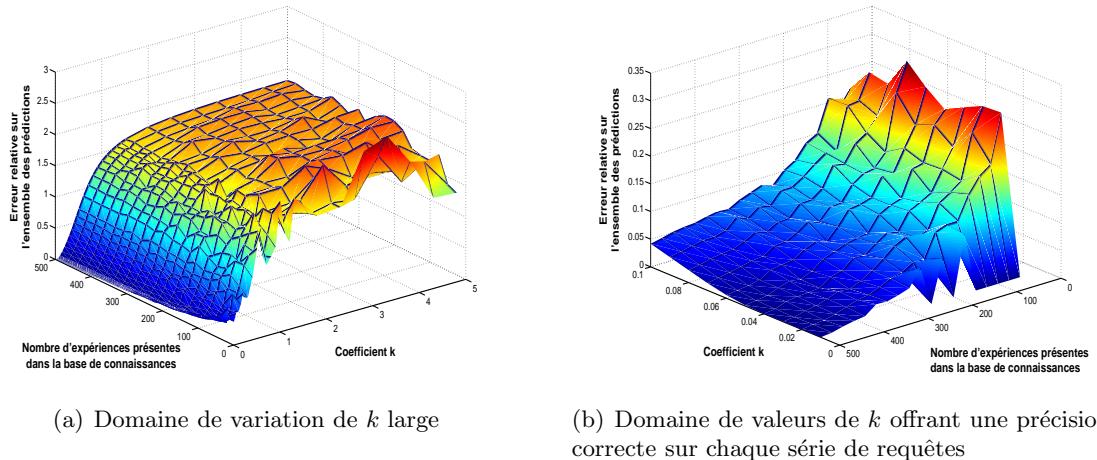


FIG. 6.20 – Erreur relative sur chaque série de 500 requêtes en fonction du nombre d’expériences présentes dans la base et du coefficient k pour la version améliorée du modèle de prédiction

6.4 Modèle complet de prédiction hybride de temps d’exécution

6.4.1 Préambule

Cette section montre la mise en œuvre complète de l’approche hybride de prédiction du temps d’exécution d’applications introduite dans cette thèse, telle que décrite dans le chapitre 4. Elle examine les résultats obtenus en se basant sur l’application de filtrage particulaire décrite dans la section 6.2.3.

Le modèle complet est utilisé : le temps d’exécution de chacun des blocs de base du programme est déterminé en s’appuyant sur la résolution itérative du système d’équations formé par les données issues des expériences présentes dans la base de connaissances (chapitre 5). Le comportement du programme est ensuite estimé à l’aide d’une approche basée sur des instances (chapitre 6). La figure 6.21 décrit la mise en pratique de l’approche hybride.

A l’issue de ces deux phases, le temps d’exécution de l’application est déterminé. Il est ensuite comparé avec celui donné par une approche classique de prédiction de temps d’exécution basée sur des instances, telle que proposée dans [13, 86], afin de prouver la validité de l’approche hybride développée dans le cadre de cette thèse.

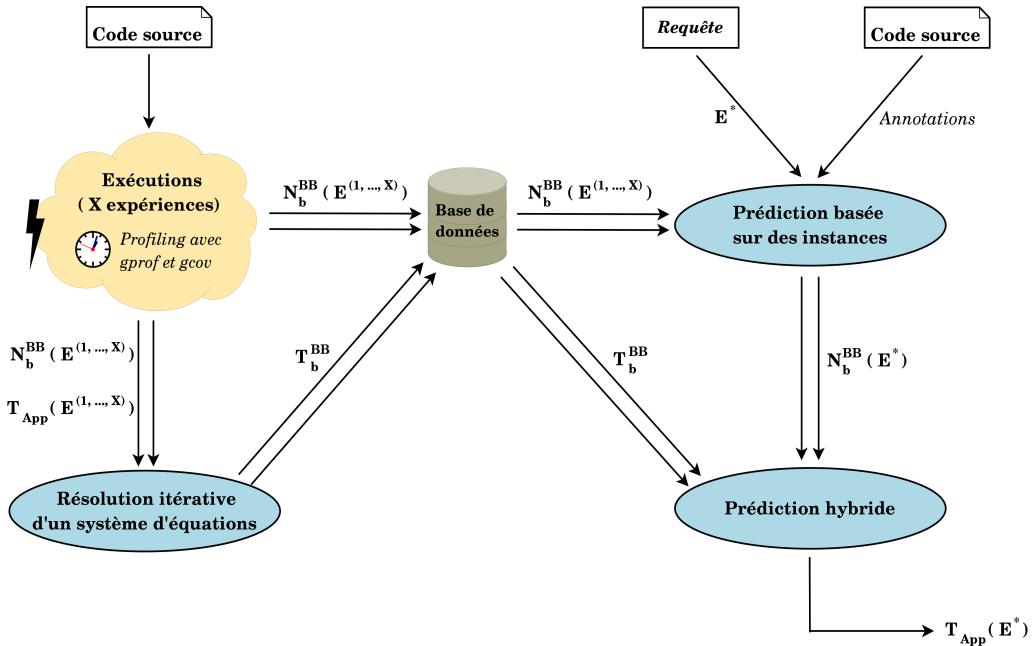


FIG. 6.21 – Mise en œuvre de l’approche hybride de prédiction de temps d’exécution

6.4.2 Prédiction du temps d’exécution d’un ensemble de requêtes

Le temps d’exécution de l’application de filtrage particulaire est déterminé à l’aide de l’approche hybride introduite dans cette thèse, en utilisant une base de connaissances contenant initialement 350 expériences. Ces expériences aléatoirement générées sont réparties sur l’ensemble de l’espace des entrées admissibles.

Un ensemble \mathbb{E}^* de 250 requêtes est alors présenté au module de prédiction, afin d’en déterminer les temps d’exécution. Ces requêtes sont choisies aléatoirement sur l’ensemble de l’espace des entrées admissibles.

La figure 6.22 met en parallèle le temps d’exécution réel du programme, ainsi que le temps déterminé par l’approche hybride de prédiction. Il apparaît que les deux courbes se confondent, le temps estimé étant proche du temps réellement mesuré, et ce quelles que soient les entrées du programme. En moyenne, on obtient une erreur de 8,70% entre ces deux valeurs.

La détermination du temps d’exécution de ces mêmes requêtes à partir d’une approche classique basée sur des instances produit une erreur moyenne de 12,7%. Il apparaît ainsi que l’approche hybride présentée dans cette thèse donne de meilleurs résultats que l’approche classique de prédiction de temps d’exécution.

6.4.3 Variation du nombre d’expériences contenues dans la base de connaissances

Nous faisons maintenant varier le nombre d’expériences présentes au sein de la base de connaissances, afin de pouvoir observer le profil des résultats obtenus en fonction de la taille de la base utilisée. Au minimum, la base de connaissances contiendra 200 expériences, afin d’avoir des résultats significatifs. Ensuite, son contenu sera augmenté, jusqu’à atteindre

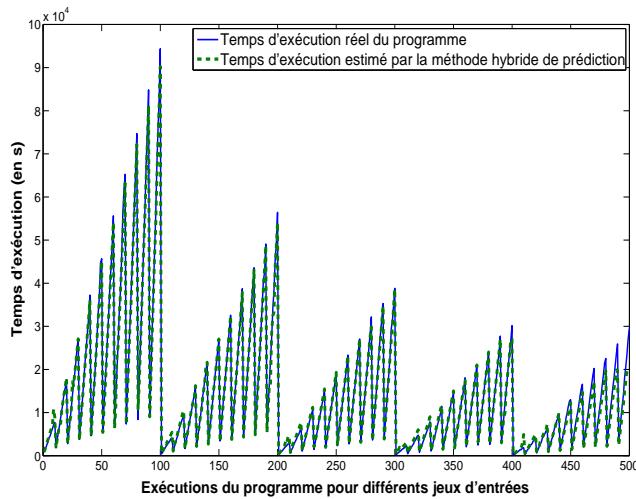


FIG. 6.22 – Temps d’exécution réel et prédit du programme pour différents jeux d’entrées

500 expériences. Un ensemble de 250 requêtes choisies aléatoirement sur l’ensemble de l’espace des entrées admissibles est toujours utilisé, quelle que soit la taille de la base.

La figure 6.23 représente l’erreur moyenne obtenue sur l’ensemble des 250 requêtes en fonction du nombre d’expériences contenues dans la base de connaissances. Les données numériques correspondantes sont présentées dans le tableau 6.2. Il apparaît ainsi que l’erreur obtenue est d’autant plus faible que le contenu de la base de connaissances est important. La courbe observée n’est cependant pas parfaitement lisse : les nombreuses fluctuations locales indiquent que la qualité des prédictions réalisées ne dépend pas uniquement du nombre d’expériences présentes dans la base, mais aussi de leur répartition sur l’ensemble (E) des entrées admissibles du programme (les expériences sont générées aléatoirement, ce qui ne leur garantit pas une répartition optimale sur l’ensemble (E)).

De plus, il est possible de remarquer que l’approche hybride donne de meilleurs résultats que l’approche classique de prédiction de temps d’exécution, et ce quel que soit le contenu de la base de connaissances. Ainsi, la prise en compte de la structure du programme permet d’améliorer la prédiction de son temps d’exécution.

	<i>Taille de la base de connaissances</i>						
	200	250	300	350	400	450	500
Approche hybride	19,9%	15,4%	11,5%	8,70%	7,03%	5,88%	4,53%
Approche classique	23,1%	19,0%	15,0%	12,7%	11,2%	10,3%	9,06%

TAB. 6.2 – Erreur relative sur l’ensemble des prédictions (250 requêtes) en fonction du nombre d’expériences contenues dans la base de connaissances

6.5 Conclusion

Dans ce chapitre, nous avons défini une méthode permettant d’estimer le comportement d’un programme pour un jeu d’entrées données, en s’appuyant sur un historique

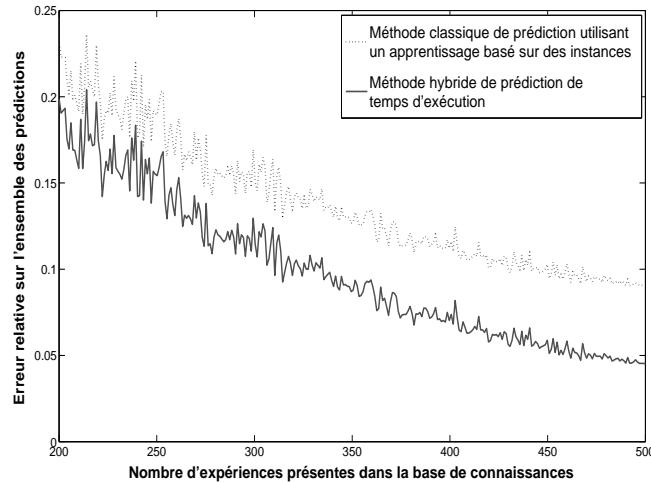


FIG. 6.23 – Erreur relative sur l’ensemble des prédictions (250 requêtes) en fonction du nombre d’expériences contenues dans la base de connaissances

d’exécutions passées. La notion de comportement correspond ici au nombre d’occurrences des blocs de base du programme au sein du chemin d’exécution.

Cette estimation est réalisée en employant une approche basée sur des instances, permettant d’effectuer une moyenne pondérée des données présentes dans une base de connaissances pour produire le résultat attendu. La pondération dépend alors de la distance séparant les expériences contenues dans la base de la requête pour laquelle l’estimation doit être effectuée.

De plus, une méthode permettant d’améliorer l’estimation a été retenue. Il s’agit d’annotationer le code source du programme considéré, dans le but d’indiquer l’influence que doivent posséder les entrées sur le calcul de chacun des blocs de base. Un système d’annotations a par ailleurs été développé pour être utilisé dans tout programme écrit en C ou en C++.

Les résultats obtenus par cette méthode sont très encourageants. Ils mettent en avant une estimation juste, et donc exploitable, du comportement d’un programme. Les études expérimentales réalisées ont notamment montré que le nombre d’expériences présentes dans la base de connaissances a un fort impact sur la précision des estimations. De même, le coefficient permettant d’ajuster la largeur de la fonction gaussienne de pondération agit également sur cette précision.

Enfin, ce chapitre nous a permis de mettre en place la méthode hybride de prédiction de temps d’exécution dans son ensemble. Ainsi, à partir d’une base de connaissances, nous avons pu à la fois déterminer les temps d’exécution des blocs de base du programme, et en estimer le nombre d’occurrences dans le chemin d’exécution, et ce pour différents jeux d’entrées. Grâce à ces deux données, le temps d’exécution de l’application a pu être déterminé pour ces jeux d’entrées.

Il apparaît que les résultats produits par cette approche sont satisfaisants. En effet, le cumul des erreurs obtenues au cours des deux phases préliminaires n’implique pas une

explosion de l'erreur sur le résultat final. Au contraire, il apparaît que cette erreur est d'autant plus faible qu'un nombre important d'expériences est contenu dans la base de connaissances. Ainsi, la constitution d'une bonne base de connaissances conditionne entièrement le bon fonctionnement de l'approche hybride de prédiction définie. Enfin, les prédictions réalisées montrent des résultats qui améliorent ceux obtenus avec une approche classique d'estimation de temps d'exécution basée sur des instances, telle que décrite dans [13, 86].

Chapitre 7

Conclusion

Depuis quelques années, l'augmentation de la puissance des stations de travail semble ne connaître aucune limite. Cependant, de nombreuses applications scientifiques ou industrielles ont des besoins de plus en plus importants en termes de puissance de calcul, si bien que, même si la puissance des stations de travail augmente, une seule d'entre elles ne suffit pas pour exécuter ces applications. Le calcul sur grille permet de répondre à ce problème en répartissant la charge de travail sur plusieurs machines.

Cette thèse s'est intéressée au domaine de la gestion des ressources présentes sur les grilles de calcul. Elle propose un système de gestion basé sur l'utilisation de critères économiques, ce qui permet de réguler la demande sur les ressources.

L'algorithme d'ordonnancement proposé dans le cadre de cette thèse, tout comme l'ensemble des algorithmes existants, repose sur la connaissance, *à priori*, de la durée d'exécution des applications soumises. Par conséquent, une partie des travaux menés permet d'apporter une réponse au problème de la prédiction de performances d'applications parallèles.

Nous avons volontairement choisi de nous placer dans le cadre de la plate-forme nationale de recherche Grid'5000. Plusieurs raisons ont motivé ce choix, telles que la possibilité d'avoir accès à une véritable grille de calcul à des fins expérimentales, ou encore l'extrême flexibilité de l'outil proposé. Ainsi, le gestionnaire de ressources OAR a naturellement servi de base au système de gestion des ressources proposé dans ces travaux.

7.1 Prédiction de comportement d'applications parallèles

Une partie de cette thèse a permis d'étudier une méthode de prédiction de performances destinée aux applications parallèles candidates à une exécution sur un support de grille. Cette méthode peut cependant s'appliquer à d'autres types d'applications, comme les applications séquentielles.

L'approche hybride de prédiction de temps d'exécution décrite dans le chapitre 4 tient son nom du fait qu'elle combine deux méthodes de prédiction existantes :

- une méthode de prédiction utilisant un historique d'exécutions passées et s'appuyant sur un apprentissage basé sur des instances,
- une méthode basée sur le profil des applications, telle qu'elle peut être appliquée dans le cadre de la détermination du WCET d'applications temps réel.

Cette approche nécessite de déterminer deux types d'informations relatives aux programmes :

- Les **informations temporelles**, c'est-à-dire le temps d'exécution de chacun des blocs de base. Le chapitre 5 montre que ces informations peuvent être obtenues à partir de la résolution d'un système d'équations linéaires obtenu à partir d'un jeu d'exécutions du programme. Ce système étant mal conditionné, les méthodes de résolution standards que nous avons testées se sont avérées peu efficaces. Nous avons ainsi mis au point une méthode de résolution de ce système d'équations par itérations successives.
- Les **informations comportementales**, c'est-à-dire le nombre d'exécutions de chacun des blocs de base. Le chapitre 6 montre comment les estimer à partir d'un historique d'exécutions passées, en mettant en œuvre une approche d'apprentissage basé sur des instances. De plus, ce chapitre définit également un système d'annotations du code source des programmes permettant d'améliorer cette estimation.

Le modèle complet de prédiction hybride a été testé en fin de chapitre 6. Il montre des résultats satisfaisants, qui améliorent ceux obtenus avec une approche traditionnelle de prédiction. En effet, l'approche hybride permet de prendre en compte la structure du programme. Il en résulte une prédiction plus fine, puisqu'il est possible d'identifier les portions du programme qui ont un temps d'exécution important et de les relier directement à un ensemble d'entrées du programme.

7.2 Placement à l'aide de modèles économiques sur une grille de calcul

Un **modèle économique pour la gestion des ressources d'une grille de calcul** a été défini dans le chapitre 3. Il permet de déterminer un placement pour des applications parallèles à gros grain. La politique d'ordonnancement s'applique à des grilles dont les ressources sont dédiées (comme cela est le cas pour Grid'5000, ou pour les grilles de type ASP), et qui fonctionnent en mode partagé (plusieurs applications peuvent être exécutées simultanément sur une même ressource).

Le modèle économique proposé est un modèle de marché, pour lequel le prix des différentes ressources peut varier au cours du temps. Ainsi, déterminer un placement revient à :

- choisir les ressources sur lesquelles exécuter l'application soumise,
- choisir une date de début pour cette application.

L'objectif d'un tel placement est de minimiser un compromis entre la date de fin de l'application (c'est-à-dire son temps de réponse), et son coût. Les proportions du compromis sont spécifiées par l'utilisateur, celui-ci pouvant choisir une exécution rapide (en ne se préoccupant pas, ou peu, du coût de l'application), ou bien une exécution économique (mais potentiellement longue). Il en résulte un problème d'optimisation non-linéaire, sous contraintes, et à variables entières.

Cette politique d'ordonnancement a été implémentée par un algorithme génétique et intégrée au sein du gestionnaire de ressources OAR. Ainsi, nous avons développé un

système de gestion de ressources complet et fonctionnel, permettant de placer des applications parallèles sur une grille de calcul, en fonction de critères économiques. Il intègre en outre une méthode de prédiction de charge des ressources, permettant d'estimer ses variations futures en fonction des applications soumises.

Notons enfin que l'intégration du modèle économique à OAR a été réalisée avec un souci permanent d'évolutivité. En effet, il paraît indispensable que notre système de gestion de ressources puisse suivre les évolutions d'OAR, qu'il s'agisse de mises-à-jour fonctionnelles ou d'ajouts de correctifs de sécurité. Ainsi, l'implémentation de notre modèle ne modifie pas OAR, elle y ajoute simplement une surcouche.

7.3 Vue d'ensemble

Les deux parties de la thèse sont loin d'être indépendantes. En effet, le calcul d'un ordonnancement nécessite de disposer à l'avance de plusieurs informations :

- *le nombre et la puissance des processeurs* : ces informations se trouvent dans la base de données d'OAR,
- *une estimation de la charge future des processeurs* : le modèle économique proposé intègre une méthode de prédiction de cette charge,
- *les paramètres de coût* : ils sont fixés par les administrateurs de la grille, et se trouvent dans la base de données,
- *les préférences sur les proportions du compromis temps / coût* : ils sont fournis par l'utilisateur,
- *les caractéristiques de l'application (nombre de tâches et temps d'exécution prévu)* : elles sont fournies par l'utilisateur.

La durée de l'application soumise peut être estimée par l'utilisateur. Dans la pratique, il a été démontré que ceci est une source d'erreur, et peut dégrader les performances de l'ordonnanceur. Il est donc utile d'automatiser cette estimation. La figure 7.1 montre comment les deux parties de la thèse s'intègrent l'une à l'autre.

Lorsque l'utilisateur soumet une application, le temps d'exécution de l'application est estimé à l'aide de l'approche hybride de prédiction de temps d'exécution. Le gestionnaire de ressources peut alors calculer un ordonnancement grâce au modèle économique proposé. Lorsque l'application est exécutée, elle sert de nouvelle expérience venant enrichir la base de connaissances.

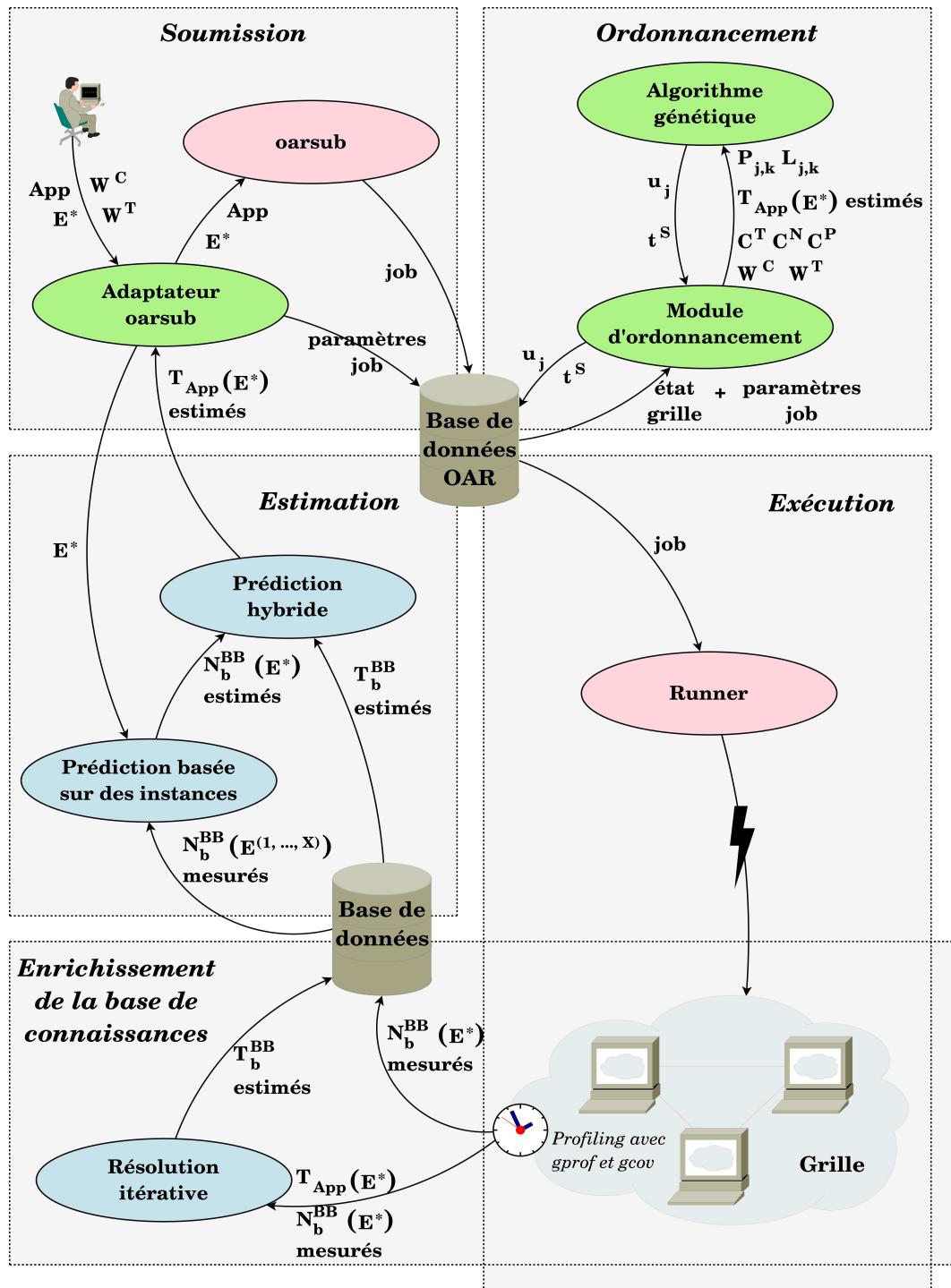


FIG. 7.1 – Vue d'ensemble des réalisations

7.4 Perspectives

Le modèle économique proposé dans le cadre de cette thèse pourrait être amélioré de plusieurs manières :

- **Extension du type de ressources prises en compte** : Le modèle actuel considère des ressources de type processeur (temps consommé, puissance, nombre utilisé). Il est envisageable de prendre en compte d'autres types de ressources, telles que la mémoire, le stockage, la bande-passante consommée, les bibliothèques utilisées, etc.
- **Notion de qualité de service** : Il est possible d'ajouter la notion de qualité de service aux ordonnancements proposés. Ainsi, il serait possible de garantir à l'utilisateur une *deadline* pour son application. Une telle garantie pourrait bien entendu lui être facturée.
- **Réservation manuelle** : Actuellement, l'ordonnanceur calcule la date de début de l'application de manière à optimiser un compromis entre le temps d'exécution de l'application et son coût. Cependant, il peut être intéressant de laisser à l'utilisateur la possibilité de spécifier lui-même la date de début de l'application (s'il désire exécuter l'application dans le cadre d'une démonstration par exemple). Dans ce cas, l'ordonnancement ne consisterait qu'à choisir les processeurs à utiliser, ce qui, en outre, simplifierait le problème d'optimisation. Cette réservation manuelle pourrait également être vue comme un service payant.
- **Augmentation du coût en fonction de la charge des processeurs** : Le coût de l'application ne dépend pas de la charge des processeurs. Seul le temps d'exécution est influencé par ce paramètre. Cependant, la facturation de la charge peut être envisagée : il en résultera alors un équilibrage de charge naturel, puisqu'à caractéristiques égales, un processeur moins chargé sera moins cher. A titre d'exemple, ce type de modèle économique est appliqué par certaines compagnies aériennes : acheter un billet d'avion peut coûter plus cher au fur et à mesure que l'avion se remplit.
- **Etendre le modèle économique à plus de types d'applications** : La version proposée du modèle ne prend en compte que les applications parallèles à gros grain. Une meilleure prise en compte des communications entre tâches permettrait de réduire le grain des applications considérées. Pour cela, il serait nécessaire de modéliser les communications, de manière à connaître le nombre de messages échangés, la quantité de données transmises pour chaque message, etc.

Concernant l'approche hybride de prédition de temps d'exécution, les points suivants pourraient permettre une évolution favorable de la méthode :

- **Instrumentation du code objet** : Une évolution pourrait consister à mettre en œuvre des techniques d'instrumentation du code objet des programmes. Ceci permettrait de ne plus estimer par un calcul le temps d'exécution des blocs de base, mais au contraire de les mesurer directement. La méthode pourrait alors y gagner en précision.
- **Annotations automatiques** : Une analyse du flot de données du programme pourrait permettre de déduire certaines annotations automatiquement. Ainsi, même si toutes les dépendances des blocs de base vis-à-vis des entrées ne pourraient pas

être déduites automatiquement, cela pourrait tout-de-même simplifier la tâche du programmeur.

- **Modèles analytiques** : Il peut enfin être envisagé de créer des modèles analytiques d'applications parallèles pour améliorer la prédiction. Ces modèles décriraient la structure des applications (un modèle correspondant à une famille d'applications, telles que les applications *maître / esclave*), permettant ainsi de considérer des caractéristiques communes dont on connaît les effets sur le temps d'exécution.

Annexes

Annexe A

Utilisation de l'adaptateur du client de soumission d'applications d'OAR

Cette annexe présente l'utilisation de l'adaptateur du client `oarsub`, avec les options qu'il met à disposition des utilisateurs.

```
[bmiegemo@frontale] $ cost_oarsub --help
Oarsub wrapper usage:
cost_oarsub [options] -- Wrapper for oarsub using cost scheduling policy

Help options:
-?, --help                                     Show help options

Application options:
-a, --application=./myapp                      The application to launch
-l, --lower-proc=1                               Lower bound for the number of computation nodes to use
-u, --upper-proc=20                             Upper bound for the number of computation nodes to use
-n, --nb-tasks=20                                Number of tasks of the application
-p, --power-ref=100                            Power of reference for the calculation of the application execution time
-d, --duration-ref=72000                         Application execution time on the reference processor
-m, --message=30                                 Communication time
-t, --time-coef=500                            Time coefficient in the target function
-c, --cost-coef=1                               Cost coefficient in the target function

Others options:
-o, --oarsub-options="OAR_OPTIONS"            Oarsub parameters.
                                                Example : " -p 'mem>512 AND deploy=NO' "
[bmiegemo@frontale] $
```

L'ensemble des données nécessaires au modèle économique est ainsi fourni en paramètre à notre client :

- Nombre de tâches de l'application (A) : option `-n`.
- Durée d'exécution de l'application sur un processeur de référence (T^R) : option `-d`.
- Puissance de ce processeur de référence (P^R) : option `-p`.
- Temps moyen de communication des tâches (T^C) : option `-m`.
- Bornes sur le nombre de processeurs à utiliser (N_{min}^P et N_{max}^P) : options `-l` et `-u`.
- Compromis entre le coût de l'application et sa date de fin (W^C et W^T) : options `-c` et `-t`.

De plus, il est possible de spécifier des options propres à l'outil **oarsub** en utilisant l'option **-o**. Ceci peut être le cas pour les variables d'environnement à initialiser, les clés *ssh* à utiliser, les clés de licence de logiciels, ou bien encore la configuration des signaux de vérification de l'application. Ainsi, l'option **-o** redirige ces paramètres, sans les modifier, vers la commande **oarsub**.

Certaines options d'**oarsub** incompatibles avec le modèle économique sont toutefois filtrées.

Ainsi, le choix de la file d'attente dans laquelle insérer le job n'est pas possible car il est imposé par l'utilisation de notre client : la file *Cost* est obligatoirement utilisée pour les applications soumises via l'ordonnanceur proposé. Un premier filtrage est donc effectué si l'option **-q** (option d'OAR correspondant au choix de la file d'attente) est détectée.

Ensuite une demande de réservation interactive n'est pas possible car ce type d'application n'est pas pris en compte par notre modèle. Un deuxième filtrage a donc lieu si l'option **-I** d'OAR est activée.

Enfin, l'option **-r** d'**oarsub**, permettant une réservation à une date de début fixe, est également filtrée, puisque l'algorithme génétique calcule cette date.

Annexe B

Stockage des paramètres nécessaires au modèle économique sous OAR

L'exécution de l'algorithme génétique requiert trois types de paramètres :

- ceux fournis par l'utilisateur, correspondant aux caractéristiques de l'application soumise,
- ceux qui sont fixés par l'administrateur de la grille ou par les propriétaires des ressources, incluant notamment les paramètres de coût,
- ceux dépendant de l'environnement, tels que le nombre de processeurs disponibles, leur puissance et leur charge.

Ces paramètres seront passés à l'algorithme génétique soit par la base de données d'OAR, soit par des fichiers de configuration.

B.1 Paramètres stockés dans la base de données d'OAR

OAR étant un logiciel en perpétuelle évolution, la modification des tables qui lui sont propres n'est pas envisageable si l'on désire découpler OAR de l'implémentation de notre module d'ordonnancement. Ainsi, pour passer les paramètres à l'algorithme génétique, nous créons nos propres tables.

Cost_scheduling	Cpu_load	Cpu_cost
Job_id Nb_proc_min Nb_proc_max Nb_tasks Puissance_ref Temps_ref Temps_com Coef_time Coef_cost Etat PID_cost Start_date	Cpu Date_start Date_stop Value	Cpu Date_start Date_stop Value

La table *Cost_scheduling* contient tous les paramètres fournis par l'utilisateur concernant l'application qu'il soumet :

- **Job_id** : identifiant unique de l'application de l'utilisateur,
- **Nb_proc_min** et **Nb_proc_max** : bornes minimales et maximales du nombre de processeurs à utiliser (N_{min}^P et N_{max}^P),
- **Nb_tasks** : nombre de tâches de l'application (A),
- **Puissance_ref** : puissance de référence (P^R),
- **Temps_ref** : temps de référence de l'application (T^R),
- **Temps_com** : temps de communication des tâches de l'application (T^C),
- **Coef_time** : coefficient de pondération du temps pour le calcul du placement de l'application (W^T),
- **Coef_cost** : coefficient de pondération du coût pour le calcul du placement de l'application (W^C),
- **Estat** : “*toSchedule*” ou “*Scheduling*” dans le cas où le job est à ordonner ou est en train de l'être, “*Scheduled*” quand l'ordonnancement est terminé, et “*Error*” en cas d'erreur,
- **PID_cost** : le PID du processus réalisant l'ordonnancement,
- **Start_date** : la date où le calcul du placement a débuté.

Les tables *Cpu_load* et *Cpu_cost* stockent respectivement la charge et les coefficients locaux de coût des processeurs, ces deux données dépendant du temps :

- **Cpu** : identifiant global unique du processeur,
- **Date_start** et **Date_stop** : les dates de début et de fin de l'intervalle de temps concerné, au format SQL,
- **Value** : la valeur de la charge du processeur (noté dans le modèle $L_{j,k}$) pour la table *Cpu_load*, et de son coefficient de coût (désigné par la variable $C_{j,k}^P$ du modèle) pour la table *Cpu_cost*.

Notons que le nombre de processeurs disponibles (N_a^P) est calculable à partir de la table des ressources d'OAR. De même, la puissance des processeurs apparaîtra dans la table contenant les propriétés des ressources fournie par OAR.

B.2 Paramètres stockés dans le fichier de configuration du module d'ordonnancement

Un fichier de configuration est mis en place permettant à l'administrateur de renseigner les coefficients de coût des différentes ressources à appliquer (C^T , C^N et C^P). Ce fichier contient également les paramètres nécessaires au fonctionnement de l'algorithme génétique (nombre d'individus constituant les populations, taux de croisement, taux de mutation, etc.).

De plus, le module d'ordonnancement utilisant le modèle économique proposé dans le cadre de cette thèse a été développé de telle sorte qu'il puisse être exécuté en deux modes (propriété configurable par la clé `simulation=true/false` dans le fichier de configuration) :

- Le mode production est le mode de fonctionnement nominal. Il utilise la base de données afin que les placements choisis soient pris en compte par OAR, et que les applications soient réellement exécutées sur la grille.
- Le mode simulation s'appuie sur la définition d'une grille virtuelle. Ce mode permet de tester le module en passant tous les paramètres nécessaires au modèle économique (données de l'application, charge des processeurs, coefficients de coût, etc.) par le fichier de configuration. Dans ce cas, OAR génère toutes les données nécessaires au placement des applications soumises, mais ces dernières ne sont pas exécutées.

Voici un exemple de fichier de configuration :

```
#####
# CONFIG SCHEDULER #
#####

[general]
# Nombre de cores sur la station d'exécution
# (pour exécuter l'algorithme génétique en multi-thread)
nb_threads=1

# L'algorithme doit-il tourner en mode simulation (true) ou production (false) sur OAR ?
simulation=true

# Aspects temporels
nb_time=50
delta_t=300

[resolution]
# Paramètres nécessaires à l'exécution de l'algorithme génétique
nb_individus=300
lambda=40
mu=25
nb_iterations=10000
coef_mutation=0.3
coef_crossover=0.6
coef_nouveaux=0.05
coef_conservation=0.05

[cost_coeff]
# Coefficients de tarification donnés par l'administrateur
coef_power=50
coef_nb_proc=1
coef_cpu_time=1

[grid_state]
# Activation des prises en charge des variations de l'état de la grille au cours du temps
is_load_enabled=true
is_cost_enabled=true

#####
# SIMULATION #
#####

[simulation_inputs_user]
# Entrées simulées de l'utilisateur correspondant à l'application soumise
nb_proc_min=1
nb_proc_max=5
nb_tasks=5
puissance_ref=100
temps_ref=72000
temps_com=30
coef_time=500
coef_cost=1

[simulation_inputs_administrator]
# Entrées simulées de l'administrateur
nb_proc=10
```


Annexe C

Processus de soumission d'une application dans OAR et synchronisation du module d'ordonnancement

Cette annexe décrit le processus de soumission d'une application dans OAR, en détaillant les états successifs dans lesquels se trouve l'application. Nous montrons ensuite quels sont les problèmes de synchronisation portant sur l'état de l'application qui peuvent survenir entre les tables gérées par le module d'ordonnancement et celles d'OAR, en décrivant pour chaque cas les actions à entreprendre pour résoudre les conflits.

C.1 Processus de soumission

Le cheminement de la soumission d'une application peut s'apparenter à une machine à états telle que représentée par la figure C.1.

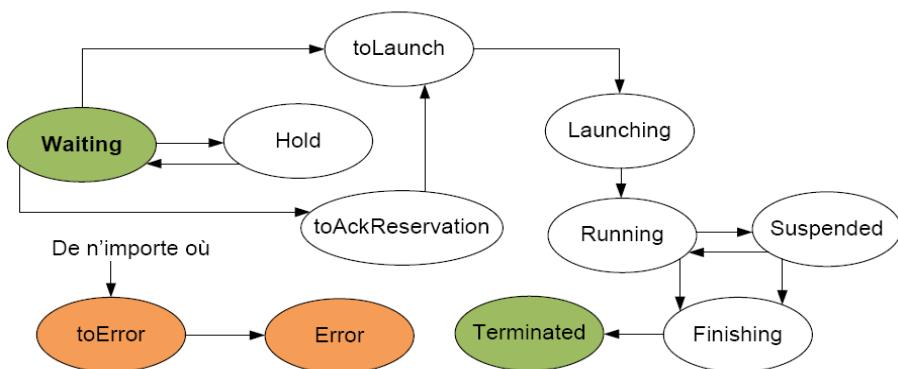


FIG. C.1 – Machine à états du processus de soumission d'une application

Un job nouvellement soumis entre dans l'état initial *Waiting*. Il a donc déjà subi des contrôles passifs, effectués par le client de soumission, portant sur la possibilité de lancement du job (nombre suffisant de ressources, etc.).

Il existe deux sortes de soumissions :

- La soumission avec **réservation explicite** : l'utilisateur fournit la date de début de l'application, et l'ordonnanceur d'OAR tente de trouver les ressources nécessaires à cette date. Dans ce cas, le job passera à l'état *toAckReservation*, dans lequel la vérification de la disponibilité des ressources demandées à la date demandée est forcée. En cas de non disponibilité, le job entre dans l'état *toError* et l'utilisateur est informé du prochain créneau horaire disponible. Si les ressources sont disponibles, le job passe à l'état *toLaunch*. Notons que la réservation explicite n'est pas compatible avec notre modèle puisque c'est lui qui calcule la date de début.
- La soumission avec **réservation implicite** : l'utilisateur soumet l'application sans préciser de date de début. L'ordonnanceur d'OAR cherche ainsi la première date à laquelle les ressources demandées sont disponibles. Le job passe alors dans l'état *toLaunch*. L'utilisation du client de soumission basé sur le modèle économique fonctionnera de cette manière, la date de début étant calculée de manière à minimiser le compromis spécifié par l'utilisateur entre le coût de l'application et sa date de terminaison.

L'état *Hold* est atteint dans le cas où l'administrateur ou l'utilisateur suspend le job à l'aide de la commande `oarhold` au cours de la phase d'attente préliminaire.

Une fois la réservation des ressources effectuée à l'aide de la politique d'ordonnancement choisie, le job entre dans l'état *toLaunch*, c'est-à-dire en attente de lancement. Il reste dans cet état jusqu'à la date à laquelle il doit être lancé. Ensuite, lorsque cette date arrive, le job passe à l'état *Launching*, dans lequel le serveur OAR se connecte sur le premier nœud de calcul en *ssh*, puis propage l'application sur tous les nœuds.

Le job entre alors dans l'état *Running*, c'est-à-dire en cours d'exécution, et l'application peut alors effectuer les calculs sur tous les nœuds qui lui sont attribués. L'utilisateur ou l'administrateur peuvent encore faire appel à la commande `oarhold`, permettant de mettre le job dans l'état *Suspended*. Dans cet état, le calcul est mis en pause, ce qui implique que la durée de réservation initiale basée sur la quantité de calculs à effectuer est faussée. Le calcul risque alors de ne pas se terminer correctement dans le temps imparti. Il est toutefois possible d'imaginer de recalculer l'ordonnancement lorsque le job est remis en marche comme cela est abordé dans la section suivante.

Enfin, dans l'état *Finishing*, les nœuds sont nettoyés afin que le job finisse dans l'état *Terminated*. Notons que si l'application ne se termine pas dans le temps qui lui est imparti, ses tâches sont alors tuées pour permettre la terminaison du job.

C.2 Synchronisation entre le module d'ordonnancement créé et le serveur OAR

Le tableau suivant met en relation la table *Jobs* gérée par OAR avec la table *Cost_scheduling* maintenue par le module d'ordonnancement basé sur le modèle économique, en se focalisant sur les champs de ces tables susceptibles de montrer des problèmes de synchronisation. Nous étudierons les actions à entreprendre dans le module d'ordonnancement pour certaines valeurs contenues dans ces tables.

Table “Jobs”		Table “Cost_scheduling”	Actions du Cost_scheduler	
State	Reservation	Etat		
Waiting	None	toSchedule	1	
Waiting	None	Scheduling	2	
Waiting	None	X	3	
X	X	Scheduling	4	
X	X	toSchedule	5	
Waiting	None	Scheduled	6	
Waiting	None	Error	7	
Suspended	X	Scheduled	8	
Suspended	X	Error	8	

TAB. C.1 – Actions à réaliser en fonction des valeurs contenues dans les tables *Jobs* et *Cost_scheduling*

~~ Action 1

Dans cet état, un job est dans la file d’attente *Cost* et l’identifiant a bien été trouvé dans la table *Cost_scheduling*, ce qui veut dire que les paramètres d’entrée devraient être disponibles. La première action est de vérifier la cohérence des paramètres d’entrée (vérification des valeurs pour chaque paramètre).

Si une erreur est détectée dans les paramètres (valeur négative, en dehors des bornes, caractères inconnus, etc.), l’ordonnanceur va alors ignorer ce job et demander à OAR de nettoyer sa base de données. Il suffit pour cela de changer l’état dans la table *Jobs* de “Waiting” à “toError”. Ainsi, lors du prochain rafraîchissement du serveur OAR, le job sera nettoyé. Il faut ensuite changer l’état du job correspondant en lui donnant la valeur “Error” dans notre table.

Dans le cas où les paramètres d’entrée sont corrects, il faudra initialiser les variables d’entrée de l’algorithme génétique de placement et le lancer. L’état du job dans notre table est modifié pour passer de “toSchedule” à “Scheduling”. Une fois l’ordonnancement terminé, il passera à l’état “Scheduled”.

~~ Action 2

Dans cet état, l’action 1 a déjà été effectuée puisque le job est dans l’état “Scheduling”. OAR n’a pas encore reçu de réponse des solutions de placement et a relancé après un *timeout* la politique d’ordonnancement. Normalement, un ordonnancement est donc déjà en cours et nous devons faire une vérification de son bon déroulement.

Pour cela nous allons utiliser dans la table *Cost_scheduling* les champs *Date_start* correspondant à la date de début de l’ordonnancement, et *PID_cost* contenant l’identifiant du processus du module d’ordonnancement lancé. Ainsi, on peut vérifier si le processus existe toujours avec la commande “`ps -p PID_cost`”. On peut aussi décider d’arrêter l’ordonnancement au bout d’une certaine date limite, par exemple 3 minutes, en tuant le processus avec la commande “`kill PID_cost`”.

De la même manière que précédemment, le job devra être nettoyé en changeant le champ **State** de la table *Jobs* en “toError”, ainsi que le champ **Etat** de la table *Cost_scheduling* en “Error”. Le passage de l'état “toError” à l'état “Error” dans la table *Jobs* sera automatiquement réalisé par OAR.

~~ Action 3

Le serveur OAR a lancé notre politique d'ordonnancement car un job est en attente dans la file *Cost*, mais les paramètres d'entrée sont introuvables dans la table *Cost_scheduling*. Cette configuration est possible car le job est chargé dans les tables d'OAR avant que les paramètres ne soient écrits dans la table *Cost_scheduling*. Il est également possible qu'il y ait des ralentissements réseaux ou du serveur de base de données qui amènent à cette même configuration.

Dans tous les cas, le job est en attente mais le client n'a toujours pas envoyé ses paramètres de coût. La solution est d'attendre un peu que le client écrive dans la table *Cost_scheduling*. Nous ferons trois vérifications : la première à 1 seconde, la deuxième après 5 secondes et la troisième après 15 secondes. Au-delà, nous considérons qu'il y a une erreur au niveau du client qui insère les données. Nous chargeons donc OAR de nettoyer le job (en plaçant l'état “toError” dans le champ **State** de la table *Jobs*). Notons que si les paramètres sont écrits après ce délai de 15 secondes, la situation obtenue sera prise en charge par l'action 5.

~~ Action 4

Nous trouvons des paramètres de coût orphelins dans la table *Cost_scheduling* et qui sont en attente d'ordonnancement : ils n'ont pas de correspondance dans les tables d'OAR.

Il faut dans un premier temps vérifier s'il est possible de trouver le job dans les tables d'OAR dans un autre état que “Waiting”. Si on trouve le job dans l'état “Hold”, il faut alors attendre qu'il repasse en “Waiting” sur décision d'un utilisateur ou d'un administrateur, et ne rien modifier dans la base de données. S'il est dans l'état “toAckReservation”, alors il faut le forcer en “toError” car il y a une incohérence, puisque dans cet état, la date de début a été forcée par l'utilisateur, alors que notre ordonnanceur est sensé la déterminer. Dans ce cas, et s'il est dans l'état “toError” ou “Error”, il faut changer le champ **Etat** de la table *Cost_scheduling* en “Error”. Les autres états du job (“toLaunch”, “Launching”, “Running”, “Suspended”, “Finishing” et “Terminated”) ne devraient pas être possibles et donc il faut changer le champ **Etat** de la table *Cost_scheduling* en “Error”, et laisser OAR réparer son erreur.

Dans un second temps, il faut vérifier si le job n'a pas un champ **Reservation** différent de “None”. Si c'est le cas, cela signifie que l'utilisateur a donné une date de début, ce qui est incompatible avec notre ordonnanceur. Il faut donc changer le champ **State** de la table *Jobs* en “toError” et le champ **Etat** de la table *Cost_scheduling* en “Error”.

~~ Action 5

Nous avons ici un job en cours d'ordonnancement, toujours orphelin. Il faut traiter les erreurs de la même manière que pour l'action 4, mais également tuer le processus qui produit l'ordonnancement (de la même manière que pour l'action 2).

~~ Action 6

Notre ordonnancement est fini car le job se trouve dans l'état "Scheduled", mais OAR a toujours le job dans sa file d'attente *Cost*. Il peut se passer un certain temps avant qu'OAR ne rafraîchisse l'état d'un job après avoir inséré les paramètres de placement dans sa base. En effet, les modules d'OAR ne viennent vérifier les réponses des politiques d'ordonnancement qu'au moment du rafraîchissement périodique (15 secondes en moyenne). Nous attendons donc avec trois *timeouts* : le premier à 3 secondes, le deuxième après 10 secondes et le troisième après 20 secondes. Au-delà, nous considérons qu'il y a une erreur. De même que si le job est passé à l'état "toError" ou "Error", nous changeons le champ **Etat** de la table *Cost_scheduling* en "Error".

~~ Action 7

Un job est en attente dans la file d'OAR et en erreur dans la nôtre, il suffit de changer le champ **State** de la table *Jobs* en "toError" afin qu'OAR se charge du nettoyage du job.

~~ Action 8

Un job a été suspendu par l'administrateur ou l'utilisateur (état "Suspended") après la détermination d'un ordonnancement (donc depuis l'état "Scheduled"). Ce cas peut être critique pour le bon déroulement du calcul, car une fois les ressources attribuées, OAR ne peut plus modifier cette attribution à moins d'annuler le calcul courant et de soumettre à nouveau un job.

La meilleure solution consisterait à recalculer automatiquement l'ordonnancement lorsque le job est remis en marche. L'inconvénient est que le coût ou la durée peut ne plus convenir à l'utilisateur. Nous décidons donc d'arrêter le job en le forçant à l'état d'erreur (changement du champ **State** de la table *Jobs* en "toError" et du champ **Etat** de la table *Cost_scheduling* en "Error").

Une amélioration possible serait de créer des bornes de sécurité rajoutées en paramètres d'utilisateur. La durée spécifiée donnerait la possibilité de suspendre le job au maximum pendant cette durée, mais la réservation totale coûterait plus cher. Au-delà de cette borne, le job passerait à l'état d'erreur.

Annexe D

Principe de fonctionnement de l'outil de profiling gprof

L'objectif de *gprof* est de fournir un profil de l'exécution d'une application. Il permet, en outre, de connaître le temps d'exécution de chacune des fonctions d'un programme, ainsi que le nombre de fois où elle a été appelée.

Gprof combine deux techniques de profiling :

- l'instrumentation,
- l'échantillonnage (*sampling*).

D.1 L'instrumentation

L'instrumentation du code est certes précise, mais elle entraîne un ralentissement considérable de l'application. Ainsi, *gprof* limite cette technique à la seule comptabilisation des appels de fonctions. Le *call-monitoring* donne des résultats précis et fiables à 100 %, et permet ainsi de connaître le nombre d'exécutions de chacune des fonctions constituant le programme profilé.

Ainsi, le nombre d'exécutions de chaque fonction est déterminé de manière déterministe, fiable et précise.

D.2 L'échantillonnage

La technique du *sampling* est utilisée pour déterminer le temps d'exécution de chacune des fonctions du programme. L'échantillonnage repose sur des interruptions du système d'exploitation à intervalles de temps réguliers, et n'entraîne qu'un faible ralentissement de l'application. En contre-partie, cette technique donne des résultats moins précis que l'instrumentation.

Principe de l'échantillonnage

Le *sampling* repose ainsi sur des interruptions du système d'exploitation, afin d'analyser périodiquement certaines données du système (pointeur d'instruction, pile, compteurs, état du processeur, état du système d'exploitation, etc.). L'échantillonnage est réalisé à intervalles fixes (typiquement 0.01s) de *runtime*. Ceci implique que *gprof* ne comptabilise

que le temps processeur, et ignore les temps passés par l'application en accès disque, swap, etc.

Erreur induite

Le procédé du *sampling* est imprécis, car il repose sur des approximations statistiques. Ainsi, si le temps d'exécution d'une fonction est très faible, il peut, soit être ignoré (car l'appel et la sortie de la fonction se font entre deux échantillons successifs), ou bien peut compter comme toute une période d'échantillonnage (si l'appel et la sortie encadrent l'échantillon).

Il est possible d'évaluer l'erreur. Si N échantillons sont pris durant l'exécution d'une application, alors l'erreur "attendue" est \sqrt{N} . Ainsi, soit D la durée d'une application, T la période d'échantillonnage, et E l'erreur sur le temps d'exécution, on a :

$$N = \frac{D}{T}$$

$$E = T \cdot \sqrt{N} = T \cdot \sqrt{\frac{D}{T}} = \frac{T}{\sqrt{T}} \cdot \sqrt{D}$$

L'erreur $E\%$ relative à l'ensemble de durée d'exécution est ainsi :

$$E\% = \frac{E}{D} = \frac{T}{\sqrt{T}} \cdot \frac{\sqrt{D}}{D}$$

Il apparaît ainsi que le temps d'exécution des fonctions doit être grand en comparaison de la période d'échantillonnage. Si ce n'est pas le cas, il faut alors compter sur un nombre d'appels important des fonctions pour parvenir à obtenir des statistiques relativement fiables.

Annexe E

Exemple de l'élévation d'une matrice carrée à la puissance désirée

Cette annexe se propose de regrouper l'ensemble des fichiers relatifs à l'exemple utilisé dans ce document. Cet exemple traite l'élévation d'une matrice carrée de dimension donnée à la puissance désirée. L'exposant ainsi que la dimension de la matrice sont les valeurs d'entrées du programme. La matrice de départ est ensuite automatiquement générée par le programme.

Toute entrée du programme s'exprime ainsi :

$$E = \begin{bmatrix} E_1 \\ E_2 \end{bmatrix}$$

où :

- E_1 est la dimension de la matrice,
- E_2 est la puissance à laquelle la matrice doit être élevée.

E.1 Code source original

Cette section présente le code source du programme utilisé. Il est à noter que ce programme possède deux valeurs d'entrées, susceptibles de faire varier son temps d'exécution. Il s'agit de la dimension de la matrice, ainsi que de l'exposant utilisé.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 // Displays a square matrix
6 void displayMatrix(int dimensions, unsigned long matrix[dimensions][dimensions]) {
7     for (int i = 0 ; i < dimensions ; i++) {
8         printf("|");
9         for (int j = 0 ; j < dimensions ; j++) {
10            printf("%10d", matrix[i][j]);
11        }
12        printf("|\n");
13    }
14 }
15
16
17 // Generates a square matrix
18 void generateMatrix(int dimensions, unsigned long matrix[dimensions][dimensions]) {
19     for (int i = 0 ; i < dimensions ; i++) {
20         for (int j = 0 ; j < dimensions ; j++) {
21             matrix[i][j] = i + j;
22         }
23     }
24 }
```

```

23     }
24     for ( int i = 0 ; i < dimensions ; i++ ) {
25         for ( int j = 0 ; j < dimensions ; j++ ) {
26             if ( matrix[i][j] % 2 == 0 )
27                 matrix[i][j] /=2 ;
28             else
29                 matrix[i][j] *=2 ;
30         }
31     }
32 }
33
34
35 // Copies the content of a matrix into another
36 void copyMatrix(int dimensions, unsigned long matrixSource[dimensions][dimensions],
37                 unsigned long matrixTarget[dimensions][dimensions]) {
38     for ( int i = 0 ; i < dimensions ; i++ ) {
39         for ( int j = 0 ; j < dimensions ; j++ ) {
40             matrixTarget[i][j] = matrixSource[i][j];
41         }
42     }
43 }
44
45
46 // Fills a matrix with ones
47 void onesMatrix(int dimensions, unsigned long matrix[dimensions][dimensions]) {
48     for ( int i = 0 ; i < dimensions ; i++ ) {
49         for ( int j = 0 ; j < dimensions ; j++ ) {
50             matrix[i][j] = 1;
51         }
52     }
53 }
54
55
56 // Multiplicates two matrixes
57 void matrixMultiplication(int dimensions, unsigned long leftMatrix[dimensions][dimensions],
58                           unsigned long rightMatrix[dimensions][dimensions],
59                           unsigned long resultMatrix[dimensions][dimensions]) {
60     for ( int i = 0 ; i < dimensions ; i++ ) {
61         for ( int j = 0 ; j < dimensions ; j++ ) {
62             resultMatrix[i][j] = 0;
63             for ( int k = 0 ; k < dimensions ; k++ ) {
64                 resultMatrix[i][j] += leftMatrix[i][k] * rightMatrix[k][j];
65             }
66         }
67     }
68 }
69
70
71 // Computes the matrix power
72 void matrixPower(int dimensions, int power, unsigned long matrix[dimensions][dimensions],
73                  unsigned long powerMatrix[dimensions][dimensions]) {
74     if ( power == 0 ) {
75         onesMatrix(dimensions, powerMatrix);
76     } else {
77         unsigned long intermediateMatrix[dimensions][dimensions];
78         copyMatrix(dimensions, matrix, powerMatrix);
79         for ( int n = 1 ; n < power ; n++ ) {
80             copyMatrix(dimensions, powerMatrix, intermediateMatrix);
81             matrixMultiplication(dimensions, intermediateMatrix, matrix, powerMatrix);
82         }
83     }
84 }
85
86
87 // The main function
88 int main(int argc, char ** argv) {
89
90     // Program usage
91     if ( argc != 3 ) {
92         printf("usage: matrixPower dimensions power\n");
93         printf("dimensions: integer\npower: integer\n");
94         return -1;
95     }
96
97     // We get the matrix dimensions
98     int dimensions = atoi(argv[1]);
99     if ( dimensions <= 0 ) {
100        printf("dimensions must be a positive number");
101        return -2;
102    }
103
104    // We build the matrix
105    unsigned long matrix[dimensions][dimensions];
106    generateMatrix(dimensions, matrix);
107
108    // We get the power
109    int power = atoi(argv[2]);
110    if ( power < 0 ) {
111        printf("power must be a positive number");
112        return -2;
113    }
114
115    // We compute the power matrix.
116    unsigned long powerMatrix[dimensions][dimensions];
117    matrixPower(dimensions, power, matrix, powerMatrix);

```

```

118     // We display the matrix
119     //printf("Original matrix :\n");
120     //displayMatrix(dimensions, matrix);
121     //printf("Power matrix :\n");
122     //displayMatrix(dimensions, powerMatrix);
123
124 }
125 }
```

E.2 Fichiers obtenus avec gprof et gcov

Les outils *gprof* et *gcov* permettent, après avoir exécuté le programme pour une entrée donnée, de connaître le temps d'exécution de chacune des fonctions le constituant, ainsi que le nombre d'exécutions de chacune des lignes de ce programme.

Les deux sous-sections suivantes détaillent les résultats produits par ces deux outils. Pour cela, des valeurs d'entrées doivent être choisies. Nous choisissons, par exemple, d'élever une matrice carrée de dimension 43 à la puissance 74, c'est-à-dire :

$$E = \begin{bmatrix} 43 \\ 74 \end{bmatrix}$$

E.2.1 Résultat produit par gprof

Le fichier suivant est produit par *gprof*. Il permet notamment de connaître le temps d'exécution global de chaque fonction du programme (temps cumulé passé à l'intérieur de cette fonction au fil des différents appels).

```

1  Flat profile:
2
3  Each sample counts as 0.01 seconds.
4  % cumulative self          self      total
5  time    seconds   seconds  calls  s/call  s/call  name
6  97.40     1.10     1.10     73    0.02    0.02  matrixMultiplication
7  2.66      1.13     0.03     74    0.00    0.00  copyMatrix
8  0.00      1.13     0.00      1    0.00    0.00  generateMatrix
9  0.00      1.13     0.00      1    0.00    1.13  matrixPower
10
11
12     Call graph
13
14
15 granularity: each sample hit covers 2 byte(s) for 0.88% of 1.13 seconds
16
17 index % time    self    children    called      name
18           0.00     1.13      1/1        main [2]
19 [1]    100.0    0.00     1.13      1        matrixPower [1]
20           1.10     0.00      73/73    matrixMultiplication [3]
21           0.03     0.00      74/74    copyMatrix [4]
22 -----
23
24 [2]    100.0    0.00     1.13      <spontaneous>
25           0.00     1.13      1/1        main [2]
26           0.00     0.00      1/1        matrixPower [1]
27           0.00     0.00      1/1        generateMatrix [5]
28
29 [3]    97.3     1.10     0.00      73/73    matrixPower [1]
30           1.10     0.00      73        matrixMultiplication [3]
31           0.03     0.00      74/74    matrixPower [1]
32 [4]     2.7      0.03     0.00      74        copyMatrix [4]
33
34           0.00     0.00      0.00      1/1        main [2]
35 [5]     0.0      0.00     0.00      1        generateMatrix [5]
36
37
38 Index by function name
39
40 [4] copyMatrix           [3] matrixMultiplication
41 [5] generateMatrix       [1] matrixPower
```

Sur cet exemple d'exécution, il est possible de constater que le programme a passé 1,10 secondes à exécuter la fonction **matrixMultiplication**.

E.2.2 Résultat produit par gcov

Gcov permet de connaître le nombre d'exécutions de chacune des lignes du code source grâce à un fichier contenant le code source du programme, dont chaque ligne est préfixée par le nombre de fois où elle a été exécutée.

```

1      -: 0:Source:matrixPower.c
2      -: 0:Graph:matrixPower.gcov
3      -: 0:Data:matrixPower.gcda
4      -: 0:Runs:1
5      -: 0:Programs:1
6      -: 1:#include <stdio.h>
7      -: 2:#include <stdlib.h>
8      -: 3:
9      -: 4:
10     -: 5:// Displays a square matrix
11    #####: 6:void displayMatrix(int dimensions,
12      unsigned long matrix[dimensions][dimensions]) {
13      #####: 7: for ( int i = 0 ; i < dimensions ; i++ ) {
14      #####: 8:   printf(" |");
15      #####: 9:   for ( int j = 0 ; j < dimensions ; j++ ) {
16      #####: 10:     printf("%10d", matrix[i][j]);
17      #####: 11:   }
18      #####: 12:   printf(" |\n");
19      -: 13: }
20      -: 14: }
21      -: 15:
22      -: 16:
23      -: 17:// Generates a square matrix
24      1: 18:void generateMatrix(int dimensions,
25        unsigned long matrix[dimensions][dimensions]) {
26      44: 19: for ( int i = 0 ; i < dimensions ; i++ ) {
27      1892: 20:   for ( int j = 0 ; j < dimensions ; j++ ) {
28      1849: 21:     matrix[i][j] = i + j;
29      -: 22:   }
30      -: 23: }
31      44: 24:   for ( int i = 0 ; i < dimensions ; i++ ) {
32      1892: 25:     for ( int j = 0 ; j < dimensions ; j++ ) {
33      1849: 26:       if ( matrix[i][j] % 2 == 0 )
34      925: 27:         matrix[i][j] /= 2 ;
35      -: 28:       else
36      924: 29:         matrix[i][j] *= 2 ;
37      -: 30:     }
38      -: 31:   }
39      -: 32: }
40      -: 33:
41      -: 34:
42      -: 35:// Copies the content of a matrix into another
43      74: 36:void copyMatrix(int dimensions,
44        unsigned long matrixSource[dimensions][dimensions],
45        unsigned long matrixTarget[dimensions][dimensions]) {
46      3256: 37: for ( int i = 0 ; i < dimensions ; i++ ) {
47      140008: 38:   for ( int j = 0 ; j < dimensions ; j++ ) {
48      136826: 39:     matrixTarget[i][j] = matrixSource[i][j];
49      -: 40:   }
50      -: 41: }
51      -: 42: }
52      -: 43:
53      -: 44:
54      -: 45:// Fills a matrix with ones
55      #####: 46:void onesMatrix(int dimensions,
56        unsigned long matrix[dimensions][dimensions]) {
57      47: for ( int i = 0 ; i < dimensions ; i++ ) {
58      48:   for ( int j = 0 ; j < dimensions ; j++ ) {
59      49:     matrix[i][j] = 1;
60      -: 50:   }
61      -: 51: }
62      -: 52: }
63      -: 53:
64      -: 54:
65      -: 55:// Multiplicates two matrixes
66      73: 56:void matrixMultiplication(int dimensions,
67        unsigned long leftMatrix[dimensions][dimensions],
68        unsigned long rightMatrix[dimensions][dimensions],
69        unsigned long resultMatrix[dimensions][dimensions]) {
70      3212: 57: for ( int i = 0 ; i < dimensions ; i++ ) {
71      138116: 58:   for ( int j = 0 ; j < dimensions ; j++ ) {
72      134977: 59:     resultMatrix[i][j] = 0;
73      5938988: 60:     for ( int k = 0 ; k < dimensions ; k++ ) {
74      5804011: 61:       resultMatrix[i][j] += leftMatrix[i][k] * rightMatrix[k][j];
75      -: 62:     }
76      -: 63:   }
77      -: 64: }
78      -: 65: }
79      -: 66:
80      -: 67:
81      -: 68:// Computes the matrix power
82      1: 69:void matrixPower(int dimensions,
83        int power,
84        unsigned long matrix[dimensions][dimensions],
85        unsigned long powerMatrix[dimensions][dimensions]) {
86      1: 70: if ( power == 0 ) {
87      #####: 71:   onesMatrix(dimensions, powerMatrix);

```

```

88      -:    72: } else {
89      1:    73:     unsigned long intermediateMatrix [dimensions][dimensions];
90      1:    74:     copyMatrix(dimensions, matrix, powerMatrix);
91      74:    75:     for ( int n = 1 ; n < power ; n++ ) {
92      73:    76:         copyMatrix(dimensions, powerMatrix, intermediateMatrix);
93      73:    77:         matrixMultiplication(dimensions, intermediateMatrix, matrix,
94      94:                                     powerMatrix);
95      -:    78:     }
96      -:    79:   }
97      -:    80: }
98      -:    81: 
99      -:    82: 
100     -:    83:// The main function
101    1:    84:int main(int argc, char ** argv) {
102    -:    85: 
103    -:    86: // Program usage
104    1:    87:     if ( argc != 3 ) {
105    ######: 88:         printf("usage: matrixPower dimensions power\n");
106    ######: 89:         return -1;
107    -:    90:     }
108    -:    91: 
109    -:    92: // We get the matrix dimensions
110   1:    93:     int dimensions = atoi(argv[1]);
111   1:    94:     if ( dimensions <= 0 ) {
112   ######: 95:         printf("dimensions must be a positive number");
113   ######: 96:         return -2;
114   -:    97:     }
115   -:    98: 
116   -:    99: // We build the matrix
117  1:   100:     unsigned long matrix [dimensions][dimensions];
118  1:   101:     generateMatrix(dimensions, matrix);
119  -:   102: 
120  -:   103: // We get the power
121  1:   104:     int power = atoi(argv[2]);
122  1:   105:     if ( power < 0 ) {
123  ######: 106:         printf("power must be a positive number");
124  ######: 107:         return -2;
125  -:   108:     }
126  -:   109: 
127  -:   110: // We compute the power matrix.
128  1:   111:     unsigned long powerMatrix [dimensions][dimensions];
129  1:   112:     matrixPower(dimensions, power, matrix, powerMatrix);
130  -:   113: 
131  -:   114: // We display the matrix
132  -:   115:     //printf("Original matrix :\n");
133  -:   116:     //displayMatrix(dimensions, matrix);
134  -:   117:     //printf("Power matrix :\n");
135  -:   118:     //displayMatrix(dimensions, powerMatrix);
136  -:   119: 
137  -:   120: }
138

```

E.3 Analyse des résultats

E.3.1 Découpage en blocs de base

Le fichier produit par *gcov* permet de déterminer les différents blocs de base étendus maximaux de chacune des fonctions du programme exécuté. Ainsi, en prenant l'exemple de la fonction `matrixMultiplication`, six blocs de base se dessinent, chacun ayant son propre nombre d'exécutions.

Même si chaque bloc de base possède son propre nombre d'exécutions, il est possible que ces nombres soient liés entre eux. La loi de dépendance entre ces nombres d'exécutions dépend de la structure du programme, plus précisément du flot de contrôle de ce dernier.

E.3.2 Expression du nombre d'exécutions des blocs de base en fonction des entrées

Considérons les nombres d'exécutions de chacun des blocs de base de la fonction `matrixMultiplication`:

$$N_{b_1}^{BB}(E) = 134977 \quad N_{b_2}^{BB}(E) = 5804011 \quad N_{b_3}^{BB}(E) = 5938988$$

$$N_{b_4}^{BB}(E) = 138116 \quad N_{b_5}^{BB}(E) = 3212 \quad N_{b_6}^{BB}(E) = 73$$

avec :

$$E = \begin{bmatrix} E_1 \\ E_2 \end{bmatrix} = \begin{bmatrix} 43 \\ 74 \end{bmatrix}$$

On peut exprimer le nombre d'exécutions de chacun des blocs de base de la fonction `matrixMultiplication` en fonction des valeurs d'entrée du programme :

$$N_{b_1}^{BB}(E) = [E_1]^2 \cdot (E_2 - 1) \quad N_{b_2}^{BB}(E) = [E_1]^3 \cdot (E_2 - 1)$$

$$N_{b_3}^{BB}(E) = [E_1]^2 \cdot (E_1 + 1) \cdot (E_2 - 1) \quad N_{b_4}^{BB}(E) = E_1 \cdot (E_1 + 1) \cdot (E_2 - 1)$$

$$N_{b_5}^{BB}(E) = (E_1 + 1) \cdot (E_2 - 1) \quad N_{b_6}^{BB}(E) = (E_2 - 1)$$

E.3.3 Dépendance entre les nombres d'exécutions des différents blocs de base

Les expressions obtenues dans la section précédente ainsi que la structure du programme, notamment les imbrications de boucles, permettent de déduire les relations suivantes :

$$N_{b_5}^{BB}(E) = N_{b_6}^{BB}(E) \cdot (E_1 + 1)$$

$$N_{b_4}^{BB}(E) = E_1 \cdot N_{b_6}^{BB}(E) \cdot (E_1 + 1) = [N_{b_5}^{BB}(E) - N_{b_6}^{BB}(E)] \cdot (E_1 + 1)$$

$$N_{b_1}^{BB}(E) = E_1 \cdot E_1 \cdot N_{b_6}^{BB}(E) = N_{b_4}^{BB}(E) - [N_{b_5}^{BB}(E) - N_{b_6}^{BB}(E)]$$

$$N_{b_3}^{BB}(E) = E_1 \cdot E_1 \cdot E_1 \cdot N_{b_6}^{BB}(E) \cdot (E_1 + 1) = N_{b_1}^{BB}(E) \cdot (E_1 + 1)$$

$$N_{b_2}^{BB}(E) = E_1 \cdot E_1 \cdot E_1 \cdot N_{b_6}^{BB}(E) = N_{b_3}^{BB}(E) - N_{b_1}^{BB}(E)$$

Annexe F

Notions mathématiques pour la résolution itérative d'un système d'équations linéaires mal conditionné

Dans cette annexe, nous définissons les notions mathématiques nécessaires à la résolution itérative d'un système d'équations linéaires mal conditionné.

F.1 Valeurs propres et valeurs singulières d'une matrice

F.1.1 Valeurs propres

Soit M une matrice carrée de dimension m . Cette matrice admet m valeurs propres $\lambda_{i,i \in [1;m]}$ telles que le vecteur λ constitué de ces m valeurs est racine de l'équation caractéristique :

$$\det(M - \lambda I) = 0$$

On note ainsi $\lambda(M)$ le vecteur contenant les valeurs propres de la matrice M , et on note $\lambda_i(M)$ la $i^{\text{ème}}$ composante de ce vecteur.

F.1.2 Valeurs singulières

Soit A une matrice de dimension $m \times n$. Les valeurs singulières de la matrice A sont les racines carrées des valeurs propres de la matrice $A^T \cdot A$, où A^T désigne la matrice transposée de A . La matrice $A^T \cdot A$ étant une matrice carrée de dimension n , la matrice A admet ainsi n valeurs singulières, notées $\sigma_{i,i \in [1;n]}$, telles que :

$$\sigma_i = \sqrt{\lambda_i(A^T \cdot A)}$$

On note ainsi $\sigma(A)$ le vecteur contenant les valeurs singulières de la matrice A , et on note $\sigma_i(A)$ la $i^{\text{ème}}$ composante de ce vecteur.

Par convention, les valeurs singulières sont triées par ordre décroissant :

$$\sigma_1(A) > \sigma_2(A) > \dots > \sigma_n(A)$$

On définit également $\sigma_{\min}(A)$ et $\sigma_{\max}(A)$ respectivement la plus petite et la plus grande valeur singulière de la matrice A :

$$\sigma_{\min}(A) = \sigma_n(A) \quad \sigma_{\max}(A) = \sigma_1(A)$$

F.2 Conditionnement d'une matrice

Le conditionnement d'un système d'équations linéaires $A \cdot x = b$ exprime l'imprécision de sa résolution sur la solution x , en traduisant l'effet d'une variation du second membre b sur l'inconnue x d'une part, et l'effet d'une variation de la matrice A sur l'inconnue x d'autre part.

En pratique, un système est qualifié de "bien conditionné" si son conditionnement est faible, c'est-à-dire si les incertitudes sur la sortie du système sont du même ordre de grandeur que celles sur ses entrées. Dans le cas contraire, le système est dit "mal conditionné".

Le conditionnement du système $A \cdot x = b$ ne dépend que de la matrice A . Ainsi, on définit le conditionnement de la matrice A , noté $\kappa(A)$ par la relation suivante :

$$\kappa(A) = \|A^{-1}\| \cdot \|A\|$$

où $\|A\|$ désigne n'importe quelle norme consistante de la matrice A . En particulier, dans le cas de la norme euclidienne, on a :

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

F.3 Décomposition en valeurs singulières

Soit A une matrice quelconque de dimension $m \times n$. Il existe alors une factorisation de la forme :

$$A = U \cdot \Sigma \cdot V^T$$

où :

- U est une matrice carrée de dimension m ,
- V est une matrice carrée de dimension n ,
- Σ est une matrice rectangulaire de dimension $m \times n$.

F.3.1 Propriétés des matrices U et V

Les matrices U et V sont des matrices unitaires respectivement de dimension m et n . Ainsi, ces deux matrices sont des matrices inversibles, et on a :

$$U^T \cdot U = U \cdot U^T = I_m$$

$$V^T \cdot V = V \cdot V^T = I_n$$

où I_m et I_n sont les matrices identités de rangs respectifs m et n .

On a alors :

$$U^{-1} = U^T$$

$$V^{-1} = V^T$$

De plus, les valeurs singulières des matrices U et V sont toutes égales à 1 :

$$\forall i \in [1; m] \quad \sigma_i(U) = 1$$

$$\forall i \in [1; n] \quad \sigma_i(V) = 1$$

On en déduit alors :

$$\kappa(U) = \frac{\sigma_{\max}(U)}{\sigma_{\min}(U)} = 1$$

$$\kappa(V) = \frac{\sigma_{\max}(V)}{\sigma_{\min}(V)} = 1$$

F.3.2 Propriétés de la matrice Σ

La matrice Σ est une matrice diagonale, de dimension $m \times n$, contenant p valeurs singulières de la matrice A , avec $p = \min(m, n)$.

Ainsi, on a :

$$m > n \implies p = n \implies \Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \sigma_n \\ 0 & 0 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & 0 \end{bmatrix}$$

$$m < n \implies p = m \implies \Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \sigma_m & 0 & \cdots & 0 \end{bmatrix}$$

On en déduit alors :

$$\kappa(\Sigma) = \frac{\sigma_{\max}(\Sigma)}{\sigma_{\min}(\Sigma)} \leq \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \kappa(A)$$

Ainsi, dans le pire des cas, le conditionnement de la matrice Σ est identique à celui de la matrice de départ A .

Pour la suite, nous considérerons que $\kappa(\Sigma) = \kappa(A)$.

Bibliographie

- [1] M. Quinson. *Découverte Automatique des Caractéristiques et Capacités d'une Plate-forme de Calcul Distribué*. PhD Thesis, ENS-Lyon, France, 2003.
- [2] I. Foster, C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco (USA), 1999.
- [3] I. Foster. *What is the Grid ? A Three Point Checklist*. GRIDToday, 2002.
- [4] M. Chetty et R. Buyya. *Weaving Computational Grids : How Analogous Are They With Electrical Grids ?* Computing in Science and Engineering, 2002.
- [5] V. Berstis. *Fundamentals of Grid Computing*. IBM Redpaper, 2002.
- [6] M. L. Bote-Lorenzo, Y. A. Dimitriadis et E. Gómez-Sánchez. *Grid Characteristics and Uses : a Grid Definition*. 1st European Across Grids Conference (ACG'03), Santiago de Compostela (Espagne), Février 2004.
- [7] F. Cappello, E. Caron, M. Daydé, F. Despres, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet et O. Richard. *Grid'5000 : A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform*. Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, Seattle (USA), 13-14 Novembre 2005.
- [8] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Despres, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi et I. Touche. *Grid'5000 : A Large Scale and Highly Reconfigurable Experimental Grid Testbed*. International Journal of High Performance Computing Applications, Vol. 20, 2006.
- [9] *Le Réseau National de télécommunications pour la Technologie, l'Enseignement et la Recherche*. <http://www.renater.fr/>
- [10] F. Khalil, B. Miegemolle, T. Monteil, H. Aubert, F. Coccetti et R. Plana. *Simulation of Micro Electro-Mechanical Systems (MEMS) on Grid*. 8th International Meeting on High Performance Computing for Computational Science (VECPAR'08), 2008.
- [11] F. Khalil, F. Coccetti, H. Aubert, R. Plana, Y. Denneulin, B. Miegemolle et T. Monteil. *Etude des Potentialités du Concept de Grille de Calcul pour la Simulation Electromagnétique de Micro-Systèmes Complexes*. 15èmes Journées Nationales Micro-ondes (JNM'2007), Toulouse (France), 23-25 Mai 2007.
- [12] F. Khalil, H. Aubert, F. Coccetti, R. Plana, Y. Denneulin, B. Miegemolle, T. Monteil et H. Legay. *Electromagnetic Simulation of MEMS-Controlled Reflectarrays Based on SCT in Grid Environment*. IEEE-AP-S International Symposium 2007, Honolulu (USA), 10-15 Juin 2007.
- [13] W. Smith et P. Wong. *Resource Selection Using Execution and Queue Wait Time Predictions*. Technical Report, NASA Advanced Supercomputing Division (NAS), Moffet Field (USA), Juillet 2002.

- [14] Y. Zhu. *A Survey on Grid Scheduling Systems*. Department of Computer Science, Hong Kong University of Science and Technology, 2003.
- [15] K. Krauter, R. Buyya et M. Maheswaran. *A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing*. Software Practice and Experience, 2002.
- [16] C. Kandagatla. *Survey and Taxonomy of Grid Resource Management Systems*. Technical Report, University of Texas, Austin (USA), 2003.
- [17] F. Dong et S. G. Akl. *Scheduling Algorithms for Grid Computing : State of the Art and Open Problems*. Technical Report 2006-504, School of Computing, Queen's University, Kingston (Ontario), Janvier 2006.
- [18] S. Basu, V. Talwar, B. Agarwalla et R. Kumar. *Interactive Grid Architecture for Application Service Providers*. First International Conference on Web Services, Las Vegas (USA), Juin 2003
- [19] A. Andrieux, D. Berry, J. Garibaldi, S. Jarvis, J. MacLaren, D. Ouelhadj et D. Snelling. *Open Issues in Grid Scheduling*. Official Technical Report of the Open Issues in Grid Scheduling Workshop, Edinburgh (Royaume-Uni), Octobre 2003.
- [20] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith et S. Tuecke. *A Resource Management Architecture for Meta-Computing Systems*. Workshop on Job Scheduling Strategies for Parallel Processing, 1998.
- [21] J. M. Schopf. *Ten Actions when Grid Scheduling : the User as a Grid Scheduler*. Grid Resource Management : State of the Art and Future Trends, 2004.
- [22] K. Baumgartner et B. W. Wah. *Computer Scheduling Algorithms : Past, Present and Future*. Information Sciences, Vol. 57 & 58, Elsevier Science Pub. Co., New York (USA), 1991.
- [23] M. Russell, G. Allen, T. Goodale, J. Nabrzyski et E. Seidel. *Application Requirements for Resource Brokering in a Grid Environment*. Grid Resource Management : State of the Art and Future Trends, 2004.
- [24] T. L. Casavant, et J. G. Kuhl. *A Taxonomy of Scheduling in General-purpose Distributed Computing Systems*. IEEE Transactions on Software Engineering, Vol. 14, Février 1988.
- [25] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik et P. Wong. *Theory and Practice in Parallel Job Scheduling*. Job Scheduling Strategies for Parallel Processing, Springer-Verlag, 1997.
- [26] D. Zoltán, P. J. Keleher. *Job-Length Estimation and Performance in Backfilling Schedulers*. 8th International Symposium on High Performance Distributed Computing, Redondo Beach (USA), 3-6 Août 1999.
- [27] A. W. Mu'alem et D. G. Feitelson. *Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling*. IEEE Transactions on Parallel and Distributed Systems, Vol. 12, Juin 2001.
- [28] D. G. Feitelson et M. A. Jette. *Improved Utilization and Responsiveness with Gang Scheduling*. Job Scheduling Strategies for Parallel Processing, Springer-Verlag, 1997.
- [29] D. G. Feitelson et L. Rudolph. *Parallel Job Scheduling : Issues and Approaches*. Job Scheduling Strategies for Parallel Processing, Springer-Verlag, 1995.
- [30] H. D. Kim et J. S. Kim. *An Online Scheduling Algorithm for Grid Computing Systems*. Grid and Cooperative Computing, Lecture Notes in Computer Science, Vol. 3033, 2004.

- [31] J. W. Jo et J. S. Kim. *A Scheduling Algorithm with Co-allocation Scheme for Grid Computing Systems*. International Conference on Grid and Cooperative Computing, Wuhan(Chine), 21-24 Octobre 2004.
- [32] H. Dail, O. Sievert, F. Berman, H. Casanova, A. Yarkahn, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo et I. Foster. *Scheduling in the Grid Application Development Software Project*. Grid Resource Management : State of the Art and Future Trends, 2004.
- [33] P. Pascal, S. Richard, B. Miegemolle et T. Monteil. *Tasks Mapping with Quality of Service for Coarse Grain Parallel Applications* Proceedings of Euro-Par 2005, Lisbonne (Portugal), 2005.
- [34] R. Buyya, D. Abramson, J. Giddy, et H. Stockinger. *Economic Models for Resource Management and Scheduling in Grid Computing*. The Journal of Concurrency and Computation : Practice Experience (CCPE), Wiley Press, Vol. 14, 2002.
- [35] I. E. Sutherland. *A futures market in computer time*. Communications of the ACM, Juin 1968.
- [36] J. Nakai. *Pricing Computing Resources : Reading Between the Lines and Beyond*. Technical Report, NASA Ames Research Center, Janvier 2002.
- [37] N. R. Nielsen. *The Allocation of Computer Resources - Is Pricing the Answer ?* Communications of the ACM, Vol. 13, Août 1970.
- [38] J. T. Hootman. *The Pricing Dilemma*. Datamation, Vol. 15, Août 1969.
- [39] H. Mendelson. *Pricing Computer Services : Queueing Effects*. Communications of the ACM, Vol. 28, Mars 1985.
- [40] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah et C. Staelin. *An Economic Paradigm for Query Processing and Data Migration in Mariposa*. 3rd International Conference on Parallel and Distributed Information Systems, Austin (USA), 28-30 Septembre 1994.
- [41] D. Ferguson, C. Nikolaou, J. Sairamesh et Y. Yemini. *Economic Models for Allocating Resources in Computer Systems*. Market-based Control : A Paradigm for Distributed Resource Allocation, World Scientific Press, Singapore, 1996.
- [42] B. A. Huberman et T. Hogg. *Distributed Computation as an Economic System*. Journal of Economics Perspectives, Vol. 9, 1995.
- [43] J. F. Kurose et R. Simha. *A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems*. IEEE Transactions on Computers, Vol. 38, Mai 1989.
- [44] D. Ferguson, Y. Yemini et C. Nikolaou. *Microeconomic Algorithms for Load-Balancing in Distributed Computer Systems*. 8th International Conference on Distributed Computing Systems, San Jose (USA), 13-17 Juin 1988.
- [45] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart et W. Stornetta. *Spawn : A Distributed Computational Economy*. IEEE Transactions on Software Engineering, Vol. 18, Février 1992.
- [46] O. Regev et N. Nisan. *The Popcorn Market - an Online Market for Computational Resources*. 1st International Conference on Information and Computation Economies, 1998.
- [47] P. Ghosh, N. Roy, S. K. Das et K. Basu. *A Game Theory Based Pricing Strategy for Job Allocation in Mobile Grids*. 18th International Parallel and Distributed Processing Symposium, 26-30 Avril 2004.

- [48] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, C. Staelin et A. Yu. *Mariposa : a Wide-Area Distributed Database System*. The International Journal on Very Large Data Bases (VLDB Journal), Vol. 5, Janvier 1996.
- [49] B. Miegommel. *Etude de Modèles Economiques pour les Grilles de Calcul*. Master's Thesis, INSA Toulouse, Septembre 2004.
- [50] B. Miegommel. *Modèles économiques pour le partage des ressources de grilles de calcul*. 7ème Congrès des Doctorants de l'Ecole Doctorale Systèmes, ENIT de Tarbes (France), 11 Mai 2006.
- [51] J. Li et R. Yahyapour. *Learning-Based Negotiation Strategies for Grid Scheduling*. 6th IEEE International Symposium on Cluster Computing and the Grid (CC-GRID'06), 2006.
- [52] B. N. Chun et D. E. Culler. *Market-Based Proportional Resource Sharing for Clusters*. Technical Report CSD-1092, University of California, Berkeley (USA), Janvier 2000.
- [53] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith et S. Tuecke. *A Directory Service for Configuring High-Performance Distributed Computations*. 6th IEEE International Symposium on High Performance Distributed Computing, Portland (USA), 5-8 Août 1997.
- [54] I. Foster et C. Kesselman. *Globus : a Metacomputing Infrastructure Toolkit*. International Journal of High Performance Computing Applications, Vol. 11, 1997.
- [55] R. Wolski, N. Spring et J. Hayes. *The Network Weather Service : A Distributed Resource Performance Forecasting Service for Metacomputing*. Journal of Future Generation Computing Systems, Janvier 1999.
- [56] R. Wolski. *Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service*. 6th IEEE International Symposium on High Performance Distributed Computing, Portland (USA), 5-8 Août 1997.
- [57] B. Gaidioz, R. Wolski et B. Tourancheau. *Synchronizing Network Probes to Avoid Measurement Intrusiveness with the Network Weather Service*. 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9 '00), 2000.
- [58] R. Wolski, N. Spring et J. Hayes. *Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid*. 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8), Août 1999.
- [59] M. L. Massie, B. N. Chun et D. E. Culler. *The Ganglia Distributed Monitoring System : Design, Implementation, and Experience*. Parallel Computing, Vol. 30, 2004.
- [60] F. Berman et R. Wolski. *The AppLeS Project : A Status Report*. 8th NEC Research Symposium, Berlin (Allemagne), Mai 1997.
- [61] D. Abramson, R. Buyya, et J. Giddy. *A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker*. Journal of Future Generation Computer Systems (FGCS), Octobre 2002.
- [62] R. Buyya, M. Murshed et D. Abramson. *A Deadline and Budget Constrained Cost-Time Optimization Algorithm for Scheduling Task Farming Applications on Global Grids*. International Conference on Parallel and Distributed Processing Techniques and Applications, 2002.
- [63] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, et O. Richard. *A Batch Scheduler with High Level Components*. Cluster computing and Grid 2005 (CCGrid05), Cardiff (Grande-Bretagne), 9-12 Mai 2005.

- [64] R. Buyya, D. Abramson et J. Giddy. *A Case for Economy Grid Architecture for Service-Oriented Grid Computing*. 10th IEEE International Heterogeneous Computing Workshop (HCW 2001), In conjunction with IPDPS 2001, San Francisco (USA), Avril 2001.
- [65] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, et D. Werthimer. *SETI@home : An Experiment in Public-Resource Computing*. Communications of the ACM, 2002.
- [66] P. Morin. *Coarse-Grained Parallel Computing on Heterogeneous Systems*. Proceedings of the 13th Annual ACM Symposium on Applied Computing (SAC'98), Atlanta (USA), 27 Février - 1er Mars 1998.
- [67] R. Wolski, J. S. Plank, J. Brevik et T. Bryan. *G-commerce : Market Formulations Controlling Resource Allocation on the Computational Grid*. International Parallel and Distributed Processing Symposium (IPDPS), San Francisco (USA), Avril 2001.
- [68] R. Buyya et M. Murshed. *A Deadline and Budget Constrained Cost-Time Optimize Algorithm for Scheduling Parameter Sweep Applications on the Grid*. GridSim Toolkit Release Document, Décembre 2001.
- [69] B. Miegemolle, R. Sharrock et T. Monteil. *Economic Model for Grid Resource Sharing*. International Conference on Grid Computing and Applications (GCA'07), Las Vegas (USA), 25-28 Juin 2007.
- [70] T. Bäck, D. B. Fogel et Z. Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Royaume-Uni, 1997.
- [71] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life* John Murray, Londres , 1859
- [72] A.E. Eiben et J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003, ISBN 3-540-40184-9.
- [73] H.-G. Beyer. *The Theory of Evolution Strategies*. Springer, Berlin, 2001.
- [74] L. J. Fogel, A. J. Owens et M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.
- [75] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, 1996.
- [76] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, 1989.
- [77] J.H. Holland. *Adaptation in Natural and Artificial Systems* MIT Press, University of Michigan Press, 1975.
- [78] K. A. De Jong. *Are Genetic Algorithms Function Optimizers ?* Parallel Problem Solving from Nature 2 (PPSN-II), Bruxelles (Belgique), 28-30 Septembre 1992.
- [79] K. A. De Jong. *Genetic Algorithms are NOT Function Optimizers*. Proceedings of the Second Workshop on Foundations of Genetic Algorithms, Vai (USA), 26-29 Juillet 1992.
- [80] N. Chaiyaratana et A. M. S. Zalzala. *Recent Developments in Evolutionary and Genetic Algorithms : Theory and Applications*. Second International Conference On Genetic Algorithms In Engineering Systems : Innovations And Applications (GA-LESIA 97), 2-4 Septembre 1997.
- [81] A. Ermedahl, F. Stappert et J. Engblom. *Clustered Worst-Case Execution-Time Calculation* IEEE Transactions on Computers, Vol. 54, Septembre 2005.
- [82] N. Audsley, A. Burns, R. Davis, K. Tindell et A. Wellings. *Fixed Priority Pre-Emptive Scheduling : An Historical Perspective*. Real-Time Systems, Vol. 8, 1995.

- [83] J. Ganssle. *Really Real-Time Systems*. Embedded Systems Conference (ESC SF), Avril 2001.
- [84] G. C. Buttazzo. *Hard-Real Time Computing Systems : Predictable Scheduling Algorithms and Applications*. Springer, Seconde Edition, Octobre 2004.
- [85] W. Smith, V. Taylor, et I. Foster. *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*. Job Scheduling Strategies for Parallel Processing, Springer Verlag, 1999.
- [86] L. J. Senger, M. J. Santana et R. H. Carlucci Santana. *An Instance-Based Learning Approach for Predicting Execution Times of Parallel Applications* Third International Information and Telecommunication Technologies Symposium, 2004.
- [87] R. Gibbons. *A Historical Application Profiler for Use by Parallel Schedulers*. Job Scheduling Strategies for Parallel Processing, Springer Verlag, 1997.
- [88] W. Smith, I. Foster et V. Taylor. *Predicting Application Run Times Using Historical Information* Lecture Notes in Computer Science, 1998.
- [89] A. W. Mu'alem et D.G. Feitelson. *Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling*. IEEE Transactions on Parallel and Distributed Systems, Juin 2001.
- [90] P. Puschner et C. Koza. *Calculating the Maximum Execution Time of Real-Time Programs*. Real-Time Systems, Vol. 1, Septembre 1989.
- [91] P. Puschner et A. Burns. *A Review of Worst-Case Execution-Time Analysis*. Journal of Real-Time Systems, Vol. 18, Mai 2000.
- [92] C. Liu et J. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*. Journal of the ACM, Vol. 20, Janvier 1973.
- [93] N. Williams. *WCET Measurement Using Modified Path Testing*. 5th International Workshop on Worst-Case Execution Time Analysis (WCET05), Espagne, Juillet 2005.
- [94] F. Mueller et J. Wegener. *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*. IEEE Real-Time Technology and Applications Symposium, Juin 1998.
- [95] J.-F. Deverge et I. Puaut. *Safe Measurement-Based WCET Estimation*. 5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.
- [96] T. Lundqvist et P. Stenström. *An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution*. Real-Time Systems, Vol. 17, Novembre 1999.
- [97] Y.-T. S. Li et S. Malik. *Performance Analysis of Embedded Software Using Implicit Path Enumeration*. Workshop on Languages, Compilers and Tools for Real-Time Systems, Californie (USA), Juin 1995.
- [98] J. Gustafsson. *Worst Case Execution Time Analysis of Object-Oriented Programs*. Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), San Diego (USA), 2002.
- [99] R. Kirner et P. Puschner. *Classification of Code Annotations and Discussion of Compiler Support for Worst-Case Execution Time Analysis*. 5th International Workshop on Worst-Case Execution Time Analysis (WCET05), Espagne, Juillet 2005.
- [100] R. Kirner. *The Programming Language WCETC*. Research Report n°2/2002, 2002.

- [101] C. Ferdinand, R. Heckmann, H. Theiling et R. Wilhelm. *Convenient User Annotations for a WCET Tool*. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET03), Porto (Portugal), 2003.
- [102] A. Mok , P. Amerasinghe, M. Chen et K. Tantisirivat. *Evaluating Tight Execution Time Bounds of Programs by Annotations*. IEEE Real-Time Systems Newsletter, Vol. 5, Mai 1989.
- [103] A. Ermedahl et J. Gustafsson. *Deriving Annotations for Tight Calculation of Execution Time*. Proceedings of EUROPAR'97, Août 1997.
- [104] J. Gustafsson. *Eliminating Annotations by Automatic Flow Analysis of Real-Time Programs*. 7th International Conference on Real-Time Computing Systems and Applications, Corée du Sud, 12-14 Décembre 2000.
- [105] J. Gustafsson, B. Lisper, C. Sandberg et N. Bermudo. *A Tool for Automatic Flow Analysis of C-Programs for WCET Calculation*. 8th IEEE Intl Workshop Object-Oriented Real-Time Dependable Systems (WORDS'03), Janvier 2003.
- [106] J. Gustafsson, A. Ermedahl et B. Lisper. *Algorithms for Infeasible Path Calculation*. 6th International Workshop on Worst-Case Execution Time Analysis (WCET06), Juillet 2006.
- [107] D. Kebbal et P. Sainrat. *Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis*. 6th International Workshop on Worst-Case Execution Time Analysis (WCET06), Juillet 2006.
- [108] D. Kebbal. *Automatic Flow Analysis Using Symbolic Execution and Path Enumeration*. 2006 International Conference on Parallel Processing Workshops (ICPPW'06), 14-18 Août 2006.
- [109] C. Sandberg. *Elimination of Unstructured Loops in Flow Analysis*. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET03), Porto (Portugal), 2003.
- [110] P. Puschner et A. Burns. *Writing Temporally Predictable Code*. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), San Diego (USA), 2002.
- [111] J. Gustafsson, B. Lisper, R. Kirner et P. Puschner. *Code Analysis for Temporal Predictability*. Real-Time Systems, Vol. 32, Mars 2006.
- [112] C. A. Healy, D. B. Whalley et M. G. Harmon. *Integrating the Timing Analysis of Pipelining and Instruction Caching*. 16th IEEE Real-Time Systems Symposium (RTSS '95), 5-7 Décembre 1995.
- [113] C. Healy, R. Arnold, F. Müller, D. Whalley et M. Harmon. *Bounding Pipeline and Instruction Cache Performance*. IEEE Transactions on Computers, Vol. 48, Janvier 1999.
- [114] S.-S. Lim, Y.H. Bae, C.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park et C.S. Ki. *An Accurate Worst-Case Timing Analysis for RISC Processors*. IEEE Transactions on Software Engineering, Vol. 21, Juillet 1995 .
- [115] S.-K. Kim, S.L. Min et R. Ha. *Efficient Worst Case Timing Analysis of Data Caching*. Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96), 10-12 Juin 1996.
- [116] R. White, F. Müller, C. Healy, D. Whalley et M. Harmon. *Timing Analysis for Data Caches and Set-Associative Caches*. Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97), 9-11 Juin 1997.

- [117] A. Colin et I. Puaut. *Worst Case Execution Time Analysis for a Processor with Branch Prediction*. Real-Time Systems, Vol. 18, Mai 2000.
- [118] T. Mitra et A. Roychoudhury. *Effects of Branch Prediction on Worst Case Execution Time of Programs*. Technical Report 11-01, National University of Singapore (NUS), Novembre 2001.
- [119] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Department of Information Technology, Uppsala University, Uppsala (Suède), Avril 2002.
- [120] J. Engblom et A. Ermedahl. *Pipeline Timing Analysis Using a Trace-Driven Simulator*. Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications, 13-15 Décembre 1999.
- [121] S. Schaefer, B. Scholz, S. M. Petters et G. Heiser. *Static Analysis Support for Measurement-based WCET Analysis*. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney (Australie), Août 2006.
- [122] I. Wenzel, R. Kirner, B. Rieder et P. Puschner. *Measurement-Based Worst-Case Execution Time Analysis*. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 16-17 Mai 2005.
- [123] M. Lindgren, H. Hansson et H. Thane. *Using Measurements to Derive the Worst-Case Execution Time*. 7th International Conference on Real-Time Computing Systems and Applications, Corée du Sud, 12-14 Décembre 2000.
- [124] S. M. Petters et G. Farber. *Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible*. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), 1999.
- [125] F. Stappert et P. Altenbernd. *Complete Worst-Case Execution Time Analysis of Straight-Line Hard Real-Time Programs*. Journal of Systems Architecture : the EUROMICRO Journal, Vol. 46, Février 2000.
- [126] F. Stappert, A. Ermedahl et J. Engblom. *Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects*. Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Atlanta (USA), 16-17 Novembre 2001.
- [127] R. Kirner et P. Puschner. *Transformation of Path Information for WCET Analysis during Compilation*. Proceedings of the 13th Euromicro Conference on Real-Time Systems, 13-15 Juin 2001.
- [128] A. Colin et G. Bernat. *Scope-Tree : A Program Representation for Symbolic Worst-Case Execution Time Analysis*. Proceedings of the 14th Euromicro Conference on Real-Time Systems, 19-21 Juin 2002.
- [129] C. Burguire et C. Rochange. *History-based Schemes and Implicit Path Enumeration*. 6th International Workshop on Worst-Case Execution Time Analysis (WCET06), Juillet 2006.
- [130] G. Ottosson et M. Sjödin. *Worst-Case Execution Time Analysis for Modern Hardware Architectures*. 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97), 1997.
- [131] Y.-T. S. Li, S. Malik et A. Wolfe. *Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software*. 16th IEEE Real-Time Systems Symposium (RTSS '95), 1995.

- [132] A. Downey. *Predicting Queue Times on Space-Sharing Parallel Computers*. 11th International Parallel Processing Symposium (IPPS '97), 1997.
- [133] R. Gibbons. *A Historical Profiler for Use by Parallel Schedulers*. Master's Thesis, University of Toronto, 1997.
- [134] S. Krishnaswamy, S.W. Loke, A. Zaslavsky. *Estimating Computation Times of Data-Intensive Applications*. IEEE Distributed Systems Online, Vol. 5, Avril 2004.
- [135] C. G. Atkeson, A. W. Moore et S. Schaal. *Locally Weighted Learning*. Artificial Intelligence Review, Vol. 11, Février 1997.
- [136] J. Schneider et A. Moore. *A Locally Weighted Learning Tutorial using Vizier 1.0*. Technical Report CMU-RI-TR-00-18, Robotics Institute, Carnegie Mellon University, Février 2000.
- [137] D. R. Wilson et T. R. Martinez. *Improved Heterogeneous Distance Functions*. Journal of Artificial Intelligence Research, Vol. 6, 1997.
- [138] N. H. Kapadia, J. A. B. Fortes et C. E. Brodley. *Predictive Application-Performance Modeling in a Computational Grid Environment*. 8th International Symposium on High Performance Distributed Computing, Redondo Beach (USA), 1999.
- [139] M. A. Iverson, F. Özgüner et L. Potter. *Statistical Prediction of Task Execution Times through Analytic Benchmarking for Scheduling in a Heterogeneous Environment*. IEEE Transactions on Computers, Vol. 48, Décembre 1999.
- [140] B. Miegemolle et T. Monteil. *Hybrid Method to Predict Execution Time of Parallel Applications*. International Conference on Scientific Computing (CSC'08), Las Vegas (USA), 14-17 Juillet 2008.
- [141] P. G. Ciarlet. *Introduction à l'Analyse Numérique Matricielle et à l'Optimisation*. Masson, Paris(France), 1982.
- [142] V. Teuliere et O. Brun. *Parallelisation of the Particle Filtering Algorithm and Application to Doppler-Bearing Tracking of Maneuvering Sources*. Parallel Computing, Vol. 29, Août 2003.
- [143] O. Brun et J.-M. Garcia. *Real-Time Parallel Particle Filtering*. 14th International Symposium of Mathematical Theory of Networks and Systems (MTNS'2000), Perpignan (France), 19-23 Juin 2000.
- [144] S. Richard, B. Miegemolle, T. Monteil et J.-M. Garcia. *Performance of MPI Parallel Applications*. International Conference on Software Engineering Advances (IC-SEA'2006), Papeete (France), 29 Octobre - 3 Novembre 2006.