Laboratoire ICPS-LSIIT
Université Louis Pasteur
Pôle API, Bd Sébastien Brant
F-67400 Illkirch

DEA d'informatique 2001/2002

Ordonnancement d'applications parallèles sur la grille

Dominique Stehly

20 juin 2002

Sujet proposé et encadré par
Stéphane Genaud, Maître de Conférence à l'Université Louis Pasteur

Remerciements

Comment ne pas remercier
Toutes les personnes qui m'ont encadré
Et qui ont su admirablement conseiller
Le petit stagiaire que j'étais.
Je suis particulièrement reconnaissant
Envers mon responsable de stage
Stéphane Genaud, toujours bienveillant
Qui m'a évité certains dérapages.
J'ai eu la chance de bénéficier
De sa passion et de sa rigueur
Qui dans mon travail m'ont fait avancer
Chaque jour, chaque heure.
Je n'oublie pas bien sûr
Schwarz Benjamin
Pour ses nombreux coups de mains
Et ses aides sur mesure.
Naturellement une petite pensée
Pour les autres membres du projet
Arnaud Giersch, Romaric David et Eric Violard
Ainsi qu'à tous les autres membres du laboratoire.
Je remercierai spécialement encore
Michel Salomon
Grâce à qui mon rapport
A pu être mis en page de la meilleure des façons.
Une pensée enfin
Pour Serge Wendling, mon rapporteur
J'espère que ces chapitres ne le rendront point
Fatigué ou de mauvaise humeur.

Table des matières

Table des figures

Introduction

Les dernières années ont vu l'émergence de ressources de calcul sans cesse plus puissantes ainsi qu'un potentiel de communication de plus en plus important avec le développement de réseaux ayant des débits toujours plus élevés. Néanmoins, certaines applications, principalement scientifiques, ne peuvent être traitées dans des temps raisonnables, même lorsque des super-calculateurs sont utilisés. Le développement des calculs sur cluster de processeurs a permis de répondre à certains desseins et fourni des résultats encourageants. Mais les clusters, par leur limitation à un ensemble géographique restreint sous le contrôle d'une seule autorité, ne sont pas pleinement satisfaisants. Le désir d'abroger ces limites géographiques et d'étendre le domaine de calcul à plus grande échelle pour offrir une performance de calcul jusqu'alors inégalée a contribué à l'apparition des grilles de calcul.

Les grilles sont des ensembles de ressources géographiquement distribuées et hétérogènes en nature c'est-à-dire que leurs architectures et leurs puissances de calcul sont distinctes. Ces ressources appartiennent à différents particuliers ou à différentes organisations qui possèdent leur propre politique de gestion, et sont fédérées grâce à une couche logicielle de type middleware. L'exploitation de la puissance de calcul des grilles passe par la programmation d'applications parallèles, telles des applications MPI, directement portables sur la grille.

Cette distribution de systèmes hétérogènes rend le problème d'ordonnancement sur grille délicat à résoudre. L'état global d'une grille ne pouvant être connu à un instant donné, il est de ce fait difficile de connaître à priori quelles sont les meilleurs ressources possibles sur lesquelles distribuer les applications. Dans ce contexte, des recherches doivent être menées pour étudier l'impact de l'hétérogénéité sur les performances des applications. Afin d'établir un classement des configurations de ressources susceptibles d'être sélectionnées, il s'agit de construire un système de mesures de performances pour attribuer à chaque configuration une estimation prévisionnelle du temps d'exécution de l'application sur cette configuration. Grâce à ce système, il conviendrait de pouvoir déterminer, au moment où un utilisateur décide de lancer une application parallèle, la configuration de ressources qui permette d'optimiser la durée d'exécution de l'application, et qui satisfasse dans le même temps des contraintes spécifiées par l'utilisateur, comme par exemple le nombre de processeurs désiré. La finalité de ce stage est de produire un outil automatisant ce processus de sélection de ressources.

Dans un premier temps, nous présentons brièvement le contexte de notre recherche. Dans un second temps, nous étudions et évaluons les modèles existants d'ordonnanceurs sur grille afin de déterminer les plus adéquats à notre projet. Dans une troisième partie, nous décrivons les constituants d'une grille. Puis nous proposons une solution à notre problème d'ordonnancement et exposons un algorithme évolutif pour la sélection de ressources dans un quatrième chapitre.

Enfin, dans une dernière partie, nous montrons, résultats à l'appui, l'efficacité de l'algorithme d'ordonnancement élaboré.

Chapitre 1

Contexte de la recherche

1.1 Cadre du projet

Notre travail se place dans le cadre du projet TAG (Transformation et Adaptations pour la Grille) mené au LSIIT (UMR 7005), équipe ICPS (Image et Calcul Parallèle Scientifique) de Strasbourg. Ce projet est mené en partenariat avec d'autres communautés scientifiques : la géophysique avec l'Institut de Physique du Globe de Strasbourg (UMR 7516), le calcul numérique avec l'Institut de Recherche Mathématique Avancée de Strasbourg (UMR 7501) et la chimie théorique avec les laboratoires de Chimie Quantique et de Modélisation Moléculaire de Strasbourg (UMR 7551). Les chercheurs de ces laboratoires utilisent des programmes manipulant des quantités très importantes de données et nécessitant une puissance de calcul considérable. Ces applications, se prêtant bien à une parallélisation pour réduire leurs temps d'exécution, sont implémentées à l'aide de la bibliothèque MPI (Message Passing Interface). MPI [Mes95] est une bibliothèque C ou Fortran portable permettant d'échanger des messages entre processus répartis sur un ensemble de processeurs.

L'objectif est d'exploiter au mieux la puissance de calcul fournie par la grille afin d'améliorer la performance d'exécution des programmes soumis. L'hétérogénéité d'une grille implique une inégalité des performances d'une application puisque les temps de communication et de calcul sont différents suivant l'ensemble des ressources sur lequel cette application est exécutée. Par conséquent, l'obtention d'un temps d'exécution le plus petit possible passe par une bonne sélection des ressources de la grille et par une transformation de code pour adapter la distribution des données et fournir un meilleur équilibrage de charge. La finalité de notre projet est d'aboutir à des outils automatisant ces processus de sélection de ressources et de transformation du code pour l'équilibrage de charge.

1.2 Traitement d'une application

Le traitement d'une application comporte trois phases principales : l'analyse de l'application, l'ordonnancement sur la grille, les transformations de code nécessaires à l'équilibrage de la charge.

1.2.1 Analyse de l'application

L'analyse de l'application est la première des trois principales phases du traitement d'une application. Son objectif principal est de déterminer la structure de l'application, à savoir l'organisation des phases de calcul par rapport aux communications. Le moyen le plus simple d'analyser cette structure est le *profiling* de l'application, c'est-à-dire l'analyse post-mortem de traces générées par l'application. Il est évidemment nécessaire d'avoir instrumenté l'application auparavant. A titre d'exemple, l'instrumentation peut tracer les appels à la primitive `MPI_Scatter()`. `MPI_Scatter()` permet à un processus source de distribuer ses données par blocs de même taille entre plusieurs processus. Dans le code instrumenté, cette primitive est remplacée par une fonction `log_scatter()` (transition T_2 sur la figure 1.1) qui permet, outre leur distribution effective, l'affichage de la quantité de données échangées. La compilation puis l'exécution du programme obtenu permettent de générer un fichier (`analyse.log` sur la figure 1.1) comprenant les traces de l'application c'est-à-dire des informations comme le volume de données à transférer durant l'exécution de l'application suivant des paramètres donnés. Ce fichier ne change pas pour une application donnée avec les mêmes paramètres, il n'est donc pas nécessaire de le regénérer si l'application s'exécute plusieurs fois dans ces conditions.

1.2.2 Ordonnancement de l'application

Définition de l'ordonnancement

Ordonnancement Le problème d'ordonnancement [LE99, CCE+93] consiste à organiser dans le temps la réalisation de tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchaînement) et de contraintes portant sur la disponibilité des ressources requises. Un ordonnancement constitue une solution au problème d'ordonnancement. Il est défini par le planning d'exécution des tâches (*ordre* et *calendrier*) et d'allocation des ressources et vise à satisfaire un ou plusieurs objectifs, comme par exemple la fin au plus tôt de l'exécution de l'ensemble des tâches.

Tâches Une tâche est une entité élémentaire localisée dans le temps par une date de début et de fin, dont la réalisation nécessite une durée, et qui consomme une ressource selon une certaine intensité. Selon les problèmes, les tâches peuvent être exécutées par morceaux, ou doivent être exécutées sans interruption ; on parle alors respectivement de problèmes préemptifs et non préemptifs. Lorsque les tâches ne sont soumises à aucune contrainte de cohérence, elles sont dites indépendantes.

Ressources La ressource est un moyen destiné à être utilisé pour la réalisation d'une tâche, elle est disponible en quantité limitée appelée capacité. Plusieurs types de ressources sont à distinguer. Une ressource est renouvelable si après avoir été allouée à une ou plusieurs tâches, elle est à nouveau disponible en même quantité : la quantité de ressource utilisable à chaque instant est limitée. Dans le cas contraire, elle est consommable : la consommation globale (ou cumul) au cours du temps est limitée. Une ressource est doublement contrainte lorsque son utilisation instantanée et sa consommation globale sont toutes deux limitées. Qu'elle soit renouvelable ou consommable, la disponibilité d'une ressource peut varier au cours du temps. Sa courbe de disponibilité est en général connue a priori, sauf dans les cas où elle dépend du placement de certaines tâches génératrices. On distingue par ailleurs, principalement dans le cas de ressources renouvelables, les ressources dédiées qui ne peuvent exécuter qu'une tâche à la fois et les ressources cumulatives qui peuvent être utilisées par plusieurs tâches simultanément, mais en nombre limité.

Contraintes Les contraintes expriment des restrictions sur les valeurs que peuvent prendre simultanément les variables de décision. On distingue :

- Contraintes temporelles
  - les contraintes de temps alloué, issues généralement d'impératifs de gestion et relatives aux dates limites des tâches ou à la durée totale d'un projet
  - les contraintes de cohérence technologique, ou contraintes de gammes, qui décrivent des relations d'ordre relatif entre les différentes tâches
- Contraintes de ressources
  - les contraintes d'utilisation de ressources qui expriment la nature et la quantité des moyens utilisés par les tâches, ainsi que les caractéristiques d'utilisation de ces moyens
  - les contraintes de disponibilité des ressources qui précisent la nature et la quantité des moyens disponibles au cours du temps

Toutes ces contraintes peuvent être formalisées sur la base des distances entre débuts de tâches ou potentiels.

Objectifs Dans la résolution d'un problème d'ordonnancement, on peut choisir entre deux grands types de stratégies, visant respectivement à l'optimalité des solutions, ou plus simplement à leur admissibilité. L'approche par optimisation suppose que les solutions candidates à un problème puissent être ordonnées de manière rationnelle selon un ou plusieurs critères d'évaluations numériques, construits sur la base d'indicateurs de performances. On cherchera donc à minimiser ou maximiser de tels critères. On note par exemple ceux :

- liés au temps : le temps total d'exécution ou le temps moyen d'achèvement d'un ensemble de tâches, le stock d'en-cours de traitement, différents retards (maximum, moyen, somme, nombre, etc.) par rapport aux dates limites fixées,
- liés aux ressources : la quantité, totale ou pondérée, de ressources nécessaires pour réaliser un ensemble de tâches, la charge de chaque ressource,
- liés à une énergie ou un débit,
- liés aux coûts de lancement, de production, de transport, mais aussi aux revenus, aux retours d'investissements.

Ordonnancement sur la grille

La seconde des trois phases du traitement d'une application est marquée par l'intervention de l'ordonnanceur, également appelé *scheduler*. Les sous-tâches d'une application à ordonnancer sont destinées à être exécutées en parallèle, ce sont des tâches indépendantes. Les ressources de la grille sont les stations de calcul. Elles ne sont pas consommables, mais bien renouvelables : une fois qu'un processseur a terminé le traitement d'une tâche, une partie de sa capacité se libère. Les ressources sont cumulatives pour la plupart : plusieurs applications peuvent être exécutées sur un processeur, en se partageant la puissance du CPU. Néanmoins, il existe aussi des ressources dédiées. Ce sont par exemple des processeurs réservés à certaines applications, utilisés dans le cas de qualité de service ou, plus souvent, les ressources munies d'un système de gestion de file. Ces processeurs ne traitent qu'une application à la fois. Les contraintes de temps et de ressources sont liées à l'utilisateur. C'est lui qui exprime des souhaits en terme de temps d'exécution maximal, quantité de RAM, taille de disque dur ou encore puissance de processeur nécessaires. La grille occasionne d'autres contraintes sur les ressources, comme la disponibilité d'un processeur, qui peut être mise en cause par un problème réseau ou par une charge CPU trop élevée. On ne considère aucune contrainte de cohérence vu que les tâches s'exécutant en parallèle sont indépendantes. La stratégie de résolution du problème d'ordonnancement sur notre grille vise à une optimalité des solutions en minimisant le critère de temps d'exécution. Cette stratégie vise à satisfaire au mieux l'attente de l'utilisateur qui est de voir son application s'exécuter le plus vite possible, sans se soucier des ressources sur lesquelles son application est distribuée.

En outre, l'ordonnanceur d'applications sur une grille possède quelques fonctionnalités particulières qui le distingue légèrement d'un ordonnanceur de tâches en général. Il doit choisir un ensemble de ressources en utilisant :
- les traces de l'application obtenues précédemment,
- l'annuaire de la grille lui permettant de s'informer sur l'ensemble des ressources,
- un modèle de performance lui permettant de déterminer un classement des ressources.

Il faut souligner que l'état de la grille est dynamique. Contrairement aux informations concernant une application donnée, celles concernant l'état des ressources changent au cours du temps. Ceci nous conduit à poser certaines hypothèses. Les débits réseaux et les charges CPU sont supposés connus au début de l'exécution grâce à l'annuaire et à NWS (que nous présentons dans 3.2.2, page 40) et constants durant l'exécution. Ces hypothèses sont réalistes pour des exécutions de courtes durées, par exemple quelques minutes. Lorqu'il s'agira d'obtenir un ordonnancement pour des applications ayant des durées d'exécutions plus longues, nous pensons qu'il sera utile de relancer l'ordonnancement périodiquement durant l'exécution. Ceci permettrait d'obtenir un gain de temps en cas de possiblité de diminution du temps de communication et/ou du temps de calcul. En outre, l'intervalle de temps entre l'ordonnancement et l'exécution doit être réduit pour garder le bénéfice de la distribution.

L'utilisateur, souhaitant profiter de la puissance de la grille pour pouvoir exécuter une application qu'il ne peut faire tourner sur une de ses machines par manque de mémoire ou pour tout simplement éviter une durée d'exécution trop grande, espère ne pas avoir à chercher lui-même les ressources à utiliser. Il désire tout bonnement exprimer des contraintes en terme de mémoire ou de temps d'exécution. L'objectif de l'ordonnanceur est donc de pouvoir déterminer, au moment où un utilisateur décide de lancer une application parallèle, la configuration de ressources de calcul qui permette d'optimiser la durée de l'application et de satisfaire dans le même temps les contraintes émises.

1.2.3 Equilibrage de charge

La troisième phase concerne l'équilibrage de charge sur l'ensemble des ressources sélectionnées lors de la phase d'ordonnancement. De nombreux programmes parallèles ont été conçus dans l'optique d'une exécution sur des machines parallèles qui offrent un ensemble de processeurs de puissance homogène et une interconnexion rapide et homogène entre ces processeurs. Une grille, en revanche, est composée d'un grand nombre de processeurs hétérogènes reliés par des canaux de communications offrant des débits nettement inférieurs à ceux d'une machine parallèle et de plus irréguliers. L'exécution de tels programmes sur une grille, à nombre de processeurs égal, entraîne une baisse de performance très importante. L'objectif est de diminuer la perte de performance en équilibrant la charge affectée aux processeurs en fonction de leurs puissances et de leurs liens réseaux. En pratique, cette phase aboutit à une transformation du code, par exemple un remplacement d'un `MPI_Scatter()` par un `MPI_Scatterv()` (transition T_1 sur la figure 1.1 qui permet à un processus source de distribuer ses données entre plusieurs processus mais par blocs de différentes tailles suivant les processus.

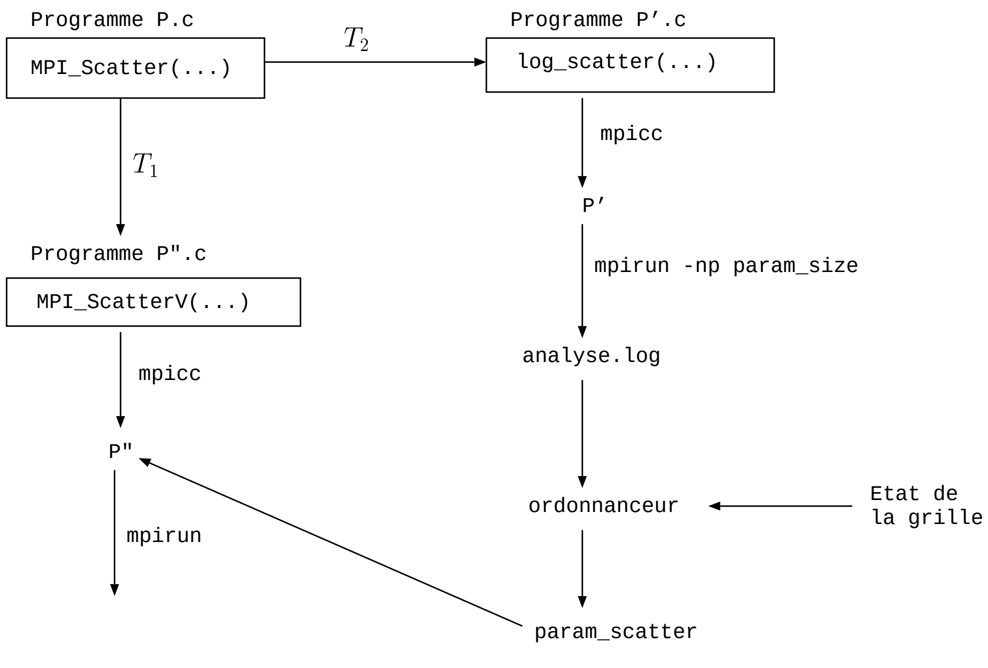

FIG. 1.1 – Traitement d'une application

Chapitre 2

Travaux relatifs aux ordonnanceurs de grille

2.1 Généralités

2.1.1 Schéma de fonctionnement d'un ordonnanceur

Les recherches menées jusqu'à présent sur l'élaboration d'ordonnanceurs de grille [Dat01, Sch00] ont permis de converger vers un schéma général de leur fonctionnement. Dans un premier temps, un ordonnanceur procède à une découverte des ressources. Il s'informe sur la disponibilité des ressources de la grille ainsi que leurs caractéristiques. Les renseignements puisés donnent une description des ressources à la fois statique, comme la configuration de la station (type de CPU, taille de la RAM, logiciels et services disponibles) mais aussi dynamique, comme le pourcentage de CPU utilisé ou l'état du réseau (estimation de la bande passante disponible). L'ordonnanceur doit de surcroît s'assurer qu'un utilisateur lui soumettant une requête d'utilisation possède bien les droits d'accès aux ressources listées. Dans un second temps, l'ordonnanceur se charge de sélectionner, au moyen d'un algorithme d'ordonnancement pensé et établi en fonction du contexte et des besoins, les ressources qui répondent au mieux à la requête de performance de l'application établie par l'utilisateur. Les ressources considérées sont celles identifiées durant la phase de découverte et l'algorithme est basé sur les informations obtenues précédemment. Enfin, dans un troisième temps, l'ordonnanceur interagit avec le gestionnaire de la ressource locale sélectionnée pour, soit débuter l'opération de manière imminente si l'état de la ressource le permet, soit planifier une exécution future en entrant en file d'attente si la ressource est momentanément en cours d'utilisation.

2.1.2 Classification

Outre l'aboutissement à un modèle de fonctionnement des ordonnanceurs, les travaux ont permis d'établir une taxinomie des ordonnanceurs. En effet, les ordonnanceurs peuvent être classifiés selon trois critères : leur organisation, leur politique d'ordonnancement et leur technique d'estimation sur les états des ressources.

Organisation L'organisation peut être centralisée, distribuée ou hiérarchique [BAG00a, FKT01, BCD00, BAG00b].

- **Centralisée** : Dans ce cas, l'opération d'ordonnancement est gérée par une unique entité, que l'on nomme *metascheduler*, qui a la responsabilité de traiter toutes les requêtes des utilisateurs. Le *metascheduler* possède une vue globale de l'ensemble des ressources (nous nommons cet ensemble le *metasystème*) ce qui rend possible une prise de décision optimale. En revanche, cette organisation pose problème en cas de panne du système dans lequel le *metascheduler* est implanté. Dans ce cas névralgique, aucune tâche ne peut alors être ordonnancée ce qui provoque à terme une inactivité du *metasystème* si la panne n'est pas réparée. Etant donné que toutes les applications sont soumises à un unique ordonnanceur, cette organisation présente également des risques de goulet d'étranglement. Elle convient aussi très peu dans des cas où la grille est constituée d'un nombre très important de ressources car il devient alors très difficile voire impossible pour le *metascheduler* de gérer la totalité de ces ressources dans des temps raisonnables.

- **Distribuée** : Dans une organisation distribuée, les décisions d'ordonnancement sont confiées aux applications et aux ressources locales. L'ordonnancement des tâches est effectué lors du lancement de l'application sur le système comprenant initialement les données du programme. Le mécanisme d'ordonnancement fait alors partie de l'application qui est libre de choisir les ressources qui lui conviennent le mieux. De même, chaque ressource est libre de décider de la manière dont vont être gérées les tâches qui lui sont soumises (file d'attente FIFO, système de priorité de certaines tâches). Cela signifie qu'une ressource procède à un ordonnancement local des tâches suivant sa propre politique. Ce second type d'organisation contrecarre les problèmes de pannes et de goulet d'étranglement évoqués pour la disposition centralisée. Toutefois, les décisions d'ordonnancement, basées sur des connaissances localement précises seulement, les informations concernant les ressources à distance pouvant être incomplètes ou obsolètes, donnent lieu couramment à une allocation de ressources sous-optimale. Le problème rencontré est en fait celui de la connaissance de l'état global d'un système distribué. Cet état global est délicat à déterminer puisque l'état d'une ressource peut avoir évolué durant le laps de temps nécessaire à la prise de connaissance de son état. Mais à grande échelle, obtenir une solution optimale n'est pas primordial. Générer un ordonnancement optimal, compte tenu de l'abondance des ressources, nécessiterait un dispositif important et un temps de calcul très élevé. Une mise en place d'un tel fonctionnement n'est guère concevable dans la réalité. D'une part, le temps mis pour trouver les meilleurs ressources à allouer serait en général bien trop grand par rapport au temps d'exécution de l'application. D'autre part, l'état de la grille est en perpétuelle évolution. Un ordonnancement optimal à un instant ne l'est plus forcément quelques secondes plus tard.

- **Hiérarchique** : Le troisième type d'organisation d'ordonnanceurs est une organisation dite hiérarchique. Les responsabilités d'ordonnancement sont distribuées sur une hiérarchie d'ordonnanceurs. Les ordonnanceurs de plus haut niveau se chargent de prendre les décisions concernant un très grand ensemble de ressources (échelle continentale par exemple). Les ordonnanceurs de bas niveau s'attèlent à gérer un petit ensemble de ressources. Enfin, ceux de plus bas niveau s'occupent des ressources locales. Dans ce modèle, les ordonnanceurs d'un niveau quelconque de la hiérarchie interagissent avec les ordonnanceurs de niveau supérieur et inférieur leur correspondant (leurs ordonnanceurs parents et enfants dans la hiérarchie) pour prendre les décisions. Ce type de disposition tente de garder les avantages des organisations centralisées et distribuées tout en éliminant leurs inconvénients. Le souci principal est la co-allocation d'une ressource par plusieurs ordonnanceurs n'ayant aucun lien dans la hiérarchie. Une coopération entre tous les ordonnanceurs est moins évidente à mettre en oeuvre que dans l'organisation distribuée.

Politique d'ordonnancement Le deuxième critère de classification des ordonnanceurs est la politique d'ordonnancement.

- **Politique orientée système** La politique dite orientée système est caractérisée par un désir d'exploiter au mieux la performance du *metasystème* et uniquement de lui. L'objectif est de maximiser son débit (nombre de processeurs ayant fini leur tâche à chaque unité de temps) et l'utilisation des processeurs (équilibre de charge) grâce à une utilisation optimale des ressources.

- **Politique orientée application** La politique d'ordonnancement dite orientée application (ou orientée utilisateur) est caractérisée par une volonté de maximiser la performance de l'utilisateur et uniquement de lui. L'objectif est de minimiser le temps global de réponse (temps de communication, temps d'attente, temps d'exécution) sans se soucier de la performance globale du *metasystème*.

Technique d'estimation Le troisième critère de classification d'ordonnanceurs est la technique d'estimation sur les états des ressources. Deux types d'ordonnanceurs sont distingués, les ordonnanceurs non-prédictifs, et les ordonnanceurs prédictifs. Pour déterminer la meilleure allocation, le but est d'estimer les performances d'exécution d'applications et d'utilisation des ressources.

- **Ordonnanceurs non-prédictifs** Les ordonnanceurs non-prédictifs effectuent des prédictions sur ces performances en supposant que l'état de la ressource ne va pas changer au cours de l'exécution. Cette technique conduit souvent à des résultats en dessous de ceux escomptés. Ceci est dû à la présence très probable d'effets de conflits sur les ressources choisies, comme le partage de la bande passante par exemple.

- **Ordonnanceurs prédictifs** Les ordonnanceurs prédictifs, quant à eux, utilisent des techniques statistiques ou heuristiques afin de prévoir les éventuels changements d'état des ressources. Ceci permet aux décisions d'ordonnancement, guidées par ces estimations, d'être plus judicieuses. En revanche, le coût de calcul est nettement supérieur à celui de la technique non-prédictive.

2.1.3 Observations

Les études menées sur ces différents genres d'ordonnanceurs ont permis d'observer qu'un ordonnanceur approprié à une grille devait posséder un certain nombre de caractéristiques essentielles. La grille est composée de plusieurs *organisations virtuelles* coexistantes. Une *organisation virtuelle* est un ensemble d'individus ou d'institutions ayant un projet commun et partageant leurs processeurs, leurs données, leurs logiciels ainsi que leurs résultats. Il paraît clair que chaque organisation virtuelle de la grille désire utiliser un ordonnanceur qui se prête bien à ses besoins, à sa politique. Le problème se posant alors réside dans le fait que les ordonnanceurs opérant simultanément et indépendamment, s'ils ne coopèrent pas entre eux, risquent de prendre des décisions entraînant des conflits. La nécessité de coopération entre ordonnanceurs indépendants incite à une organisation distribuée. En outre, un ordonnanceur se doit d'être prédictif afin d'obtenir de bonnes performances malgré les états dynamiques des ressources. Il faut également souligner que la politique d'ordonnancement doit pouvoir répondre aux attentes des utilisateurs soumettant un ensemble de jobs de taille considérable et nécessitant un haut débit aussi bien qu'aux utilisateurs soumettant des applications individuelles nécessitant un temps de réponse petit. La politique d'ordonnancement doit donc être à la fois orientée système et orientée application. Pour finir, notons une qualité complémentaire requise à savoir la fiabilité en terme de soumission de jobs. Comme les grilles sont caractérisées par une grande volatilité des ressources, il est fondamental de pouvoir resoumettre automatiquement le job en cas de panne d'une machine ou d'une partie du réseau et ceci sans repasser par l'utilisateur.

2.2 Stratégies d'ordonnancement

2.2.1 Méthodes de minimisation de la dégradation de performance

Comme évoqué précédemment, la principale difficulté d'un ordonnancement distribué est l'impossibilité de connaître l'état global de la grille. Aussi, des dégradations inopinées des performances des liens réseaux ou bien des taux de charge fortuitement élevés de processeurs choisis peuvent rendre l'ordonnancement infructueux. Des stratégies, visant à trouver les ordonnancements susceptibles d'être les moins exposés à d'éventuelles dégradations, ont été élaborées. Elles se basent sur l'étude de deux points cruciaux lors d'une exécution d'une application distribuée :

- *les temps d'attente jusqu'à l'entame des calculs sur les ressources choisies*, qui dépendent essentiellement de l'état du réseau puisque ces temps correspondent aux temps de tranfert de données. Ils peuvent également dépendre de la gestion de tâches des ressources (système de file d'attente, ...),
- *les temps de calculs sur les ressources choisies*, qui dépendent de l'état des ressources (vitesse du processeur, taille de la RAM, taux de charge du CPU).

Des études sur ces deux perspectives ont permis de converger vers un modèle d'ordonnancement [MBHJ98] basé sur deux critères. Le premier critère est la disponibilité des ressources qui permet de prédire quand chaque ressource pourra être utilisée. Deux cas sont alors à distinguer, celui où l'accès à la ressource est dit garanti (G : réservations de ressources, ressources dédiées à une application spécifique) et celui où l'accès est non garanti (N : nous ne pouvons pas exactement prévoir quand l'utilisation de la ressource nous sera permise). Le deuxième critère est l'utilisation de la ressource qui permet de prédire combien de temps la ressource sera utilisée. A nouveau, deux cas sont à distinguer, celui où l'exécution est dite déterministe (D : le temps d'utilisation de la ressource ne va pas dépasser nos prévisions) et celui où l'exécution est non déterministe (N : nous ne pouvons pas prévoir la durée d'utilisation de la ressource). Ces deux critères permettent d'obtenir quatre cas suivant lesquels une politique d'ordonnancement spécifique est adoptée.

GD Ceci est la situation la plus commode. Etant donné que la date d'accès et le temps de calcul sont connus, nous pouvons prévoir le temps d'exécution total de l'application suivant les ressources choisies. Il s'agit alors d'effectuer un précalcul d'un ordonnancement optimal. Le problème majeur se situe au niveau du temps de précalcul lors d'une recherche exhaustive des solutions parmi un nombre élevé de ressources disponibles. L'idée est alors d'utiliser un algorithme génétique pour converger vers une bonne solution sans avoir à parcourir l'espace des ressources en entier. Toutefois, les conditions de cette situation idyllique ne sont quasiment jamais vérifiées.

ND Dans cette situation, nous pouvons prévoir quand l'exécution de l'application va finir, une fois celle-ci lancée. En revanche, nous ne pouvons pas prévoir quand aura lieu ce lancement. L'idée est de procéder par recuit simulé [SM97] pour trouver les ressources vers lesquelles le flot de données est maximal offrant ainsi un temps d'accès minimal.

GN Dans cette configuration, nous pouvons prédire quand les tâches seront lancées sur les ressources. Par contre, nous ne sommes pas en mesure de savoir combien de temps leurs exécutions vont durer. L'idée est de construire un algorithme qui va sélectionner les ordonnancements les plus robustes [BM99]. Un ordonnancement est dit plus robuste qu'un autre s'il est moins vulnérable aux effets du hasard. En d'autres termes, il s'agit de sélectionner des ordonnancements en mesure de convenir du mieux possible aux requêtes émises même en cas de dégradation de la performance des ressources (bande passante, charge CPU). Concrètement, l'algorithme se base sur un graphe de dépendance des tâches. Trois sortes de dépendances peuvent engendrer un retard dans l'exécution.

- **Dépendance de données** : Une donnée nécessaire à une tâche t_2 met plus de temps que prévu à être calculée par une tâche t_1, ce qui affecte l'exécution de t_2.
- **Dépendance de communication** : Le transfert d'une donnée entre la tâche t_1 productrice et la tâche t_2 consommatrice met plus de temps que prévu à être réalisé, ce qui affecte l'exécution de t_2.
- **Dépendance de station** : Deux tâches t_1 et t_2, n'ayant aucun lien à priori, sont placées sur une même station H. L'exécution de t_2 est programmée après celle de t_1. La tâche t_1 met plus de temps que prévu à être exécutée, ce qui affecte le temps d'exécution de t_2.

Le graphe de dépendance des tâches, dont les arêtes représentent ces trois sortes de dépendances, est un diagramme de Pert qui nous informe des dates d'exécution au plus tôt et au plus tard de chaque tâche du graphe. Intuitivement, nous pouvons constater que pour deux ordonnancements aboutissant à un temps d'exécution global équivalent, le graphe de dépendance de l'ordonnancement le plus robuste comprendra le plus de marge entre les sous-tâches c'est-à-dire le moins de chemins critiques. Ainsi, d'éventuelles détériorations de l'efficacité des ressources sont moins susceptibles de provoquer un retard dans le temps d'exécution global de l'application ordonnancée. Un exemple, présenté sur la figure 2.1, nous montre que l'ordonnancement S_2 est plus robuste que S_1.

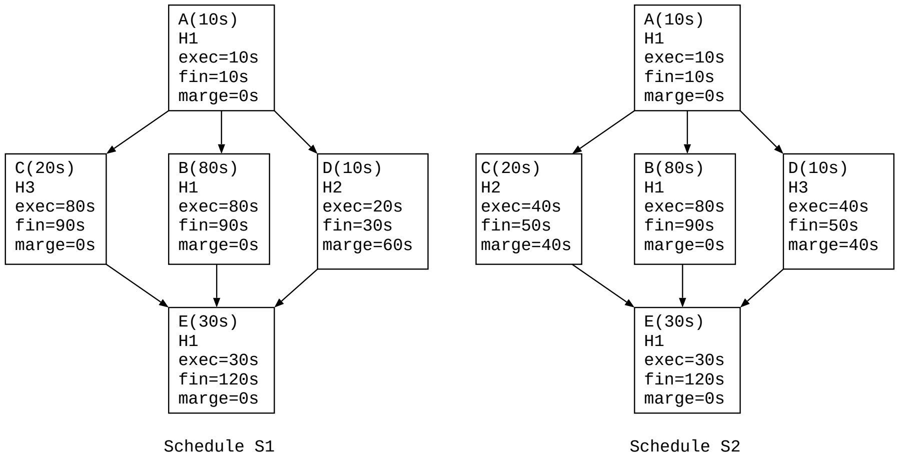

FIG. 2.1 – Comparaison de robustesse d'ordonnancement

NN Cette situation est la plus critique. Nous ne pouvons ni prévoir quand l'application pourra être lancée sur la ressource ni combien de temps l'exécution va durer. L'idée est alors d'utiliser un modèle de Bourse (Figure 2.2). L'allocation d'une ressource à une tâche se fait par consensus de marché. Les ressources sont vues comme des produits, les applications comme des consommateurs qui haussent ou baissent le prix des ressources suivant que la demande soit forte ou non. Dans cette configuration, un *metascheduler* est vu comme un client qui souhaite acquérir des ressources lui permettant de minimiser le temps d'exécution de l'application le plus possible. Pour cela, le *metascheduler* dispose d'un budget fonction de la taille de l'application. Plus l'application nécessite une grande quantité de calculs, plus le budget disponible du *metascheduler* est conséquent, lui permettant de ce fait d'améliorer son achat de ressources, aussi bien en qualité (il a les moyens d'acheter une ressource puissante malgré la forte demande) qu'en quantité (il peut éventuellement acheter plusieurs ressources si cela est bénéfique). Un *metascheduler* n'a aucune connaissance du stock de ressources. Il travaille en collaboration avec des *brokers* qui eux ont une connaissance globale du stock, fournissent des conseils et agissent par procuration pour le client. Deux catégories de *brokers* sont à distinguer : les *system brokers*, agents capables de localiser les ressources et d'effectuer les transactions, et les *local brokers*, qui gèrent les ressources disponibles sur chaque noeud de la grille. Les travaux exécutés sur les ressources sont vus comme des contrats. Un *local broker*, connaissant l'état du système local qu'il gère, effectue des offres de contrat lorsque la ressource est disponible, il établit alors un droit de vente de la ressource. Les *system brokers*, quant à eux, acquièrent des options sur des ressources ayant des droits de vente afin d'anticiper les besoins de consommateurs potentiels. Ils obtiennent ainsi des droits d'achats qu'ils se chargeront de vendre aux *metaschedulers* qui les contacteront en fournissant des contraintes sur la date de lancement de la tâche et le temps estimé d'utilisation des ressources. Les transactions entre *local brokers* et *system brokers* s'effectuent sur un système équivalent à une chambre de compensation.

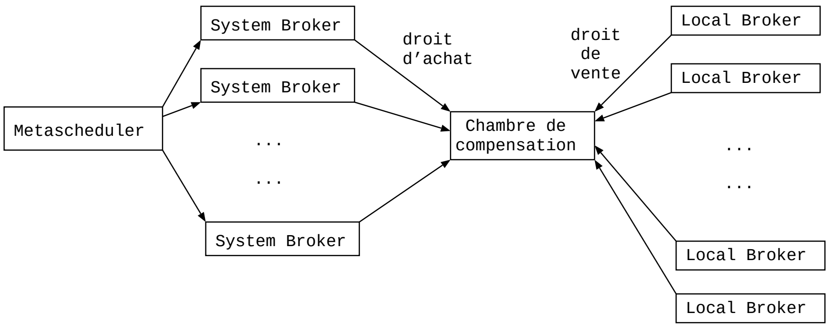

FIG. 2.2 – Architecture du modèle de Bourse

L'avantage de ce modèle est la mise en place de réservations des ressources de calcul, évitant ainsi des conflits et permettant de dédier l'ensemble d'un processeur à une tâche. De plus, un équilibrage de la charge s'effectue de manière intrinsèque. Les applications de taille importante possèdent les budgets les plus élevés et acquièrent ainsi les ressources les plus puissantes.

Toutefois, ce modèle ne tient pas compte du comportement des applications ordonnancées, de la manière dont communiquent les sous-tâches. L'étude du comportement de l'application est une approche très intéressante car l'ordonnancement orientée alors application fournit un résultat plus avantageux à l'utilisateur.

2.2.2 Ordonnancement orienté application

Afin d'obtenir des ordonnancements plus puissants, certaines recherches se sont tournées vers l'analyse du comportement des applications ordonnancées. Cette stratégie, se basant sur la structure de l'application, permet de minimiser de manière plus efficace les communications engendrées par l'application.

L'application modélisée en graphe acyclique de tâches Un arbre ou dag (graphe acyclique), modélise l'application à ordonnancer. Les noeuds du graphe représentent des tâches. Chaque tâche v possède un coût de calcul $w(v)$, qui décrit la taille de la tâche en terme de quantité de calculs. Chaque processeur de l'ensemble des ressources considérées possède un temps de cycle t_i qui caractérise sa puissance de calcul. Par exemple, si un processeur P_1 est deux fois plus rapide qu'un processeur P_2 alors $t_2 = 2 \times t_1$. Plus le temps de cycle est petit, plus le processeur est puissant. Le nombre de pas de temps nécessaire à l'exécution d'une tâche v sur le processeur P_i est alors :

$$w(v) \times t_i$$

On définit la date du début d'allocation d'une tâche v_i par $\sigma(v_i)$, et l'indice du processeur affecté à v_i par $alloc(v_i)$. Les tâches sont considérées non préemptives, une exécution commencée doit finir, aucun découpage de tâche n'est autorisé. Dans le graphe de tâches, les arêtes définissent les contraintes de précédence. Une arête entre deux tâches v_i et v_j est notée :

$$e_{i,j} : v_i \rightarrow v_j$$

Chaque arête $e_{i,j}$ est étiquetée par le volume de données transférées de v_i à v_j après l'exécution de v_i, l'étiquette est notée $data(i, j)$. Les frais de communication entre une tâche v_i placée sur le processeur P_q et une tâche v_j placée sur le processeur P_r sont données par la relation :

$$comm(i, j, q, r) = data(i, j) \times link(q, r)$$

où $link$ est une matrice 2×2 indiquant le temps requis pour transférer une unité de données d'un processeur à un autre. A tout niveau du graphe de tâches, on peut écrire la contrainte suivante :

$$\sigma(v_i) + w(v_i) \times t_q + comm(i, j, q, r) \leq \sigma(v_j)$$

L'objectif est de minimiser le temps d'exécution total du graphe de tâches c'est-à-dire obtenir une fin d'exécution de la dernière tâche du graphe le plus tôt possible. On cherche donc à minimiser :

$$\max_v \sigma(v) + w(v) \times t_{alloc(v)}$$

Basées sur ce modèle, des heuristiques telles HEFT et ILHA [BBR01] permettent de dériver des ordonnancements offrant des temps de complétion intéressants, en s'attelant à minimiser les communications.

- **l'heuristique HEFT**

  Cette heuristique se base sur la calcul du *bottom level* (BL) de chaque tâche. Le *bottom level* d'une tâche est la longueur du plus long chemin (en terme de calcul et de communication) qui va de la tâche vers un noeud de sortie du graphe (feuille) augmentée de la quantité de données nécessaires à envoyer pour que l'exécution de la tâche puisse débuter. Etant donné que l'on cherche à ce que la dernière tâche du graphe se termine le plus tôt possible, plus le *bottom level* d'une tâche est grand, plus la tâche est considérée comme urgente à traiter. Un classement de priorité de tâches est établi à partir des valeurs du *bottom level* de chaque tâche. HEFT consiste à considérer la tâche la plus prioritaire et à lui assigner le processeur offrant un temps de complétion minimal. Il s'agit non seulement d'ordonnancer la tâche mais aussi les communications vers cette tâche pour qu'elles aient lieu le plus tôt possible. On réitère l'opération pour chaque tâche dans l'ordre du classement.
  Un exemple est développé dans l'annexe A (A.1, page 69).

– **l'heuristique ILHA**

L'objectif de l'heuristique ILHA est d'assurer un meilleur équilibrage de la charge. Contrairement à l'heuristique HEFT, ILHA considère plusieurs tâches (sous-ensemble de tâches) à chaque étape et non plus une seule. ILHA alloue aux processeurs choisis lors de l'algorithme un nombre de tâches proportionnel à la puissance de calcul. La seconde idée directrice de ILHA, pour minimiser les communications, est de placer les tâches dépendantes sur le même processeur si le placement n'engendre pas un excès de travail du processeur. On définit une borne maximale de la charge de travail d'un processeur P_i par la valeur :

$$c_i = \lceil \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times W_{total} \rceil$$

où W_{total} correspond à la charge totale de l'application. Les tâches restantes de la liste, n'ayant pu être placées, sont traitées par l'heuristique HEFT. Comme HEFT, un exemple est expliqué en annexe A (A.2, page 69).

Néanmoins, ce modèle d'ordonnancement, ne tenant pas compte de la manière dont les tâches de l'application sont calculées, nous semble un peu trop approximatif dans le cadre de notre projet. Des études plus poussées ont permis d'élaborer des modèles d'ordonnanceur se référant à une description se voulant la plus précise possible de l'application. Pour cela, les phases de calcul et de communication sont analysées pour pouvoir établir des fonctions d'estimation de la quantité de données transférées durant l'exécution de l'application ainsi que la manière dont se déroulent les calculs. Cette stratégie d'ordonnancement se veut la plus contigüe possible de l'application et donc de son utilisateur.

Modèle d'ordonnancement guidé par la structure des applications et des ressources
Ce type de stratégie, plus poussé, analyse finement le comportement de l'application pour tirer des informations primordiales concernant les communications inter-tâches. Outre les renseignements sur les applications, ce modèle d'ordonnanceur considère des indications sur l'organisation des ressources, ce qui lui permet de distribuer localement les applications de manière à minimiser les communications.

Des recherches comme celles menées sur Prophet [WG95, WZ97, WZ99] ou bien sur MARS (*Metacomputer Adaptive Runtime System*)[GR96], explorent ce type de tactique d'ordonnancement. Prophet est un système fournissant un ordonnancement d'exécution de tâches pour des calculs data-parallèles dans des réseaux de stations de travail hétérogènes. Guidé par des informations sur les programmes et les ressources, Prophet sélectionne des ressources de la grille en découpant l'application en tâches, distribuant ces tâches sur les ressources sélectionnées, exploitant la structure de l'application pour la prise de décision d'ordonnancement, utilisant des informations sur la granularité de l'exécution pour sélectionner le meilleur nombre de processeurs. L'opération d'ordonnancement se déroule en deux temps, par l'intermédiaire d'une phase de partitionnement suivie d'une phase de placement.

- **Partitionnement**

  L'idée est d'utiliser des informations sur la granularité de calcul de l'application et sur l'état du réseau (volume de communications autorisé) pour déterminer le meilleur nombre de processeurs sur lesquels l'application sera décomposée et distribuée. L'application est ainsi partitionnée en autant de sous-tâches que le nombre déterminé. Obtenir un meilleur grain de parallélisme contribue à diminuer le temps d'exécution de l'application. Lorsqu'une application est parallélisée, dans un premier temps, le fait d'ajouter des processeurs fait diminuer le temps d'exécution. Puis, à partir d'un certain seuil, plus on ajoute de processeurs, moins la parallélisation est efficace, ceci dû à des temps de communications trop grands. Obtenir le meilleur grain de parallélisme signifie trouver ce seuil.

- **Placement**

  Il s'agit ensuite de déterminer sur quelle ressource chaque tâche va être placée. Pour cela, l'idée est de se baser sur des informations sur la structure de l'application et sur les caractéristiques des ressources pour construire des fonctions de coût de calcul et de coût de communication. L'objectif est évidemment de sélectionner les ressources qui permettent de réduire le temps d'exécution de l'application. Lors du placement d'une tâche sur le processeur qui lui a été choisi, une partie du domaine de données de l'application est assignée à la tâche. L'assignation d'une portion proportionnelle à la puissance du processeur permet d'obtenir un équilibrage de la charge et donc de réduire le temps d'exécution total de l'application.

  Afin de minimiser les temps de communication, l'approche consiste à placer les tâches sur des clusters de processeurs. Un cluster de processeurs est un ensemble de processeurs homogènes interconnectés avec une bande passante importante. Ceci permet de minimiser les coûts de communication de manière efficace.

Deux fonctions de coût sont construites à partir des informations et permettent d'obtenir une estimation du temps de communication de l'application ainsi que de son temps de calcul.

- **Fonction de coût de communication**

  Elle dépend de la topologie de communication de l'application, de la topologie d'interconnexion des clusters, de la taille des messages échangés ou encore du nombre de processeurs communiquants. Trois topologies de communication sont explorées :

  - *unidimensionnelle*
    Les processeurs échangent des messages avec leurs deux processeurs voisins (ou leur voisin, si le processeur est à une extrémité du ruban). Cette topologie correspond à un partitionnement simple de l'application.

– *anneau*
  Chaque processeur reçoit des messages d'un voisin et envoie des messages à son autre voisin. Cette topologie se prête bien aux algorithmes systoliques par exemple.

– *arbre*
  Cette topologie est utilisée pour des applications d'opérations telles les réductions.

Une fonction de coût de communication est définie spécifiquement à la topologie de l'application. Une description détaillée est fournie en annexe B (B.1, page 73). Les objectifs sont de maintenir les communications locales pour éviter les pénalités de routage, d'exploiter la bande passante des clusters, de minimiser les coûts de communication intraclusters et interclusters.

– **Fonction de coût de calcul**

  Elle dépend de la quantité de calcul à exécuter (taille du problème), du nombre et de la puissance des processeurs selectionnés. Une définition complète se trouve en annexe B (B.2, page 74).

– **Estimation du coût total**

  Elle prend en compte les frais de calcul et de communication.

$$T_{exe} = nbcycles \times (T_{cal} + T_{comm})$$

  où *nbcycles* est le nombre de cycles d'itération de l'exécution.
  La problématique est donc de minimiser T_{exe}.

Il n'est pas réalisable de procéder à une exploration de toutes les configurations possibles de processeurs car les possibilités sont exponentielles en nombre de clusters et de processeurs. Deux heuristiques ont été élaborées pour permettre d'éviter une recherche exhaustive des solutions.

– **Heuristique H_1**

  Cette heuristique se place dans le cadre d'un *metasystème* avec des capacités de communication identiques entre les ressources. Dans cette hypothèse, il s'agit de baser la recherche sur la puissance de calcul des clusters. Les clusters sont donc classés selon la puissance cumulée de leurs processeurs. On commence par explorer le meilleur cluster. On attribue à chaque tâche un processeur disponible. S'il n'y a plus de processeurs disponibles et s'il reste des tâches à placer, on explore le cluster suivant dans le classement, et ainsi de suite.

– **Heuristique H_2**

  Cette heuristique se place dans un cadre plus général, où les capacités de communication du *metasystème* sont hétérogènes. Un simple classement des clusters selon la puissance n'est alors plus approprié car le coût de routage vers un cluster puissant peut être trop pénalisant par rapport à un routage vers un cluster certes moins puissant mais nécessitant des temps de communication inférieurs. L'heuristique H_2 est basée sur la valeur de T_{exe} définie précédemment, qui tient compte à la fois des temps de calcul T_{cal} et des temps de communication T_{comm}. Tout d'abord, on établit un nouveau classement d'exploration des clusters. En procédant toujours dans l'ordre des clusters établi par H_1, on cherche, **pour chaque cluster pris individuellement**, celui permettant d'obtenir une valeur de T_{exe} minimale. On peut remarquer que si un cluster ne dispose que d'un nombre inférieur de ressources par rapport aux nombres de tâches à placer, certaines de ces tâches sont, lors de cette phase d'évaluation de clusters uniquement, considérées comme en attente sur certaines ressources derrière des premières tâches déjà placées. Les clusters sont ensuite classés suivant la valeur minimale de T_{exe} obtenue. On explore chaque cluster dans l'ordre du classement en exécutant deux phases pour chaque cluster :

- *Phase 1*
  On emploie la même stratégie que dans H_1. On sélectionne tous les processeurs disponibles et on attribue une tâche à chaque processeur sélectionné.

- *Phase 2*
  On essaie de réduire la composante T_{comm} de T_{exe}. Si nous sommes dans le cas d'un placement interclusters (toutes les tâches n'ont pas pu être placées sur le même cluster), on considère parmi les clusters explorés celui qui engendre le plus de temps de communication T_{comm}. On enlève une des tâches placées sur un processeur sélectionné et on la place sur le cluster en cours d'exploration, ceci tant que le temps de communication global diminue et tant qu'il existe des processeurs disponibles sur le cluster en cours d'exploration. A ce stade, nous avons sélectionné des processeurs diminuant le temps de communication mais appartenant à un cluster moins bien classé. Ce sont donc des processeurs peut être moins puissants, engendrant ainsi une hausse du temps de calcul. Il s'agit donc de considérer la configuration, parmi celles obtenues avant et après la minimisation de la composante T_{comm}, qui permet d'obtenir un temps d'exécution minimal.

Soient s le nombre de processeurs déjà sélectionnés, v le nombre de processeurs voulu et d la fonction qui, à partir d'un cluster, retourne le nombre de ses processeurs disponibles.
Classer les clusters selon leur puissance de calcul
Pour i $\leftarrow$ 1 à X // nb de clusters à considérer suivant le temps maximal d'ordonnancement
    Calculer $T_{exe}[C_i]$
Classer les clusters C_i, $i \in [1, X]$ selon les valeurs $T_{exe}[C_i]$
$explore \leftarrow 1$ // indice du cluster en cours d'exploration
$s \leftarrow 0$
Tant que $s \neq v$
    Tant que $d(C_{explore}) \neq 0$
        Sélectionner un processeur de $C_{explore}$
        $d(C_{explore}) \leftarrow d(C_{explore}) - 1$
        $s \leftarrow s + 1$
    config $\leftarrow$ sauvegarder configuration de sélection actuelle
    $T_{comm_{new}} \leftarrow 0$
    Tant que $(d(C_{explore}) \neq 0$ et $T_{comm_{new}} < T_{comm_{globale}})$
        $C_{max} \leftarrow \{C_k \mid T_{comm}[C_k] = max_{j \in [1, explore]}\{T_{comm}[C_j]\}\}$
        Retirer un processeur de C_{max} de la sélection
        Sélectionner un processeur de $C_{explore}$
        Calculer $T_{comm_{new}}$
        Si $T_{comm_{new}} < T_{comm_{globale}}$
            config $\leftarrow$ sauvegarder configuration de sélection actuelle
    Revenir à config
    $explore \leftarrow explore + 1$

Pour pouvoir satisfaire les utilisateurs mais surtout pour que l'optimisation de *scheduling* ait un sens, l'algorithme de sélection de ressources doit se terminer dans un temps suffisamment borné. Il est donc primordial d'établir un temps de recherche proportionnel à la taille de l'application. Par exemple, un temps de recherche inférieur à 10 % au temps d'exécution estimé de l'application peut être requis. Cette contrainte peut être satisfaite en limitant le nombre de clusters explorés dans les classements établis par les heuristiques.

L'équilibrage de la charge s'effectue lors de la décomposition du domaine de données de l'application. L'objectif est d'assigner à un processeur une quantité de travail proportionnelle à sa puissance de calcul. Une description de la manière dont cette quantité de travail est calculée se trouve en annexe B (B.3, page 74).

Modèle d'ordonnancement avec modèle de performance de l'application Le projet AppLeS [BW97] vise à développer des méthodes d'ordonnancement d'applications s'exécutant dans un *metasystème* hétérogène distribué. La politique d'ordonnancement mise en oeuvre est orientée application, le choix des ressources se fait en vue de satisfaire au mieux l'utilisateur et non en vue d'optimiser la gestion de l'ensemble des ressources. L'objectif est donc de sélectionner les ressources qui conviennent le mieux à l'utilisateur. Pour cela, les ressources sont évaluées selon leur impact sur l'application. En fonction de ses caractéristiques et des contraintes sur les qualités requises spécifiées par l'utilisateur, une ressource est ainsi jugée plus ou moins avantageuse pour l'application.

Le modèle AppLeS consiste en une organisation distribuée d'agents d'ordonnancement appelés des *AppLeS* (*Application-Level Scheduling*). Chaque application possède son propre *AppLeS* dont le rôle est de déterminer et de mettre en oeuvre un ordonnancement adapté à l'application. L'état du *metasystème* changeant continuellement, un *AppLeS* utilise à la fois des informations statiques et dynamiques pour pouvoir déterminer le meilleur ordonnancement pour l'utilisateur. En fait, la décision est basée sur la combinaison d'informations spécifiques aux systèmes, mais aussi sur des critères de performance voulus par l'utilisateur, sur une étude des communications pour notifier l'importance éventuelle de la bande passante et de la latence du réseau, ainsi que des informations dynamiques permettant d'évaluer l'état d'un système à un instant donné. Pour bien prendre en compte l'hétérogénéité du *metasystème*, chaque ressource est considérée comme une entité avec ses propres attributs et offrant une performance variable selon le temps en fonction de son état. Un *AppLeS* a la faculté d'évaluer des ressources sur la performance obtenue au moment où l'application est exécutée et non à l'instant où les informations sont récoltées. Pour cela, un *AppLeS* emploie un système de prédiction de l'état de la ressource. De plus, pour diminuer les conflits d'accès systèmes et réseaux, un *AppLeS* est capable d'estimer dynamiquement les ressources disponibles par d'autres méthodes de prédiction. Un autre moyen pour éviter les conflits est la mise en place d'un système de réservation de ressources ou de garantie de qualité de service, mais ces solutions ne sont pas toujours possibles sur la totalité des systèmes.

Une fois avoir déterminé un ordonnancement, un agent *AppLeS* se charge de mettre en oeuvre cet ordonnancement en interaction avec l'infrastructure de gestion des ressources du *metasystème*. Un agent *AppLeS* n'est cependant pas un système de gestion de ressource, il s'appuie sur des toolkit tel Globus pour procéder à ces opérations. D'un point de vue structurel, un agent *AppLeS* est composé de quatre sous-systèmes et d'un agent Coordinateur qui jouent chacun un rôle précis dans la prise de décision.

- **Sélectionneur de ressource** Son rôle est de retenir, parmi la totalité, différentes combinaisons possibles de ressources. Il opère par filtrage pour ne retenir qu'un ensemble de ressources qui satisfassent les critères de disponibilité et de caractéristiques escomptés.

- **Planificateur** Sa tâche est de générer, à partir de différentes combinaisons de ressources possibles, un ordonnancement potentiellement acceptable par l'utilisateur (c'est-à-dire en mesure de répondre à ses exigences).

- **Prédicteur de performance** Il se charge de fournir une estimation de la performance des ordonnancements candidats. Une note est établie selon la métrique de performance souhaitée par l'utilisateur.

- **Réalisateur** Sa fonction est de mettre en oeuvre le meilleur ordonnancement sur les ressources cibles en contactant leur système de gestion.

Le fonctionnement de ces sous-systèmes d'un agent *AppLeS* se base sur le *Pool d'Information* mis à disposition et partagé par tous les composants d'un *AppLeS*. Le *Pool d'Information* est l'ensemble des informations spécifiques à l'application, spécifiques aux systèmes et dynamiques sur l'état des ressources. Cette base de données est remplie par l'intermédiaire de trois sources d'information.

NWS fournit des informations dynamiques sur l'état du système et prévoit la charge des ressources à la date à laquelle l'application sera exécutée. Cette source d'information est utilisée pour prévoir dynamiquement la performance que chaque ressource peut offrir à une date ultérieure. NWS capte les performances pouvant être obtenues dans tout le *metasystème*. Les données captées sont stockées dans une base de données servant en entrée à un ensemble de modèles de prédiction. A partir des données extraites de la base de données et de ces modèles de prédiction, NWS prévoit les futures performances offertes par chaque ressource et enregistre les informations de prédiction dans les clients *AppLeS* intéressés.

– *Capture de données*

  Des capteurs de données enregistrent la performance qu'une ressource est capable de délivrer à la date où la mesure est prise. Par exemple, un capteur sur un lien réseau, effectue périodiquement des mesures de la bande passante disponible sur ce lien. Deux catégories de capteurs sont à distinguer, les capteurs actifs et les capteurs passifs. Un capteur passif utilise des données déjà disponibles et les modifie suivant des règles de comparaison d'états du système. Un capteur actif, quant à lui, mesure la performance des expériences qu'il exécute sur la ressource qu'il contrôle.

– *Prédiction de performance*

  NWS tient à sa disposition plusieurs modèles de prédiction qu'il invoque dynamiquement en passant des mesures de performances en paramètres. Ces modèles reposent sur des méthodes statistiques qui leurs sont spécifiques. La méthode fournissant des informations se rapprochant le plus de la réalité varie au cours du temps suivant l'état de l'environnement. C'est la raison pour laquelle NWS sélectionne plusieurs méthodes de prédiction et favorise la prise en compte d'une des méthodes suivant les conditions.

– *Enregistrement*

  Les informations de NWS sont utilisées principalement par le prédicteur de performance et le sélectionneur de ressource de l'agent *AppLeS*. Deux schémas de propagation de l'information sont possibles : par interrogation et par rappel. Dans le premier cas, l'agent *AppLeS* contacte NWS et demande simplement les données de prévision. Dans le second cas, l'*AppLeS* enregistre en plus une condition qui, une fois satisfaite, va déclencher un rappel de NWS vers l'*AppLeS* avec les nouvelles valeurs de prédiction.

Une interface utilisateur fournit des informations sur la structure, les caractéristiques, l'implantation de l'application et de ces tâches ainsi que des informations sur les critères de performance et les contraintes d'exécution spécifiés par l'utilisateur.

Les applications sont organisées en différents modèles ce qui aide l'*AppLeS* dans l'utilisation des prédictions. Ce tri des applications est basé selon les informations fournies par l'utilisateur et sur les exécutions antérieures des applications de même classe. Si l'utilisateur n'a donné aucun renseignement sur son application, celle-ci est considérée comme appartenant à la classe par défaut des applications.

L'utilisateur fournit des informations à l'agent via l'interface utilisateur. Le coordinateur utilise ces informations pour filtrer les ordonnancements non satisfaisant et ceux qui semblent ne pas pouvoir répondre à la demande de l'utilisateur. Si l'utilisateur ne provoque aucune information, des valeurs par défaut sont alors substituées. Guidé par les informations de l'interface utilisateur, le sélectionneur de ressources repère un ensemble de ressources prometteuses et le soumet au Coordinateur. Des informations sur les droits d'accès, les attributs des ressources et les directives de l'utilisateur permettent de procéder à un filtrage pour ne conserver qu'un ensemble de ressources potentiellement acceptables. Le sélectionneur de ressources classent ensuite les ressources selon la valeur d'une fonction ayant comme paramètres la puissance de calcul et la puissance de communication vers cette ressource. A l'aide de ce classement, il effectue une première reconnaissance de configurations de ressources intéressantes. Pour chaque configuration, le planificateur génère un ordonnancement potentiel. Le coordinateur utilise ensuite le prédicteur de performance pour évaluer chaque ordonnancement en termes d'objectifs de performances désirées par l'utilisateur. L'ordonnancement qui optimise le mieux les objectifs de l'utilisateur est choisie pour la mise en oeuvre par le réalisateur.

Un *AppLeS* utilise des modèles de structure d'application pour mieux être guidé dans ses choix d'ordonnancement. A partir de la classe de l'application, il peut dériver une structure d'estimation de temps d'exécution qui est utile pour le planificateur. Ceci permet d'élaborer des modèles d'estimation adaptés aux types d'applications et d'effectuer des choix plus judicieux. Deux exemples de modèle sont présentés en annexe C (page 75) : le calcul maître-esclave et les codes PMHD3D [DOB+00].

Rescheduling Malgré les efforts consentis pour lutter contre les problèmes d'information dûs à l'évolution continuelle de l'état du *metasystème*, grâce aux informations dynamiques et aux prédictions, des doutes quant aux choix pris subsistent toujours. En effet, la charge et la disponibilité des ressources continuent de varier pendant l'exécution de l'application, c'est-à-dire après le *scheduling* de celle-ci. Dans un souci de maintien d'une bonne performance et pour éviter de ne plus satisfaire les requêtes de l'utilisateur à cause d'événements inopinés, il peut s'avérer profitable de procéder à un *rescheduling*. La condition pour lancer un *rescheduling* éventuel est la suivante :

$$T_{rest} \gg T_{resched} + T_{nouv}$$

Le temps estimé d'exécution est supérieur au temps mis pour *rescheduler* et exécuter l'application sur une nouvelle configuration de ressources choisies.

Négociation inter-*AppLeS* Un scénario critique consiste à ce que plusieurs *AppLeS* choisissent simultanément une même ressource qualifiée de bonne pour leurs utilisateurs. Grâce aux prédictions, ils peuvent se rendre compte que la ressource n'est plus disponible dans le futur et procèdent tous à un *rescheduling* de leur application sur une autre ressource. La ressource n'est ainsi pas utilisée car aucun *AppLeS* ne l'assigne au bout du compte. Pour contrer ce problème, deux idées sont à l'étude. La première est de mettre en oeuvre un système de Bourse pour l'assignation des ressources. La seconde idée est d'organiser les *AppLeS* hiérarchiquement, certains *AppLeS* prioritaires bénéficient de qualité de service et de réservations de ressources.

Chapitre 3

Les constituants d'une grille

Le déploiement des grilles repose sur l'utilisation d'un intergiciel (middleware), brique logicielle placée entre les applications et le système d'exploitation, et permettant en l'occurrence aux applications de communiquer dans un environnement distant. Nous présentons ici le logiciel Globus, le middleware actuellement le plus utilisé pour construire des grilles.

3.1 Globus

Le toolkit développé dans le projet Globus [FK97] comprend un ensemble de composants implémentant des services de sécurité, localisation de ressources, gestion de ressources, accès aux données ou encore communication, ceci afin de faciliter la création d'une grille de calcul exploitable. Plutôt que de fournir un modèle de programmation uniforme monolithique (d'un seul bloc), le toolkit Globus fournit un ensemble de services à partir desquels et sur lesquels les développeurs peuvent à leur gré construire des outils ou applications.

3.1.1 Les différentes couches de la structure du toolkit Globus

Couche 1 : Grid Fabric Elle est composée de toutes les ressources matérielles : ordinateurs, systèmes d'exploitation, systèmes de gestion de ressource locale, mémoire ...

Couche 2 : Grid Services Cette couche gère les composants de la Grid Fabric. Elle est composée de modules fournissant des services spécifiques (information, gestion de ressource, accès à des données distantes ...). Chacun des modules a une interface bien définie, que les services de la couche 3 utilise pour invoquer les mécanismes désirés. De surcroît, chaque module fournit une implémentation qui utilise les opérations de bas niveau pour implémenter ces fonctionnalités. Voici quelques exemples de services fournis par le toolkit Globus :

- Grid Security Infrastructure (GSI) : services fournissant authentification et autorisation concernant les utilisateurs
- Grid Information Service (GIS) ou Metacomputing Directory Service (MDS) : annuaire des ressources de la grille
- Globus Resource Allocation Manager (GRAM) : service d'allocation de ressources

Couche 3 : Application Toolkits Cette couche est composée de commandes de soumission de job à distance, MPICH-G2 (une implémentation de la bibliothèque MPI adaptée à l'environnement Globus).

Couche 4 : Applications Cette couche supérieure est composée des applications clients, des travaux utilisant les ressources de la grille.

4	Applications	Astrophysique, Géophysique, Biologie
3	Application Toolkits	Calcul à distance, grille de données (ex: MPICH-G2)
2	Grid Services	Protocoles, authentification, politique, instrumentation, gestion de ressource, découverte, événements
1	Grid Fabric	Ordinateurs, mémoire, réseau, services associés

FIG. 3.1 – La structure du toolkit Globus

3.1.2 Globus Security Infrastructure (GSI)

Le GSI est un service d'identification des utilisateurs émettant des requêtes vers des noeuds de la grille. Une autorité de certification, dans laquelle tous les administrateurs de la grille ont confiance, délivre à chaque utilisateur un nom unique (Distinguished Nanme (DN)) et une signature électronique (certificat). L'identification repose sur la vérification de la signature électronique portée par chaque utilisateur. Cette PKI (Public Key Infrastructure) joue un rôle important pour les grilles, puisqu'elle permet à chaque utilisateur enregistré de bénéficier d'espace disque protégé sur l'ensemble de la grille. Pour exécuter une application sur plusieurs machines, un utilisateur n'a besoin que d'un seul compte. Lors d'une requête émise par cet utilisateur, une machine sollicitée possède un fichier décrivant l'association entre le DN de l'utilisateur et l'espace disque local associé.

3.1.3 Globus Grid Information Service (GIS)

Le GIS est l'annuaire de Globus. Il joue un rôle fondamental dans l'environnement de la grille puisque la découverte de ressources et les prises de décisions lors de l'ordonnancement se basent sur l'infrastructure de cet annuaire. Les types d'informations sont les suivants :

- informations statiques et dynamiques sur les ressources de calcul : configuration de la station, état de la station, liste de services, protocoles et dispositifs que l'on peut trouver sur la station, politique d'accès
- état du réseau et prévisions
- dépôts des logiciels : localisation des packages des logiciels

Le GIS fournit une infrastructure pour enregistrer et gérer les informations statiques et dynamiques sur l'état et les composants de la grille de calcul. Ce service est implémenté à travers le protocole LDAP [FFK+97]. LDAP est le protocole d'annuaire sur TCP/IP. Les annuaires permettent de partager des bases d'informations sur le réseau interne ou externe. Ces bases peuvent contenir toute sorte d'information, que ce soit des coordonnées de personnes ou bien des données système. LDAP fournit :

- le protocole permettant d'accéder à l'information contenue dans l'annuaire
- un modèle d'information définissant le type de données contenues dans l'annuaire
- un modèle de nommage définissant comment l'information est organisée et référencée
- un modèle fonctionnel qui définit comment on accède à l'information
- un modèle de sécurité qui définit comment données et accès sont protégés
- un modèle de duplication qui définit comment la base est répartie entre serveurs
- des APIs pour développer des applications clientes
- LDIF, un format d'échange de données

Chaque noeud de la grille enregistre dans l'annuaire des caractéristiques le concernant comme la description du matériel (processeurs, espace disque, ...) ainsi que des informations dynamiques comme le taux d'occupation de la mémoire. Chaque utilisateur peut donc utiliser les requêtes standard LDAP pour retirer ces informations.

Le nouveau modèle de l'annuaire comprend une hiérarchie d'annuaires à deux niveaux :

- d'une part, des collecteurs clients GRIS (Grid Resource Information Service) que l'on peut interroger directement,
- d'autre part, des serveurs GIIS (Grid Index Information Service) qui collectent les annuaires locaux.

3.1.4 Le service de gestion de ressources dans Globus

Globus Resource Allocation Manager (GRAM) Le GRAM (figure 3.2) est le service d'allocation de ressources. Présent sur chaque noeud de la grille, il est responsable de la création et de la gestion des processus lancés sur ce noeud après avoir reçu une requête de travail. Ce service est implémenté par un démon nommé gatekeeper. La requête est exprimée à travers la soumission, soit d'une simple commande Unix, soit d'un job. Le GRAM a pour rôle de :

- traiter des soumissions de requêtes utilisateurs
- lancer la gestion et la surveillance d'un job à distance
- périodiquement mettre à jour le GIS

General-purpose Architecture for Reservation and Allocation (GARA) L'objectif de GARA est de fournir des applications avec une méthode de réservation de ressources comme l'espace disque, cycles CPU, bande passante pour garantir une qualité de service point-à-point (QoS).

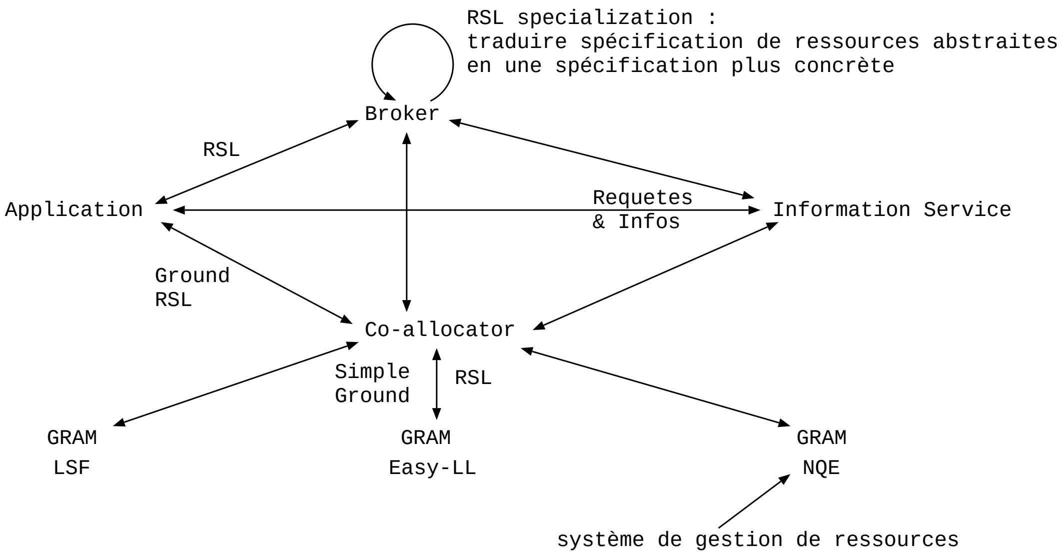

FIG. 3.2 – Architecture de gestion de ressources

3.2 La prédiction de performance des ressources

La grille pouvant être considérée comme un système géographiquement distribué de ressources, le problème classique de l'état global de ce système distribué se pose alors. L'état global de la grille ne peut pas être connue à un instant donné. Certes, certaines des informations concernant la grille sont statiques, comme la vitesse des processeurs des noeuds. En revanche, certaines informations, comme la bande passante disponible entre les noeuds de la grille, sont dynamiques. Le temps de récupérer ces informations, celles-ci sont déjà obsolètes. La prédiction de performance des ressources joue alors un rôle très important dans un environnement de type grille. Elle permet de limiter les erreurs d'estimation lors de l'ordonnancement, vu que ce dernier se base sur des valeurs de bande passante et de latence pour minimiser les temps de communication ou encore de la charge CPU pour minimiser le temps de calcul, et d'obtenir un ordonnancement plus efficace. Deux principaux outils sont utilisés dans les environnements de type grille.

3.2.1 Le système REMOS

Le système REMOS (Resource Monitoring System) [DGK+01] a été développé afin de fournir aux applications distribuées des informations concernant les ressources du réseau. La capacité de REMOS à pouvoir effectuer des mesures sur des ressources hétérogènes et destinées à différentes applications fait de ce système un outil approprié aux environnements de type grille. En effet, REMOS possède trois caractéristiques essentielles qui sont l'extensibilité, la flexibilité et la portabilité. Pour répondre aux besoins d'extensibilité, REMOS est capable de fournir aux applications un niveau de précision d'information approprié en filtrant les informations non requises. Le filtrage permet également un caractère de flexibilité par le fait que différents utilisateurs requièrent différents types d'information et REMOS s'adapte donc aux différentes situations.

Enfin, REMOS s'assure que les techniques de mesure utilisées soient adaptées à la variété de systèmes d'exploitation, d'architecture de réseaux et autres matériels. D'un point de vue architectural, REMOS est constitué de trois catégories d'éléments : des collecteurs, des modelers et des prédicteurs.

Collecteurs Le rôle des collecteurs est de recueillir les informations obtenues grâce à des capteurs et d'acheminer les résultats vers le modeler. Pour des raisons d'extensibilité, les collecteurs sont organisés de manière hiérarchique. Au niveau le plus bas se trouvent des collecteurs locaux responsables de la récolte d'informations sur le LAN correspondant. En amont se trouvent des collecteurs globaux responsables d'obtenir des informations sur les connexions inter-LANs.

- **SNMP Collector** Certains collecteurs de REMOS recueillent directement les informations des routeurs et des switchs par le biais du protocole SNMP (Simple Network Management Protocol). Le protocole SNMP est un protocole définissant le dialogue entre une station de contrôle et un noeud du réseau. Il permet de connaître l'état d'un appareil et de mesurer le trafic sur les liens du réseau.

- **Bridge Collector** Les switchs présents dans des réseaux de type Ethernet ne fournissent pas d'information explicite sur la topologie du réseau. C'est pourquoi, des bridge collector sont utilisés pour déterminer la topologie d'un LAN Ethernet.

- **Benchmark Collector** Bien que SNMP permette d'obtenir un système d'information efficace, REMOS ne peut généralement pas obtenir un accès SNMP à des informations sur des WANs ou bien sur des réseaux dans lesquels l'administrateur REMOS n'a de compte sur aucune machine. Dans ce cas, REMOS utilise des collecteurs benchmark qui déterminent la performance des ressources du réseau par l'intermédiaire de tests. Une fois les caractéristiques des liens du réseau établies, les collecteurs benchmarks donnent leurs résultats au collecteur maître.

- **Collecteur maître** Le collecteur maître se trouve au plus haut niveau de la hiérarchie des collecteurs. Il se charge de rassembler toutes les informations recueillies par les collecteurs de plus bas niveau. Le modeler de REMOS soumet une requête à son collecteur maître. Ce dernier identifie les réseaux comprenant les stations présentes dans la requête ainsi que les réseaux intervenant dans la connexion de ces stations. Le collecteur maître divise la requête en la répartissant sur les collecteurs responsables des ressources en question. Il réunit enfin les réponses qu'il combine. Le résultat obtenu est retourné au modeler.

Notons que REMOS comprend des collecteurs spécifiques aux LANs, d'autres aux WANs et certains aux réseaux sans fil. De plus, l'administrateur peut choisir le mode d'exécution des collecteurs :

- mode à la demande : le collecteur recueille les informations lorsqu'il reçoit une requête.
- mode périodique : le collecteur recueille les informations à intervalles de temps périodiques.

Modelers Un modeler fournit une API (Application Program Interface) à l'application qui lui est associée et communique avec le collecteur maître pour obtenir des informations désirées par la requête de l'utilisateur émise à travers l'API. Chaque application possède son propre modeler ce qui permet une flexibilité de l'information puisque suivant l'application, les informations obtenues par le collecteur maître sont interprétées par un modèle de prédiction spécifique.

Prédicteurs Les prédicteurs ont la charge de générer des prédictions sur l'état du réseau à partir d'un historique des mesures. Deux sortes de fonctionnement de prédicteurs ont été développées.

- **Mode client-serveur** A un instant donné, le prédicteur prend en entrée un vecteur de mesures et renvoie la prédiction résultante. Ce mode, où chaque client est considéré au cas par cas, permet l'intégration d'options désirées par le client, comme le pouvoir de spécifier la quantité de valeurs prises en compte de l'historique ou encore d'indiquer la date à laquelle sera lancée l'application et ainsi faire des prédictions sur la période d'exécution.

- **Mode flot** Le prédicteur reçoit des flots de mesures et retourne des flots de prédiction. L'interêt de ce mode de fonctionnement est l'amortissement du coût de calcul de prédictions sur plusieurs clients. Les résultats du prédicteur sont donc utilisés pour plusieurs applications.

L'idéal pour le système REMOS serait de pouvoir choisir entre un fonctionnement en mode client-serveur lorsque la spécificité de l'application est essentielle, ou un mode flot minimisant des temps de calculs inutiles dans les cas appropriés. Lorsqu'une prédiction est demandée, le modeler utilise le toolkit RPS (Resource Prediction System). RPS est un système indépendant à REMOS. L'interaction entre les deux systèmes se fait uniquement dans le cadre des prédictions. RPS fournit des services de prédiction à REMOS tandis que REMOS fournit les services de mesures à RPS. RPS possède une bibliothèque de prédiction incluant divers modèles de prédiction. Une fonction de la bibliothèque est appliquée suivant l'application et un patron contenant la structure de l'application est associé. Le patron est choisi parmi les modèles disponibles en analysant l'application ou bien est créé par l'utilisateur lui-même, RPS mettant à disposition une interface pour le faire.

3.2.2 Network Weather Service (NWS)

Tout comme le système REMOS, NWS [WSH99] est un outil très intéressant pour les environnements de type grille. NWS fournit des informations dynamiques sur l'état du système et prévoit la charge des ressources à la date à laquelle l'application sera exécutée. Cette source d'information est utilisée pour prévoir dynamiquement la performance que chaque ressource peut offrir à une date ultérieure. NWS capte les performances pouvant être obtenues dans tout le *metasystème*. Les données captées sont stockées dans une base de données servant en entrée à un ensemble de modèles de prédiction. A partir des données extraites de la base de données et de ces modèles de prédiction, NWS prévoit les futures performances offertes par chaque ressource et enregistre les informations de prédiction auprès des clients intéressés. D'un point de vue architectural, NWS est composé de quatre catégories d'éléments : processus continuels, serveurs de nom, capteurs et prédicteurs. Ces composants interagissent pour aboutir à des prédictions de performance.

Processus continuel Son rôle est de rechercher les mesures et les sauvegarder de manière continuelle. La fonction étant de générer des prédictions, il est évident que le système ne doit pas stocker indéfiniment les mesures sauvegardées. Ainsi, la taille du fichier comportant les mesures demeure égale au fil du temps. Les données sont gérées dans une file, dès qu'une nouvelle mesure est sauvegardée, la plus ancienne est effacée.

Serveur de nom Chaque processus NWS enregistre le nom de sa localisation auprès du serveur du nom. De cette manière, le serveur de nom se chargera par la suite de contacter les capteurs de performances correspondants aux ressources impliquées dans la prédiction ainsi que d'associer la sauvegarde des mesures obtenus aux bonnes ressources.

Capteur Un capteur de données enregistre la performance qu'une ressource est capable de délivrer à la date où la mesure est prise. NWS comporte deux types de capteurs : capteur CPU et capteur réseau.

- **Capteur CPU** Un capteur CPU de NWS combine les informations des primitives UNIX *uptime* et *vmstat* avec des tests périodiques d'occupation du CPU pour fournir des mesures de la quantité de CPU disponible sur une ressource. La quantité de CPU disponible est considérée comme la fraction de temps CPU qu'un processus d'un utilisateur ayant pleine priorité peut occuper. Le capteur CPU effectue des mesures sur une minute pour déterminer la fraction de temps CPU qu'un processus pourrait obtenir s'il était traité à ce moment là.

- **Capteur réseau** Pour mesurer les performances réseaux de liens, c'est-à-dire bande passante disponible entre deux stations et latence observée, un capteur réseau de NWS transfère une quantité fixe de données entre les deux stations en question. Néanmoins, cette méthode peut s'avérer critique lorsque le nombre de capteurs augmentent dans un réseau, engendrant un trafic non négligeable. Pour éviter que ces transferts n'affectent les performances réseaux, les capteurs réseaux ont été organisés de manière hiérarchique (transferts entre sous-ensembles de capteurs et non plus point-à-point).

Prédicteur Pour générer une prédiction, un prédicteur de NWS traite un historique de mesures stocké par un processus continuel. Il applique un ensemble de modèles statistiques [WSH98] de prédiction sur les séries de mesures qui lui sont soumises. Le modèle statistique pris en compte est choisi de manière dynamique. Lorque le prédicteur doit fournir une estimation, il applique chaque modèle de prédiction sur le début de la série de mesures puis compare avec les valeurs effectivement observées dans la fin de série. Le modèle engendrant une prédiction avec le taux d'erreur le plus faible est choisi pour les estimations futures.

Chapitre 4

Un algorithme évolutif pour la sélection de ressources

4.1 Introduction

4.1.1 Définitions

Algorithme évolutif Un algorithme génétique est un algorithme d'exploration d'un ensemble de solutions d'un problème. Le déroulement d'un tel algorithme est fondé sur les mécanismes de la sélection naturelle et de la génétique. Selon Charles Darwin [Dar59], la sélection naturelle est le processus selon lequel les individus mieux adaptés à leur environnement tendent à survivre en plus grand nombre et à se reproduire plus fréquemment que leurs congénères moins bien lotis. Les calculs sont ainsi basés sur les principes de survie de structures les mieux adaptées et sur les échanges d'information concernant le patrimoine génétique. L'idée générale d'un algorithme génétique est de promouvoir les meilleurs individus et de les faire évoluer pour améliorer leur adaptation ou qualité [Hol92]. Un algorithme génétique est une méthode d'optimisation dans laquelle un ensemble, appelé *population*, de solutions potentielles, appelées *individus*, est progressivement mis à jour par le biais d'un mécanisme de sélection et par des opérations génétiques : le *croisement* et la *mutation*.

Opérateurs génétiques Les éléments d'une population sont évalués par l'intermédiaire d'une fonction appelée *fonction fitness*. Elle permet d'établir un classement des individus. Les meilleurs individus, dont la structure répond le mieux aux besoins du problème, sont ensuite sélectionnés puis utilisés dans une phase de reproduction dans laquelle ils transmettent une partie de leur bagage génétique. Cette opération, appelée croisement, permet de former un nouvel individu qui, héritant des gènes de parents bien classés, a des chances d'être lui aussi une bonne solution. Chaque population étant calculée à partir de la précédente, un algorithme génétique simule l'évolution d'une population qui, à terme, sera composée des meilleurs individus qui auront survécu au fil du temps. Un tel algorithme permet de converger rapidement vers une bonne solution sans avoir à procéder de manière exhaustive en évaluant chaque candidat possible. De cette façon, le temps de résolution d'un problème possédant un ensemble de solutions immense est réduit de manière importante. Cependant, une convergence trop précoce, dite locale, peut survenir, si toutes les régions de l'espace des solutions ne sont pas explorées. C'est la raison pour laquelle une opération de mutation est mise en oeuvre lors de la génération d'une nouvelle population. Certains gènes, choisis aléatoirement, de certains individus pris au hasard dans la population prennent ainsi une nouvelle valeur. Ces individus dits mutants permettent d'explorer aléatoirement l'espace des solutions et d'éviter une convergence trop rapide.

La figure 4.1 représente un espace de recherche d'individus solutions en abscisse et la valeur du fitness de ces individus en ordonnée. Le problème revient à maximiser (ou minimiser selon les critères voulus) la fonction fitness qui est fonction des individus c'est-à-dire trouver l'individu pour lequel la valeur de la fonction fitness sera maximale (ou minimale).

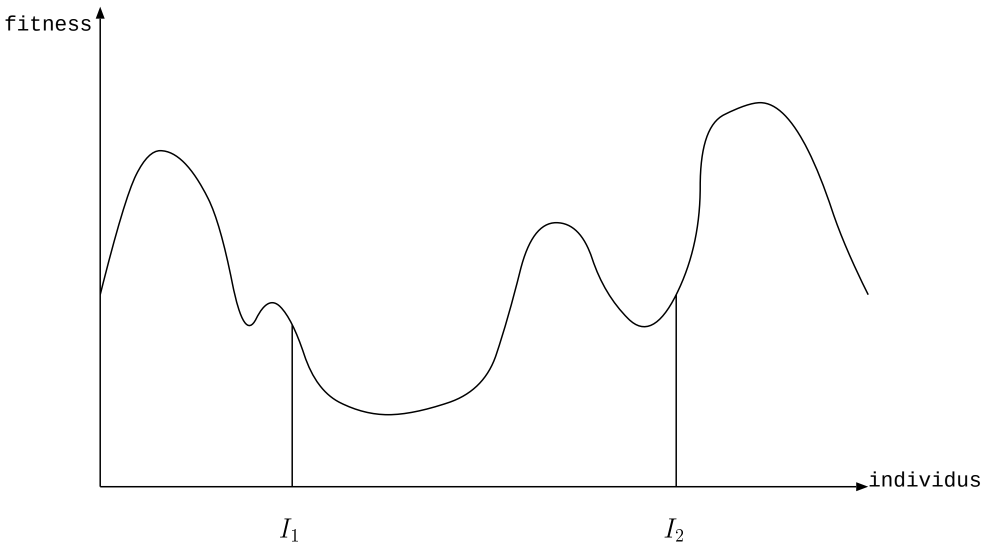

FIG. 4.1 – Espace de recherche

4.1.2 Schéma de fonctionnement d'un algorithme évolutif

Soit I l'ensemble des individus de l'espace de recherche. Les individus d'une population P de cardinal n ($P \subset I^n$) à un instant t sont représentés par des vecteurs de gènes notés $\vec{i_{x,t}}$, $x \in [1, n]$. La population initiale est générée de manière aléatoire, aucun sous-espace de recherche n'est favorisé. La fonction fitness est notée $\phi : I \to E$. Elle consiste à associer à un individu un score, par exemple une valeur réelle positive (dans notre cas, $E \subset \mathbb{R}^+$). Le schéma de fonctionnement général d'un algorithme évolutif [Bäc96] est le suivant :

$t = 0$
initialise(P_t) $= \{\vec{i_{1,t}}, ..., \vec{i_{n,t}}\}$
Tant que (Condition non satisfaite)
  croisement : $P'_t = croise(P_t)$
  mutation : $P''_t = mute(P'_t)$
  évaluation : $S = map(\phi, P_t \cup P'_t \cup P''_t)$
  sélection : $P_{t+1} = selectionne(P_t \cup P'_t \cup P''_t, S)$
  $t = t + 1$

La condition à satisfaire peut par exemple être un nombre de générations d'individus à calculer en fonction du temps d'exécution de l'algorithme si l'on souhaite que l'exploration s'arrête dans des délais plus ou moins brefs. La fonction *croise* construit une population P'_t en procédant à des croisements d'individus issus de la population P_t. A chaque croisement, deux individus parents sont pris aléatoirement de P_t pour former un enfant. La fonction *mute* construit une population P''_t en procédant à des mutations d'individus issus de la population P'_t. Tous les individus générés ($P''_t \cup P'_t \cup P_t$) sont ensuite évalués, puis une sélection est faite sur cet ensemble d'individus pour obtenir une nouvelle population P_{t+1} formée de n individus (les autres sont éliminés).

4.2 Contexte d'utilisation de l'algorithme évolutif

Dans ce travail, notre objectif est d'utiliser un algorithme génétique tel que celui présenté afin de sélectionner des bonnes configurations de ressources. Un individu représente ici une configuration.

4.2.1 Présentation

Au moment de procéder à la phase de sélection de ressources, l'ordonnanceur possède diverses informations, tels l'ensemble des stations de calcul disponibles, les bandes passantes effectives entre ces stations ou encore des informations spécifiées par l'utilisateur. Le nombre de configurations susceptibles d'intéresser l'utilisateur devient rapidement élevé à partir d'un certain nombre de processeurs. Afin de bénéficier du choix de l'ordonnanceur, celui-ci doit prendre sa décision dans des délais relativement brefs suivant la taille de l'application. En effet, il a été estimé [WZ97] que le choix de sélection de ressources ne devait pas excéder 10% du temps d'exécution de l'application, pour que l'ordonnancement soit suffisamment profitable. Dans cette optique, il n'est pas envisageable de procéder à une recherche exhaustive de la configuration optimale. La méthode d'optimisation de l'algorithme évolutif permet donc de converger vers un ensemble de configurations de machines satisfaisant du mieux possible l'utilisateur et de les lui proposer. L'utilisateur choisit alors, parmi la liste des configurations qui lui ont été suggérées, celle qui lui convient le mieux. Le fichier décrivant le job à soumettre sur les ressources correspondantes est alors construit. La figure 4.2 illustre ces étapes.

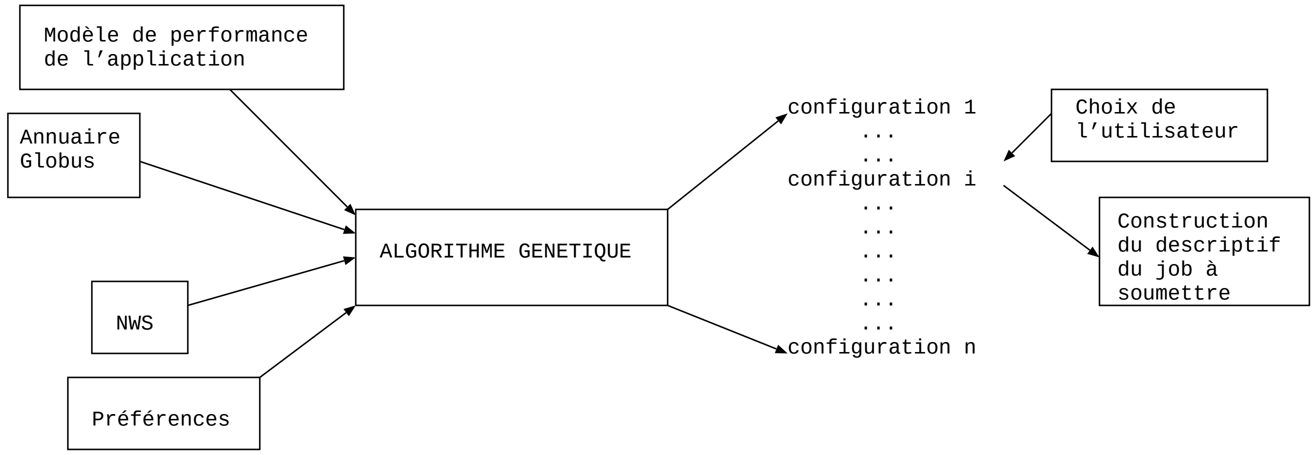

FIG. 4.2 – Contexte d'utilisation de l'algorithme génétique

4.2.2 Modèle de performance de l'application

L'utilisateur indique le modèle de l'application qu'il soumet. Ce modèle est primordial, c'est lui qui détermine la manière dont les calculs de l'application sont effectués ainsi que la manière dont communiquent les processus durant l'exécution de l'application. Nous verrons (section 4.4) comment la fonction d'évaluation d'une configuration de ressources est construite à partir de deux différents modèles correspondant à deux types d'applications mises en jeu dans notre projet. Par défaut, une stratégie d'analyse du code pourra être mise en oeuvre pour déterminer quel modèle est le plus approprié pour l'application.

4.2.3 Annuaire Globus

L'utilisation de l'annuaire Globus permet de connaître les ressources disponibles de la grille. Cet ensemble de ressources est celui pris en compte par l'algorithme évolutif. L'annuaire donne des informations également importantes dans la construction de la fonction fitness comme la quantité de RAM de la station, la vitesse et le type de processeur. Pour des raisons encore une fois matérielles, des benchmarks ont été effectués pour observer le temps de calcul mis par chaque ressource pour les applications considérées. A la suite de ces études, des coefficients de calcul relatifs aux processeurs ont été déterminés.

4.2.4 NWS

L'utilisation du logiciel NWS permet de connaître dynamiquement la bande passante disponible entre chaque station, la latence observée, ainsi que la charge CPU des stations. Ces valeurs sont importantes car la fonction fitness est basée sur elles. Toutefois, l'utilisation de NWS sur chaque station de la grille est contraignant, car le logiciel doit être installé et des démons lancés sur chacune des stations. Des valeurs moyennes, donc constantes, ont été établies par benchmarks. Les applications étudiées ont fait l'oeuvre de tests entre les diverses ressources disponibles dans notre grille pour évaluer les performances des ressources. Ces valeurs sont prises en compte par défaut lorsque NWS est indisponible sur une station de la grille.

4.2.5 Préférences de l'utilisateur

L'utilisateur peut spécifier un ensemble de préférences comme le nombre de processeurs désiré ainsi que des contraintes sur les ressources (vitesse de processeur). Nous espérons bientôt pouvoir intégrer une requête concernant la taille de la RAM escomptée par l'utilisateur. Ces préférences sont prises en compte dans les processus de sélection, croisement et mutation car les individus ne satisfaisant pas les critères de l'utilisateur ne doivent pas survivre. L'algorithme ne doit explorer que le sous-espace de recherche satisfaisant aux conditions.

4.3 Les individus et leur évaluation

4.3.1 Génotype des individus

Gènes Les individus représentent des solutions potentielles à notre problème d'ordonnancement. Le code génétique doit donc refléter quelles ressources ont été sélectionnées pour l'exécution de l'application. Il est composé d'autant de gènes que de stations. Autrement dit, c'est un vecteur de gènes de taille le nombre de machines. Les valeurs des gènes représentent le nombre de processeurs sélectionnés sur la station correspondante. Suivant la position d'un gène dans le code génétique, les différentes valeurs possibles pour ce gène ne sont pas identiques puisque les stations mettent à disponibilité un nombre différent de processeurs. Un individu se doit donc de respecter les conditions de validité pour pouvoir survivre.

Fonction de codage La fonction de codage est une fonction qui, à partir d'un individu, renvoie son code génétique. Elle prend en entrée la liste des stations disponibles associées au nombre de leurs processeurs qui ont été sélectionnés. Ainsi, le code génétique est un vecteur dont le i^{eme} élément indique le nombre de processeurs sélectionnés sur la i^{eme} machine de la liste. On le représente de la manière suivante :

$$c_1. \dots .c_i. \dots .c_n.$$

où $n \in \mathbb{N}^*$ est le nombre de machines disponibles et $c_i \in \mathbb{N}$ est le nombre de processeurs sélectionnés sur la i^{eme} machine de la liste.

Dans l'ensemble des exemples que nous allons présenter, nous nous plaçons dans la situation décrite dans le tableau qui suit.

Ressource disponible	Nombre de processeurs disponibles
seven.u-strasbg.fr	4
pellinore.u-strasbg.fr	2
opera.u-strasbg.fr	1
minerve.cines.fr	18
sekhmet.u-strasbg.fr	1

Exemple L'individu correspondant à une distribution de l'application sur deux processeurs de seven, un processeur de opera et six processeurs de minerve possède le code génétique suivant :

$$2.0.1.6.0.$$

Contraintes de validité

1. **Liées au nombre de processeurs disponibles**
   Soit I un individu de code génétique $C = c_1. \dots .c_n.$, I est dit p-valide si et seulement si :
   $$\forall i \in [1, n] \quad 0 \leq c_i \leq maxproc(i)$$
   où $maxproc$ est la fonction qui renvoie le nombre de processeurs disponibles d'une station.

   Exemple L'individu ayant pour code génétique $C = 2.0.1.6.0.$ est p-valide. En revanche, l'individu ayant pour code génétique $C' = 0.1.2.0.1$ n'est pas p-valide car $c'_3 = 2$ et $maxproc(3) = 1$. En effet, nous ne disposons pas de deux processeurs sur opera.

2. **Liées à l'utilisateur**
   Soit I un individu de code génétique $C = c_1. \dots .c_n.$. Soit k le nombre de processeurs désiré par l'utilisateur. I est dit u-valide si :
   i)
   $$\sum_{i=1}^{n} c_i = k$$
   ou

   ii) l'utilisateur n'a pas spécifié de contraintes sur le nombre de processeurs

   Exemple L'utilisateur souhaite huit processeurs. L'individu ayant pour code génétique $C = 2.1.1.4.0.$ est u-valide. En revanche, l'individu ayant pour code génétique $C' = 0.1.1.1.1.$ n'est pas u-valide car $\sum_{i=1}^{n} c'_i \neq 8$.

Validité Un individu I est dit valide si et seulement si il est p-valide et u-valide.

4.3.2 La fonction fitness

La fonction fitness a pour objectif d'évaluer les individus afin de les classer et de promouvoir les meilleurs d'entre eux. Dans notre cas, le fitness est une prédiction du temps d'exécution de l'application suivant la configuration de ressources choisies. Cette estimation est exprimée en seconde, de ce fait plus la valeur du fitness d'un individu est petite, meilleur est cet individu.

4.4 Paramètres de la fonction fitness

Le calcul du fitness dépend de plusieurs paramètres dont les plus importants sont 1) la puissance des processeurs, 2) la vitesse du réseau et 3) le comportement de l'application, c'est-à-dire la structure des calculs et des communications.

4.4.1 Puissance des processeurs

La fonction fitness utilise une note de puissance attribuée aux processeurs de la grille par des benchmarks effectués sur nos applications. La note arbitraire de 1 à été donnée à un processeur Intel/PIII 800 MHz. Le tableau ci-dessous présente les notes relatives des autres processeurs. Cette note intervient lors de l'estimation du temps de calcul de l'application.

Type de la ressource	Coefficient de puissance
Athlon 1800XP	1.61
PIII 933	1.14
R14K 500	1.05
PIII 800	1.00
R12K 350	0.65
R10K 195	0.46
USparc 450	0.38

4.4.2 Vitesse du réseau

La fonction fitness exploite les valeurs de bande passante et de latence entre les processeurs pour estimer le temps de transfert de données durant l'exécution de l'application. Pour cela, elle se base sur un tableau de valeurs rempli au moyen de NWS ou de benchmarks. Voici un exemple de tableau pour notre grille de tests (tableau 4.1).

Bande passante (Mo/s)						Latence (s)				
Illkirch 1	*Illkirch 2*	*Strasbourg*	*Clermont*	*Montpellier*		*Montpellier*	*Clermont*	*Strasbourg*	*Illkirch 2*	*Illkirch 1*
7.5	4	3.7	0.6	0.5	*Illkirch 1*	0.67	0.18	0.02	0.02	0.01
	30	2.5	0.6	0.2	*Illkirch 2*	0.74	0.29	0.02	0.002	
		22	0.6	0.2	*Strasbourg*	0.74	0.29	0.02		
				0.26	*Clermont*	0.57				
				35	*Montpellier*	0.002				

TAB. 4.1 – Bandes passantes et latences sur la grille

4.4.3 Modèles de performance

La méthode de calcul du fitness varie selon le type d'applications car celles-ci sont composées de phases de calcul et de communication se déroulant d'une manière qui leur est spécifique. Pour le calcul d'estimation d'une exécution pour un type d'application donné, on parle alors de modèle de performance de ce type d'application. Elaborer un modèle de performance pour un type d'application revient à approximer de manière la plus précise possible le comportement de l'application en décrivant les phases de calcul et de communication. Outre une approximation précise du temps d'exécution de l'application, cette méthode présente un grand avantage : l'extensibilité. Notre algorithme évolutif peut s'appliquer à plusieurs types d'applications. Différents modèles de performance, servant à l'évaluation des individus, sont encapsulés sur l'algorithme. Ainsi, pour une application donnée, l'algorithme évolutif prévoit de la bonne manière les temps d'exécution et les configurations résultantes sont fructueuses.

Nous présentons ici deux modèles d'applications utilisées dans le projet TAG : l'application Ray2Mesh et l'application Vador.

L'application Ray2Mesh L'application Ray2Mesh [GGM02] est un mailleur 3D parallèle pour la construction d'un modèle de terre par tomographie sismique dans le but de modéliser et représenter la structure interne du globe. La qualité de la solution tomographique dépend principalement de la géométrie des rais sismiques qui relient les foyers des séismes aux stations d'enregistrement. L'objectif étant de construire un maillage constitué de cellules uniformes afin d'obtenir une résolution fine, les temps de calcul ainsi que la mémoire nécessaires sont considérables, ce qui fait de l'application ray2mesh un bon client pour la grille.

L'application Ray2Mesh utilise la bibliothèque MPI. Initialement, les données se trouvent sur un processus maître. Celui-ci découpe l'application en un nombre de tâches spécifié par l'utilisateur, et envoie à chaque tâche mis en jeu les données qui leur sont assignées. Cette phase est appelée phase de communication. La phase de calcul, quant à elle, consiste en l'exécution en parallèle des tâches placées sur les processeurs. Nous supposons que le processus maître ne peut communiquer qu'avec un processus à la fois. Une fois qu'un processus reçoit les données qui lui sont assignées, il démarre sa tâche de calcul. A la suite de cette phase de calcul, les résultats des exécutions parallèles sont récupérées par le processus maître. Cette dernière phase de communication est négligeable. Pour estimer le temps d'exécution d'une telle application, nous modélisons d'une part la phase de communication et d'autre part la phase de calcul avant d'interfacer les deux suivant le fonctionnement global de l'application.

- *Phase de communication*

  Nous rappelons que l'équilibrage de charge a lieu après la phase d'ordonnancement. Par conséquent, nous considérons que la même quantité de données est assignée à chaque processeur. Nous pouvons estimer les temps de communication de la manière suivante :

Soient n le nombre de ressources disponibles, $s(i)$ le nombre de processeurs sélectionnés sur la ressource i et v le nombre de processeurs voulu
Pour $i = 1$ à n
$T_{com_i} = \sum_{k=1}^{i-1} T_{com_k} + (bandwidth_{master,i} \times \frac{datasize}{v} + latency_{master,i}) \times s(i)$

où $bandwidth_{master,i}$ est la bande passante disponible entre le processeur maître et le i^{eme} processeur, $datasize$ est la quantité de données au total de l'application et $latency_{master,i}$ est le temps de latence observé entre le processeur maître et le i^{eme} processeur.

Nous considérons que le processeur maître ne peut communiquer qu'avec un processeur à la fois. De ce fait, le temps mis pour envoyer les données à un processeur dépend de deux points :
- du temps mis pour le transfert des données aux processeurs précédents,
- du temps de transfert effectif des données à ce processeur, qui dépend de la taille des données transférées, de la bande passante et de la latence entre le processeur maître et lui.

- *Phase de calcul*

  Le temps de calcul sur un processeur est estimé comme suit :

$$\forall i \in [1, n], T_{cal_i} = \frac{K_{ray2mesh} \times \frac{datasize}{v}}{procval_i}$$

  où $K_{ray2mesh}$ est le coefficient de traitement de rais, en seconde par octet, déterminé empiriquement à partir de benchmarks, et $procval_i$ est la note du processeur i. Cette note a été déterminée par des benchmarks. Le processeur unitaire, c'est-à-dire ayant pour coefficient 1, est un pentium III 800 MHz. Un athlon 1800 XP, par exemple, a un coefficient de 1,61 et un Ultrasparc 450 MHz a un coefficient de 0,38.

- *Estimation totale*

  Soit S l'ensemble des processeurs sélectionnés. L'estimation du temps d'exécution totale de l'application est :

$$T_{exec} = \max_{i \in S}\{T_{com_i} + T_{cal_i}\}$$

L'application Vador L'application Vador [VF02] concerne la résolution de problèmes en physique des plasmas et des faisceaux de particules. Un problème majeur est celui de la fusion thermonucléaire contrôlée qui vise à obtenir une énergie propre et renouvelable. Ces problèmes sont modélisés essentiellement par le couplage des équations de Vlasov [FSB01] et de Maxwell [Cos90].

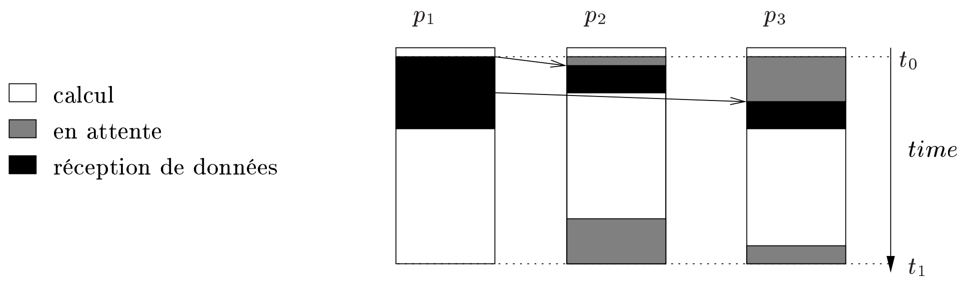

FIG. 4.3 – Exemple d'exécution

La résolution numérique du système de Vlasov-Maxwell nécessite une puissance de calcul importante et beaucoup de mémoire disponible d'où l'interêt de porter l'application vador sur la grille.

L'exécution de Vador opère sur une matrice stockant les coordonnées d'un point ainsi que celles de son vecteur vitesse à un instant t. Les dimensions de la matrice sont paramétrées par l'utilisateur. L'exécution de vador est composée d'un nombre n d'itérations spécifié par l'utilisateur avant le lancement de l'application. La matrice est découpée en v parties et chaque processeur procède à des calculs sur sa partie de matrice. Une itération est composée de quatre phases (Figure 4.4).

Il faut également ajouter une phase d'initialisation où chaque processeur reçoit du processeur maître les données qui lui sont assignées.

Phase 0 Le processeur maître distribue les données aux processeurs sélectionnés. Il ne peut procéder à plusieurs envois simultanément. Nous estimons le temps d'initialisation par :

$$T_{initcom} = \sum_{i \neq master} (bandwidth_{master,i} \times \frac{datasize}{v} + latency_{master,i})$$

Phase 1 et 3 Ce sont deux phases de calcul. Le temps de calcul dans une itération, sur un processeur i est noté T_{cal_i}. On considère qu'une étape de calcul sur un processeur donne toujours le même temps, vu qu'il doit traiter à chaque fois la même quantité de données :

$$T_{cal_i} = \frac{K_{vador} \times \frac{datasize}{v}}{procval_i}$$

où K_{vador} est le temps de traitement d'un octet de données dans cette application. Il a été déterminé empiriquement d'après des benchmarks. *datasize* est la taille totale en octets des données mises en jeu (c'est-à-dire la taille de la matrice).

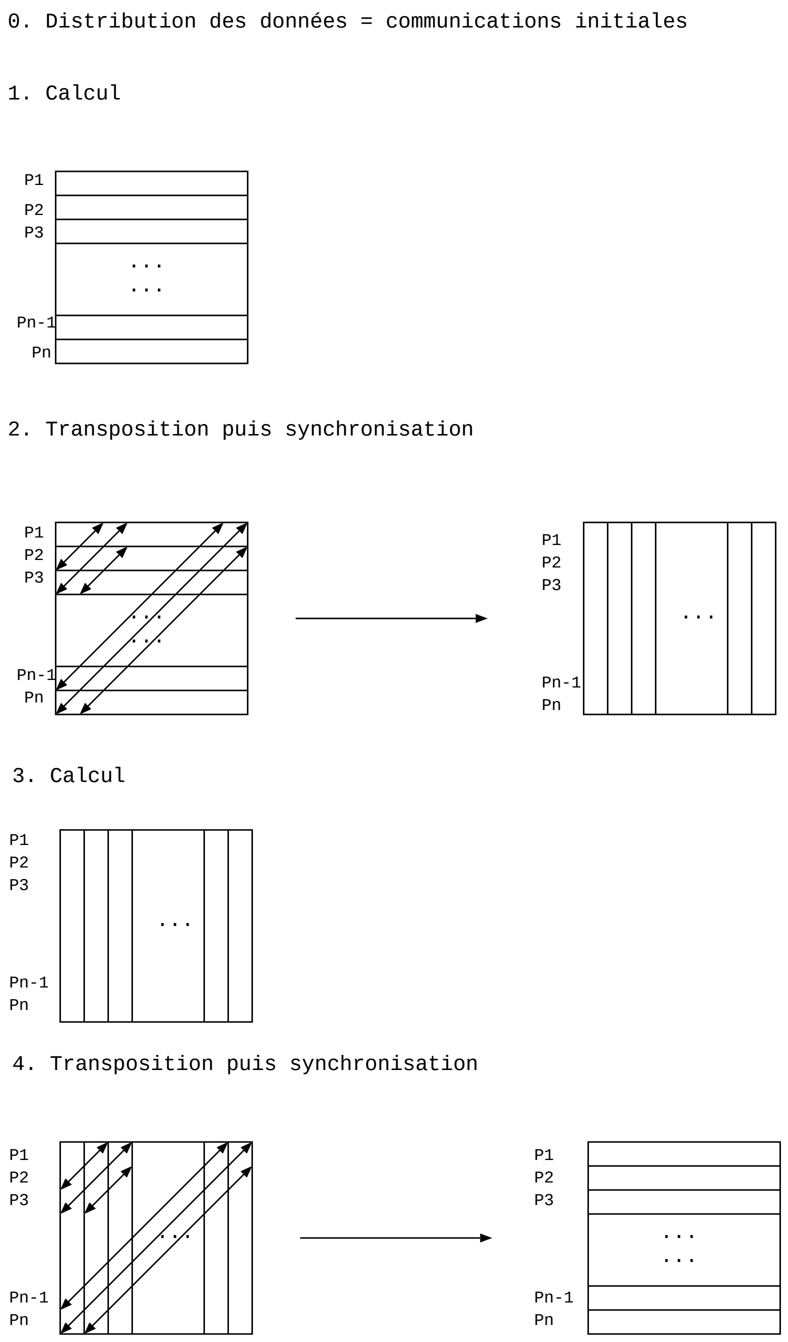

FIG. 4.4 – Une itération de l'application Vador

Phase 2 et 4 Ce sont deux phases de communication dans lesquelles chaque processeur va envoyer aux autres processeurs les données qui leur sont destinées pour réaliser une transposition de la matrice. Les temps de communication sont estimés de la manière suivante :

$$T_{com_i} = \sum_{j \neq i} (bandwidth_{i,j} \times \frac{datasize}{v^2} + latency_{i,j})$$

On considère qu'un processeur ne peut communiquer qu'avec un autre processeur à la fois donc c'est bien une somme de temps d'envoi de données que l'on doit estimer. Les éléments transmis lors de la transposition sont des facteurs $\frac{1}{v^2}$.

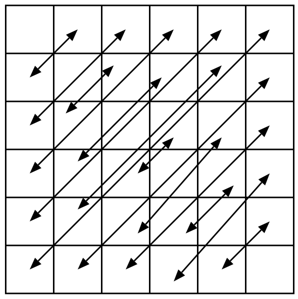

FIG. 4.5 – Communications mises en jeu par vador

Estimation totale Une synchronisation sépare les deux premières phases des deux dernières phases. Soit S l'ensemble des processeurs sélectionnés. Nous estimons le temps d'exécution de l'application vador par :

$$T_{exec} = 2 \times nbiter \times \max_{i \in S}\{T_{cal_i} + T_{com_i}\} + T_{initcom}$$

4.5 Les opérations génétiques

4.5.1 La stratégie de croisement

Nous avons vu dans le premier volet de ce chapitre qu'un algorithme génétique est un modèle d'aprentissage stochastique simulant le processus de l'évolution naturelle. L'opération de croisement représente la reproduction. Nos individus ne sont pas séparés en différents sexes, la reproduction est donc assexuée. Elle consiste à échanger certains gènes entre deux individus pour former des enfants héritant des gènes de leurs parents. Le croisement contribue à explorer au mieux l'espace de recherche en combinant deux sous-espaces jugés bons. Notre opération de croisement s'effectue en un point à la moitié du code génétique des parents. Ainsi, les $\lfloor n/2 \rfloor$ premiers gènes de l'individu enfant sont hérités de son premier parent et les $n - \lfloor n/2 \rfloor$ derniers gènes sont hérités de son second parent (figure 4.6).

L'individu résultant du croisement possède bien les traces d'hérédité (partie gauche du code de son premier parent et partie droite du code de son second parent). Il est clair que la condition de p-validité est satisfaite puisque les deux parents la vérifie. En revanche, il se peut très bien que la condition d'u-validité soit perdue rendant l'individu enfant invalide (comme dans l'exemple de la figure 4.7).

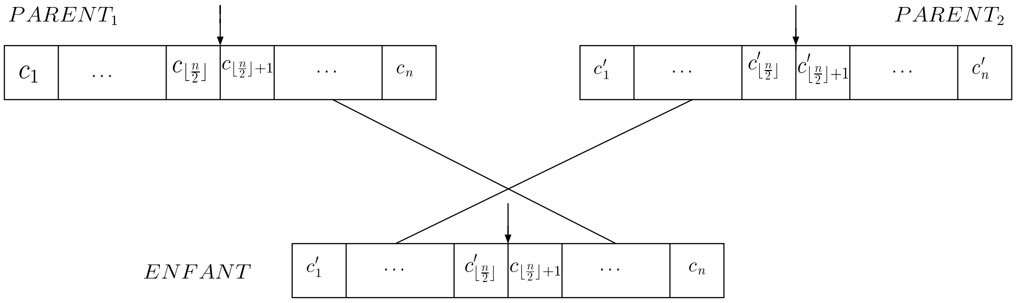

FIG. 4.6 – Croisement de deux individus

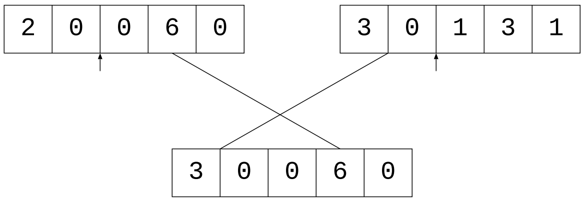

FIG. 4.7 – Perte de l'u-validité

Nous ne pouvons pas simplement considérer l'individu résultant de ce croisement comme non valide et le supprimer. En effet, la probabilité de perte d'u-validité est élevée et très peu d'individus pourraient alors se reproduire. La convergence vers un individu serait trop rapide et l'exploration de l'espace des solutions ne serait pas suffisamment poussée. Il s'agit donc de trouver une méthode qui adapte l'individu fils à l'environnement en ajustant son code génétique, afin qu'il soit u-valide tout en possédant un maximum de gènes possibles hérités des parents. Evidemment, cet ajustement de la valeur de gènes doit respecter la contraite de p-validité.

L'idée de l'ajustement est de prendre un gène au hasard et de retrancher 1 si cela est possible (valeur du gène strictement positive) ou d'ajouter 1 si cela est possible (valeur du gène strictement inférieur au nombre de processeurs disponibles sur la ressource), ceci suivant que l'on veuille augmenter le nombre de processeurs de l'individu ou le diminuer. On réitère l'opération tant que le nombre de processeurs de l'individu n'est pas égal à la somme désirée.

Algorithme de croisement avec adaptation à l'environnement Nous disposons d'une fonction random(n) qui renvoie un nombre entier k ($k \in [1, n]$). L'ensemble de la population est ici implémenté sous la forme d'un tableau P.

Soient n le nombre de ressources,
v le nombre de processeurs voulu,
P le tableau représentant l'ensemble de la population.

$C \leftarrow P[random(popusize)]$
$C' \leftarrow P[random(popusize)]$

Pour i $\leftarrow$ 1 à $\lfloor n/2 \rfloor$
    $C''_i \leftarrow C_i$

Pour i $\leftarrow \lfloor n/2 \rfloor + 1$ à n
    $C''_i \leftarrow C'_i$

Tant que $\sum_{i=1}^{n} C''_i \neq v$
    $h \leftarrow random(n)$
    Si $\sum_{i=1}^{n} C''_i > v$
        Si $C''_h > 0$
            alors $C''_h \leftarrow C''_h - 1$
    Si $\sum_{i=1}^{n} C''_i < v$
        Si $C''_h < maxproc(h)$
            alors $C''_h \leftarrow C''_h + 1$

4.5.2 La stratégie de mutation

L'opération de mutation consiste à modifier la valeur de certains gènes d'un individu. Elle permet l'exploration de nouveaux sous-espaces de solutions. Notre opération de mutation consiste à prendre un individu de la population et de modifier un certain nombre de ses gènes. Le nombre de gènes transformés est compris entre 1 et le nombre total de gènes du code génétique. Afin de préserver la p-validité de l'individu, la nouvelle valeur du gène modifié est choisie aléatoirement entre 0 et le nombre maximum de processeurs que peut représenter le gène. Toutefois, en modifiant la valeur d'un gène, l'u-validité est automatiquement perdue (sauf si la nouvelle valeur du gène obtenue aléatoirement est la même que la précédente). Il s'agit donc, comme avec l'opération de croisement, d'adapter l'individu à l'environnement et d'ajuster la valeur de gènes afin de le rendre u-valide et donc valide. La stratégie d'ajustement est identique à celle présentée dans le cas du croisement.

Algorithme de mutation avec adaptation à l'environnement

Soient n le nombre de ressources, v le nombre de processeurs désirés.

Pour i $\leftarrow$ 1 à n
 $C'_i \leftarrow C_i$

$h = random(popusize)$
$h1 = random(n)$

Pour i $\leftarrow$ 1 à $h1$
 $h2 \leftarrow random(n)$
 $h3 \leftarrow random(maxproc(h2) + 1) - 1$
 $C'_{h2} \leftarrow h3$

Pour i $\leftarrow$ 1 à $\lfloor n/2 \rfloor$
 $C''_i \leftarrow C_i$

Pour i $\leftarrow \lfloor n/2 \rfloor + 1$ à n
 $C''_i \leftarrow C'_i$

Tant que $\sum_{i=1}^{n} C''_i \neq v$
 $h \leftarrow random(n)$
 Si $\sum_{i=1}^{n} C''_i > v$
  Si $C''_h > 0$
   alors $C''_h \leftarrow C''_h - 1$
 Si $\sum_{i=1}^{n} C''_i < v$
  Si $C''_h < maxproc(h)$
   alors $C''_h \leftarrow C''_h + 1$

4.5.3 La stratégie de sélection

L'opération de sélection permet de retenir des individus pour l'opération de croisement en privilégiant les individus les mieux côtés. Elle assigne la force de reproduction aux individus selon leur qualité et favorise ainsi l'exploration des bons sous-espaces de solutions. Il existe différentes méthodes de sélection. Nous distinguerons, en particulier, les méthodes élitistes, qui garantissent la survie des meilleurs individus, des méthodes non élitistes, qui attribuent une probabilité non nulle d'élimination aux meilleurs individus. Nous en présentons ici rapidement quelques une, avant de les évaluer pour déterminer les plus appropriées à notre problème.

La sélection par le fitness Il s'agit de classer les individus selon leur fitness, du meilleur au moins bon. Les N premiers individus sont sélectionnés pour être croisés. Cette sélection est donc élitiste. Dans le pire des cas, la solution n'est pas améliorée. En outre, cette méthode converge rapidement vers un résultat. Néanmoins, cette convergence peut être trop rapide et mener à un résultat non optimal. Il peut donc être intéressant d'utiliser d'autres types de sélections.

La roue de la roulette La roue de la roulette est une méthode de sélection probabiliste dans laquelle on attribue, selon la qualité de l'individu i, une probabilité plus ou moins forte d'être sélectionné. Dans une population de N individus, cette probabilité est :

$$P_i = \frac{Fitness_i}{\sum_{j=1}^{N} Fitness_j}$$

Graphiquement, on représente l'ensemble des individus par une roue découpée en autant de portions que d'individus. Plus la probabilité qu'un individu soit choisi est grande, plus sa portion sur la roue est grande. On "tourne" ensuite la roue, pour simuler le tirage aléatoire d'une valeur sur la roue. L'individu qui possède dans sa portion cette valeur est sélectionné. Pour sélectionner M individus, il s'agit de tourner M fois la roue.

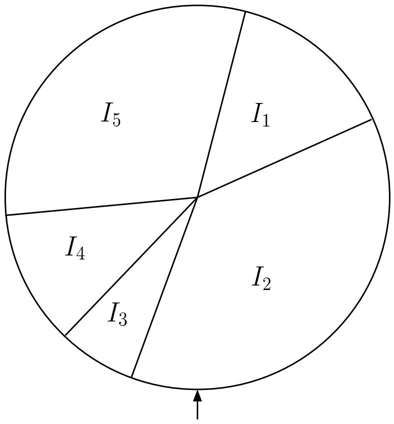

FIG. 4.8 – La roue de la roulette

L'inconvénient de cette méthode est qu'un individu dont le score donné par la fonction d'évaluation est nettement meilleur par rapport aux autres, risque de dominer la génération de façon trop importante. De plus, le meilleur individu a une probabilité non nulle d'être éliminé, c'est une méthode non élitiste. La sélection, dans ce cas, est directement liée à la fonction d'évaluation : si les scores des individus sont proches les uns des autres, les meilleurs individus n'ont pas beaucoup plus de chances d'être sélectionnés que d'autres moins bons.

Algorithme de la roue de la roulette Cet algorithme construit les intervalles de la roue et procède à la sélection aléatoire des individus.

$Somme_0 = 0$
Pour i = 1 à N
$\quad Somme_i = Somme_{i-1} + \frac{Fitness_i}{\sum_{j=1}^{N} Fitness_j}$
Pour i = 1 à M
$\quad$ r = random([0,1])
$\quad$ sélectionner P_k tel que $Somme_k \leq r < Somme_{k+1}$

La sélection sur le rang Ce type de sélection ressemble à la précédente. La sélection sur le rang est une roulette dont les probabilités de retenir les individus sont calculées non plus à partir de leur fitness, mais à partir de leur position au classement. Il s'agit de définir une formule de calcul de la probabilité de sélection P_i d'un individu i selon son classement $Rang_i$. Nous choisissons de déterminer cette probabilité suivant cet algorithme :

Pour i = 1 à $\lfloor \frac{N}{2} \rfloor$
$\quad P_i = \frac{2 \times (N - Rang_i)}{N^2}$
$P_{\lceil \frac{N}{2} \rceil} = \frac{1}{N}$
Pour i = 1 à $\lceil \frac{N}{2} \rceil + 1$ à N
$\quad P_i = \frac{2 \times (N - Rang_i + 1)}{N^2}$

A titre d'exemple, pour une population de 10 individus classés, on calcule leurs probabilité de sélection respectives :

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
$\frac{9}{50}$	$\frac{4}{25}$	$\frac{7}{50}$	$\frac{3}{25}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{2}{25}$	$\frac{3}{50}$	$\frac{1}{25}$	$\frac{1}{50}$

Graphiquement, ces probabilités peuvent être représentées par un schéma comme celui de la figure 4.9.

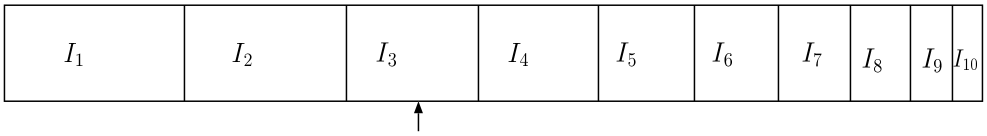

FIG. 4.9 – Sélection par le rang

L'avantage de cette méthode est de ne pas favoriser excessivement un individu. En revanche, elle n'est pas élitiste et le meilleur individu peut être perdu. Contrairement à la sélection par la roulette, la sélection par le rang n'est pas directement liée à la valeur des fitness mais au classement. Ainsi, que les scores des fitness soient prochent ou non n'influent en rien sur la probabilité de sélection, qui se base sur la position au classement.

La sélection par tournoi Dans cette méthode de sélection, il s'agit de prendre au hasard un nombre n d'individus parmi l'ensemble formant la population considérée. Ces n individus sont ensuite classés et le meilleur d'entre eux est sélectionné. Le procédé est réitéré N fois pour obtenir la nouvelle population de futurs parents. Cette méthode est également non élitiste.

Notre souhait est d'obtenir un algorithme d'ordonnancement convergeant de la manière la plus rapide possible afin de fournir à l'utilisateur des solutions dans des délais brefs. La sélection de fitness semblerait à priori être la méthode de sélection la plus adéquate à notre problème. Toutefois, il est capital de tester la méthode pour observer si la convergence est précoce ou non. Une convergence précoce pourrait entrainer des résultats non satisfaisants. En outre, le fitness de nos individus sont des temps en seconde souvent très proches. Aussi, on peut penser que la sélection par la roulette aurait des difficultés à converger, voire ne convergerait pas, dans notre environnement. Notre dernier chapitre présente les expériences que nous avons faites et nous permet de déduire quelles sont les méthodes de sélection adaptées à notre algorithme d'ordonnancement.

Chapitre 5

Expérimentations

5.1 Notre grille de tests

Les tests ont été effectués sur une grille composée de 51 processeurs répartis géographiquement à Illkirch, Strasbourg-Esplanade, Clermont-Ferrand et Montpellier. Avant de débuter l'algorithme d'ordonnancement, un programme s'informant sur l'état de la grille est exécuté. Ce programme utilise le logiciel de prédiction de performance NWS ainsi que les valeurs par défaut pour créer un fichier de configuration de la grille qui est passé en entrée de l'algorithme d'ordonnancement.

Ressource	Lieu	proc. disponibles	Note de puissance
seven.u-strasbg.fr(R10K)	Illkirch	4	0.65
seven.u-strasbg.fr(R12K)	Illkirch	4	0.46
sekhmet.u-strasbg.fr	Illkirch	1	1.61
pellinore.u-strasbg.fr	Illkirch	2	1.00
dinadan.u-strasbg.fr	Illkirch	1	1.14
opera.u-strasbg.fr	Illkirch	1	0.40
lattice.u-strasbg.fr	Illkirch	1	1.61
caseb.u-strasbg.fr	Illkirch	1	1.61
irmasrv2.u-strasbg.fr	Strasbourg-Esplanade	4	0.38
irmasrv3.u-strasbg.fr	Strasbourg-Esplanade	12	0.63
marathon.u-strasbg.fr	Strasbourg-Esplanade	1	0.91
biogrid01.in2p3.fr	Clermont-Ferrand	1	0.96
minerve.cines.fr	Montpellier	18	1.05

5.2 Ordonnancement d'une application

Au moment de lancer l'algorithme d'ordonnancement l'utilisateur entre en paramètres :
- le nombre de processeurs désiré
- la taille de l'application (en octets)
- la ressource maître (détenant initialement toutes les données de l'application)

- le modèle de l'application (Ray2mesh ou Vador)
- le nombre de configurations résultantes de l'algorithme (taille du classement des configurations retourné par le programme)
- des options spécifiques au modèle de l'application
- des options spécifiques à l'algorithme génétique (nombre de générations à calculer, taille de la population ...)

```
[stehly@opera src]$ ./genetic4 -h
Usage: genetic4 [OPTION...]
  -h, --help                        help
  -m, --master=machine.domain.ext   host having the master process
  -d, --datasize=int                datasize
  -n, --np=int                      desired number of processors
  -t, --top=n                       lists the n most efficient configurations
  -i, --output=html|text            output mode
  -a, --appmodel=ray2mesh|vador     application model
  -v, --version                     Print version number
```

Exemple 1 On se propose d'étudier un ordonnancement de l'application Ray2mesh. Les paramètres de l'ordonnancement sont les suivants :
- l'application s'exécute sur 8 processeurs
- la taille de l'application est 20196438 octets : l'application traite 325749 rais, la taille d'un rai est de 62 octets
- l'application est lancée sur sekhmet (sekhmet possède les données en entrée de l'algorithme)
- le classement final est composé des 10 meilleurs configurations

Résultat de l'algorithme évolutif

```
------- TOP 10 -------  :

(1) sekhmet:1  pellinore:2  dinadan:1  lattice:1  caseb:1  minerve.cines.fr:2
        =>fitness: 631.745 =  calc:631.139 + comm:0.606 (slowest:pellinore)

(2) sekhmet:1  pellinore:2  lattice:1  caseb:1  minerve.cines.fr:3
        =>fitness: 636.239 =  calc:601.084 + comm:35.154 (slowest:minerve.cines.fr)

(3) sekhmet:1  pellinore:2  dinadan:1  caseb:1  minerve.cines.fr:3
        =>fitness: 636.241 =  calc:601.084 + comm:35.157 (slowest:minerve.cines.fr)

(4) sekhmet:1  pellinore:2  dinadan:1  lattice:1  minerve.cines.fr:3
        =>fitness: 636.241 =  calc:601.084 + comm:35.157 (slowest:minerve.cines.fr)

(5) sekhmet:1  pellinore:1  dinadan:1  lattice:1  caseb:1  minerve.cines.fr:3
        =>fitness: 636.241 =  calc:601.084 + comm:35.157 (slowest:minerve.cines.fr)
```

```
(6) pellinore:2  dinadan:1  lattice:1  caseb:1  minerve.cines.fr:3
        =>fitness: 636.545 =  calc:601.084 + comm:35.460 (slowest:minerve.cines.fr)

(7) sekhmet:1  pellinore:2  caseb:1  minerve.cines.fr:4
        =>fitness: 647.250 =  calc:601.084 + comm:46.165 (slowest:minerve.cines.fr)

(8) sekhmet:1  pellinore:2  lattice:1  minerve.cines.fr:4
        =>fitness: 647.250 =  calc:601.084 + comm:46.165 (slowest:minerve.cines.fr)

(9) sekhmet:1  pellinore:1  lattice:1  caseb:1  minerve.cines.fr:4
        =>fitness: 647.250 =  calc:601.084 + comm:46.165 (slowest:minerve.cines.fr)

(10) sekhmet:1  pellinore:2  dinadan:1  minerve.cines.fr:4
        =>fitness: 647.253 =  calc:601.084 + comm:46.168 (slowest:minerve.cines.fr)
```

Commentaires Les communications occasionnées par l'application se produisent lors de la distribution initiale des données du processus maître. Lors de l'exécution du programme parallèle, les processus ne communiquent pas entre eux. De ce fait, les communications n'influent pas considérablement sur le temps d'exécution de l'application. Nous constatons que l'ordonnanceur ne sélectionne pas forcément les machines les plus proches possibles géographiquement. La configuration en tête du classement est composée de ressources d'Illkirch et de Montpellier. La ressource la plus lente est pellinore. Même si le débit entre sekhmet (processeur maître) et pellinore est supérieur à celui entre sekhmet et minerve, le temps de calcul passé sur pellinore suffit à rendre son temps d'exécution supérieur à minerve. L'aspect puissance de processeurs est donc très importante dans ce modèle. A partir de la deuxième configuration du classement, la ressource la plus lente est minerve. Ceci est expliqué par le fait qu'ajouter un processeur de minerve augmente les temps de communication qui cette fois-ci rendent l'exécution sur le troisième processeur de minerve supérieure à celui sur pellinore.

Exemple 2 On se propose d'étudier un ordonnancement de l'application Vador. Les paramètres de l'ordonnancement sont les suivants :

- l'application s'exécute sur 4 processeurs
- la taille de l'application est 37349000 octets ($65^3 \times 17 \times 8$)
- l'application est composée de 128 itérations
- l'application est lancée sur seven (seven possède les données en entrée de l'algorithme)
- le classement final est composé des 15 meilleurs configurations

Résultat de l'algorithme évolutif

```
----- TOP 15 -----  :

(1) sekhmet:1   dinadan:1       lattice:1       caseb:1
        =>fitness: 6537.113 =  calc:6290.358 + comm:246.755
(2) pellinore:2 dinadan:1       lattice:1
```

```
        =>fitness: 7359.800 =  calc:7171.008 + comm:188.793
(3) sekhmet:1   pellinore:2     dinadan:1
        =>fitness: 7359.800 =  calc:7171.008 + comm:188.793
(4) pellinore:2 dinadan:1       caseb:1
        =>fitness: 7359.800 =  calc:7171.008 + comm:188.793
(5) sekhmet:1   pellinore:2     lattice:1
        =>fitness: 7398.786 =  calc:7171.008 + comm:227.778
(6) pellinore:2 lattice:1       caseb:1
        =>fitness: 7398.786 =  calc:7171.008 + comm:227.778
(7) sekhmet:1   pellinore:2     caseb:1
        =>fitness: 7398.786 =  calc:7171.008 + comm:227.778
(8) pellinore:1 dinadan:1       lattice:1       caseb:1
        =>fitness: 7469.714 =  calc:7171.008 + comm:298.706
(9) sekhmet:1   pellinore:1     dinadan:1       caseb:1
        =>fitness: 7469.714 =  calc:7171.008 + comm:298.706
(10) sekhmet:1  pellinore:1     dinadan:1       lattice:1
        =>fitness: 7469.714 =  calc:7171.008 + comm:298.706
(11) sekhmet:1  pellinore:1     lattice:1       caseb:1
        =>fitness: 7508.699 =  calc:7171.008 + comm:337.691
(12) minerve.cines.fr:4
        =>fitness: 7645.442 =  calc:6829.532 + comm:815.911
(13) pellinore:2        dinadan:1       marathon:1
        =>fitness: 8086.166 =  calc:7880.229 + comm:205.938
(14) sekhmet:1  pellinore:2     marathon:1
        =>fitness: 8193.373 =  calc:7880.229 + comm:313.144
(15) pellinore:2        marathon:1      lattice:1
        =>fitness: 8193.373 =  calc:7880.229 + comm:313.144
```

Commentaires Nous constatons que l'ordonnanceur sélectionne des ressources proches géographiquement, en l'occurrence à Illkirch. Ceci peut être justifié par le fait que l'application Vador est composé de processus communiquant fortement entre eux. Ainsi, une configuration composée de ressources éloignées entraîne une trop forte pénalité de communication. En outre, on peut observer que la ressource minerve, bien que possédant quatre processeurs, permettant ainsi aux processus de communiquer rapidement, est pénalisée par les communications initiales. En effet, les données de l'application se trouvent initialement sur seven, ressource se situant à Illkirch. Le temps de transfert des données de Illkirch à Montpellier est trop pénalisant dans cette situation.

5.3 Convergence de notre algorithme évolutif

La convergence de notre algorithme dépend de la méthode de sélection considérée. Nous avons mis en oeuvre trois types de sélection : la sélection par le fitness, par la roulette, et par le rang. Pour étudier la convergence de l'algorithme, nous avons mesuré le fitness moyen de chaque génération. La figure 5.1 est représentative des résultats observés dans la plupart des cas. La courbe correspond à l'ordonnancement sur 16 processeurs de l'application ray2mesh, de taille 62000000 octets, et dont le processeur maître est sekhmet.

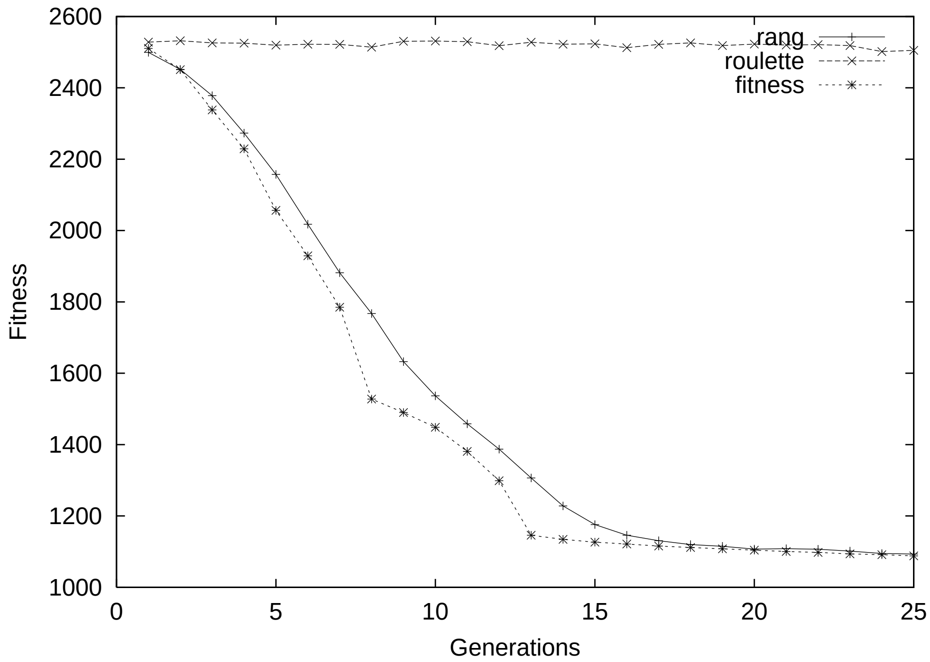

FIG. 5.1 – Convergence

Nous constatons qu'avec une sélection par la roulette, l'algorithme n'explore pas de meilleurs sous-espaces au fil des générations, nous n'observons pas de convergence. Ceci peut être expliqué par le fait que les valeur du fitness des individus sont trop proches les uns des autres. Ainsi, les probabilités qui sont affectées aux individus sont également proches les unes des autres. La sélection de la roulette est une méthode trop probabiliste étant donné les valeurs de nos fonctions fitness.

En revanche, la sélection par le fitness et par le rang convergent rapidement vers des solutions. Aucune de ces deux méthodes de sélection semble meilleure par rapport à l'autre. Suivant les cas, c'est l'une ou l'autre qui est la plus adaptée. En outre, l'algorithme a été exécuté plusieurs fois avec les mêmes paramètres pour s'assurer que la convergence rapide n'est pas une convergence prématurée vers un minimum local. Les résultats observés ont confirmé que la convergence n'était pas précoce.

Ces attitudes de convergence se répercutent au niveau du classement des configurations fourni par l'algorithme au final. La sélection par la roulette explorant moins de bons sous-espaces de recherche, les configurations retournées ne sont pas aussi bonnes que celles fournies avec les deux méthodes de sélection. Pour ces dernières, il a été constaté que les classements sont équivalents, ce qui est normal, vu leur convergence vers un même individu.

5.4 Performance de notre algorithme évolutif

Outre le fait que notre algorithme converge vers des solutions satisfaisantes, son temps d'exécution est suffisamment petit, montrant ainsi son efficacité. En effet, l'algorithme d'ordonnancement dans l'exemple 1 de ce chapitre, exécuté sur un pentium III 800 MHz, a convergé en 3 secondes, et en 22 secondes en étant exécuté sur un pentium II 350 MHz. L'application ordonnancée ayant un temps d'exécution de plus de 600 secondes, les résultats sont donc convaincants. De la même manière, les résultats en général ont répondu à l'attente portée sur eux, l'algorithme d'ordonnancement fournissant des configurations dans un temps satisfaisant.

Dans le pire des cas pour une exploration exhaustive, où la grille est composée de p monoprocesseurs, et où l'on souhaite une configuration de v processeurs, le nombre de configurations à explorer serait égal à :

$$C_p^v = \frac{p!}{v!(p-v)!}$$

ce qui nous donnerait, pour p=64 et v=8, environ 4.43×10^9 configurations à explorer.

En outre, l'espace de solutions de notre grille est composée de 11856000 individus. Les résultats sont trouvés en parcourant 50000 individus. On peut donc s'apercevoir que résoudre le problème de manière exhaustive serait irréalisable car explorer tous les individus pour contrôler s'ils satisfont les critères puis les évaluer rendrait l'exécution de l'algorithme d'ordonnancement très longue !

Conclusion

L'ordonnancement d'applications parallèles sur une grille, de par sa complexité liée à la distribution géographique des ressources, a suscité de nombreuses recherches sur des algorithmes de sélection de ressources appropriés. Ces études ont mis en évidence l'importance de la prédiction de performance pour estimer de la manière la plus précise le temps d'exécution de l'application ordonnancée et établir ainsi un classement des configurations de ressources le plus conforme possible à la réalité. Pour cela, deux catégories de prédiction sont primordiales : les prédictions sur les performances de ressources et les prédictions sur le comportement de l'application ordonnancée. Notre algorithme considère ces aspects puisque la fonction d'évaluation de nos individus est construite à partir des performances des ressources, prédites par l'intermédiaire du Network Weather Service, et à partir de l'analyse du comportement de l'application.

En outre, la nécessité de devoir explorer un ensemble de configurations très vaste, étant donné que des grilles peuvent être composées à grande échelle de milliers de processeurs, rend la recherche exhaustive de la meilleure solution irréalisable dans des délais convenables. L'algorithme d'ordonnancement se devant de trouver un ensemble de bonnes configurations dans des délais relativement brefs, la mise en oeuvre d'un algorithme évolutif semble être une bonne alternative à l'ordonnancement d'applications dans un environnement de type grille.

Notre algorithme d'ordonnancement donne des résultats suffisament précis dans le cadre de notre projet. Toutefois, il serait intéressant d'affiner les modèles de performances utilisés. Une amélioration viserait d'une part à introduire la prédiction de performance CPU, c'est-à-dire d'intégrer le taux de charge d'un processeur, et d'autre part à prendre en compte la taille de la RAM disponible sur un processeur donné. En effet, il a été constaté dans certaines expériences que certaines ressources swappaient durant l'exécution de l'application, engendrant par cette occasion un retard assez conséquent sur le temps d'exécution escompté. L'insertion de ces deux critères supplémentaires permettraient d'obtenir des estimations encore plus précises sur les temps de calcul.

Annexe A

Exemples d'ordonnancements

A.1 Exemple d'ordonnancement avec l'heuristique HEFT

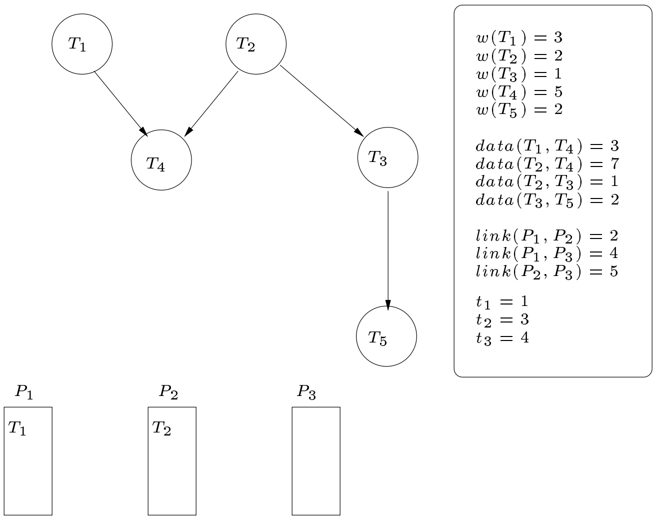

FIG. A.1 – Hypothèses

Considérons le *scheduling* de cinq tâches T_1 à T_5 (Figure A.1). Nous disposons d'un ensemble de trois processeurs P_1 à P_3. T_1 a déjà été placée sur P_1 et T_2 sur P_2. Il reste donc à placer T_3, T_4 et T_5. On calcule le *bottom level* de ces tâches.
$BL(T_3) = data(T_2, T_3) + w(T_3) + data(T_3, T_5) + w(T_5) = 1 + 1 + 2 + 2 = 6$ (2)
$BL(T_4) = data(T_1, T_4) + data(T_2, T_4) + w(T_4) = 3 + 7 + 5 = 15$ (1)
$BL(T_5) = data(T_3, T_5) + w(T_5) = 2 + 2 = 4$ (3)
Entre parenthèses, on en déduit le classement de priorité des tâches. On commence donc par traiter T_4. Si on décide de placer T_4 sur P_1, alors la communication entre T_1 et T_4 peut être négligé. Il s'agit alors d'ordonnancer la communication entre T_2 et T_4. Pour cela, il faut trouver l'intervalle de temps le plus tôt possible où il n'y a pas de communication entre P_2 et P_1. Cet intervalle

doit se trouver après la complétion de T_2 et doit durer au minimum $comm(T_2, T_4, P_2, P_1) = data(T_2, T_4) \times link(P_2, P_1)$. Une fois l'intervalle trouvé, on obtient une estimation de la date de complétion t_c de T_4 qui, pour le placement sur le processeur P_i d'une tâche T_i dépendante de T_j se trouvant sur P_j, est donné par

t_c = max(date au plus tôt de calcul de T_i sur P_i,date au plus tôt de communication entre P_j et P_i) + $comm(T_j, T_i, P_j, P_i)$ + $w(T_i)$ $\times t_i$.

$T_4 \rightarrow P_1 : t_c = 6 + 7 \times 2 + 5 \times 1 = 25$
$T_4 \rightarrow P_2 : t_c = 6 + 3 \times 2 + 5 \times 3 = 27$
$T_4 \rightarrow P_3 : t_c = 0 + 3 \times 4 + 7 \times 2 + 5 \times 4 = 46$
On place donc T_4 sur P_1. Puis on traite T_3 de la même manière.
$T_3 \rightarrow P_1 : t_c = 22 + 1 \times 2 + 1 \times 1 = 25$
$T_3 \rightarrow P_2 : t_c = 6 + 1 \times 3 = 9$
$T_3 \rightarrow P_3 : t_c = 0 + 1 \times 5 + 1 \times 4 = 9$
On choisit de placer T_3 sur P_2 (pas de communication engendrée). Pour finir, on traite T_5 de façon identique.
$T_5 \rightarrow P_1 : t_c = 22 + 2 \times 2 + 2 \times 1 = 28$
$T_5 \rightarrow P_2 : t_c = 9 + 2 \times 3 = 15$
$T_5 \rightarrow P_3 : t_c = 0 + 2 \times 5 + 2 \times 4 = 18$
On place donc T_5 sur P_2.

A.2 Exemple d'ordonnancement avec l'heuristique ILHA

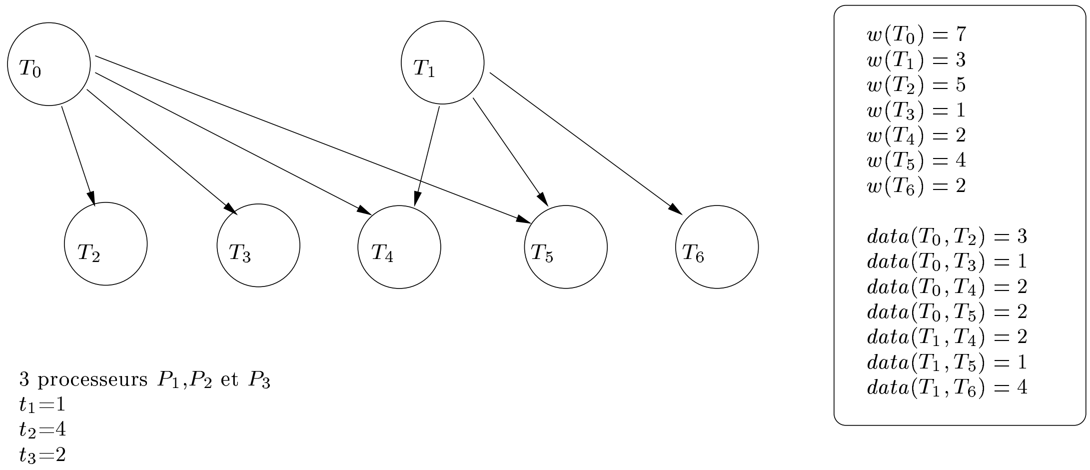

Fig. A.2 – Hypothèses

Exemple Considérons l'ordonnancement de sept tâches T_0 à T_6 (Figure A.2). Nous disposons de trois processeurs P_1,P_2 et P_3. On calcule leur charge de travail maximale.
$c_{1_{max}} = \lceil \frac{4}{7} \times 24 \rceil = 14$
$c_{2_{max}} = \lceil \frac{1}{7} \times 24 \rceil = 4$
$c_{3_{max}} = \lceil \frac{2}{7} \times 24 \rceil = 7$
On établit ensuite un classement de priorité des tâches en calculant les *bottom level.*
$BL(T_0) = w(T_0) + max_{i \in succ(T_0)}\{data(T_0, T_i) = w(T_i)\} = 7 + 8 = 15$ (1)
$BL(T_1) = w(T_1) + max_{i \in succ(T_1)}\{data(T_0, T_i) = w(T_i)\} = 3 + 6 = 9$ (2)
$BL(T_2) = data(T_0, T_2) + w(T_2) = 3 + 5 = 8$ (3)
$BL(T_3) = data(T_0, T_3) + w(T_3) = 1 + 1 = 2$ (7)
$BL(T_4) = data(T_0, T_4) + data(T_1, T_4) + w(T_4) = 2 + 2 + 2 = 6$ (5)
$BL(T_5) = data(T_0, T_5) + data(T_1, T_5) + w(T_4) = 2 + 1 + 4 = 7$ (4)
$BL(T_6) = data(T_1, T_6) + w(T_6) = 4 + 2 = 6$ (6)
On fixe le nombre de tâches considérées par étape à trois.
Première étape : on commence par traiter T_0,T_1 et T_2. T_0 et T_1 n'ont pas de prédecesseur, on applique la méthode HEFT pour les placer. Il est évident que T_0 est placée sur P_1 :
$c_1 = 0 + 7 = 8$
On traite ensuite T_1.
$T_1 \to P_1 : t_c = 7 + 3 \times 1 = 10$
$T_1 \to P_2 : t_c = 0 + 3 \times 4 = 12$
$T_1 \to P_3 : t_c = 0 + 3 \times 2 = 6$
On place donc T_0 sur P_3 :
$c_1 = 0 + 3 = 3$.
On traite ensuite T_2. T_2 dépend de T_0, on essaie donc de placer T_2 sur le même processeur que T_0, à savoir P_1 :
$c_1 = 7 + 5 = 12$ ce qui est encore bon.
Deuxième étape : on traite T_5,T_4 et T_6. T_5 dépend de T_0 et T_1, on essaie de le placer sur l'un des deux processeurs traitant T_0 et T_1. On constate que celui traitant T_0 excèderait sa charge maximale en traitant T_5 car c_1 vaudrait alors $12 + 4 = 16$. On essaie donc de placer T_5 sur celui traitant T_1, à savoir P_3 :
$c_3 = 3 + 4 = 7$ ce qui est encore bon (mais qui sature P_3). La tâche T_4 ne peut pas être placée sur P_3 saturé. On essaie de la placer sur P_1 :
$c_1 = 12 + 2 = 14$ ce qui est encore bon (mais qui sature P_1).
On traite ensuite T_6, qui dépend de T_1 mais P_3 est saturé. On essaie donc de placer T_6 sur le processeur non saturé restant (P_2) :
$c_2 = 0 + 2 = 2$ ce qui fonctionne.
Dernière étape, on traite T_3. P_1 et P_3 étant saturés, on choisit donc de placer T_3 sur P_2 ce qui fonctionne ($c_2 = 2 + 1 = 3$).

Annexe B

Prophet

B.1 Fonction de coût de communication

$$T_{comm}[C_i, \tau](mtaille, p) = (c_1 \times p) + \frac{p \times mtaille}{c_2}$$

où C_i est le cluster sur lequel les tâches sont placées, τ est la topologie de communication de l'application, *mtaille* est la taille de messages (nombre d'octets) tranmis en un cycle d'itération de l'exécution, p est le nombre de processeurs sélectionnés communiquant, c_1 est la latence du réseau, c_2 est la bande passante.

Il se peut que toutes les tâches ne puissent pas être placées sur un même cluster. La communication entre différents clusters entraîne une pénalité de routage. Il faut donc considérer en plus, dans la fonction de coût de communication, un coût de routage et un coût de conversion de données.

$$T_{routage}[C_i, C_j](mtaille) = r_1 + r_2 \times mtaille$$

où r_1 est une pénalité de latence, r_2 une pénalité par octet routé de C_i vers C_j

$$T_{conversion}[C_i, C_j](mtaille) = e_1 \times mtaille$$

où e_1 est le coût pour convertir un octet codé en mode big-endian en octet codé en mode little-endian (ou l'inverse) suivant les architectures des machines (sur le réseau circulent des données en séquences de 8 bits au format big-endian)

Une fonction de coût total de communication est définie comme suit :

$$T_{comm_{total}}[C_i, \tau] = T_{comm}[C_i, \tau] + k \times (T_{routage}[C_i, C_j] + T_{conversion}[C_i, C_j])$$

où k est le nombre de messages routés par cycle

Cette fonction dépend de la topologie τ de l'application, comme évoqué précédemment :

$$T_{comm_{total}}[C_i, 1D] = \max_i T_{comm_{total}}[C_i, 1D]$$

$$T_{comm_{total}}[C_i, anneau] = \sum_i T_{comm_{total}}[C_i, anneau]$$

$$T_{comm_{total}}[C_i, arbre] = T_{comm_{total}}[C_{racine}, arbre] + \max_{i\epsilon feuilles} T_{comm_{total}}[C_i, arbre]$$

B.2 Fonction de coût de calcul

Elle est définie comme suit :

$$T_{cal} = \frac{ctaille \times coutarchi \times nbPDU}{p}$$

où $nbPDU$ est le nombre de PDU (Primitive Data Unit : plus petit morceau du domaine de données, par exemple 4 octets) manipulés durant la phase de calcul, $ctaille$ est la quantité de calcul exécuté sur un PDU, $coutarchi$ est un coût d'exécution spécifique à l'architecture du processeur.

B.3 Equilibrage de la charge

Soient A_i le nombre de PDU assignés au processeur p_i et g( ) la quantité de calcul à exécuter , qui est fonction de A_i. On décrit la quantité de calcul à exécuter sur le processeur p_i comme suit :

$$T_{cal}[p_i] = ctaille \times coutarchi \times g(A_i)$$

Soient $d_1^{(i)}$ le coût de calcul sur le processeur p_i indépendant de la quantité de PDU assignée, $d_2^{(i)}$ le coût de calcul sur le processeur p_i dépendant de la quantité de PDU assignée. Nous pouvons réécrire la quantité de calcul à exécuter sur le processeur p_i :

$$T_{cal}[p_i] = d_1^{(i)} + d_2^{(i)} \times g(A_i)$$

Un équilibrage de la charge nous donne :

$$d_1^{(1)} + d_2^{(1)} \times g(A_1) = d_1^{(2)} + d_2^{(2)} \times g(A_2) = ... = d_1^{(p)} + d_2^{(p)} \times g(A_p) \forall p$$

En émettant l'hypothèse que le coût de calcul indépendant de la quantité de PDU assignée est nul, nous obtenons :

$$d_1^{(i)} = 0 \forall i$$

Soient P_i le nombre de processeur selectionnés sur le cluster C_i, W_i le poids relatif du processeur p_i

$$W_i = \frac{d_2^{(i)}}{\max_j d_2^{(j)}}$$

On obtient le nombre de PDU assignés au processeur p_i

$$A_i = \sum_j \frac{W_i}{W_j \times P_j} \times nbPDU$$

Annexe C

Exemples de modèle d'application

C.1 Calcul Maître-esclave

Pour une classe d'application correspondant à un calcul maître-esclave, la structure de temps d'exécution dérivée est :

$$T_{exec} = T_{calcmaitre} + Scatter + \max_i T_{calcesclave_i} + Gather$$

C.2 Les codes PMHD3D

Les codes PMHD3D [DOB$^+$00], développés pour des simulations magnetohydrodynamiques, sont des codes data-parallèles MPI Fortran composés d'opérations stencil [Mei02]. Les opérations stencil sont des nids de boucles parcourant une grille de références en effectuant en chaque point un calcul de voisinage c'est-à-dire des boucles dans lesquelles les références à un même tableau sont égales à une translation près. Le même calcul est itérativement exécuté sur chaque point en utilisant les données de ses voisins. Cette régularité dans le code permet d'établir un modèle de performance adapté. Le temps d'exécution T_i sur un processeur P_i est estimé à :

$$T_i = Comp_i + Comm_i + Over_i$$

où $Comp_i$ est le temps prévu de calcul, $Comm_i$ le temps prévu de communication et $Over_i$ une estimation de frais dépendante du nombre de processeurs impliqués dans le calcul et de la taille du problème.

Ces trois valeurs sont estimées comme suit :

$$Comp_i = \frac{x_i \times BM_i}{Avail_i}$$

où x_i représente la quantité de travail allouée au processeur P_i, BM_i est une valeur caractérisant le temps mis pour exécuter l'application (obtenue par benchmark) et $Avail_i$ est une prédiction de la charge du CPU (obtenue par l'intermédiaire de NWS).

$$Comm_i = \frac{MB}{(b_{i,i+1} + b_{i,i-1})} + M \times (l_{i,i+1} + l_{i,i-1})$$

où MB est le nombre de Mo transférés, M est le nombre de messages transférés, $b_{i,j}$ et $l_{i,j}$ sont des prédictions de la bande passante disponible et de la latence entre les processeurs P_i et P_j (obtenues par NWS).

$Over_i$ a été estimé empiriquement :

$$Over_i = 16 - 1,5 \times \frac{probsize}{1000} + 0,094 \times P^2$$

où P est le nombre de processeurs impliqués dans le calcul et $probsize$ définit la taille de l'application (taille des boucles).

En outre, pour mieux guider le choix des ressources, une contrainte importante de performance de codes PMHD3D est considérée. Ce type d'applications nécessite une quantité de mémoire disponible suffisamment élevée. Cette contrainte est exprimée de la manière suivante :

$$BMmem_i \times x_i \leq MemAvail_i$$

où $MemAvail_i$ est la quantité de mémoire disponible sur le processeur P_i (information fournie par un capteur NWS) et $BMmem_i$ est la quantité de mémoire utilisée par l'application lors du benchmark.

Bibliographie

[BAG00a] Rajkumar Buyya, David Abramson, and Jonathan Giddy. An economy driven resource management architecture for global computational power grids. *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.

[BAG00b] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g : An architecture for a resource management and scheduling system in a global computational grid. *The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000)*, May 2000.

[BBR01] Olivier Beaumont, Vincent Boudet, and Yves Robert. A realistic model and efficient heuristic for scheduling with heterogeneous processors. Technical Report 2001-37, ENS Lyon - INRIA, September 2001.

[Bäc96] T. Bäck. *Evolutionary Algorithms in Theory and Practice, Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford Press, 1996.

[BCD00] Rajkumar Buyya, Steve Chapin, and David DiNucci. Architectural models for resource management in the grid. *The First IEEE/ACM International Workshop on Grid Computing (GRID 2000)*, December 2000.

[BM99] Ladislau Bölöni and Dan C. Marinescu. Robust scheduling of metaprograms. *Proceedings of 8-th Heterogeneous Computing Workshop HCW'99*, pages 146–155, 1999.

[BW97] Francine Berman and Richard Wolski. The apples project : A status report. Technical report, University of California, San Diego, 1997.

[CCE+93] J. Carlier, P. Chrétienne, J. Erschler, C. Hanen, P. Lopez, A. Munier, E. Pinson, M.-C. Portmann, C. Prins, C. Proust, and P. Villon. *Les problèmes d'ordonnancement*, volume 27. RAIRO-Recherche Opérationnelle, 1993.

[Cos90] M. Costabel. A remark on the regularity of solutionsof Maxwell's equations on Lipschitz domains. *Math. Meth. in the AppliedSciences*, 12 :365–368, 1990.

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection.* 1859.

[Dat01] WP01 report on current technology. Technical report, DataGrid, 2001.

[DGK+01] Peter Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, , and Dean Sutherland. The architecture of the Remos system. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, August 2001.

[DOB+00] Holly Dail, Graziano Obertelli, Francine Berman, Rich Wolski, and Andrew Grimshaw. Application-aware scheduling of a magnetohydrodynamics application in the legion metasystem. In *Proceedings of the 9th Heterogenous Computing Workshop*, May 2000.

[FFK+97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[FK97] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.

[FSB01] Francis Filbet, Eric Sonnendrücker, and Pierre Bertrand. Conservative numerical schemes for the Vlasov equation. *Journal of Comput. Phys.*, 172 :166–187, 2001.

[GGM02] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Parallel seismic raytracing in a global earth mesh. In *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[GR96] J. Gehring and A. Reinefeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. In *Proceedings of Future Generation Computer Systems*, 1996.

[Hol92] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1992.

[LE99] P. Lopez and P. Esquirol. *L'ordonnancement.* Economica, Paris, 1999. ISBN 2-7178-3798-1.

[MBHJ98] Dan C. Marinescu, Ladislau Bölöni, Ruibing Hao, and Kyungkoo Jun. An alternative model for scheduling on a computational grid. *The Thirteenth International Symposium on Computer and Information Sciences*, pages 473–480, 1998.

[Mei02] Benoît Meister. Localité de données et compression mémoire dans les opérations stencil. Technical report, LSIIT ICPS Université Louis Pasteur de Strasbourg, 2002.

[Mes95] Message Passing Interface Forum. *MPI : A message-passing Interface Standard*, June 1995.

[Sch00] Jennifer M. Schopf. Ten steps for superscheduling. Technical report, Northwestern University, February 2000.

[SM97] M. G. Sirbu and D. C. Marinescu. A scheduling expert advisor for heterogeneous environments. In *Proceedings of the IICW 97 Heterogeneous Computing Workshop*, pages 74–82. IEEE Press, 1997.

[VF02] Eric Violard and Francis Filbet. Parallelization of a Vlasov solver by communication overlapping. In *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[WG95] Jon B. Weissman and Andrew S. Grimshaw. A framework for partitioning parallel computations in heterogeneous environments. *Concurrency : Practice and Experience*, 7(5) :455–478, August 1995.

[WSH98] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*. Institute of Electrical and Electronics Engineers, 1998.

[WSH99] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service : A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6) :pp. 757–768, October 1999.

[WZ97] Jon B. Weissman and Xin Zhao. Run-time support for scheduling parallel applications in heterogeneous nows. *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 347–355, August 1997.

[WZ99] Jon B. Weissman and Xin Zhao. Scheduling parallel applications in distributed networks scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1, 1999.