NUMERO D'ORDRE : nnnn

ANNÉE : 1998

USTL

LIFL Laboratoire d'Informatique Fondamentale de Lille

THÈSE

Version Provisoire

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Zouhir HAFIDI

MARS : Un environnement de programmation parallèle adaptative dans les réseaux de machines hétérogènes multi-utilisateurs

Thèse soutenue le xx septembre1998, devant la commission d'examen :

Président :	xxx YYY
Directeurs de thèse :	Jean-Marc GEIB
	El-Ghazali TALBI
Rapporteurs :	Catherine ROUCAIROL
	Hervé GUYENNET
Examinateurs :	xxx YYY

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 03.20.43.47.24 Fax. 03.20.43.65.66

Table des matières

Table des figures

Table des tableaux

Introduction

Motivations, objectifs et cadre de travail

De nos jours, la plupart des applications les plus importantes sont les applications dites *de longue durée de vie* notamment en optimisation combinatoire, en calcul scientifique et en imagerie. Celles-ci sont non seulement de plus en plus complexes, mais aussi très exigeantes en matière de ressources de traitement. Malgré la montée en puissance et en vitesse des super-calculateurs, les applications et les problèmes actuels restent intraitables en temps raisonnable. De plus, l'utilisation de ces super-calculateurs au sein des institutions qui les possèdent demeure réglémentée car ils sont trop onéreux pour être banalisés.

La prolifération de stations de travail puissantes avec un rapport coût/performance constamment en baisse et l'évolution rapide des technologies de communication (FDDI, ATM, Myrinet, etc.), a donné lieu à l'émergence des réseaux de stations (Networks Of Workstations ou NOWs) comme une alternative architecturale pour le traitement parallèle à moindre coût. Des études récentes telles que les nôtres vont jusqu'à l'intégration de ces plateformes avec les clusters de processeurs (Clusters Of Workstations ou farms) et les machines dites massivement parallèles (MPPs) pour donner une vue macroscopique de tout le système : *un méta-système.*

Les différents composants d'un méta-système sont reliés par des réseaux d'interconnexion de type LAN et/ou WAN et permettent de délivrer une puissance globale non négligeable pouvant avoisiner plusieurs dizaines de gigaflops. Cependant, l'analyse de la charge des NOWs durant de longues périodes de temps a montré que seulement un faible pourcentage de la puissance de calcul disponible est utilisé. Les machines sont non seulement inexploitées durant la nuit, les week-ends et les périodes de vacances mais aussi sous-utilisées pendant les heures de travail. Des études ont d'ailleurs révélé que les réseaux de stations sont globalement sous-utilisés à 75% du temps au moins [Nic87] [TL89]. Ceci montre la puissance de calcul potentielle d'un méta-système en vue d'exécuter des applications parallèles de longue durée de vie.

Malheureusement, l'exploitation d'un méta-système reste délicate car non seulement il

est dépourvu d'une couche logicielle fournissant toutes les fonctionnalités nécessaires à sa gestion, mais aussi il présente des « irrégularités » à différents niveaux et qui doivent être prises en compte judicieusement :

- **hétérogénéité matérielle :** au niveau des processeurs (jeu d'instructions, format de données), architectures (PC, station de travail, station multiprocesseurs, machine massivement parallèle), réseaux (type, performance, topologie, protocole).
- **hétérogénéité logicielle :** au niveau du système d'exploitation (type, nature, système de fichiers, gestion des ressources, gestion des processus, gestion de la mémoire virtuelle), modèle de programmation (échange de messages, parallélisme de données, mémoire partagée), modèle d'exécution (send/receive, rendez-vous, client/serveur).
- **irrégularité de l'utilisation du méta-système :** on distingue en effet trois catégories d'utilisateurs :
  - **fortuits :** ce sont les utilisateurs qui lancent essentiellement des tâches interactives (édition, courrier électronique, web, etc.). Ils exploitent seulement l'aspect interactif de leurs machines mais rarement sinon jamais toute la puissance à leur disposition.
  - **intermittents :** il s'agit des utilisateurs qui exécutent principalement des tâches de courtes durées (compilation, brève exécution, etc.). Ils exploitent pleinement leurs machines mais de façon sporadique.
  - **frustrés :** ce sont les utilisateurs dont les besoins en puissance de traitement sont largement supérieures à leurs machines. Ces utilisateurs souhaitent exploiter les ressources des deux premiers groupes d'utilisateurs sans trop gêner leurs activités.
- **pannes des machines :** cette plateforme à grande échelle est sujette à des pannes très fréquentes qui empêchent les applications qu'elle véhicule de s'exécuter à terme.

Face à la complexité de ces plateformes ainsi que leur caractère fortement dynamique, l'objectif de la thèse est double. D'une part, la nécessité de mettre en place un *système logiciel* offrant une gestion efficace des différentes ressources d'un méta-système, et un maximum de transparence pour les développeurs d'applications parallèles. D'autre part, l'importance de la spécification de *méthodologies de développement* d'applications parallèles basées sur l'ajustement adaptatif du degré de parallélisme des applications en fonction de la charge du système.

Ce travail s'inscrit dans le cadre du projet ESPACE (Execution Support for Parallel Applications in high performance Computing Environments) de l'axe ParDis (Informatique Parallèle et Distribuée) du LIFL (Laboratoire d'Informatique Fondamentale de

Lille). L'objectif du projet ESPACE est de définir un cadre méthodologique et de développer un environnement d'exécution pour applications parallèles sur des architectures à mémoire distribuée. La métaphore est une fleur dont le cœur est constitué de l'*environnement d'exécution* et dont les pétales représentent des *classes d'applications* (figure 0.1) :

- **environnement d'exécution :** vu les inconvénients d'une structure monolithique, l'environnement d'exécution est disposé en couches. La couche la plus interne représente un *support d'exécution* (PM^2 : Parallel Multithreaded Machine) dont le modèle est basé sur le concept du multithreading distribué [Nam96]. Ce dernier combine l'utilisation de la notion d'appel de procédure à distance (Remote Procedure Call ou RPC), et la notion de processus légers (threads) pour arriver au concept de LRPC (Light weight Remote Procedure Call) qui consiste à créer un processus léger pour exécuter une procédure distante. PM^2 repose donc sur une bibliothèque de processus légers développée au sein de l'équipe, et sur une bibliothèque de communication en l'occurrence PVM [BDG+93][1].

  Pour des raisons de généricité, PM^2 n'intègre pas directement des mécanismes de *régulation de charge et d'ordonnancement en contexte distribué.* En effet, la couche au dessus est dédiée aux régulateurs et aux ordonnanceurs. Deux pistes complémentaires sont actuellement étudiées :

  - *régulation de charge dans un milieu dédié :* cette stratégie a pour but d'équilibrer la charge des machines en s'appuyant sur l'ordonnancement en contexte distribué des processus légers véhiculés par une même application parallèle. Cette dernière s'exécute dans un environnement mono-application et la régulation de charge est intra-application. Ceci constitue le travail de thèse d'Yves Denneulin qui a mis en place le régulateur LBMP (Load Balancing with Migration directed by Priorities) [DNGM96]. LBMP est basé essentiellement sur des fonctionnalités fournies par la bibliothèque de processus légers (priorité des processus légers), et PM^2 (migration de processus légers).
  - *régulation de charge dans un milieu non dédié :* cette stratégie a pour but d'équilibrer la charge des machines en s'appuyant sur l'ordonnancement inter-applications dans un contexte multi-utilisateurs. Ceci constitue le travail de cette thèse dont le résultat a permis la mise en place de l'environnement MARS (Multi-user Adaptive Resource Scheduler). Il s'agit d'un exécutif permettant le contrôle de la charge d'un ensemble de machines et d'une bibliothèque de primitives permettant le développement d'applications parallèles bénéficiant du modèle MARS. Ce dernier repose sur l'*adaptabilité* du degré de parallélisme des applications en fonction de la charge et de l'utilisation interactive des machines.

[1] Notons que la nouvelle version de PM^2 appelée PM^2Hi-perf repose sur une couche, nommée Madeleine, complètement indépendante de la bibliothèque de communication utilisée.

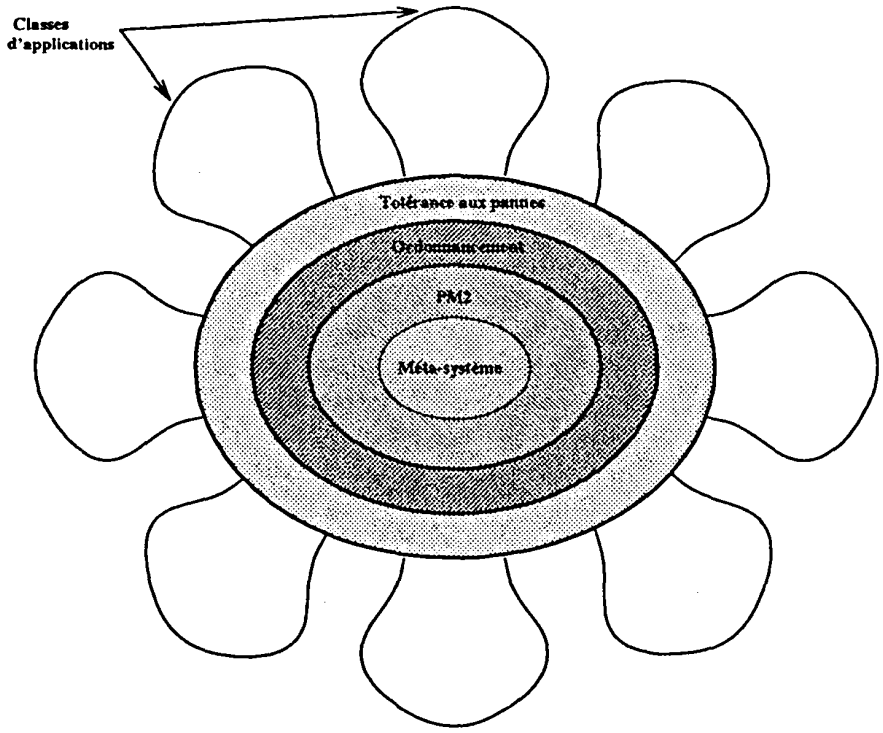


Figure 0.1 : Le projet ESPACE

Vu la fréquence élevée des pannes dans un méta-système, la couche la plus externe renferme des mécanismes de tolérance aux pannes assurant une robustesse pour les applications qui s'y exécutent.

- **classes d'applications :** bien que l'environnement d'exécution complet soit entièrement conçu et développé par notre équipe, les diverses applications sont en partie menées par des équipes de recherche essentiellement françaises. Les domaines varient des langages fonctionnels à l'optimisation combinatoire en passant par le parallélisme de données et les langages à objets. Dans le cadre du système MARS, les applications que nous avons développées sont surtout axées autour de l'optimisation combinatoire en raison des compétences de l'équipe dans ce domaine. Des collaborations sont entreprises notamment dans le domaine de l'optimisation combinatoire (équipe PNN de l'UVSQ à Versailles, équipe ALG de l'université du Littoral, Institut de Physique Nucléaire à Orsay, IDSIA en Suisse) ainsi que dans le domaine du calcul scientifique (équipe MAP du LIFL à Lille, Université Centrale du Venezuela) et de l'imagerie (University of Al-Akhawayn au Maroc, équipe VISC de l'université du Littoral).

Organisation de la thèse

Le rapport de la thèse est réparti sur six chapitres regroupés en deux parties. La première partie, constituée des chapitres 1, 2 et 3, est consacrée à l'aspect système tandis que les chapitres 4, 5 et 6 de la deuxième partie sont axés sur l'aspect applicatif. Les conclusions et les perspectives de ce travail sont présentées à la fin de la thèse suivies de trois annexes.

Dans le chapitre 1, nous présentons la notion de méta-système et son utilisation comme une alternative architecturale pour le traitement parallèle à moindre coût. Nous passons en revue les différents indicateurs de charge permettant de caractériser la puissance potentielle de traitement que peut fournir un méta-système. Une étude expérimentale permet de confirmer le fait qu'une partie importante des ressources est sous-utilisée et pourrait être exploitée moyennant la mise en place d'une infrastructure adéquate.

Le chapitre 2 présente un état de l'art des environnements d'exécution et plus particulièrement ceux permettant l'utilisation du temps perdu dans un méta-système. Nous montrons qu'on peut classer les différents environnements d'exécution en trois catégories selon un critère d'adaptabilité. Ce dernier est lié à l'exploitation des machines en fonction de leur état de charge et de leur utilisation interactive. La catégorie des environnements d'exécution adaptative regroupe les systèmes qui tiennent compte du critère d'adaptabilité en ajustant dynamiquement le nombre et la localisation des processus en fonction de la fluctuation de charge dans le méta-système. Une étude comparative en fin de chapitre permet de montrer l'apport de notre infrastructure par rapport à celles déjà existantes.

Dans le chapitre 3, nous présentons de façon détaillée tous les éléments faisant partie de l'environnement développé dans cette thèse. Nous commençons par l'exécutif MARS composé d'un ensemble de gestionnaires permettant de contrôler la fluctuation de charge des machines et leur utilisation interactive, et d'un serveur de groupe ayant une vue globale de tout le méta-système. Le serveur de groupe a la capacité de prendre des décisions quant à l'attribution ou non des machines aux applications parallèles qu'il gère. La présence d'une bibliothèque de primitives contribue au développement d'applications parallèles de type maître/esclaves exploitant le modèle adaptatif de MARS. Ce dernier est caractérisé par le dépli (création de nouveaux processus) dans le cas de la disponibilité de machines et de repli (retrait de processus) dans le cas contraire.

Dans le chapitre 4, nous présentons une méthodologie pour le développement d'applications parallèles adaptatives. Dans la programmation parallèle adaptative, l'utilisateur doit respecter certaines règles contribuant à l'exécution adaptative de son application. Il doit fournir certains éléments dépendant de l'application tels que l'action à entreprendre au moment du dépli, du repli et de la mort d'un processus appartenant à son application suite à une panne de machine. Nous présentons par la suite des règles générales, s'appliquant à des classes d'applications, pour la gestion du travail entre les différents processus. Nous appuyons notre démarche par l'étude d'un exemple

d'application.

Le chapitre 5 est consacré aux méta-heuristiques parallèles adaptatives. Nous nous sommes intéressés particulièrement à la recherche tabou. Nous présentons la méthode et les différentes parallélisations existantes dans la littérature puis nous décrivons notre modèle adaptatif. Des résultats expérimentaux montrent le comportement de telles applications dans un environnement opportuniste tel que celui géré par MARS. Des efforts ont été déployés pour améliorer la qualité des solutions obtenues en appliquant ces méthodes au problème d'affectation quadratique.

Dans le chapitre 6, nous nous intéressons aux méthodes exactes d'optimisation combinatoire. Ces méthodes s'appliquent à des problèmes généralement NP-complets et sont réputées pour être gourmandes en temps machine et en espace mémoire. Nous avons étudié la parallélisation sous MARS de deux méthodes d'optimisation en l'occurence la recherche itérative en profondeur d'abord avec heuristique (IDA*) appliquée au problème du taquin 15 et la méthode de séparation/évaluation (Branch-and-Bound) appliquée au problème d'affectation quadratique.

En annexe A, nous présentons des informations concernant l'installation et l'utilisation de la distribution MARS. L'annexe B contient toutes les primitives de l'API de MARS. Enfin, l'annexe C contient des portions de code d'un exemple d'application de recherche de nombres premiers.

Terminologie

Dans la suite du manuscrit, nous utiliserons le terme *nœud* pour désigner un processeur d'une MPP ou une station de travail dans le méta-système exécutant le système d'exploitation UNIX. Une *application* parallèle est définie comme étant un ensemble de *tâches*. Une tâche est elle même constituée d'un ensemble de *threads*. Une *unité de travail* définit la granularité d'une application.

Première partie

Vers un environnement d'exécution parallèle adaptative

Chapitre 1

Caractérisation de la puissance potentielle des méta-systèmes

Il est primordial pour un ordonnanceur d'applications parallèles de définir la disponibilité d'un nœud et par conséquent de quantifier la puissance potentielle qui peut être mise à la disposition des applications parallèles. Les conditions dans lesquelles un nœud est libre sont souvent liées aux indicateurs de charge utilisés. Dans ce chapitre, nous énumérons la plupart des indicateurs de charge, décrivons les outils qui permettent de les calculer, et passons en revue les critères de disponibilité utilisés par certains ordonnanceurs. En dernier lieu, nous présentons une étude expérimentale qui nous a permis de caractériser la puissance potentielle du méta-système existant autour de notre laboratoire.

1.1 Les méta-systèmes comme plateformes de machines parallèles

Durant les deux dernières décennies, il y a eu un accroissement exponentiel des ressources de traitement interconnectées. Le dernier rapport du Network Information Systems Center datant de 1992 [Lot92] résume cette situation en montrant le nombre croissant de machines connectées à Internet (figure 1.1). Plus de 725000 machines ont été connectées via approximativement 17000 domaines en dix ans seulement. Cette évolution a été accompagnée en amont par l'amélioration des ressources de traitement [Che92] :

- les processeurs ont doublé de performance tous les huit mois et continuent à monter en puissance au même rythme que les super-calculateurs.

- les fréquences d'horloge sont passées de 0.2 Mhz en 1971 à 50 Mhz en 1991 et dépassent le cap des 200 Mhz actuellement.
- les débits des réseaux locaux sont améliorés par un facteur de 10 chaque décennie. En 1980, Ethernet opérait à 10 Mbits/s. En 1990, FDDI permet des débits de 100 Mbits/s. Des débits de l'ordre de 1 Gbits/s sont annoncés par certains constructeurs pour l'an 2000.
- les puces mémoire ont quadruplé en capacité tous les trois ans depuis 1972.
- les capacités de stockage des disques magnétiques ont évolué d'une densité de 1Kbits/$pouce^2$ (1957) à 1Gbits/$pouce^2$ en 1990. Elles ont doublé tous les trois ans.

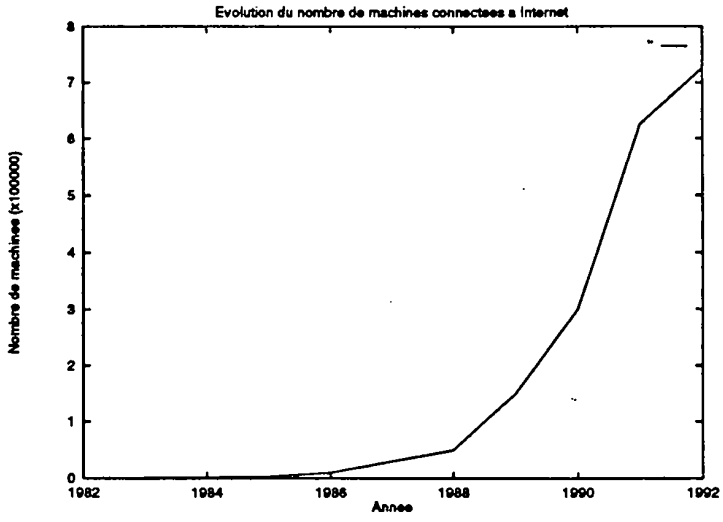


Figure 1.1 : Evolution du nombre de machines connectées à Internet

Cette évolution rapide des réseaux de stations de travail performantes et leur intégration avec des machines massivement parallèles a donc donné naissance à une nouvelle alternative architecturale pour le traitement parallèle à moindre coût (méta-système), et un nouveau paradigme de programmation appelé traitement sur machines interconnectées ou méta-traitement (metacomputing). Nous allons montrer par la suite, à travers une étude expérimentale, qu'un tel méta-système est une source importante de puissance de traitement qui peut être mise à la disposition des applications parallèles. Nous commençons dans un premier lieu par présenter les différents indicateurs de charge qui nous permettent de quantifier la puissance de traitement que peut fournir un méta-système.

1.2 Aperçu sur les principaux indicateurs de charge

Contrairement à ce que pourraient penser la plupart des utilisateurs, l'unité centrale (*CPU*) n'est pas le seul facteur impliqué dans le calcul de la charge. Il faut en plus tenir compte de la mémoire et du système d'entrées/sorties (essentiellement disque et réseau). Pour être plus significative, la détermination de la charge d'une machine doit donc porter sur l'ensemble des ressources. La communauté des utilisateurs est un autre facteur très important à prendre en considération car l'aspect « propriété de la machine » est omniprésent surtout dans les réseaux de stations de travail.

1.2.1 Facteurs impliqués dans le calcul de charge

- **CPU** : dans tout système à temps partagé même mono-utilisateur, plusieurs processus utilisent la CPU au même moment. Dans la plupart des cas, le noyau UNIX est capable d'allouer le processeur de façon équitable mais la CPU devient vite saturée dès que le nombre de processus dépasse une certaine valeur proportionnelle entre autres à la vitesse du processeur. Un moyen rapide de constater la charge du processeur est le calcul de la charge moyenne (*load average*) telle que définie par le système BSD[1] et reportée par la commande `uptime`[2]. Pour une vision plus détaillée de l'utilisation de la CPU, le système fournit la commande `ps`.

- **mémoire** : la saturation de la mémoire survient lorsque les besoins en mémoire des processus actifs dépassent la mémoire physique disponible. Pour gérer ce manque de mémoire sans causer un « crash » du système ou tuer des processus, le système utilise le mécanisme de *pagination* qui consiste à déplacer des portions de processus actifs sur disque afin de libérer la mémoire physique. La pagination est différente du *swapping* (va et vient) qui consiste à déplacer des processus entiers sur disque. La pagination et le swapping indiquent que le système ne peut plus fournir de la mémoire aux processus qui s'y exécutent. A ce stade, les performances du système se dégradent considérablement. Cependant, sous certaines circonstances, le swapping peut paraître comme un phénomène normal (voir la commande `ps` plus loin dans ce chapitre). Sous le système BSD, les outils tels que `vmstat` et `pstat` montrent si le système est en train de paginer. La commande `ps` permet de reporter les demandes en mémoire de chaque processus. Sous le système V, l'utilitaire `sar` fournit des informations sur tous les aspects concernant la mémoire.

- **système d'E/S disque** : le système d'E/S est fréquemment une source de surcharge et de congestion. Les E/S disque et les E/S réseau sont particulièrement importantes. Les bus du système d'E/S peuvent transférer seulement quelques méga octets par seconde. Cette bande passante est partagée par tous les processus

[1] La charge moyenne tente de mesurer le nombre de processus actifs, cependant une définition plus précise est donnée plus loin dans ce chapitre.

[2] `sar -q` sous le système V.

y compris le système d'exploitation. UNIX ne possède pas des outils performants pour analyser le système d'E/S. Sous UNIX BSD, `iostat` donne des informations sur les taux de transfert de chaque disque. Les commandes `ps` et `vmstat` peuvent donner des informations sur le nombre de processus bloqués en attente d'une opération d'E/S disque. Sous le système V, `sar` peut fournir plusieurs informations concernant l'efficacité des E/S, et `sadp` (V.4) donne des informations détaillées concernant les accès disque. Cependant, il n'existe pas d'outil standard permettant de mesurer le temps de réponse du système d'E/S à une charge de travail donnée.

- **système d'E/S réseau** : les problèmes d'E/S réseau se présentent généralement sous deux formes : un réseau peut être surchargé (congestion) ou un réseau peut transmettre des données dont l'intégrité est altérée. Lorsqu'un réseau est surchargé, la quantité de données à transférer à travers le réseau est supérieure à la capacité de celui-ci. Ainsi, le taux de transfert actuel pour tout processus devient relativement lent. Les problèmes de surcharge sont essentiellement liés à la configuration du réseau. Les problèmes d'intégrité apparaîssent une fois que le réseau est défaillant et que le transfert de données se fait de façon incorrecte. Afin de délivrer correctement les données à une application via le réseau, les protocoles réseau peuvent être amenés à transmettre chaque bloc de données plusieurs fois.

  Comme tout support de communication, le réseau possède une bande passante finie qui permet le transfert de données à des taux limités. Pour Ethernet par exemple, le taux maximum théorique est de 10 méga octets par seconde. Même dans un réseau très fluide, les taux de transfert réels sont nettement inférieurs. L'inefficacité des contrôleurs du réseau ainsi que le surcoût induit par les couches des protocoles réseau réduisent de façon significative les performances. Lorsqu'un réseau est en congestion, des collisions de paquets se produisent. Une collision de paquets est la conséquence d'une transmission simultanée de plusieurs paquets sur le réseau. Les transmissions se heurtent et altèrent les données. Après avoir détecté une collision, le réseau tente de retransmettre les mêmes données après un délai d'attente. Le moyen le plus simple de détecter les collisions est la commande `netstat -i`. Un nombre élevé de collisions pendant de longues périodes de temps indique un trafic important sur le réseau et donc une éventuelle congestion.

- **communauté des utilisateurs** : le comportement des utilisateurs a souvent une répercussion significative sur la façon d'utiliser l'ensemble des ressources et par conséquent sur le calcul de charge. On peut classer les utilisateurs dans plusieurs catégories :

  - **fortuits** : ce sont les utilisateurs qui exploitent seulement l'aspect interactif de leurs machines mais rarement sinon jamais toute la puissance à leur disposition. En effet, ces utilisateurs lancent essentiellement des tâches interactives (édition, courrier électronique, web, etc.) dont les besoins en ressources telle

que la CPU sont pratiquement nuls. En général, tous les utilisateurs qui ne font pas de développement d'applications font partie de cette catégorie.

- **intermittents :** il s'agit des utilisateurs qui exploitent pleinement leurs machines mais de façon sporadique. Ces utilisateurs se limitent généralement à la seule utilisation de leurs machines. Ils exécutent principalement des tâches de courtes durées entre-coupées dans le temps (commandes UNIX, compilation, brève exécution, etc.). Les développeurs d'applications séquentielles font partie de cette catégorie. Lors de la phase de développement et de mise au point de leurs applications, ces utilisateurs peuvent tolérer la présence de processus distants. Cette situation devient critique une fois que ces utilisateurs sont dans la phase d'évaluation de leurs applications en effectuant des mesures de performances. En effet, toute intrusion même mineure est intolérable.
- **frustrés :** ce sont les utilisateurs dont les besoins en puissance de traitement sont largement supérieurs à leurs machines. Typiquement, ce sont les développeurs d'applications parallèles et distribuées. Ces utilisateurs souhaitent exploiter les ressources des deux premiers groupes d'utilisateurs sans trop gêner leurs activités. En pratique, il est très difficile de savoir à quel moment il faut éviter d'utiliser telle ou telle machine car son propriétaire n'est pas obligé de prévenir pour l'exploiter de façon exclusive. Ceci engendre parfois des conflits entre utilisateurs.

La plupart des processus lancés sur les machines appartiennent au premier groupe d'utilisateurs. Des études statistiques sur 122000 processus observés ont montré que 70% à 80% nécessitent moins de 0.5 seconde de temps CPU, plus de 78% des processus ont une durée de vie inférieure à 1 seconde et 97% ont une durée de vie inférieure à 8 secondes [Cab86]. La durée de vie moyenne d'un processus étant de 0.4 seconde.

Parmi les utilisateurs, certains tolèrent le partage de leur machine avec d'autres utilisateurs pourvu que le temps de réponse de la machine reste acceptable. Ils ne se manifestent que dans des cas extêmes où ils ressentent une lourdeur lors de la manipulation du clavier ou de la souris. D'autres utilisateurs se réservent l'exclusivité de l'utilisation de leur machine même si cette dernière n'est pas pleinement exploitée. En effet, ces utilisateurs considèrent que toute dégradation des performances est essentiellement liée à l'existence de processus des utilisateurs distants même si ces processus sont tout le temps dormants, ou consommant très peu de ressources. Toutes ces contraintes doivent être prises en compte pour une meilleure gestion des ressources disponibles.

1.2.2 Indicateurs de charge

Pour pouvoir analyser la charge d'un système, on a souvent recours à des indicateurs de charge. Ce sont en quelque sorte des capteurs qui permettent de renseigner sur

le comportement et l'utilisation de chaque composant d'une machine (CPU, mémoire, système d'E/S). Différents indicateurs de charge ont été exploités par les ordonnanceurs (tableau 1.1). La plupart des systèmes présentés dans le tableau 1.1 se basent sur le taux d'utilisation CPU ou sur la charge moyenne pour déterminer si une machine est candidate à accueillir des processus (DAWGS [CM92], LSF [Cor94], RES [Car92], Sprite [OCD+88], Piranha [GK91], GATOSTAR [Fol92]). L'activité du clavier et de la souris ainsi que le nombre de sessions distantes permettent de contrôler l'utilisation interactive d'une machine (Butler [Nic87] et Godzilla [Cra90] entre autres). Des environnements du domaine commercial tel que LSF [Cor94] utilisent plusieurs indicateurs de charge (espace mémoire, activité disque, performance réseau, etc.). Une connaissance préalable de ces indicateurs permet de mieux les interpréter et les utiliser. Une liste non exhaustive est présentée ci-dessous :

- **charge moyenne** : la charge moyenne permet de résumer l'activité d'une machine sur une période de temps. UNIX définit la charge moyenne comme étant le nombre moyen de processus à l'état prêt durant un intervalle de temps. Un processus est considéré à l'état prêt si :
  - il n'est pas en attente d'un événement externe (e.g. appui d'une touche).
  - il n'est pas en attente par lui-même (e.g. appel de `wait`).
  - il n'est pas stoppé (e.g. par CTRL-Z).

  Sous le système BSD, le calcul de la charge se fait par des échantillonnages de 5 secondes lissés dans le temps.

  Quoique la charge moyenne est pratique, elle peut ne pas fournir une image précise de la charge du système pour deux raisons principales :

  - les processus en attente d'une E/S disque y compris ceux utilisant NFS sont considérés comme étant à l'état prêt et figurent donc dans le calcul de la charge moyenne. Si un serveur NFS ne répond plus suite à une défaillance du réseau ou à un crash du serveur, un processus peut attendre longtemps la fin d'une opération NFS. Il est considéré à l'état prêt pendant toute cette période. Ainsi, la charge moyenne grimpe alors que le système n'est pas en train de faire du travail utile.
  - la charge moyenne ne tient pas compte de la priorité des processus. Elle n'effectue aucune distinction entre les processus s'exécutant à une faible priorité (commande `nice`) et donc ne consommant pas beaucoup de temps CPU, et les processus ayant une priorité forte.

- **utilisation CPU** : c'est le pourcentage du temps CPU utilisé. Deux visions sont possibles : utilisation CPU globale sur une machine ou ponctuelle pour chaque processus.

ordonnanceur	utilisation CPU, charge moyenne	utilisateurs connectés	activité clavier/souris	autre
Butler [Nic87]		X	X	créneaux horaires
DAWGS [CM92]	X	X		
Godzilla [Cra90]			X	
LSF [Cor94]	X	X	X	espace mémoire, zone de swap, pagination, transfert disque, espace /tmp, performance NFS
RES [Car92]	X		X	
Sprite [OCD+88]	X		X	
Piranha [GK91]	X	X	X	créneaux horaires
GATOSTAR [Fol92]	X			espace mémoire, localisation des fichiers, besoins en communication
Radio [BSS91]	X			

Tableau 1.1 : Indicateurs de charge utilisés dans certains environnements.

- **espace mémoire** : c'est le pourcentage de l'espace mémoire physique utilisé. Là encore deux visions sont possibles : utilisation mémoire globale sur une machine ou ponctuelle pour chaque processus.
- **pagination** : c'est le nombre de pages-ins (transferts de pages du disque vers la mémoire) et de pages-outs (transferts de pages de la mémoire vers le disque). Les défauts de page montrent une insuffisance de la mémoire physique par rapport à la demande des processus en cours.
- **activité du swap** : c'est le nombre de swaps-ins (transferts de processus du disque vers la mémoire) et de swaps-outs (transferts de processus de la mémoire vers le disque). Ceci montre si la mémoire physique est extrêmement insuffisante. C'est un cas limite de la pagination.
- **activité disque** : c'est le nombre de transferts disque par unité de temps ainsi que les temps d'accès. Notons que les problèmes de mémoire physique et les problèmes d'E/S disque sont intrinsèquement liés.
- **performance NFS et réseau** : c'est le nombre de retransmissions, collisions, time outs, etc. Ceci montre si le réseau est surchargé ou défaillant ainsi que l'état du serveur de fichiers (saturé, défaillant).
- **nombre d'utilisateurs connectés** : on distingue les utilisateurs présents physiquement en face de leurs machines et les utilisateurs distants (connectés

par `rlogin` et `telnet` par exemple). L'utilisation peut être interactive où la notion de propriété de la machine[3] apparaît ou en traitement par lots. La présence du propriétaire est détectée en contrôlant l'activité du clavier et de la souris ainsi que les sessions distantes.

- **espace /tmp** : conditionne souvent l'exécution de nouveaux processus. Une gêne est occasionnée lorsque l'espace /tmp devient saturé.

1.3 Commandes et outils permettant le calcul de la charge

UNIX fournit un ensemble de commandes permettant de fournir la plupart des indicateurs de charge sus-cités. On distingue les commandes à usage général et les commandes spécifiques à une ressource particulière. D'autres utilitaires non-standards ont été développés dans le but de contrôler la charge d'une ou de plusieurs machines.

1.3.1 Commandes système

- **uptime** : fournit entre autres le temps écoulé depuis la dernière mise en service du système, le nombre d'utilisateurs actuellement connectés et la charge moyenne du système durant la dernière minute, les 5 dernières minutes et les 15 dernières minutes. La charge moyenne permet de savoir si la charge du système augmente/diminue pendant le dernier quart d'heure.

  Les mesures données par `uptime` sont souvent utiles quoique grossières (par exemple le nombre d'utilisateurs connectés peut inclure plusieurs fois le même utilisateur). `uptime` est disponible sous le système BSD.

- **ruptime** : il s'agit d'une commande permettant l'obtention d'informations, équivalentes à celles données par `uptime`, de toutes les machines du réseau local. Les informations sont diffusées par chaque machine distante toutes les minutes. `ruptime` permet aussi de détecter les machines défaillantes (celles qui ne répondent plus au bout de 5 minutes[4]) et de spécifier depuis combien de temps elles le sont. Les utilisateurs ayant des sessions inactives depuis plus d'une heure peuvent être exclus du calcul. Toutefois, les informations données par la commande sont imprécises car elles sont extraites d'une base de données locale.

- **rup** : fonctionne exactement comme la commande `uptime` mais sur une ou toutes les machines du réseau local. Les informations sont puisées directement du noyau via un démon appelé `rpc.rstatd`. La présence de ce démon conditionne

[3] Cette notion de propriété ne signifie pas forcément propriété matérielle mais surtout propriété de l'aspect interactif de la machine.

[4] Cette valeur est donnée à titre indicatif. Elle peut être différente pour une autre version d'UNIX BSD tournant sur une architecture différente (par exemple, 11 minutes sous LINUX).

l'exécution de la commande. Contrairement à la commande ruptime, les résultats fournis par rup sont toujours fiables.

- **ps** : pour avoir des informations précises que la charge moyenne ne peut fournir, on a généralement recours à la commande ps. Les résultats de cette commande permettent de distinguer la surcharge dûe aux problèmes de mémoire virtuelle, d'E/S ou autres. Les résultats permettent aussi de préciser quels processus appartenant à quels utilisateurs sont responsables d'une éventuelle surcharge du système. Ci dessous les informations les plus importantes renvoyées par la commande[5] :

  - **%CPU** : pourcentage CPU utilisé par le processus. Pour des raisons d'imprécisions, la somme des pourcentages CPU peut ne pas être égale à 100.
  - **%MEM** : pourcentage de mémoire physique utilisée par le processus. Là aussi, la somme peut être différente de 100. Si le système est en train de paginer, cette mesure peut montrer quels sont les processus responsables.
  - **RSS** : quantité de mémoire physique, exprimée en kilo octets, actuellement allouée au processus (*Resident Set Size*).
  - **STAT** : état du processus.
  - **TIME** : temps total CPU consommé par le processus.

  La page de référence de la commande ps avertit toujours que les résultats de la commande ne sont pas toujours valides au moment de leur lecture. Cependant, un problème important concernant une distortion des informations données par la commande n'a pas été mentionné. Cette distortion vient du fait que le processus qui exécute la commande ps figure dans la liste des processus. Ce processus consomme au moins 25 à 50% du temps CPU plus une portion de la mémoire. Cette distortion altère la cohérence des informations recueillies, de plus il n'y a aucun moyen d'y remédier.

  Les informations renvoyées par la commande ps sont très détaillées. On a parfois besoin d'informations globales concernant tout le système (par exemple, le pourcentage d'utilisation CPU) et non pas d'informations relatives à chaque processus. Pour cela, on peut utiliser des commandes telles que `vmstst`, `iostat` et `sar`. La commande ps est disponible également sous le système V et diffère dans les options et les formats de sortie.

- **vmstat** : permet de reporter des statistiques concernant la mémoire virtuelle essentiellement mais aussi des statistiques sur les processus, l'activité des disques et l'utilisation CPU. `vmstat` est le meilleur moyen de déterminer s'il y a des pages-outs et donc une insuffisance mémoire. Elle permet d'afficher instantanément ou

[5] sous UNIX BSD

périodiquement les informations ainsi que des moyennes depuis le dernier reboot. En général, ces moyennes ne sont pas fiables.

Les informations les plus importantes sont celles qui donnent les pages-outs et les swaps-outs. Ceci montre si le système est en train de paginer ou de swapper. Dans la plupart des systèmes, la pagination et le swapping commencent lorsque l'espace mémoire résiduel est en deçà des seuils fixés par le système en fonction de la taille mémoire physique de la machine.

- **iostat** : en plus des statistiques globales sur les E/S, la commande `iostat` fournit des informations sur l'activité des terminaux et l'utilisation CPU. Les données sont affichées périodiquement ainsi qu'une moyenne depuis le dernier lancement du système (la moyenne est non fiable sur la plupart des systèmes). Cette commande existe sous le système BSD seulement.

- **netstat** : cette commande reflète l'état du réseau. Elle permet d'afficher le contenu de plusieurs structures de données liées au réseau. Une première forme de la commande permet d'afficher la liste des sockets actives pour chaque protocole. La deuxième forme sélectionne une parmi la variété de structures de données du réseau. Les formes restantes affichent les tables de routage. Cependant, les informations les plus importantes sont les suivantes : l'état des interfaces utilisées pour le trafic TCP/IP et l'état des connexions (sockets) actives.

- **nfsstat** : fournit des statistiques concernant l'utilisation de NFS et des RPCs du côté client et/ou serveur. Contrairement à la commande `netstat`, la commande `nfsstat` fournit les informations depuis le dernier lancement du système ou depuis la dernière mise à zéro des compteurs. Ceci permet d'avoir des données pendant des durées relativement courtes ou des données récentes.

- **sar** : fournit des informations sur l'activité du système. Elle est disponible sous le système V. Les plus importantes informations données sont les suivantes :
  - utilisation CPU.
  - longueur moyenne de la file des processus prêts et le pourcentage d'occupation.
  - activité et statistiques d'utilisation des disques.
  - allocation mémoire.
  - statistiques sur la pagination.
  - pages mémoire et blocs disque inutilisés.
  - statistiques sur le swapping et les changements de contexte.
  - activité des buffers et des caches du système.
  - activité du serveur RFS.

- activité des terminaux.
- appels système.
- états des tables du système.

- **sa** : génère un rapport d'utilisation montrant les commandes qui ont été exécutées sur la machine, leur fréquence ainsi que les ressources utilisées. Les informations concernant l'exécution de chaque processus sont d'abord placées dans un fichier une fois que le mécanisme de création de rapport est déclenché par la commande `accton`. Des informations résumées et condensées seront générées par la suite grâce à la commande **sa**. **sa** est également capable de générer des informations par utilisateur plutôt que par processus.

  Cette commande est très utile si l'on veut étudier le comportement des utilisateurs et la façon dont leurs machines sont exploitées. Elle nous informe sur le type d'applications lancées par les utilisateurs (commandes UNIX, utilitaires, etc.), le temps pris par le système pour les exécuter (temps réel et temps CPU en minutes), la quantité moyenne de mémoire physique utilisée, (en kilo octets), le nombre moyen d'opérations d'E/S effectuées, les utilisateurs exploitant fortement le système, etc.

  La commande **sa** possède plusieurs options, les plus importantes permettent de montrer :

  - les processus limités par la CPU.
  - les processus limités par les E/S.
  - les processus gourmands en mémoire.
  - les processus fréquemment lancés.
  - les processus fréquemment dormants.

  La commande **sa** existe sous BSD uniquement. Un ensemble de programmes shell (`runacct`, `prdaily`, `dodisk`, etc.) est disponible sous le système V et permet de génèrer un ensemble de rapports d'utilisation semblables à ceux générés par la commande **sa**.

Souvent ces outils ne répondent pas entièrement aux besoins des utilisateurs ou ne sont pas utilisés car ils induisent eux-même une surcharge qui parasite les résultats recherchés. C'est pourquoi certains outils ont été développés.

1.3.2 Autres utilitaires

- **perfmeter** : développé par Sun, il génère un rapport visuel décrivant une multitude de statistiques. Cet utilitaire peut être utilisé en local ou à distance. Il utilise le démon `rpc.rstatd`. Les statistiques sont données sous forme graphique ou icônifée et sont actualisées périodiquement. Les statistiques receuillies sont les suivantes :

- taux d'utilisation CPU.
- paquets Ethernet émis par seconde.
- activité de la pagination en pages par seconde.
- processus swappés par seconde.
- nombre d'interruptions par seconde.
- activité disque en nombre de transferts par seconde.
- nombre de changements de contexte par seconde.
- nombre moyen de processus à l'état prêt durant la dernière minute.
- nombre de collisions Ethernet par seconde.
- nombre d'erreurs par seconde sur les paquets Ethernet reçus.

Les machines distantes peuvent être d'une architecture différente de Sun pourvu que le démon `rpc.rstatd` existe.

Malgré les différents paramètres contrôlés par l'outil `perfmeter`, celui-ci possède certaines limitations. Les informations concernant le swapping et la pagination sont brutes, elles regroupent à la fois les transferts dans les deux sens de et vers la mémoire alors que seules les pages-outs et les swaps-outs sont importants pour déterminer s'il y a une insuffisance mémoire.

- **Top** : c'est un programme qui permet d'afficher périodiquement les informations suivantes[6] :
  - informations données par la commande `uptime`.
  - nombre de processus dormants, stoppés, zombies, à l'état prêt ainsi que le nombre total de processus.
  - pourcentages de temps CPU dans le mode utilisateur, système, priorité réduite, et oisif.
  - statistiques sur l'utilisation de la mémoire (espace mémoire total, mémoire disponible, mémoire utilisée, mémoire partageable, mémoire utilisée par les buffers).
  - statistiques sur la zone de swap (espace swap total, swap disponible et swap utilisé).
  - informations données par la commande `ps`.

  Ces informations sont en général non fiables durant le premier intervalle de temps à cause de l'exécution de la commande `top` elle même. Elles deviennent fiables à partir du deuxième intervalle car la commande `top` puise les informations dans des fichiers reflétant l'image mémoire du noyau. Ainsi, la commande `top` doit être

[6] Top est disponible par ftp anonyme sur la machine eecs.nwu.edu dans le répertoire /pub/top.

installée de la même manière que la commande ps l'est, *i.e. setuid root.* Enfin, la commande top est portable sur un grand nombre de machines mais pas sur toutes les architectures.

- **Network Analyser** : contrairement à tous les outils présentés précédemment où la collecte d'informations se fait sur une seule machine, le système Network Analyser [MG95] a été conçu dans le but de connaître l'état d'un réseau de machines afin d'évaluer les ressources mises à la disposition des utilisateurs. Parmi ses objectifs, l'extensibilité (gestion d'un grand parc de machines), la généricité (plusieurs types d'applications parallèles LANDA [Mon96] ou PVM [BDG+93]), la non-intrusion, la robustesse, et la convivialité (interface graphique). Le Network Analyser offre plusieurs services accessibles via une bibliothèque de primitives :

  - service de collecte d'informations.
  - service de sauvegarde des informations collectées.
  - service de diffusion des caractéristiques instantanées.
  - service de placement.
  - service de suivi des processus et des utilisateurs.
  - service d'observation du réseau.

  Le Network Analyser puise les différentes informations directement dans la zone de données du noyau. Ceci permet l'obtention d'une diversité d'informations fiables au détriment de la portabilité.

- **Analyseur de charge de travail** : cet outil a été mis en œuvre pour des applications distribuées faisant appel à des opérations d'équilibrage de charge [Dow95]. Il s'agit en fait d'une couche intermédiaire entre le niveau applications et le niveau système. L'analyseur se présente sous forme de serveurs lancés sur toutes les machines d'un réseau. Chaque serveur est chargé de la collecte des informations liées à l'état de charge locale de la machine (nombre d'utilisateurs actifs, taux d'utilisation CPU, charge moyenne, statistiques sur l'utilisation du réseau, etc.). Selon les requêtes des applications, l'analyseur de charge est capable de fournir tous ou une partie des indicateurs de charge renvoyés par le démon `rpc.rstatd` (voir plus loin dans ce chapitre). Cependant, les applications distribuées doivent scruter tous les serveurs à la recherche de la machine la moins chargée.

  L'avantage de cet outil est qu'il est à la fois indépendant des applications et du système d'exploitation. Ceci lui confère une utilisation générique et portable.

1.3.3 Analyse critique

La plupart des commandes et outils système décrits précédemment ont été développés à l'origine pour l'administration du système. Leur but est surtout de fournir un moyen pour :

- observer et contrôler le fonctionnement d'un système.
- régler les problèmes de dégradation des performances.
- détecter et localiser les pannes.
- faire évoluer le système (changer la configuration et la topologie du réseau, changer la configuration des machines, du système d'exploitation, etc.).

Le fait d'exploiter ces outils pour des besoins d'ordonnancement dans des environnements de programmation parallèle pose un certain nombre de problèmes :

- **problème de fiabilité** : les informations recueillies sont parfois non fiables.
- **problème de portabilité** : les commandes ne sont pas disponibles sous tous les systèmes ou n'ont pas toutes la même syntaxe et les mêmes formats de sortie.
- **problème d'intrusion** : l'utilisation de commandes nécessite d'une part la création d'un processus pour les exécuter, et d'autre part une redirection des sorties vers un fichier ou la création d'un tube (commande popen) pour être exploitées. Ceci engendre un surcôut qui en général parasite les informations de charge.

Pour pallier au problème de fiabilité et d'intrusion, on peut penser à récupérer les informations de charge directement du noyau. En effet, toutes les données concernant l'activité d'une machine sont stockées dans la mémoire du noyau et accessibles via des fichiers tels que `/dev/kmem`, `/dev/mem` et `/vmunix`. En général, l'accès à ces fichiers est protégé ou réservé aux utilisateurs privilégiés (groupe *root*, groupe *kmem*), ce qui pose un problème de sécurité. De plus, le format des variables et structures de données n'est pas le même pour toutes les architectures ce qui pose de nouveau le problème de portabilité.

Après des investigations multiples, le meilleur compromis que nous avons trouvé pour garder la portabilité tout en assurant la fiabilité et la non intrusion, est de passer par le démon `rpc.rstatd`. Ce démon n'est autre qu'un serveur de statistiques du noyau. Il est en principe lancé par un super démon appelé `inetd` exécuté automatiquement lors du lancement du système. Le démon `rpc.rstatd` utilise le mécanisme d'appel de procédure à distance (RPC) pour communiquer les statistiques aux processus clients. Les informations renvoyées par ce serveur sont :

- pourcentages de temps CPU dans le mode utilisateur, système, priorité réduite, et oisif.
- taux de transfert pour chaque disque.
- nombre de pages-ins et pages-outs.

- nombre de swaps-ins et swaps-outs.
- nombre d'interruptions.
- nombre de paquets en entrée et en sortie.
- nombre d'erreurs en entrée et en sortie.
- nombre de collisions.
- nombre de changements de contexte.
- charge moyenne durant la dernière minute, 5 dernières minutes et 15 dernières minutes.
- temps écoulé depuis le dernier reboot.

Cette panoplie d'informations renvoyées par le démon `rpc.rstatd` recouvre pratiquement tous les facteurs impliqués dans le calcul de charge. Une étude expérimentale de la disponibilité des machines basée sur le choix de certains de ces indicateurs est présentée en section 1.6.

1.4 Critères de disponibilité de ressources

Dans la plupart des ordonnanceurs prenant en compte la charge du système, l'allocation d'une machine à une application est basée sur la disponibilité de la machine. La disponibilité est fortement liée aux indicateurs de charge utilisés et au comportement de l'ensemble des utilisateurs. Les critères servant à quantifier la disponibilité d'un nœud doivent être sélectionnés pour être non intrusifs et relativement stables dans le temps. Ils peuvent être pris par défaut par le système ou établis par les propriétaires des stations de travail.

1.4.1 Critères pris par défaut

Le choix des indicateurs de charge et des valeurs associées varie en fonction de l'environnement en question et parfois du type d'applications supportées. Ci-dessous, nous présentons les différents critères fixés par défaut par certains environnements :

- *Sprite* : le système Sprite [OCD+88] déclare un nœud disponible si l'utilisation CPU est au dessous de 1% et le clavier est inactif pendant 30 secondes.
- *Piranha* : le système Piranha [GK91] requiert une inactivité de 5 minutes pour le clavier, la souris et les connexions distantes, et la charge moyenne doit être inférieure à 0.4, 0.3 et 0.1 pendant les 1, 5 et 10 dernières minutes respectivement.

- *Condor* : le système Condor [LLM88] détecte la disponibilité d'un nœud si l'utilisation CPU est au dessous de 0.25%. Les auteurs ont noté qu'il fallait 7 minutes pour que la charge redescende à cette valeur une fois que l'activité du nœud cesse.

1.4.2 Critères établis par les propriétaires des stations

Pour plus de flexibilité, certains environnements privilégient les utilisateurs interactifs en leurs fournissant un moyen pour intervenir dans le choix des indicateurs de charge ou des seuils de charge pour telle ou telle machine. Nous citons à titre d'exemples de critères :

- Si la charge est faible alors autorisation du time-sharing.
- Si le nombre de logins actifs dépasse un certain seuil alors exclure l'exécution de processus distants.
- Exclure l'exécution de processus distants à des intervalles de temps précis pendant les jours de semaine (par exemple, entre 14h00 et 17h00).
- Supprimer un nœud temporairement du pool des nœuds.

1.5 Quantification de la puissance de calcul disponible dans un méta-système

Une fois les indicateurs de charge déterminés, la quantification de la puissance de calcul disponible devient possible. En se basant sur les critères de disponibilité utilisés par le système Sprite, les auteurs ont reporté 66% à 78% de temps machine perdu. Pour Condor, la sous-utilisation des ressources varie de 70% à 80% durant les week-ends et est estimée à 50% pendant les heures de travail. Dans le V-système [TLC85], les auteurs ont détecté 80% de machines disponibles même durant les heures de travail. Cette différence de mesure avec celle de Condor s'explique par le fait que dans le V-système, les nœuds faiblement chargés peuvent être utilisés en time-sharing. Nichols [Nic87] a reporté que 50 à 70 stations de travail parmi 350 étaient sous-utilisées. L'auteur n'a pas révélé les critères utilisés pour savoir pourquoi le nombre de nœuds inutilisés est si faible. Avec les indicateurs de charge utilisés par le système Piranha, les auteurs ont constaté que les machines sont sous-utilisées à 82% pendant les jours de semaine et à 96% durant les nuits des week-ends.

1.6 Etude expérimentale de la disponibilité des machines

Cette section tente d'évaluer la puissance d'un méta-système à travers l'aggrégat de temps perdu de toutes les machines qui le composent. Le méta-système en question se trouve dans le campus de l'université de Lille (figure 1.2). Il est composé d'une ferme de processeurs ALPHA et de deux réseaux locaux reliés entre eux :

- **ferme d'ALPHA** : il s'agit d'un cluster de 16 processeurs DEC-ALPHA disponible au Centre de Ressources Informatiques (CRI). Les processeurs ont une fréquence de 133 Mhz, et sont reliés entre eux par un giga-switch de fibres optiques (FDDI) et au monde extérieur par Ethernet. Le système d'exploitation supporté est OSF/1. La ferme de processeurs est dédiée aux applications séquentielles et parallèles très gourmandes en ressources de calcul.

- **réseau enseignement** : il est composé de 46 stations de travail de type Sun/Sparc reliés par Ethernet et dotés du système Solaris 2.5. Ce réseau est dédié aux étudiants pour faire des travaux pratiques. En général, il est utilisé en interactif durant la journée et demeure inutilisé durant les nuits, les week-ends et les périodes de vacances.

- **réseau recherche** : il est constitué d'une centaine de machines incluant essentiellement des stations de travail (Sun/Sparc sous Solaris, Sun/Sparc sous SunOs, Silicon Graphics sous IRIX) et des PCs sous Linux. Ce réseau est utilisé par les enseignants, les chercheurs et les doctorants (on y trouve les 3 catégories d'utilisateurs décrits en section 1.2.1). Chaque machine est généralement acquise par un seul utilisateur.

Le méta-système sous-jacent est composé d'environ 150 nœuds d'une puissance globale très importante (plusieurs gigaflops). Pour quantifier la puissance potentielle d'un tel méta-système, nous avons effectué des mesures en utilisant un échantillon (pool) d'une cinquantaine de machines. Pour mesurer la charge d'une machine, nous avons utilisé la charge moyenne et l'activité du clavier et de la souris comme indicateurs de charge.

Un nœud est considéré disponible si la charge moyenne est inférieure à 2.0, 1.5 et 1.0 pendant les 1, 5 et 10 dernières minutes respectivement, et que le nœud n'est pas utilisé en mode interactif pendant 3 minutes. Un nœud dont le clavier et/ou la souris sont actifs est considéré comme étant un nœud dans un état de charge maximal. L'état d'un nœud oscille entre disponible (IDLE), réquisitionné (OWNED) et occupé (BUSY) selon que le nœud n'est pas chargé, que la console du nœud est active ou que le nœud est fortement chargé. Deux états supplémentaires ont été définis par rapport aux pannes des machines : SUSPENDED (nœud rebooté) et DELETED (nœud en panne ou définitivement supprimé du pool de nœuds).

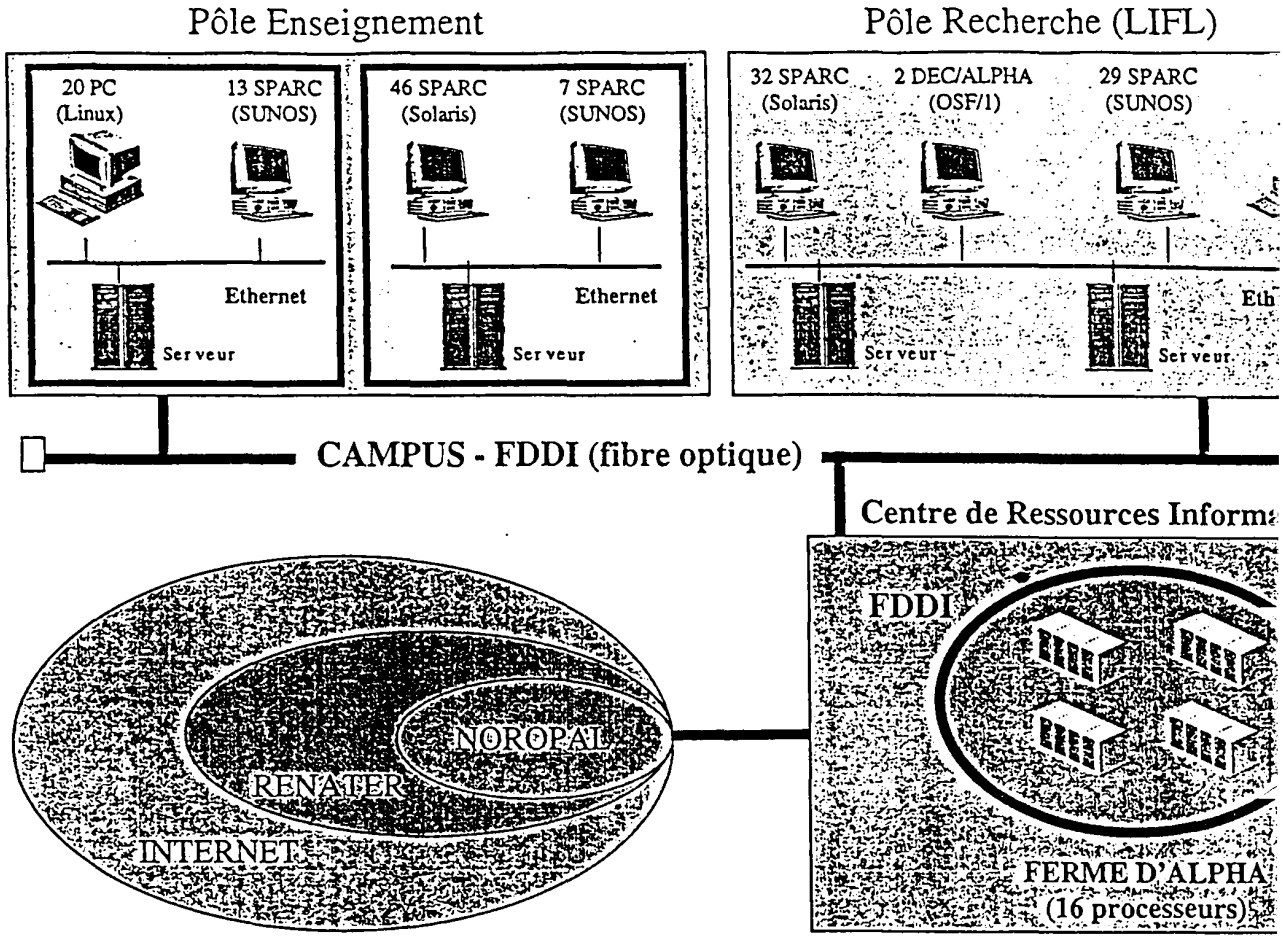


Figure 1.2: Un exemple de méta-système sur le campus de l'université de Lille

1.6.1 Définitions

- La période de scrutation (Scan Period ou SP) est la durée en minutes qui précède chaque balayage des états des nœuds du méta-système.

- La somme globale journalière de tous les états (Total Day States ou TDS) est définie comme suit :

$$TDS = \sum_{i=1}^{1440/SP} \sum_{j=1}^{TN} NS[j]$$

où :

  - TN (Total Nodes) est le nombre total de nœuds qui composent le méta-système.
  - NS (Node State) est l'état d'un nœud. Les différentes possibilités sont IDLE (I), BUSY (B), OWNED (O), SUSPENDED (S) et DELETED (D).

- Le pourcentage de temps où les nœuds sont à un état donné s $\in$ {I, B, O, S, D} est défini comme étant :

$$P_s = \frac{\sum_{i=1, NS[i]=s}^{TN} NS[i]}{TDS}$$

- Une tranche (Burst) est définie par une succession d'un même état. La longueur d'une tranche en minutes (Burst Length ou BL) est le produit du nombre d'états qui la composent par SP.

- La moyenne des longueurs de tranches (Average Burst Length ou ABL_s) pour un état s $\in$ {I, B, O, S, D} donné est définie par :

$$ABL_s = \frac{\sum_{i=1}^{nombre\ de\ tranches} BL[i]}{nombre\ de\ tranches}$$

1.6.2 Expérimentation et analyse des résultats

Pour illustrer l'utilisation des nœuds au sein du laboratoire sur une période de trois jours (un jour de semaine et le week-end) et sur un pool de 51 machines, nous avons tracé les courbes représentant respectivement le nombre de nœuds IDLE, BUSY et OWNED (SP = 1mn) machine par machine (figures 1.3, 1.4 et 1.5) et globalement pour toutes les machines (figures 1.6 et 1.7). Il apparaît d'après la figure 1.3 que les machines sont le plus souvent à l'état IDLE surtout en dehors des heures de travail. La figure 1.5 montre que les machines sont relativement utilisées en interactif dans une période située entre 8h00 et 19h00, et que même durant cette période, les machines sont sous-exploitées. Dans la figure 1.6, on peut remarquer que la courbe des nœuds à l'état OWNED présente deux pics qui correspondent à l'utilisation des machines durant la matinée et l'après-midi. Cette courbe a tendance à se lisser pendant le week-end (figure 1.7) à l'avantage des nœuds à l'état IDLE. La figure 1.6 montre aussi qu'à tout

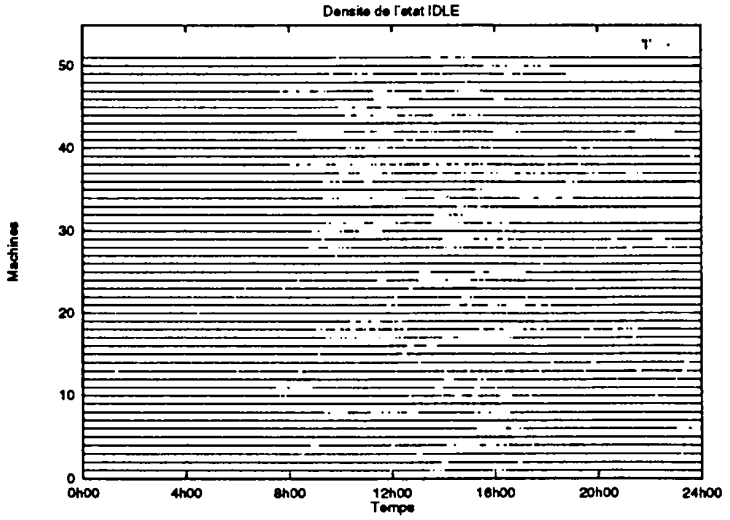


Figure 1.3 : Temps perdu machine par machine (état IDLE)

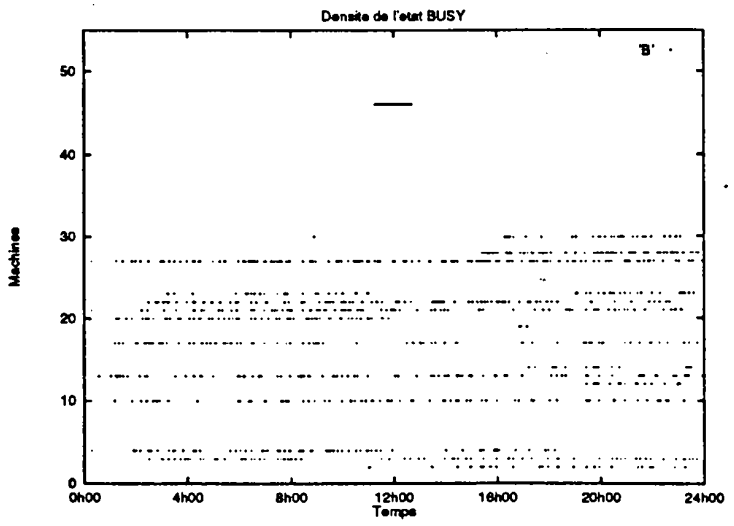


Figure 1.4 : Surcharge machine par machine (état BUSY)

moment le nombre d'utilisateurs travaillant en interactif n'excède pas 17 et au moins 25 machines sur les 51 observées restent libres.

Les tableaux 1.2 et 1.3 montrent des statistiques et des informations globales concernant l'utilisation des machines au LIFL (TDS = 73440). Le tableau 1.2 montre que les

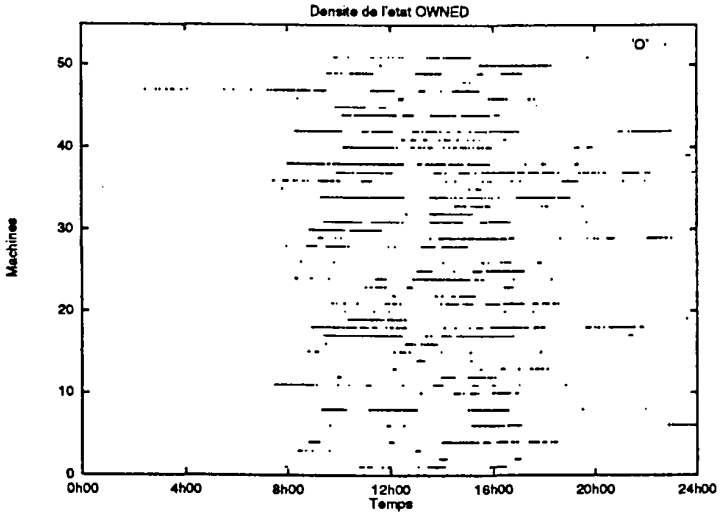


Figure 1.5 : Utilisation interactive machine par machine (état OWNED)

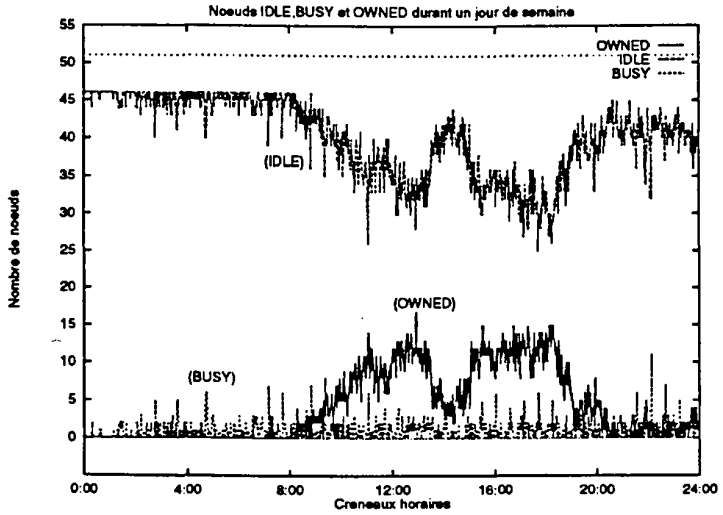


Figure 1.6 : Utilisation des machines durant un jour de semaine (Vendredi)

machines sont à l'état IDLE pendant au moins 85% du temps contre 2.7% et 2.1% pour les états OWNED et BUSY respectivement. La longueur d'une tranche de temps où une machine est à l'état IDLE varie d'une minute à une journée entière pour une moyenne de 41 minutes durant un jour de semaine. Les utilisateurs travaillent en interactif de façon continue entre 1 minute et 2 heures et en moyenne durant 8 minutes. L'écart type

est très important, et est dû au fait que les utilisateurs n'utilisent pas leurs machines de manière régulière et homogène.

Le nombre de machines chargées (à l'état BUSY) est pratiquement insignifiant. Par contre le nombre de machines redémarrées ou en panne est parfois important (10 % ce qui correspond à 5 machines).

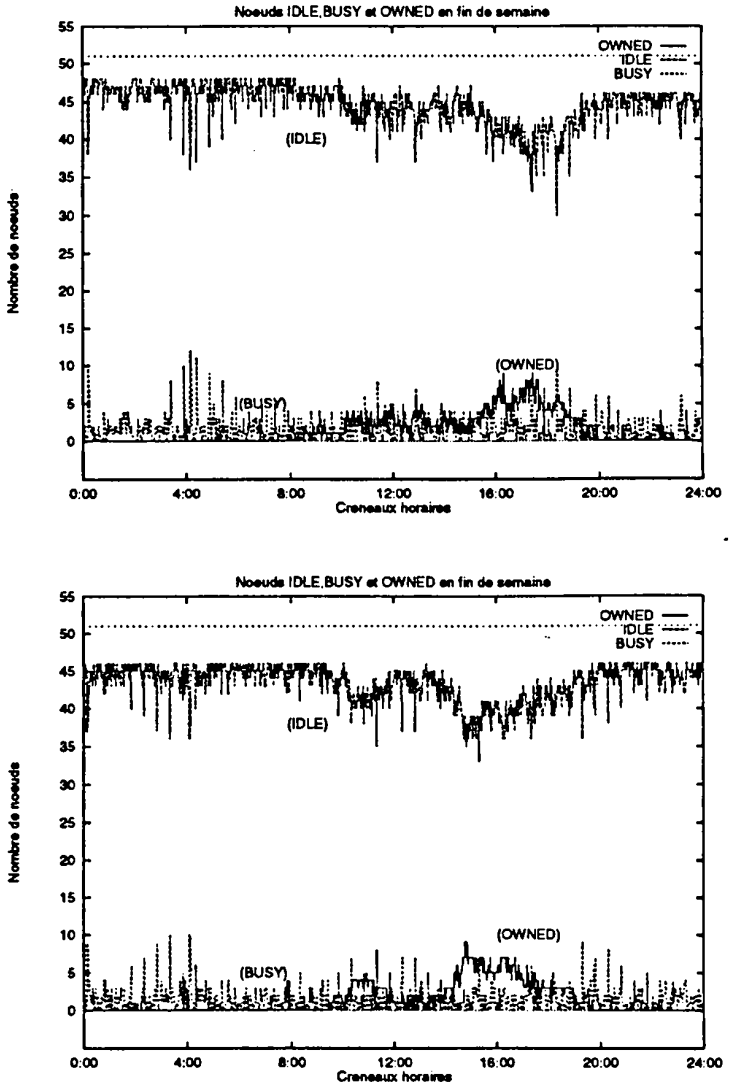


Figure 1.7: Utilisation des machines durant un week-end (Samedi et Dimanche)

Date		IDLE	BUSY	OWNED
Vendredi (04/10/96)	$P_{I,BetO}$	87.5 %	2.1 %	8.6 %
	Nombre de tranches	1549	822	726
	$ABL_{I,BetO}$	41	1	8
	Déviation standard	5156.39	83.99	362.85
	Min BL	1 (1h19-1h20)	1 (0h17-0h18)	1 (2h28-2h29)
	Max BL	1440 (0h00-0h00)	64 (11h17-12h21)	111 (8h18-10h09)
Samedi (05/10/96)	$P_{I,BetO}$	87.4 %	2.8 %	2.8 %
	Nombre de tranches	1216	983	201
	$ABL_{I,BetO}$	52	1	2
	Déviation standard	6710.79	62.31	226.93
	Min BL	1 (0h37-0h38	1 (0h00-0h01)	1 (8h52-8h53)
	Max BL	1440 (0h00-0h00)	27 (1h12-1h39)	151 (15h57-18h28)
Dimanche (06/10/96)	$P_{I,BetO}$	85 %	2.5 %	2.7 %
	Nombre de tranches	1178	993	145
	$ABL_{I,BetO}$	52	1	13
	Déviation standard	7211.87	58.75	197.15
	Min BL	1 (0h03-0h04)	1 (0h02-0h03)	1 (10h54-10h55)
	Max BL	1440 (0h00-0h00)	15 (12h58-13h13)	88 (14h59-16h27)

Tableau 1.2 : Statistiques concernant l'utilisation des machines au LIFL

Date		SUSPENDED	DELETED
Vendredi (04/10/96)	P_{SetD}	0.1 %	1.8 %
	Nombre de tranches	5	3
	ABL_{SetD}	8	437
	Déviation standard	9.06	165.54
	Min BL	3 (15h43-15h46)	303 (18h57-0h00)
	Max BL	12 (15h13-15h25)	515 (15h25-0h00)
Samedi (05/10/96)	P_{SetD}	0 %	6.9 %
	Nombre de tranches	2	5
	ABL_{SetD}	9	1019
	Déviation standard	1	1151.9
	Min BL	9 (16h56-17h05)	363 (17h57-0h00)
	Max BL	10 (17h47-17h57)	1440 (0h0-0h00)
Dimanche (06/10/96)	P_{SetD}	0 %	9.8 %
	Nombre de tranches	2	5
	ABL_{SetD}	3	1440
	Déviation standard	0	0
	Min BL	3 (14h53-14h56)	1440 (0h00-0h01)
	Max BL	3 (14h53-14h56)	1440 (0h00-0h01)

Tableau 1.3 : Statistiques concernant les pannes des machines au LIFL

1.7 Conclusion

Dans ce chapitre, nous avons illustré la disponibilité des machines dans un méta-système. La puissance potentielle obtenue à partir de l'aggrégat du temps perdu est très importante et peut être récupérée et mise à la disposition des applications parallèles nécessitant beaucoup de ressources de calcul.

Il est clair que la quantification de la puissance d'un méta-système est directement liée à la sélection et l'interprétation des critères de charge. Cependant, quelque soient les indicateurs choisis, les résultats montrent toujours une forte sous-utilisation des machines.

Plusieurs environnements ont été développés pour pouvoir exploiter le temps machine inutilisé dans un méta-système. Dans le chapitre suivant, nous allons présenter une synthèse des environnements d'exécution et particulièrement ceux qui permettent d'exploiter le temps perdu.

Chapitre 2

Environnements d'exécution : Etat de l'art

Dans ce chapitre, nous présentons une synthèse des environnements d'exécution dont la problématique omniprésente est l'ordonnancement d'applications en vue de minimiser leurs temps d'exécution et de mieux exploiter les ressources mises à disposition. Contrairement aux applications séquentielles, les applications parallèles compliquent la tâche de l'ordonnanceur car d'autres facteurs interviennent tels que les communications, la synchronisation et la terminaison. Notre attention est portée sur les environnements d'exécution parallèle et plus précisement ceux qui permettent d'adapter le degré de parallélisme des applications en fonction de l'état de charge d'un méta-système. Nous commençons par une classification des environnements d'exécution séquentielle et parallèle, puis nous présentons de façon plus détaillée les environnements dits adaptatifs qui sont au cœur de cette thèse.

2.1 Taxonomie des environnements d'exécution

L'exploitation rationnelle des ressources de traitement est un défi perpétuel dans le domaine de la recherche en informatique. En particulier, la prolifération des réseaux de stations de travail puissantes est devenue une source attrayante de puissance de calcul. Les systèmes développés pour exploiter ce type de plate-formes étaient à l'origine focalisés soit sur la gestion des ressources, ou sur la mise en place d'interfaces de telle sorte qu'un certain nombre de machines puisse être utilisé en parallèle. Les deux approches sont intéressantes et peuvent être complémentaires. En effet, des travaux récents tentent de fédérer ces concepts en interfaçant des systèmes déjà existants ou en proposant de nouveaux environnements.

Plusieurs classifications des environnements d'exécution ont été proposées dans la littérature [BNK89] [Tur93] [KN94] [KRR95] [BFY96] [Tal95] [BF96]. Parmi les critères les plus importants permettant l'évaluation des environnements d'exécution, on cite :

- **la portabilité** : s'agit-il d'une extension du système d'exploitation natif (une nouvelle couche) ou une modification de celui-ci (un nouveau noyau) ?
- **l'hétérogénéité** : l'exécution se fait-elle en milieu homogène ou hétérogène ?
- **le parallélisme** : quel est le type d'applications supporté, séquentielles ou parallèles ?
- **la politique d'ordonnancement** : est-elle basée sur des indicateurs de charge ? sur des spécifications par l'utilisateur des ressources requises ? le placement de processus est-il statique ou dynamique ? préemptif ou non-préemptif ? quelles sont les stratégies d'équilibrage de charge utilisées ?
- **la dynamicité** : est-ce que l'environnement est dynamiquement paramétrable et reconfigurable ?
- **l'impact sur les propriétaires des stations de travail** : est-il possible de minimiser l'impact sur les propriétaires des stations de travail ?
- **la tolérance aux pannes** : est-il possible de sauvegarder sur disque l'état des processus en vue de les relancer en cas de panne ? existe-t-il un point de panne unique « Single Point of Failure » ?
- **la migration** : est-il possible de migrer un processus en exécution d'une machine à une autre ?
- **la sécurité** : existe-t-il des problèmes de sécurité qu'il faut prendre en considération ?

A ces critères peuvent s'ajouter d'autres facteurs tels que :

- **l'interactivité** : s'agit-il d'applications en traitement par lots ou interactives ?
- **la configuration matérielle et logicielle requise** : quel type de machine ? quelle version de système d'exploitation ? faut-il l'existence de NFS ? etc.
- **l'appartenance de l'environnement** : relève-t-il du domaine commercial ou académique ?
- **l'existence d'une interface graphique** : existe-t-il une interface graphique permettant entre autres le contrôle et l'observation ?
- **la facilité d'utilisation** : quelles sont les contraintes d'installation et d'utilisation ?

La figure 2.1 illustre la différence entre les environnements d'exécution existants par leur façon d'ordonnancer les applications (séquentielles ou parallèles) sur les différents nœuds d'un méta-système. En effet, dans les environnements d'exécution, le placement de processus est réalisé soit de façon aveugle ou en fonction de l'état de charge des machines et de leur utilisation interactive.

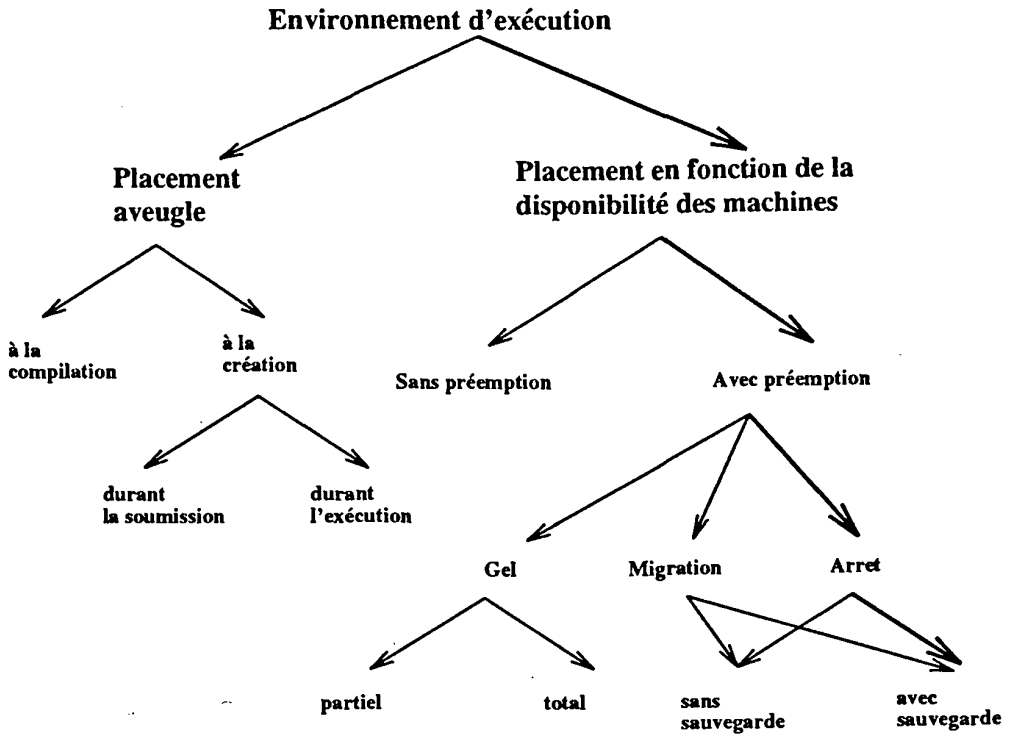


Figure 2.1 : Ordonnancement dans les environnements d'exécution

2.1.1 Placement aveugle

Le placement aveugle ne tient pas compte de la charge des nœuds et se fait de façon locale (UNIX et dérivés, NewCastle [BMR82]) ou distante (Accent [RR81], Locus [Wa83]). Les machines cibles peuvent être déterminées à la compilation ou au moment de la création des processus. Les processus sont créés lors de la soumission de l'application ou au cours de l'exécution de celle-ci. L'allocation des nœuds aux processus demeure inchangée durant l'exécution, et ne tient pas compte de l'état courant du système (Amber [Ca89],

Clouds [DCM+90], Eden [LLA+81]).

Depuis le milieu des années 80, le problème d'utiliser plusieurs machines interconnectées a été abordé de plusieurs manières. La plupart des groupes de recherche se sont limités à l'approche « calcul parallèle intensif » en faisant abstraction de la prospective « gestion de ressources distribuées ». On distingue les environnements à échange de messages, à mémoire virtuellement partagée, ou combinant les deux, et à appel de procédure distante :

- **Environnements à échange de messages** : les environnements à échange de messages ont été parmi les premiers résultats guidés par l'approche calcul parallèle intensif. L'environnement le plus populaire devenu un standard de fait est PVM (Parallel Virtual Machine) [BDG+93]. Les principales caractéristiques de PVM sont la dynamicité de la machine virtuelle qu'il gère, l'hétérogénéité, la détection de pannes, et la simplicité de son interface et de son utilisation. PVM a été largement utilisé pour développer des applications réelles dans différents domaines. Le succès escompté a incité d'autres groupes de recherche à étendre l'environnement pour pouvoir supporter des fonctionnalités permettant d'exploiter au mieux la sous-utilisation des machines (voir section suivante). MPI (Message Passing Interface) constitue aujourd'hui le standard en matière d'environnements de programmation parallèle [For94]. À ce jour, MPI n'est pas aussi largement utilisé que PVM mais constitue une orientation future évidente. PVM et MPI ne sont pas les seuls environnements de programmation parallèle qui ne tiennent pas compte de l'utilisation des machines dans l'ordonnancement de processus, on cite à titre d'exemples : P4 [BL92], Express [FKB91], ISIS [BM89], TCGMSG [Har91], PVC [GHM+95], LANDA [Mon96][1], etc.

- **Environnements à mémoire virtuellement partagée** : contrairement aux environnements de programmation parallèle à échange de messages où la communication est explicite, dans les environnements à mémoire virtuellement partagée un processus envoyant un message n'a à connaître ni l'identité du processus récepteur ni sa localisation. Par conséquent, la programmation est simplifiée au détriment d'une implémentation complexe de l'environnement. En effet, la caractéristique des environnements à mémoire virtuellement partagée est le concept de transparence de localité des données ou objets (Midway [BZS93], DoPVM [HS93]), et la communication anonyme (Linda [CG89]).

  Linda est un modèle de programmation parallèle qui peut être greffé à une variété de langages tel que le langage C. Il est basé sur la communication associative qui se traduit par une sorte de mémoire partagée appelée *tuple space*. Les processus partagent des données en les mettant dans le tuple space sous forme de *tuples* qui peuvent être lus ou modifiés par d'autres processus. Le contrôle de processus

[1] Notons que l'environnement LANDA peut tirer parti du logiciel « Network Analyser » [MG95] permettant le contrôle de la charge d'un réseau de machines.

provient du tuple space : pour faire un traitement concurrent, un processus crée un tuple actif (live tuple) entraînant la création d'un processus. Ce dernier entame le traitement en utilisant les données présentes dans le tuple qui deviendra un tuple passif une fois le traitement terminé. Linda a été incorporé dans des systèmes d'exploitation [Lel90]. Ses dérivés tels que POSYBL [Sch94] et Glenda [Aru94] ont été réalisés sur machines à mémoire partagée et machines à mémoire distribuée.

- **Environnements mixtes** : se sont des environnements permettant la communication à la fois par échange de messages et par mémoire virtuellement partagée (p4-linda [BLL93], Shrimp [BLA+94]).

- **Environnements à appel de procédure à distance** : ces environnements sont basés sur le concept d'appel de procédure à distance (Remote Procedure Call ou RPC) [RPC88b][RPC88a], ou d'appel léger de procédure à distance (Lightweight Remote Procedure Call ou LRPC) en utilisant les threads (Athapascan [CCRG95], PM^2 [Nam96], Nexus [FKOT94]). Le principe consiste à appeler un service distant avec un retour éventuel de résultats. Les appels peuvent être synchrones ou asynchrones.

Malgré la présence d'une stratégie de placement dans les environnements sus-cités, ceux-ci demeurent plus adaptés à des plate-formes dédiées ou mono-utilisateur. Leur utilisation dans un méta-système tel qu'il a été présenté au chapitre 1 pose un certain nombre de problèmes notamment des conflits entre utilisateurs et propriétaires de stations de travail.

2.1.2 Placement en fonction de l'état courant du méta-système

Le placement en fonction de la disponibilité et de l'utilisation interactive des machines peut être préemptif ou non-préemptif. Un ordonnanceur est dit non-préemptif si la décision concernant le placement d'un processus est prise au moment de sa création et ne change pas durant son exécution. Par conséquent, ces systèmes tiennent compte de l'état de charge des machines seulement au moment de la création de processus et non durant l'exécution (Radio [BSS91], Utopia [ZGG90], Amoeba [MvRT+90], CONNECT:Queue [Sof93], HetNOS [BSFG94], NEST [Ezz86] [AE87], Plan 9 [PPTT90], Sidle [JXT93], Spawn [WHH+92], VaxClusters [KLS86], PRM [NR94]).

Dans le placement préemptif, l'allocation d'un nœud à un processus n'est pas figée et est sujette à une nouvelle prise de décision une fois que le nœud deviene chargé ou que le nœud est réclamé par son propriétaire. Plusieurs cas de figures peuvent être envisagés selon l'existence de mécanismes de sauvegarde (checkpointing) [DVCL93] ou de réallocation dynamique de processus pendant leur exécution (migration) [Smi88] :

- **gel de processus** : faute de mécanismes de « checkpointing » et de migration, certains environnements gèlent complètement les processus en suspendant leur

exécution (Task Broker [GT93], Batch [Pre94], QBatch [QBA], DJM [DJM93], NQE [CP95], MDQS [Kin89], DQS [Gre95], PBS [HT95]) ou partiellement en réduisant leur priorité (Stealth [KC91], Process Server [Hag86], GNQS [Her], EASY [LHR95], The PARFORM [CS93]). Ce mécanisme peut gêner l'utilisateur de la machine du fait que les processus détiennent toujours une partie des ressources telle que la mémoire.

- **migration de processus** : d'autres environnements déplacent les processus d'un site à un autre. Cette migration peut être réalisée de façon indirecte via des mécanismes de checkpointing (Condor [LLM88] [LL90] [BL91], DAWGS [CM92], Load Balancer [UT94], LoadLeveler [Sup94], GATOSTAR [Fol92], CODINE [Sof95]) ou directe (Demos/MP [PM83], Sprite [OCD+88] [DO91], Charlotte [ACF86] [AF89], Mach [ABB+86], LSF [Cor94], V [Che88] [TLC85], MPVM [CCK+95]), MMPVM [JCG+95]). Implémenter un mécanisme de migration de processus est une tâche très complexe, coûteuse et parfois limitée à un type de processus[2]. De plus, la migration est généralement limitée aux architectures homogènes.

- **arrêt de processus** : le restant des environnements tuent tout simplement les processus pour libérer les machines. Deux cas de figures peuvent être envisagés suivant la sauvegarde préalable du travail effectué ou non :

  - **arrêt sans sauvegarde** : en l'absence de sauvegarde préalable du travail effectué, des systèmes tels que Far [GMW], Godzilla [Cra90] et RES [Car92] réexécutent depuis le début les processus tués sur d'autres nœuds sous-utilisés. L'inconvénient de cette approche est que l'application peut ne pas évoluer si elle revient souvent au point de départ. Pour pallier à ce problème, certains systèmes arrêtent le processus après une courte période suivant l'envoi d'un signal qui doit être capté et géré par le processus pour sa propre sauvegarde ou migration (Butler [Nic87]). D'autres systèmes se comportent d'une façon assez particulière comme c'est le cas du projet Worm [SH82] considéré comme le précurseur des systèmes mettant en œuvre l'aspect « exploitation du temps perdu ». En effet, un Worm est une collection de segments. Un segment scrute continuellement le réseau à la recherche d'une machine disponible pour se dupliquer et donc d'augmenter son espérance de vie. Ces programmes, inspirés d'un film de science fiction, ont été qualifiés de « programmes vampires » car ils grossissent durant la nuit et disparaissent peu à peu le lendemain lorsque les utilisateurs rejoignent leurs stations de travail. Ce mécanisme a été repris récemment dans Charlotte [BKKW94] d'une façon légèrement différente. Le modèle de programmation est basé sur le principe d'une mémoire partagée et utilise le langage Java avec des classes pour l'expression du parallélisme. La décision d'exploiter un nœud émane explicitement de la part de l'utilisateur de la machine en activant une application à partir d'une liste figurant dans une page web. L'ordonnacement

[2] Par exemple, les processus qui n'utilisent pas les signaux comme c'est le cas avec Condor.

repose sur une approche spéculative (eager scheduling) où un même processus est lancé sur toutes les machines disponibles tant que son exécution n'est pas terminée.

- **arrêt avec sauvegarde** : pour éviter le risque de reprise d'exécution depuis le début ou de duplication inutile de processus, de nouveaux environnements intègrent des mécanismes de sauvegarde du travail déjà accompli par un processus ou un groupe de processus. Cette sauvegarde peut être effectuée sur disque (checkpointing) comme c'est le cas pour CARMI [PL95] et GLUnix [GPR+97] (faisant partie du projet NOW [VGA94]), ou mieux encore au niveau applicatif comme c'est le cas dans Piranha [GK91]. Le checkpointing d'un processus consiste à sauvegarder sur disque son contexte de plus bas niveau (registres, table des signaux, fichiers ouverts, etc.). Cette opération est non seulement dépendante du matériel mais aussi très complexe à mettre en œuvre. Par contre, la sauvegarde au niveau applicatif consiste à garder la trace du travail effectué par l'état courant des données. Cette approche possède l'avantage de ne plus dépendre du matériel mais possède l'inconvénient de dépendre des applications.

2.1.3 Ordonnancement adaptatif

À partir de la figure 2.1, on peut ressortir un critère de classification qui tient compte des caractéristiques principales d'un méta-système en l'occurrence l'hétérogénéité, l'aspect multi-utilisateurs et l'aspect propriété des stations de travail. Ces facteurs font qu'un méta-système possède une charge qui varie dynamiquement de façon irrégulière et imprévisible. De ce fait, le critère retenu pour notre classification est l'*adaptabilité* des applications avec la fluctuation de charge et l'utilisation interactive des machines. Ainsi, les environnements d'exécution peuvent être classés en trois catégories selon que le *nombre* et/ou la *localisation* des tâches dépendent ou non de l'état de charge du méta-système sous-jacent (tableau 2.1). Bien que cette classification s'applique à la fois aux environnements d'exécution séquentielle et parallèle, nous allons plus nous focaliser sur les environnements d'exécution parallèle. Ces derniers sont plus aptes à utiliser le maximum de ressources inutilisées dans un méta-système contrairement aux environnements d'exécution séquentielle où une application utilise au plus une machine à la fois.

	tâches	
	nombre	**localisation**
non-adaptatifs	statique	statique
semi-adaptatifs	statique	dynamique
adaptatifs	dynamique	dynamique

Tableau 2.1 : Classification des ordonnanceurs d'applications séquentielles et parallèles.

Dans les systèmes non-adaptatifs et semi-adaptatifs, le nombre de tâches[3] est fixé en dehors de toute information de charge et d'utilisation interactive des machines. Il est déterminé à la compilation ou lors de l'exécution et reste inchangé. La différence entre les deux catégories réside au niveau de la localisation des tâches.

Dans les systèmes non-adaptatifs, les machines cibles sont déterminées là encore en dehors de toute information de charge et d'utilisation interactive des machines. Les machines sont affectées aux différentes tâches de façon définitive comme s'il s'agit d'une plate-forme dédiée à l'application. On retrouve dans cette catégorie tous les environnements d'exécution faisant du placement aveugle.

La catégorie des systèmes semi-adaptatifs comporte tous les environnements qui tiennent compte de l'état courant du système soit au moment de la création de processus seulement, ou au cours de leur exécution. Dans le premier cas, il s'agit des environnements faisant du placement en fonction de la charge sans préemption. Dans le second cas, on y trouve une partie des environnements faisant appel au placement en fonction de la charge avec préemption. Il s'agit de la préemption réalisée en gelant partiellement ou totalement les processus, ou en les migrant directement ou indirectement via des mécanismes de sauvegarde[4]. La semi-adaptativité vient du fait que d'une part les processus gelés détiennent toujours une partie des ressources, et d'autres part le degré de parallélisme n'est pas lié à la variation de charge dans le système. Lorsque le nombre de tâches composant une application dépasse le nombre de nœuds disponibles, plusieurs tâches doivent partager le même nœud, et lorsqu'il y a plus de nœuds disponibles que de tâches, certains nœuds ne seront pas exploités.

Dans la catégorie des environnements adaptatifs, l'ordonnancement supporte des applications dont l'ensemble des tâches, en nombre et en localisation, change dynamiquement en fonction de la disponibilité des nœuds. La gestion de la disponibilité des nœuds est totalement transparente à l'utilisateur. Ces systèmes sont dits adaptatifs car la localisation des tâches et l'augmentation/réduction de leur nombre sont liées à la variation de charge dans le méta-système. De plus, il n'est plus besoin d'implémenter un mécanisme de migration de processus qui est, par nature, complexe. Tous les environnements faisant appel au placement en fonction de la disponibilité des nœuds avec préemption par arrêt de processus rentrent dans cette catégorie. Cependant, les systèmes où une sauvegarde du travail déjà effectuée précède l'arrêt des processus se distinguent par rapport aux systèmes qui tuent brutalement les processus. Les environnements utilisant un mécanisme de sauvegarde non pas sur disque mais au niveau applicatif bénéficient, par nature, de l'hétérogénéité inhérente à un méta-système.

Un récapitulatif des principales caractéristiques de certains environnements décrits dans

[3] Dans une application parallèle, le nombre de tâches définit le degré de parallélisme de celle-ci.

[4] Malgré la présence de mécanismes de migration dans certains systèmes tels que Amber [Ca89] et Clouds [DCM+90], ceux-ci sont considérés comme étant des environnements non-adaptatifs car ils ne possèdent aucun contrôle sur la disponibilité des machines. Par conséquent, ils n'utilisent pas les mécanismes de migration de façon transparente à l'utilisateur.

ce chapitre est présenté dans le tableau 2.2. Pour plus de détails, le lecteur intéressé peut se reporter à [BNK89][Tur93][KN94][KRR95][BFY96].

	Modification du noyau	Adaptabilité	Parallélisme	Gestion de la disponibilité des machines	Hétérogénéité	Exécution à distance	Migration	Checkpointing	Tolérance aux pannes
Newcastle		NA			X				X
Accent	X	NA				T			X
Locus	X	NA			X	T			X
PVM		NA	X		X	T			
PM^2		NA	X		X	T	NT		
Clouds	X	NA	X			T	NT		
Amber		NA	X			T	NT		
Eden	X	NA	X			T	NT	X	X
Task Broker		SA		X		T			X
Demos/MP	X	SA		X		T	T		
Plan 9	X	SA		X	X	T			
Sprite	X	SA		X	X	T	T		
NEST	X	SA		X	X	T			X
Sidle		SA		X	X				X
Spawn		SA		X	X	NT			X
Batch		SA		X	X	T			
QBatch		SA		X	X	T			X
DJM		SA	X			T			X
Amoeba	X	SA	X	X	X	T		X	
Charlotte	X	SA	X	X		T	T		
V	X	SA	X	X	X	T	T		X
Mach	X	SA	X	X	X	T	T		X
MMPVM		SA	X	X	X	T	T		
GATOSTAR		SA	X	X	X	T	T	X	X
CODINE		SA	X	X	X	T	T	X	X
LSF		SA	X	X	X	T	T		X
NQE		SA	X	X	X	T			X
DQS		SA	X	X	X	T		X	X
PRM		SA	X	X	X	T	NT	X	
Butler		A		X	X	NT			
Condor		A		X	X	T	T	X	X
CARMI		A	X	X	X	T	NT	X	
Piranha		A	X	X		T			X

X : Caractéristique existante
T : Caractéristique Transparente
NT : Caractéristique Non-Transparente
NA : Système Non-Adaptatif
SA : Système Semi-Adaptatif
A : Système Adaptatif

Tableau 2.2 : Tableau récapitulatif des caractéristiques de certains environnements

2.2 Exemples d'environnements d'exécution parallèle adaptative

On dénombre très peu d'environnements d'exécution parallèle adaptative car ils n'ont commencé à voir le jour que depuis quelques années. Parmi les environnements mentionnés dans le tableau 2.2 nous allons nous intéresser à la réalisation de deux d'entre eux : Piranha [GK91] et CARMI [PL95].

2.2.1 Piranha

Piranha [GK91] est un système développé par la même équipe de recherche qui a conçu l'environnement de programmation Linda. L'idée de base était d'exploiter le modèle Linda, qui repose sur la communication anonyme, pour faire du parallélisme adaptatif. Du fait que les processus Linda ne communiquent pas de façon explicite, ceci facilite la création ou le retrait de processus.

2.2.1.a Modèle

Le modèle de programmation dans Piranha est basé sur la communication par mémoire virtuellement partagée de l'environnement Linda (*tuple space*). Les applications Piranha sont du type maître/esclaves. Chaque application est composée d'un processus « feeder » unique et d'un certain nombre de processus « Piranha » déterminé en fonction de la disponibilité des machines. Le « feeder » génère des unités de travail et les place dans le tuple space. Les processus « Piranha » prélèvent du travail, font le traitement, et déposent les résultats dans le tuple space. Si un nœud devient indisponible, le processus Piranha qui s'y exécute se retire après avoir exécuté une « courte » fonction appelée « retreat ». Les démons Piranha accordent un délai pour le retrait du processus après lequel ils tuent le processus. Le « feeder » est persistant dans le sens où il reste figé sur la machine sur laquelle il est lancé même en cas d'indisponiblité de celle-ci. En cas de panne du « feeder » ou d'un processus « Piranha », l'application s'arrête.

2.2.1.b Implémentation

Piranha lance des démons sur chaque machine pour contrôler son état de charge et son utilisation par d'autres utilisateurs. Toutes les communications entre démons, entre démons et processus Piranha, et entre processus Piranha passent par le tuple space. Le diagramme des états possibles d'une machine sont présentés dans la figure 2.2.

Chaque application Piranha est représentée par un descripteur (Application Descriptor ou AD) contenant des informations concernant l'application. Ces descripteurs sont stockés dans le tuple space. Un utilisateur lance son application en tapant le nom de l'exécutable sur la ligne de commande. Un processus appelé *manager* sera créé pour

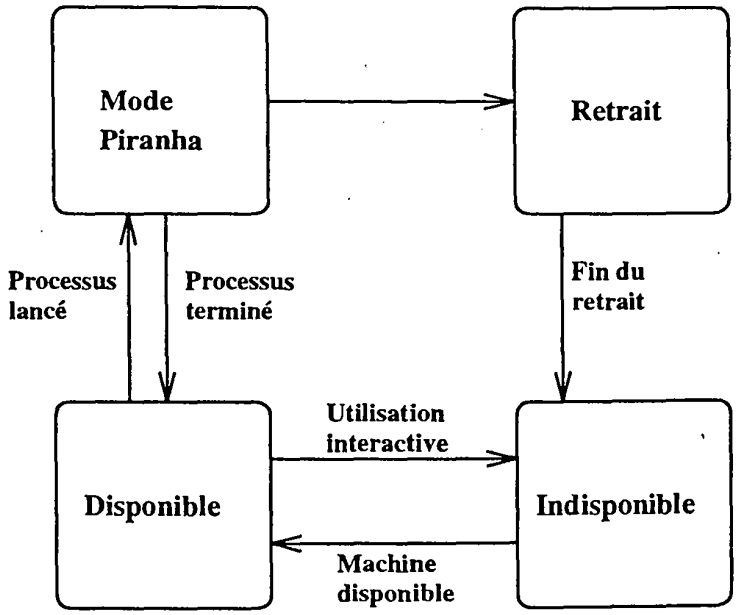


Figure 2.2 : Etats possibles d'une machine Piranha

construire le descripteur de l'application et l'envoyer au démon Piranha local (figure 2.3). Le démon local place le descripteur de l'application dans le tuple space pourqu'il soit visible par les autres démons. Ceci permettrait d'y retrouver la description du processus Piranha à lancer sur la machine une fois qu'elle est disponible. Le manager crée ensuite le feeder et attend qu'il se termine. Une fois l'exécution du feeder terminée, le manager informe le démon local.

2.2.1.c Avantages

Piranha possède les avantages suivants :

- **Simplicité** : la programmation parallèle adaptative sous Piranha est très simple de part l'utilisation du modèle Linda qui comporte très peu de primitives de manipulation du tuple space (quatre opérations de base).
- **Ordonnancement multi-applications** : Piranha possède plusieurs politiques d'ordonnancement multi-applications centralisées et distribuées permettant d'assurer l'équité entre les applications et un temps d'exécution globalement optimisé.
- **Sauvegarde du travail** : Piranha assure l'évolution de l'application en mettant les outils nécessaires afin de permettre la sauvegarde du travail traité par un

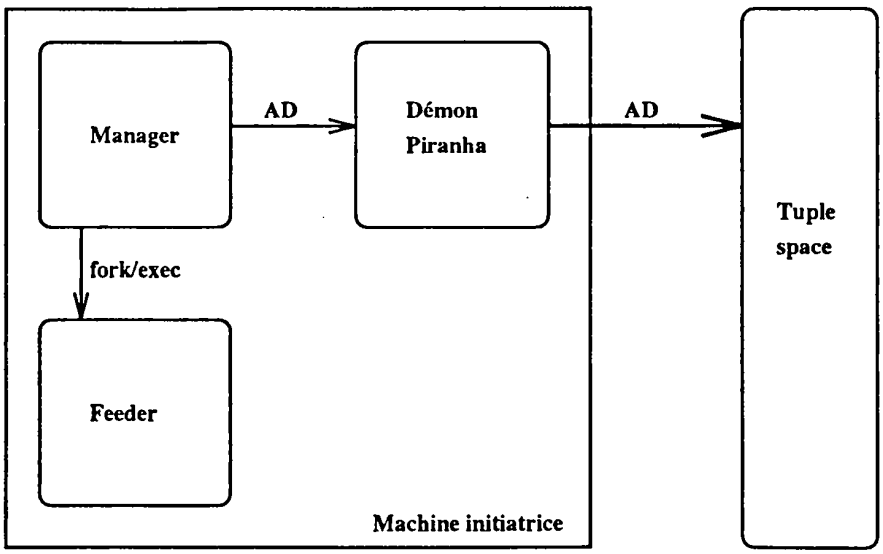


Figure 2.3 : Lancement d'une application dans Piranha

processus Piranha au moment du retrait.

2.2.1.d Inconvénients

Malgré l'apport de Piranha dans le parallélisme adaptatif, il possède les inconvénients suivants :

- **Hétérogénéité limitée** : actuellement Piranha est restreint aux architectures homogènes possédant le même format de données. Ceci limite nettement le champ d'utilisation de Piranha dans un méta-système composé typiquement de plusieurs centaines de machines hétérogènes.
- **Absence de tolérance aux pannes** : Piranha n'implémente pas de mécanismes de tolérances aux pannes que ce soit du « feeder » ou des processus « Piranha ». L'exécution donc d'une application de longue durée de vie est quasiment impossible dans un méta-système où les pannes sont relativement fréquentes.
- **Absence de reconfiguration systématique** : Dès qu'une machine tombe en panne même s'il s'agit d'un redémarrage de la machine, celle-ci n'est pas réintégrée plus tard dans le pool de nœuds. Cette perte progressive de machines peut conduire à un écroulement du système.
- **Accès limité** : du moment que les démons Piranha créent les processus Piranha, ces derniers possèdent les droits d'accès des démons Piranha et non de l'utilisateur.

Ceci peut pénaliser les applications nécessitant l'accès à des fichiers dans le compte de l'utilisateur.

- **Forçage du retrait** : le fait de tuer un processus Piranha après un délai de retreat peut poser un problème aux utilisateurs car ils ne peuvent pas déterminer à priori la durée nécessaire au retrait.

2.2.2 CARMI

Les chercheurs de l'université de Wisconsin-Madison sont partis d'une idée simple : combler le manque d'un gestionnaire de ressources dans un environnement de programmation parallèle pour arriver à un environnement complet pouvant exploiter pleinement les capacités d'un méta-système. Ils ont donc développé le système CARMI (Condor Resource Management Interface) [PL95] qui n'est autre que le résultat de l'interfaçage de Condor [LLM88] et de PVM [BDG+93].

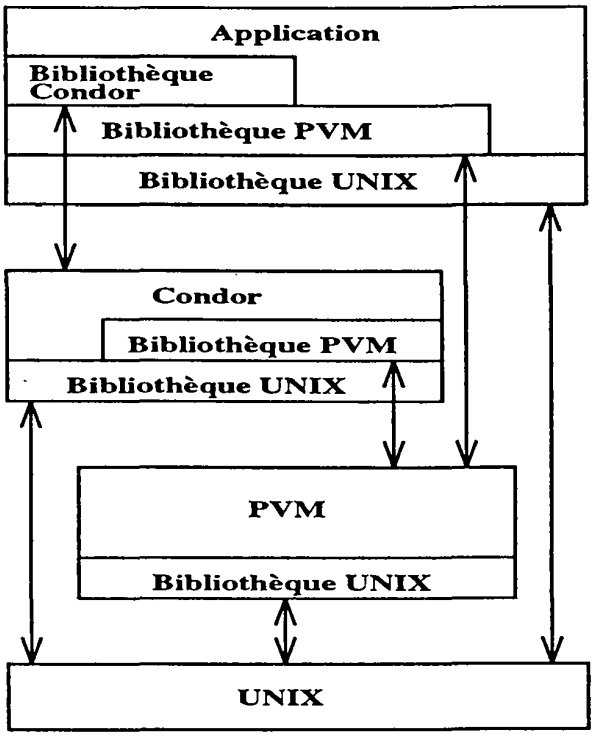


Figure 2.4 : Interfaçage de Condor et de PVM dans CARMI

2.2.2.a Modèle

Le modèle de programmation dans CARMI est basé sur les communications par échange de message. Les applications CARMI sont du type maître/esclaves. Plusieurs processus esclaves prélèvent du travail, font le traitement, et renvoient les résultats au processus maître. Le processus maître génère des unités de travail et les envoie aux processus esclaves tout en gardant une copie. Tant qu'un résultat n'est pas reçu, le travail est considéré comme non traité. En cas de mort d'un esclave, le processus maître remet en jeu le travail correspondant pour être traité depuis le début. Si un nœud devient indisponible, le processus esclave qui s'y exécute migre ou meurt.

La méthodologie de développement dans CARMI est rendue aisée grâce à la mise en place d'un module pour la gestion du travail appelé WoDi (Work Distributor). WoDi s'intercale entre l'application et CARMI pour offrir plus de transparence. L'utilisateur présente son application sous forme de cycles de traitements composés éventuellement de plusieurs étapes. Le rôle de WoDi est de gérer l'exécution de ces étapes et de contrôler la consistence de l'application en vérifiant que les résultats reçus correspondent aux étapes constituées. WoDi utilise l'historique de l'exécution pour gérer au mieux les ressources et ordonnancer de façon intelligente les unités de travail. Les unités de travail dont le traitement est long sont envoyées aux machines les plus rapides.

2.2.2.b Implémentation

CARMI s'appuye sur la gestion de ressources offerte par Condor pour faire du parallélisme adaptatif. Vu que Condor est conçu pour des applications séquentielles, PVM a été utilisé et adapté pour assurer les communications entre les différents composants de CARMI (figure 2.4). Récemment, CARMI a inclut un mécanisme de *checkpointing* d'une application parallèle pour la tolérance aux pannes en intégrant CoCheck [Ste95].

Chaque machine du pool Condor exécute un démon appelé « startd ». Ce dernier est responsable du contrôle de l'état de la machine, de son affectation à une classe de machines, et de l'autorisation de son accès aux applications. Dès que Condor détermine un nombre suffisant de machines disponibles, il crée un processus appelé « shadow » sur la machine sur laquelle l'application a été soumise (figure 2.5). Le *shadow* va à son tour créer un processus appelé « starter » sur chaque machine disponible. Le *starter* est responsable du contrôle du processus de l'application qu'il va créer sur la machine à laquelle il est affecté. Pour une transparence de localité, le *shadow* et le *starter* ramènent les exécutables sur les machines si besoin est. Le *shadow* traite toutes les requêtes de gestion de ressources provenant de l'application via le *starter*.

Pour définir une unité de travail, le processus maître emballe un message définissant le travail à faire et l'envoie à WoDi. WoDi envoie alors ces messages aux processus esclaves créés et reçoit les résultats correspondants ainsi que des statistiques concernant l'exécution. WoDi enregistre la terminaison du traitement correspondant à une unité de travail et informe le processus maître. Si un processus esclave meurt suite à une panne,

WoDi renvoie de nouveau l'unité de travail correspondante.

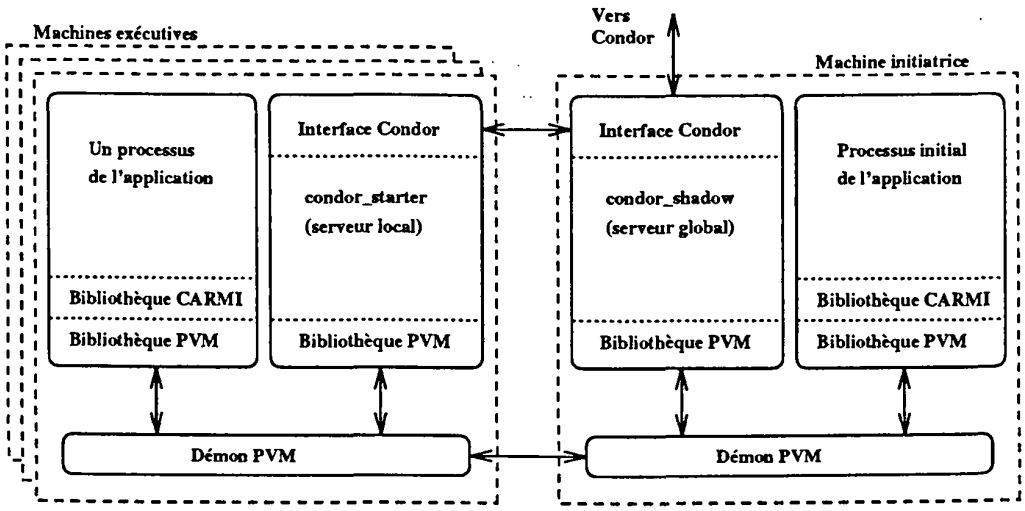


Figure 2.5 : Exécution adaptative dans CARMI

2.2.2.c Avantages

CARMI est le fruit d'un effort de fédération de deux environnements déjà existants destinés respectivement à la communication et à la gestion des ressources. Par conséquent, en plus des avantages liés directement à son modèle, CARMI hérite de certaines caractéristiques de ces environnements :

- **Facilité d'utilisation** : CARMI peut être vu comme une extension de la bibliothèque PVM. Par conséquent, les utilisateurs familiarisés avec le développement sous PVM peuvent porter leur applications de façon aisée.
- **Possibilité de spécification des ressources requises** : l'utilisateur peut fournir une spécification et une description des ressources nécessaires au bon déroulement de son application. Cette caractéristique est importante car elle peut contribuer à un bon ordonnancement des processus d'une application en fonction des informations apportées par l'utilisateur qui est mieux placé pour connaître les exigences de son application.
- **Transparence** : la complexité de gestion d'un environnement hétérogène et opportuniste est cachée aux applications

2.2.2.d Inconvénients

- **Effet stationnaire** : du moment que la panne d'un processus esclave implique la reprise du traitement depuis le début, l'application peut « piétiner » et ne plus évoluer.
- **Phénomène de ralentissement** : dès qu'une machine devient indisponible, Condor suspend les processus qui s'y exécutent pendant une durée allant jusqu'à 10 minutes. Ceci peut entraîner un ralentissement de l'exécution.
- **Mécanisme de sauvegarde non adapté au modèle** : CARMI réutilise le mécanisme de checkpointing de CoCheck [Ste95] qui consiste à sauvegarder l'état des processus qui composent une application parallèle. La relance d'une application sauvegardée implique la reconstituion de l'application avec le même nombre de processus sur les mêmes architectures qu'au moment de la sauvegarde. Ceci ne peut être effectué que si les machines sont disponibles au moment de la relance.
- **Absence de reconfiguration systématique** : tout comme Piranha, CARMI ne gère pas systématiquement la réintégration des machines défaillantes.
- **Absence de politique d'ordonnancement globale** : Condor a été conçu à l'origine pour ordonnancer des applications séquentielles. Le fait de l'intégrer dans un environnement parallèle ne lui permet pas d'avoir une vue globale sur les applications parallèles de CARMI.

2.3 Conclusion

A travers ces deux premiers chapitres, on peut dire qu'il y a une source de puissance de calcul importante dans un méta-système qui peut être exploitée pour faire du calcul intensif. Cependant, le caractère multi-utilisateurs, hétérogène, défaillant et opportuniste d'un méta-système rend son exploitation très difficile. La conception d'un environnement fournissant des fonctionnalités transparentes est souhaitable, et permet aux développeurs d'applications de ne pas se soucier de la complexité de la plateforme.

A travers l'étude faite sur les environnements déjà existants, nous ne pouvons pas affirmer l'existence d'un tel environnement. Les environnements présentés dans ce chapitre ne tiennent pas compte ou tiennent compte partiellement de ces caractéristiques qui sont intrinsèquement liées. Par conséquent, nous avons pensé à développer un environnement d'exécution parallèle adaptative appelé MARS (Multi-user Adaptive Resource Scheduler) dont les grandes lignes sont détaillées dans le chapitre qui suit.

Chapitre 3

Infrastructure MARS

MARS (Multi-user Adaptive Resource Scheduler) est un environnement de programmation parallèle adaptative permettant d'exécuter plusieurs applications (multi-applications) pouvant appartenir à différents utilisateurs (multi-utilisateurs), en cohabitation avec d'autres applications non-MARS (séquentielles et parallèles). Il intègre essentiellement l'ordonnancement adaptatif et la régulation dynamique de charge. MARS est opérationnel depuis plus d'un an et a été utilisé par différents utilisateurs dans divers domaines.

Ce chapitre est destiné à montrer les mécanismes internes de l'environnement MARS qui sont complètement transparents à l'utilisateur. Nous commençons par présenter en détails les différents éléments qui composent l'exécutif MARS, puis nous abordons l'interaction entre l'exécutif MARS et les applications. Nous terminons par quelques expérimentations permettant de montrer l'apport de notre approche.

3.1 Exécutif MARS

MARS est à la fois un environnement d'exécution et un environnement de programmation parallèle. L'exécutif MARS a pour rôle de gérer les ressources d'un méta-système tandis que la bibliothèque de primitives sert d'interface entre les applications des utilisateurs et l'environnement d'exécution MARS. Les applications sont du type maître/esclaves.

3.1.1 Objectifs conceptuels

Lors de l'élaboration du cahier de charge de MARS, nous nous sommes fixé un certain nombre d'objectifs conceptuels liés aux caratéristiques d'un méta-système :

- **portabilité** : MARS ne nécessite aucune modification au niveau du système d'exploitation UNIX sous-jacent. L'avantage est qu'il peut être porté sur différentes architectures. Actuellement il fonctionne sur 5 architectures (Sparc/SunOs, Sparc/Solaris, Alpha/OSF-1, Silicon Graphics/IRIX et PC/Linux).

- **reconfiguration dynamique** : tout changement dans la configuration du méta-système (ajout, retrait et réintégration de nœuds) est automatiquement pris en considération par le système MARS de façon dynamique sans nécessité de relancer le système.

- **protection** : MARS est un ordonnanceur global destiné à gérer l'ensemble des applications MARS. Cependant, il ne nécessite aucun droit super-utilisateur pour être lancé. Toute application MARS s'exécute dans la limite des droits d'accès de son propriétaire. Ceci permet non seulement de protéger les applications les unes contre les autres, mais aussi d'écarter toute éventualité d'une intrusion ou de la présence d'une défaillance dans la sécurité.

- **transparence** : MARS offre un maximum de transparence pour les développeurs d'applications. Les utilisateurs n'ont pas à gérer des tâches complexes telles que la disponibilité des nœuds et la régulation de charge.

- **tolérance aux pannes** : MARS détecte et gère les pannes des nœuds qui sont très fréquentes dans un méta-système. Entre autres, un mécanisme de checkpointing adapté aux applications MARS a été développé [KTG97]. Il permet de sauvegarder périodiquement les applications en vue de les relancer en cas de panne.

- **extensibilité** : MARS possède une structure hiérarchique qui lui permet de gérer plus efficacement des réseaux à large échelle.

- **convivialité** : une interface graphique a été développée pour l'administrateur du système MARS afin d'observer et de contrôler le méta-système de façon conviviale. Une autre interface graphique est mise à la disposition des utilisateurs pour faciliter le développement et la mise au point d'applications MARS.

3.1.2 Fonctionnement et architecture de MARS

Le système MARS contrôle un ensemble de nœuds. Sur chaque nœud tourne un gestionnaire de nœud (*node manager* ou NM) dont le rôle est de surveiller l'état de charge du nœud sur lequel il se trouve (figure 3.1). Les gestionnaires de nœuds dépendent d'un serveur de groupe (*group server* ou GS) vers lequel ils envoient les changements d'état de charge détectés.

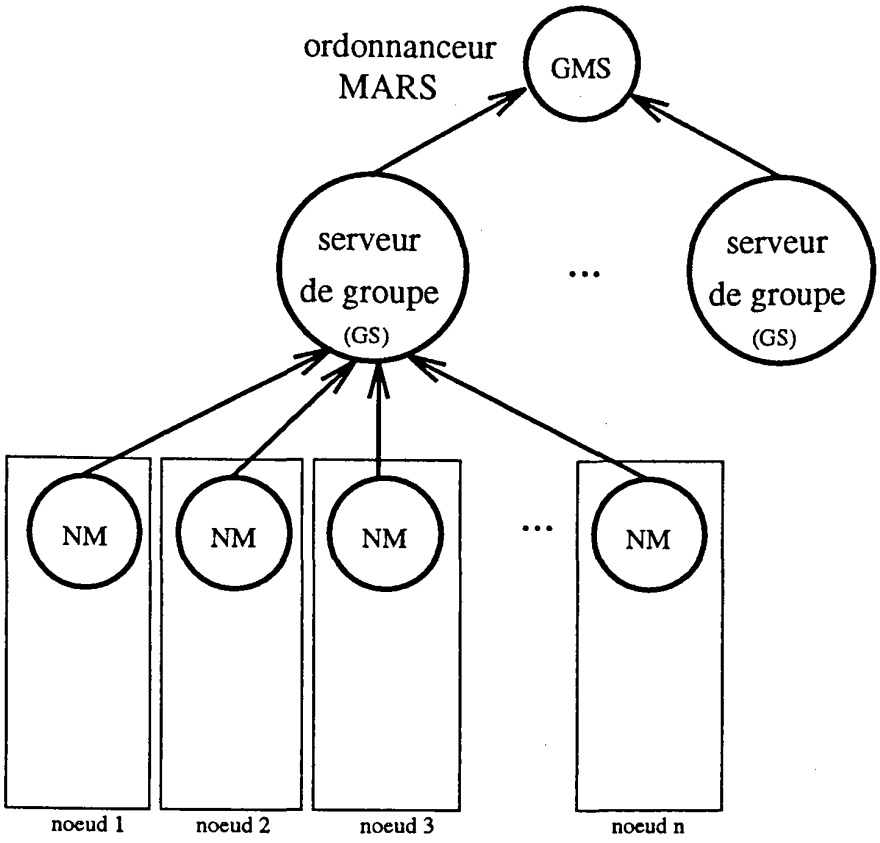


Figure 3.1 : Architecture de MARS

Il a été prévu que plusieurs serveurs de groupe puissent coexister gérant chacun un pool de nœuds sélectionnés selon certains critères (lieu géographique, cadre d'administration, homogénéité, etc.), et coopérant par l'intermédiaire d'un ordonnanceur global (*global MARS scheduler* ou GMS) chargé de la gestion et du contrôle d'un ensemble de pools. Cette structure hiérarchique à deux niveaux n'a pas été réalisée dans le cadre de cette thèse. Actuellement, l'implémentation se limite à un seul niveau d'hiérarchie. On dispose donc d'un seul GS contrôlant un ensemble de NMs. Cette structure, quoique centralisée, permet tout de même de gérer un grand parc de nœuds (une centaine) sans risque d'écroulement du système. Un exécutif MARS, dans la suite du rapport, désigne un serveur de groupe et les gestionnaires de nœuds qu'il contrôle. MARS est conçu au dessus d'une plateforme multi-threadée appelée PM^2 (figure 3.2) décrite succinctement dans le paragraphe suivant.

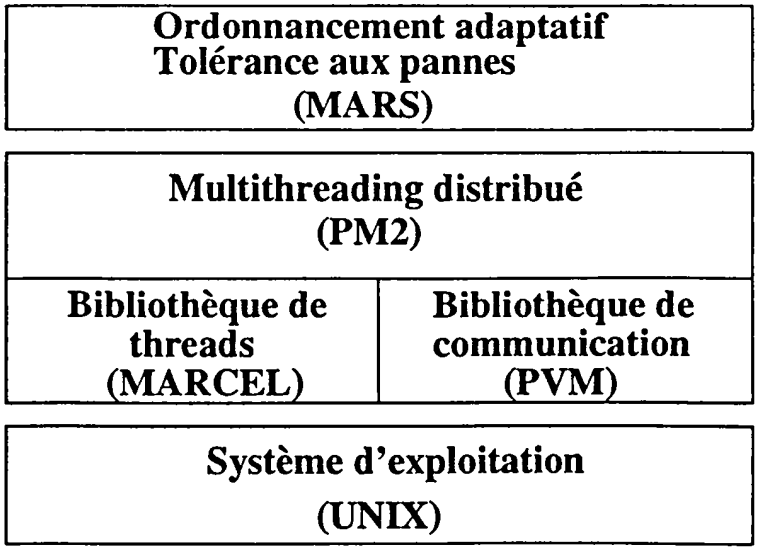


Figure 3.2 : Structure en couches de MARS

3.1.3 Environnement PM^2

PM^2 [NM95] repose sur une bibliothèque de processus légers (threads) appelée MARCEL et la bibliothèque de communication PVM [BDG+93].

MARCEL permet d'ordonnancer un ensemble de processus légers créés au sein d'un unique processus UNIX de façon préemptive. L'ordonnancement tient compte des priorités attribuées aux processus légers. MARCEL fournit en plus un ensemble de primitives POSIX et étendues (non POSIX) permettant, entre autres, la gestion de processus légers (création, destruction, etc.) et l'ajustement dynamique de la pile d'exécution. L'hétérogénéité permet d'exploiter un parc important de machines. La dynamicité de la machine virtuelle PVM permet de rajouter ou de supprimer des nœuds à tout moment. Les mécanismes de notification permettent de transmettre des informations correspondants à des événements tels que l'ajout d'un démon PVM, et la mort d'un démon ou d'une tâche PVM. L'ordre de messages permet la réception de messages dans l'ordre de leur émission entre deux processus.

La bibliothèque de communication PVM est devenue un standard de fait car elle est largement utilisée dans le développement d'applications parallèles. Par ailleurs, l'environnement de programmation PVM présente certaines propriétés intéressantes qui ont contribué au développement du système MARS. Il s'agit de l'hétérogénéité, la dynamicité, la notification et l'ordre des messages.

PVM a été associée à MARCEL pour étendre la notion de processus léger dans un contexte distribué. L'environnement PM^2 résultant est basé sur le concept d'appel de

procédure à distance (LRPC) qui consiste à exécuter un service distant (thread) synchrone ou asynchrone avec éventuellement un retour de résultat. Ceci se traduit par une création distante d'un processus léger (figure 3.3). PM^2 offre certaines fonctionnalités avancées telle que la migration de processus légers.

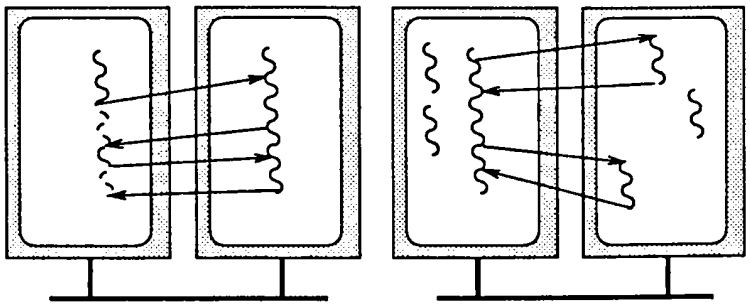


Figure 3.3 : Concept de LRPC dans l'environnement PM^2

3.1.4 Serveur de groupe

Pour lancer l'exécutif MARS, il suffit d'exécuter le GS sur une machine quelconque faisant partie du pool de nœuds MARS. Ce dernier est généralement calqué sur une machine virtuelle PVM préalablement créée. Le GS procède systématiquement à la création des NMs sur tous les nœuds du pool à l'exception du nœud initiateur. En effet, la machine sur laquelle le GS tourne ne doit pas être chargée pour garantir un temps de réponse optimal ou une meilleure réactivité du GS. Par conséquent, ce nœud est automatiquement exclu du pool de nœuds MARS. Pour des informations supplémentaires concernant l'installation de l'exécutif MARS, voir l'annexe A.

Tout nœud est contrôlé par un seul GS. L'intersection de deux pools de nœuds contrôlés par deux GS différents doit être vide. Ceci permet d'écarter tout conflit et de préserver la nature d'un ordonnaceur global. Le test de présence d'un autre GS ou d'un autre NM se fait en interrogeant un port de communication prédéterminé. Le GS est constitué des éléments suivants (figure 3.4) :

- un thread dispatcher (RPC) qui est perpétuellement à l'écoute des requêtes provenant des applications MARS via les primitives de la bibliothèque MARS. Une fois qu'une requête est reçue, elle est directement aiguillée vers le service correspondant. Les services appelés par la bibliothèque MARS comprennent entre autres l'enrôlement (voir section 3.2.1).
- un service LRPC qui récupère les informations remontées par les NMs. Ces in-

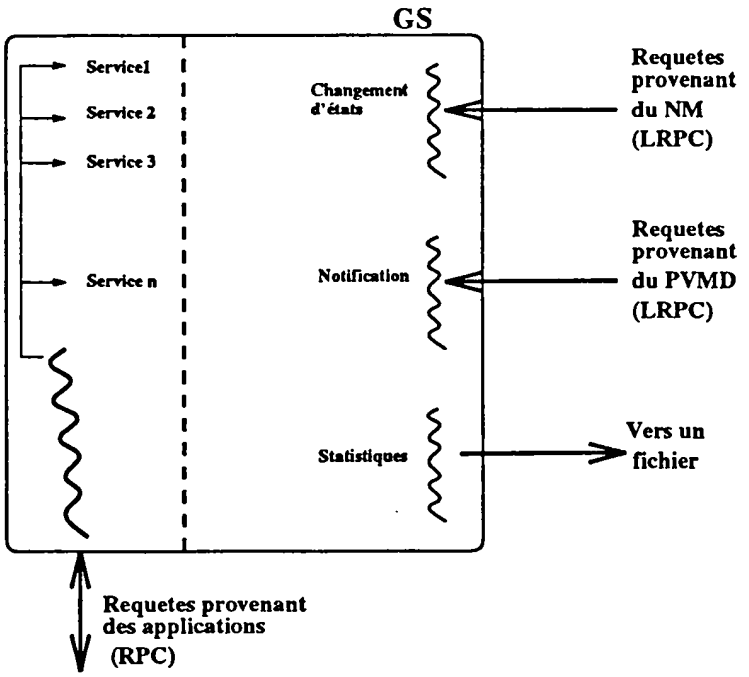


Figure 3.4 : Composants du GS

formations sont simplement les états des nœuds déduits à partir de valeurs de charge associées à un ensemble d'indicateurs de charge. Ce service LRPC est chargé de la mise à jour des informations liées à l'état global du méta-système et du déclenchement éventuel de certaines opérations de l'ordonnanceur.

- un thread de notification qui gère l'ajout d'un nœud. Pour rajouter une nouvelle machine au pool de nœuds MARS, il suffit de passer par la console PVM. Une fois que la machine est rajoutée avec succès, le démon PVM créé envoie un signal de notification qui sera capturé par le GS. Ce dernier crée, en conséquence, un thread qui se charge de lancer un NM sur la machine ajoutée.

- un autre thread de notification qui gère la suppression d'un nœud. Il est créé systématiquement à chaque fois qu'une machine est supprimée de la machine virtuelle PVM. Deux cas peuvent se présenter : une suppression définitive (par exemple une panne) ou une suppression momentanée (par exemple un redémarrage de la machine ou la mort du démon PVM). Dans ce dernier cas, la machine doit être réintégrée par le GS. Pour différencier les deux possibilités, le GS consulte un fichier déterminé à la recherche du nom de la machine. Si le nom est trouvé, la machine doit être définitivement supprimée et un mail est envoyé à l'administrateur du système MARS pour l'informer.

- un thread de statistiques (optionnel) permettant de sauvegarder périodiquement dans un fichier les états de tous les nœuds qui composent le méta-système. La période peut être réglée par l'administrateur du système MARS. Le fichier produit permet de tracer des courbes pour observer l'utilisation des machines en présence ou en l'absence d'applications MARS.

De plus, le GS reçoit des requêtes en provenance des applications via la bibliothèque MARS, comme l'enrôlement d'une application au système (voir l'API en annexe B). Le rôle du GS est ensuite d'envoyer au processus maître d'une application MARS les ordres de création ou de destruction de processus esclaves sur les nœuds devenus disponibles ou indisponibles respectivement. Cela est détaillé dans le paragraphe sur la liaison entre l'exécutif et les applications. Le contrôle de plusieurs applications MARS simultanément est détaillé dans le paragraphe sur l'ordonnancement multi-applications.

3.1.5 Gestionnaires de nœuds

Une fois créés, les NMs déterminent les valeurs de charge, en déduisent un état de charge et informent le GS d'un changement éventuel d'état. Dans le cadre de cette thèse, on ne s'est pas trop focalisé sur le problème de sélection et d'interprétation des critères de charge en répondant aux deux questions : quels sont les indicateurs de charge que l'on doit choisir ? et comment déduire un état de charge à partir de ces indicateurs ?

En réalité, il n'existe pas de règle générale pour sélectionner tel ou tel indicateur. Cela dépend à la fois de l'environnement et du type d'applications supportées (utilisation CPU intensive, E/S, ou mémoire). La forte hétérogénéité d'un méta-système a une répercussion prépondérante sur le choix des indicateurs pour telle ou telle machine mais, à notre connaissance, très peu de systèmes en tiennent compte. La déduction d'un état de charge se fait, en général, par rapport à un ou plusieurs seuils de charge. Dans la plupart des cas, ces seuils s'obtiennent par des combinaisons (linéaires ou autres) de valeurs de charge associées aux indicateurs de charge choisis. Les valeurs de charge doivent être sélectionnées de façon qu'une machine déclarée chargée, non chargée, ou utilisée en interactif le soit vraiment.

La majorité des environnements se limite à la charge moyenne pour avoir une vue globale stable de l'état du système. Dans le système MARS, un nœud est considéré disponible si la charge moyenne est inférieure à 2.0, 1.5 et 1.0 pendant les 1, 5 et 10 dernières minutes respectivement, et que le nœud n'est pas utilisé en mode interactif pendant 3 minutes. Un nœud dont le clavier et/ou la souris sont actifs est considéré comme étant un nœud dans un état de charge maximal et par conséquent il ne sera pas alloué aux applications par le système MARS.

La figure 3.5 montre un diagramme des états de transition des nœuds MARS. Les états des nœuds qui n'exécutent aucune application MARS oscillent entre disponible (IDLE), réquisitionné (OWNED) et occupé (BUSY) selon que le nœud n'est pas chargé, que

la console du nœud est active ou que le nœud est fortement chargé par des applications non-MARS. Si le nœud est IDLE au moment de la soumission d'une application MARS, le nœud devient dans l'état RUNNING et exécute des tâches MARS. Dans cet état, d'autres tâches peuvent être créées pour d'autres applications MARS (time sharing autorisé) tant que la charge du nœud ne dépasse pas un certain seuil, auquel cas le nœud devient chargé (LOADED). Deux états supplémentaires SUSPENDED et DELETED sont attribués respectivement à un nœud temporairement inutilisé et un nœud définitivement exclu du pool de nœuds.

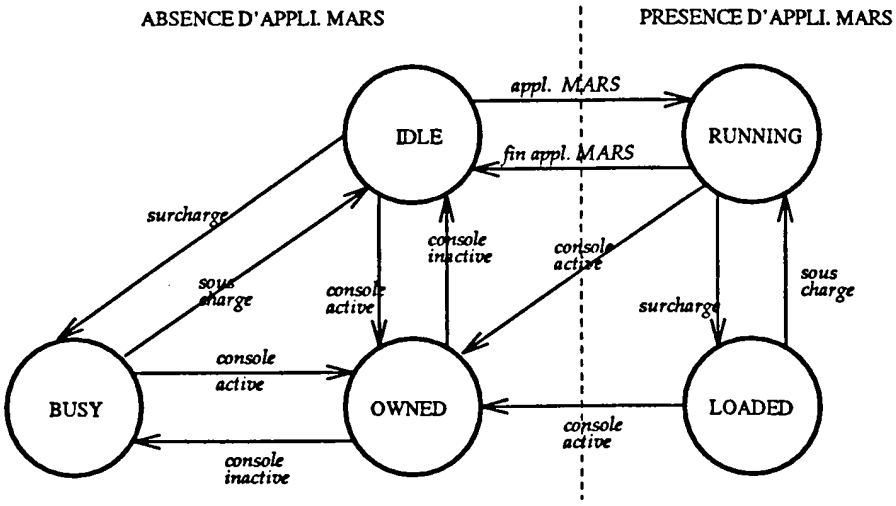


Figure 3.5 : Etats de transition d'un nœud MARS

La scrutation des indicateurs de charge se fait selon deux périodes : toutes les 5 secondes si le clavier et la souris ne sont pas en activité et après 3 minutes sinon. La remontée des informations par le NM au GS ne se fait que lors d'un changement d'état. Chaque NM contient un thread dispatcher (RPC) qui assure un seul service en l'occurence le « ping ». Ce service est appelé dès qu'une application MARS est lancée. Il permet au processus maître de l'application de savoir si le nœud sur lequel il s'exécute fait partie du pool de nœuds MARS (présence d'un NM). Dans ce cas, le NM informe le processus maître sur la localisation du GS afin de pouvoir communiquer avec lui. Ceci constitue le seul moyen de déterminer directement le nom de la machine sur laquelle le GS tourne.

Le tableau 3.1 regroupe les différents états possibles attribués à un nœud MARS. S signifie le seuil de charge utilisé.

Etat	Signification
OWNED	nœud en utilisation interactive (état de charge maximale)
IDLE	charge < S et absence d'applications MARS
BUSY	charge > S et absence d'applications MARS
RUNNING	charge < S et présence d'applications MARS
LOADED	charge > S et présence d'applications MARS
SUSPENDED	nœud temporairement inutilisé (redémarrage), réintégration automatique
DELETED	nœud définitivement supprimé (panne), notification à l'administrateur

Tableau 3.1 : Tableau récapitulatif des différents états d'un nœud MARS

3.2 Interaction entre l'exécutif et les applications MARS

Le modèle de programmation supporté par MARS est le style maître/esclaves (SPMD). La tâche maître génère du travail à traiter par les tâches esclaves et contrôle toute l'application. Chaque tâche esclave reçoit du travail de la part du maître, effectue un traitement et renvoie un résultat au maître. Le modèle SPMD fonctionne bien dans un environnement adaptatif tel que MARS. L'exécution adaptative d'une application se traduit par la création de nouveaux processus esclaves sur les nœuds disponibles, et par le retrait des tâches esclaves s'exécutant sur des nœuds chargés ou réquisitionnés tout en récupérant le travail non encore achevé pour le reprendre en cas de disponibilité de nœuds.

Le système MARS gère de façon transparente à l'utilisateur l'exécution adaptative de l'application :

- lorsqu'un nœud devient disponible, le système MARS informe le maître pour y créer un nouveau esclave. On dira dans ce cas que MARS *déplie* l'application.
- si un nœud exécutant une tâche esclave est requisitionné ou devient chargé, le système MARS *replie* l'application en retirant l'esclave. Dans ce cas, l'esclave remet éventuellement les données pendantes au processus maître et meurt.

Les processus maître et esclaves sont aussi développés au dessus de PM^2. Ils communiquent grâce aux LRPCs pour la distribution des unités de travail et les retours de résultats. En revanche, les communications entre l'exécutif MARS et les applications MARS ne se font pas par LRPCs.

La figure 3.6 illustre les communications entre les différents composants dans l'environnement MARS. L'exécutif MARS et les applications MARS peuvent être vus comme étant des applications parallèles multi-threadées PM^2. De ce fait, les communications intra-MARS et intra-applications MARS se font par LRPCs. L'exécutif MARS et les applications peuvent appartenir à différents utilisateurs et ne peuvent donc commu-

niquer par LRPCs car PM[2] est mono-utilisateur[1]. Pour établir un pont entre l'exécutif MARS et les applications nous avons eu recours aux RPCs SUN [RPC88b][RPC88a]. L'utilisation de ces mécanismes permet d'éviter de lancer l'exécutif MARS dans le mode super-utilisateur et par conséquent de protéger le système contre toute intrusion.

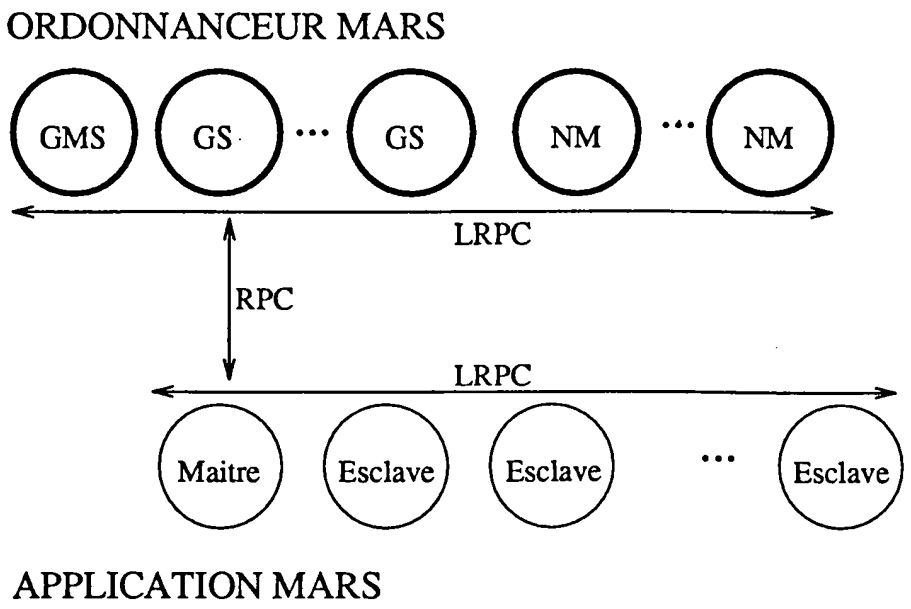


Figure 3.6 : Communications dans l'environnement MARS.

3.2.1 Lancement, enrôlement et fin d'une application MARS

Pour lancer une application MARS, il suffit d'exécuter le processus maître sur une machine comportant un NM. Le processus maître doit se faire connaître auprès de l'exécutif MARS en s'enrôlant dans celui-ci. Ceci se traduit par un appel à une primitive correspondante dans la bibliothèque MARS. Cette bibliothèque contient un ensemble de primitives permettant la gestion et le contrôle des applications MARS. On en distingue les primitives qui nécessitent la contribution du GS tel que l'enrôlement et les primitives qui ne nécessitent pas la contribution du GS tel que la détermination du nombre d'esclaves en exécution.

L'enrôlement d'une application MARS passe, de façon transparente, par les étapes suivantes :

[1] Caractéristique héritée de PVM.

- tentative d'interrogation d'un NM local sur un port prédéterminé. Si la localisation du NM n'est pas fructueuse alors le nœud ne fait pas partie du pool de nœuds MARS et par conséquent l'enrôlement échoue.
- le NM communique au processus maître le nom de la machine sur laquelle le GS tourne.
- le processus maître envoie une requête RPC d'enrôlement au GS.
- le GS enregistre l'application, établit une connection avec celle-ci et lui envoie une confirmation d'enrôlement.

Après enrôlement, la gestion adaptative de l'application n'est effective qu'après appel explicite de la part du processus maître via une primitive de création de tâches. Ainsi, le nombre initial de tâches esclaves créées est égal au nombre total de nœuds disponibles dans le méta-système.

La terminaison d'une application MARS est contrôlée par le processus maître. Une fois la condition d'arrêt satisfaite, le processus maître doit appeler une autre primitive pour informer le GS de la fin d'exécution de l'application. Ceci permet au GS de mettre à jour toutes ses structures de données. Néanmoins, si ce protocole n'est pas respecté, l'exécutif MARS est capable de détecter une terminaison impropre d'une application et par conséquent de cesser toute interaction avec elle. Dans tous les cas, l'utilisateur ayant lancé l'application est informé par messagerie électronique sur l'état de terminaison (normal ou anormal) de son application.

3.2.2 Dépli d'une application MARS

Le dépli d'une application se traduit par l'allocation d'un nœud disponible à l'application. Dès que l'état d'un nœud transite à l'état IDLE, le NM qui s'exécute sur ce nœud informe le GS duquel il dépend en lui envoyant un LRPC (figure 3.7). Le thread ainsi créé pour exécuter le service enregistre ce changement d'état et fait appel à l'ordonnanceur. Ce dernièr sélectionne une application pour lui attribuer le nœud disponible. Le GS envoie un RPC au processus maître de l'application lui demandant de créer un nouveau processus esclave sur ce nœud. Tous ces événements sont gérés par MARS et sont donc transparents à l'utilisateur. Une fois le processus esclave créé, il va procéder à la demande du travail et entamer le traitement.

3.2.3 Repli d'une application MARS

Le repli survient lorsqu'un nœud n'est plus disponible et qu'une application MARS utilise ce nœud. Dès que l'état d'un nœud transite à l'état OWNED ou LOADED, le NM qui s'exécute sur ce nœud informe le GS duquel il dépend en lui envoyant un LRPC (figure 3.8). Le thread ainsi créé pour exécuter le service enregistre ce changement d'état

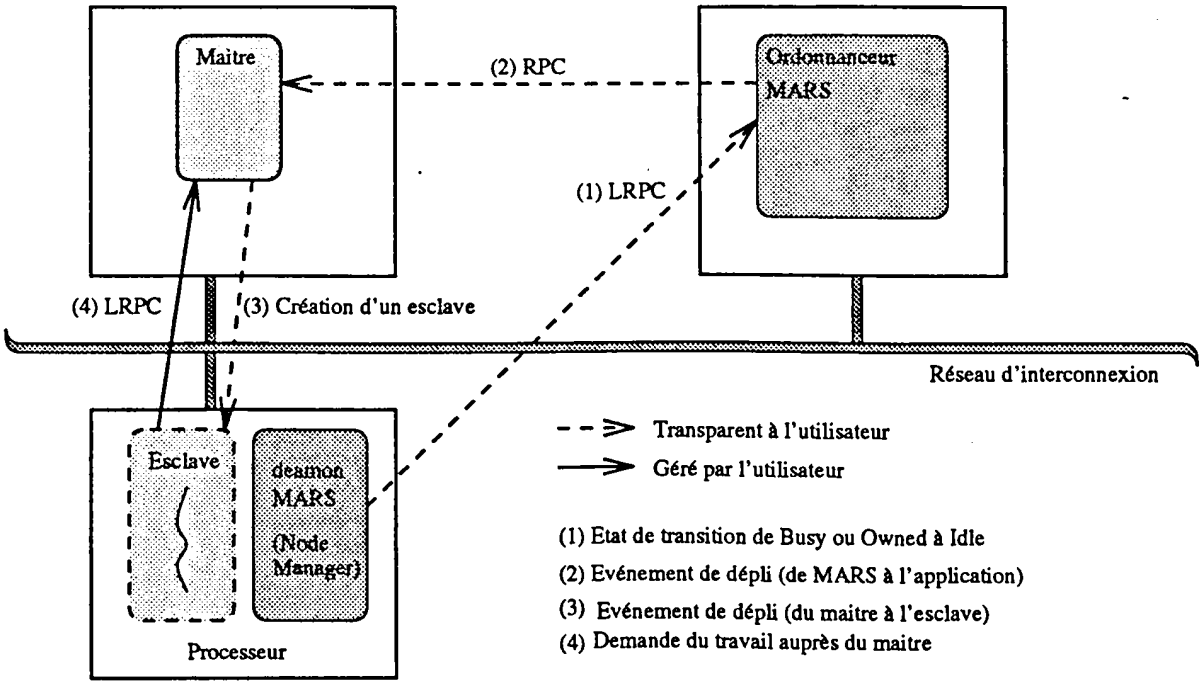


Figure 3.7 : Evénements lors d'un dépli.

et fait appel à une fonction chargée du repli. Cette dernière détermine les applications ayant un processus esclave sur ce nœud. Le GS envoie un RPC à chaque processus maître des applications correspondantes leur demandant de retirer leurs processus esclaves sur ce nœud. Une fois l'événement de repli reçu, les processus esclaves vont procéder au repli en renvoyant l'état courant des données à leurs processus maîtres respectifs. La manière dont les événements de dépli et de repli sont gérés par l'utilisateur sera détaillée dans le chapitre suivant.

3.2.4 Ordonnancement multi-applications

L'ordonnancement d'applications parallèles a toujours été un problème central dans les environnements d'exécution parallèle. Le caractère adaptatif des applications gérées par MARS apporte une dimension supplémentaire au problème d'ordonnancement. Le choix d'une politique d'ordonnancement dépend des objectifs fixés. Par exemple, on peut considérer la maximisation de l'utilisation des ressources ou l'équité. Le nombre de nœuds contrôlés peut influencer notre choix. En général, on doit répondre aux questions suivantes :

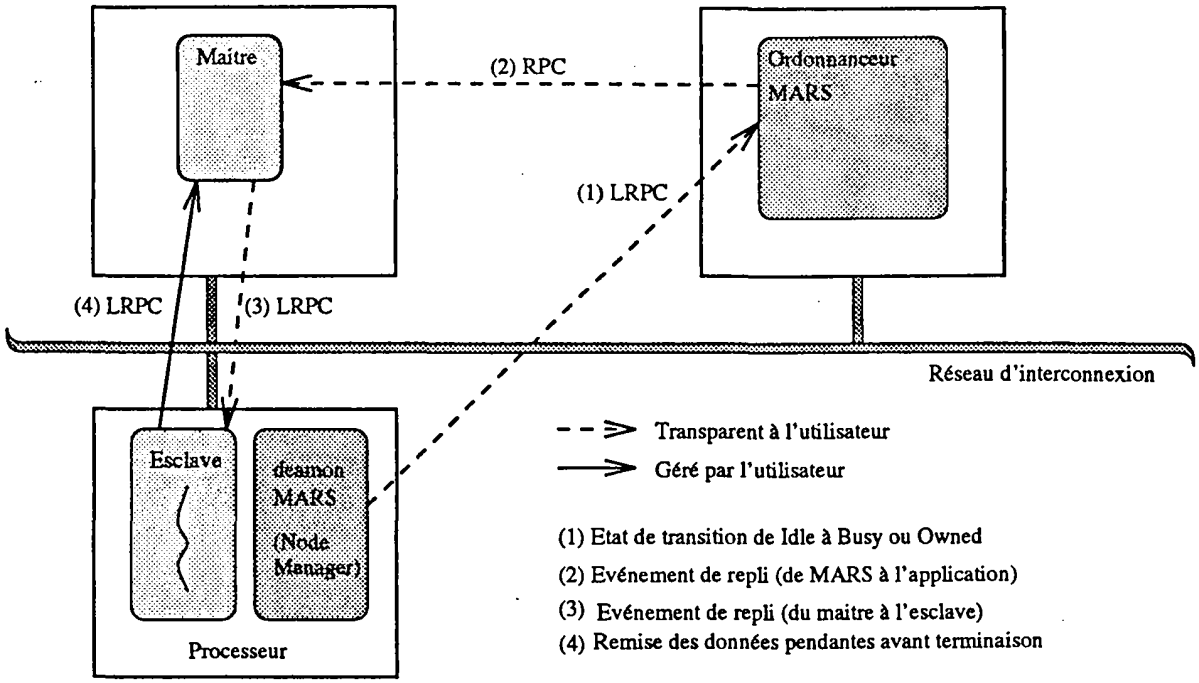


Figure 3.8 : Evénements lors d'un repli

- L'ordonnancement doit-il être centralisé ou distribué ?
- Le système doit-il ordonnancer une seule application à la fois ?
- Sinon, comment les applications doivent-elles partager les nœuds ?
- Dans le cas où plusieurs applications s'exécutent de façon concurrente, peuvent-elles utiliser les nœuds de façon exclusive ou en temps partagé ?

Le choix d'un ordonnanceur centralisé ou distribué dépend principalement de la configuration du système *i.e.* le nombre et la localisation des nœuds. Une politique d'ordonnancement distribuée résoud certes le problème d'engorgement présent dans un ordonnanceur centralisé mais conduit à une implémentation et une prise de décision complexes. Une politique d'ordonnancement centralisée est généralement adoptée par les systèmes de part sa simplicité. C'est le cas des systèmes Condor [LLM88] [LL90] [BL91], DQS [Gre95], Godzilla [Cra90], LSF [Cor94] et RES [Car92].

Plusieurs stratégies peuvent être considérées dans l'ordonnancement de plusieurs applications parallèles dans un méta-système :

- Un ordonnanceur peut attribuer l'ensemble des nœuds disponibles à une seule application à la fois. Il peut permettre à une application de s'exécuter jusqu'à son terme avant d'activer une autre application (« queueing »), ou préempter l'application active pour pouvoir exécuter une autre (« time sharing »). Dans ce dernier cas, les algorithmes classiques tels que le tourniquet, le « plus petit temps résiduel d'abord » ou « le plus long temps écoulé d'abord » peuvent être utilisés. Cependant, l'ordonnancement peut perdre de l'efficacité suite aux replis multiples induits par la préemption.

- Au lieu d'allouer tous les nœuds à une seule application, un ordonnanceur peut partitionner les nœuds disponibles sur toutes les applications. Ce partitionnement peut être temporel (« time sharing »), spatial (« space sharing ») [ASOW][ALM+91] ou combinant les deux. Dans le premier cas, un nœud peut être utilisé par plusieurs applications contrairement au second où un nœud doit être utilisé de façon exclusive.

Face à l'importance et la complexité de l'ordonnancement multi-applications parallèles en plus de certaines considérations tels que l'hétérogénéité et les besoins en ressources spécifiés par les applications, une autre thèse a été lancée sur le sujet pour déterminer une politique d'ordonnancement adéquate [KTG98]. Cependant, on peut dire qu'actuellement dans le système MARS, la prise de décision quant à l'ordonnancement est centralisée au niveau du GS. Les nœuds disponibles peuvent être utilisés en temps partagé par toutes les applications actives. L'exécutif MARS réagit au changement d'état d'un nœud comme suit :

- lorsqu'un nœud devient disponible, l'exécutif MARS l'alloue aux applications tour à tour (en tourniquet).

- dans le cas où un nœud devient indisponible, l'exécutif MARS prévient tous les processus maîtres, ayant des processus esclaves en exécution sur ce nœud, pour effectuer le repli.

3.2.5 Les composantes d'une application MARS

Cette section présente les éléments servant à gérer des applications MARS qui sont transparents du point de vue de l'utilisateur. Les parties qui doivent être gérées ou contrôlées par l'utilisateur font partie de la méthodologie de développement d'une application MARS et sont décrits dans le chapitre suivant.

3.2.5.a Le processus maître

Un processus maître d'une application MARS est constitué des éléments suivants :

- un thread dispatcher (RPC) répondant aux services appelés par le GS (essentiellement le dépli et le repli).
- un thread de notification de la terminaison (normale ou anormale) des esclaves.
- un thread de statistiques (optionnel) permettant de sauvegarder dans des fichiers les timings et l'allocation des nœuds dans le temps.

3.2.5.b Les processus esclaves

Un processus esclave d'une application MARS est constitué des éléments suivants :

- un thread répondant aux services appelés par le processus maître (essentiellement le repli).

3.2.6 Console MARS

Dans le cadre d'un projet de maîtrise et de DESS, nous avons développé deux interfaces graphiques pour rendre convivale et ergonomique l'utilisation de l'environnement MARS. La première console destinée à l'administrateur du système MARS permet d'observer et de contrôler toute l'infrastructure MARS. L'administrateur peut intervenir à tout moment en ajoutant ou en supprimant des nœuds, en arrêtant certaines applications MARS, etc. La seconde console est destinée aux utilisateurs du système MARS. Celle-ci peut être lancée indifféremment avant ou après le lancement d'une application. Elle permet l'observation et le contrôle des applications lancées par un utilisateur. Toutes les informations concernant le déroulement de l'application sont affichées (nombre de nœuds, noms des machines utilisées, courbe d'allocation des nœuds dans le temps, nombre de replis et de déplis, etc.). Les interfaces graphiques ont été développées en utilisant Tcl/Tk.

3.2.7 Tolérance aux pannes

Une panne de machine peut induire la disparition d'un composant du système MARS ou d'une application MARS. La prise en compte de la panne diffère selon le type du composant défaillant. On distingue les différents cas suivants :

- panne du GS.
- panne d'un NM.
- panne d'un processus maître.
- panne d'un processus esclave.

La panne d'un NM est automatiquement détectée par le GS et nécessite simplement la réintégration du nœud défaillant (redémarrage) ou sa suppression définitive du pool de nœuds MARS (panne de longue de durée ou réquisition de la machine) en attendant sa réintégration ultérieure de façon explicite par l'administrateur MARS.

La panne d'un processus esclave est automatiquement détectée par le processus maître et nécessite la restauration du travail délégué au processus esclave. Pour ce faire, le processus maître doit tout simplement garder une copie du travail envoyé au processus esclave jusqu'à ce que ce dernier informe le processus maître que le travail a été complètement traité (voir chapitre suivant).

La panne d'un GS ou d'un processus maître sont beaucoup plus difficiles à traiter. Dans l'état actuel des choses, la panne du GS induit l'arrêt du système MARS et de toutes les applications qui s'exécutent. La panne d'un processus maître provoque la terminaison prématurée de l'application. La panne d'un GS est résolue par un mécanisme d'élection et la panne d'un processus maître par un mécanisme de checkpointing [KTG97].

Le système MARS est auto-reconfigurable et donc insensible aux pannes :

- Récupération et réintégration systématique des machines perdues (celles qui sont dans l'état SUSPENDED).
- S'il y a impossibilité de récupération après trois tentatives, la machine devient dans l'état DELETED (inaccessible) et un mail sera envoyé à l'administrateur de MARS.

3.3 Evaluation des performances du système MARS

Cette section tente de mettre en valeur certaines caractéristiques de l'exécutif MARS telles que la réactivité, la non-intrusion et l'efficacité. Pour cela, nous avons exécuté une application de recherche tabou (voir la deuxième partie de la thèse) en relevant des statistiques permettant d'analyser le comportement de l'exécutif MARS. La plateforme utilisée pour les expérimentations est un échantillon de machines du méta-système décrit dans la section 1.6. Ci-dessous on présentera des analyses de résultats après avoir donné quelques définitions utiles.

3.3.1 Définitions

- A chaque terminaison d'un processus maître ou esclave, on récupère essentiellement les informations suivantes :
  - temps d'exécution (Run Time ou RT).
  - temps d'utilisation CPU (CPU Time ou CT).

- A chaque repli d'un processus esclave, on récupère :
  - le temps de repli (Fold Time ou FT).
- Le temps d'exécution en séquentiel d'une application MARS (Sequential Time ou ST) est évalué par le temps d'utilisation CPU de l'ensemble des processus esclaves :

$$ST = \sum_{i=1}^{nombre\ d'esclaves} CT[i]$$

- Le temps d'exécution en parallèle d'une application MARS est le temps d'exécution du processus maître (Master Time ou MT).
- Le temps de possession de machines d'une application MARS (Possession Time ou PT) définit le temps d'occupation des machines par les différents processus esclaves. Le temps de possession est donc le temps d'exécution cumulé de l'ensemble des processus esclaves :

$$PT = \sum_{i=1}^{nombre\ d'esclaves} RT[i]$$

- NbUF (Nb UnFolds) est le nombre de processus dépliés. C'est exactement le nombre de machines rajoutées dynamiquement à une application MARS vu que nous avons un processus esclave par machine pour une application donnée.
- NbF (Nb Folds) est le nombre de processus repliés. Il représente le nombre de machines retranchées dynamiquement à une application MARS. NbUF et NbF reflètent le comportement adaptatif d'une application MARS.
- Le temps total de repli des processus esclaves (Total Fold Time ou TFT) est défini par :

$$TFT = \sum_{i=1}^{NbF} FT[i]$$

- AFT (Average Fold Time) est le temps moyen d'un repli. Il est défini par :

$$AFT = \frac{\sum_{i=1}^{NbF} FT[i]}{NbF}$$

- Le speed-up S est défini comme suit :

$$S = \frac{ST}{MT}$$

- AN (Average Nodes) est la moyenne des nœuds alloués à une application MARS.
- L'efficacité E est définie comme la fraction de temps CPU utilisé par une application. Les facteurs conduisant à une perte d'efficacité sont les communications, les synchronisations, et les attentes suite à des appels à des primitives bloquantes ou au time sharing UNIX vu qu'une application MARS s'exécute en compétition avec d'autres applications. E est formulée comme suit :

$$E = \frac{ST}{PT}$$

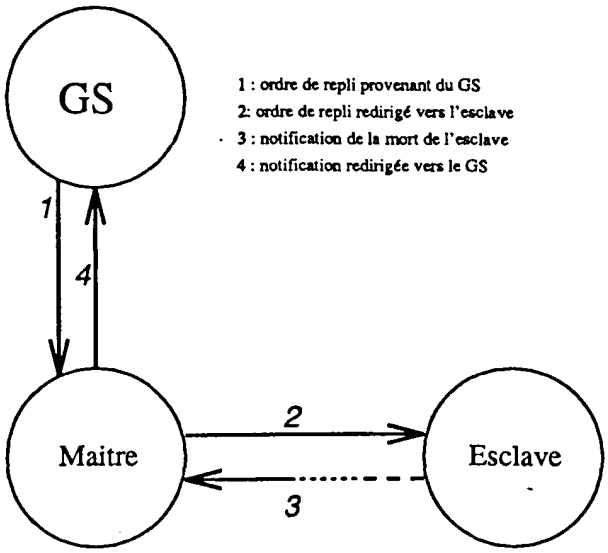


Figure 3.9 : Détermination du temps d'un repli

Le temps de repli renvoyé par le GS correspond au temps mis par les messages suivants (figure 3.9) : l'ordre de repli envoyé par le GS (1)(2), la notification au processus maître (par les démons PVM) de la mort du processus esclave (3) et la notification au GS (par le maître) de la mort du processus esclave (4).

3.3.2 Analyse des résultats

Pour illustrer le comportement de l'exécutif MARS, nous avons tracé les courbes représentant respectivement le nombre de machines utilisées dans le temps par l'application tabou durant trois exécutions (figure 3.10) et l'état des nœuds durant l'exécution de l'application (figures 3.11, 3.12 et 3.13).

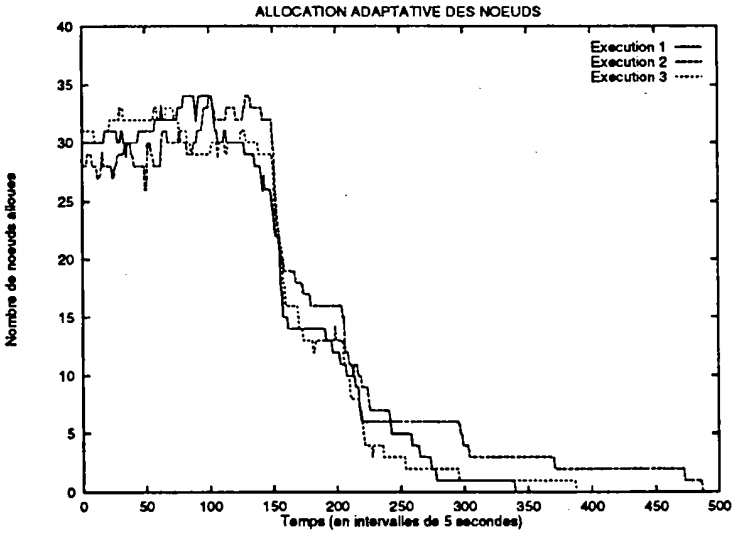


Figure 3.10 : Allocation dynamique des nœuds pour une application MARS (3 exécutions)

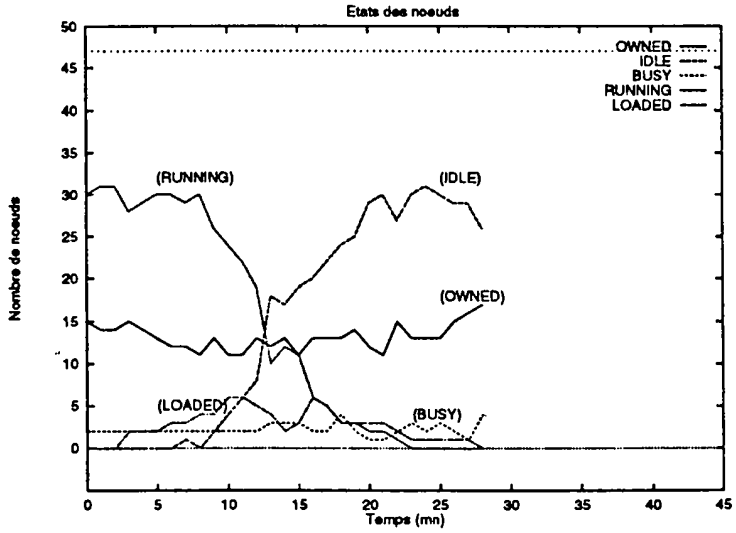


Figure 3.11 : Etats des machines pendant l'exécution d'une application MARS (exécution 1)

La figure 3.10 montre l'évolution du repli pour une application exécutée plusieurs fois à des intervalles de temps différents. Il apparaît que les courbes ne coincident pas du fait que l'état de charge du méta-système n'est pas le même d'une exécution à une autre. Ceci montre que le système MARS alloue les nœuds, non pas de façon statique, mais en fonction de l'état courant du méta-système. Par ailleurs, les courbes semblent

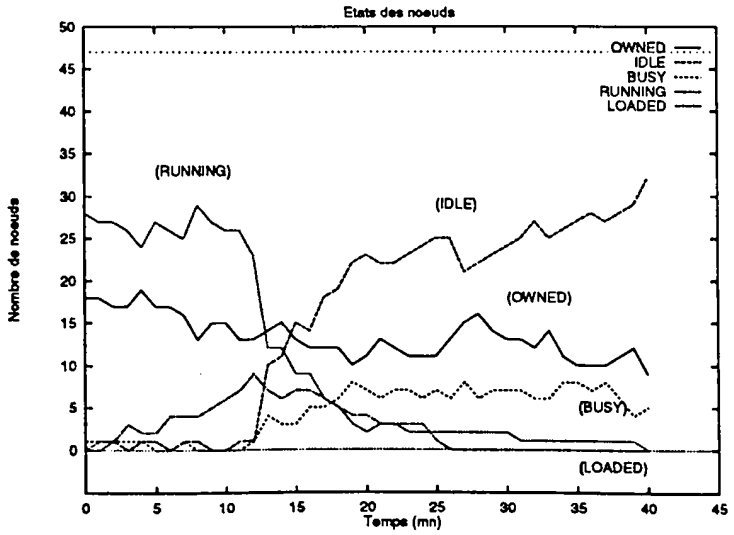


Figure 3.12 : Etats des machines pendant l'exécution d'une application MARS (exécution 2)

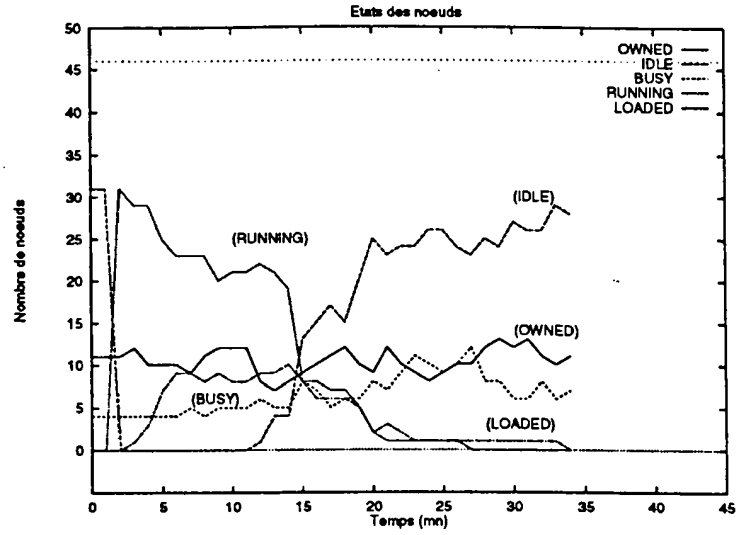


Figure 3.13 : Etats des machines pendant l'exécution d'une application MARS (exécution 3)

avoir la même allure qui présente des fluctuations de plus ou moins un certain nombre de machines à partir du début de l'exécution. Les courbes subissent une décroissance régulière à partir d'un point qui correspond à la fin de l'application, c'est-à-dire le moment à partir duquel il n'y a plus de travail à faire et que l'application libère les machines.

Les figures 3.11, 3.12 et 3.13 montrent les états des machines du méta-système tels qu'ils sont perçus par l'exécutif MARS. Ceci montre l'allocation adaptative des nœuds à l'application. On voit bien que sur les trois exécutions, tous les nœuds qui sont à l'état IDLE au moment de l'exécution de l'application passent à l'état RUNNING. Ceci se traduit par un dépli initial de l'application sur toutes les machines disponibles. Dans ces expérimentations les nœuds IDLE représentent 66% de l'ensemble des nœuds. À travers ces courbes, on remarque que l'exécutif MARS alloue correctement les nœuds à l'application durant son exécution : le nombre de nœuds IDLE est nul si l'application est en mesure de se déplier (présence de travail), et ne devient non nul que si l'application n'a plus besoin de ressources (absence de travail).

Statistiques	Exécution 1	Exécution 2	Exécution 3	Moyenne
MT (s)	1716.92	2456	1955.08	2042.66 (34 mn)
PT (s)	15211.3	15856.6	15335	15467.6 (4 h 18 mn)
ST (s)	14530.6	15070	14294.6	14631.73 (4 h 4 mn)
TFT (s)	15.4623	28.993	9.42337	17.96
NbUF	30	28	31	29.66
NbF	12	25	13	16.66
Min FT (s)	0.17408	0.236887	0.257773	0.223
Max FT (s)	2.82386	3.61876	2.14207	2.86
AFT (s)	1.29	1.16	0.72	1.05
TFT / MT	0.9 %	1.2 %	0.48 %	0.86 %
S	8.46	6.14	7.31	7.3
AN	17.32	13.64	15.13	15.36
E	95.52 %	95.04 %	93.21 %	94.59 %

Tableau 3.2 : Statistiques concernant l'exécution d'une application

Le tableau 3.2 montre quelques statistiques concernant l'exécution de l'application. En moyenne, le temps d'exécution de l'application est de 34 minutes sur un nombre de machines proche de 15. Le temps de possession est d'environ 4 heures et 18 minutes dont 4 heures et 4 minutes de temps CPU ce qui donne une efficacité de 94.59%. Le nombre moyen de déplis effectués est d'environ 30 contre 17 pour les replis. Ceci nous amène à une machine allouée par le système MARS toutes les minutes et une machine perdue par l'application toutes les 2 minutes. Le temps moyen d'un repli oscille entre 0.22 seconde et 2.86 secondes pour une moyenne d'une seconde. Rappelons que ce temps majore le temps de réactivité de l'exécutif MARS car il résulte du cumul des temps mis par un certain nombre d'événements illustrés dans la figure 3.9. Ce temps représente 0.86% du temps d'exécution de l'application et montre une bonne réactivité de l'exécutif MARS. Le speed-up d'un peu plus de 7 peut être interprété de la façon suivante : l'exécution de l'application telle qu'elle s'est déroulée sur ce méta-système équivaut à l'exécution de la même application sur un réseau « fictif » comprenant 7

machines homogènes dont le CPU est exploité à 100% du temps durant 34 minutes. La puissance d'une machine de ce réseau est bornée par la puissance de la machine la plus lente et la plus rapide du méta-système.

3.4 Comparaison avec d'autres systèmes adaptatifs

Nous avons évoqué dans le chapitre précédent qu'il y a très peu de systèmes supportant le parallélisme adaptatif. Le plus important est Piranha [GK91]. Il est conçu pour des applications maître/esclaves tout comme MARS mais il est restreint au modèle Linda (tuple-space) alors que MARS est destiné à tout environnement à échange de messages. Une tâche Piranha puise ses données du tuple-space, fait un certain traitement puis remet le résultat dans le tuple-space. La gestion du tuple-space est le principal engorgement du système. Les mécanismes de tolérance aux pannes ne sont pas fournis ce qui engendre des problèmes de robustesse vis-à-vis des pannes qui sont fréquentes dans les réseaux de stations de travail.

WoDi/CARMI est un autre ordonnanceur adaptatif conçu au dessus de PVM et Condor. Il supporte des applications de type SPMD mais n'est pas multi-utilisateurs, ce qui peut engendrer des conflits d'allocation de ressources entre applications appartenant à différents utilisateurs. De plus, dès qu'un nœud devient réquisitionné, le processus qui s'exécute sur ce nœud est tué ce qui engendre une perte de temps machine. CARMI utilise des mécanismes de checkpointing pour réaliser la migration de processus qui sont inopérants dans un environnement hétérogène.

Le tableau 3.3 met en valeur les caractéristiques principales des environnements Piranha et CARMI en comparaison avec le système MARS.

3.5 Conclusion

Le premier chapitre nous a révélé la capacité potentielle importante d'un méta-système et les fluctuations rapides de la charge des nœuds le composant. Par conséquent, toute tentative d'exploitation du temps machine inutilisé implique forcément une exécution *discontinue* si l'on veut respecter les critères de disponibilité des machines. La prise en charge d'un tel modèle d'exécution par les utilisateurs est une tâche fastidieuse qui nécessite un certain degré de transparence. Pour répondre à ce besoin, nous avons été amené à concevoir l'environnement MARS.

Contrairement à la plupart des environnements existants, MARS permet d'ajuster dynamiquement le degré de parallélisme des applications en fonction de la disponibilité des machines. MARS prend en compte les caractéristiques importantes d'un méta-système à savoir l'hétérogénéité et la fréquence élevée des pannes. MARS fournit un nouveau modèle de programmation parallèle appelé *programmation parallèle adaptative* s'appuyant sur des mécanismes d'interruption, de sauvegarde et de reprise de travail. Pour bien

	MARS	CARMI	Piranha
Style de programmation	maître/esclaves	maître/esclaves	maître/esclaves
Modèle de programmation	SPMD	SPMD	SPMD
Processus maître	Transitoire	Transitoire	Persistant
Processus esclaves	Transitoires	Transitoires	Transitoires
Bibliothèque de communication	PVM	PVM	Linda
Support d'exécution multi-threadé	Oui	Non	Non
Spécification des ressources	Non	Oui	Non
Gestion du repli	Diminution de la priorité UNIX	Suspension puis arrêt	Arrêt après délai
Sauvegarde du travail lors du repli	Oui	Non	Oui
Réintégration systématique	Oui	Non	Non
Tolérance aux pannes	Oui	Oui	Non
Hétérogénéité	Oui	Oui	Non
Ordonnancement	multi-applications time sharing	mono-application	multi-applications gang scheduling
Interface graphique	Oui	Non	Non

Tableau 3.3 : Tableau comparatif des environnements adaptatifs

exploiter le modèle MARS, des méthodologies de développement d'applications sont nécessaires et feront l'objet du chapitre suivant.

Deuxième partie

Applications parallèles adaptatives

Chapitre 4

Méthodologie de développement d'applications parallèles adaptatives

Contrairement à la première partie de la thèse plus liée à l'aspect système, cette deuxième partie concerne plutôt l'aspect applicatif. Avant d'aborder les différentes applications développées sous le système MARS, nous avons jugé utile de présenter dans un premier temps l'aspect méthodologique qui est en quelque sorte un pont reliant les deux parties de la thèse. En effet, tous les événements liés à l'adaptabilité aux changements d'état du méta-système sont produits et contrôlés de façon transparente par l'exécutif MARS. Par contre, les actions à entreprendre suite à ces événements doivent être spécifiées au niveau des applications car elles dépendent généralement de celles-ci.

La plupart des applications parallèles non-adaptatives et semi-adaptatives étaient destinées à s'exécuter sur des architectures homogènes dans un contexte mono-utilisateur et mono-application. Cela permettait de résoudre plus ou moins facilement les problèmes classiques liés au parallélisme (granularité du travail, ordonnancement des travaux et équilibrage de charge, communication/synchronisation, terminaison, anomalies d'accélération, etc.). Les applications parallèles adaptatives quant à elles sont destinées à s'exécuter sur des plate-formes hétérogènes dans un contexte multi-utilisateur et multi-application. En plus des problèmes liés au parallélisme sus-cités, elles intègrent un élément nouveau qui est le repli.

À travers ce chapitre, nous allons montrer comment aborder l'essentiel de ces problèmes lors du développement d'applications à caractère adaptatif sous le système MARS. Nous allons donc répondre aux deux questions suivantes : comment concevoir et réaliser une application parallèle adaptative dans le contexte MARS ? et comment générer et ordonnancer le travail à traiter ?

4.1 Parallélisation d'une application sous MARS

Nous avons vu dans la section 3.2 qu'une application MARS est une application du type maître/esclaves où les processus esclaves effectuent les mêmes tâches (modèle SPMD). Paralléliser une application sous MARS revient donc à spécifier deux modules : le module maître (*Application Manager Module* ou AMM) pour la tâche maître et le module esclave (*Worker Module* ou WM) pour les tâches esclaves[1]. L'AMM est composé principalement du code du thread serveur de travail (*work server thread*) et sera exécuté explicitement par l'utilisateur sur le nœud initiateur (figure 4.1). Le WM définit essentiellement le code du thread de traitement (*worker thread*), et sera lancé sur chaque machine disponible et retiré dans le cas contraire.

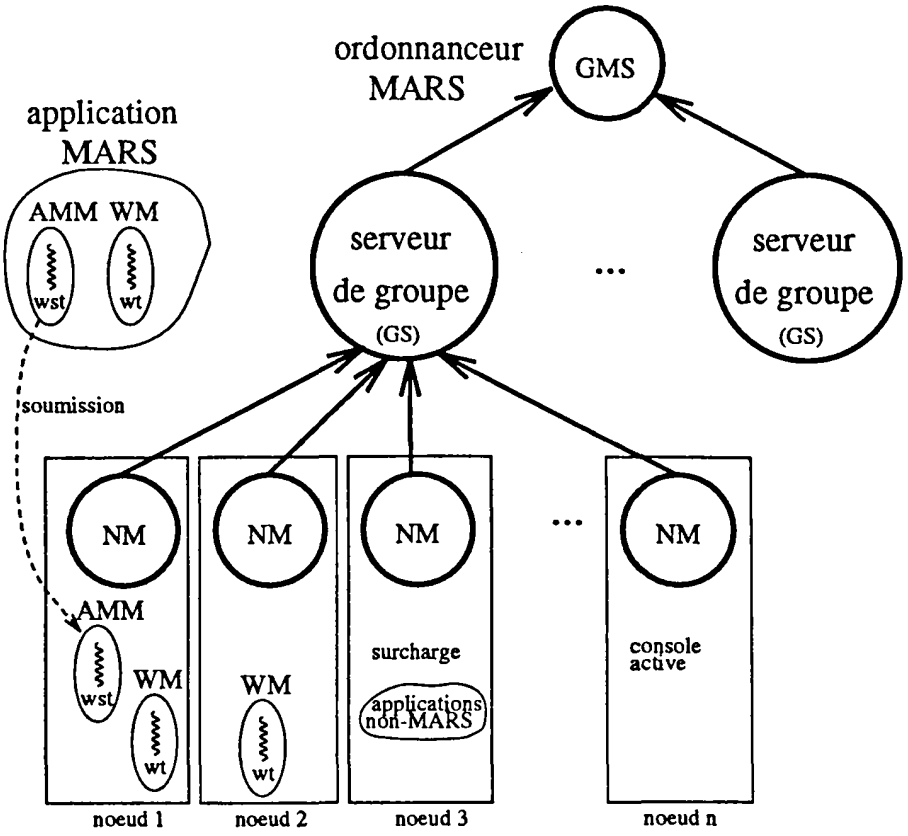


Figure 4.1 : Exécution d'une application MARS

[1]L'AMM et le WM correspondent à des processus UNIX.

4.1.1 Structure d'un module maître

L'AMM constitue l'élément central de l'application. Il possède une vision globale de l'application, détient des descripteurs (données) qui symbolisent le travail à faire et assure la coordination entre les processus esclaves. Pour garantir le bon déroulement de l'exécution de l'application, l'utilisateur doit fournir dans l'AMM un certain nombre de services et de fonctions (figure 4.2) :

- Service *get_work* (lignes AMM2 à AMM7) : Ce service est invoqué par un processus esclave pour réclamer du travail. Ceci peut être la conséquence d'une opération de dépli ou de la famine d'un processus esclave déjà existant.
- Service *put_back_work* (lignes AMM9 à AMM11) : Ce service est invoqué par un processus esclave au moment du repli suite à une indisponibilité de la machine sur laquelle il s'exécute. Il permet de récupérer le travail inachevé du processus esclave en vue de le traiter ultérieurement par un autre processus esclave.
- Service *results* (lignes AMM13 à AMM15) : Ce service est invoqué par tout processus esclave ayant terminé le travail qui lui a été délégué.
- Fonction *fault* (lignes AMM19 à AMM22) : Cette fonction est appelée automatiquement par l'exécutif MARS en vue de traiter la mort d'un processus esclave dûe à une cause autre que le repli.
- Fonction *term* (lignes AMM26 à AMM33) : Cette fonction est appelée automatiquement par l'exécutif MARS à chaque fois que le nombre d'esclaves atteint la valeur 0. Elle permet entre autres de détecter la terminaison de l'application.

4.1.2 Structure d'un module esclave

Le WM doit être conçu de telle sorte que les processus esclaves demandent continuellement du travail, renvoient les résultats finaux en cas de fin de traitement et les résultats intermédiaires en cas de repli. L'utilisateur doit fournir les éléments suivants (figure 4.3) :

- Fonction de traitement (lignes WM12 à WM31) : Cette fonction constitue le code du worker thread. Ce dernier exécute des actions répétitives de demande de travail, traitement et renvoi de résultats. Il s'arrête dès qu'il n'y a plus de travail à faire.
- Fonction de repli (lignes WM2 à WM9) : Cette fonction doit être positionnée avant de commencer le traitement. Elle est appelée automatiquement par l'exécutif MARS en vue de replier le processus esclave. Elle sert essentiellement à ressortir le travail inachevé et à l'envoyer au processus maître avant que le processus esclave ne soit tué.

```
AMM1  : /* LRPC_GET_WORK & LRPC_PUT_BACK_WORK constituent le "WST" */
AMM2  : LRPC_SERVICE(LRPC_GET_WORK)
AMM3  : if (travail en attente)
AMM4  :     /* Envoyer du travail : plusieurs stratégies */
AMM5  : else
AMM6  :     /* Plus de travail ou pas de travail momentanément */
AMM7  : END_SERVICE(LRPC_GET_WORK)
AMM8  :
AMM9  : LRPC_SERVICE(LRPC_PUT_BACK_WORK)
AMM10 : /* Récupérer le travail non encore terminé */
AMM11 : END_SERVICE(LRPC_PUT_BACK_WORK)
AMM12 :
AMM13 : LRPC_SERVICE(LRPC_RESULTS)
AMM14 : /* Retour de résultats : travail traité */
AMM15 : END_SERVICE(LRPC_RESULTS)
AMM16 :
AMM17 : /* Détection de la mort d'un esclave pour la tolérance aux */
AMM18 : /* pannes. Fonction automatiquement appelée par MARS */
AMM19 : void fault()
AMM20 : {
AMM21 : /* Gestion de la panne */
AMM22 : }
AMM23 :
AMM24 : /* Détection de la terminaison lorsque le nombre d'esclaves */
AMM25 : /* atteint 0. Fonction automatiquement appelée par MARS */
AMM26 : void term()
AMM27 : {
AMM28 : if (plus de travail) {
AMM29 :     tfprintf(stderr, "L'application est terminée ...\n");
AMM30 :     mars_exit();
AMM31 :   }
AMM32 : /* Attendre la disponibilité de resources */
AMM33 : }
AMM34 :
AMM35 : int main(int argc, char **argv)
AMM36 : {
AMM37 : /* Enrolement dans le système MARS */
AMM38 : mars_init(MASTER, ...);
AMM39 :
AMM40 : /* Positionnement des fonctions fault() et term() */
AMM41 : mars_setworkerfault_func(fault);
AMM42 : mars_stestterm_func(term);
AMM43 :
AMM44 : /* Création d'un nombre quelconque d'esclaves */
AMM45 : NbWorkers = mars_spawn("worker", ..., -1, -1);
AMM46 :
AMM47 : mars_waitexit();
AMM48 : }
```

Figure 4.2 : L'AMM pour une application MARS.

```
WM1  : /* Fonction utilisée automatiquement au moment du repli */
WM2  : void cleanup(any_t arg)
WM3  : {
WM4  : /* Envoyer le travail non achevé au processus maitre */
WM5  : ASYNC_LRPC(Master_tid, LRPC_PUT_BACK_WORK, ...);
WM7  :
WM8  : /* L'esclave termine à ce niveau */
WM9  : }
WM10 :
WM11 : /* Ceci est le "Worker Thread" */
WM12 : any_t wt(any_t arg)
WM13 : {
WM14 : /* Positionnement de la fonction cleanup() */
WM15 : mars_sputbackwork_func(cleanup, NULL);
WM16 :
WM17 : for (;;) {
WM18 :      /* Demander du travail de la part du maitre */
WM19 :      LRPC(Master_tid, LRPC_GET_WORK, ...);
WM20 :
WM21 :      if (plus de travail)
WM22 :          mars_exit();
WM23 :
WM24 :      if (pas de travail momentanément)
WM25 :          /* Réitérer la demande plus tard */
WM26 :
WM27 :      /* Traitement */
WM28 :
WM29 :      /* Renvoi de résultats */
WM30 :      ASYNC_LRPC(Master_tid, LRPC_RESULTS, ...);
WM31 : }
WM32 :
WM33 : int main(int argc, char **argv)
WM34 : {
WM35 : /* Enrolement dans le système MARS */
WM36 : mars_init(WORKER, ...);
WM37 :
WM38 : /* Créer le "worker thread" */
WM39 : mars_startworkerthread(wt, NULL);
WM40 :
WM41 : mars_waitexit();
WM42 : }
```

Figure 4.3 : Le WM pour une application MARS.

4.2 Comportement d'une application MARS durant l'exécution

4.2.1 Au moment de la soumission

Au moment de la soumission d'une application MARS, l'utilisateur exécute l'AMM sur le nœud initiateur. Ceci se traduit, comme c'est toujours le cas avec PM^2, par la création de deux threads. L'un sera chargé des communications et l'autre sera associé à la fonction *main()*. Le système MARS ne prendra connaissance de la présence de l'AMM qu'après enrôlement (AMM38). L'enrôlement se traduit par l'envoi au GS de certaines informations concernant ce processus (pid, nom de la machine, nom de login de l'utilisateur, etc.). Le thread *main* doit ensuite positionner les fonctions de traitement de la panne d'un esclave (AMM41) et de terminaison (AMM42) avant de faire appel à la primitive *mars_spawn()*. L'utilisateur doit alors donner le nom de l'exécutable correspondant au WM ainsi qu'une limite inférieure et une limite supérieure sur le nombre de WMs en exécution. Si aucune contrainte n'est imposée sur le nombre de processus esclaves, le joker -1 peut être utilisé. Dans ce cas, c'est l'exécutif MARS qui gère le nombre d'esclaves qui va osciller entre 0 et le nombre maximum de nœuds présents dans le pool MARS.

La bibliothèque MARS ne fournit aucune primitive de création explicite de processus. L'appel de la primitive *mars_spawn()* ne détermine que le moment à partir duquel l'exécution adaptative doit commencer. L'exécutif MARS détermine alors le nombre de machines disponibles et les alloue à l'application. Ceci se traduit par un dépli initial de l'application sur toutes les machines disponibles. Le thread *main* n'a plus qu'à appeler la primitive *mars_waitexit()* qui permet de mettre l'AMM en attente des requêtes provenant des processus esclaves.

4.2.2 Au moment du dépli

Une fois lancée, chaque tâche esclave commence à son tour par s'enrôler dans le système MARS avant de créer le thread de traitement ou le worker thread (WM39). Dès que ce dernier prend la main, il doit avant tout positionner la fonction *cleanup()* servant à replier le processus. Le worker thread entre par la suite dans un cycle qui consiste à demander du travail, le traiter, et renvoyer les résultats en fin de traitement. Notons qu'à cause de la nature multi-threadée de PM^2, plusieurs worker threads peuvent être créés au sein d'un même processus esclave pour permettre un recouvrement du traitement avec les communications, ou une virtualisation. Par conséquent, nous n'autorisons la création que d'un seul processus esclave par machine pour une application donnée.

Le worker thread demande du travail en invoquant le service *get_work()* du processus maître (WM19). Trois éventualités peuvent être envisagées :

- **Il n'y a plus de travail à faire** (WM21 à WM22) : C'est le cas où l'application

est en phase terminale d'exécution. Le processus esclave quitte alors le système MARS et meurt.

- **Il n'y a pas de travail momentanément** (WM24 à WM25) : Dans certaines applications, les processus esclaves peuvent être amenés à générer du travail au cours du traitement. Le processus maître peut être dans une situation d'attente de réception de l'excès de travail qui doit être renvoyé par les processus esclaves (voir les applications du chapitre 6). Dans ce cas, au lieu de mourir faute de travail, un processus esclave peut attendre la disponibilité de travail. Pour cela, il doit réitérer sa demande après un délai d'attente ou attendre qu'il soit réveillé par le processus maître.

- **Il y a du travail à faire** (WM27 à WM30) : Dans ce cas le processus esclave entame le traitement et renvoie éventuellement les résultats en fin traitement. Dans le cas où il n'y a aucun résultat à renvoyer, le processus esclave informe tout simplement le processus maître que le traitement est achevé pour que ce dernier puisse mettre à jour ses structures de données.

4.2.3 Au moment du repli

Dès qu'une machine devient indisponible, le processus maître est informé par le GS pour retirer son processus esclave qui s'exécute dessus. Le processus maître redirige l'ordre de repli émanant du GS vers le processus esclave de façon transparente (figure 3.8). L'arrivée de l'événement de repli déclenche automatiquement l'exécution de la fonction *cleanup()*. Cette fonction sert essentiellement à renvoyer au maître les données caractérisant le travail partiel afin que celles-ci puissent être reprises et traitées par une autre tâche esclave.

Cependant, on peut être dans des situations où il y aurait des inconsistences vis-à-vis du repli. Ces inconsistences sont explicitées et résolues dans ce qui suit :

- **Repli en l'absence de travail** : comme l'événement de repli peut arriver à tout instant, la fonction *cleanup()* peut être invoquée pour envoyer le travail intermédiaire alors que le processus esclave ne dispose d'aucun travail. Cette situation peut apparaître avant de réclamer du travail (juste avant l'exécution de la ligne WM19) et ce, de deux façons différentes :

  - le processus esclave vient d'être créé : dans ce cas un travail non existant sera renvoyé au processus maître.
  - le processus esclave vient de terminer un traitement et de renvoyer les résultats : dans ce cas, un travail déjà traité sera renvoyé de nouveau au processus maître (dédoublement).

  Pour résoudre ce problème, nous avons mis en place une primitive *mars_workavailable()* qui admet en paramètre une valeur booléenne signifiant

la présence ou l'absence de travail au niveau du processus esclave. Cette primitive doit être appelée avec la valeur VRAI juste après la réception de travail et avec la valeur FAUX (valeur par défaut au moment de la création d'un processus esclave) juste après l'envoi des résultats.

- **Repli en présence d'incohérences de données :** même si l'événement de repli parvient alors que le processus esclave dispose d'un travail, on peut être dans une situation où on renvoie des données intermédiaires alors qu'elles sont en train d'être modifiées. Par conséquent, les données peuvent être dans un état incohérent.

  Pour résoudre ce problème, nous avons mis en place une primitive *mars_lockfold()* qui permet de bloquer l'événement de repli afin de modifier les données de façon cohérente. Une primitive duale *mars_unlockfold()* permet la prise en compte de l'événement de repli. Ces deux primitives utilisées conjointement permettent de réaliser des sections critiques durant le traitement. Cependant, la prise en compte du repli peut être tardive ou un utilisateur mal intentionné peut contourner le problème en bloquant indéfiniment le repli et donc risquer de gêner les autres utilisateurs de la machine. Par conséquent, la priorité UNIX d'un processus esclave est automatiquement diminuée dès l'arrivée de l'événement de repli.

Malgré la réduction de la priorité UNIX d'un processus esclave pour lui laisser le temps d'effectuer le repli, le problème de persistence du processus esclave en cas de repli peut toujours se poser même en utilisant conjointement les deux primitives *mars_lockfold()* et *mars_unlockfold()*. En général, un traitement consiste à faire des cycles. Si un cycle complet constitue une section critique, il peut s'exécuter pendant une durée suffisamment longue pour retarder la prise en compte du repli à la fin du cycle et par conséquent porter préjudice aux utilisateurs interactifs. Une solution à ce problème consiste, non pas à travailler directement sur les données à envoyer au moment du repli, mais à travailler sur une copie de celles-ci et de faire des mises à jour périodiques. Une autre solution consiste à découper le traitement correspondant à la section critique en plusieurs étapes dont chacune est exécutée de façon protégée contre le repli tout en mémorisant l'étape courante. Le temps d'exécution d'une étape ne doit pas être pénalisant vis-à-vis du repli.

4.2.4 Au moment d'une panne d'un processus esclave

Lorsqu'un processus esclave meurt à cause d'une panne (redémarrage de la machine ou autre), le système MARS remonte l'événement au niveau applicatif en appelant la fonction *fault()* préalablement positionnée. Le système MARS fournit à cette fonction l'identificateur de l'esclave défaillant afin de retrouver le travail qui lui a été délégué. Une fois ce travail repéré, il sera remis dans la liste des travaux en attente pour être traité de nouveau (voir section 4.3.3). Ceci assure une parfaite cohérance de l'application.

4.2.5 Au moment de la terminaison

À chaque fois que le nombre de processus esclaves atteint la valeur 0, le processus maître en est informé par le système MARS en appelant automatiquement la fonction *term()*. S'il ne reste plus de travail à faire, l'application est terminée sinon cela veut dire qu'il n'y a plus de ressources disponibles. Le processus maître doit alors attendre la disponibilité de machines et en profiter pour faire certains traitements tels que la réorganisation de l'espace de travail ou le checkpointing.

4.3 Gestion du travail dans un environnement adaptatif

Une fois l'AMM et le WM réalisés et la gestion du repli définie, il reste à régler le problème de gestion du travail. Ceci se traduit par les réponses aux deux questions suivantes : comment définir des travaux élémentaires ? et comment les ordonnancer ?

4.3.1 Découpage de l'espace de données en unités de travail : gestion de la granularité

Un travail élémentaire ou unité de travail constitue la granularité (ou grain) d'une application. Il est définie comme étant la taille du travail effectivement traité par un processus esclave entre deux appels au service **get_work()**. La granularité peut être spatiale ou temporelle. Une granularité spatiale définit la quantité de données à traiter tandis que la granularité temporelle définit le quantum de temps durant lequel les données doivent être traitées. La granularité temporelle est bien adaptée aux applications s'exécutant sur des plateformes opportunistes hétérogènes.

La granularité peut être fine, moyenne ou grosse. Sa détermination dépend de la plateforme utilisée et du type d'application en question. En effet, on distingue différentes classes d'applications selon la connaissance préalable de l'espace de données et de la façon de le découper (figure 4.4) :

L'espace de données d'une application peut être connu à priori comme c'est le cas dans une multiplication de matrices, ou généré dynamiquement durant l'exécution de celle-ci comme dans un parcours d'arbre de recherche par exemple. En général, le découpage de l'espace de données est assuré par le processus maître lorsqu'il est connu à priori, et par les processus esclaves lorsqu'il est généré dynamiquement. Le partitionnement de l'espace de données peut se faire de façon statique ou dynamique, homogène ou hétérogène.

Dans le partitionnement statique, la granularité est fixée en dehors de toute information d'état du système et de vitesse relative des machines. Cette situation est bien adaptée aux plateformes homogènes dédiées, mono-application et mono-utilisateur. L'utilisation d'une stratégie de partitionnement statique homogène dans une application destinée à s'exécuter dans un milieu opportuniste peut entraîner une dégradation des perform-

ances. En effet, la granularité peut paraître fine pour les machines les plus rapides ou les moins chargées, et moyenne voire grosse pour les machines les plus lentes ou chargées. Dans le premier cas, on peut observer des communications plus fréquentes pour demander continuellement du travail. Dans le second cas, on peut observer une exécution plus lente de l'application et une mauvaise utilisation des ressources. De plus, du fait que le nombre d'unités de travail est fixe, l'application peut ne pas bénéficier pleinement des ressources si ce nombre est nettement inférieur au nombre de machines disponibles.

Dans le partitionnement dynamique, la granularité peut être déterminée et ajustée dynamiquement en fonction de l'état du système et des vitesses relatives des machines.

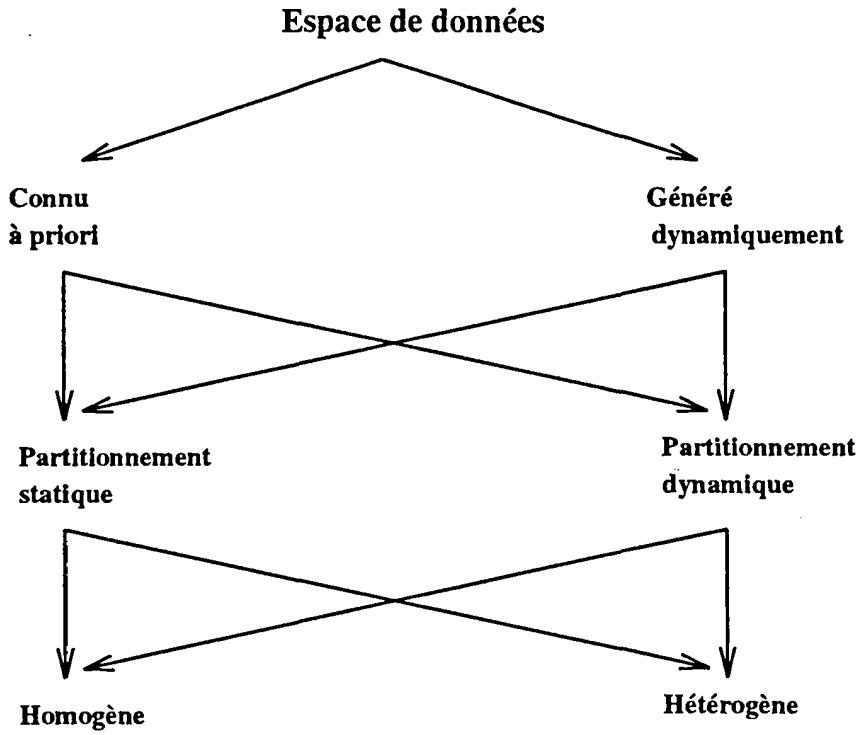


Figure 4.4 : Gestion de la granularité spatiale

À travers cette analyse, nous pouvons dire que la détermination et la gestion de la granularité a un impact direct sur l'efficacité de l'exécution d'une application [Cun94] [Mel97]. L'efficacité est limitée par le facteur

$$\frac{t_{get\_work} + t_{grain}}{t_{get\_work}}$$

où t_{get_work} est le temps mis pour invoquer le service *get_ work* et acquérir du travail, et t_{grain} est le temps nécessaire pour traiter un travail de granularité *grain*. Si la granularité est très fine alors t_{grain} a tendance à être proche ou inférieur à t_{get_work} et par conséquent on perd en efficacité. Pour améliorer l'efficacité, on peut réduire t_{get_work} et augmenter t_{grain}. Cependant, t_{grain} ne peut être augmenté de façon incontrôlable car la granularité peut devenir grosse et diminue par conséquent le degré de parallélisme. Cette diminution du degré de parallélisme ne permet pas d'exploiter toutes les machines disponibles et conduit généralement à un déséquilibre de charge intra-application.

Le temps d'acquisition du travail t_{get_work} peut être divisé en deux parties : le temps d'envoi et de réception de messages $t_{transmission}$, et le temps $t_{accès}$ d'accès aux données par le service *get_ work*. Alors que $t_{transmission}$ dépend du support et du protocole de communication utilisés, $t_{accès}$ peut être réduit en utilisant des structures de données adéquates pour la gestion du travail.

Une granularité grosse ralentit donc une application en fin de parcours et une granularité fine provoque trop de communications mais permet dans certaines applications d'éviter de gérer le repli[2].

4.3.2 Ordonnancement des unités de travail

Une fois qu'une méthode de gestion de la granularité est adoptée, il reste à définir une stratégie pour l'ordonnancement intra-application des unités de travail. Le problème consiste à déterminer quelle est l'unité de travail, parmi celles en attente, qu'il faut envoyer à un processus esclave demandant du travail. Ceci est lié à la présence ou non de graphe de dépendances entre les unités de travail. En effet, les processus esclaves peuvent être complètement indépendants ou coopératifs (figure 4.5). Les processus esclaves coopératifs peuvent être communicants ou non. Dans le cas où les processus esclaves ne sont pas communicants, ils s'échangent tout simplement des informations via le processus maître.

L'absence de communications entre processus esclaves et de dépendances entre unités de travail rend les applications très facilement implémentables et parfaitement adaptables à l'environnement. En général, aucune stratégie d'ordonnancement précise n'est requise. Les unités de travail sont banalisées et peuvent être envoyées aux processus esclaves dans un ordre quelconque. Toutefois, dans certaines applications il est préférable de faire un choix basé sur la qualité du travail à envoyer. Par exemple, il est préférable d'envoyer les unités de travail de forte granularité en priorité pour avoir un meilleur comportement de l'application en fin de parcours.

La figure 4.6 montre comment sont gérées les unités de travail dans une application MARS. L'AMM dispose de deux files d'attente contenant respectivement les unités de travail en attente de traitement et les unités de travail en cours de traitement. Dans cet

[2]Vu que la granularité est fine, le traitement correspondant est très court et la prise en compte de l'événement de repli en l'absence de travail est quasiment immédiate.

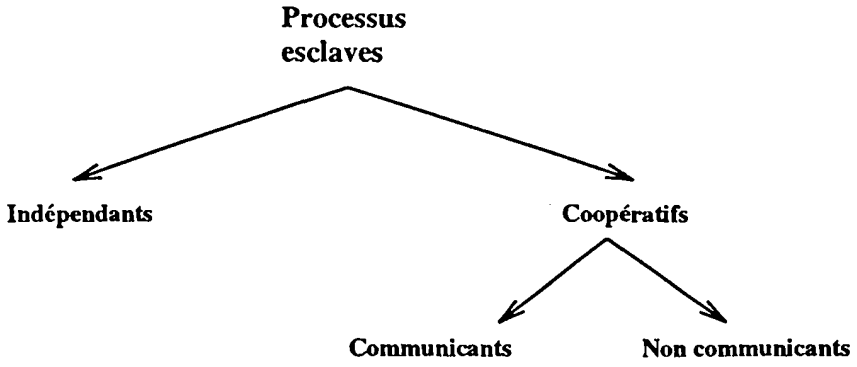


Figure 4.5 : Nature des processus dans une application parallèle

exemple, l'application possède 5 unités de travail dont 3 sont en attente et 2 sont en train d'être traitées par les deux esclaves WM1 et WM2. Dans le cas où une machine devient disponible et qu'un nouveau processus esclave est créé (WM3), ce dernier envoie un LRPC à l'AMM pour réclamer du travail. Un thread sera créé pour effectuer le service *get_work* qui consiste à déterminer le travail à envoyer. L'utilisation d'une liste chaînée permet de choisir une parmi les unités de travail éligibles et donc d'implémenter de façon aisée une stratégie d'ordonnancement intra-application. Une fois l'unité de travail sélectionnée, elle sera envoyée au processus demandeur en prenant soin de garder une copie de celle-ci dans la liste des travaux actifs.

Pour améliorer les performances, les files d'attente peuvent être matérialisées par des structures de données parallèles (actives en passant par des threads ou passives à accès concurrent) [LC96].

Le problème se complique dans le cas de dépendances entre unités de travail. L'application doit alors gérer un graphe de dépendance qui peut être statique ou dynamique et ordonnancer uniquement les unités de travail éligibles [MTP98]. Si des processus esclaves se voient bloqués car ils dépendent des unités de travail repliées, le processus maître doit débloquer la situation en repliant des processus esclaves bloqués et en les dépliant avec les unités de travail dont elles dépendent.

4.3.3 Gestion de la panne des processus esclaves

La liste des travaux actifs joue un rôle très important pour la survie de l'application. En effet, lors de la panne d'un esclave, la fonction de traitement de la panne préalablement positionnée par l'utilisateur utilise l'identificateur du processus défaillant envoyé par l'exécutif MARS afin de retrouver, dans la liste des travaux actifs, l'unité de travail déléguée au processus défaillant. Le traitement de la panne consiste tout simplement à réinsérer, dans la liste des travaux en attente, l'unité de travail retrouvée. Dans le

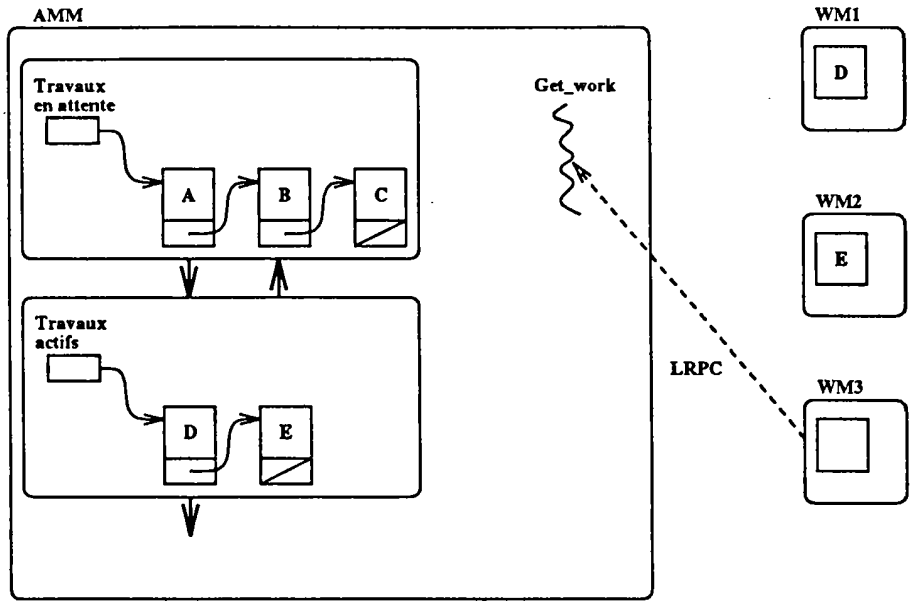


Figure 4.6 : Ordonnancement des unités de travail

cas où un processus esclave ne tombe pas en panne et qu'il réussit à traiter les données qui lui ont été déléguées, il doit renvoyer un message au processus maître pour que ce dernier supprime de la liste des travaux actifs la copie de travail correspondante.

4.4 Exemple illustratif : recherche de nombres premiers

Dans cette section, nous allons montrer comment développer une application parallèle adaptative qui consiste à calculer le nombre de nombres premiers dans un intervalle [A, B] donné. Une solution consiste à subdiviser l'intervalle en sous-intervalles. On se trouve alors dans une situation où l'espace de données est connu à priori et où le découpage ou la génération du travail peuvent être effectués par le processus maître. Les processus esclaves reçoivent les sous-intervalles à traiter et peuvent travailler de façon complètement indépendante.

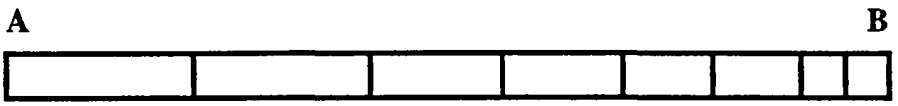
Du moment qu'une unité de travail représente un sous-intervalle à traiter, l'étape suivante consiste à déterminer les informations à renvoyer au processus maître au moment du repli. Si l'on considère qu'un processus esclave traite un sous-intervalle [a, b] de la borne min a à la borne max b, les données à renvoyer au moment du repli sont : le sous-intervalle [a', b] où a' est le nombre dont le test de primalité est en cours ($a \leq a' \leq b$), et le nombre de nombres premiers trouvés jusqu'ici par le processus esclave (ceux compris entre a et a'). Pour le test de primalité d'un nombre N, on utilise

la méthode simple qui consiste à vérifier si N n'est pas divisible par tous les nombres impairs compris entre 2 et $\sqrt{N}$.

La dernière étape consiste à déterminer comment découper l'intervalle [A, B] et comment ordonnancer les sous-intervalles. A priori, la notion de dépendance entre les unités de travail et de qualité (ou de priorité) n'existe pas. L'ordonnancement ne pose donc aucun problème dans ce cas vu que les unités de travail peuvent être traitées dans un ordre quelconque. Par contre, il faut déterminer une stratégie de découpage de l'intervalle [A, B]. On peut opter pour un partitionnement statique ou un partitionnement dynamique.

Un découpage statique consiste à définir initialement, à la compilation ou au moment de la soumission de l'application, le nombre et la taille des sous-intervalles. Dans ce cas, on opte pour un partitionnement spatial homogène ou hétérogène (découpage logarithmique par exemple, voir figure 4.7). La granularité spatiale va dépendre de la taille des sous-intervalles et influence directement l'efficacité de l'application : une grosse granularité conduit à un ralentissement de l'application en fin de parcours, et une granularité très fine ne permet pas le recouvrement des communications par le calcul. Une granularité fine à moyenne est nécessaire mais elle ne peut être déterminée en l'absence d'informations supplémentaires concernant le méta-système.

(a) Partitionnement spatial homogène



(b) Partitionnement spatial hétérogène

Figure 4.7 : Découpage statique du travail

Un découpage dynamique consiste à déterminer les unités de travail à la demande et permet, par conséquent, de prendre en considération certaines informations telles que le nombre courant de processus esclaves en exécution, la différence relative des machines utilisées, etc. Nous avons utilisé une heuristique simple qui consiste à découper l'intervalle restant proportionnellement au nombre de processus esclaves au moment de

la demande. Le facteur de découpage est donc égal à $c.NbEsclaves$ où c est une constante positive (figure 4.8). L'avantage de cette méthode réside dans le fait que le grain est important au moment de l'exécution car il y a beaucoup de travail à traiter, et qu'il diminue avec la diminution de la quantité de travail restante. Ceci permet d'éviter une perte de ressources et d'assurer une meilleure répartition du travail et donc un meilleur équilibrage de charges intra-application.

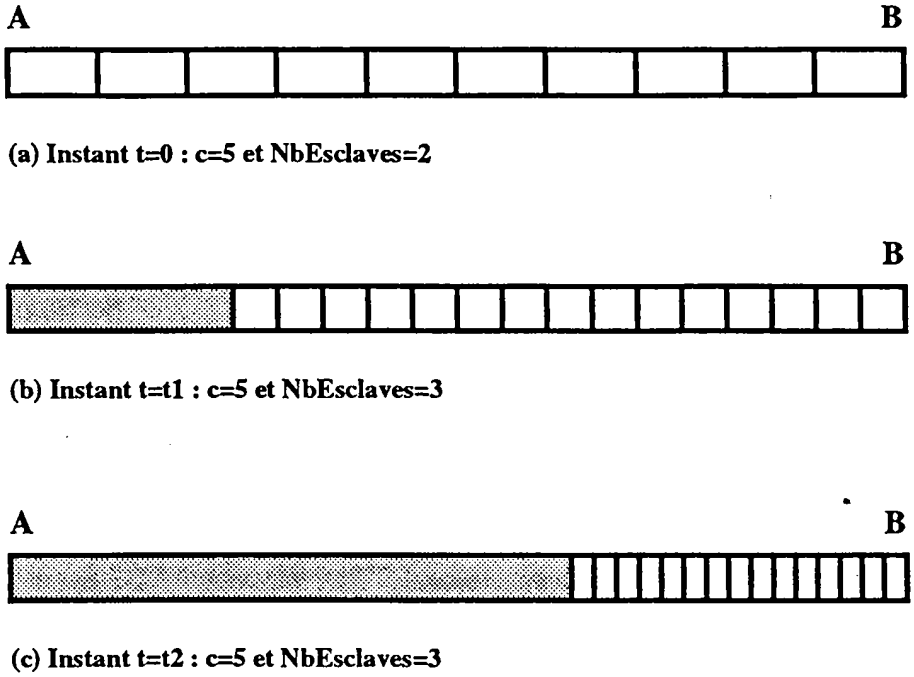


Figure 4.8 : Découpage dynamique du travail

Des portions de code de l'AMM, du WM et d'un module de gestion du travail pour cette application sont présentées en annexe C.

4.5 Conclusion

Le modèle de programmation parallèle adaptative qui découle de l'environnement MARS fait apparaître des méthodologies permettant aux applications d'utiliser au mieux un environnement opportuniste. Pour qu'une application puisse bénéficier du modèle MARS, elle doit tenir compte d'un élément nouveau qui est le repli. Définir le repli consiste à déterminer l'état du travail partiel à renvoyer au processus maître. En général, cela dépend de l'application en question.

Outre le repli, des problèmes liés à la gestion de la granularité et l'ordonnancement des unités de travail doivent être pris en compte pour permettre un meilleur comportement de l'application et une utilisation efficace des ressources. Ainsi, les applications existentes ont été regroupées dans des classes d'applications selon des critères liés à la connaissance préalable de l'espace de données, l'existence d'un graphe de précédence, etc. Nous avons présenté un exemple simple d'application où les processus esclaves sont complètement indépendants et où l'espace de données est connu à priori. Dans les chapitres suivants, nous allons appuyer nos démarches méthodologiques par des applications réelles appartenant à différentes classes d'applications. Notons que nous avons mis en place un « squelette » d'application contenant tous les éléments essentiels liés à l'adaptabilité, et un module de génération et d'ordonnancement du travail. Le but étant de permettre de simplifier le développement d'une application sous MARS.

Chapitre 5

Méta-heuristiques parallèles adaptatives

La plupart des problèmes d'optimisation combinatoire rencontrés dans divers domaines d'application tels que la robotique, la vision, la mécanique et la médecine sont NP-complets. Les algorithmes permettant de résoudre ces problèmes peuvent se scinder en deux parties. D'une part les méthodes exactes et d'autre part les méthodes heuristiques (figure 5.1). Les méthodes exactes, qui feront l'objet du chapitre suivant, permettent de trouver une solution optimale mais leur applicabilité est restreinte aux problèmes de « petite taille ». Le recours aux méthodes heuristiques est souvent nécessaire pour résoudre des problèmes de « grande taille ». Les méthodes heuristiques ne garantissent pas de trouver une solution optimale, mais la recherche d'une solution acceptable se fait en un temps « raisonnable ».

Plusieurs méthodes heuristiques ont été proposées dans la littérature. Elles peuvent être divisées en deux classes : d'une part les algorithmes spécifiques à un problème donné, et d'autre part les algorithmes génériques applicables à une grande variété de problèmes d'optimisation. Notre intérêt dans cette thèse porte sur la deuxième classe d'algorithmes connue sous le nom de méta-heuristiques.

5.1 Revue des principales méta-heuristiques

Les méta-heuristiques se développent à l'heure actuelle de manière considérable. Le succès de ces méthodes est dû à plusieurs facteurs comme leur facilité d'implantation, leur capacité et leur flexibilité à prendre en considération des contraintes spécifiques apparaissant dans les applications pratiques ainsi que la bonne qualité des solutions qu'elles sont capables de trouver. D'un point de vue théorique, l'usage de telle ou telle méta-

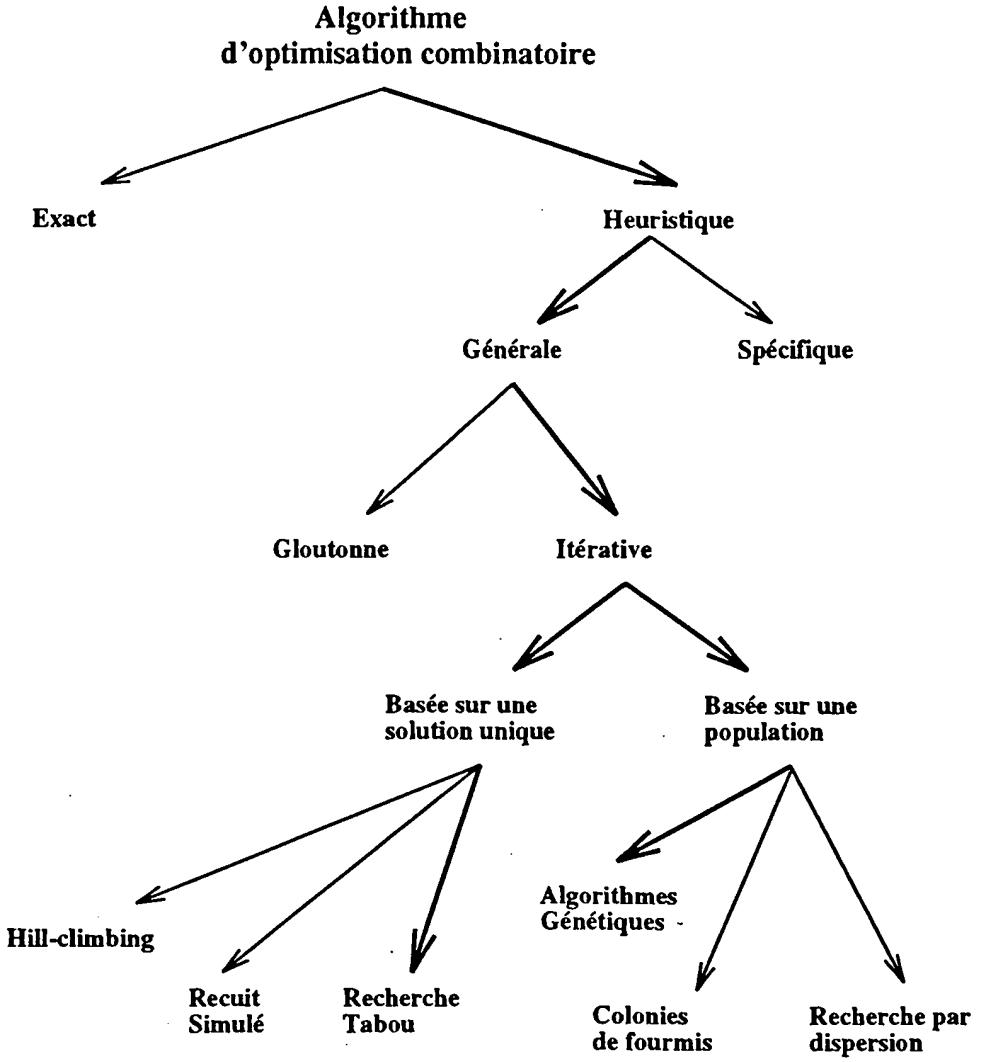


Figure 5.1 : Classification des algorithmes heuristiques d'optimisation combinatoire

heuristique n'est pas vraiment justifié et les résultats concernant l'efficacité et la convergence sont infimes. Cependant, en pratique, les performances de ces méthodes sont appréciables et sont souvent extrêmement compétitives par rapport aux heuristiques spécifiques. Dans cette course aux performances, on observe que les méthodes les plus efficaces hybrident deux (ou plusieurs) méta-heuristiques [Tal98].

Les méta-heuristiques partagent en général trois particularités :

- Elles partent d'une solution initiale générée aléatoirement dans la plupart des cas, ou construite à partir d'une heuristique. La solution initiale est généralement de

piètre qualité.

- Elles mémorisent des solutions ou des caractéristiques des solutions visitées durant le processus de recherche. Ceci constitue souvent un moyen de diriger la recherche afin de ne pas tomber dans des solutions déjà visitées ou dans un optimum local.
- Elles utilisent une procédure qui permet de créer une nouvelle solution à partir de la solution courante et des informations mémorisées. Il s'agit d'un élément essentiel dans l'évolution de la recherche car il permet l'exploration de l'espace des solutions.

Les méta-heuristiques les plus utilisées sont les algorithmes itératifs de recherche locale (« Hill-climbing ») [AHU74], le recuit simulé [KGV83], les algorithmes de recherche tabou [Glo86] et les algorithmes génétiques [Hol75]. Les algorithmes itératifs de recherche locale garantissent de trouver une solution optimale uniquement pour une fonction coût convexe. Dans le cas contraire, ils convergent généralement de façon prématurée vers un optimum local. Pour pouvoir « échapper » à des optima locaux et essayer de converger vers un optimum global, les techniques de recherche tabou et du recuit simulé peuvent être utilisées. Les solutions trouvées sont souvent meilleures, mais le prix à payer est un coût d'exécution plus important. Des méthodes récentes s'inspirant de la nature tels que les colonies de fourmis [CDM91] et les guêpes [Kob97] ont été appliquées avec succès pour certains problèmes d'optimisation. La différence fondamentale entre les différents algorithmes cités ci-dessus réside dans leur façon d'explorer l'espace de recherche et d'exploiter l'historique de celle-ci.

5.2 Recherche tabou et parallélisme

5.2.1 Principe de la recherche tabou

La recherche tabou (« Tabu Search » ou TS) est une méta-heuristique qui a été proposée par Glover [Glo86]. L'idée à la base de cette technique est de modifier localement une solution de manière itérative, tout en gardant trace des modifications pendant un certain nombre d'itérations dans le but d'éviter de réaliser les modifications inverses susceptibles de mener à des solutions déjà visitées. Certaines caractéristiques des modifications effectuées ou encore les solutions entières sont mémorisées dans des listes qui interdisent l'usage de ces modifications pour les itérations futures, d'où l'appellation de liste tabou ou mémoire à court terme.

La recherche tabou commence par la construction d'une solution initiale puis par l'amélioration de celle-ci par des modifications locales. Ces modifications n'améliorent pas forcément la solution à tous les coups mais dirigent la recherche vers une portion prometteuse de l'espace des solutions. Ainsi, à chaque itération, la recherche tabou sélectionne la meilleure solution voisine même si cette dernière est plus mauvaise que la

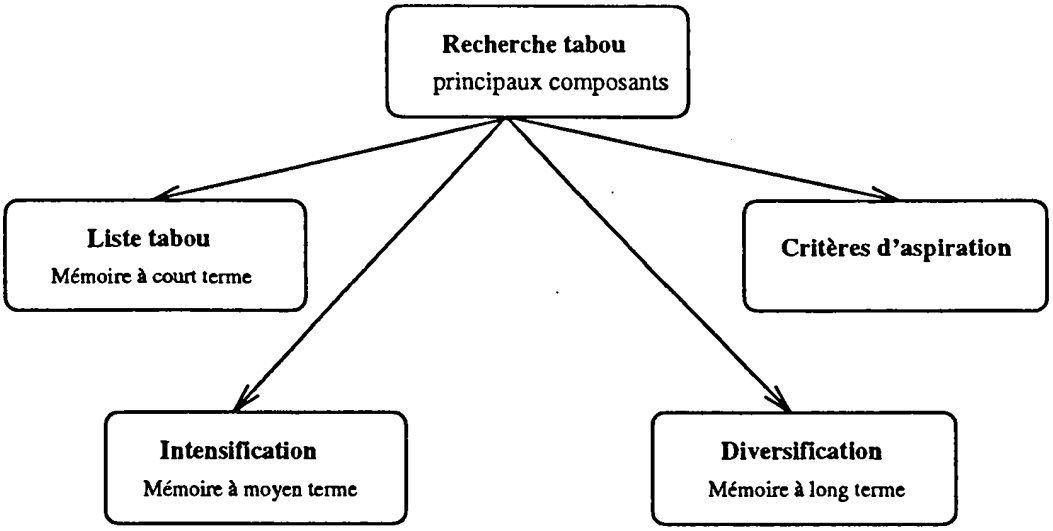


Figure 5.2 : Composants de la recherche tabou

solution courante. Ceci peut conduire à un processus cyclique qui peut être évité par l'usage de la liste tabou.

Un certain nombre de stratégies permettant de diriger la recherche et la rendre plus efficace ont également été proposées par Glover mais elles n'ont été utilisées sur le plan pratique que tardivement (figure 5.2). Les implémentations connues jusqu'alors utilisent généralement une liste tabou avec un critère d'aspiration qui consiste à prendre une solution meilleure même si celle-ci est obtenue à partir d'un mouvement interdit. L'usage d'une mémoire à long terme dans le but de diversifier le processus de recherche s'est répandu depuis. La diversification consiste à forcer l'exploration des portions non encore examinées de l'espace des solutions. Ceci se traduit par le forçage de l'utilisation d'une modification jamais élue pendant un grand nombre d'itérations [Tai91], ou par la pénalisation des modifications proportionnelles à leur fréquence d'élection [Tai93]. Une autre composante négligée dans les premières implantations de la recherche tabou mais qui prend de plus en plus d'importance est l'intensification de la recherche dans une portion jugée prometteuse de l'espace des solutions.

D'une manière formelle, un problème d'optimisation combinatoire est défini par la spécification d'un couple (X, f), où l'espace des solutions X est en ensemble discret de solutions réalisables, et la fonction objectif f est définie par $f : X \longrightarrow R$. Un voisinage N est une application $N : X \longrightarrow P(X)$, qui spécifie pour chaque solution $S \in X$ un sous-ensemble $N(S)$ de X contenant les solutions voisines de S.

D'une manière schématique et en l'absence de mécanismes d'intensification et de diver-

sification, la recherche tabou peut être formulée comme le montre la figure 5.3 avec, comme entrées, les informations suivantes [HdW89] :

- S_0 : une solution initiale.
- $|T|$: taille de la liste tabou T (statique ou dynamique).
- *nbmax* : nombre maximum d'itérations entre deux améliorations de la fonction objectif f.

La recherche tabou a obtenu de nombreux succès dans la résolution de problèmes pratiques d'optimisation dans différents domaines (gestion de ressources, conception, logistique, télécommunications, etc.). Les résultats prometteurs de son utilisation sur une variété de problèmes académiques d'optimisation (voyageur de commerce, affectation quadratique, emploi du temps, ordonnancement, etc.) ont été reportés dans la littérature [GL92]. Néanmoins, la résolution de problèmes de grande taille a conduit un nombre de chercheurs à recourir à une implémentation parallèle de la méthode de recherche tabou.

Etape 1 : Initialisation
- Choisir une solution initiale arbitraire $S_0 \in X$ $(S = S_0)$
- $nbiter = 0$ /* itération courante */
- $bestiter = 0$ /* itération où une meilleure solution a été trouvée */
- $bestsol = S_0$ /* meilleure solution trouvée */
- $T_i = \emptyset$ /* la liste tabou est initialement vide */

Etape 2 : Itération

Tant que $(nbiter - bestiter < nbmax)$ **Faire**
- $nbiter = nbiter + 1$;
- Générer un ensemble $V^* \subset N(S)$ contenant les solutions $t \in N(S)$ qui ne sont pas tabous ou qui satisfont le critère d'aspiration
- Choisir une solution S^* minimisant f dans V^*
- Mettre à jour la liste tabou T_i ;

Si $f(S^*) \leq f(bestsol)$ **Alors** $bestsol = S^*$; $bestiter = nbiter$;
- $S = S^*$;

Figure 5.3 : Un algorithme de base pour la recherche tabou

5.2.2 Recherche tabou parallèle

Plusieurs classifications des algorithmes parallèles de recherche tabou ont été proposées dans la littérature [Vos92][CTG93]. Elles sont basées sur plusieurs critères dont : nombre de solutions initiales, paramètres identiques ou différents, et stratégies de contrôle et de communication. Nous avons identifié deux catégories principales (figure 5.4) :

5.2.2.a Décomposition du domaine

Le parallélisme dans cette catégorie d'algorithmes repose essentiellement sur :

- **La décomposition de l'espace de recherche** : le problème principal est décomposé en sous-problèmes. Chacun de ces derniers est traité par un algorithme de recherche tabou [Tai93].
- **La décomposition du voisinage** : à chaque itération, la recherche de la meilleure solution voisine est effectuée en parallèle. Chaque tâche évalue une partie du voisinage [Tai91][CSK93].

Un fort degré de synchronisation est nécessaire pour l'implémentation de cette catégorie d'algorithmes.

5.2.2.b Recherches tabou multiples

Cette catégorie d'algorithmes consiste en l'exécution en parallèle de plusieurs recherches tabou. Les différentes tâches débutent par des paramètres identiques ou différents (solution initiale, taille de la liste tabou, nombre maximum d'itérations, etc.). Elles peuvent être complètement indépendantes (sans communication) [MGPO89][RR95] ou coopératives. Dans l'algorithme coopératif proposé dans [CTG93], chaque tâche effectue un certain nombre d'itérations avant de diffuser sa meilleure solution trouvée aux autres tâches. La meilleure solution reçue devient la solution initiale pour la phase suivante.

5.2.2.c Discussion

Paralléliser la décomposition de l'espace de recherche ou du voisinage est une tâche délicate étant donnée la nature séquentielle de la méthode de recherche tabou. De plus, la décomposition est souvent dépendante du problème traité. La seconde catégorie d'algorithmes parallèles est moins contraignante et donc plus générale.

La plupart des algorithmes proposés dans la littérature sont non-adaptatifs. Un exemple de cette approche est présenté dans [PR96]. Le voisinage est découpé en partitions de tailles équivalentes en fonction du nombre de processus. Ce dernier est égal au nombre

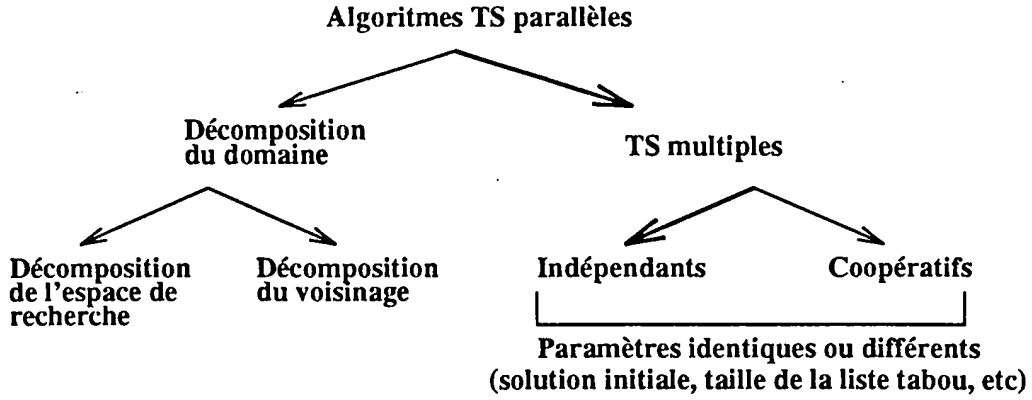


Figure 5.4 : Différentes stratégies de recherche tabou parallèle

de processeurs de la machine parallèle. Dans [CSK93], le nombre de tâches générées dépend de la taille du problème et est égal à n^2, où n est la taille du problème.

S'il existe une différence notable entre la charge ou la puissance des différentes machines (comme c'est souvent le cas dans un méta-système), le temps de recherche de l'approche non-adaptative est égal au temps d'exécution maximum sur les machines les plus chargées ou les moins puissantes. Plusieurs tâches attendent alors que les tâches les plus « paresseuses » terminent leur travail.

Pour améliorer les performances des algorithmes parallèles non-adaptatifs, l'équilibrage de charge est introduit [PR96][BGG+95]. L'équilibrage de charge est accompli dans [PR96] par une redistribution dynamique du travail entre les processeurs. Pendant la recherche, à chaque fois qu'une tâche termine son travail, elle procède à une demande de travail. L'équilibrage dynamique de charge dans un voisinage partitionné se traduit par la migration des partitions.

5.3 Recherche tabou parallèle adaptative

Dans ce qui suit, nous allons détailler l'approche utilisée pour introduire le parallélisme adaptatif dans la méthode de recherche tabou. Elle consiste en plusieurs algorithmes de recherche tabou indépendants s'exécutant en parallèle. Ceci ne demande aucune communication entre les différentes tâches. Différents paramètres ont été utilisés telle que la solution initiale et la taille de la liste tabou.

Dans cette application, le processus maître génère les unités de travail devant être traitées par les processus esclaves. Une unité de travail consiste en une solution initiale générée aléatoirement et associée à une liste tabou initialement vide. Le nombre d'unités

de travail est fixé par le programmeur et correspond au nombre maximum d'esclaves pouvant s'exécuter en parallèle. Chaque processus esclave reçoit une unité de travail de la part du processus maître, effectue une recherche tabou séquentielle, et renvoie un résultat au processus maître. Le résultat n'est autre que la meilleure solution trouvée par l'esclave. Le nombre d'itérations est le même pour tous les esclaves. Cette application possède une « grosse » granularité qui n'est pas ajustée durant l'exécution.

Le processus maître implémente donc une mémoire centrale qui possède une vision globale de la recherche. Le nombre d'esclaves initialement créés est égal au nombre de machines disponibles dans le méta-système. L'application parallèle adaptative réagit aux événements de dépli et de repli comme le montre la figure 5.5.

En cas de repli, le travail pendant d'un processus esclave est constitué de la solution courante, de la meilleure solution trouvée jusqu'ici, de l'état de la mémoire à court terme, de l'état de la mémoire à long terme et du nombre d'itérations effectuées sans amélioration de la meilleure solution. Le processus maître met à jour la meilleure solution globale de l'application si celle-ci est moins bonne que la meilleure solution locale reçue de la part d'un processus esclave.

En cas de dépli, avant d'entamer une recherche tabou, le processus esclave demande du travail au processus maître. L'unité de travail est constitué d'une solution initiale générée aléatoirement ou d'une solution intermédiaire obtenue suite à un repli. Dans ce cas, le processus esclave reçoit également l'état de la liste tabou, la mémoire à long terme et le nombre d'itérations effectuées sans amélioration de la meilleure solution. En cas d'existence de plusieurs unités de travail en attente, le processus maître peut sélectionner aléatoirement l'unité de travail à envoyer (pas de notion de priorité). Cependant, nous envoyons un travail qui n'a pas encore été traité pour éviter l'attente en fin d'application.

5.3.1 Application au problème d'affectation quadratique

Le problème d'affectation quadratique (QAP) représente une classe importante de problèmes d'optimisation combinatoire avec plusieurs applications dans différents domaines (placement, ordonnancement, synthèse d'image, conception de circuits électroniques, etc.).

Le QAP est un problème NP-dur [GJ79]. Trouver une solution ϵ-approchée est également un problème NP-dur [SG76]. De ce fait, des algorithmes exacts tel que le Branch-and-Bound (voir chapitre suivant) ne peuvent traiter que des instances de petite taille ($n < 25$) [BMCP97]. Le lecteur intéressé trouvera un état de l'art récent dans [PRW94].

5.3.1.a Formulation du problème

La première formulation du QAP a été donnée par Koopmans et Beckmann en 1957 [KB57]. Une instance du QAP peut être définie par :

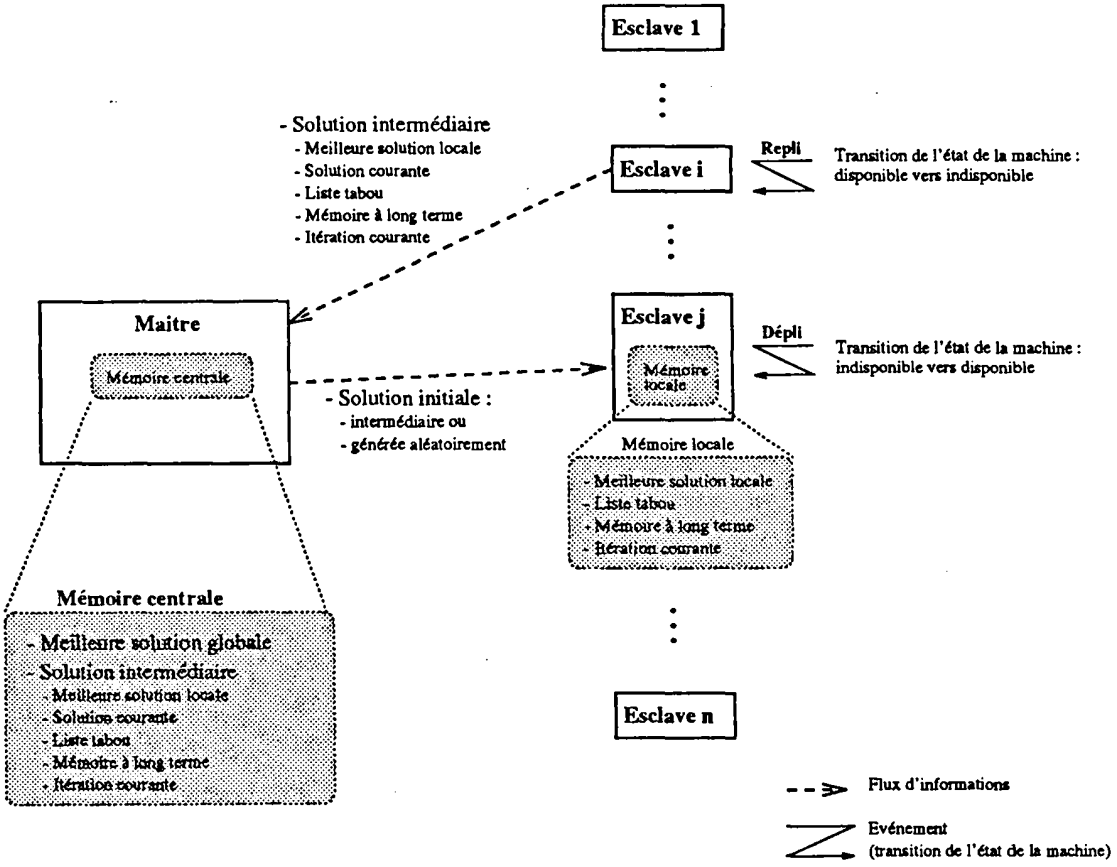


Figure 5.5 : Recherche tabou parallèle adaptative

- n, la taille de l'instance ;
- un ensemble de n objets $O = \{O_1, O_2, ..., O_n\}$;
- un ensemble de n positions $P = \{P_1, P_2, ..., P_n\}$;
- une matrice de flux C, dans laquelle chaque élément (c_{ij}) indique le flux entre les objets O_i et O_j ;
- une matrice de distance D dans laquelle chaque élément (d_{kl}) donne la distance entre les positions P_k et P_l.

Etant donnés ces éléments, le problème consiste à trouver un positionnement des objets, c'est-à-dire une bijection $U : O \longrightarrow P$, qui minimise la fonction Φ :

$$\Phi(U) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}.d_{U(i)U(j)}$$

5.3.1.b Codification du problème

Une solution au QAP peut être représentée de multiples manières. Dans cette thèse, une solution est représentée par une permutation de n entiers :

$$s = (s_1, s_2, ..., s_n)$$

dans laquelle $s_i = U(i)$ est la position de l'objet O_i.

Pour pouvoir appliquer la méthode parallèle adaptative de recherche tabou sur le QAP, on doit d'abord définir la structure du voisinage, son évaluation, la mémoire à court terme permettant d'éviter de cycler, et la mémoire à long terme servant pour les phases d'intensification et de diversification :

- **Structure du voisinage et son évaluation** : Etant donné qu'une solution est représentée par une permutation de n entiers, le voisinage est construit à partir de cette permutation en effectuant des mouvements dans lesquels deux objets de la permutation sont interchangés.

  La formule donnée dans [Tai95] permet de calculer efficacement la variation dans la fonction coût dûe à un mouvement entraînant deux objets. Par conséquent, l'évaluation du voisinage peut se faire durant $O(n^2)$ opérations au lieu de $O(n^3)$.

- **Mémoire à court terme** : La liste tabou contient les couples (i, j) d'objets qui ne peuvent pas être interchangés. La taille de la liste tabou varie entre $\frac{n}{2}$ et $\frac{3n}{2}$. Le nombre maximum d'unités de travail initialement généré est positionné à n. Chaque tâche est initialisée avec une taille de liste tabou différente allant de $\frac{n}{2}$ à $\frac{3n}{2}$ par incrément de 1.

  Le critère d'aspiration autorise un mouvement si ce dernier génère une solution meilleure que celle trouvée jusqu'ici. Le nombre total d'itérations est limité à $1000n$.

- **Mémoire à long terme** : La mémoire à long terme, qui vient en complément des informations données par la liste tabou, est basée sur la fréquence des mouvements. Une matrice $F = (f_{i,k})$ contenant les mouvements fréquemment effectués représente cette mémoire. Soit S la suite de toutes les solutions générées. La valeur $f_{i,k}$ représente le nombre de solutions dans S pour lesquelles $s(i) = k$.

Cette quantité reflète le nombre de fois que l'objet i a été affecté à la location k. Les différentes valeurs sont normalisées en les divisant par la valeur moyenne qui est égale à $\frac{1+nb\_iterations}{n}$.

Si aucune amélioration de la solution n'est trouvée au bout de $100n$ itérations, la phase d'intensification est démarrée. Elle part de la meilleure solution trouvée dans la région courante de l'espace des solutions, et d'une liste tabou vide. L'utilisation de la matrice des fréquences va pénaliser les mouvements qui n'améliorent pas la solution en infligeant une forte pénalité aux mouvements les plus fréquemment élus.

Une méthode basée sur le recuit simulé est utilisée dans la phase d'intensification (figure 5.6). L'originalité de notre approche comparée à un recuit simulé traditionnel est qu'elle exploite l'historique de la recherche tabou pour sélectionner des mouvements. Pour chaque mouvement $m = (i, j)$, une pénalité P_m est introduite dans la fonction coût et elle est égale à $Max(f_{i,s(i)}, f_{j,s(j)})$.

La phase de diversification est entamée après $100n$ durant lesquels aucune amélioration de la solution n'est observée. La diversification est effectuée pendant $10n$ itérations. La recherche est forcée à explorer de nouvelles régions en introduisant une restriction tabou aux mouvements moins fréquents. La valeur associée à un mouvement $m = (i, j)$ est $I_m = Min(f_{i,s(i)}, f_{j,s(j)})$. La restriction tabou est définie par $I_m < 1$. Une fois la phase de diversification terminée, l'état tabou basé sur la mémoire à long terme est levé.

Etape 1 : Initialisation
- Choisir la meilleure solution trouvée S_0 ($S = S_0$)
- $nbiter = 0$ /* itération courante */
- $T := T_{max}$ /* Température initiale $T_{max} = 10$ */

Etape 2 : Itération
- **Répéter**
  - nbiter := nbiter + 1
  - **Pour** j:=1 **à** NB_MAX **Faire** /* $NB\_MAX = n$ */
    - Générer un mouvement aléatoire m transformant S en S'
    - $\Delta S = f(S') - f(S)$
    - **Si** ($\Delta S < 0$) ou ($random(0,1) < exp(-\frac{\Delta S * P_m}{T})$) **Alors** $S := S'$
  - **Fin Pour**
  - $T := a.T$ /* recuit avec $a = 0.9$ */
- **Jusqu'à** $T < T_{min}$ /* $T_{min} = 0.5$ */

Figure 5.6 : Recuit simulé avec pénalité pour la phase d'intensification

5.3.2 Évaluation des performances

Les mesures de performance ont été obtenues en exécutant l'application parallèle dans les conditions habituelles d'utilisation du méta-système. L'application s'exécutait dans un contexte multi-utilisateur conjointement avec d'autres applications séquentielles et parallèles, et en présence d'utilisateurs travaillant en mode interactif. Les résultats présentés ci-dessous montre d'une part l'aptitude de l'application à s'adapter aux changements dans le méta-système, et d'autre part les améliorations apportées concernant les solutions connues sur certaines instances du QAP.

5.3.2.a Adaptabilité de l'application

Les facteurs pris en compte lors de l'évaluation de l'adaptabilité de l'application sont : le temps d'exécution, le surcoût induit par l'ordonnanceur, le nombre de nœuds alloués à l'application, le nombre d'opérations de repli et de dépli, et le nombre de processus esclaves défaillants. Le surcoût est le temps total CPU requis par les opérations d'ordonnancement. Le tableau 5.1 résume les résultats obtenus en exécutant 4 fois un problème de grande taille dans un méta-système composé de 100 machines.

	Moyenne	Déviation standard	Min	Max
Temps d'exécution (mn)	145.75	23.75	124	182
Surcoût (s)	8.36	0.24	8.18	8.78
Nombre de nœuds alloués	71	15.73	50	92
Nombre de replis	79	49.75	24	159
Nombre de déplis	179	45.55	120	248
Nombre de pannes des esclaves	1	2.45	0	3

Tableau 5.1 : Résultats obtenus pour 4 exécutions.

Le nombre moyen de nœuds alloués à l'application ne varie pas de façon significative contrairement à la variation de charge (nombre d'opérations de repli et de dépli). Pendant un temps d'exécution de 2h 25mn, 79 opérations de repli et 179 opérations de dépli ont été observées. Ceci correspond à un nouveau nœud alloué à l'application toutes les 0.8 mn et un nœud perdu par l'application toutes les 2 mn. Le surcoût dû à l'exécution adaptative est négligeable en comparaison avec le temps d'exécution (0.09% du temps total). Le nombre de processus esclaves défaillants montre la forte probabilité de panne d'une machine, où s'exécute un processus esclave, dès que le temps d'exécution d'une application est important.

5.3.2.b Performances de l'application

Pour évaluer les performances de l'algorithme parallèle adaptatif de recherche tabou en terme de qualité de la solution et du temps de recherche, nous avons utilisé des problèmes standards du QAP de différents types extraits de la bibliothèque QAP-lib [BSR91] :

- distances et flux aléatoires et uniformes : Tai35a, Tai100a.
- flux aléatoires sur des grilles : Nug30, Sko100a.
- problèmes réels ou pseudo-réels : Bur26d, Ste36b, Tai256c.

L'application a été exécutée 10 fois pour obtenir une performance moyenne. Le tableau 5.2 montre la meilleure solution connue, la meilleure et la plus mauvaise solution trouvées sur les 10 exécutions, la valeur moyenne de la solution ainsi que la déviation standard pour les instances de petite taille sélectionnées ($n < 50$). Le coût de la recherche a été estimé par le temps d'exécution mis pour trouver la meilleure solution et contient donc tous les surcoûts induits.

Nous avons toujours réussi à trouver les meilleures solutions connues pour les problèmes de petite taille. Ceci montre l'efficacité et la robustesse de notre algorithme parallèle adaptatif.

Instance	Meilleure connue	Meilleure trouvée	Mauvaise	Moyenne	Dév. std	Temps de rech. moyen (sec)
Tai35a	2 422 002	2 422 002	2 422 002	2 422 002	0	566
Nug30	6124	6124	6124	6124	0	337
Bur26d	3 821 225	3 821 225	3 821 225	3 821 225	0	623
Ste36b	15 852	15 852	15 852	15 852	0	763

Tableau 5.2 : Résultats pour des petits problèmes ($n < 50$) de différents types

Le tableau 5.3 montre les résultats obtenus pour des problèmes de grande taille. Pour les problèmes aléatoires sur des grilles, nous avons réussi à obtenir soit les meilleures solutions connues (*Sko100c*) soit des solutions très proches des meilleures solutions trouvées. Le problème le plus difficile pour notre algorithme est le problème aléatoire uniforme *Tai100a* dans lequel nous avons obtenu un *gap* de 0.32% au dessus de la meilleure solution connue. En effet, pour cette classe de problèmes, il est difficile de trouver les meilleures solutions connues mais il est « facile » de trouver de « bonnes » solutions.

La difficulté est liée au fait que ces problèmes ne sont pas structurés et que la recherche est vite piégée dans des optima locaux.

Dans la troisième classe de problèmes (réels ou pseudo-réels), les meilleures solutions pour les instances *Tai150b* et *Tai256c* ont été améliorées. Notons que des travaux récents ont réussi à améliorer les résultats que nous avons obtenus [TG97][CMMT97].

Ces résultats montrent que notre algorithme est bien adapté aux problèmes de grande taille mais que ses performances diminuent pour la première classe de problèmes où les instances sont générées aléatoirement et de façon uniforme.

Instance	Meilleure connue	Meilleure trouvée	Gap	Temps de recherche (mn)
Tai100a	21 125 314	21 193 246	0.32%	117
Sko100a	152 002	152 036	0.022%	142
Sko100b	153 890	153 914	0.015%	155
Sko100c	147 862	147 862	0%	132
Sko100d	149 576	149 610	0.022%	152
Sko100e	149 150	149 170	0.013%	124
Sko100f	149 036	149 046	0.006%	125
Wil100	273 038	273 074	0.013%	389
Tho150	8 134 030	8 140 368	0.078%	287
Esc128	64	64	0%	230
Tai150b	499 348 972	**499 342 577**	-0.0013%	415
Tai256c	44 894 480	**44 810 866**	-0.19%	593

Tableau 5.3 : Résultats pour des grands problèmes ($n \geq 50$) de différents types

5.3.2.c Restriction du voisinage

Les résultats ont été très encourageants pour les problèmes réels ou pseudo-réels et notamment les *Tainnc* : génération de trames de niveaux de gris. Pour générer un niveau de gris d'une densité $\frac{m}{n}$ ($m \leq n$), le problème revient à placer m cases noires et $n-m$ cases blanches dans une grille contenant $n = n_1 * n_2$ cases (numérotées de 1 à n). Les cases noires (ou blanches) doivent être dispersées sur la grille de la manière la plus régulière possible pour obtenir de « belles » trames. Le problème a été modélisé comme un QAP dans [Tai95]. Le $i$ème élément ($i \leq m$) d'une solution π donne la position d'une case noire dans la grille. Les matrices de distances et de flux sont définies comme suit :

$$d_{ij} = \begin{cases} 1 \; si \; i \leq m \; et \; j \leq m \\ 0 \; autrement \end{cases}$$

$$c_{ij} = c_{n_2*(r-1)+s\ n_1*(t-1)+u} = \max_{v,w\in\{-1,0,1\}} \frac{1}{(r-t+n_1*v)^2+(s-u+n_2*w)^2}$$

$(i,j) \in [1,n]$

$(r,t) \in [1,n1], (s,u) \in [1,n2]$

où les cases i et j sont placées respectivement aux coordonnées (r,s) et (t,u).

Permuter deux cases noires ou deux cases blanches ne change en rien l'aspect d'une trame. On en déduit que dans ce cas la valeur de la fonction objectif reste inchangée, et que par conséquent plusieurs solutions dans un voisinage possèdent la même valeur de la fonction coût. La recherche atteint alors un plateau du paysage de la fonction coût et elle a du mal à en sortir. Ainsi, les propriétés spécifiques de ce genre de problèmes doivent être prises en considération si l'on veut améliorer la qualité de la recherche en évitant le piège des optimas locaux. L'exploration du voisinage peut être restreinte aux mouvements faisant intervenir une case noire et une case blanche. Il en résulte que la taille du voisinage décroît de $O(n^2)$ à $O(m(n-m))$.

Le tableau 5.4 montre les résultats obtenus par l'algorithme parallèle adaptatif de recherche tabou avec restriction du voisinage. Les instances sont désignées $greyn_1\_n_2\_m$ pour une trame de densité $\frac{m}{n_1*n_2}$. Les problèmes *Grey8_8_13* et *Grey16_16_92* correspondent respectivement aux instances *Tai64c* et *Tai256c* de QAP-lib. Nous avons amélioré les meilleures solutions pour trois instances et avons trouvé les meilleures solutions connues pour les autres. Par exemple, le problème *Tai256c* a été amélioré de 0.053% par rapport au résultat donné dans le tableau 5.3, et de 0.24% par rapport à la meilleure solution connue. Notons que le temps de recherche a été considérablement réduit.

Instance	Taille pb	Densité	Meilleure connue	Meilleure trouvée	Gap	Temps de rech. (mn)
Grey8_8_13	64	$\frac{13}{64}$	1 855 928	1 855 928	0%	< 1
Grey16_16_64	256	$\frac{64}{256}$	19 065 968	**19 050 432**	-0.081%	213
Grey16_16_86	256	$\frac{86}{256}$	38 702 396	**38 381 310**	-0.83%	250
Grey16_16_92	256	$\frac{92}{256}$	44 894 480	**44 787 190**	-0.24%	392
Grey16_16_98	256	$\frac{98}{256}$	51 486 642	51 486 642	0%	17
Grey16_16_128	256	$\frac{128}{256}$	90 565 248	90 565 248	0%	15

Tableau 5.4 : Résultats pour les problèmes de niveaux de gris

5.4 Conclusion

La nature dynamique associée à la charge et l'utilisation d'un méta-système rend essentielle l'exécution adaptative d'applications parallèles. La principale caractéristique des méta-heuristiques développées est l'ajustement, de manière adaptative, du degré de parallélisme de ces applications en fonction de l'état courant du système afin d'exploiter pleinement la disponibilité des machines. Les algorithmes adaptatifs développés peuvent bénéficier largement d'une plate-forme combinant des ressources de traitement allant des réseaux de stations de travail jusqu'aux MPPs.

Une étude expérimentale destinée à la résolution du problème d'affectation quadratique (QAP) a été conduite. Les résultats obtenus pour différentes instances du problème sont encourageants en terme de :

- Adaptabilité : le surcoût induit par les opérations d'ordonnancement d'une application parallèle adaptative (dépli et repli) est négligeable, et les algorithmes réagissent très rapidement aux changements de l'état de charge des machines et de leur utilisation interactive.
- Efficacité et robustesse : l'algorithme TS adaptatif trouve souvent les meilleures solutions connues jusqu'ici. Ceci est dû en partie au niveau élevé de la diversité de la recherche. Les mécanismes sophistiqués tels que l'hybridation et l'usage d'une mémoire à long terme doivent être utilisés afin d'améliorer la qualité des solutions pour certains problèmes de grande taille.

Dans le cadre d'une collaboration, nous avons développé une application basée sur la recherche tabou telle qu'elle a été décrite dans ce chapitre mais en intégrant un diversificateur basé sur les algorithmes génétiques au niveau du processus maître [BHPT98]. Les processus esclaves effectuent une recherche tabou et communiquent régulièrement leur matrice de fréquence au processus maître. L'algorithme génétique s'appuye sur ces matrices de fréquence pour diversifier la recherche.

Enfin, bien que ces algorithmes ont été appliqués sur le QAP, il est très aisé de les adapter à d'autres types de problèmes d'optimisation combinatoire tels que le voyageur de commerce (TSP), l'affectation généralisée (GAP), et le problème de coloriage de graphes.

Chapitre 6

Méthodes exactes parallèles adaptatives

L'exploration d'espaces de recherche est une technique très répandue dans le domaine de l'optimisation combinatoire notamment en intelligence artificielle (IA) et en recherche opérationnelle (RO). Dans le chapitre précédent, nous nous sommes intéressés aux méta-heuristiques qui permettent de trouver une solution de « bonne » qualité en un temps raisonnable. Dans ce chapitre, nous allons focaliser sur les méthodes dites exactes (figure 6.1) qui permettent, sous certaines conditions, de garantir l'optimalité de la solution trouvée au prix d'un temps d'exécution relativement important. La performance d'un algorithme d'exploration est déterminée par la complexité en termes de temps d'exécution, d'espace mémoire et de qualité de la solution trouvée (optimale ou non).

Outre la représentation d'un espace de recherche qui est souvent matérialisée par un graphe d'états en IA et par des arborescences de sous-problèmes en RO, un algorithme de recherche est caractérisé par trois éléments étroitement liés :

- **Opérateur de génération** : l'opérateur de génération est souvent lié au problème à résoudre et permet de construire de nouveaux sommets fils à partir d'un sommet père donné[1].
- **Fonction d'évaluation** : la fonction d'évaluation ou la fonction coût permet d'estimer la « distance » entre un sommet courant et les sommets finaux. Une estimation exacte conduit à une exploration optimale de l'espace de recherche. Malheureusement, une telle estimation est souvent très coûteuse et parfois impossible.

[1] Un sommet représente un état dans un graphe d'états et un sous-problème dans une arborescence de sous-problèmes.

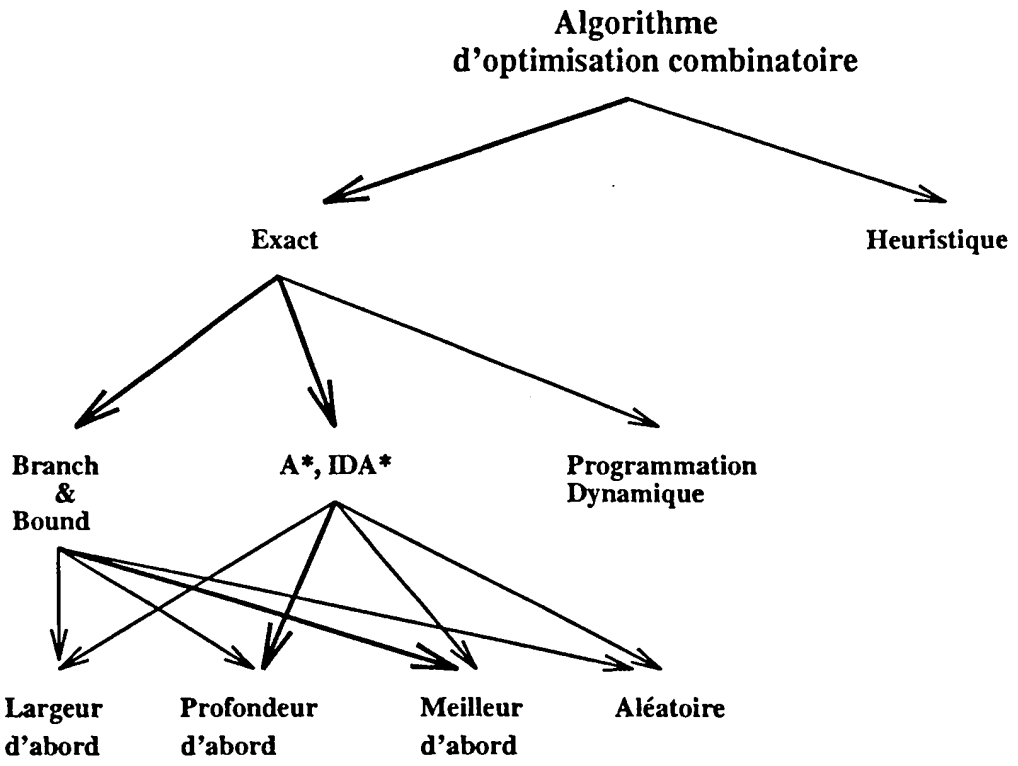


Figure 6.1 : Classification des algorithmes exacts d'optimisation combinatoire

En général, on opte pour un meilleur compromis entre l'effort d'évaluation et l'effort d'exploration en utilisant des heuristiques. La fonction d'évaluation a une répercussion directe sur le temps de recherche en réduisant l'espace de recherche par l'« élaguage » des sommets jugés inutiles.

- **Statégie de contrôle** : la stratégie de contrôle permet de guider l'exploration afin d'atteindre les sommets finaux (s'ils existent) plus ou moins rapidement. Ce guidage est basé sur la sélection du sommet le plus « prometteur ». Les principales stratégies utilisées sont les stratégies en profondeur d'abord, en largeur d'abord, meilleur d'abord et aléatoires. Ces stratégies peuvent être combinées pour donner naissance à des stratégies qualifiées d'« hybrides » [Pea84].

Ce chapitre est structuré de la façon suivante : nous commençons par une revue des principales stratégies de parcours d'un espace de recherche suivie par une étude de la parallélisation adaptative de deux algorithmes de recherche. Le premier algorithme est une variante de l'algorithme A* tandis que le deuxième fait partie de la famille des algorithmes dits par « évaluation et séparation » (Branch-and-Bound ou B&B).

6.1 Revue des principaux algorithmes exacts

Dans cette section on passera en revue les différents algorithmes de parcours d'arbres de recherche mais d'abord notons que la complexité d'un algorithme de recherche sur les trois dimensions espace, temps et qualité de la solution est exprimée à l'aide de deux paramètres : le « facteur ou degré de branchement » (*branching factor*) d'un sommet de l'arbre de recherche et la profondeur (*depth*) de la solution dans l'arbre[2].

Le facteur de branchement b pour un problème donné est défini comme étant le nombre moyen d'états nouveaux qui peuvent être générés après avoir appliqué un opérateur sur un état donné. La profondeur d de la solution est le nombre d'états qui ramènent l'état initial à l'état final. De ce fait, la complexité en temps de recherche d'un algorithme est proportionnelle au nombre d'états générés, et la complexité en espace mémoire est proportionnelle au nombre d'états qui doivent être sauvegardés en mémoire durant la recherche.

6.1.1 Recherche en largeur d'abord « breadth-first search »

La recherche en largeur d'abord développe, à partir du sommet initial, tous les sommets à la profondeur 1, puis ceux à la profondeur 2 et ainsi de suite jusqu'à trouver la solution. Comme cet algorithme développe tous les sommets à une profondeur donnée avant de passer à la suivante, la solution trouvée est obligatoirement celle de plus court chemin dans l'arbre.

Avant de passer à la profondeur d, l'algorithme doit sauvegarder tous les sommets générés à la profondeur $d-1$ pour pouvoir les développer à l'étape suivante c'est-à-dire b^{d-1}. À la profondeur d contenant la solution, l'algorithme doit mémoriser à la limite b^d sommets et en moyenne $\frac{1}{2}(b^d)$ sommets. La complexité en espace est donc en $O(b^d)$.

A la limite, à la profondeur d, l'algorithme génère $b + b^2 + b^3 + ... + b^d = \frac{b^{d+1}-1}{b-1} - 1$ sommets et en moyenne il génère $b + b^2 + b^3 + ... + b^{d-1} + \frac{1}{2}b^d$. La complexité en temps est elle aussi en $O(b^d)$.

La complexité en espace présente le principal inconvénient de cet algorithme du fait qu'il sature très rapidement la mémoire.

6.1.2 Recherche en profondeur d'abord « depth-first search »

La recherche en profondeur d'abord génère toujours un descendant à partir du sommet le plus récemment développé jusqu'à atteindre une profondeur limite (sinon l'algorithme risque de ne jamais s'arrêter). Dans ce cas, l'algorithme fait un retour arrière (*back-tracking*) et tente de générer un autre descendant. Ces opérations sont répétées jusqu'à ce que la solution soit trouvée ou qu'il ne reste plus de sommets à développer.

[2]Un arbre est un graphe particulier (graphe sans cycle).

Si on suppose que l'on s'arrête à la première solution trouvée alors la solution, si elle est trouvée, n'est pas forcément celle de plus court chemin parce-qu'il peut y avoir un autre chemin contenant la solution et qui soit optimal.

L'algorithme doit seulement mémoriser la branche allant du sommet initial au sommet courant et par conséquent la complexité en espace est en $O(d)$ où d est la profondeur limite[3].

Au pire des cas, l'algorithme génère $b + b^2 + b^3 + ... + b^d$ sommets où d est la profondeur limite et en moyenne il génère la moitié. Donc la complexité en temps de la recherche en profondeur d'abord est en $O(b^d)$.

Cet algorithme évite la limitation mémoire de la recherche en largeur d'abord. Son inconvénient vient du fait qu'il faut choisir *arbitrairement* la profondeur limite. Ceci peut conduire à une sous-estimation de sa valeur et à ne pas trouver la solution, ou à une sur-estimation de sa valeur et à trouver une solution non optimale.

6.1.3 Recherche itérative en profondeur d'abord « depth-first iterative-deepening search »

La recherche itérative en profondeur d'abord procède par itérations. Durant la première itération, elle effectue une recherche en profondeur d'abord jusqu'à la profondeur 1. Durant la deuxième itération, elle recommence une recherche en profondeur d'abord du sommet initial jusqu'à la profondeur 2. Durant la troisième itération, elle recommence une fois de plus une recherche en profondeur d'abord en partant du sommet initial jusqu'à la profondeur 3 et ainsi de suite jusqu'à trouver la solution.

Cet algorithme garantit l'optimalité de la solution trouvée car il développe d'abord tous les sommets à une profondeur donnée avant de passer à une profondeur supérieure.

Tout comme l'algorithme de recherche en profondeur d'abord, la complexité en espace est en $O(d)$ où d est la profondeur de la solution.

Le nombre de sommets générés par la recherche itérative en profondeur d'abord jusqu'à la profondeur d est : $b^d + 2b^{d-1} + 3b^{d-2} + ... + db$. Cette somme est majorée par la série

$$b^d \sum_{i=0}^{+\infty} (i+1)(\frac{1}{b})^i$$

qui converge vers $b^d(1 - \frac{1}{b})^{-2}$ si $b > 1$ et par conséquent la complexité en temps est en $O(b^d)$.

Quoique cet algorithme effectue du traitement inutile (en générant les mêmes sommets d'une itération à l'autre), ceci n'affecte pas asymtotiquement le temps de recherche pour la simple raison intuitive que le plus gros du travail se fait toujours à la dernière

[3]Si tous les sommets fils sont engendrés à une profondeur donnée, la complexité en espace devient en $O(bd)$.

itération. L'inconvénient de cet algorithme est qu'il explore tous les chemins possibles jusqu'à une profondeur donnée pour trouver la solution optimale.

6.1.4 Recherche bi-directionnelle « bi-directional search »

La recherche se fait en partant du sommet initial et du sommet solution simultanément en mémorisant les sommets générés jusqu'à trouver un sommet commun permettant de relier les deux chemins.

L'algorithme utilisé dans les deux sens peut être celui de la recherche itérative en profondeur d'abord, dans ce cas la solution est celle de plus court chemin.

Si d est le nombre d'états qui ramènent le sommet initial au sommet solution alors l'algorithme itératif en profondeur d'abord va être exécuté dans les deux sens pendant d/2 itérations en mémorisant à chaque fois tous les sommets générés dans un seul sens pour pouvoir les comparer avec les sommets feuilles générés dans l'autre sens. A la limite il faut mémoriser $(b + b^2 + ... + b^{d/2})$ d'où la complexité en espace est en $O(b^{d/2})$.

Le nombre de sommets générés est $2(b^{d/2} + 2b^{d/2-1} + 3b^{d/2-2} + ... + (d/2)b)$ d'où le temps d'exécution est lui aussi d'une complexité en $O(b^{d/2})$.

Cet algorithme n'est adapté qu'aux problèmes ayant un état final *unique explicite* et dont les opérateurs sont inversibles. Il permet de réduire le temps de recherche relativement aux algorithmes déjà vus mais requiert beaucoup d'espace mémoire.

6.1.5 Recherche heuristique A* « heuristic search »

La recherche heuristique est basée sur l'estimation du coût pour atteindre le sommet final en partant d'un sommet donné. L'algorithme A* utilise une stratégie meilleur d'abord pour choisir toujours, parmi les sommets feuilles, le sommet de moindre coût et le développer [Nil80]. Le coût d'un sommet n est calculé comme étant :

$$f(n) = g(n) + h(n),$$

où $g(n)$ est le coût du chemin parcouru du sommet initial au sommet n (généralement c'est la profondeur dans l'arbre du sommet n) et $h(n)$ est une estimation minorante du coût du chemin à parcourir du sommet n au sommet solution (dépend généralement du problème en question). Des étiquettes sont attribuées aux sommets. Elles représentent les coûts associés et c'est le sommet de moindre coût qui sera développé en priorité.

La solution est garantie optimale si la fonction heuristique ne sur-estime pas le coût pour atteindre le sommet solution.

L'algorithme A* doit mémoriser tous les sommets générés (sommets feuilles) afin de les développer plus tard ce qui requiert un espace mémoire très important.

Des études ont montré que le nombre de sommets générés par l'algorithme A* augmente

de manière exponentielle avec la profondeur [Pea84]. Par conséquent, la complexité en temps de la recherche A* est en $O(b^d)$.

L'inconvénient majeure de la recherche A* est qu'elle sature très rapidement l'espace mémoire. Une amélioration serait de détecter les doublons, ceci réduit l'espace de recherche mais ne permet pas de contourner complètement le problème. Des variantes limitant l'espace mémoire exigé par une recherche de type A* ont été développées (MA* [CGAS89], MREC [SB89], RA* [EHM90], IDA* [Kor85], etc.). Ces algorithmes ont été utilisés avec succès sur des problèmes jusqu'alors jamais résolus par l'algorithme A* faute d'espace mémoire. La section 6.2 est consacrée à l'étude de la parallélisation de la recherche IDA*.

6.1.6 Recherche aléatoire « random search »

La recherche aléatoire utilise une fonction qui attribue à chaque sommet généré un coût déterminée au hasard. Le sommet de meilleur coût sera alors développé.

Comme cette stratégie ne s'appuye pas sur certaines caractéristiques du problème traité au moment de l'évaluation, la solution trouvée est en général non-optimale.

Tout comme la stratégie du meilleur d'abord, cette stratégie possède une complexité spatiale et temporelle en $O(b^d)$ au pire des cas.

Cette méthode peut être intéressante à cause de son caractère indéterministe et du fait que le coût lié à la fonction d'évaluation est quasiment nul. Cependant, elle possède l'inconvénient de ne pas garantir l'optimalité de la solution trouvée.

6.2 Algorithme IDA* parallèle adaptatif

Dans cette section, nous allons nous intéresser de plus près à l'algorithme IDA* puisque non seulement il a montré son optimalité sur les trois dimensions espace, temps et qualité de la solution mais aussi il est très simple à implémenter. IDA* a été appliqué à un problème bien connu en intelligence artificielle : le jeu de taquin.

6.2.1 Principe de la recherche IDA*

La recherche itérative en profondeur d'abord avec heuristique (*depth-first iterative-deepening A* search* ou IDA*) est la combinaison de la recherche itérative en profondeur d'abord et de la recherche heuristique A*. Contrairement à la recherche itérative en profondeur d'abord, l'idée est non pas de limiter la profondeur mais le coût de la recherche. De ce fait, l'algorithme IDA* effectue à chaque itération une recherche en profondeur d'abord jusqu'à ce que le coût du sommet feuille dépasse un seuil fixé au début de l'itération. Ce seuil correspond au départ au coût du sommet initial et est augmenté à chaque nouvelle itération. A une itération donnée, le coût limite est le minimum de

tous les coûts ayant dépassé le seuil à l'itération précédente (figure 6.2). La fonction coût est du type $f(n) = g(n) + h(n)$.

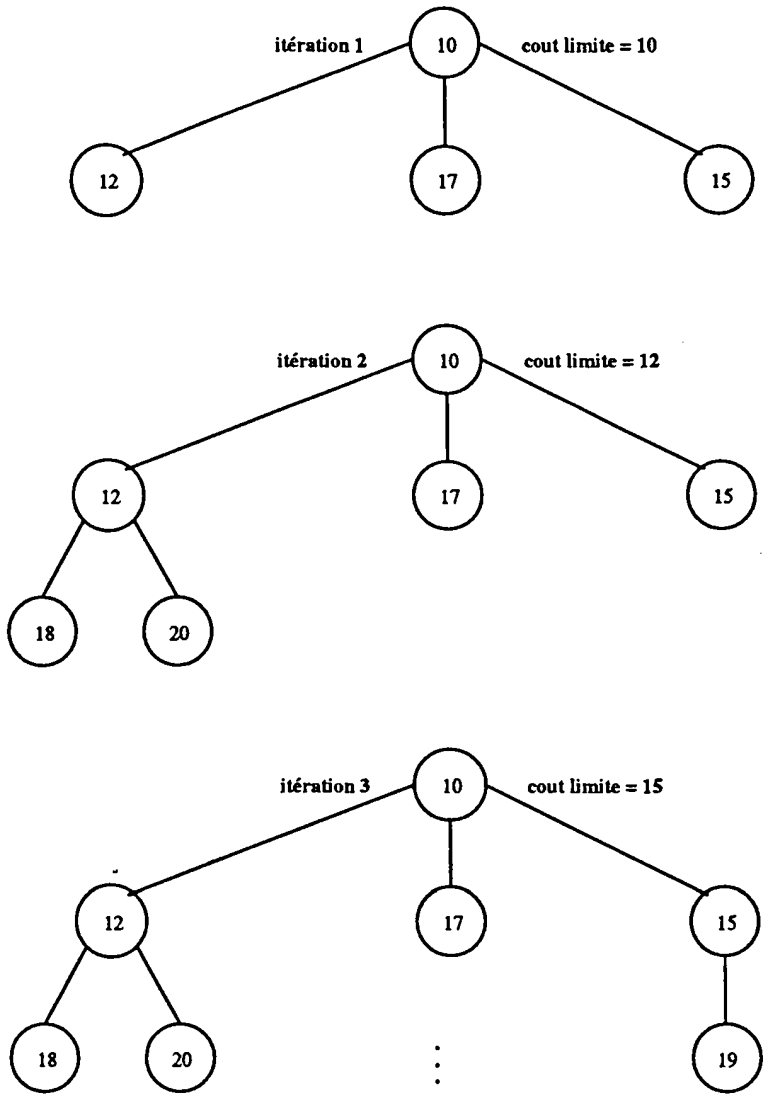


Figure 6.2: Recherche itérative en profondeur d'abord avec heuristique A*

Avant de présenter les propriétés de l'algorithme IDA* en terme de complexité et de qualité de la solution, donnons quelques définitions.

Définition 1 *Une fonction coût f(n) est dite* ***monotone croissante ou consistente***

si pour tous les sommets n et n' où n' est le descendant de n, f(n) ≤ f(n') [Kor85].

Définition 2 *Une fonction coût f(n) est dite* ***admissible*** *si elle ne sur-estime jamais le coût pour aller au sommet solution en partant du sommet courant [HNR68].*

Etant donné une fonction coût $f(n) = g(n) + h(n)$ admissible et consistente, si $g(n)$ représente la profondeur du sommet n et $h(n)$ est l'estimation du coût pour aller du sommet n au sommet solution alors :

- La solution (s'il en existe au moins une) est à une profondeur au moins égale à la valeur de l'heuristique initiale f(1) [Haf94]. Cette propriété est très importante car elle donne une idée sur le nombre minimum d'états séparant l'état initial de l'état final (profondeur de la solution) avant même d'entamer la recherche.
- IDA* garantit l'optimalité de la solution si elle existe. Du moment que IDA* développe tous les sommets ayant un coût ne dépassant pas un coût limite avant de développer les sommets ayant un coût supérieur, la première solution trouvée est forcément celle de moindre coût. De plus, en cas d'existence de plusieurs solutions, celles-ci sont toutes optimales et situées à la même profondeur dans l'arbre de recherche [Haf94].

A une itération donnée, IDA* a besoin seulement de mémoriser la branche de l'arbre qu'il est en train d'explorer. Si d est la profondeur du sommet le plus profond alors la complexité en espace est en $O(d)$.

Dechter et Pearl ont montré que A* est optimal en termes de nombre de sommets générés sur la classe des recherches meilleur-d'abord avec heuristiques admissibles monotones [DP83]. Quoique IDA* développe en réalité plus de sommets que l'algorithme A*, les itérations précédant la dernière n'affectent pas le comportement asymptotique de IDA* en matière de nombre de sommets générés vis-à-vis de la recherche A*. Donc IDA* génère asymptotiquement le même nombre de sommets que A*. Plus étonnant encore, IDA* est parfois plus rapide que A* parce-qu'il y a moins de surcoût par sommet généré.

L'algorithme IDA* réduit donc de façon considérable la complexité du parcours d'arbres de recherche en coupant les branches dès que le coût des sommets dépasse le coût seuil. IDA* surpasse la contrainte espace mémoire de la recherche A* et permet de trouver la solution de moindre coût si la fonction d'évaluation utilisée est admissible et monotone. Comme la plupart des algorithmes de recherche, le problème de IDA* réside dans la complexité exponentielle du temps de recherche ce qui motive le recours au parallélisme.

L'algorithme correspondant à la recherche IDA* est illustré par la figure 6.3.

6.2.2 Recherche IDA* parallèle

Il existe essentiellement quatre approches de parallélisation de IDA* [PFK93] : décomposition de la génération et de l'évaluation, recherche par fenêtres, recherche par

```
Possible = VRAI                                  /* L'état initial est décomposable */
Solution = FAUX
REPETER
   TANT QUE (Possible ET !SolutionTrouvee())     /* Descente en profondeur */
     FAIRE
        DeveloppperSommetSuivant()
     FINFAIRE
   SI (!Possible)
    ALORS
      TANT QUE (!Possible ET (Profondeur != 0)) /* Remontée dans l'arbre */
        FAIRE
           BackTracking()
           DeveloppperSommetSuivant()
        FINFAIRE
      SI (!Possible ET (Profondeur = 0))                /* Nouvelle itération */
       ALORS
         ChangerHeuristique()
         DeveloppperSommetSuivant()
        FINSI
    FINSI
JUSQU'A (Solution)
```

Figure 6.3 : Algorithme IDA* de base

projection, et recherche par décomposition.

6.2.2.a Décomposition de la génération et de l'évaluation

Cette approche consiste à paralléliser les tâches associées aux traitement de chaque sommet telles que la génération des sommets fils possibles et l'évaluation du coût. Elle a été utilisée de manière cablée pour la détermination en parallèle des mouvements dans un échiquier [Ebe90]. Le gain apporté est cependant limité par le degré de parallélisme disponible lors de la génération de mouvements et de l'évaluation. Cette approche n'est donc pas adaptée aux problèmes ayant un facteur de branchement réduit. De plus, elle est spécifique aux problèmes traités.

6.2.2.b Recherche par fenêtres

La recherche parallèle par fenêtres « Parallel Window Search » confie à chaque processeur tout l'espace de recherche mais divise l'intervalle du coût (fenêtres) sur les processeurs. Dans le cas de IDA*, chaque processeur se voit attribuer une itération et dès qu'un processeur est disponible il traitera une autre itération [PK91]. Dans cette approche, le temps de recherche est donc limité par le temps d'exécution de l'itération contenant la solution.

6.2.2.c Recherche par projection

Cette troisième approche de mapping répartit l'espace de recherche sur les processeurs [EHM90]. A chaque sommet lui correspond un processeur particulier déterminé par une fonction de hachage dont le but est de détecter les doublons. Les sommets sont créés dynamiquement durant la recherche et envoyés à leurs processeurs correspondants. La limitation de cette approche de mapping apparaît lorsque les sommets doivent être transférés à d'autres processeurs. Le temps de communication devient alors rédhibitoire.

6.2.2.d Recherche par décomposition

Cette approche repose sur le caractère récursif d'un arbre. En fait un arbre est constitué de plusieurs sous-arbres qui sont eux-mêmes des arbres. L'idée est de décomposer l'arbre de recherche, d'affecter à chaque processeur un sous-arbre et d'effectuer la recherche en parallèle. De ce fait, l'algorithme de recherche est le même pour tous les processeurs mais la zone de recherche diffère d'un processeur à un autre [PFK93]. Généralement, le découpage de l'arbre de recherche est effectué initialement en fonction du nombre de processeurs utilisés.

Cette approche est relativement simple à mettre en oeuvre et permet d'accélérer la recherche par rapport aux méthodes précédentes. De plus, elle est indépendante des spécifications du problème et peut être implémentée sur un nombre quelconque de processeurs. Le seul problème avec cette méthode est le déséquilibre de charge de travail des processeurs dû à la forte irrégularité de l'arbre et à la synchronisation à la fin de chaque itération. Des solutions basées sur l'équilibrage de charge pendant la distribution initiale des sous-arbres, entre les itérations et pendant les itérations ont été apportées en SIMD [PFK93] [FM90] [MD92] et en MIMD [RK87] [HTG95] [MDLT96].

6.2.3 Modèle parallèle adaptatif

Dans ce qui suit, nous allons détailler l'approche utilisée pour introduire le parallélisme adaptatif dans la recherche IDA*. Etant donné la nature dynamique et récursive de l'arbre généré par IDA*, nous avons opté pour une recherche par décomposition dynamique de l'arbre et non un découpage initial (statique). Ainsi, ce sont les processus esclaves qui décomposent l'arbre indépendamment les uns des autres. Ils coopèrent à trouver le nouveau seuil pour l'itération suivante.

Le travail des processus esclaves consiste à explorer récursivement, selon une stratégie en profondeur d'abord avec heuristique, les sommets des sous-arbres obtenus par « émiettement » de l'état initial. Au départ, la file des travaux actifs positionnée par le processus maître contient une seule unité de travail correspondant à l'état initial. Ainsi, un seul processus esclave parmi les processus en exécution aura l'opportunité d'effectuer le traitement. Ce dernier consiste a effectuer l'exploration tout en remontant au proces-

sus maître les sommets feuilles de l'arbre généré[4] afin qu'ils soient traités par d'autres processus esclaves. Le choix du moment après lequel la remontée des sommets feuilles est effectuée détermine la granularité de l'application. On peut utiliser une granularité spatiale basée sur le nombre de sommets explorés, ou une granularité temporelle basée sur une durée d'exploration. Dans les deux cas, une granularité grosse peut conduire à un déséquilibrage de charge voire une famine de processus esclaves tandis qu'une granularité fine permet de faire travailler tous les processus esclaves et donc de réaliser un équilibrage de charge optimal. Nous avons opté pour une granularité temporelle fine.

Durant la recherche, l'opérateur de génération est récursivement appliqué à un état donné pour générer tous les états possibles. Le choix d'un état fils pour descendre le plus profondément dans l'abre nécessite dans ce cas un ordonnancement des états générés. En effet, les états fils engendrés sont toujours traités dans l'ordre croissant des valeurs de h. Cet ordonnancement permet d'explorer l'état potentiellement le plus proche de la solution finale et par conséquent le plus prometteur parmi les états fils.

Si une solution est trouvée, le processus maître est informé pour arrêter la recherche. Dans le cas contraire, dès qu'une itération est terminée, le processus maître prépare une nouvelle itération en replaçant le sommet initial dans la file des travaux actifs mais avec un coût seuil égal au minimum des coûts seuils trouvés par les processus esclaves à l'itération précédente.

En cas de repli, le travail pendant d'un processus esclave est constitué de tous les sommets non encore explorés dans la branche courante de l'arbre. Notons qu'on réutilise le service de repli pour remonter les sommets en exccès au processus maître.

En cas de dépli, avant d'entamer une exploration, le processus esclave demande du travail au processus maître. L'unité de travail correspond à une description du sommet à explorer, sa position dans l'arbre, son coût et une indication sur les sommets déjà générés à partir de ce sommet. En cas d'existence de plusieurs unités de travail en attente, le processus maître peut sélectionner aléatoirement l'unité de travail à envoyer (pas de notion de priorité). Cependant, nous envoyons un sommet prometteur (potentiellement proche de la solution).

6.2.4 Application au problème du jeu du taquin

Le jeu du taquin est devenu un problème standard pour les algorithmes de recherche parce-qu'il est facile à implémenter mais requiert beaucoup de temps de calcul afin de trouver une solution optimale. Il s'agit bien entendu d'un problème NP-complet [Kor85].

[4] Le processus esclave ne remonte pas la totalité des sommets. Il en garde un ou plusieurs sommets pour pouvoir poursuivre le traitement.

6.2.4.a Principe du jeu du taquin

Le jeu du taquin consiste en une grille carrée de NxN cases parmis lesquelles figure une case vide appelée le *blanc*. Les autres cases étant numérotées de 1 à $N^2 - 1$. Il s'agit donc du taquin 15[5] pour une grille 4x4, du taquin 24 pour une grille 5x5, du taquin 35 pour une grille 6x6 et ainsi de suite (figure 6.4).

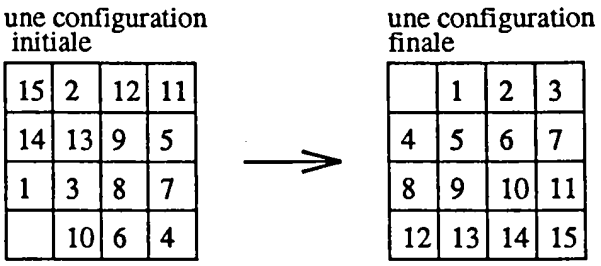


Figure 6.4 : Le jeu du taquin 15

Chaque case *horizontalement* ou *verticalement adjacente* au blanc peut « glisser » et prendre la place de celui-ci (figure 6.5). Le but du jeu est, partant d'une configuration initiale aléatoire, de réarranger les cases pour aboutir à une certaine configuration finale en un nombre minimum de déplacements de cases.

A partir d'une configuration donnée on peut générer deux nouvelles configurations si le blanc est aux coins de la grille, trois si le blanc est sur les côtés (coins exclus) et quatre si le blanc est à l'intérieur de la grille[6]. Ce raisonnement peut être appliqué aux nouvelles configurations obtenues ce qui nous conduit à une structure arborescente. Par conséquent, la recherche d'une solution au jeu du taquin nous amène à un parcours d'arbre.

6.2.4.b Codification du problème

Une configuration du taquin est représentée par un tableau uni-dimensionnel contenant N^2 éléments. L'avantage est qu'il y a moins d'indexation à gérer qu'avec un tableau à deux dimensions. Par convention, le blanc est représenté par la valeur 0. La figure 6.6 montre la représentation des configurations de la figure 6.4. Dans le cas de la configuration initiale, on dira que la case numéro 15 occupe la position 0, la case numéro 2 occupe la position 1, etc.

L'opérateur de génération permettant le passage d'une configuration à une autre est

[5] Appelé « fifteen-puzzle » et fait l'objet de notre étude

[6] Si on ne compte pas l'ascendant alors le facteur de branchement du taquin 15 est environ égal à 2.

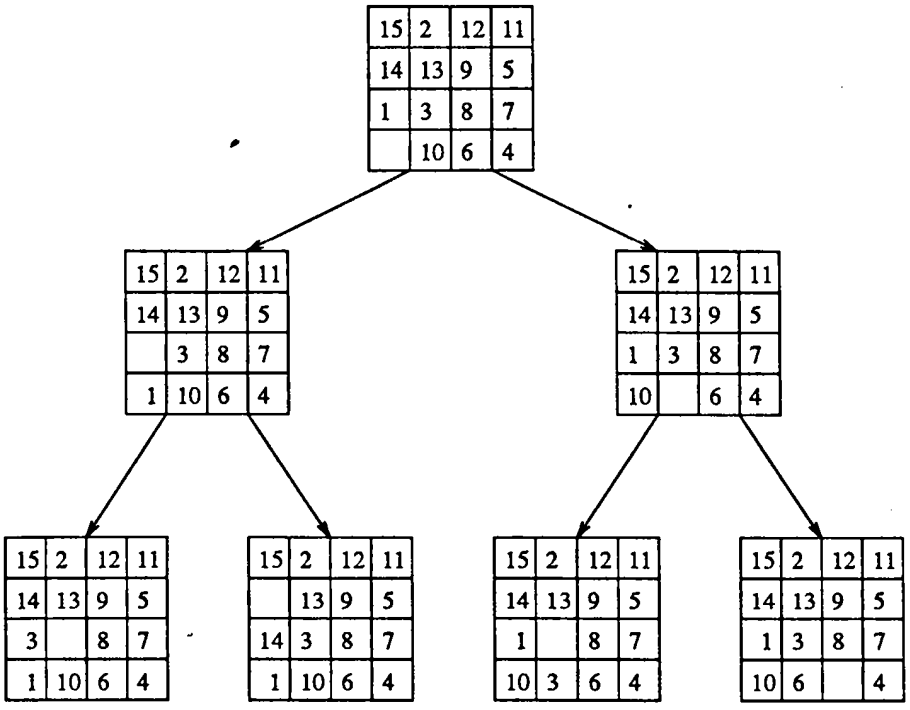


Figure 6.5 : Mouvements possibles dans un exemple du taquin 15

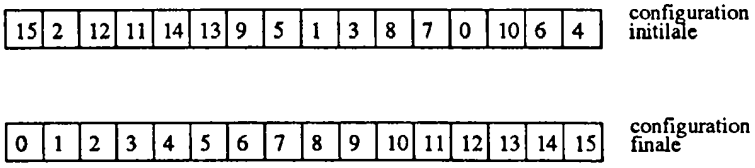


Figure 6.6 : Représentation du taquin 15 par un tableau uni-dimensionnel

matérialisé par *un déplacement du blanc.* Au plus, quatre nouvelles configurations peuvent être générées. La fonction coût est choisie de la façon suivante :

- $g(n)$ représente la profondeur de la configuration courante n ou le nombre de mouvements du blanc déjà effectués.
- $h(n)$ représente la somme des *distances de Manhattan* (différence de lignes et de colonnes) de toutes les cases de la configuration courante n à l'exclusion du blanc.

La fonction coût est admissible et consistente car :

- La somme des distances de Manhattan représente le nombre minimum de mouvements nécessaires pour ramener chaque case de sa position courante à sa position finale.
- La fonction coût d'une nouvelle configuration garde le coût de l'ascendant ou augmente de 2 :

  f(n') = f(n) ou alors f(n') = f(n) + 2, n' est le descendant de n

  – La génération d'une nouvelle configuration à partir d'une autre équivaut au passage à une profondeur supérieure d'où $g(n)$ augmente toujours de 1 :

  g(n') = g(n) + 1, n' est le descendant de n

  – Pour passer d'une configuration à une autre, on ne fait que déplacer une case. On a tendance à la rapprocher ou l'éloigner de sa position finale d'où $h(n)$ augmente ou diminue de 1 :

  h(n') = h(n) $\pm$ 1, n' est le descendant de n

Pour passer d'une itération à une autre, il faut changer la valeur de l'heuristique. La nouvelle valeur correspond au minimum des coûts des sommets feuilles de l'itération précédente (voir section 6.2.1). Dans le cas du taquin, il est inutile de faire une recherche de minimum car les coûts des sommets feuilles dépassent de 2 la valeur de l'heuristique

courante. D'où la nouvelle valeur de l'heuristique est égale à celle de l'heuristique précédente augmentée de deux.

$$nouveau\ f_{seuil} = ancien\ f_{seuil} + 2$$

Pour gérer la stratégie d'exploration en profondeur d'abord, nous mémorisons la branche courante de l'arbre en utilisant une pile qui contient tous les états fils générés à partir du sommet initial jusqu'au sommet courant. A chaque sommet on lui associe sa profondeur, la position du blanc dans le sommet précédent[7], une indication sur les sommets fils déjà générés, et le coût du sommet.

Pour éviter de recalculer les distances de Manhattan à chaque fois qu'une nouvelle configuration est générée, il est convenable d'effectuer ces calculs au départ une fois pour toutes et de les ranger dans une matrice. Le premier indice désigne le numéro de la case et du coup sa position finale, le deuxième indice indique sa position actuelle.

Pour réduire la complexité de l'évaluation de la fonction f, il est possible de ne pas recalculer entièrement le coût d'une configuration nouvellement générée. En effet, pour passer d'une configuration à une autre on intervertit seulement la position de la case vide avec une case voisine. Il suffit alors de retrancher de l'ancien coût l'ancienne distance de Manhattan de la case en déplacement et d'en rajouter la nouvelle + 1 (on passe à une profondeur supérieure). La case vide n'intervient pas dans le calcul des coûts.

6.2.4.c Evaluation des performances

Nous avons implémenté l'algorithme IDA* séquentiel et nous l'avons exécuté sur un processeur de la ferme d'ALPHA [HTG95]. Sur les 100 configurations du taquin 15 [Kor85], l'exécution a duré 17 heures et 49 minutes en générant 36.3 milliards d'états. Une version parallèle semi-adaptative, intégrant une stratégie d'équilibrage de charge pendant les itérations, a permis de ramener le temps d'exécution à 1 heure et 29 minutes sur 16 processeurs ALPHA (exécution mono-application, mono-utilisateur, sur la plate-forme homogène réservée à cet effet). Le tableau 6.1 montre les résultats d'exécution de l'application adaptative sur les 5 premières configurations du taquin 15 en utilisant 30 machines. Ces résultats sont sont comparés à ceux obtenus en exécutant l'application non-adaptative sur une ferme de 16 processeurs ALPHA.

6.3 Algorithme Branch-and-Bound parallèle adaptatif

Dans cette section, nous allons nous intéresser à l'algorithme Branch-and-Bound (B&B) faisant partie des méthodes dites par séparation et évaluation. L'algorithme B&B [DFJ54] [LMSK63] est largement utilisé en RO pour la résolution d'un grand nombre de problèmes pratiques.

[7] Il est inutile de mémoriser la configuration en entier car on ne permute que deux cases (le blanc et une case adjacente) donc la position précédente du blanc suffit pour revenir en arrière.

Configuration	Nombre de déplacements du blanc	Nœuds générés	Temps d'exécution en secondes (ferme d'ALPHA) (16 processeurs)	Temps d'exécution en secondes (méta-système)
1	57	685 253 878	114	345
2	55	81 958 255	25	71
3	59	2 364 629 062	335	702
4	56	275 641 438	52	115
5	56	74 741 937	21	67

Tableau 6.1 : Résultats pour quelques configurations du taquin 15

6.3.1 Principe de l'algorithme B&B

En RO, le processus d'exploration d'un espace de recherche peut être vu comme une décomposition successive de problèmes en sous-problèmes de taille inférieure. Une fonction d'évaluation f est généralement utilisée afin de pouvoir choisir un sous-problème parmi ceux qui sont en attente d'exploration[8]. La fonction f donne une estimation minorante de la valeur des solutions de chaque sous-problème. Dans la plupart des problèmes en RO, la fonction f s'exprime comme la somme des évaluations des choix (fixations de variables) déjà faits et des estimations (bornes inférieures) sur les choix restants à faire.

A partir d'un problème initial, l'algorithme B&B engendre des sous-problèmes à l'aide des opérateurs de séparation (figure 6.7). Ces opérateurs sont généralement liés au problème à résoudre. Chaque nouveau sous-problème est alors évalué par la fonction f et sa valeur est comparée à la borne supérieure du problème en question. En général, la borne supérieure correspond à la valeur de la meilleure solution connue jusqu'alors[9]. Si cette borne est dépassée, le sous-problème ne contient pas de meilleure solution et il est alors élagué. Dans le cas où un sous-problème correspond à une solution réalisable et que sa valeur est meilleure que la borne supérieure, celle-ci est mise à jour et les sous-problèmes dont la valeur dépasse la nouvelle borne supérieure seront élagués. Si un sous-problème ne correspond pas à une solution réalisable et qu'il est décomposable alors le processus d'exploration est réitéré. L'exploration s'arrête lorsqu'il n'y a plus de sous-problèmes à traiter.

Notons l'impact de la valeur de la borne supérieure et de la borne inférieure sur le processus d'exploration. Plus ces bornes se resserrent autour de la valeur de la solution optimale, mieux la recherche est. En effet, il y aurait plus d'élagage de sommets inutiles. Pour plus de détails concernant l'algorithme B&B, les lecteurs intéressés peuvent trouver une étude exhaustive dans [Ber92].

[8] On parlera alors d'algorithme B&B meilleur d'abord.

[9] Souvent, il s'agit d'une solution déterminée par une heuristique telle que la recherche tabou.

Comme pour l'algorithme A*, l'algorithme B&B possède une complexité temporelle en $O(b^d)$ au pire des cas. Dans un B&B meilleur d'abord, la sauvegarde des sommets correspondants aux sous-problèmes non encore traités conduit à une complextié spatiale en $O(b^d)$.

```
Etape 1 : Initialisation
  Si x0 est une « bonne » solution réalisable
    Alors ub = f(x0) /* initialiser la borne supérieure ub */
    Sinon ub = ∞
  - Initialiser(tas)
Etape 2 : Itération
  Tant que (tas ≠ ∅) Faire
    - x = Extraire_sommet(tas) /* sélectionner un sous-problème à traiter */
    Pour chaque descendant y de x Faire /* expansion de x */
      Si y réalisable et (f(y) < ub) /* mise à jour de ub ? */
        Alors ub = f(y) ; Elaguer(tas, ub)
      Si y non terminal et (g(y) < ub)
        Alors Insérer (tas, y)
```

Figure 6.7 : Algorithme B&B de base

Dans un B&B meilleur d'abord, la structure de données communément utilisée est une *file de priorité* initialement vide. Les opérations de base sont :

- **deletemin** : elle correspond à la fonction *Extraire_sommet()* et supprime puis renvoie le sous-problème avec la plus grande priorité.
- **insert** : elle inclut dans la file de priorité les nouveaux sous-problèmes générés pour être traités ultérieurement.
- **deletegreater** : elle est utile à chaque amélioration de la borne supérieure et permet d'écarter tous les sous-problèmes dont l'évaluation dépasse la valeur de la borne supérieure. Elle correspond à la fonction *Elaguer()*.

6.3.2 Algorithme B&B parallèle

Tout comme pour la recherche tabou et la recherche IDA*, la parallélisation de l'algorithme B&B peut être générale ou spécifique. Une parallélisation spécifique consiste à paralléliser l'étape de génération de sous-problèmes et d'évaluation. Cependant, cette approche est non seulement dépendante du problème traité mais aussi présente des

limitations lorsque le processus de génération et d'évaluation ne nécessite pas beaucoup de ressources. Le surcoût engendré peut alors être très important.

L'approche souvent utilisée dans la littérature consiste plutôt à paralléliser l'exploration de l'espace de recherche. Dans ce cas, chaque processeur effectue de façon indépendante (séquentiellement) la procédure d'exploration sur une ou plusieurs parties de l'arbre de recherche. Les sommets en attente d'exploration peuvent être centralisés sur un seul processeur ou stockés localement.

6.3.2.a Modèle centralisé

Le modèle centralisé correspond à l'utilisation d'une seule liste commune contenant les sommets en attente d'exploration [Ber92]. Cette file de priorité globale est gérée par un seul processus - le serveur ou le maître. Ce processus serveur reçoit des requêtes provenant des processus esclaves, leur fournit des sous-problèmes à développer localement, et reçoit des sous-problèmes à insérer dans la file de priorité globale.

Les principaux problèmes posés par ce modèle sont les suivants :

- Structure de données complexe : ce problème se pose lors de l'implémentation de la file de priorité. En effet, non seulement elle nécessite un espace mémoire important mais aussi elle doit garantir un accès exclusif.
- Goulot d'étranglement : ce problème se pose lors d'une forte sollicitation du processus maître. Néanmoins, en plus d'un ajustement de la granularité, l'utilisation d'une strucutre de données parallèle pour un accès concurrent peut conduire à un meilleur comportement de l'application.

L'avantage essentiel d'une file de priorité globale est le bon partage des sommets prometteurs sur l'ensemble des processeurs ce qui conduit à un bon équilibrage de charge. De plus, un assouplissement de l'élagage des sommets non prometteurs est accompli grâce à la facilité d'accès à la borne supérieure après sa mise à jour.

6.3.2.b Modèle distribué

Le modèle distribué correspond à l'utilisation d'une file de priorité sur chaque processeur [DLCMM96]. Les sous-problèmes sont répartis sur l'ensemble des files de priorité et l'exploration se fait localement. Ceci permet de contourner le problème de goulot d'étranglement du modèle centralisé mais fait surgir d'autres problèmes :

- Équité du travail : du fait de l'irrégularité des sous-arbres générés en explorant les sous-problèmes, on est souvent confronté à un deséquilibre de charge entre les différents processeurs. Des mécanismes sophistiqués d'équilibrage de charge doivent donc être utilisés pour assurer un bon comportement de l'application.

Les opérations d'équilibrage de charge induisent un surcoût qui est parfois non négligeable.

- Terminaison : la détection de la terminaison est un problème crucial dans ce modèle totalement distribué.

6.3.3 Modèle parallèle adaptatif

Dans ce qui suit, nous allons détailler l'approche utilisée pour introduire le parallélisme adaptatif dans l'algorithme B&B. Nous avons opté pour une approche générale d'exploration de la recherche basée sur le modèle centralisé et ce pour deux raisons essentielles : le modèle correspond parfaitement au modèle maître/esclaves de l'environnement MARS, et l'équilibrage de charge est implicite. Ce modèle a été initialement implémenté sous PM^2 selon une stratégie non-adaptative [DLCMM96]. Nous avons repris cette application et nous l'avons adapté pour le modèle MARS.

Le rôle du processus maître est de gérer la file de priorité globale :

- Réception des requêtes des processus esclaves et envoi des sous-problèmes à traiter.
- Réception des sommets en attente d'exploration.
- Maintien et mise à jour de la borne supérieure.
- Elagage des sommets non prometteurs.
- Détection de la terminaison.

L'activité de chaque processus esclave se résume en ce qui suit :

- Demande d'un sommet en attente.
- Développement partiel du sous-arbre correspondant au sommet reçu.
- Envoi des sommets en attente au processus maître.
- Envoi de la nouvelle borne supérieure.

Le comportement de l'application B&B parallèle adaptative est similaire à celui de la recherche IDA* parallèle adaptative durant une itération. Au début de l'exécution, la file de priorité globale ne contient qu'un seul sommet correspondant au nœud racine de l'arbre. L'exécution de l'application parallèle passe par trois phases : la phase de démarrage dans laquelle le travail est initialement en cours de génération (alimentation de la file de priorité), la phase stationnaire correspondant à une activité de tous les processus esclaves, et la phase terminale qui reflète la fin de l'exécution de l'application. Le développement partiel par un processus esclave du sous-arbre correspondant au sommet

reçu est lié à un choix judicieux de la granularité. Une granularité trop fine conduit à des communications trop fréquentes avec le processus maître et des accès intensifs à la file de priorité globale, et une grosse granularité provoque une famine des processus esclaves dans la phase de démarrage et terminale. En effet, la granularité doit être relativement fine au début pour pouvoir remonter rapidement des sommets à explorer par d'autres processus esclaves. La granularité est par la suite ajustée durant la phase stationnaire pour éviter des communications trop fréquentes avec le processus maître et minimiser le nombre de sommets à insérer dans la file de priorité globale. Dans cet algorithme, la granularité est spatiale car elle représente le nombre de sommets à générer avant de communiquer avec le processus maître. Elle constitue un paramètre de l'algorithme.

Dans le cas où une nouvelle borne supérieure est trouvée, elle est diffusée via le processus maître à tous les processus esclaves pour effectuer un élagage local. L'application se comporte de la même façon que l'application IDA* en cas de dépli et de repli.

6.3.4 Application au problème d'affectation quadratique

Le problème test utilisé dans notre implémentation est le problème d'affectation quadratique décrit en section 5.3.1.

6.3.4.a Codification du problème

La file de priorité utilisée est du type « funnel table » (figure 6.8). Il s'agit d'un tableau statique dont les entrées correspondent aux valeurs allant de la borne inférieure à la borne supérieure. Pour chaque valeur, une file d'attente permet de regrouper tous les sous-problèmes ayant une même évaluation. Chaque file d'attente est gérée en FIFO.

L'opération *deletemin* est exécutée, en partant du premier élément du tableau, sur la première file d'attente non vide. L'opération *insert* place un sous-problème directement dans la file située à l'indice associé à son évaluation. Enfin, l'opération *deletegreater* consiste à vider toutes les files d'attente dont la valeur dépasse la nouvelle borne supérieure.

La funnel table est une structure de données « intelligente » dans le sens où elle intègre une stratégie d'ordonnancement. En effet, elle renvoie toujours le sommet le plus prioritaire (celui de faible coût) et insère correctement un sommet selon son évaluation.

Malgré l'efficacité de la funnel table (les opérations de base s'exécutent en O(1)) par rapport à d'autres structures de données (funnel-tree, skew-heap, D-heap, etc.), elle n'est adaptée qu'aux problèmes dont la différence entre la borne supérieure et la borne inférieure n'est pas très importante tels que les problèmes de type **nugxx** et **escxxx** de QAPlib.

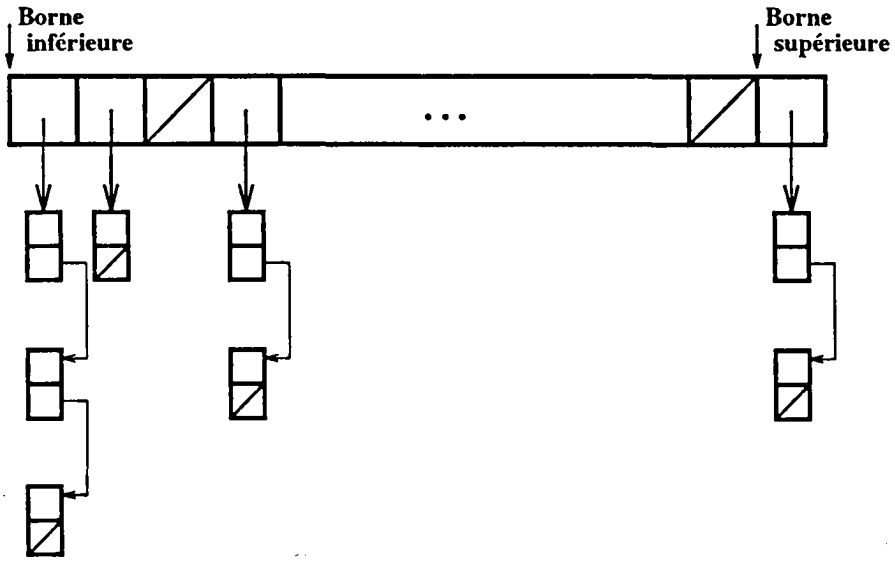


Figure 6.8 : La « Funnel Table »

6.3.4.b Evaluation des performances

Le tableau 6.2 montre les résultats obtenus sur quelques problèmes de QAPlib sur un méta-système de 30 machines. Ces résultats sont comparés avec ceux obtenus en utilisant l'IBM SP2 de 32 processeurs en mode mono-utilisateur, mono-application [DLCMM96].

Instance	Solution	Nœuds générés (IBM-SP2) (32 processeurs)	Temps d'exécution en secondes (IBM-SP2) (32 processeurs)	Nœuds générés (méta-système)	Temps d'exécution en secondes (méta-système)
Esc16a	68	525 285	59	510 308	310
Esc16b	292	10 221 885	333	10 238 920	1822
Esc16c	160	63 649 543	2087	63 647 124	10431
Esc16d	16	43 884 805	1793	43 879 901	9097
Esc16e	28	452 266	38	452 266	253

Tableau 6.2: Résultats pour quelques problèmes de QAPlib

6.4 Conclusion

La résolution de problèmes d'optimisation combinatoire pose toujours le problème de ressources de calcul. Le parallélisme adaptatif apparît comme une solution prometteuse pour accélérer la recherche d'une solution optimale d'une part, et d'autre part d'augmenter la taille des problèmes résolus.

Les résultats d'exécution sur des problèmes de grande taille sont en cours et nous permettront de trouver des solutions à des problèmes jusqu'alors non résolus.

Notons qu'une approche similaire à la notre a été utilisée dans une autre équipe de recherche [BMCP97]. Elle consiste en l'élaboration d'une bibliothèque spécifique à la résolution de problèmes d'optimisation combinatoire par la méthode de Bran-and-Bound. Elle intègre des mécanismes de sauvegarde au niveau applicatif et a permis non seulement d'obtenir de bons résultats mais aussi de trouver des solutions à des problèmes non encore résolus.

Conclusions et perspectives

La résolution de problèmes réputés « difficiles » notamment en optimisation combinatoire, en calcul scientifique et en imagerie est un défi perpétuel à cause de leur importance et leur vaste champ d'application. Les exigences de ces problèmes en matière d'espace mémoire et de temps machine poussent sans cesse les chercheurs à être en quête de performances. Les progrès technologiques accomplis sur des machines individuelles ne permettent pas de répondre une fois pour toute à cette attente. Si le parallélisme apporte une solution satisfaisante aux problème de performances, encore faut-il l'utiliser de façon efficace et non coûteuse. En effet, les machines parallèles les plus puissantes sont non seulement trop onéreuses mais aussi vite déclassées. Par conséquent, une autre alternative architecturale s'impose.

Cette alternative architecturale repose sur le fait que, depuis quelques années, on observe de plus en plus de ressources de traitement interconnectées : un méta-système. Les études réalisées autour de l'utilisation de ces méta-systèmes révèlent une sous-utilisation prépondérante et donc une perte considérable de temps de traitement. Pour bénéficier de la sous-utilisation des machines qui composent un méta-système, des environnements d'exécution et de programmation ont vu le jour. Cependant, la plupart de ces environnements proposent des fonctionnalités restreintes ne tenant pas compte des caractéristiques essentielles d'un méta-système à savoir l'hétérogénéité, l'opportunisme, l'aspect multi-utilisateurs, l'aspect propriété des machines et la fréquence élevée de pannes.

Face à une telle carence, nous avons pensé à développer un environnement adéquat mettant en œuvre le parallélisme adaptatif ou le parallélisme à degré variable en fonction de la charge et de l'utilisation interactive des machines.

Synthèse

Les travaux de cette thèse ont mis en valeur notre contribution aussi bien sur le plan *exécutif* qu'*applicatif*. Sur le plan exécutif, nous avons pu développer un environnement de programmation parallèle adaptative appelé MARS (Multi-user Adaptive Resource

Scheduler) qui répond aux principaux objectifs conceptuels définis dans le cahier de charge : portabilité (aucune modification au niveau du système d'exploitation), protection (aucun droit d'accès super-utilisateur), transparence (l'utilisateur n'a pas à gérer la disponibilité des ressources et à réguler la charge) et tolérance aux pannes (reconfiguration automatique et reprise de traitement). MARS permet d'une part d'exploiter la sous-utilisation des machines (en tenant compte du caractère personnel des stations de travail), et d'autre part de supporter le parallélisme adaptatif pour reconfigurer dynamiquement l'ensemble des processeurs supportant une application parallèle de type maître/esclaves. Ceci se traduit par un *dépli* (création de nouveaux processus) et un *repli* (retrait de processus) d'une application dynamiquement en fonction de l'état de charge du méta-système et de l'utilisation interactive des machines qui le composent.

L'originalité du nouveau concept sur lequel repose le modèle MARS, en l'occurence le *repli*, ne réside pas seulement dans le fait de retirer des processus au cours de leur exécution pour répondre à des contraites d'utilisation des machines. Le concept de *repli* est également à la base des événements suivants :

- Transfert de traitement : le repli élimine le recours à la migration de processus qui est une opération complexe à mettre en œuvre et quasiment inopérante dans un contexte hétérogène. En cas de réquisition d'une machine, plutôt que de migrer un processus, il suffit de le replier puis le déplier sur une autre machine. Cette opération de repli/dépli peut aussi être utilisée pour transférer un traitement d'une machine moins puissante vers une machine plus rapide en cas de disponibilité de celle-ci (exploitation de l'hétérogénéité). Ce cas de figure peut être observé lorsqu'une application est dans sa phase terminale d'exécution.

- Préemption de machines : le repli peut être utilisé pour des besoins d'ordonnancement. En effet, l'ordonnanceur MARS peut être amené à préempter une machine déjà allouée à une application pour pouvoir l'attribuer à une autre application pour garantir par exemple l'équité. Dans ce cas le repli est utilisé sans que la machine ne soit chargée ou utilisée interactivement.

- Tolérance aux pannes : un mécanisme de sauvegarde d'application (checkpointing) basé sur le repli a été développé dans l'équipe en vue de garantir une robustesse des applications vis-à-vis des pannes [KTG97]. Pour effectuer une sauvegarde d'une application, plutôt que de sauvegarder individuellement chaque processus, un repli de tous les processus esclaves d'une application est simulé et la sauvegarde est effectuée uniquement sur le processus maître. Ceci a l'avantage de contourner le problème de consistence, de réduire la taille du fichier de sauvegarde, et de pouvoir restaurer une application en tenant compte de l'état du méta-système au moment de la restauration.

- Equilibrage de charge intra-application : dans les applications où les processus esclaves génèrent du travail, le repli peut être effectué partiellement pour renvoyer au processus maître l'excès de travail en vue d'être traité par d'autres processus

esclaves. Ceci permet d'éviter le problème de famine de processus esclave et par conséquent de réaliser un équilibrage de charge intra-application implicite.

Sur le plan applicatif, nous avons présenté une méthodologie pour le développement d'applications parallèles adaptatives basée sur le modèle maître/esclaves. Le processus maître assure la distribution du travail et le contrôle de l'application. Les processus esclaves effectuent le traitement en cas de dépli et remettent le travail inachevé au processus maître en cas de repli. Nous avons montré comment générer du travail et comment ordonnancer les travaux élémentaires ou unités de travail dans certaines classes d'applications, et discuté de l'impact du choix de la granularité sur le comportement et les performances d'une application. Nous avons montré que pour pouvoir adapter la granularité à un environnement opportuniste hétérogène, il est judicieux d'opter pour une granularité hétérogène fine à moyenne déterminée à la demande. Le fait de déterminer la granularité à la demande permet non seulement de l'ajuster mais aussi conduit à un équilibre de charge intra-application implicite.

Pour appuyer notre démarche méthodologique, nous avons présenté en détail trois applications dans le domaine de l'optimisation combinatoire. La première fait partie des méta-heuristiques parallèles adaptatives et consiste en une recherche tabou appliquée au problème d'affectation quadratique. Dans cette application, le travail à faire est connu à priori, la granularité est grosse et la génération des unités de travail est effectuée par le processus maître. Les deux dernières applications font partie des méthodes exactes parallèles adaptatives et consistent respectivement en une recherche itérative en profondeur d'abord avec heuristique appliquée au jeu du taquin 15, et en une recherche de type Branch-and-Bound appliquée au problème d'affectation quadratique. Dans ces applications, le travail à faire est généré dynamiquement, la granularité est fine à moyenne et la génération des unités de travail est effectuée par les processus esclaves. Dans l'application de Branch-and-Bound, les processus esclaves coopèrent via le processus maître en s'échageant des informations concernant la mise à jour de la borne supérieure. Les résultats en termes d'adaptabilité des applications et de performances sont très satisfaisants.

MARS est opérationnel depuis près de deux ans et il est disponible à l'adresse `http://www.lifl.fr/~hafidi`. L'exécutif et la bibliothèque de primitives représentent plus de 10000 lignes de code. MARS est facilement installable et utilisable. Un squelette d'application a été mis en place pour permettre un développement aisé d'applications parallèles adaptatives.

Perspectives

L'importance de notre modèle adaptatif a permis d'ouvrir de nouveaux horizons notamment sur le plan collaborations avec d'autres équipes de recherche et l'extension de notre environnement.

Collaborations

Les collaborations déjà menées avec certaines équipes de recherche (équipe MAP du LIFL à Lille, équipe ALG de l'université du Littoral, University of Al-Akhawayn au Maroc) et qui ont abouti à de bons résultats nous incitent à maintenir notre intérêt pour les collaborations. Ainsi, des collaborations sont entreprises au niveau local, régional, national et international notamment dans le domaine de l'optimisation combinatoire (équipe PNN de l'UVSQ à Versailles, Institut de Physique Nucléaire à Orsay, IDSIA en Suisse) ainsi que dans le domaine du calcul scientifique (Université Centrale du Venezuela) et de l'imagerie (équipe VISC de l'université du Littoral).

Extensions de l'environnement

Les résultats prometteurs obtenus tant sur le plan exécutif qu'applicatif ne peuvent que nous encourager à faire évoluer de notre travail. L'évolution à court terme concerne :

- L'étude détaillée pour la détermination des indicateurs de charge adéquats pour les gestionnaires de nœuds.
- L'exploitation de l'hétérogénéité car dans l'état actuel des choses on utilise l'hétérogénéité mais on ne l'exploite pas pleinement.
- La détermination d'une stratégie d'ordonnancement inter-applications garantissant une maximisation d'utilisation des ressources et l'équité.
- Le portage de notre système sur de nouvelles architectures telle que Windows NT.
- L'évaluation des performances d'une implémentation parallèle adaptative d'une bibliothèque d'algorithmes génétiques LibGA effectuée dans le cadre d'un projet de maîtrise en collaboration avec l'équipe ALG de l'université du Littoral.
- L'étude de classes d'applications présentant des graphes de dépendances entre unités de travail. En effet, la présence d'une topologie de communication entre processus esclaves peut poser un certain nombre de problèmes dans un environnement opportuniste notamment dans le cas où un processus esclave désire communiquer avec un autre processus esclave qui n'existe plus suite à un repli. Les communications explicites basées sur des informations propres à un processus, tel que son identificateur, sont inadaptées à ce genre d'environnements car elles conduisent à un blocage de l'application. Pour résoudre ce problème, les communications doivent être anonymes comme c'est le cas dans l'environnement Piranha [GK91].

À moyen terme, notre vision du méta-système prendra une ampleur régionale voire nationale ou internationale en intégrant des machines goégraphiquement distantes comme l'illustre la figure 6.9. De nouveaux paramètres tels que la localité des données et les

performances du support de communication doivent être prises en compte. L'évolution vers un modèle hiérarchique du système tel qu'il a été décrit dans le chapitre 3 est évidente. Dans le souci d'une intrusion minimale, le fait de hiérarchiser les applications elles-mêmes en passant par des sous-maîtres permettra des prises de décisions rapides.

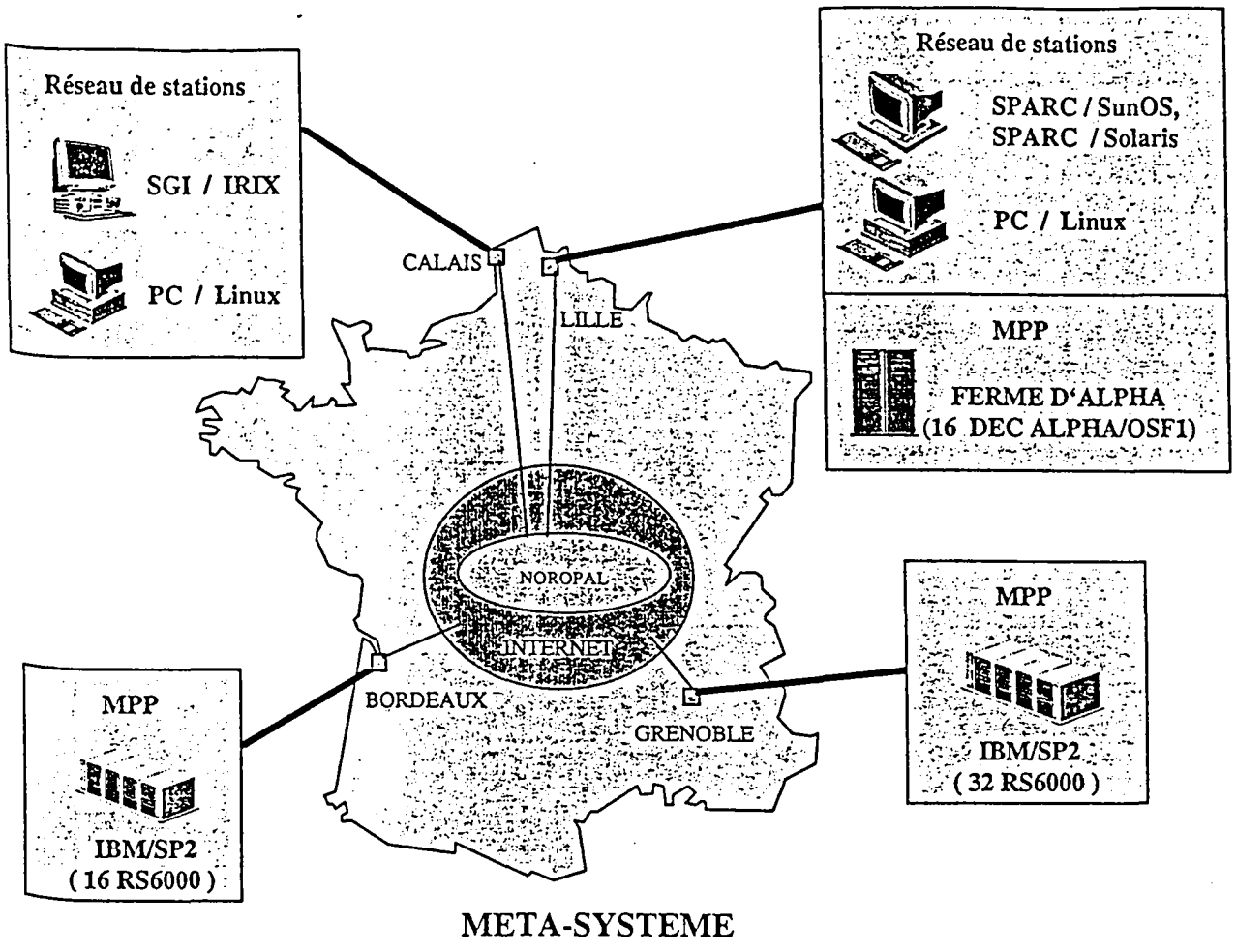


Figure 6.9 : Un méta-système à l'échelle nationale

Enfin, à long terme, nous envisageons de pencher vers un environnement basé sur Java qui possède la particularité d'être portable et sécurisant.

Annexes

Annexe A

A propos de la distribution MARS

DESIGNATION

MARS - Multi-user Adaptive Resource Scheduler Version 1

DESCRIPTION

MARS est un environnement destiné à la *programmation parallèle adaptative.* Il contrôle un ensemble de machines hétérogènes, et récupère le temps inutilisé pour le mettre à la disposition des applications parallèles de longue durée de vie.

Les différentes machines peuvent être des PCs, des stations de travail ou des MPPs. Elles peuvent être interconnectées par une variété de réseaux tels que Ethernet, FDDI, ATM et Myrinet.

Des démons (*marsnm*) assurent le contrôle de la charge des machines et de leur utilisation. Ils remontent ces informations à un démon maître appelé (*marsgs*). Ce dernier est responsable de l'allocation des ressources disponibles aux différentes applications.

Les applications écrites dans le langage C peuvent bénéficier des fonctionnalités de MARS via une bibliothèque de primitives appelée *libmars.a.*

ARCHITECTURES SUPPORTEES

Les architectures suivantes sont supportées par la distribution actuelle du système MARS :

ALPHA	DEC Alpha/OSF-1

LINUX	80[345]86 sous Linux
SUN4	Sun 4, 4c, sparc, etc.
SUN4SOL2	Sun 4 sous Solaris 2.x
SGI5	Silicon Graphics sous IRIX

INSTALLATION

La distribution actuelle de MARS contient les codes sources de l'exécutif et de la bibliothèque MARS ainsi que des exemples d'applications. Les fichiers de la distribution sont placés dans un répertoire appelé *mars*. Des installations pour des architectures différentes peuvent co-exister du fait que les exécutables sont placés dans des sous répertoires pour chaque architecture (ARCH) comme le fait PVM.

Les répertoires les plus importants sont :

Répertoire	contient
doc	documentation
examples	Quelques applications MARS ainsi qu'un squelette d'une application MARS (une application "vide" pour aider les développeurs MARS)
include	Fichier entête pour les applications MARS
lib/ARCH	Bibliothèque MARS (libmars.a)
man/man1	Pages en ligne du manuel (format nroff)
source	Sources de la bibliothèque libmars et des démons marsgs/marsnm

Avant de compiler et d'exécuter MARS, assurez-vous que PM2 a été correctement installé. Les variables d'environnement suivantes doivent être positionnées afin d'utiliser correctement le système MARS :

MARCEL_ROOT : Chemin où Marcel est installé (voir la documentation de Marcel).

PVM_ROOT : Chemin où PVM est installé (voir la documentation de PVM).

PM2_ROOT : Chemin où PM2 est installé (voir la documentation de PM2).

MARS_ROOT : Le chemin où le système MARS est installé, par exemple /usr/local/mars ou $HOME/mars. Cette variable doit être existante sur chaque machine utilisée par MARS. Pour cela, rajouter les lignes :

```
setenv MARS_ROOT ${HOME}/mars
```

ou

```
MARS_ROOT=$HOME/mars
export MARS_ROOT
```

respectivement dans le fichier *.cshrc* ou *.profile* selon le shell utilisé.

Si vous désirez des statistiques concernant l'exécutif MARS, changez la commande de compilation correspondante dans le fichier *common.mak* par la ligne en commentaire. Seul le compilateur GCC doit être utilisé.

Pour installer l'exécutif MARS ainsi qu'un exemple d'application, mettez vous dans un des sous répertoires du répertoire *examples/* et tapez *make*. Les exécutables seront placés directement dans le répertoire `${PVM_ROOT}/bin/$PVM_ARCH`, où PVM_ARCH désigne l'architecture (par exemple LINUX).

Vous pouvez rajouter le chemin `$MARS_ROOT/man` à la variable d'environnement MANPATH pour pouvoir exploiter les pages en ligne du manuel.

LANCER ET ARRETER MARS

Pour lancer MARS, exécutez le fichier `$PVM_ROOT/bin/$PVM_ARCH/marsgs`. Ceci crée un *marsnm* sur chaque nœud de la machine virtuelle PVM préalablement créée. D'autres machines peuvent être rajoutées/supprimées en utilisant les commandes "add"/"delete" de la console PVM. Dans le cas de suppression, vous devez placer les noms des machines à supprimer dans le fichier *mars/marsdelete* sinon les machines seront réintégrées dans le pool des nœuds MARS.

Pour arrêter l'exécutif MARS, utilisez la commande "reset" à partir de la console PVM.

VOIR AUSSI

`marsgs(1MARS), marsnm(1MARS)`

AUTEURS

Z. Hafidi, J-M. Geib, D. Kebbal et E-G. Talbi

LIFL - Université de Lille, FRANCE.

La page web `http://www.lifl.fr/~hafidi/` contient toutes les informations concernant le système MARS (package, papiers, etc). L'alias `mars-users@lifl.fr` regroupe tous les utilisateurs du système MARS et permet d'échanger des discussions, points de vue et autres.

DESIGNATION

`marsgs - MARS group server.`

DESCRIPTION

`marsgs` est un processus démon qui contrôle un ensemble de machines pouvant être hétérogènes et possédant éventuellement des systèmes de fichiers différents. Le pool de machines constitue une machine virtuelle créée par *pvm*.

Sur chaque machine, *marsgs* exécute un autre démon appelé *marsnm* qui remonte les changements d'état de la machine au *marsgs* . Les valeurs prédéfinies des états des machines sont :

- **IDLE** : machine disponible.
- **ACTIVE** : machine disponible mais si elle est allouée elle deviendra chargée.
- **BUSY** : machine chargée par des applications non-MARS.
- **OWNED** : machine utilisée de façon interactive. Pour MARS, ceci correspond à un état de charge maximale.
- **RUNNING** : machine allouée à des applications MARS.
- **LOADED** : machine allouée à des applications MARS et qui devient chargée.
- **SUSPENDED** : machine temporairement inutilisée par MARS suite à un redémarrage par exemple.
- **DELETED** : machine définitivement supprimée du pool de nœuds MARS.

marsgs est responsable de l'allocation des ressources disponibles aux applications. Il replie et déplie les applications MARS selon la fluctuation de charge dans la machine virtuelle.

VOIR AUSSI

`marsnm(1MARS), mars_intro(1MARS)`

DESIGNATION

`marsnm - MARS node manager.`

DESCRIPTION

`marsnm` est un processus démon qui assure un contrôle de la charge de la machine sur laquelle il s'exécute ainsi que son utilisation. Il remonte ces informations à un démon maître appelé *marsgs*.

marsnm demande périodiquement certaines informations au démon *rpc.rstatd* local, et détermine l'état de charge correspondant. Si celui-ci est différent de l'état précédent, il informe le *marsgs* duquel il dépend du changement d'état. Pour détecter la présence des utilisateurs interactifs, le *marsnm* contrôle l'activité du clavier et de la souris.

VOIR AUSSI

`marsgs(1MARS), mars_intro(1MARS), rpc.rstatd(1M)`

Annexe B

API de MARS

B.1 Enrôlement

DESIGNATION

mars_init - Enrolement dans le système MARS.

SYNTAXE

```
#include <mars.h>

void mars_init(module_status master_or_worker,
               mars_declare_func services)

enum module_status {
     MASTER = 0,
     WORKER = 1,
};
typedef enum module_status module_status;

typedef void (*mars_declare_func) ();
```

DESCRIPTION

La routine mars_init() informe le *marsgs* que ce processus s'enrôle dans le système MARS. Le processus appelant doit préciser s'il s'agit du processus maître (MASTER) ou esclave (WORKER) et donner un pointeur sur une fonction servant à déclarer les services LRPC.

mars_init() doit être la première primitive libmars appelée par tous les processus avant d'appeler tout autre routine.

ERREURS

mars_init() échoue dans les cas suivants :
1- le premier argument n'est pas MASTER ou WORKER.
2- le marsnm est absent.

DESIGNATION

`mars_exit - Quitte le système MARS.`

SYNTAXE

```
#include <mars.h>

void mars_exit(void)
```

DESCRIPTION

La routine `mars_exit()` informe le *marsgs* que ce processus est en train de quitter le système MARS. Cette routine termine aussi le processus appelant.

`mars_exit()` doit être appelée par tous les processus MARS pour terminer de façon propre.

Si `mars_exit()` est invoquée par un processus esclave, elle tue le *worker thread* après avoir tué tous les autres threads. Elle permet également d'envoyer certaines statistiques concernant l'exécution au processus maître qui se chargera de les stocker dans un fichier.

Le nombre total de replis et de déplis est également obtenu si la primitive `mars_exit()` a été appelée au niveau du processus maître.

Si le processus appelant n'est pas encore enrôlé dans le système MARS, son exécution sera tout simplement terminée.

VOIR AUSSI

`mars_stats(1MARS), mars_startworkerthread(1MARS), mars_waitexit(1MARS)`

B.2 Contrôle adaptatif de tâches

DESIGNATION

```
mars_spawn - Crée un nombre d'esclaves compris dans l'intervalle
             [Min, Max].
```

SYNTAXE

```
#include <mars.h>

int mars_spawn(char *WorkerModule, char **args, int Min, int Max)
```

DESCRIPTION

La routine `mars_spawn()` demande au système MARS de créer un nombre d'esclaves, appelés *WorkerModules*, compris dans l'intervalle [*Min* , *Max*] tout au long de l'exécution. Le jocker -1 signifie une valeur arbitraire. Cette routine doit être appelée une seule fois. Elle détermine le moment à partir duquel l'exécution adaptative commence.

MARS alloue à l'application un nombre d'esclaves égal au nombre de machines disponibles.

VALEURS RETOURNEES

En cas de succès, la routine retourne le nombre d'esclaves créés au moment de l'appel.

ERREURS

mars_spawn() échoue dans les cas suivants :
1- le processus appelant n'est pas le processus maître. *MarsCantSpawn* est retournée.
2- la routine est appelée plus d'une fois. *MarsSpawnCalled* est retournée.
3- l'intervalle n'est pas valide i.e *Min* >*Max* || *Min* <-1 || *Max* <-1 || *Max* ==0. *MarsBadRange* est retournée.
4- le nombre de machines requises est au delà du nombre maximum de nœuds présents dans le pool MARS. *MarsOutOfRes* est retournée.
5- il n'y a pas assez de ressources à allouer. *MarsNoEnoughRes* est retournée.

NOTES

```
Utilisée seulement au niveau du processus maitre.
```

B.3 Gestion du repli

DESIGNATION

`mars_sputbackwork_func - Positionne une fonction appelée au moment du repli.`

SYNTAXE

```
#include <mars.h>

void mars_sputbackwork_func(mars_putback_func putback, any_t arg)

typedef void (*mars_putback_func) (any_t arg);

any_t est définie dans pthread.h
```

DESCRIPTION

La routine `mars_sputbackwork_func()` positionne une fonction appelée automatiquement par MARS au moment du repli avec le paramètre *arg*. Cette primitive est responsable de l'envoi du travail intermédiaire, s'il existe, au processus maître. Ce travail doit être traité plus tard par un autre processus esclave.

La routine doit être appelée avant de demander tout travail au processus maître.

VOIR AUSSI

`mars_startworkerthread(1MARS)`

NOTES

`Utilisée seulement au niveau du processus maitre.`

DESIGNATION

mars_startworkerthread - Crée le worker thread.

SYNTAXE

```
#include <mars.h>

pthread_t mars_startworkerthread(mars_worker_thread wt, any_t arg,
pthread_attr_t *attr)

typedef any_t (*mars_worker_thread) (any_t arg);
```

pthread_t, any_t et pthread_attr_t sont définies dans pthread.h

DESCRIPTION

La routine mars_startworkerthread() crée le worker thread afin de traiter les unités de travail. Vous devez passer un pointeur sur le code du thread à créer, un pointeur sur ses arguments, et un dernier pointeur sur ses attributs.

Au moment du repli et après avoir appelé mars_exit(), ce thread sera le dernier thread tué.

VALEURS RETOURNEES

En cas de succès, le pid du thread créé est retourné.

VOIR AUSSI

mars_exit(1MARS)

NOTES

Utilisée seulement au niveau du processus esclave.

DESIGNATION

`mars_lock_fold - Bloque l'événement de repli.`

SYNTAXE

```
#include <mars.h>

void mars_lock_fold(void)
```

DESCRIPTION

La routine `mars_lock_fold()` permet de prévenir du repli. Puisque ce dernier peut surgir à tout moment, l'exécution du processus esclave dans certaines applications ne doit pas être interrompue n'importe où car le processus esclave peut se trouver dans un état inconsistant. Cela peut conduire donc à des incohérences des données renvoyées au processus maître.

En général, un traitement consiste en un cycle sur un certain nombre d'actions. Si le temps d'exécution d'une itération est très petit, alors le bloquage de l'événement de repli doit se faire autour de l'itération. Par contre, si le temps d'exécution d'une itération est trop long, alors vous pouvez utiliser soit le bloquage du repli à des endroits bien précis, ou bien travailler sur une copie des données. Dans ce dernier cas, l'événement de repli peut arriver à tout moment et concerne la dernière copie effectuée.

VOIR AUSSI

`mars_unlock_fold(1MARS)`

NOTES

`Utilisée seulement au niveau du processus esclave.`

DESIGNATION

`mars_unlock_fold - Autorise le repli.`

SYNTAXE

```
#include <mars.h>

void mars_unlock_fold(void)
```

DESCRIPTION

La routine `mars_unlock_fold` permet de prendre en compte l'événement de repli.

VOIR AUSSI

`mars_lock_fold(1MARS)`

NOTES

`Utilisée seulement au niveau du processus esclave.`

DESIGNATION

`mars_work_available` - Spécifie l'existence ou non de travail au niveau du processus esclave.

SYNTAXE

```
#include <mars.h>

void mars_work_available(int which)
```

DESCRIPTION

Du moment que l'envoi du travail intermédiaire au processus maître se fait automatiquement, la routine `mars_work_available()` spécifie si le processus esclave dispose ou non de travail préalablement reçu de la part du processus maître. Cette routine doit être appelée à chaque fois que le travail est traité et qu'un résultat est renvoyé au processus maître.

ERREURS

Une inconsistence peut surgir si l'appel à cette primitive n'est pas protégé contre le repli.

VOIR AUSSI

`mars_lock_fold(1MARS)`

NOTES

Utilisée seulement au niveau du processus esclave.

B.4 Terminaison

DESIGNATION

`mars_waitexit - Bloque le thread appelant jusqu'à ce que mars_exit() est appelée.`

SYNTAXE

```
#include <mars.h>

void mars_waitexit(void)
```

DESCRIPTION

La routine `mars_waitexit()` bloque le thread appelant. Elle est utilisée typiquement à la fin du code du thread *main* pour empêcher la fin d'exécution de tout le processus. Ce blocage est interrompu juste après un appel à *mars_ exit()* par un autre thread.

VOIR AUSSI

`mars_exit(1MARS)`

DESIGNATION

`mars_settestterm_func - Positionne une fonction appelée pour tester la fin de l'application.`

SYNTAXE

```
#include <mars.h>

void mars_settestterm_func(mars_testterm_func testterm)

typedef void (*mars_testterm_func) ();
```

DESCRIPTION

La routine `mars_settestterm_func()` positionne une fonction appelée automatiquement par MARS à chaque fois que le nombre d'esclaves atteint 0. Cette fonction servira à tester la fin de l'application.

NOTES

`Utilisée seulement au niveau du processus maitre.`

B.5 Messages d'erreurs

DESIGNATION

`mars_perror - Affiche un message d'erreur.`

SYNTAXE

```
#include <mars.h>

void mars_perror(char *msg)
```

DESCRIPTION

La routine `mars_perror()` affiche un message décrivant la dernière erreur retournée par un appel à une primitive MARS. La chaîne de caractères *msg* donnée par l'utilisateur est concaténée au message d'erreur MARS.

BUGS

`Puisque MARS est multithreadé, le message d'erreur peut refléter un événement autre que celui attendu.`

B.6 Statistiques

DESIGNATION

`mars_stats - Retourne des statistiques d'exécution.`

SYNTAXE

```
#include <mars.h>

void mars_stats(FILE *f, nodesalloc_struc *my_nodesalloc)

typedef struct{
        FILE *f;
        int step;
}nodesalloc_struc;
```

DESCRIPTION

La routine `mars_stats()` est utile si vous désirez faire des mesures de performances.

Si *f* est différent de *NULL*, il contiendra le temps d'exécution, le temps CPU, et le temps d'attente et de communication pour chaque processus esclave et pour toute l'application. D'autres informations sont également renvoyées telles que l'architecture utilisée, le nombre de déplis et le nombre de replis.

Si *my_nodesalloc* est différent de *NULL*, le nombre de machines allouées durant la durée de vie de l'application est sauvegardé dans le fichier *my_nodesalloc->f* toutes les *my_nodesalloc->step* secondes.

NOTES

`Utilisée seulement au niveau du processus maitre.`

B.7 Positionnement de paramètres

DESIGNATION

`mars_getparam - Retourne les valeurs des paramètres de libmars.`

SYNTAXE

```
#include <mars.h>

int mars_getparam(mars_params what)

enum mars_params {
     MarsMaxHosts          = 0,
     MarsFold              = 1,
     MarsUnfolding         = 2,
     MarsVerbose           = 3,
     MarsNbWorkers         = 4,
     MarsIdleNodes         = 5,
     MarsSendMail          = 6,
};
typedef enum mars_params mars_params;
```

DESCRIPTION

La routine `mars_getparam()` retourne la valeur du paramètre correspondant dans libmars. *what* est une valeur entière correspondant au paramètre. Les valeurs prédéfinies sont :

- **MarsMaxHosts** : Nombre maximum de machines gérées par MARS.
- **MarsFold** : Teste si le processus esclave est en repli.
- **MarsUnfolding** : Valeur du flag de dépli : continuer à être demandeur de ressources ou non.
- **MarsVerbose** : Les affichages de libmars sont autorisés/inhibés.
- **MarsNbWorkers** : Nombre courant de processus esclaves.
- **MarsIdleNodes** : Nombre de machines disponibles dans le méta-système.
- **MarsSendMail** : L'envoi de messages électroniques est autorisé/inhibé.

L'utilité des différents paramètres est présentée ci-dessous :

- **MarsMaxHosts** : Ce paramètre est utile si vous désirez connaître le nombre de nœuds présents dans le pool MARS.
- **MarsFold** : Ce paramètre est nécessaire pour savoir si un processus esclave est dans un état de repli après avoir bloqué longtemps l'événement de repli. Si c'est le cas, vous pouvez décider de débloquer le repli. Notez que la priorité UNIX du processus esclave est tout de même réduite au moment du repli pour éviter de gêner éventuellement les autres applications des autres utilisateurs.
- **MarsUnfolding** : Ce paramètre est utile dans plusieurs situations. Vous pouvez informer le système MARS que vous n'avez plus besoin de ressources car il n'y a plus de travail à faire (par exemple, à la fin de l'application). D'un autre côté, vous pouvez demander, durant l'exécution, au système MARS d'allouer des machines juste après un repli ou après avoir regénéré du travail. Par défaut, le dépli est autorisé (*MarsUnfolding = VRAI*).
- **MarsVerbose** : Ce paramètre est utile dans la phase de développement et de mise au point de l'application. libmars vous informe alors de certains événements, mises en garde ou erreurs. Par défaut, l'affichage est autorisé (*MarsVerbose = VRAI*).
- **MarsNbWorkers** : Ce paramètre donne le nombre instantané de processus esclaves et donc le degré de parallélisme courant de l'application. Vous pouvez réagir dans certaines situations telles qu'un nombre d'esclaves en dessous du minimum ou la reconfiguration de la topologie de communication.
- **MarsIdleNodes** : Ce paramètre est utile juste avant l'appel à la primitve *mars_spawn()* pour voir si vous pouvez disposer déjà d'un nombre de machines suffisant.
- **MarsSendMail** : Ce paramètre est utile lors de la soumission de votre application après la phase de développement et de mise au point. Le système MARS vous informe par messagerie électronique du moment et de l'état de terminaison de votre application. Par défaut, l'envoi de courrier est autorisé (*MarsSendMail = VRAI*) mais il est recommandé de l'inhiber dans la phase de développement et de mise au point afin d'éviter une avalanche de courriers électroniques.

VALEURS RETOURNEES

En cas de succès, la valeur du paramètre spécifié est renvoyée sinon *MarsBadParam* est retournée.

ERREURS

mars_ getparam() échoue dans les cas suivants :
1- le processus appelant ne s'est enrôlé dans MARS.
2- le paramètre n'est pas valide.

VOIR AUSSI

`mars_setparam(1MARS)`

DESIGNATION

`mars_setparam - Positionne des paramètres de libmars.`

SYNTAXE

```
#include <mars.h>

iint mars_setparam(mars_params what, int val)

enum mars_params {
     MarsMaxHosts          = 0,
     MarsFold              = 1,
     MarsUnfolding         = 2,
     MarsVerbose           = 3,
     MarsNbWorkers         = 4,
     MarsIdleNodes         = 5,
     MarsSendMail          = 6,
};
typedef enum mars_params mars_params;
```

DESCRIPTION

La routine `mars_setparam()` positionne la valeur du paramètre correspondant dans libmars. *val* est la nouvelle valeur de *what* qui représente le paramètre à modifier. Les valeurs prédéfinies sont :

- **MarsUnfolding** : Positionne la valeur du flag de dépli : continuer à être demandeur de ressources ou non.
- **MarsVerbose** : Autorise/inhibe les affichages de libmars.
- **MarsSendMail** : Autorise/inhibe l'envoie d'un message électronique lors de la terminaison de l'application.

Pour plus de détails sur les valeurs des paramètres, voir mars_getparam().

VALEURS RETOURNEES

En cas de succès, l'ancienne valeur du paramètre spécifié est retournée sinon la primitive renvoie *MarsBadParam.*

ERREURS

mars_setparam() échoue dans les cas suivants :

1- le processus appelant ne s'est enrôlé dans MARS.

2- le paramètre n'est pas valide.

VOIR AUSSI

`mars_getparam(1MARS)`

B.8 Synchronisation

DESIGNATION

`mars_lock_section - Bloque tous les événements MARS.`

SYNTAXE

```
#include <mars.h>

void mars_lock_section(void)
```

DESCRIPTION

La routine `mars_lock_section()` permet de bloquer tous les événements émanants du système MARS. Elle est typiquement utilisée par le processus maître dans des applications où les processus esclaves communiquent. Elle offre un état stable de l'application en dehors de tout repli ou dépli.

Cette routine peut être appelée , par exemple, avant d'appeler *mars_ getworkerstids()* pour déterminer la topologie de communication courante.

VOIR AUSSI

`mars_unlock_section(1MARS), mars_getworkerstids(1MARS)`

NOTES

`Utilisée seulement au niveau du processus maitre.`

DESIGNATION

`mars_unlock_section - Permet la prise en compte des événements MARS.`

SYNTAXE

```
#include <mars.h>

void mars_unlock_section(void)
```

DESCRIPTION

La routine `mars_unlock_section()` permet de prendre en compte tous les événements émanants du système MARS.

VOIR AUSSI

`mars_lock_section(1MARS)`

NOTES

`Utilisée seulement au niveau du processus maitre.`

B.9 Tolérance aux pannes

DESIGNATION

`mars_setworkerfault_func - Positionne une fonction appelée lors de la mort d'un processus esclave.`

SYNTAXE

```
#include <mars.h>

void mars_setworkerfault_func(mars_workerfault_func workerfault)

typedef void (*mars_workerfault_func) (any_t arg);
```

DESCRIPTION

La routine `mars_setworkerfault_func()` positionne une fonction appelée automatiquement par MARS à chaque fois qu'un processus esclave meurt. L'événement qui cause la mort est un événement autre que le repli. Cette routine aide l'utilisateur à détecter la mort des esclaves s'exécutant sur des machines défaillantes, et par conséquent de récupérer le travail délégué à l'esclave défaillant en vue de le traiter de nouveau. Ceci permettra de préserver une exécution cohérente de l'application.

NOTES

`Utilisée seulement au niveau du processus maitre.`

B.10 Divers

DESIGNATION

`mars_getworkerstids - Renvoie les tids des esclaves en exécution.`

SYNTAXE

```
#include <mars.h>

int mars_getworkerstids(int **tids)
```

DESCRIPTION

La routine `mars_getworkerstids()` retourne un tableau contenant les *tids* des processus esclaves en exécution.

Cette routine est nécessaire pour les applications avec des esclaves communicants. Elle permet de définir une topologie de communication changeante durant l'exécution.

L'allocation de l'espace mémoire nécessaire à la sauvegarde du tableau est faite par la routine.

VALEURS RETOURNEES

En cas de succès, une valeur positive est retournée indiquant le nombre courant de processus esclaves. Si la primitive échoue, *MarsCantGetTids* est retournée.

VOIR AUSSI

`mars_lock_section(1MARS)`

NOTES

`Utilisée seulement au niveau du processus maitre.`

Annexe C

Code source pour une application de recherche de nombres premiers

C.1 AMM

```
LRPC_SERVICE(LRPC_GET_WORK)
pthread_mutex_lock(&mutex); /* Access protected to pending work */
if ((res.work = get_work(req.tid)) == NULL) {
    res.which = NO_WORK;
    mars_setparam(MarsUnfolding, FALSE); /* I need no more resources */
    tfprintf(stderr, "GET_WORK: no work for [t%x]\n", req.tid);
   }
else
    res.which = WORK;
pthread_mutex_unlock(&mutex);
END_SERVICE(LRPC_GET_WORK)
/*-------------------------------------------------------------------*/
LRPC_SERVICE(LRPC_PUT_BACK_WORK)
pthread_mutex_lock(&mutex);
free_work(drop_active_work(req.work->tid)); /* Suppress previous work */
put_work(WITHOUT_COPY, req.work); /* and insert current one */
tfprintf(stderr, "PUT_BACK_WORK: receiving pending subrange [%u, %u] and
%d prime numbers from [t%x]\n", req.work->min, req.work->max, req.cnt,
req.work->tid);
primes_cnt += req.cnt; /* Primes found yet */
mars_setparam(MarsUnfolding, TRUE); /* I need more resources */
```

```
pthread_mutex_unlock(&mutex);
END_SERVICE(LRPC_PUT_BACK_WORK)
/*-------------------------------------------------------------------------*/
LRPC_SERVICE(LRPC_RESULTS)
pthread_mutex_lock(&mutex);
free_work(drop_active_work(req.tid));
/* Supress corresponding work unit */
tfprintf(stderr, "RESULTS: receiving %d prime numbers from [t%x]\n",
req.cnt, req.tid);
primes_cnt += req.cnt;
pthread_mutex_unlock(&mutex);
END_SERVICE(LRPC_RESULTS)
/*-------------------------------------------------------------------------*/
/* Worker death detection for fault tolerance */
void fault(any_t arg)
{
int tid = (int) arg;
PTR_WORK drop_work;

pthread_mutex_lock(&mutex);
if ((drop_work = drop_active_work(tid)) != NULL) { /* Find corresponding
     put_work(WITHOUT_COPY, drop_work);
/* work unit et insert it again */
     tfprintf(stderr, "\nFAULT TOLERANCE: processing again subrange [%u, %u]
lost by [t%x]\n\n", drop_work->min, drop_work->max, drop_work->tid);
    }
pthread_mutex_unlock(&mutex);
}
/*-------------------------------------------------------------------------*/
```

C.2 WM

```
/* Find all primes between min and max */
void find_primes(PTR_WORK my_work)
{
cnt = 0;
while (my_work->min <= my_work->max) {
       if (is_prime(my_work->min)) {
           mars_lock_fold(); /* Beginning of critical section */
           cnt++;
           my_work->min ++;
```

```
            mars_unlock_fold(); /* End of critical section */
          }
        else
            my_work->min ++;
      }

/* Sending results to the master process. This must be protected from
 * folding otherwise inconsistence may occur by sending the results twice
 */
mars_lock_fold();
RESULTS_req.cnt = cnt;
ASYNC_LRPC(Parent_tid, LRPC_RESULTS, STD_PRIO, MY_STACK_SIZE, &RESULTS_req);

mars_work_available(FALSE); /* Don't forget: now we have no work */
mars_unlock_fold();
}
/*-------------------------------------------------------------------------*/
/* If we have work then it will be sent else no work received yet and nothing
 * will be sent. The checking for work availability is hidden to the user.
 * This function is automatically called at fold time
 */
void cleanup(any_t arg)
{
PUT_BACK_WORK_req.work = my_work; /* Put back remainder subrange ... */
PUT_BACK_WORK_req.cnt = cnt; /* ... and numer of primes found yet */
ASYNC_LRPC(Parent_tid, LRPC_PUT_BACK_WORK, STD_PRIO, MY_STACK_SIZE,
&PUT_BACK_WORK_req);
tfprintf(stderr, "FOLDING [%x], min=%u and max=%u, %u prime numbers
found yet\n", pthread_self(), my_work->min, my_work->max, cnt);
}
/*-------------------------------------------------------------------------*/
/* This is the "worker thread" */
any_t wt(any_t arg)
{
/* Always me ! */
GET_WORK_req.tid = RESULTS_req.tid = pm2_self();

/* Set cleanup function */
mars_sputbackwork_func(cleanup, NULL);

for (;;) {
     /* Get work from the master process; this must be protected from folding
      * otherwise the incoming data may be lost
```

```
      */
     mars_lock_fold();
     LRPC(Parent_tid, LRPC_GET_WORK, STD_PRIO, MY_STACK_SIZE, &GET_WORK_req,
&GET_WORK_res);

     /* If no work then die */
     if (GET_WORK_res.which == NO_WORK)
         mars_exit();

     /* We have work */
     my_work = GET_WORK_res.work;
     my_work->tid = pm2_self();
     mars_work_available(TRUE);

     mars_unlock_fold();


     tfprintf(stderr, "receiving range [%u, %u], searching for prime
numbers...\n", my_work->min, my_work->max);
     find_primes(my_work);
    }
}
/*-------------------------------------------------------------------------*/
```

C.3 Gestion du travail

```
#define NO_WORK 0
#define WORK 1

typedef struct {
int tid; /* Used as a key search */
u_long min;
u_long max;
} TYPE_WORK, *PTR_WORK;

typedef struct TYPE_PENDING_WORK {
PTR_WORK work;
struct TYPE_PENDING_WORK *next;
} TYPE_PENDING_WORK, *PTR_PENDING_WORK;

#define WITHOUT_COPY 0
```

```
#define WITH_COPY 1

/* Granularity of work is determined dynamically */
#define WORK_UNIT() \
(5*mars_getparam(MarsNbWorkers)) \

u_long next_min, max;
int NoWork = FALSE;

static PTR_PENDING_WORK pending_work = NULL, active_work = NULL;


/*-------------------------------------------------------------------*/
void alloc_work(PTR_WORK *new_work)
{
(*new_work) = (PTR_WORK) tmalloc(sizeof(TYPE_WORK));
}
/*-------------------------------------------------------------------*/
void copy_work(PTR_WORK dst_work, PTR_WORK src_work)
{
dst_work->tid = src_work->tid;
dst_work->min = src_work->min;
dst_work->max = src_work->max;
}
/*-------------------------------------------------------------------*/
void free_work(PTR_WORK work)
{
tfree(work);
}
/*-------------------------------------------------------------------*/
void put_work(int flag, PTR_WORK work)
{
PTR_PENDING_WORK new;

new = (PTR_PENDING_WORK) tmalloc(sizeof(TYPE_PENDING_WORK));
new->next = pending_work; /* Insert a new element in "pending" list */
pending_work = new;

/* Insert work */
if (flag == WITHOUT_COPY)
    new->work = work;
else {
    alloc_work(&(new->work));
```

```
    copy_work(new->work, work);
   }
NoWork = FALSE;
}
/*--------------------------------------------------------------------*/
PTR_WORK get_work(int tid)
{
if (!pending_work) { /* Any work from folded workers? */
    if (next_min < max) {
        PTR_WORK work;

        alloc_work(&work);
        work->tid = tid;
        work->min = next_min; /* Split remainder range */
        work->max = next_min + (max - next_min)/WORK_UNIT();
        if (work->max == work->min)
            work->max = next_min = max;
        else
            next_min = work->max + 1;

        tfprintf(stderr, "GET_WORK: sending new subrange [%u, %u] to
 [t%x]\n", work->min, work->max, tid);
        put_active_work(WITHOUT_COPY, work);
        return work;
       }
    else {
        NoWork = TRUE;
        return NULL; /* There's no work to do */
       }
   }
else {
    /* Get work from the linked list (many strategies) */
    PTR_PENDING_WORK pwork = pending_work;
    PTR_WORK work = pending_work->work;

    pending_work = pending_work->next;
/* Delete work from "pending" list and ... */
    tfree(pwork);
    work->tid = tid;
    tfprintf(stderr, "GET_WORK: sending pending subrange [%u, %u] to
[t%x]\n", work->min, work->max, tid);
    put_active_work(WITHOUT_COPY, work);
/* ... put it into "active" list */
```

```
    return work;
   }
}
/*-------------------------------------------------------------------------*/
void put_active_work(int flag, PTR_WORK work)
{
PTR_PENDING_WORK new;

new = (PTR_PENDING_WORK) tmalloc(sizeof(TYPE_PENDING_WORK));
new->next = active_work; /* Insert a new element in "active" list */
active_work = new;

/* Insert work */
if (flag == WITHOUT_COPY)
    new->work = work;
else {
    alloc_work(&new->work);
    copy_work(new->work, work);
   }
}
/*-------------------------------------------------------------------------*/
static PTR_WORK drop_work(PTR_PENDING_WORK *awork, int tid)
{
if (!(*awork))
    return NULL;

if ((*awork)->work->tid == tid) { /* If there's work processed by tid */
    PTR_PENDING_WORK drop_awork = (*awork); /* then drop it out */
    PTR_WORK work = (*awork)->work;

    (*awork) = (*awork)->next;
    tfree(drop_awork); /* Delete element in "active" list */
    tfprintf(stderr, "Dropping out subrange [%u, %u] processed by [t%x]\n",
work->min, work->max, tid);
    return work;
   }
return drop_work(&((*awork)->next), tid);
}
/*-------------------------------------------------------------------------*/
PTR_WORK drop_active_work(int tid)
{
return drop_work(&active_work, tid);
}
```

Bibliographie

[ABB+86] Accetta (M.), Baron (R.), Bolosky (W.), Golub (D.), Rashid (R.), Tevanian (A.) et Young (M.). – Mach : A new kernel foundation for Unix development. *Proceedings Summer Usenix*, juillet 1986, pp. 93–112.

[ACF86] Artsy (Y.), Chang (H. Y.) et Finkel (R.). – *Processes migrate in Charlotte.* – Rapport technique n No.655, University of Wisconsin-Madison, août 1986.

[AE87] Agrawal (R.) et Ezzat (A. K.). – Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, vol. 13, n 8, août 1987, pp. 905–912.

[AF89] Artsy (Y.) et Finkel (R.). – Designing a process migration facility: The Charlotte experience. *IEEE Computer*, vol. 22, n 9, septembre 1989, pp. 47–56.

[AHU74] Aho (A. V.), Hopcroft (J. E.) et Ullman (J. D.). – *The design and analysis of computer algorithms.* – Reading, MA:Addison-Wesley, 1974.

[ALM+91] Atallah (M. J.), Lock (C.), Marinescu (D. C.), Siegel (H. J.) et Casavant (T. L.). – Co-Scheduling Compute-Intensive Tasks on a Network of Workstations. *In : Proceedings 11th International Conference on Distributed Computing Systems*, pp. 344–352.

[Aru94] Arumughum (B.R. Seyfarth J.L. Bickham M.). – *Glenda Installation and Use.* – Rapport technique, 1994.

[ASOW] Al-Saqabi (K.), Otto (S. W.) et Walpole (J.). – *Gang Scheduling in Heterogeneous Distributed Systems.* – Rapport technique, Oregon Graduate Institute, Portland.

[BDG+93] Beguelin (A.), Dongarra (J. J.), Geist (G. A.), Jiang (W.), Manchek (R.), Moore (K.) et Sunderam (V. S.). – *The PVM Project.* – Ornl/tm-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, mai 1993.

[Ber92] Bernard (M.). – *Contribution à l'algorithmique non numérique parallèle : Parallèlisations de méthodes de recherche arborescente.* – Thèse de PhD, Paris VI, 1992.

[BF96] Bernard (G.) et Folliot (B.). – Caractéristiques Générales du Placement Dynamique : Synthèse et Problématique. *In : Ecole Française de Parallélisme, Réseaux et Systèmes, Presqu'île de Giens, France*, pp. 3–22.

[BFY96] Baker (M. A.), Fox (G. C.) et Yau (H. W.). – *A Review of Commercial and Research Cluster Management Software.* – Rapport technique, Syracuse University, New York, juin 1996.

[BGG+95] Badeau (P.), Gendreau (M.), Guertin (F.), Potvin (J-Y.) et Taillard (E.). – A parallel tabu search heuristic for the vehicle routing problem with time windows. *RR CRT-95-84, Centre de Recherche sur les Transports, Université de Montréal*, décembre 1995.

[BHPT98] Bachelet (V.), Hafidi (Z.), Preux (P.) et Talbi (E. G.). – Vers la coopération de méta-heuristiques parallèles. *Calculateurs parallèles*, vol. 10, n 2, mai 1998.

[BKKW94] Baratloo (A.), Karaul (M.), Kedem (Z.) et Wyckoff (P.). – Charlotte: Metacomputing on the web. pp. 181–188.

[BL91] Briker (A.) et Litzkow (M. J.). – *Condor technical summary.* – Rapport technique, University of Wisconsin, 1991.

[BL92] Butler (R.M.) et Lusk (E.L.). – *User's guide to the P4 Programming System.* – Rapport technique nANL-92/17, Argonne National Laboratory, Argonne, IL, 1992.

[BLA+94] Blumrich (M.), Li (K.), Alpert (R.), Dubnicki (C.), Felten (E.) et Sandberg (J.). – A virtual memory mapped network interface for the Shrimp multicomputer. *In : Proceedings of 21st Annual International Symposium on Computer Architecture.*

[BLL93] Butler (R. M.), Leveton (A. L.) et Lusk (E. L.). – P4-Linda: A Portable implementation of Linda. *In : Proceedings of 2th International Symposium on High Performance Computing*, éd. par IEEE, pp. 50–58. – Washington, USA, juillet 1993.

[BM89] Birman (K.) et Marzullo (K.). – ISIS and the meta project. *Sun Technology*, 1989.

[BMCP97] Brungger (A.), Marzetta (A.), Clausen (J.) et Perregaard (M.). – Joining forces in solving large-scale quadratic assignment problems in parallel. *In : 11th International Parallel Processing Symposium*, éd. par Gottlieb (A.). – Geneva, Switzerland, avril 1997.

[BMR82] Brownbridge (D.), Marshall (L.) et Randell (B.). – The NewCastle connection or UNIXes of the World Unite! *Software Practice and Experience*, vol. 12, 1982, pp. 1147–1162.

[BNK89] Borghoff (U. M.) et Nast-Kolb (K.). – *Distributed Systems: A Comprehensive Survey.* – Rapport technique nTUM-I8909, University of Munich, novembre 1989.

[BSFG94] Barcellos (A.), Shramm (J.), Filho (V.) et Geyer (C.). – The HetNOS Network Operating System: a tool for writing distributed applications. *Operating Systems Review*, 1994.

[BSR91] Burkard (R. E.), S.Karisch et Rendl (F.). – Qaplib: A quadratic assignment problem library. *European Journal of Operational Research*, vol. 55, 1991, pp. 115–119.

[BSS91] Bernard (G.), Stève (D.) et Simatic (M.). – Placement et migration de processus dans les systèmes répartis faiblement couplés. *T.S.I. Technique et Science Informatique*, vol. 10, n5, novembre 1991, pp. 375–392.

[BZS93] Bershad (B. N.), Zekauskas (M. J.) et Sawdon (W. A.). – *The Midway distributed shared memory system.* – Rapport technique nCMU-CS 93-119, University of Carnegie Mellon, Pittsburg, PA, 1993.

[Ca89] Chase (J. S.) et al. – The Amber system: Parallel programming on a network of multiprocessors. *ACM Operating system Review*, vol. 23, n5, décembre 1989, pp. 147–158.

[Cab86] Cabrera (L. F.). – The influence of workload on load balancing strategies. *In: Proceedings Usenix Summer'86, Atlanta, Georgia*, pp. 446–458.

[Car92] Carlson (W.). – *RES: A simple system for distributed computing.* – Src-tr-92-067, Supercomputing Research Center, 1992.

[CCK+95] Casas (J.), Clark (D.), Konuru (R.), Otto (S.), Prouty (R.) et Walpole (J.). – *MPVM: a Migration Transparent Version of PVM.* – Rapport technique, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Thechnologiy PO BOX 91000 Portland, OR97291-1000, USA, février 1995.

[CCRG95] Christaller (M.), Castaneda-Retiz (MR.) et Gautier (T.). – Control Parallelism on top of PVM: The Athapascan Environment. *In: EuroPVM'95 proceedings, Hermes , Lyon.*

[CDM91] Colorni (A.), Dorigo (M.) et Maniezzo (V.). – Distributed Optimization by Ant Colonies. *In: Proceedings European Conference Artificial Life*, éd. par Publishing (Elsevier), pp. 134–142.

[CG89] Carriero (N.) et Gelernter (D.). – Linda in context. *Communications of the ACM*, vol. 32, n4, avril 1989, pp. 444—558.

[CGAS89] Chakrabarti (P. P.), Ghose (S.), Acharya (A.) et Sarkar (S. C. De). – Heuristic search in restricted memory. *Artificial Intelligence*, vol. 41, n2, 1989, pp. 197–221.

[Che88] Cheriton (D. R.). – The V distributed system. *Communications of the ACM*, vol. 31, n3, mars 1988, pp. 314–333.

[Che92] Cheong (F. C.). – *OASIS: An agent-oriented programming language for heterogeneous distributed environment.* – Thèse de PhD, University of Michigan. School of Computer Science and Engeneering, 1992.

[CM92] Clark (H.) et McMillin (B.). – DAWGS–a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, vol. 14, 1992, pp. 175–186.

[CMMT97] Cung (V-D.), Mautor (T.), Michelon (P.) et Tavares (A.). – Improving the Efficiency of Scatter Search. *In: 2nd Metaheuristics International Conference MIC'97.* – Sophia-Antipolis, France, juillet 1997.

[Cor94] Corporation (Platform Computing). – Load Sharing Facility: User's and Administrator's Guide. *Toronto, Canada*, février 1994. – `http://www.platform.com/products/overview.html`.

[CP95] Craysoft Publications (Cray Research Inc). – Introducing NQE-IN-2153 2.0. mai 1995. – `http://www.cray.com/PUBLIC/product-info/craysoft/WLM/NQE/NQE.html`.

[Cra90] Crandall (R.). – Tales of Godzilla: Adventures in distributed computing. *NeXTon Campus*, 1990.

[CS93] Cap (C.) et Strumpen (V.). – Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 1993.

[CSK93] Chakrapani (J.) et Skorin-Kapov (J.). – Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, vol. 41, 1993, pp. 327–341.

[CTG93] Crainic (T. D.), Toulouse (M.) et Gendreau (M.). – Towards a taxonomy of parallel tabu search algorithms. – Rapport technique nCRT-933, Centre de Recherche sur les Transports, Université de Montreal, septembre 1993.

[Cun94] Cung (V-D.). – *Contribution à l'algorithmique non numérique parallèle : Exploration d'espaces de recherche.* – Thèse de PhD, Université de Paris VI, avril 1994.

[DCM+90] Dasgupta (P.), Chen (R.), Menon (S.), Pearson (M.), Ananthanarayanon (R.), Ramachandran (U.), Ahamad (M.), LeBlanc (R.), Applebe (W.), Bernabeu-Auban (J.), Hutto (P.), Khalidi (M.) et Wilkenloh (C.). – The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, vol. 3, n 1, 1990.

[DFJ54] Dantzig (G. B.), Fulkerson (D. R.) et Johnson (S. M.). – On linear programming, combinatorial approach to the traveling-salesman problem. *Operations Research*, vol. 7, 1954, pp. 58–66.

[DJM93] The Distributed Job Manager Administration Guide. *Minnesota Supercomputing Center, Inc*, mai 1993. – `http://www.msc.edu/`.

[DLCMM96] Denneulin (Yves), Le Cun (Bertrand), Mautor (Thierry) et Méhaut (Jean-François). – Distributed branch and bound algorithms for large quadratic assignment problems. *In : 5th Computer Science Technical Section on Computer Science and Operations Research.* – Dallas-USA, Janvier 1996.

[DNGM96] Denneulin (Y.), Namyst (R.), Geib (J-M.) et Méhaut (J-F.). – LBMP: Load-Balancing with Migration directed by Priorities. *In : ESPPE'96 proceedings, Alpe d'Huez, France*, pp. 115–118.

[DO91] Douglis (F.) et Ousterhout (J.). – Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice and Experience*, vol. 21, août 1991, pp. 757–785.

[Dow95] Dowadji (S.). – *Contribution à l'étude des problèmes d'équilibrage de charge dans des environnements distribués.* – Thèse de PhD, Université Paris 6, Juillet 1995.

[DP83] Dechter (R.) et Pearl (J.). – The optimality of A* revisited. *In : Proceedings of the National Conference on Artificial Intelligence, Washington*, pp. 95–99.

[DVCL93] Deconinck (G.), Vounckx (J.), Cuyvers (R.) et Lauwereins (R.). – *Survey of Checkpointing and Rollback techniques.* – Rapport technique n 03.1.8 and 03.1.12, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, juin 1993.

[Ebe90] Ebeling (C.). – *All The Right Moves.* – MIT Press, Cambridge, MA, 1990.

[EHM90] Evett (M.), Hendler (J.) et Mahanti (A.). – PRA*: a memory-limited heuristic search procedure for the connection machine. *Proceedings Third Symposium on the Frontiers of Massively Parallel Computation*, 1990, pp. 145–149.

[Ezz86] Ezzat (A. K.). – Load balancing in NEST: A network of workstations. *In : Proceedings ACM/IEEE Fall Joint Computer Science, Dallas, Texas*, pp. 1138–1149.

[FKB91] Flower (J.), Kolawa (A.) et Bharadwaj (S.). – The Express way to distributed processing. *Supercomputing Review*, mai 1991, pp. 54–55.

[FKOT94] Foster (I.), Kesselman (C.), Olson (R.) et Tuecke (S.). – *Nexus: An interoperability toolkit for parallel and distributed computer systems.* – Rapport technique nANL/MCS-TM-189, Argonne National Laboratory, 1994.

[FM90] Frye (R.) et Myczkowski (J.). – Exhaustive search of unstructured trees on the Connection Machine. *Thinking Machines Corporation Tech. Rept.*, vol. TMC-196, 1990, p. Cambridge MA.

[Fol92] Folliot (B.). – *Méthodes et outils de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes.* – Thèse de PhD, Université Paris 6, Institut Blaise Pascal, décembre 1992.

[For94] Forum (Message Passing Interface) (édité par). – *Document for a Standard Message Passing Interface.* – Draft, 1994.

[GHM+95] Geib (J-M.), Hémery (F.), Méhaut (J-F.), Roos (J-F.) et Talbi (E. G.). – PVC : Un environnement de programmation parallèle avec régulation de charge. *Calculateurs parallèles*, vol. 7, n2, 1995, pp. 139–158.

[GJ79] Garey (M.) et Johnson (D.). – *Computers and Intractability: A guide to the theory on NP-completeness.* – New York, W.H. Freeman and Co. Publishers, 1979.

[GK91] Gelernter (D.) et Kaminsky (D. L.). – Supercomputing out of recycled garbage: Preliminary experience with Piranha. *In: 6th ACM International Conference on Supercomputing.*

[GL92] Glover (F.) et Laguna (M.). – *Modern Heuristic Techniques for Combinatorial Problems*, chap. Tabu search, pp. 70–150. – Blackwell Scientific Publications, 1992.

[Glo86] Glover (F.). – Future paths for integer programming and links to artificial intelligence. *Computing Operations Research*, vol. 13, n5, 1986, pp. 533–549.

[GMW] Gittings (C. J.), Morgan (J. S.) et Wetherall (J.). – Far - A Tool for Exploiting Space Workstation Capacity. *draft paper.* – `http://www.liv.ac.uk/HPC/farHomepage.html`.

[GPR+97] Ghormley (D. P.), Petrou (D.), Rodrigues (S. H.), Vahdat (A. M.) et Anderson (T. E.). – *GLUnix: a Global Layer Unix for a Network of Workstations.* – Rapport technique, University of California, Berkeley, août 1997. http://now.cs.berkeley.edu/Glunix/glunix.html.

[Gre95] Green (T. P.). – DQS 3.0.2 Readme/Installation Manual. *Supercomputer Computations Research Institute, Florida State University, Tallassee*, juillet 1995. – http://www.scri.fsu.edu/~pasko/dqs.html.

[GT93] Graf et Terrance (P.). – HP Task Broker: A Tool for Distributing Computational Tasks. *Third Cluster Workshop, Florida State University*, décembre 1993. – http://www.hp.com/.

[Haf94] Hafidi (Z.). – *Parcours parallèle IDA* sous PVM.* – Mémoire de DEA, Université de LILLE1 (LIFL), 1994.

[Hag86] Hagmann (R.). – Process servers: Sharing processing power in a workstation environment. *In: IEEE International Conference on Distributed Computing systems, Cambridge*, pp. 260–267.

[Har91] Harrison (R. J.). – Portable tools and applications for parallel computers. *International Journal Quantum Chemistry*, no40, 1991, pp. 847–863.

[HdW89] Hertz (A.) et de Werra (D.). – The tabu search metaheuristic: How we use it ? *Annals of Mathamatics and Artificial Intelligence*, 1989, pp. 111–121.

[Her] Herbert (S.). – Generic NQS - Free Batch Processing for UNIX. *Academic Computing Services, The University of Sheffield, UK.* – http://www.shef.ac.uk/uni/projects/nqs/Products/Generic-NQS/v3.4x/.

[HNR68] Hart (P. E.), Nilsson (N. J.) et Raphael (B.). – A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Sciences and Cybernetics*, vol. 4, n2, 1968, pp. 100–107.

[Hol75] Holland (J. H.). – *Adaptation in natural and artificial systems.* – Ann Arbor, MI, Michigan Press University, 1975.

[HS93] Hartley (C.) et Sunderam (V. S.). – Concurrent programming with shared objects in networked environments. *In: Proceedings of 7th International Parallel Processing Symposium*, pp. 471–478.

[HT95] Henderson (R. L.) et Tweten (D.). – *Portable Batch System: Requirement Specification.* – Rapport technique n NAS Technical Report, NASA Ames Research Center, avril 1995. http://www.nas.nasa.gov/NAS/Projects/pbs/.

[HTG95] Hafidi (Z.), Talbi (E. G.) et Goncalvez (G.). – Load balancing and parallel tree search: the mpida* algorithm. *In : International Conference on Parallel Computing ParCo'95*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trystram (D.). pp. 93–100. – Gent, Belgique, septembre 1995.

[JCG+95] J.Casas, Clark (D.), Galbiati (P.), Konuru (R.), Otto (S.), Prouty (R.) et Walpole (J.). – MIST: PVM with Transparent Migration and Checkpointing. *Presented at the 3rd Annual PVM user's Group Meeting Pittsburgh, PA*, May 7-9 1995.

[JXT93] Ju (J.), Xu (G.) et Tao (J.). – Parallel Computing using Idle Workstations. *Operating Systems Review*, 1993, pp. 87–96.

[KB57] Koopmans (T. C.) et Beckmann (M. J.). – Assignment problems and the location of economic activities. *Econometrica*, vol. 25, 1957, pp. 53–76.

[KC91] Krueger (P.) et Chawla (R.). – The Stealth distributed scheduler. *In : International Conference on Distributed Computing Systems.*

[KGV83] Kirkpatrick (S.), Gelatt (C. D.) et Vecchi (M. P.). – Optimization by simulated annealing. *Science*, vol. 220, n4598, mai 1983, pp. 671–680.

[Kin89] Kingston (D. P.). – *A Tour Through the Multi-Device Queuing Systems Revised for MDQS 2.12.* – Rapport technique nTechnical paper, Ballistics Research Laboratory, USA, février 1989. `ftp://ftp.arl.mil/arch/`.

[KLS86] Kronenberg (N.), Levy (H.) et Strecker (W.). – VAXClusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, vol. 4, n2, mai 1986, pp. 130–146.

[KN94] Kaplan (J. A.) et Nelson (M. L.). – *A Comparison of Queueing, Cluster and Distributed Computing Systems.* – Rapport technique nTM 109025, NASA Langely Research Center, juin 1994.

[Kob97] Kobler (D.). – A new Strategy in Evolutionary Algorithms. *In : 2nd Metaheuristics International Conference MIC'97.* – Sophia-Antipolis, France, juillet 1997.

[Kor85] Korf (Richard E.). – Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 1985, pp. 97–109.

[KRR95] Keeton (K.), Rodrigues (S.) et Roselli (D.). – *Previous Work in Distributed Operating Systems.* – Rapport technique, NOW Retreat, avril 1995.

[KTG97] Kebbal (J.), Talbi (E. G.) et Geib (J-M.). – A new approach for checkpointing parallel adaptive applications. *In : Proceedings of the international conference on parallel and distributed processing technique and applications (PDPTA'97)*, pp. 1643–1651.

[KTG98] Kebbal (D.), Talbi (E-G.) et Geib (J-M.). – Ordonnancement multi-applications. *In : Proceedings de JRPRC2, Deuxièmes Journées de Recherche sur le Placement Dynamique et la Répartition de Charge, Lille, France.*

[LC96] Le Cun (Bertrand). – *Des structures de données parallèles.* – Thèse de PhD, Université Paris VI - Pierre et Marie Curie, Janvier 1996.

[Lel90] Leler (Wm). – Linda Meets Unix. *IEEE Computer*, février 1990, pp. 43–54.

[LHR95] Lifka (D. A.), Henderson (M. W.) et Rayl (K.). – *Users Guide to the Argonne SP Scheduling System.* – Rapport technique n Technical Memorandum No. ANL/MCS-TM-201, Mathematics and Computer Science Division, Argonne National Laboratory, USA, mai 1995. `http://info.mcs.anl.gov/Projects/sp/scheduler/scheduler.html`.

[LL90] Litzkow (M. J.) et Livny (M.). – Experience with the Condor distributed batch system. *In : Proceedings of the IEEE Workshop on Experimental Distributed Systems.*

[LLA+81] Lazowska (E. D.), Levy (H. M.), Almes (G. T.), Fisher (M. J.), Fowler (R. J.) et Vestal (S. C.). – The architecture of the Eden system. *In : Tech. Report 81-04-01, University of Washington.*

[LLM88] Litzkow (M. J.), Livny (M.) et Mutka (M. W.). – Condor - a hunter of idle workstations. *In : Proceedings of 8th International Conference on Distributed Computing Systems, San Jose, California.*

[LMSK63] Little (J. D.), Murty (K. G.), Sweeny (D. W.) et Karel (C.). – An algorithm for the traveling salesman problem. *Operations Research*, vol. 11, 1963, pp. 972–989.

[Lot92] Lottor (M.). – Internet growth (1981-1991). *Request for Comment 1296, Network Information Systems Center, SRI International*, janvier 1992.

[MD92] Mahanti (A.) et Daniels (C. J.). – IDPS : a massively parallel heuristic search algorithm. *Proceedings Sixth International Parallel Processing Symposium*, 1992, pp. 220–223.

[MDLT96] Melab (N.), Devesa (N.), Lecouffe (M.P.) et Toursel (B.). – Adaptive load balancing of irregular applications. A case study: IDA* applied to the

15-puzzle problem. *Springer-Verlag LNCS 1117, Proc. of the Third Intl. Workshop, IRREGULAR'96, Santa Barbara, California, USA*, 19–21 Aug 1996, pp. 327–338.

[Mel97] Melab (Nouredine). – *Gestion de la granularité et régulation de charge dans le modèle P^3 d'évaluation parallèle des langages fonctionnels.* – Thèse de PhD, Université de LilleI, 1997.

[MG95] Monteil (T.) et Garcia (M.). – Network-Analyser : un système de collecte et de prédiction de l'état d'un réseau local pour le placement dynamique de processus. *Journées de recherche avec actes sur le placement dynamique et la répartition de charge : Application aux systèmes répartis et parallèles*, mai 1995.

[MGPO89] Malek (M.), Guruswamy (M.), Pandya (M.) et Owens (H.). – Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, vol. 21, 1989, pp. 59–84.

[Mon96] Monteil (T.). – *Etude de nouvelles approches pour les communications, l'observation et le placement de tâches dans l'environnement de programmation parallèle LANDA.* – Thèse de PhD, Institut National Polytechnique de Toulouse - France, Nov 1996.

[MTP98] Melab (N.), Talbi (E-G.) et Petiton (S.). – Contôle adaptatif du degré de parallélisme : Application à l'algorithme de Gauss-Jordan par blocs. *In: Proceedings de JRPRC2, Deuxièmes Journées de Recherche sur le Placement Dynamique et la Répartition de Charge, Lille, France.*

[MvRT+90] Mullender (S.), van Rossum (G.), Tanenbaum (A.), van Renesse (R.) et van Staveren (H.). – Amoeba: A Distributed Operating System for the 1990's. *IEEE Computer*, mai 1990.

[Nam96] Namyst (R.). – *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* – Thèse de PhD, L.I.F.L., Université des Sciences et Technologies de Lille 59655 Villeneuve d'Ascq Cedex - France, Dec 1996.

[Nic87] Nichols (D. A.). – Using idle workstations in a shared computing environment. *ACM Operating System Review*, vol. 21, n5, novembre 1987, pp. 5–12.

[Nil80] Nilsson (N. J.). – *Priciples of Artificial Intelligence.* – Tioga Publishing Co., 1980.

[NM95] Namyst (R.) et Mehaut (J. F.). – PM^2: Parallel Multithreaded Machine, a computing environment for distributed architectures. *Parco'95 proceedings, Gent*, 1995.

[NR94] Neuman (B. C.) et Rao (S.). – The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, vol. 6, n4, juin 1994, pp. 339–355.

[OCD+88] Ousterhout (J. K.), Cherrenson (A. R.), Douglis (F.), Nelson (M. N.) et Welch (B. B.). – The Sprite network operating system. *IEEE Computer*, février 1988, pp. 23–36.

[Pea84] Pearl (J.). – *Heuristics.* – Addison-Wesley, 1984.

[PFK93] Powley (C.), Ferguson (C.) et Korf (R. E.). – Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, vol. 60, 1993, pp. 199–241.

[PK91] Powley (C.) et Korf (R. E.). – Single-agent parallel window search. *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, n5, 1991, pp. 466–477.

[PL95] Pruyne (J.) et Livny (M.). – Parallel processing on dynamic resources with CARMI. *In: Proceedings of the Workshop on Job Scheduling for Parallel Processing IPPS'95.*

[PM83] Powell (M. L.) et Miller (B. P.). – Process migration in Demos/mp. *In: Proceedings of the 9th ACM Symposium on Operating system Principles, New York*, pp. 110–119.

[PPTT90] Pike (R.), Presotto (D.), Thompson (K.) et Trickey (H.). – Plan 9 from Bell Labs. *In: Proceedings UKUUG.*

[PR96] Porto (C. S.) et Ribeiro (C.). – Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints. *Journal of heuristics*, vol. 1, n2, 1996, pp. 207–223.

[Pre94] Presnell (S. R.). – The Batch Reference Guide, 3rd Edition, Batch version 4.0. *Dept. Pharmaceutical Chemistry, University of California*, mars 1994. – `http://cornelius.ucsf.edu/~srp/batch/ucsf/batchusr_toc.html`.

[PRW94] Pardalos (P. M.), Rendl (F.) et Wolkowicz (H.). – The quadratic assignment problem: A survey and recent developments. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 16, 1994, pp. 1–42.

[QBA] Qbatch. – `http://gatekeeper.dec.com/pub/usenet/comp.sources.misc/volume25/QBAT`

[RK87] Rao (V. N.) et Kumar (V.). – Parallel depth first search. Part:I. Implementation. *International journal of parallel programming*, vol. 16, n 6, 1987, pp. 479–499.

[RPC88a] Remote Procedure Call Programming Guide. *Sun Microsystems, Inc.*, 1988.

[RPC88b] Remote Procedure Call Protocol Specification. *Request for Comment 1050, Sun Microsystems, Inc., DARPA*, juin 1988.

[RR81] Rashid (R. F.) et Robertson (G. G.). – Accent: A communication oriented network operating system kernel. *ACM Operating System Review*, vol. 15, n 5, 1981, pp. 64–75.

[RR95] Rego (C.) et Roucairol (C.). – A parallel tabu search algorithm using ejection chains for the vehicle routing problem. *Proceedings of the Metaheuristics International Conference, Breckenridge*, juillet 1995, pp. 253–295.

[SB89] Sen (A. K.) et Bagchi (A.). – Fast recursive formulations for best-first search that allow controlled use of memory. *Proceedings International Joint Conference on Artificial Intelligence*, Aug. 1989, pp. 297–302.

[Sch94] Schoinas (G.). – *Issues on the implementation of PRogramming SYstem for distriButed appLications: A free Linda implementation for Unix Networks.* – Rapport technique, Department of Computer Science, University of Crete, 1994.

[SG76] Sahni (S.) et Gonzales (T.). – P-complete approximation problems. *Journal of the ACM*, vol. 23, 1976, pp. 556–565.

[SH82] Shoch (J. F.) et Hupp (J. A.). – The worm programs - early experience with a distributed computation. *Communications of the ACM*, vol. 25, n 3, 1982, pp. 172–180.

[Smi88] Smith (J. M.). – A survey of process migration mechanisms. *ACM Operating System Review*, vol. 22, n 3, juillet 1988.

[Sof93] Software (Sterling). – CONNECT:Queue User's Guide. 1993. – `http://www.sterling.com/`.

[Sof95] Software (GENIAS). – CODINE: Computing in Distributed Networked Environments. septembre 1995. – `http://www.genias.de/genias/english/codine.html`.

[Ste95] Stellner (G.). – Consistent Checkpointing of PVM Applications. *Technical report, Institute fur Informatik der Technischen Universtat Munchen Lehrstuhl fur Rechnertechnik und Rechnerorganisation, D_80290 Munchen*, 1995.

[Sup94] Suplick (J.). – *An Analysis of Load Balancing Technology.* – Rapport technique, CXSOFT technical report, Richardson, Texas, janvier 1994. `http://lscftp.kng.ibm.com/pss/products/loadlev.html`.

[Tai91] Taillard (E.). – Robust tabu search for the quadratic assignment problem. *Parallel Computing*, vol. 17, 1991, pp. 443–455.

[Tai93] Taillard (E.). – Parallel iterative search methods for vehicle routing problem. *Networks*, vol. 23, 1993, pp. 661–673.

[Tai95] Taillard (E.). – Comparison of iterative searches for the quadratic assignment problem. *Location Science*, vol. 3, 1995, pp. 87–103.

[Tal95] Talbi (E. G.). – *Allocation dynamique de processus dans les systèmes distribués et parallèles : Etat de l'art.* – Rapport technique n TR-162, Université des Sciences et Technologies de Lille, janvier 1995.

[Tal98] Talbi (E. G.). – A Taxonomy of Hybrid Metaheuristics. *Submitted to Journal of Combinatorial Optimization*, mars 1998.

[TG97] Taillard (E. D.) et Gambardella (L. M.). – An Ant Approach for Structured Quadratic Assignment Problems. *In: 2nd Metaheuristics International Conference MIC'97.* – Sophia-Antipolis, France, juillet 1997.

[TL89] Theimer (M. M.) et Lantz (K. A.). – Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, vol. 15, n 11, novembre 1989, pp. 1444–1458.

[TLC85] Theimer (M. M.), Lantz (K. A.) et Cheriton (D. R.). – Preemptable remote execution facilities in the V system. *In: ACM SIGOBS 10th Symposium on Operating System Principles, New York*, pp. 2–12.

[Tur93] Turcotte (L. H.). – *A Survey of Software Environments for Exploiting Networked Computing.* – Rapport technique, Mississipi State University Report, juin 1993.

[UT94] Unison-Tymlabs. – Load Balancer: Automatic job queuing and load distribution over heterogeneous Unix networks. *Sunnyvale, CA*, 1994. – `http://www.unison.com/main-menu/products/auto_op.html`.

[VGA94] Vahdat (A.), Ghormley (D.) et Anderson (T.). – *Efficient, portable, and robust extension of operating system functionality.* – Rapport technique n CS-94-842, UC Berkeley, 1994.

[Vos92] Voss (S.). – *Tabu search: Applications and prospects.* – Rapport technique, Germany, Technische Hochshule Darmstadt, 1992.

[Wa83] Walker (B.) et al. – The LOCUS distributed operating system. *In : Proceedings 9th Symposium on Operating System Principles*, pp. 49–70.

[WHH+92] Waldspurger (C. A.), Hogg (T.), Hubermann (B. A.), Kephart (J. O.) et Stornetta (W. S.). – Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, vol. 18, n2, février 1992, pp. 103–117.

[ZGG90] Zhu (W.), Goscinski (A.) et Gerrity (G.). – *UTOPIA : A Load Sharing Facility for large, heterogeneous distributed computer systems.* – Rapport technique nTR-CS 90/9, University of New South Wales, Australia, mars 1990.