Gestion de ressources pour des services déportés sur des grappes d'ordinateurs avec qualité de service garantie

Patricia Stolf

Gestion de ressources pour des services déportés sur des grappes d'ordinateurs avec qualité de service garantie.

Patricia PASCAL

25 novembre 2004

Avant propos

Les travaux présentés dans ce manuscrit représente le travail de trois années de recherche effectuées au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS) dans le groupe Réseaux et Systèmes de Télécommunications (RST). J'exprime ma reconnaissance à Monsieur Malik GHALLAB, directeur du laboratoire pour m'avoir accueillie au LAAS et m'avoir donnée tous les moyens nécessaires à la réalisation de ces travaux.

Je remercie Messieurs Gérard Authié et Thierry Monteil pour avoir encadré mon travail. Leur confiance, leurs conseils, leur soutien et leur disponibilité m'ont permis de mener à bien la tâche qui m'a été confiée.

Je tiens à exprimer ma grande considération à Monsieur Monteil dont les compétences et les qualités humaines ont été très précieuses.

Je suis très reconnaissante envers Messieurs Bertil Folliot et Jean- Francois Méhaut d'avoir accepté d'être les rapporteurs de cette thèse. Je les remercie tout particulièrement.

Je remercie Messieurs Germain Garcia, Jean-Marie Garcia, Marcel Soberman de l'honneur qu'ils me font en participant au jury de thèse.

J'exprime toute ma reconnaissance à Philippe Bat, Bernard Miegemolle et Samuel Richard pour l'aide qu'ils m'ont apportée.

Je profite de cet avant propos pour saluer les membres du laboratoire avec lesquels j'ai travaillé, mes camarades de bureau, mes collègues enseignants.

Je n'oublierai pas de remercier le service documentation et le service reproduction pour leur disponibilité.

Merci à toutes les personnes que j'ai pu côtoyer durant ces années et avec lesquelles j'ai travaillé avec grand plaisir.

Enfin, mon dernier mot sera pour mes proches que je remercie tout particulièrement pour leur soutien.

Table des matières

Table des figures

Lexique

Liste des acronymes :
- *API :* Application Programming Interface.
- *AROMA :* scAlable Resources Manager and wAtcher.
- *ASP :* Application Service Provider.
- *CRU :* Cluster Resource Unit.
- *DQS :* Distributed Queueing System.
- *DRU :* Domain Resource Unit.
- *EDF :* Earliest Deadline First.
- *FCFS :* First Come First Serve.
- *FIFO :* First In First Out.
- *GRAM :* Globus Resource Allocation Managers.
- *GRID :* Globalisation des Ressources Informatiques et des Données.
- *GRU :* Grid Resource Unit.
- *HRU :* Host Resource Unit.
- *HTML :* HyperText Markup Language.
- *IDL :* Interface Descripion Langage.
- *IP :* Internet Protocol.
- *JAAS :* Java Authentication and Authorization Service.
- *JSSE :* Java Secure Socket Extension.
- *LEWF :* Least Expected Work First.
- *LSF :* Load Sharing Facility.
- *LWP :* LightWeight Process.
- *MDS :* Meta Directory Service.
- *MPI :* Message Passing Interface.
- *NQE :* Network Queuing Environment.
- *NQS :* Network Queueing System.
- *NWS :* Network Weather Service.
- *OGSA :* Open Grid Service Architecture.
- *PBS :* Portable Batch System.
- *PVM :* Parallel Virtual Machine.
- *QoS :* Quality of Service.
- *RED :* Robust Earline Deadline.
- *REMOS :* REsource MOnitoring System.
- *RPC :* Remote Procedure Call.
- *RSL :* Resource Specification Langage.
- *SNMP :* Simple Network Management Protocol.
- *SSL :* Secure Sockets Layer.

- *TCP :* Tansmission Control Protocol.
- *TLS :* Transport Layer Security.
- *UDP :* User Datagram Protocol.
- *XDR :* eXternal Data Representation.
- *XML :* eXtensible Markup Language.

Chapitre 1

INTRODUCTION

Le calcul haute performance voit apparaître depuis quelques années un nouveau support d'exécution : le cluster ou grappe de machines. L'intérêt de ce support réside entre autres dans son rapport qualité/prix, sa facilité de maintenance et d'évolutivité. De nombreux clusters se créent à travers le monde. Au début, ils étaient mono-bloc et constitués de dizaines de noeuds. La tendance actuellement est de faire des clusters et sous-clusters interconnectés par des réseaux rapides qui font globalement plusieurs milliers de noeuds et ne sont pas forcément sur le même site géographique ; c'est la grille nommée en anglais "GRID" (Globalisation des Ressources Informatiques et des Données).

Si la partie matérielle du cluster est relativement stabilisée, il n'en est rien pour la partie logicielle et plus particulièrement pour l'"intergiciel". L'utilisation par une personne non expérimentée reste délicate et nécessite souvent une bonne connaissance de l'architecture physique du ou des clusters, de leurs systèmes d'exploitation et des logiciels s'exécutant. Ceci rend à l'heure actuelle la grille difficilement exploitable.

L'objectif de cette thèse est de proposer un composant de l'"intergiciel" nécessaire en fournissant des modèles et outils pour partager et accéder aux ressources disponibles sur une grille de manière transparente. L'originalité réside entre autre dans l'insertion de la notion de qualité de service dans l'utilisation des ressources.

1.1 L'évolution des supports d'exécution

Le support des applications scientifiques évolue. Après les supercomputers, le "network computing" est apparu avec les grappes de machines de processeurs performants, des réseaux rapides (également appelés cluster). Actuellement, le "Grid Computing" constitue l'évolution des clusters. Le futur du calcul parallèle sera sûrement distribué et hétérogène. Le "Grid Computing" consiste à utiliser des ensembles distribués de plates-formes hétérogènes. C'est le nouveau support pour la nouvelle génération de calcul haute performance permettant le partage, l'agrégation et la sélection de ressources hétérogènes distribuées. Un "intergiciel" doit permettre de rendre l'ensemble des ressources transparent aux utilisateurs ; ils voient l'ensemble du système comme un système informatique unique [1].

L'hétérogénéité des ressources, les politiques d'administration, les priorités d'usage rendent

la gestion des ressources et le placement difficiles. Ian Foster et Carl Kesselman disent de la grille dans [2] : "We will probably see the spread of computer utilities, which, like present electric and telephone utilities, will service individual homes and office across the country. A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities."

Une idée à long terme pour le "Grid Computing" est de louer la puissance de calcul et de la capacité mémoire à travers l'Internet. En effet, les applications ont toujours besoin de plus de puisance de calcul et de capacité de mémoire. Il s'agit de faire de l'ASP : Application Service Provider.

Les caractéristiques principales du support d'exécution sont : support hétérogène (système d'exploitation, nombre de processeurs, mémoire, vitesse de calcul, bibliothèques) ; le réseau qui interconnecte les machines introduit de la latence et peut perdre des messages : les communications entre les machines seront un point important à gérer pour assurer une qualité de service. Dans un support d'exécution comme une grille, les réseaux reliant les différents sites peuvent être également hétérogènes ; les différents domaines de la grille sont gérés par différents administrateurs.

1.2 Les applications concernées

De nombreuses applications sont concernées dans différents domaines scientifiques. Les grilles sont utiles aux applications relevant de calculs intensifs : en physique nucléaire, bio-informatique, dans le domaine de la santé, en écologie, dans les sciences de l'univers [1]. Des applications de calcul haute performance ont été considérées (par exemple des applications du monde des télécommunications). Ces applications sont généralement parallèles. Les tâches ont soit une date de début identique soit des relations de précédence (une tâche ne démarre que lorsque la ou les précédentes ont terminé). On peut dire que l'implémentation est essentiellement un problème de synchronisation entre les tâches des applications parallèles. La difficulté sera à présenter plus tard.

1.3 Les problèmes à gérer

Les problèmes à gérer sont nombreux ; ils peuvent être classés en trois catégories : les problèmes liés au mode déporté de l'utilisation en mode ASP, les problèmes liés à l'aspect économique du fonctionnement des fournisseurs de service et enfin les problèmes liés à la gestion des ressources et à la qualité de service.

1.3.1 Le mode déporté : ASP

Le mode ASP consiste en l'exécution à distance d'applications via l'Internet. Il a plusieurs avantages : il n'est plus nécessaire de disposer "chez soi" des ressources matérielles (machines puissantes et/ou supercalculateur très coûteux) ni des ressouces logicielles qui peuvent correspondre à des besoins ponctuels ou une diversité de logiciels entraînant des coûts importants. Un tel mode de fonctionnement nécessite une authentification des utilisateurs, une sécurisation des échanges et un cryptage des données. La sécurité est un point très important et fait partie de la qualité de service implicite demandée par n'importe quel utilisateur mais également

par les administrateurs des différents domaines de la grille qui ne veulent pas voir des intrus pénètrer leur système par ce moyen-là. Toutefois des exécutions d'applications distantes sont considérées mais le code exécutable est déjà situé à distance, il n'y a pas de transfert de code. Il est nécessaire de gérer les comptes et les accès des utilisateurs de manière fiable : un client ne doit pas pouvoir contester une utilisation des ressources.

1.3.2 Aspect économique

L'aspect économique du fonctionnement des fournisseurs de service entraîne des difficultés. Tout client passe un contrat avec le fournisseur de services, ainsi il faut gérer les contrats passés avec les clients et vérifier leur respect. En fonction de son contrat, des ressources utilisées, le client doit recevoir une facturation adéquate. Le fournisseur de services doit aussi assurer une tolérance aux pannes pour garantir que les ressources disponibles sont utilisables. Le client peut aussi faire des choix économiques : le moins cher, le plus rapide quelque soit le prix ...

1.3.3 Gestion des ressources et qualité de service

Il s'agit de gérer le partage des ressources entre utilisateurs de manière efficace et optimale pour le fournisseur de services tout en garantissant le respect des contrats et de la qualité de service demandée. Une application constitue un ensemble de tâches dont l'ordonnancement n'est pas simple : les ressources sont distribuées, plusieurs ordonnanceurs doivent coopérer (surtout lorsque le support est étendu aux grilles de calcul) et il faut respecter une qualité de service prédéfinie.

La gestion des ressources peut se décomposer en quatre parties : l'observation, la représentation des données, la modélisation de l'accès aux données (les modèles doivent être suffisamment fins pour être les plus représentatifs possibles de la réalité mais doivent aussi être utilisables en temps réel : un compromis doit donc être trouvé) et enfin l'utilisation optimale des ressources (ce point est lié au placement des tâches et à l'expression des besoins). L'addition et la suppression de ressources doivent être transparentes pour l'utilisateur.

Après la modélisation de l'accès aux ressources, le placement constitue la partie la plus importante. Il s'agit de placer l'application ou les tâches de l'application parallèle de façon optimale. les outils de placement actuels ne gèrent pas de manière satisfaisante la qualité de service et aucun outil n'intègre la facturation.

L'algorithme de placement doit tenir compte :
- de la qualité de service demandée (classe de service de l'application) et la respecter
- de l'utilisation optimale des ressources (adéquation entre le besoin exprimé par l'utilisateur et les ressources disponibles).

L'algorithme de placement doit :
- permettre l'exécution la plus proche de l'optimal possible de l'application (durée d'exécution minimale)
- permettre un lancement immédiat (intéractif) ou en mode batch de l'application.
- pouvoir réserver des ressources à l'avance pour garantir la qualité de service (deadline ...).
- permettre l'arrêt d'une application moins prioritaire au profit d'une application très prioritaire. Cela implique la préemption. Deux possibilités sont ensuite envisageables : redémarrer l'application depuis le début ou utiliser le checkpointing c'est à dire le redémarrage du pro-

cessus à partir d'une sauvegarde intermédiaire (cela implique des contraintes au niveau de l'application : pas d'appels systèmes, pas de threads ...). Une solution peut être de choisir en fonction des caractéristiques de l'application.
- gérer un calendrier des lancements. En cas d'insertion d'applications dans le calendrier, il faut revoir le placement et l'ordonnancement prévu pour les applications non encore lancées. Le calendrier des lancements peut être modifié pour les applications non prioritaires ; toutefois si aucune contrainte n'a été donnée, l'application pourrait toujours être retardée, pour éviter cet inconvénient on peut fixer un nombre maximal de modifications au bout desquelles tout report de leur date d'exécution est interdite.
- en fonction de l'avancement de l'exécution, remettre en cause un placement : arrêt, recherche de meilleur placement, reprise d'exécution ailleurs. Ceci impose le mécanisme de checkpointing au niveau système. Ceci ne doit être fait que si des ressources se sont libérées et permettent d'assurer une meilleure exécution (plus rapide).
- gérer le partage des ressources ou les ressources dédiées pour garantir la QoS ("Quality of Service") et la définition des contraintes.
L'expression "garantie de la qualité de service" représente le respect des contraintes posées par le client par exemple : date de terminaison de l'application, exécution immédiate, exécution en étant seul sur la machine. - en cas de panne d'une machine sur laquelle une application a été placée (ou a été prévue), il faut calculer un nouveau placement. Si l'application a démarré, il y a deux possibilités : reprendre l'exécution depuis le début ou depuis le dernier checkpoint.
- l'algorithme de placement doit être rapide pour être utilisable en temps réel.

1.4 Plan de la thèse

L'objectif de la thèse est de s'intéresser au problème de la gestion des ressources pour des services déportés sur des grappes d'ordinateurs avec qualité de service garantie. Elle vise l'étude et la mise en place d'applications parallèles [1] sur des grappes d'ordinateurs en assurant une gestion fine des ressources afin de garantir une qualité de service prédéfinie. Elle aborde la modélisation du support d'exécution (processeurs, réseaux, mémoire, bibliothèques) et le placement. Le mode de fonctionnement ASP (Application Service Provider) est intégré. Le fournisseur de services doit pouvoir gérer son parc de machines : politique d'attribution des ressources, priorités des demandes, facturation, garantie de la qualité de service en fonction du contrat passé avec le client.

Le premier chapitre consiste à présenter les différents gestionnaires de ressources qui existent. Des outils d'observation de ressources sont exposés : Remos, NetLogger, NWS et Network Analyser. Puis des algorithmes de placement sont expliqués : placement statique, dynamique, avec réservation de ressources, le gang-scheduling, le backfilling et le placement avec deadline. Trois gestionnaires de ressources sont mentionnés : Legion, Globus, Sun Grid Engine. Des outils de type ASP : Citrix Metaframe, Microsoft .NET, Ninf et Netsolve sont présentés. Enfin le gestionnaire de ressources AROMA (scAlable Resources Manager and wAtcher) dans lequel les algorithmes de placement proposés dans cette thèse sont implémentés, sera présenté.

Le deuxième chapitre est consacré à la modélisation de l'accès aux ressources processeurs ;

[1] les applications séquentielles sont aussi considérées quoique l'intérêt d'utiliser un cluster ou une grille de calcul soit alors limité.

pour cela trois systèmes d'exploitation sont étudiés Linux, Solaris, Irix ; un modèle commun et simplifié est proposé.

Deux contextes sont envisageables pour le placement d'application : soit considérer une charge des machines stochastique (les caractéristiques des processus s'exécutant sur la machine sont inconnues) soit considérer une charge des machines déterministe (tous les processus qui s'exécutent sur la machine ont été placés par l'ordonnanceur donc sont connus pour le placement). La durée des tâches est supposée connue. Le chapitre quatre explique et valide le modèle d'exécution pour le placement avec charge stochastique.

Le chapitre cinq concerne le placement avec qualité de service et avec charge déterministe. Seul le temps de calcul demandé par les tâches est considéré, les communications entre les tâches ne sont pas prises en compte.

Le chapitre six prend en compte les communications entre les tâches d'une application parallèle. La modélisation de l'accès au réseau est expliquée : deux réseaux sont étudiés Ethernet et Myrinet. Le placement sur une grille de calcul avec qualité de service est présenté et validé.

Enfin, le dernier chapitre introduit la notion de modèle économique et propose un schéma simple de facturation. Les modèles économiques du marché réel sont présentés, l'objectif est d'adapter un modèle pour mettre en place la facturation dans AROMA.

Chapitre 2

LES GESTIONNAIRES DE RESSOURCES

2.1 Introduction

Les ressources dans un environnement distribué sont partagées. De multiples accès entraînent une variation de charge et de disponibilité. De nombreuses applications requièrent des garanties et une qualité de service. La gestion efficace des ressources nécessite leur observation, la prédiction de leur performance et un placement adéquat des applications sur celles-ci. Tout d'abord des outils d'observation et de prédiction de performances de ressources seront présentés ; dans une deuxième partie les algorithmes de placement seront étudiés puis dans une troisième partie, un état de l'art sur les outils de placement et de gestion de ressources sera établi avant de présenter des outils du type ASP et AROMA.

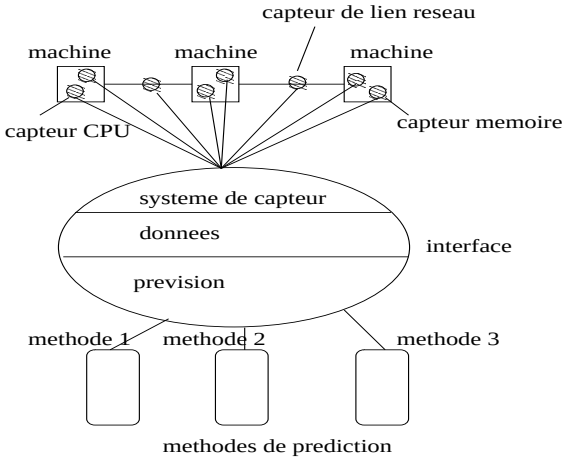
2.2 Les outils d'observation et de prédiction

Des outils d'observation de ressources sont nécessaires en vue d'effectuer un placement efficace des applications. Certains outils permettent également de prédire la performance des ressources : latence et bande passante réseau, le pourcentage de processeur disponible. Dans cette partie quatre outils sont présentés : Remos, NetLogger, NWS et Network Analyser.

2.2.1 Remos : REsource MOnitoring System :

Remos [24][25] permet à des applications réseaux d'obtenir des informations sur leur environnement d'exécution. L'objectif est de définir une interface uniforme masquant l'hétérogénéité des réseaux, la diversité du trafic, la variabilité de l'information et le partage de ressources sur le réseau. Des "Collectors" récoltent l'information à partir des routeurs. Un "Collector" est écrit en Java et il est prévu dans le plus long terme de le télécharger sur le réseau. Un "Modeler" est une librairie qui est liée à l'application et qui répond aux requêtes à partir des informations collectées par les "Collectors". Il a été conçu pour fonctionner sur un réseau IP dont les composants supportent SNMP (Simple Network Management Protocol). Remos supporte des requêtes concernant la topologie statique du réseau. La bande passante et la latence peuvent être connues entre deux noeuds du réseau.

FIG. 2.1: La structure du Network Weather Service



2.2.2 NetLogger : Networked Application Logger :

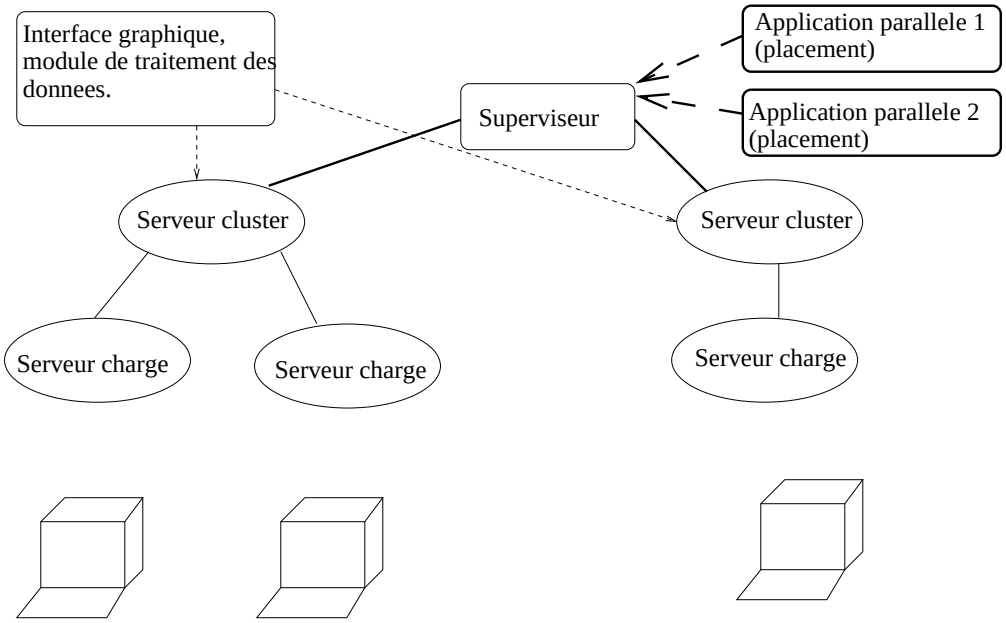
NetLogger [24] permet d'obtenir une analyse détaillée de bout en bout des applications distribuées. Il inclut des outils pour les applications, les machines et les réseaux. Il permet de détecter les goulots d'étranglement et de connaître la performance. Cet outil n'est pas adapté à l'analyse de programmes massivement parallèles (programmes MPI) et ne gère que les événements qui ont une durée supérieure à 0.5ms. Il utilise un démon "netlogd" pour collecter les messages sur une machine centrale, ce qui permet d'éviter d'avoir plusieurs fichiers log à différents endroits. Il utilise les outils : netstat, vmstat, iostat et ping qui sont utilisés dans des programmes Perl qui construisent des messages NetLogger contenant les résultats.

2.2.3 NWS : Network Weather Service :

NWS est un outil d'observation fournissant la prédiction de performance de ressources dynamiques dans des environnements distribués. NWS prédit la performance réseau (latence et bande passante)[26][27], le pourcentage de CPU disponible sur chaque machine qu'il contrôle [28] et la performance de la mémoire. NWS effectue des mesures périodiques sur la performance délivrable des ressources, utilise l'historique des mesures et des techniques statistiques pour la prédiction. Il communique ensuite les résultats de la prédiction aux ordonnanceurs. Il y a trois catégories de méthodes de prédiction : méthodes basées sur la moyenne, méthodes basées sur la médiane et les méthodes autorégressives. NWS choisit la meilleure prédiction pour une ressource en comparant les erreurs de prédiction avec les mesures. Trois modules existent dans NWS : "sensory subsystem" (système de capteur) qui collecte les informations sur la performance des ressources, "forecasting subsystem" (prévision) qui prédit la performance et donne l'information à un "reporting subsystem" (données). La figure 2.1 présente la structure de NWS.

Un serveur NWS doit s'exécuter sur chaque machine qu'il supervise. Chaque serveur a un capteur de performance réseau et un capteur CPU. Tous les serveurs connaissent les machines supervisées et le numéro de port TCP auquel chaque serveur est relié.

FIG. 2.2: La structure hiérarchique du Network-Analyser .



2.2.4 Network Analyser :

Introduction :

Le système distribué Network-Analyser développé au LAAS-CNRS [29] permet la mesure de charge et de surveiller un réseau de machines. Network-Analyser possède un ensemble d'utilitaires graphiques pour analyser cette charge. Ce système est donc très proche d'un outil d'administration réseau. A partir des statistiques réalisées sur tous les éléments du réseau, Network-Analyser identifie des comportements spécifiques qui permettent la prédiction de charge de machines monoprocesseur et de réseau simple de type "bus". Ce système dispose aussi d'un module de calcul pour placer efficacement des tâches d'applications parallèles. Network-Analyser est structuré suivant un système hiérarchique à trois niveaux (figure 2.2).

Les machines sont regroupées en clusters contrôlés par un serveur de cluster. Les clusters sont gérés par un superviseur qui possède une vue condensée de l'état de toutes les machines. Bien qu'il y ait des serveurs avec différents rôles (serveur de charge, serveur de cluster, superviseur), une particularité intéressante de ce système est d'avoir un unique processus par machine pouvant assumer un ou plusieurs rôles.

Network-Analyser a ainsi été pensé pour observer un parc de stations pouvant dépasser la centaine afin d'exploiter de manière efficace ces ressources pour le calcul parallèle (placement de tâches, équilibrage de charge, etc). Son rôle est donc déterminant. Or la probabilité d'une panne de machine sur un réseau augmente très fortement avec la taille du réseau. Tout d'abord, les machines en panne doivent être détectées rapidement. Ensuite, si cette panne se produit sur une machine jouant un rôle clé dans le système (superviseur ou serveur de cluster) il est indispensable de continuer d'assurer le service. Un mécanisme de tolérance aux fautes permet d'assurer un service permanent.

Les services offerts par ce système :

Les principaux services sont les suivants :

- Collecte d'information sur l'état de la machine : pour accéder à toutes les informations rapidement, les bibliothèques spécifiques à chaque machine sont utilisées (ouverture du fichier miroir du noyau UNIX et parcours de son espace de données). Un processus spécifique sur chaque machine collecte régulièrement les données : le serveur de charge. Une périodicité de 6 secondes a été choisie. Ce temps donne des informations suffisamment «rafraîchies» (interface graphique) et crée une occupation du processeur inférieure à 0.5%.

- Persistance d'une partie des données pendant 24 heures : ceci est intéressant pour observer à posteriori le comportement des machines pendant le déroulement d'une application parallèle. Pour des raisons de tolérance aux fautes, l'information doit être sauvegardée sur disque. La sauvegarde est locale et l'accès à l'information se fait machine par machine. Ce service est rendu par duplication du serveur de charge.

- La visualisation graphique de l'ensemble des informations collectées : pour réaliser cette fonction, il faut collecter les informations pour les machines visualisées. Le réseau est découpé en "clusters" de machines chacun étant géré par un serveur de cluster. Un client interroge alors le serveur de cluster pour obtenir des informations sur des stations incluses dans ce cluster. Le serveur de cluster gère la mise à jour de ses données. Ce type de fonctionnement est plus intéressant lorsque beaucoup de clients observent le même parc de machines. Il offre en outre une excellente flexibilité en créant des groupes de machines.

- Un placement des processus d'un utilisateur : trois contraintes ont été imposées. Le système doit répondre rapidement, avoir une vue condensée de l'état du parc de machines et tenir compte des placements déjà effectués. Ces caractéristiques amènent simplement à un système avec un point central de décision qui reçoit périodiquement des informations de toutes les machines. Ces informations d'état sont envoyées par chaque machine à leur serveur de cluster qui les transmet au point central appelé superviseur. Le système propose alors un placement en fonction d'une prédiction de l'état des machines.

- Une bibliothèque en C rendant le système transparent : l'API du Network-Analyser est utilisable par tous types d'outils (LANDA [30], PVM [31], MPI [32], interface graphique, utilisateur quelconque, etc). Elle permet d'accéder à toutes les informations du système.

La tolérance aux fautes dans le Network-Analyser :

Pour connaître l'état d'une machine distante, uniquement les communications avec elle sont disponibles (envoi d'un message et attente de réponse). Pour ne pas surcharger l'activité du réseau, les communications déjà présentes dans le système Network-Analyser sont utilisées : les envois périodiques d'informations condensées au niveau du superviseur. Ce protocole permet à un serveur de détecter la mort des agents du niveau inférieur et d'une partie du niveau qui lui est supérieur. Seul le superviseur possède une vue d'ensemble et tient à jour les données de l'ensemble du système (servant au placement de processus). Afin de pouvoir répondre rapidement à la mort d'un noeud important du système (serveur de cluster, superviseur), un nouveau rôle a été créé : les remplaçants ou «replica». Ces derniers font des communications sécurisées périodiquement avec le processus qu'ils sont susceptibles de remplacer. Quand ils détectent sa mort, ils votent pour en élire un nouveau. L'élection se fait grâce à un anneau virtuel en faisant circuler un jeton de vote. Les remplaçants étant classés, c'est le premier de la liste encore vivant et ayant une majorité de voix qui est promu. Chaque niveau du système

possède des informations spécifiques. Elles sont redondantes au niveau des remplaçants pour assurer leur persistance en cas de panne du remplacé. Il existe un nombre de remplaçants compris entre une borne inférieure et supérieure pour chaque noeud important. Si le nombre de remplaçants passe en dessous de la borne inférieure, de nouveaux remplaçants sont nommés par le superviseur.

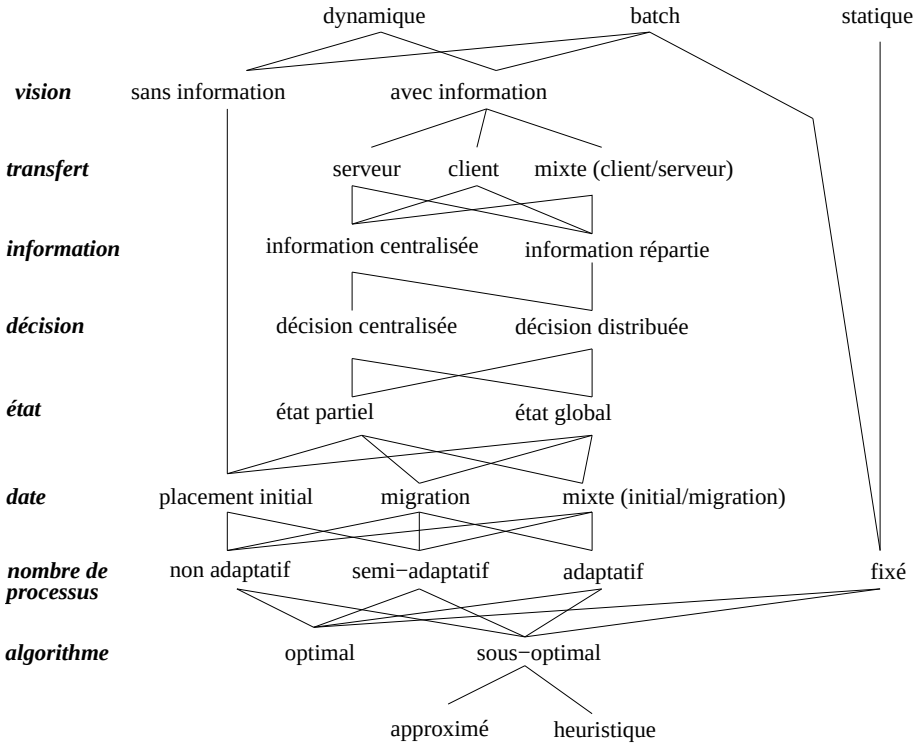
2.3 Algorithmes de placement :

Baumgartner et Wah ont effectué une étude des algorithmes d'ordonnacement dans [33]. Un tel problème se présente avec cinq composants : les événements, l'environnement, les besoins, l'ordonnanceur et l'ordonnancement. Les caractéristiques du travail (temps de calcul, contraintes de précédence), l'environnement machine (nombre de processeurs, interconnexion, puissance des processeurs) et les objectifs de performance sont les entrées de l'ordonnanceur et l'ordonnancement est la sortie.

Il y a plusieurs niveaux d'ordonnancement dans les systèmes distribués : "intracomputer scheduling" (ordonnancement local à l'ordinateur) et "intercomputer scheduling" (ordonnancement global à travers plusieurs ordinateurs, cela entraîne des communications). Pour chacun d'eux l'ordonnancement est une mise en adéquation entre les événements et les ressources. Si le contrôle est centralisé, l'algorithme sera plus simple mais il y a aura un goulot d'étranglement et un point de fragilité. Si la décision est distribuée, la surcharge causée par la distribution de l'information d'état peut atténuer les bénéfices. Une comparaison a été faite et a montré qu'aucune stratégie n'était tout le temps supérieure et que la surcharge liée à la communication était dans les deux cas présente.

Feitelson a étudié dans [34] le placement des applications parallèles. Chaque travail parallèle est exécuté dans une partition de processeurs. Cette partition peut être fixe (définie par l'administrateur système), variable (la taille de la partition est déterminée lors de la soumission basée sur la demande de l'utilisateur), adaptative (la taille de la partition est déterminée par l'ordonnanceur en fonction de la charge et de la demande de l'utilisateur), dynamique (la taille de la partition peut varier au cours de l'exécution en fonction des besoins et de la charge). Plusieurs niveaux de préemption peuvent être considérés : aucune préemption, préemption locale (la suite de l'exécution aura lieu sur les mêmes processeurs), préemption avec migration (la suite de l'exécution peut se faire sur un autre processeur), "gang scheduling" (tous les threads d'un processus sont suspendus et terminés simultanément ; il peut être implémenté avec ou sans migration). La préemption dans un système réel entraîne un coût supplémentaire et n'est pas facile à mettre en oeuvre : tous les threads doivent être arrêtés dans un état consistant et dans le cas de la préemption avec migration, il faut conserver les chemins de communication entre les threads. Il y est mentionné des résultats d'expérience de Gibbons : une politique d'ordonnancement du type LEWF ("Least Expected Work First") réduit le temps de réponse par rapport à une politique FIFO, si les temps de service sont inconnus ou seulement estimés l'ordonnancement de type "backfilling" (expliqué au paragraphe 2.3.4 qui consiste dans le cadre d'une politique FIFO à permettre à une application de s'exécuter avant son tour si cela ne retarde pas les applications avant elle) donne de meilleurs résultats. Les différentes possibilités pour le placement de processus et la migration sont décrites figure 2.3 [35] [38] [43]. Nous allons présenter plusieurs types de placement : statique, dynamique, batch, avec

FIG. 2.3: Classification de placement



réservation, le "gang-scheduling" et le "backfilling".

2.3.1 Placement statique, placement dynamique

Dans le cas du placement statique, la décision de placement est prise hors ligne (la décision est prise avant l'exécution sans prendre en compte l'état des ressources). Il est bien adapté aux applications déterministes (dont le comportement est totalement maîtrisé). Il ne prend pas en compte l'état des ressources (charges des processeurs, mémoire ...) [36], [30].

Dans le cas du placement dynamique, la décision tient compte des informations dont il dispose sur les ressources. L'algorithme peut éventuellement s'adapter au cours de l'éxécution des processus à la charge des processeurs ; c'est un placement en-ligne. On parle alors de migration [37]. Dans le cas de réseau de machines, plusieurs modes d'échange de l'information sont possibles [38] : échange explicite sur demande d'informations, échange périodique, échange relatif (lors d'une variation de charge relative), échange implicite profitant des autres communications pour diffuser les informations.

Plusieurs politiques de placement de processus existent dans la littérature : placement cyclique, placement aléatoire. Une politique de placement dynamique est généralement basée sur la minimisation ou la maximisation d'une fonction. Cette dernière découle principalement de modèles à base de files d'attente. Le choix du placement peut se faire en cherchant à minimiser : le nombre de clients dans les files [39], le temps de réponse du système [40] [41], le temps pour finir tous les processus déjà placés [40] ou le temps libre des processeurs [42]. Les algorithmes multi-critères permettent de faire le placement de processus en tenant compte de plusieurs directives. Une priorité est mise sur chaque critère. La sélection des machines

s'effectue en essayant de satisfaire au mieux les critères.

Enfin, un algorithme avec prédiction de charge (PPC) [44] peut être utilisé, il est basé sur une prédiction du temps d'exécution des processus d'une application [45]. Par exemple, le critère d'optimisation choisi peut être de minimiser la durée de l'application. Les processus sont placés sur chaque machine en minimisant la date de terminaison de l'application parallèle à N tâches. Un algorithme du type "Longuest Processing TIme" permet de placer d'abord les processus les plus longs en minimisant au fur et à mesure la date de terminaison du dernier processus placé. Pour le mettre en oeuvre, pour chaque tâche, sur chaque machine, il faut calculer sa date de terminaison avec l'algorithme de prédiction. Les informations sur les machines sont nécessaires [1]. Les données sur l'application doivent être fournies lors de la demande de placement. Dans le cas où ceci n'est pas possible, par exemple pour des applications non déterministes, il sera nécessaire de faire une approximation de la durée des processus. Cet algorithme est plus complexe à mettre en oeuvre que les algorithmes classiques de la littérature. Ceci est surtout dû à la prise en compte non seulement de l'état instantané des stations mais aussi de la prédiction de leurs états. C'est sur ce type d'algorithme que cette thèse s'est appuyée.

2.3.2 Le placement avec réservation de ressources :

Dans [56], les auteurs évaluent la performance d'algorithmes supportant la réservation. Ils examinent deux cas : les applications ne peuvent pas être arrêtées et redémarrées ultérieurement (cela impose alors de considérer un temps maximum comme temps d'exécution ; le backfilling est utilisable) ; deuxième cas : les applications peuvent être redémarrées : les prédictions sur le temps d'exécution sont utilisables et une équation sert pour calculer le coût d'arrêt des applications et permet de choisir l'application à arrêter ; dans ce dernier cas les performances d'ordonnancement sont augmentées.

La réservation permet à l'utilisateur de demander des ressources à plusieurs systèmes à une date donnée et d'avoir accès à ces ressources simultanément pour son application. La réservation à l'avance est ajoutée à PBS et à l'ordonnanceur MAUI (présentés paragraphe 2.4). Pour évaluer la performance, trois critères sont calculés : l'utilisation de la machine par les applications, le temps moyen pendant lequel les applications attendent de recevoir les ressources et le temps moyen entre le moment où l'utilisateur veut réserver les ressources et le moment où il obtient les réservations.
La plupart des systèmes d'ordonnancement supposent que les applications ne peuvent être “restartable” (sauf Condor).

Deux hypothèses sont faites : une application qui a réservé ne peut être arrêtée pour démarrer une autre. Quand l'ordonnanceur a accepté une réservation, l'application commencera à la date prévue.
Les algorithmes d'ordonnancement étudiés sont : FCFS et LWF (les applications sont classées en fonction du temps demandé : nombre de noeuds utilisés * temps d'exécution). Le backfilling est aussi considéré.

[1] Eventuellement, des informations sur les liens réseau peuvent être utiles pour prédire les temps de communication.

Modèle de réservation : une demande de réservation contient le nombre de noeuds demandés, le temps utilisé par les noeuds, la date de départ et l'application. Si l'ordonnanceur peut faire la réservation, il la fait sinon il donne à l'utilisateur une liste de dates à partir desquelles l'application pourrait démarrer, l'utilisateur choisit. Pour éviter des conflits, des simulations avec des "timelines" (date à partir de laquelle l'ordonnanceur envisage d'utiliser les noeuds du système dans le futur) sont utilisées.

Le système Silver [57] est un metascheduler basé sur la réservation. Il a été conçu pour équilibrer la charge à travers plusieurs sites pouvant appartenir à des domaines administratifs différents.

2.3.3 Le gang-scheduling :

Une décision de placement a un impact imprévisible sur le futur : à l'instant où la décision est prise, les arrivées de processus dans le futur sont inconnues, la décision de placement est optimale lorsqu'elle est prise mais peut ne plus l'être lorsque d'autres processus arriveront. Cela peut conduire à une perte de la puissance de calcul.
Le "gang scheduling" [58] est présenté comme étant une solution permettant d'avoir un meilleur temps de réponse et améliore l'utilisation. Les ordonnanceurs de la plupart des systèmes commerciaux utilisent le “variable partitioning”. L'utilisateur spécifie le nombre de processeurs demandés lors de la soumission. L'ordonnanceur fournit ensuite la partition de la bonne taille et la dédie au processus pour la durée de son exécution. Si le nombre de processeurs demandés n'est pas disponible, le processus est rejeté ou remis en file d'attente.

Il y a alors plusieurs solutions : permettre aux petits travaux de passer en tête de file pour être exécutés immédiatement. Comme cela peut provoquer une situation de famine des travaux longs, c'est souvent combiné à la possibilité pour les processus longs de réserver des processeurs pour le futur. Tout ceci nécessite de connaître le temps d'exécution des travaux.

Une autre solution au problème de la fragmentation des ressources est d'utiliser l' “adaptative partitioning” plutôt que le “variable partitioning” : le nombre de processeurs utilisés est une négociation entre le nombre demandé et le nombre disponible.

La solution préférée est le "gang scheduling" : les travaux disposent du nombre de processeurs demandés mais uniquement pour un quantum de temps. Ainsi tous les processus peuvent s'exécuter de manière concurrente en utilisant le "time slicing". La décision d'ordonnancement n'a alors d'impact que sur un pas ; cela augmente la flexibilité et la performance. Cela ne nécessite pas d'information sur le temps d'exécution. Plusieurs chercheurs ont craint la surcharge due au changement à la fin du quantum mais au contraire le "gang scheduling" augmente la performance. Un obstacle au "gang scheduling" : tous les processus doivent être en mémoire ce qui diminue l'espace mémoire disponible. Ils peuvent être "swappés" sur le disque lorsqu'ils sont préemptés.

2.3.4 Le Backfilling :

Le "backfilling" [59] est un moyen simple d'améliorer l'utilisation des ordonnanceurs “space-sharing”. Les approches FCFS sont moins efficaces car les processus longs peuvent fragmenter les ressources disponibles (exemple avec un processus qui a besoin de cinq processeurs alors que

seulement trois sont disponibles, il retarde tout le monde). Le "backfilling" résout ce problème en permettant aux travaux de passer en tête de file s'ils n'interfèrent pas avec les autres.

FCFS est très utilisée mais n'est pas une politique idéale. La fragmentation entraîne l'oisivité de processeurs et dégrade l'utilisation. Il existe des solutions pour optimiser la performance : "gang-scheduling" et "dynamic partitioning". L'approche la plus simple qui augmente l'efficacité est le "backfilling".
Il y a deux techniques de "backfilling" : "conservative backfilling" (permet aux processus de passer en tête de file s'ils ne retardent AUCUN travail arrivé avant) et "easy backfilling" (permet aux travaux de passer en tête de file s'ils ne retardent pas LE premier processus de la file). Le backfilling est un algorithme utilisé par l'ordonnanceur "EASY" [60], [61] sur IBM SP2. Si un processus n'a pas fini à la date prévue, il est tué pour éviter de retarder les autres. A la fin d'un travail, l'algorithme regarde si le travail en tête de file peut être démarré. Si le nombre de processeurs disponibles est insuffisant, tous les travaux en cours d'exécution sont triés en fonction de leur date de terminaison et l'algorithme détermine le "shadow time" temps à attendre avant qu'un nombre suffisant de processeurs soit disponible pour démarrer le premier processus de la file. Les processeurs en excès par rapport au nombre de processeurs demandés par le premier processus à la date correspondant à ce "shadow time" sont appelés "extra nodes". L'algorithme essaie d'exploiter les processeurs libres avant le "shadow time" pour exécuter un processus qui se terminera avant le "shadow time" et les "extra nodes" pour exécuter un processus qui peut se terminer après le "shadow time". L'algorithme se répète jusqu'à ce qu'aucun processus ne puisse exploiter ces processeurs. Aucun de ces travaux ne retarde le premier processus de la file mais cela ne veut pas dire qu'aucun processus ne sera retardé par le backfilling et des travaux peuvent même être retardés plusieurs fois avant d'arriver en tête de file. L'algorithme favorise l'utilisation du système.
Les processus courts bénéficient davantage que les processus longs du backfilling. Deux techniques sont présentées pour améliorer les ordonnanceurs utilisant le backfilling : estimation du temps d'exécution en agrandissant les périodes temporelles pour le backfilling et tri des processus par longueur (l'absence de famine n'est pas garantie).

2.3.5 Le placement avec date de terminaison :

Un placement avec "deadline" doit garantir que l'exécution des tâches sera terminée avant leur "deadline". Pour cela l'ordonnanceur connaît le temps CPU demandé par les tâches ainsi que la date de terminaison demandée. Lors de la demande de placement, l'ordonnanceur effectue un "contrôle d'admission" au cours duquel il vérifie la faisabiblité du placement. Si toutes les tâches ne consomment pas plus que le temps CPU qu'elles avaient demandé, le contrôle d'admission suffit. Si une tâche dépasse son temps processeur, elle peut mettre en péril les "deadlines" des autres tâches. Dans ce cas, une politique peut être mise en place et par exemple arrêter la tâche qui dure trop longtemps.

Le premier algorithme dynamique temps réel a été appelé Earliest Deadline First (EDF) [62]. Liu et Layland ont prouvé que si un ordonnancement respectant toutes les "deadlines" existe alors le placement qui exécutera la tâche ayant la "deadline" la plus proche en premier garantira toutes les "deadlines". Toutefois, EDF se dégrade lorsque le système devient chargé. Cet algorithme est optimal pour des tâches indépendantes, qui supportent la préemption et avec un seul processeur. Il n'est plus optimal dans une situation avec multi-processeurs. Il n'y a aucune garantie.

Un algorithme Robust Earliest Deadline Scheduling (RED) a été proposé par Buttazzo et Stankovic [63]. Lors de la soumission d'une tâche, il conclut si la tâche peut être acceptée. Si ce n'est pas le cas, elle sera réétudiée lors de la terminaison des autres tâches car il se peut que le temps processeur des tâches soit sur-estimé. Il fonctionne avec des tâches préemptives, avec des tolérances de "deadline".

Une étude a été faite par Chan, Lam et To [64] pour éliminer la migration dans un algorithme de placement "online" sur des multi-processeurs. Le problème est le suivant : il y a m processeurs identiques, un ensemble de travaux caractérisés par une date de soumission, une "deadline" et une durée. L'objectif est de les exécuter avant leur "deadline". Kalyanasumdaram et Pruhs ont montré dans [65] que si un algorithme hors-ligne sur m processeurs avec migration est disponible, il est possible de construire un algorithme hors-ligne sans migration avec 6m-5 processeurs. La migration semble être nécessaire pour les algorithmes en ligne. Chan, Lam et To ont montré que l'ensemble des processus pouvant être exécutés par un algorithme hors-ligne avec migration peut être exécuté par un algorithme en ligne sans migration sur m speed-5.828 processeurs (c'est à dire des processeurs qui sont 5.828 fois plus rapides : qui traitent 5.828 unités de travail par unité de temps). Ces résultats ont été intégrés dans l'algorithme appelé PARK. Il est basé sur le fait qu'un processeur n'est pas engagé avec un processus tant que le processeur n'est pas dégagé de tout engagement avant sa deadline. Un processeur est dit "engagé" si un processus a été admis pour s'exécuter avec lui avant sa "deadline". Lors de la soumission du processus, s'il ne peut être admis sur un processeur il est mis dans un pool. Ensuite, chaque processus du pool est étudié : un processeur n'ayant pas d'engagement avant sa deadline est cherché. Si un processeur est trouvé, le processus est mis dans sa file. Chaque processeur traite ensuite les processus de sa file avec l'algorithme EDF.

Henri Casanova dans [66] étudie le placement avec "deadline" dans une grille de calcul. Son objectif est de minimiser les dépassements de "deadline" ainsi que leur amplitude.

Rajkumar Buyya dans [67] propose un algorithme basé sur les "deadlines" et budget contraint par un critère de temps et de coût. Cet algorithme est appelé DBC ("Deadline and budget constrained").

Ces deux études considèrent les contraintes dues à des "deadlines" mais ne considèrent pas un mélange de processus avec différents types de besoins au niveau de la qualité de service (deadline, début immédiat, ressources dédiées) comme il est fait dans cette thèse.

2.3.6 Placement batch :

La soumission de travaux en mode batch permet aux utilisateurs de ne pas attendre la fin du processus et permet au système de l'ordonnancer avec flexibilité en tenant compte des priorités et des ressources disponibles. La difficulté est de partager des ressources hétérogènes et distribuées (leur utilisation est plus difficile que sur une seule machine). Les machines sont différentes et peuvent aussi être utilisées pour des processus interactifs qui ne doivent pas être perturbés. Les ressources contrôlées par un système batch sont généralement possédées par une même organisation dans le même domaine administratif. Les administrateurs ont le contrôle de toutes les ressources et ont la responsabilité des politiques d'ordonnancement. Codine [68],

[69], DQS [70] et LSF [71] mettent l'accent sur les environnements hétérogènes. Loadleveler [72], NQE et PBS orientent surtout leurs efforts sur les supercalculateurs. Le "scheduling batch" doit résoudre les conflits entre demandes d'utilisateurs.

2.4 Etat de l'art sur les outils de placement et de gestion de ressources :

L'objectif de cette section est de présenter brièvement quelques outils de placement et les principaux gestionnaires de clusters ou grilles existants, leurs caractéristiques et leurs carences.

Trois systèmes de placement en mode "batch" seront présentés : le Maui Scheduler, LS-Batch et PBS.

Maui Scheduler :

Maui Scheduler Molokini Edition [46] [47] est un logiciel de "batch scheduling" écrit en Java avec des composants modulaires. Il est donc extensible.
Il y a différentes parties dans Maui :

- **la partie client/serveur :** le maui serveur écoute un socket pour les demandes des utilisateurs (XML-based socket protocol).
- **scheduler subsytem :** ce composant est responsable de la prioritisation des travaux parallèles ou série batch. Il est également responsable de leur soumission, de la gestion des files et du backfilling des processus.
- **resource manager subsystem :** ces composants (node daemons) sont responsables de la distribution des tâches, de l'observation de l'état des noeuds et des travaux.

Parmi les caractéristiques il y a : support MPICH-MPI sur Myrinet, support MPI sur Ethernet, possibilité de réserver des ressources, cryptage et authentification pour les communications par socket (utilisation de la librairie Java Cryptix), protocole client/serveur XML, possibilité d'utiliser UDP ou multicast (TCP est recommandé pour les petits clusters $<$100 noeuds).
Il y a deux sortes de réservations : "job reservation" (réservation après l'ordonnacement à partir d'une certaine date et pendant un certain temps) et "sys reservation" (réservation sur les ressources système par un administrateur).

Le module d'ordonnancement par défaut a la liste des travaux à qui l'ordonnaceur a donné une priorité et essaie de mettre en place une réservation immédiate (dans l'ordre des priorités) ; lorsqu'il n'est pas possible de faire une réservation immédiate, le processus de "backfilling" est démarré (tentative de faire une réservation immédiate ou future. Si le processus est l'objet d'une réservation future il reste dans la file des travaux en attente, s'il peut être exécuté immédiatement il est supprimé de la file d'attente et est mis dans la file des processus en exécution).
Le module "resource manager" par défaut gère les communications avec les "node daemons" : démarre, arrête les processus, exécute les tâches à distance, récupère les mises à jour de l'état. Toutes les mises à jour de l'état des node daemons sont envoyées via UDP (peut être configuré pour utiliser multicast ou TCP). Un module "JobChecker" observe les travaux qui ont terminé ou ceux qui ont été annulés et les enlève du système. Un "node daemon" (tourne sur chaque

noeud) est responsable du statut et de la gestion des travaux et tâches sur une machine.
Il est possible d'écrire un ordonnanceur personnel et des "node daemons" modules.

LSBatch :

LSBATCH [48] est un système distribué batch pour des systèmes hétérogènes et à grande échelle. LSBATCH est implémenté au dessus de UTOPIA ("network operating system"). La complexité est masquée à l'utilisateur.

Les demandes d'un processus ainsi que la file dans laquelle il doit être inséré sont spécifiées dans la ligne de commande. Contrairement à des outils tels que NQS, DQS ou Condor qui ont des files associées à des machines ou des ensembles de machines dans LSBATCH, les files sont différenciées par services à rendre à différents types de travaux. "Job queue" est un mécanisme pouvant être configuré. Chaque file de batch a des attributs : priorité, (les travaux d'une file de forte priorité sont considérés avant les autres), le "nice", "runWindows" (intervalle de temps pendant lequel les processus de la file peuvent être exécutés ; en dehors de cet intervalle ils seront suspendus), "cpuLimit" (temps cpu maximum que peut utiliser un processus s'il le dépasse, il sera tué), "users" (liste d'utilisateurs pouvant soumettre des travaux à cette file), "hosts" (liste de machines sur lesquelles les travaux de la file peuvent s'exécuter) et "exclusive". En plus de ces attributs, une file a aussi deux variables décrivant l'état de la file ("enabled" : accepte des demandes et "active" exécute des processus).

Les machines sont groupées en cluster. Dans chaque cluster l'architecture est centralisée autour d'un serveur batch maître qui après avoir obtenu des informations sur la charge et sur les ressources trouve la ou les machines nécessaires et envoie le travail au serveur de batch esclave de la machine concernée. Le maître conserve l'état des processus en cours.

UTOPIA fournit des informations sur la charge, utilisation CPU, espace mémoire disponible, I/O, nombre d'utilisateurs connectés.
Après une requête, le maître conserve le processus dans une file en attendant que la contrainte de date soit résolue (l'utilisateur pouvant avoir spécifié une date de début pour son processus) et de trouver une machine satisfaisant les contraintes. Un mécanisme d'élection de nouveau maître est mis en place en cas de panne.

Il y a deux niveaux d'ordonnancement : au niveau du maître qui répartit les processus et au niveau des esclaves qui choisissent les processus à suspendre et à reprendre (pour cela les esclaves utilisent un "vector scheduling" contenant les informations fournies par le maître : priorité, consommation de ressources, run Windows).
Le partage des ressources respecte plusieurs contraintes : de temps, d'accès de l'utilisateur, d'accès à la machine et d'utilisation de ressources.

Pour la soumission de travaux parallèles, il faut donner le nombre de processeurs à utiliser. Contrairement à DQS il n'est pas nécessaire de mettre une option dans la ligne de commande. L'ordonnancement d'un travail parallèle se fait comme pour un travail séquentiel : le maître fournit la liste des machines utilisées à l'esclave ; si une des tâches dépasse ses ressources l'application entière est suspendue et un signal est envoyé à l'application.
LSBATCH utilise deux composants du système UTOPIA : LIM et LSLIB. LIM permet de localiser le maître, de décider sur quelle machine l'exécuter et donne des informations sur la

charge des ressources.

LSBATCH et UTOPIA sont portables sur une variété de plate-formes UNIX : SunOS, HP-UX, IRIX, ConvexOS, AIX, Solaris, Ultrix et DEC OSF.

PBS (Portable Batch System) :

PBS [49], [50], [51] est un système d'ordonnancement "batch" et un outil de gestion de charge développé par Veridian Systems pour la NASA. Il gére l'authentification des utilisateurs et permet également de soumettre des travaux de manière interactive. Les utilisateurs peuvent préciser la priorité de leurs processus. Des bibliothèques de programmation parallèles comme MPI, MPL, PVM et HPF peuvent également être utilisées. L'utilisateur peut avoir la garantie que le processus aura accès à toutes les ressources demandées pour s'exécuter. Si l'ordonnanceur par défaut ne convient pas il est possible d'en implémenter un en remplacement. La sécurité est contrôlée (mécanisme d'authentification) et il est possible de restreindre les accès des utilisateurs. La tolérance aux fautes est variable : elle peut conduire à la suppression des processus. PBS ne permet pas de spécifier une date de terminaison pour un processus et ne permet pas de facturation du client en fonction de l'utilisation du système.

C'est un produit configurable [52] permettant l'exécution de processus interactifs ou batch sous AIX, CMost, Digital Unix, HP-UX, Irix, Linux, OSF/1, Solaris, SunOS et Unicos. L'espace utilisateur est supposé uniforme. Aucun support pour la migration de processus n'est fourni et les mécanismes de checkpointing au niveau utilisateur ne sont pas possibles ; les mécanismes de checkpointing au niveau noyau ne sont possibles que sous Irix et Unicos.
PBS fournit trois ordonnanceurs configurables. Plusieurs politiques sont possibles : par files ou globales FIFO, round-robin, priorités ...

PBSWeb [53] [54] a été créé pour simplifier l'utilisation de PBS : l'utilisateur peut indiquer ses instructions et ses options dans des formulaires HTML ce qui simplifie la création des scripts. Tout est fait de manière sûre grâce à l'utilisation de SSL. PBSWeb pourrait après modifications fonctionner avec d'autres ordonnanceurs batch que PBS. Deux technologies sont utilisées pour fournir une interface Web à PBS : JavaScript et PHP. Il permet aussi de contrôler les soumissions de processus ce qui fait qu'il se rapproche du meta ou du grid computing.

Le scheduler Maui peut être intégré au système PBS [55].

NQS (Network Queueing System) :

NQS [13] est un système ordonnanceur “batch” développé à la NASA pour apporter un support à l'utilisation des supercalculateurs CRAY. La version commerciale permet l'équilibrage de charge et un plus grand contrôle des ressources. La version Generic NQS [68] permet de lier des processus à des processeurs, de réserver des processeurs dédiés et de réordonner les files dynamiquement pour éviter la domination d'un utilisateur ou d'un groupe.

L'efficacité d'un système batch dépend du “mélange” entre les processus qui sont dans la file de manière à ce que l'utilisation des ressources soit optimale.

Sur les systèmes Cray multiprocesseurs, NQS optimise l'utilisation des ressources processeurs et mémoire. Chaque processus soumis à NQS précise ses limites en temps CPU et en mémoire. Les processus sont classés en fonction de leur limite ; chaque classe ayant une file

d'attente séparée.

L'ordonnanceur parcourt les files dans l'ordre de priorité et dans chaque file choisit les processus qui sont éligibles jusqu'à atteindre la "run queue limit". On en distingue deux : un global à la machine et un par file. Il s'agit du nombre maximum de processus pouvant être en exécution en même temps.

Trois gestionnaires de ressources sont maintenant présentés : Legion, Globus et Sun Grid Engine ainsi que le projet e-TOILE.

2.4.1 Legion

Caractéristiques :

Legion [75] [76] [77] [78] est un environnement de méta-computing orienté objet qui se positionne au dessus du système d'exploitation existant. C'est un système flexible qui s'adapte à la politique de placement de chaque machine. Legion répond à dix objectifs : autonomie des sites (chaque machine contrôle ses ressources), extensibilité du coeur du système (tous les composants de Legion sont extensibles et remplaçables), extensibilité de l'architecture, facilité d'utilisation (la complexité est masquée à l'utilisateur), utilisation du parallélisme pour la haute performance, stockage des données dans un annuaire unique, prise en compte de la sécurité, gestion de l'hétérogénéité des ressources, support de plusieurs langages pour les applications, tolérance aux fautes.

Le système Legion est constitué de plusieurs objets hiérarchiques : un objet représentant les capacités de la machine et utilise la réservation (Host object), un objet représentant le stockage d'informations (Vault object) sauvegardant l'état persistant d'un objet Legion.

Architecture :

Le modèle de gestion de ressources contient les ressources (Hosts et Vaults), la base de données d'informations statiques (Collection), l'ordonnanceur (Enactor) et un objet réalisant l'exécution (execution Monitor). Dans l'implémentation actuelle, il n'y a pas d'objet séparé réalisant le monitoring : l'Enactor et le Scheduler le réalisent. Trois groupes de fonctions sont possibles pour les objets de type Host : la gestion de réservation (Unix n'a pas de notion de réservation donc un objet standard Unix de type host maintient une table de réservation dans le Host objet), gestion des objets et diffusion de l'information.

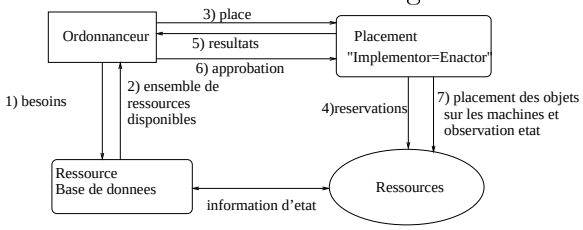
Placement :

Le placement est une négociation entre des agents agissant au nom des applications et des ressources. Le placement proposé est un placement aléatoire où les dépendances entre objets ne sont pas considérées.

Interaction avec un client :

Quand un client demande un placement pour une application, différentes étapes sont suivies. Le "Scheduler" interroge le "Collection" pour obtenir de l'information sur les ressources, envoie l'ordonnancement à l'"Enactor" qui négocie l'instanciation avec les objets ressources.

FIG. 2.4: Modèle de Legion



L'"Enactor" fait les réservations, renvoie les résultats au "Scheduler" et attend son approbation. Ensuite, il gère le placement sur les machines et supervise le statut des travaux. L'"Enactor" peut réaliser une co-allocation. L'"Execution Monitor" peut demander de recalculer un ordonnancement. Le modèle de Legion et les étapes suivant une requête d'un client sont décrits sur la figure 2.4.

2.4.2 Globus

Caractéristiques :

L'objectif de Globus [79] [80] [81] est de regrouper un ensemble de composants pré-existants et de les faire interagir, ces composants forment le toolkit Globus. Le gestionnaire de ressources de Globus répond à cinq objectifs : autonomie des sites, hétérogénéité des ressources, extensibilité, co-allocation et contrôle en ligne de l'état du support.

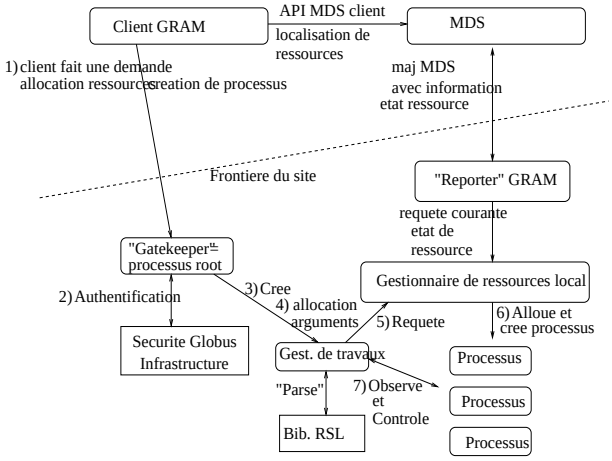
Architecture :

Le toolkit Globus comprend entre autres les composants suivants :

- "resource broker" : redirige une demande de ressource soit vers un "resource manager" (cas où une seule machine est nécessaire), soit vers un "co-allocator" (plusieurs sites sont nécessaires)
- service information : "Meta Directory Service" (MDS). Annuaire stockant des informations sur les différentes machines, leurs caractéristiques et leur état. Informe sur le gestionnaire de ressources associé.
- "co-allocator" : alloue les ressources sur plusieurs sites
- "resource manager" : gère les ressources d'une machine

Le "Globus Resource Allocation Managers" GRAM crée les travaux satisfaisant la demande de ressources si possible, il les gère et met à jour périodiquement la "MDS information service". Les composants de GRAM sont : la bibliothèque client GRAM, le "gatekeeper" (processus s'exécutant en tant que root et existant sur les machines distantes avant toute requête), la bibliothèque RSL (Le Resource Specification Language fournit une méthode pour échanger des informations sur les ressources entre les composants), le "job manager" et le "GRAM reporter" (il est responsable du stockage d'information sur la structure de l'ordonnanceur et son état dans la MDS). Les composants de GRAM sont décrits sur la figure 2.5.

FIG. 2.5: Composants de GRAM



Placement :

Le placement est réalisé par les ordonnanceurs locaux. Globus propose une interface standard vers plusieurs ordonnanceurs : Condor, LSF, NQE, Fork, EASY and LoadLeveler.

Interaction avec un client :

GRAM propose une API pour soumettre, suspendre un travail et contrôler le statut. Cela permet aux utilisateurs d'exécuter les travaux à distance. L'exécution commence quand une application utilisateur GRAM sur une machine locale envoie une requête (en RSL) à un ordinateur distant. Le "gatekeeper" gère la requête : il autentifie le client, démarre un "job manager" sur la machine locale et lui fournit les arguments de l'allocation. Le "job manager" démarre le processus sur le système local et soumet la requête d'allocation de ressources au système de gestion de ressources sous-jacent. Après la création des processus, le "job manager" est responsable du monitoring de l'état des travaux et gère les communications avec le client.

GARA : Globus Architecture for Reservation and Allocation [82] [83] étend l'architecture de gestion de ressources de Globus. Il introduit un objet ressource et la réservation (immédiate ou à l'avance), il offre une qualité de service sur différents types de ressources : réseaux, processeurs et disques. Avec les réservations, le "co-scheduling" est simplifié. La réservation et la création de l'objet sont réalisées par GRAM : Globus Reservation and Allocation Manager.

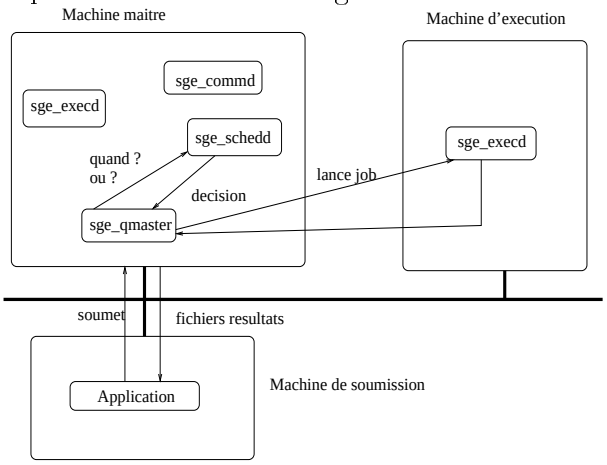
La dernière version de Globus (version 3) introduit les services Web en se conformant à la technologie OGSA (Open Grid Service Architecture).

2.4.3 Sun Grid Engine

Caractéristiques :

Sun Grid Engine [84] est un système de gestion de ressources de clusters en mode batch ou interactif, qui permet la répartition de la charge dans un environnement distribué hétérogène. Ses principaux composants sont SGE MASTER et SGE SCHEDULER. Les travaux sont soumis au SGE MASTER qui interroge le SGE SCHEDULER ; SGE SCHEDULER renvoie

FIG. 2.6: Composants de Sun Grid Engine et interaction avec un client



sa décision de placement au SGE MASTER, qui lance l'application et contrôle son exécution (par l'intermédiaire de EXEC DAEMON). Les deux composants SGE MASTER et SGE SCHEDULER sont implémentés par des daemons (respectivement sge-qmaster et sge-schedd). Ces daemons s'exécutent sur une machine particulière (Master host) qui est le point central de l'activité du cluster. Le daemon EXEC DAEMON sge-execd s'exécute sur les machines d'exécution du cluster (Execution hosts). Enfin, un daemon de communication sge-commd est utilisé pour permettre la communication entre les différents composants de Sun Grid Engine.

Placement :

Sun Grid Engine permet le placement dynamique (l'état des machines est observé avant d'effectuer un placement mais également durant l'exécution). Plusieurs politiques de placement sont proposées, par exemple il est possible de spécifier une date de terminaison (si le traitement n'est pas terminé avant la date demandée, sa priorité est augmentée afin de terminer le plus tôt possible ; il est juste garanti que tout est fait pour que le processus finisse avant la date demandée).

Interaction avec le client :

L'utilisateur spécifie les besoins en ressources de l'application (mémoire, système d'exploitation, processeur...). L'observation de charge peut être faite régulièrement avec des paramètres de charge par défaut mais il est aussi possible de définir de nouveaux paramètres (un script shell est exécuté et augmente la charge). Les utilisateurs connectés sur les "Submit host" peuvent soumettre des travaux et contrôler leur statut. La figure 2.6 décrit les composants de Sun Grid Engine et explique l'interaction avec un client.

2.4.4 Synthèse

Les trois gestionnaires décrits précédemment possèdent des fonctionnalités communes :

- une base de données regroupant les informations sur les ressources
- une architecture de communication entre les différents composants

- une gestion de la tolérance aux fautes
- le contrôle des utilisateurs (identification et autorisation).

Legion mise sur l'évolutivité et la réutilisabilité, mais le placement doit être géré par les utilisateurs. De plus, en pratique ce système s'avère très gourmand en ressources.

Globus utilise des composants pré-existants, les fait interagir et étend leurs fonctionnalités pour pouvoir faire face à l'hétérogénéité. L'ajout de composants à Globus peut rapidement devenir complexe.

Legion tout comme Globus ne garantit pas de qualité de service. La facturation n'est pas prise en compte et l'interface avec les utilisateurs est primaire (shell et API de programmation).

Sun Grid Engine est un système complet, c'est un système centralisé et les informations observées sur chaque machine peuvent être étendues. Il est plus convivial que les deux produits précédents (IHM, qualité de service), mais la gestion de la qualité de service est assez limitée.

2.4.5 e-TOILE

Le projet RNTL e-TOILE a pour objectif d'interconnecter des ressources de calcul de différents laboratoires et d'utiliser cette plate-forme expérimentale pour explorer la technologie des grilles. Des applications réelles sont utilisées [85].

Globus est utilisé pour interfacer les composants d'e-TOILE. Tous les sites e-Toile sont raccordés à VTHD à travers des interfaces Gigabit Ethernet. Pour observer les ressources de la grille, ils ont adpaté un outil existant Map Center [87] (c'est un outil développé au cours du projet européen DATAGRID non intrusif et transparent) et Ganglia développé à l'Université de Berkeley [86] pour vérifier le bon équilibrage de la charge des noeuds.

2.5 Outils du type ASP :

2.5.1 Citrix Metaframe :

Il est important de permettre aux utilisateurs distants d'avoir des accès aux données et applications. Il y a un grand choix de matériel : PCs, téléphones cellulaires ... Les utilisateurs accèdent de plus en plus aux informations à distance (depuis une conférence par exemple). L'architecture client/serveur traditionnelle est coûteuse et compliquée à administrer. Cela limite l'ajout de nouveaux utilisateurs, la performance d'applications, la sécurité de l'information. Il y a différentes possibilités pour que les entreprises fournissent des applications aux clients :

- architecture client/serveur traditionnelle : exécution locale au client (application de type Windows)
- architecture distribuée réseau : téléchargement et exécution sur le client (application de type Java)
- architecture basée sur serveur : exécution à 100% sur le serveur (application Windows, Java ou Unix).

Certaines fois, il est nécessaire d'ajouter un serveur (middle tier) entre l'application serveur et le client : c'est un serveur dédié entre des serveurs Windows 2000/NT 4.0 ou Unix et des clients Windows ou Java. Ce serveur traduit les protocoles Windows ou Unix en un protocole permettant de fournir l'accès à l'application au client. Citrix Metaframe [88] permet

de centraliser des applications Windows, Unix et Java sur des serveurs et d'y accéder depuis n'importe quel support à travers n'importe quelle connexion (un utilisateur peut par exemple accéder à partir de n'importe quel matériel par des pages Web à des applications Windows, Java, Unix). Lorsqu'un ajout d'un serveur middle tier est nécessaire, les systèmes Citrix installent le logiciel directement au dessus de l'application serveur. Ce logiciel est installé sur les serveurs Microsoft Windows 2000/NT 4.0, Sun Solaris, IBM AIX ou HP-UX. Le protocole ICA (Independent Computing Architecture) permet de connecter un client sans traduction. Citrix Metaframe permet :

- moins de complexité (pas de middle-tier)
- gain de performance (pas de traduction de protocoles)
- administration centralisée (pas de séparation serveur et serveur middle-tier)

Le produit Citrix Extranet est un logiciel virtuel privé qui permet de déployer de manière sûre des applications à des utilisateurs à travers Internet. Les données sont protégées grâce à plusieurs authentifications des utilisateurs.

La technologie Citrix NFuse permet de créer des portails Web pour des applications et de définir les applications spécifiques à un utilisateur, un groupe ... L'utilisateur les utilisera comme si l'application s'exécutait localement. Le produit Citrix Installation management services permet une installation automatisée d'applications pour les environnements Citrix Metaframe 1.8.
Le logiciel Citrix load balancing services permet d'automatiser l'équilibrage de charge sur un environnement multi-serveurs. L'utilisateur spécifie l'application qu'il veut faire tourner. Citrix Metaframe détermine quels serveurs sont capables d'exécuter l'application et quel serveur est le moins chargé (déterminé grâce à un facteur de charge : CPU utilisée, nombre d'utilisateurs ...) et route la demande de l'utilisateur vers ce serveur (de façon transparente pour l'utilisateur). Aucun mécanisme de tolérance aux fautes n'est fourni. Le produit Citrix Resource Management Services est destiné aux environnements Citrix et Microsoft NT multi-utilisateurs.

La sécurité des données envoyées entre client et serveur est assurée grâce au codage des informations et autorisations d'accès des utilisateurs.
Les différents produits de Citrix peuvent se classer de la manière suivante :

- Citrix NFuse : pour les portails Web
- Citrix Metaframe pour les applications serveurs
- Les services de management (Unix management, Load management, Resource management, Installation Management et Security).

2.5.2 Microsoft .NET :

.NET [89] permet à l'utilisateur d'accéder à ses données, à ses applications à partir de n'importe quel support (PC, téléphone, pager ...) et de n'importe quel endroit. Les données sont disponibles de manière sûre sur Internet. Les applications adapteront leurs fonctionnalités au support à partir duquel l'utilisateur y accède. Les applications sont configurées en fonction de paramètres propres à chaque utilisateur.
La plate-forme .NET est basée sur le standard XML et permet de développer de nouvelles applications. Toutes les interactions sont des messages SOAP (SOAP est un protocole léger

et simple basé sur XML utilisé pour échanger des informations structurées et typées à travers le Web. SOAP peut être utilisé avec de nombreux protocoles d'Internet existants ; il permet une grande variété d'applications). Cela permet aux employés, clients, données, applications commerciales de former un ensemble cohérent, intelligent et interactif. Quatre composants sont nécessaires pour former cette nouvelle génération de services :

- Microsoft Visual Studio.NET
- Microsoft .NET Entreprise Servers
- .NET Framework
- Microsoft Windows.NET

Le .NET XML Web services inclut différentes fonctionnalités notamment pour l'authentification des utilisateurs, paramètres utilisateurs ... Les services Web XML permettent aux applications de partager les données et d'appeler d'autres applications à partir de n'importe quelle plate-forme ou système d'exploitation. .NET My Services fait partie du concept "Microsoft .NET", propose différents services pour gérer et protéger les informations d'un utilisateur et y accéder de différents supports, applications. C'est l'utilisateur qui contrôle ses données qui décide avec qui les partager pendant combien de temps. Les mécanismes d'authentification sont basés sur Kerberos.
Les utilisateurs payeront les données reçues. Les développeurs payeront les outils, les prestataires de service payeront une licence à Microsoft pour leur permettre d'utiliser ".Net My services".
Microsoft ".NET My services" peut être accédé depuis Microsoft Windows, Macintosh, Palm PC, Pocket PC et de nombreux outils UNIX. .NET My Services sera facilement extensible. Dans la version Beta 3 de .NET servers, un service d'équilibrage de charges réseau pour cluster virtuel est fourni. Il y a également la possibilité de partager la charge selon le modèle round-robin entre plusieurs serveurs pour une application. Il est possible d'avoir une affinité entre processus et processeurs. Les communications sont gérées par le SIP : Session Initiation Protocole IETF RFC 2543.
Avec ASP.NET (ASP = Active Server Page) il est possible de définir un service Web.

2.5.3 Ninf :

Ninf [90] [91] [92] s'inscrit dans le Network Computing, il permet aux utilisateurs d'accéder aux ressources matérielles, logicielles, aux données scientifiques distribuées à partir d'une interface. Ninf permet d'exploiter la haute performance dans le "Network parallel computing" mais aussi des accès aux publications des chercheurs. Les ressources de calcul sont partagées par des bibliothèques distantes exécutables sur un serveur distant. Les utilisateurs peuvent construire une application en appelant les bibliothèques avec le RPC (Remote Procedure Call) de Ninf (interface de programmation similaire à des appels de fonctions d'autres langages de programmation).
Le "metaserver" de Ninf maintient des informations sur les ressources, alloue et ordonnance les applications pour avoir un bon équilibrage de charge. Il y a une interface WEB permettant de lister, trier ou d'utiliser (deux manières d'utiliser les bibliothèques sont proposées : par appel de fonction dans les progammes ou par le WEB) les bibliothèques enregistrées sur les serveurs. Ainsi, le réseau permet de partager des ressources de calcul (CPU, mémoire) en plus des données.

Le système Ninf utilise le modèle client-serveur. La machine serveur et le client peuvent être connectés à travers un réseau local ou via l'Internet, les machines peuvent être hétérogènes : les données sont transmises lors de communications dans un format de données commun.
Un "library provider" fournit les bibliothèques et les ressources de calcul au réseau. Il prépare les exécutables de Ninf en :

- écrivant les interfaces de description de chaque fonction de bibliothèque dans le Ninf IDL (Interface Description Language)
- exécutant le Ninf IDL compilateur qui fournit les fichiers d'entête et le code "stub"
- compilant la bibliothèque avec le bon compilateur (bon langage)
- faisant le lien avec les bibliothèques RPC de Ninf.
- enregistrant les bibliothèques sur le serveur Ninf s'exécutant sur la machine.

Certaines bibliothèques comme LAPACK (paquetages d'algèbre linéaire) sont déjà disponibles de cette manière. Dans l'IDL, il y a la définition de l'interface, la fonction à enregistrer, un nom de module pour spécifier le nom de la bibliothèque, son langage de programmation, les options pour lier et créer les exécutables, les alias de la fonction. La description de l'interface est compilée par le générateur d'interface de Ninf pour fournir un programme stub pour chaque fonction de la bibliothèque. Le générateur d'interface fournit automatiquement un makefile avec lequel les exécutables peuvent être créés en liant les programmes stub et les fonctions de librairie. Dans l'implémentation courante, les communications entre client et serveur se font avec une connexion TCP/IP pour assurer une communication sûre entre les processus. Dans un environnement hétérogène, Ninf utilise le format de données SUN XDR (eXternal Data Representation) comme protocole par défaut. Le client peut spécifier le serveur de différentes manières :

- dans le fichier .ninfrc
- dans la variable d'environnement
- avec une URL
- allocation automatique par le Ninf Metaserver.

La procédure est la suivante : le client interroge le serveur pour avoir l'information sur l'interface, en fonction de la description renvoyée par le serveur, le client envoie les arguments et récupère les résultats. Le Ninf "Metaserver" est un agent qui regroupe l'information sur les serveurs Ninf (adresse, numéro de port, liste de fonctions enregistrées sur le serveur, la distance avec le serveur en respectant la bande passante, la possibilité de calcul de la machine, le statut du serveur et la charge) et permet au client de choisir un serveur approprié. Le client peut interroger le "metaserver" et déléguer le problème. Passer par le "metaserver" permet d'assurer l'équilibrage de charge (le "metaserver" trouve le meilleur serveur en fonction des ressources et de la charge). Le "metaserver" est implémenté en Java (portabilité, facilité de programmation réseau, multi-threading). L'infrastructure de Ninf comporte plusieurs "metaservers" et serveurs. Les "metaservers" échangent des informations périodiquement.

Le travail futur de Ninf : authentification, comptabilité (facturation), exportation de code client (pour l'instant rien n'est offert), tolérance aux fautes (l'objectif est de récupérer les transactions lors de panne réseau) et sécurité (les objectifs sont de sécuriser les noeuds serveurs et de crypter les données).

2.5.4 NetSolve :

NetSolve [93] s'inscrit dans le Grid Computing, il a été installé sur de nombreux sites dans le monde. C'est un système client/serveur permettant aux utilisateurs de résoudre des problèmes complexes à distance. Les utilisateurs peuvent accéder aux ressources matérielles et logicielles à travers un réseau. Il y a de la tolérance aux fautes. Une politique d'équilibrage de charges est mise en oeuvre pour utiliser les ressources de manière performante. L'ajout d'un logiciel comme ressource est facile (sans aucune restriction sur le type de logiciel).

Le projet NetSolve [93] [94] est similaire à Ninf mais n'a pas de langage de description pour les informations de l'interface. Dans NetSolve, un processus agent permet d'avoir un équilibrage de charge comme le "metaserver"de Ninf.

Il y a différentes interfaces d'accès à NetSolve : Fortran, C, Matlab, Mathematica. Les interfaces Matlab, Mathematica et Java permettent de manipuler directement les objets ; pour les interfaces qui ne sont pas orientées objet, il est nécessaire d'utiliser une "calling sequence" pour décrire les objets. NetSolve est construit sur les protocoles Internet (TCP/IP). Il est utilisable sur UNIX, et des parties du système sont utilisables pour Microsoft Windows 95, 98, NT et 2000.
Le système est composé de trois parties : le client NetSolve, l'agent et le serveur. L'agent maintient une base de données de serveurs avec leurs capacités et leurs statistiques d'usage dynamique. Cette information est utilisée pour répondre aux demandes des clients. La politique de placement consiste à optimiser l'utilisation des ressources plutôt que la performance d'une application industrielle. La décision d'ordonnancement est basée sur des informations statiques et dynamiques maintenues sur les ressources. Le premier agent démarré est le master. Les serveurs doivent s'enregistrer auprès des agents. Il ne peut y avoir qu'un agent sur une machine à un instant donné. Il peut y avoir plusieurs serveurs sur une même machine mais seulement s'ils ont des agents différents. Les serveurs ont un fichier de configuration dans lequel est spécifié le nom de l'agent à contacter pour s'enregistrer, le nombre de requêtes provenant d'une machine ou d'un domaine. Le client demande à l'agent la liste de serveurs (il faut connaître l'adresse IP ou le nom de la machine sur laquelle il y a l'agent), le client contacte le serveur et lui envoie les paramètres, le serveur lance le service approprié. Le serveur renvoie les paramètres à l'utilisateur ou une erreur. Il est possible d'accéder à CONDOR, Globus, IBP, Legion (le client peut utiliser les graphes de Legion pour les dépendances).
NetSolve a intégré NWS (Network Weather Service) dans son agent, cela permet d'améliorer la performance. Il y a deux possibilités d'installation : installer uniquement le client ou installer le client, agent et serveur. Pour invoquer une fonction, il doit y avoir un "problem description file" (PDF) ; plusieurs PDF sont inclus dans NetSolve mais il est possible d'ajouter de nouvelles fonctionnalités aux serveurs en en écrivant d'autres. Il existe le concept du "farming" qui permet de gérer plusieurs demandes pour un problème NetSolve ; ceci n'est possible qu'à partir du C. Dans le but de réduire les communications inutiles mais assurer que toutes les comunications nécessaires seront transmises, un graphe de dépendance est créé et envoyé au serveur où il est ordonnancé.

La sécurité dans NetSolve est assurée avec Kerberos : les requêtes d'exécution d'un client à un serveur nécessitent une authentification. Il peut y avoir des serveurs n'utilisant pas Kerberos. Par défaut les serveurs ne sont pas dans le mode "kerberos", il faut ajouter une

option dans la ligne de commande. Il n'y a pas de cryptage des données échangées entre clients serveurs ou agents et il n'y a pas non plus de protection de l'intégrité du flot de données. Si le client a besoin de fournir au serveur du code, il ne peut y avoir de migration de code compilé car l'environnement est hétérogène ; dans ce cas, le code est dans un fichier transféré au serveur qui le compile et peut l'exécuter. Pour assurer la sécurité la commande UNIX *nm* est utilisée pour interdire tout appel système mais ce n'est pas robuste dans le cas d'une attaque par un hacker.

2.5.5 DIET :

DIET Distributed Interactive Engineering Toolbox a été développé dans le cadre des projets GASP du RNTL et ASP de l'ACI GRID. L'objectif est de fournir des outils pour des environnements de calculs sur la grille.

L'environnement permet à partir d'une requête d'un client de localiser un serveur approprié.
L'architecture générale de DIET comprend une hiérachie d'agents pour améliorer les performances lors de nombreuses requêtes de clients [73], [74]. DIET utilise les outils SLIM (il permet d'identifier les ressources capables de résoudre un problème) et FAST (il prédit les performances en utilisant NWS).

2.6 AROMA

Le logiciel AROMA (scAlable ResOurces Manager and wAtcher) fait partie des résultats de cette thèse. AROMA est un outil chargé de gérer l'exécution à distance avec une qualité de service définie [3], [4], [5]. Il comprend un gestionnaire de ressources, un lanceur d'applications, un ordonnanceur, un module statistique et un système de facturation. Ces travaux sont la suite de ceux effectués pour le système Network-Analyser [6]. AROMA permet de contrôler l'utilisation des ressources dans une grille. Deux entités sont définies : les clients et les fournisseurs de services (figure 2.7). Ce sont les fournisseurs de service qui possèdent les ressources (processeurs, disques, mémoire et applications serveurs). Le fournisseur de service et le client peuvent communiquer soit par l'Intranet soit par l'Internet.

Un contrat lie les deux entités : une entité de gestion gère les contrats. Les fournisseurs de services sont avertis de toutes les modifications de contrat ou de tout nouveau contrat. Le contrat définit le niveau de qualité de service auquel le client aura droit, les ressources utilisables, la priorité et la tolérance aux fautes.

2.6.1 Architecture

Principes de base

Le gestionnaire de ressources est structuré en quatre niveaux hiérarchiques : Grille, Domaine, Cluster et Machine. La grille est composée de domaines. Un domaine peut regrouper les machines d'un même réseau ou les machines d'un même bâtiment. Un domaine est composé de clusters. Chaque cluster étant un ensemble de machines reliées en réseau ou dans la même pièce par exemple. Le dernier niveau correspond aux machines individuelles composant le

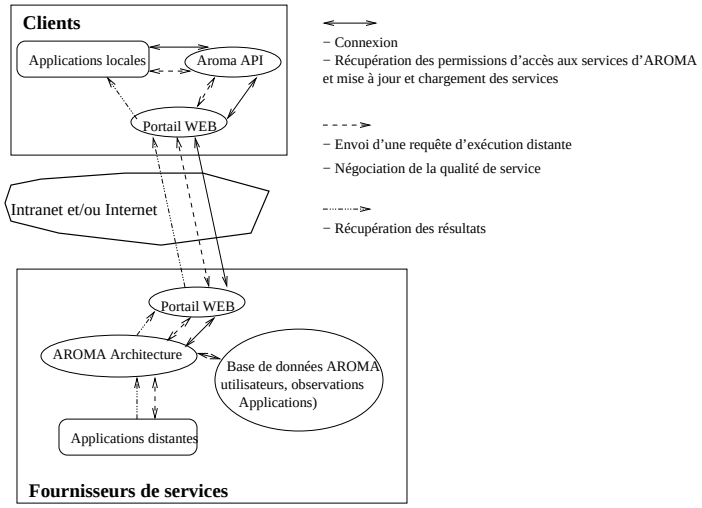


FIG. 2.7: Communication entre les fournisseurs de services et les clients

cluster. Chaque niveau est représenté par un serveur : GRU (Grid Resource Unit), DRU (Domain Resource Unit), CRU (Cluster Resource Unit) et HRU (Host Resource Unit). Chaque serveur est une instance d'un serveur générique (figure 2.8) et peut implémenter toutes les fonctionnalités ou uniquement un sous-ensemble.

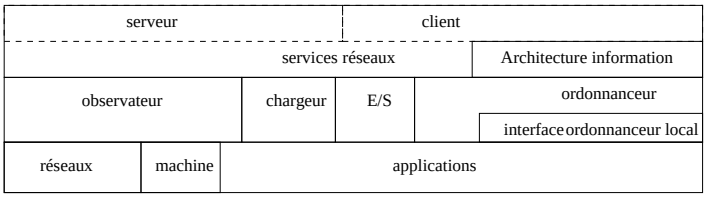


FIG. 2.8: Architecture du serveur générique

Le premier niveau collecte les informations d'états relatives aux machines, réseaux et applications. Ces données sont recueillies par l'observateur des HRUs. Elles sont ensuite envoyées périodiquement aux CRUs. Chaque niveau a une vue agrégée de l'état des niveaux inférieurs. Le chargeur s'occupe de l'exécution des applications : lancement et suivi des applications s'exécutant sur la machine. Le module E/S s'occupe des entrées/sorties des applications. L'ordonnanceur place les applications sur les noeuds appropriés en fonction de la charge de la machine et des contraintes données par l'utilisateur. Il utilise les informations collectées par l'observateur. L'ordonnanceur peut coopérer avec un ordonnanceur local ou externe. Les deux derniers niveaux (services réseaux, serveur et client) gèrent les communications entre les clients et les serveurs.

Un HRU est implanté sur toutes les machines de la grille.
Un CRU supervise un ensemble de HRUs. Les HRUs envoient périodiquement les informations d'état à leur CRU. Le CRU garde une trace de cette information et en calcule une moyenne pour obtenir sa charge. Une décision de placement peut être prise à ce niveau.
Un DRU supervise un ensemble de CRUs. Le DRU peut prendre une décision de placement

en coopération avec ses CRUs. Les CRUs envoient périodiquement l'information à leur DRU. Cette information est ensuite utilisée pour calculer la charge du DRU.
Les GRUs sont distribués sur la grille, ils représentent ses points d'accès. Un exemple d'architecture est donné figure 2.9. Les communications d'informations d'état sont représentées.

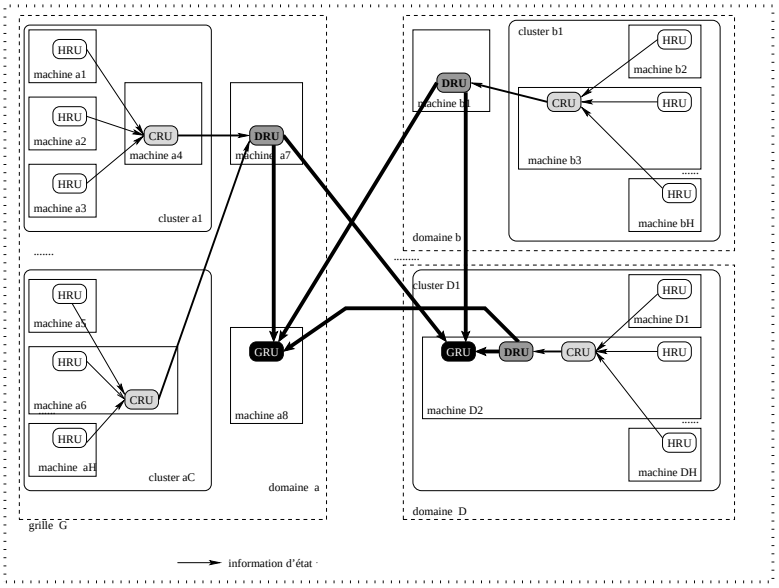


FIG. 2.9: Exemple d'architecture

Architecture de communication

Les communications entre les clients et les services sont assurées par les technologies offertes par Java et Jini [2] de SUN. La portabilité et la dynamicité du code sont assurées grâce à Java. La coopération des services au sein d'un réseau est assurée grâce à Jini. La figure 2.10 représente une communication entre un client et un service.

Le "lookup service" est le composant principal de Jini : c'est en quelque sorte une base de données des services disponibles sur le réseau à un instant donné. Cette base est dynamique : un nouveau service s'enregistre automatiquement auprès d'un "lookup service" puis il doit régulièrement l'informer de son existence grâce au mécanisme de "lease" (figure 2.10-1). Un service ne s'étant pas manifesté depuis un certain laps de temps sera automatiquement supprimé. Un client souhaitant accéder à un service interroge un "lookup service" (figure 2.10-2) pour obtenir la référence du service (figure 2.10-3). La multiplication des "lookups services" et des services permet d'assurer la tolérance aux pannes (des réseaux ou des machines). Le dialogue entre le client et le service correspond à l'appel de méthodes distantes de Java (figure 2.10-4). L'architecture logicielle du système avec les différents "lookups services" est illustrée figure 2.11.

[2] Java, et toutes les technologies basées sur Java, Jini et toutes les technologies basées sur Jini sont des marques déposées de Sun Microsystems, Inc. aux U.S. et dans les autres pays.

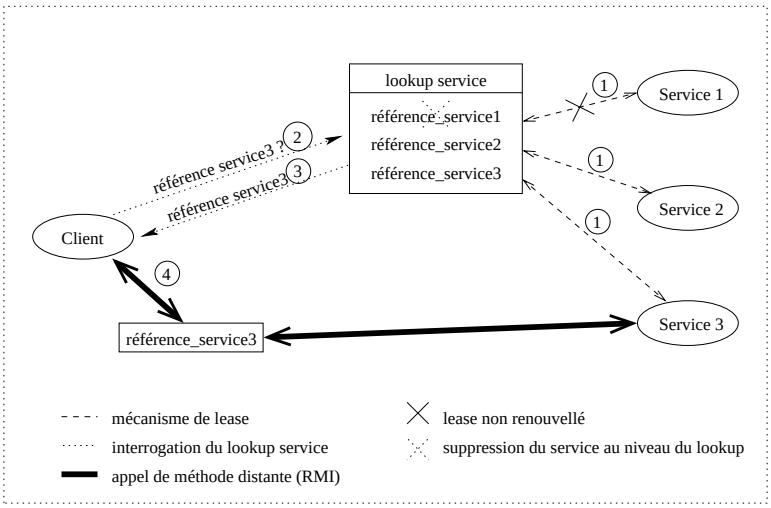


FIG. 2.10: Dialogue entre un client et un service

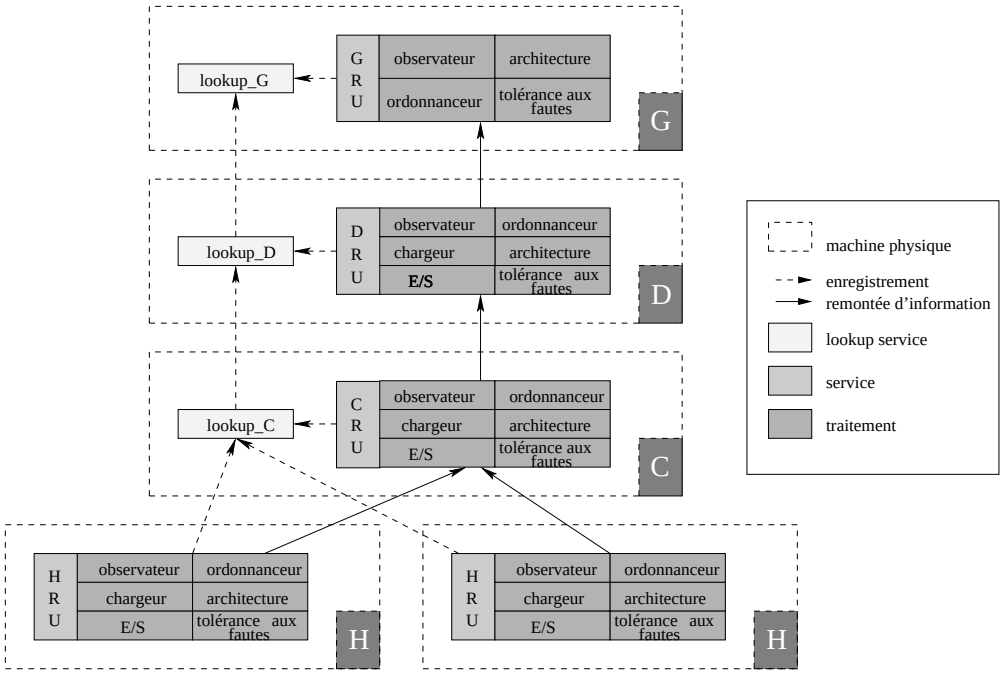


FIG. 2.11: Architecture du gestionnaire de ressources AROMA

2.6.2 Caractéristiques

Dynamicité et portabilité

La Grille est un environnement dynamique par sa distribution géographique et la multitude de machines. La notion de dynamicité se rencontre à trois niveaux :

– la topologie de la grille : le gestionnaire de ressources doit être attentif aux modifications de la configuration de la grille : ajout, suppression de noeuds volontaires ou involontaires.

- les services : de nouveaux services doivent pouvoir être ajoutés de manière transparente pour l'utilisateur. Les services doivent également se supprimer de manière dynamique.
- les permissions des utilisateurs : un client ne peut avoir accès qu'aux services pour lesquels il a souscrit un contrat. En cas de modification du contrat l'accès aux services doit être modifié dynamiquement.

Les noeuds de la grille ont différentes architectures et différents systèmes d'exploitation. Le code doit être portable.

Qualité de service et facturation

Une qualité de service est définie entre les clients et les fournisseurs de service en fonction du contrat. Cela inclut par exemple la quantité de ressources utilisées, la puissance des ressources, la priorité, la sécurité, la tolérance aux fautes. Le problème de placement sera présenté en détails dans les chapitres suivants.

AROMA contrôle l'exécution (consommation réelle des ressources, détection de la fin d'une application). Cette fonctionnalité est utilisée par le système de facturation : par exemple pour arrêter un processus qui a consommé trop de ressources. Quand l'application distante est terminée, les clients en sont informés et peuvent récupérer les résultats manuellement ou automatiquement.

Sécurité

Le gestionnaire AROMA doit répondre à plusieurs besoins de sécurité :

- authentification : lors des communications entre le client et les services, les deux parties doivent s'identifier pour vérifier les autorisations d'accès.
- confidentialité : toute communication doit être cryptée pour éviter qu'un intrus n'accède aux informations échangées. Ainsi même si un message est intercepté il ne peut être déchiffré.
- non répudiation : l'utilisation d'un service par un client ne doit pas pouvoir être contestée.

La sécurité est assurée grâce à Java Secure Socket Extension (JSSE) [7] et Java Authentication and Authorization Service (JAAS) [8]. L'utilisation de certificats électroniques pour toutes les connexions permet d'authentifier les clients et les services.
Le protocole Transport Layer Security (TLS) assure le chiffrement de la communication entre le client et le service et garantit la confidentialité. Ce protocole combine l'utilisation d'algorithmes de chiffrement à clés publiques et à clés privées.
Les mécanismes mis en place pour garantir l'authentification et la confidentialité assurent également la non répudiation. En effet, le mécanisme de poignée de main mis en place pour l'authentification implique l'accord des deux parties. Le mécanisme de confidentialité assure la non modification des messages transmis. L'utilisation d'un service ne peut pas être contestée si les communications sont impossibles sans l'accord des deux parties et si les messages échangés ne peuvent être modifiés.

2.7 Conclusion :

La gestion de ressources dans des systèmes distribués de type clusters ou grilles est importante pour obtenir un ordonnancement optimal. L'observation des ressources est primordiale,

cela permet de connaître la charge des différents composants du système et le placement peut alors en tenir compte.

Il existe différents outils d'observation, le plus complet est NWS car il intègre également la prédiction de performance. Il existe différents types d'algorithmes de placement ; un algorithme dynamique est plus performant qu'un algorithme statique ; l'étude faite dans cette thèse s'orientera vers un algorithme de type batch dynamique avec la possibilité de réserver des ressources. Le "backfilling" et le "gang-scheduling" peuvent permettre d'augmenter la performance. En règle générale, l'utilisateur fournit le nombre de processeurs demandés et l'application est exécutée sans préemption. Des études peuvent être menées pour intégrer la préemption et la migration, l'allocation des processeurs se fait en fonction des caractéristiques du processus et de la charge courante du système, des ordonnancements de type FIFO doivent être améliorés car les processus courts sont retardés par les processus longs. Une API standard est développée à la NASA Ames Research Center pour permettre à un site d'écrire un ordonnanceur pour une variété de travaux parallèles : MPI, PVM pour différentes machines. Trois outils de gestion de ressources ont été présentés : Globus, Legion et Sun Grid Engine ; aucun ne permet de répondre totalement à des exigences en terme de qualité de service et de facturation. Quatre outils de type ASP ont pour terminer été présentés, ils répondent à des besoins très précis.

Pour finir, le gestionnaire de ressources AROMA implémenté durant la thèse a été présenté. Ainsi, l'exécution d'applications de haute performance peut être déportée à distance tout en garantissant une qualité de service et une facturation adéquate. Java et Jini permettent de définir des services dynamiques. Un utilisateur peut se connecter au niveau machine, cluster, domaine ou grille. AROMA peut collecter des informations d'état et utilise ces informations pour la facturation ou pour faire des statistiques pour l'administration.

Chapitre 3

MODELE D'ACCES AUX RESSOURCES

3.1 Introduction

AROMA a comme objectif d'ordonnancer les processus entre les différentes machines. Sur la machine, c'est l'ordonnanceur du système d'exploitation qui arbitre l'accès des processus aux différents processeurs.

Dans ce chapitre, la modélisation de l'accès aux ressources pour les processus sera traitée. La modélisation de l'accès aux ressources peut être basée soit sur la théorie des files d'attente soit sur la théorie des graphes. L'ordonnancement en multiprocesseurs de trois systèmes d'exploitation (Linux, Solaris 7 et Irix 6.5) a été étudié pour comprendre comment l'ordonnanceur choisit un processus, comment l'accès aux différents processeurs est géré. L'objectif était de pouvoir les comparer, en déduire une politique approchée qui puisse être valable si possible pour les trois et permettre ainsi de trouver un modèle exploitable.

Quatre ordonnanceurs : Linux 2.4.2, Linux 2.4.6, Solaris 7 et Irix 6.5 sont présentés puis le modèle déduit est expliqué.

3.2 Ressource processeur

3.2.1 Linux 2.4.2

L'analyse du "scheduling" de Linux s'est basée sur l'étude des fichiers composant le noyau de Linux 2.4.2. Le "scheduler" est implémenté dans le fichier kernel/sched.c (et include/linux/sched.h). Dans cette version de Linux, le "scheduler" a été entièrement réécrit. Tout d'abord quelques généralités seront évoquées puis une des trois politiques de "scheduling" sera détaillée.

A tout processus Linux est allouée une structure task_struct dont voici quelques champs importants :

- volatile long state : état du processus qui peut être : TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, TASK_ZOMBIE.

- volatile long need_resched : 1 si la fonction schedule() doit être rappelée sinon 0.
- long counter : nombre de "ticks" (un tick = 10 ms) d'horloge restant du temps alloué au processus. Cette variable est décrémentée par un timer. Quand sa valeur devient inférieure à 0 elle est remise à 0 et le champ need_resched est positionné à 1. Après un fork(), un fils a une variable counter plus élevée que celle de son père.
- long nice : correspond à la priorité du processus, elle est comprise entre 0 et 19 pour un utilisateur n'ayant pas tous les droits ; seuls les super-utilisateurs peuvent le modifier avec des valeurs négatives. Plus le "nice" est élevé et positif et plus le processus est pénalisé.
- unsigned long rt_priority : priorité des processus temps réel.
- unsigned long policy : politique de placement pour le processus. Il en existe trois :
  SCHED_OTHER : processus UNIX traditionnels.
  SCHED_FIFO : processus temps réel, politique FIFO.
  SCHED_RR : processus temps réel, politique round-robin.
  Seul un processus d'un super-utilisateur peut avoir comme politique SCHED_FIFO ou SCHED_RR. Un processus temps réel fifo tournera jusqu'à : blocage sur une entrée/sortie, abandon explicite de la CPU ou préemption par un autre processus temps réel avec une priorité (rt_priority) plus élevée. Avec la politique SCHED_RR, lorsque le temps alloué (quantum de temps) au processus est terminé, celui-ci est mis à la fin de la "runqueue".
- struct mm_struct mm et struct mm_struct active_mm : ces deux champs représentent respectivement l'espace d'adressage du processus et l'espace d'adressage actif si le processus n'a pas d'espace d'adressage réel. Tous les processus ont un champ active_mm non nul mais les "kernel threads" ont le champ mm NULL. Si ce n'est pas un "kernel thread" mm == active_mm.
- int has_cpu et processor : si le processus a un processeur has_cpu = 1 et le champ processor correspond au processeur sur lequel tourne la tâche sinon cela correspond au processeur sur lequel a tourné la tâche précédemment.

La politique SCHED_OTHER : cette politique correspond à celle utilisée en tant que simple utilisateur.

Initialement, une tâche est dans la file appelée "runqueue" sans processeur alloué et a son champ counter à 10. A l'appel de la fonction schedule(), des tâches sont affectées aux processeurs. Ainsi, pour chaque tâche de la file "runqueue" : l'ordonnanceur vérifie que cette tâche ne tourne pas déjà sur un autre processeur. En effet une tâche dans la file "runqueue" tourne sur un processeur ou est en attente de processeur. Puis il calcule "sa priorité" grâce à la fonction goodness(). Il gardera la tâche renvoyant le meilleur "goodness". Si toutes les tâches renvoient un "goodness" nul (leur temps alloué est terminé) il recalcule le champ counter de tous les processus (pas seulement ceux qui sont dans la file "runqueue"), il re-parcourt la file "runqueue" et recalcule le "goodness" pour tous les processus susceptibles d'être affectés à un processeur. Lorsque le meilleur processus a été trouvé, il modifie les champs has_cpu et processor et il effectue ensuite les changements de contexte nécessaires. Quand une tâche qui possédait un processeur a besoin

d'attendre un événement, elle est supprimée de la "runqueue" et est ajoutée dans la file appelée "waitqueue". Lors de l'appel de la fonction schedule(), une autre tâche prend le processeur.

La fonction reschedule_idle permet de trouver un processeur pour un processus qui en a besoin (soit parce qu'il a perdu son processeur lors d'un appel à schedule() soit parce que le processus vient de se réveiller). Dans un premier temps, l'ordonnanceur regarde si le processeur sur lequel tournait le processus s'est libéré dans ce cas il le lui attribue sinon il choisit le premier processeur oisif, sinon il enlève le processeur à un processus ayant un "goodness" inférieur à celui du processus pour lequel il cherche un processeur.

La fonction goodness() permet de déterminer si un processus est bien placé pour obtenir un processeur. Plus la valeur retournée par la fonction est élevée et plus le processus a des chances d'obtenir un processeur. Différents cas sont envisagés :
* lorsque le temps alloué au processus est terminé (counter = 0) le "goodness" est nul.
* si le processus a tourné précédemment sur ce processeur l'ordonnanceur lui donne un avantage dont la valeur se trouve dans le fichier /asm/smp.h et vaut 15.
* si le processus (p) dont le "goodness" est calculé et le processus précédent ont le même espace d'adressage ou si p est un "kernel thread" alors un avantage de 1 est donné au processus.

En conclusion, le "goodness" se calcule de la manière suivante :
goodness = counter + Avantage(s) + 20 - nice. Si la fonction goodness a renvoyé 0 pour tous les processus de la file "runqueue", les variables counter de tous les processus sont recalculées en fonction du nice de la manière suivante :
counter = counter » 1 + ((20-nice) » 2) + 1.

Remarque : dans le cas de processus temps réel : le "goodness" est calculé uniquement à partir du champ rt_priority :
goodness = 1000 + rt_priority.

La figure 3.1 montre la proposition de modélisation pour l'accès aux processeurs Linux dans le cas d'une machine biprocesseurs. La modélisation pourrait n'utiliser qu'une file d'attente ; dans ce cas, l'algorithme aurait trié les processus de la file. La modélisation propose des files séparées par priorité. Le tri se fait donc avant de mettre un processus dans une file lorsqu'il libère un processeur. Ces files n'existent pas dans la réalité. En A l'affinité processeur est considérée. Lorsqu'un processus quitte un processeur, il est marqué en fonction du processeur sur lequel il a tourné et est placé dans la file correspondant à sa priorité.

3.2.2 Linux 2.6

Le dernier noyau stable Linux possède un nouvel ordonnanceur.

Dans la version 2.4, l'ordonnanceur utilisait un algorithme en O(n). Le temps que met l'ordonnanceur à faire son travail est proportionnel au nombre de tâches se trouvant dans sa file d'attente. Si n est le nombre de tâches en attente alors la complexité est de O(n) (linéaire en fonction du nombre de tâches). Avec le nouveau noyau, quelque soit n, il faudra toujours un temps constant à l'ordonnanceur pour choisir la tâche à traiter (complexité O(1)).

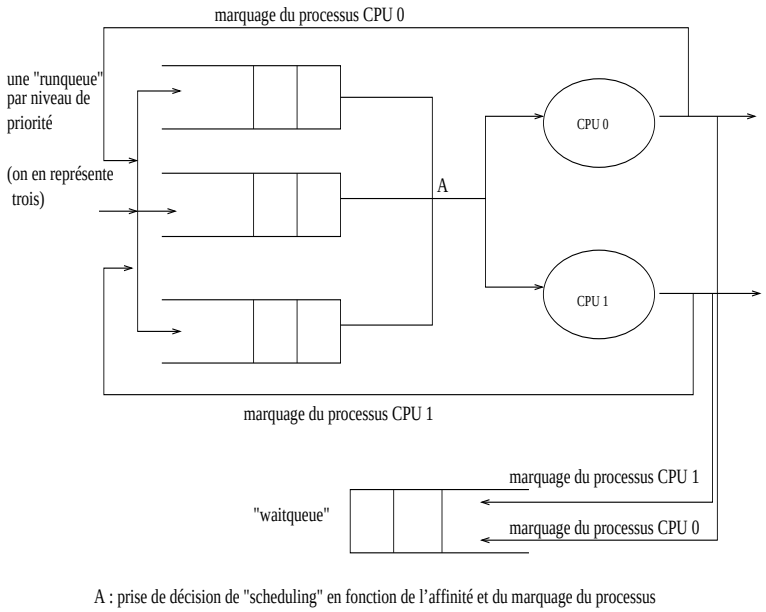


FIG. 3.1: L'ordonnanceur de Linux 2.4

Il existe trois politiques différentes d'ordonnancements :

- deux politiques pour les tâches temps réels (First In First Out et Round Robin)
- une politique pour les tâches utilisateurs dites "normales".

Seule cette dernière politique d'ordonnancement de Linux temps partagé (politique par défaut) est intéressante dans notre cas. Chaque CPU du système possède sa propre file d'exécution (runqueue). Cette file possède deux tableaux de liste de tâches (nommés "active" et "expired") indexés par des niveaux de priorité. Il y a 140 niveaux (plus le niveau est faible, plus la tâche est prioritaire) mais seulement les niveaux compris entre 100 et 139 sont accessibles pour des tâches "normales".

Une valeur nice est allouée à chaque tâche (toujours comprise entre -20 et 19), l'ordonnanceur calcule sa priorité (120 + nice) et la place dans le tableau "active".

La figure 3.2 explique le schéma d'une file d'exécution processeur sous Linux 2.6.

L'ordonnanceur prend dans le tableau "active" la tâche de priorité maximale (valeur la plus faible). Il calcule son quantum de temps nommé "timeslice" puis l'exécute. Si il y a plusieurs tâches ayant la même priorité, l'ordonnanceur effectue un Round Robin entre elles. Quand le "timeslice" est épuisé, il place la (ou les) tâche(s) exécutée(s) dans le tableau "expired" puis passe à la priorité suivante. Quand tout le tableau a été traité, le "scheduler" replace les tâches dans le tableau "active" (avec certaines modifications au niveau des priorités dynamiques dans le tableau mais aucune au niveau des priorités statiques).

3.2.3 Solaris 7

Solaris est un système d'exploitation de SUN implémenté sur différents systèmes allant des systèmes monoprocesseur à des systèmes de 64 processeurs [9]. Solaris supporte le "multithreading", un processus peut avoir de nombreux "threads" qui partagent le contexte du

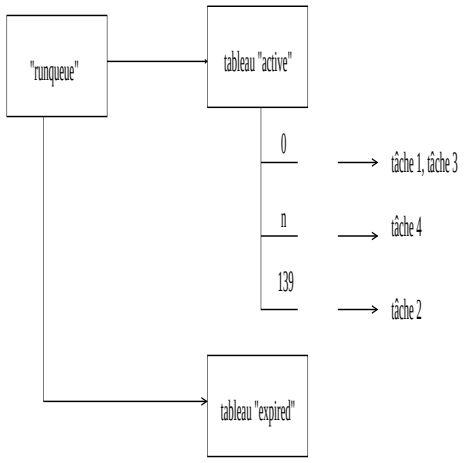


FIG. 3.2: L'ordonnanceur de Linux 2.6

processus et peuvent être ordonnancés séparément, ce qui permet un haut-degré de parallélisme sur les systèmes multiprocesseurs. Tout d'abord quelques généralités seront explicitées, puis l'organisation des files d'attente de Solaris sera traitée.

A chaque processsus est allouée une structure proc contenant toutes les informations nécessaires pour que le noyau puisse ordonnancer l'accès du processus aux processeurs. Il existe différentes notions : thread utilisateur (user thread), thread noyau (kernel thread) et processus léger (LightWeight Process : LWP). La figure 3.3 montre le modèle processus / thread / lwp de Solaris.

Les "user threads" sont créés explicitement par le programmeur et sont gérés par la librairie des "threads". Un processus sans "thread" aura un "LWP" et un "kernel thread" associé ; le noyau crée et lie les structures quand le processus est créé. Le modèle en multi-couches permet de découpler le "scheduling" et la gestion des threads utilisateurs du noyau. Les "user threads" ont leur propre schéma de priorités et sont ordonnancés par un "scheduling thread" créé par la librairie lorsqu'un programme multithread est compilé et exécuté. Les "user threads" ne sont pas visibles par le noyau ; ils ne sont visibles que lorsqu'ils sont liés à un "LWP". En général les applications multithread ont plus de threads utilisateurs que de "LWPs". Les threads de niveau utilisateur sont donc rendus exécutables par le "scheduler" de threads en les liant à un "LWP" associé à un "kernel thread". C'est le "kernel thread" qui appartient à une classe de "scheduling", qui a une priorité, qui est mis dans la "dispatch queue" et qui est ordonnancé.

Les "LWPs" et les "kthreads" (kernel threads) ont deux structures distinctes déclarées respectivement dans sys/klwp.h et sys/thread.h. Parmi les informations maintenues par les "LWPs" il y a : le process control block (pcb) structure, l'état courant, un pointeur sur le kthread, un pointeur sur la structure du processus. Parmi les informations maintenues par les "kthreads" il y a : link pointer (lie le "kthread" avec les autres "kthreads" de la même file : "dispatch queue", "sleep queue"), CPU affinity, priorité de "scheduling", classe de "scheduling", "dispatcher queue" et timer, wait channel (ce qu'attend un kthread quand il est bloqué), état (running, runnable, sleeping, stopped, zombie), pointeurs sur le LWP et proc structure,

identificateur de thread...

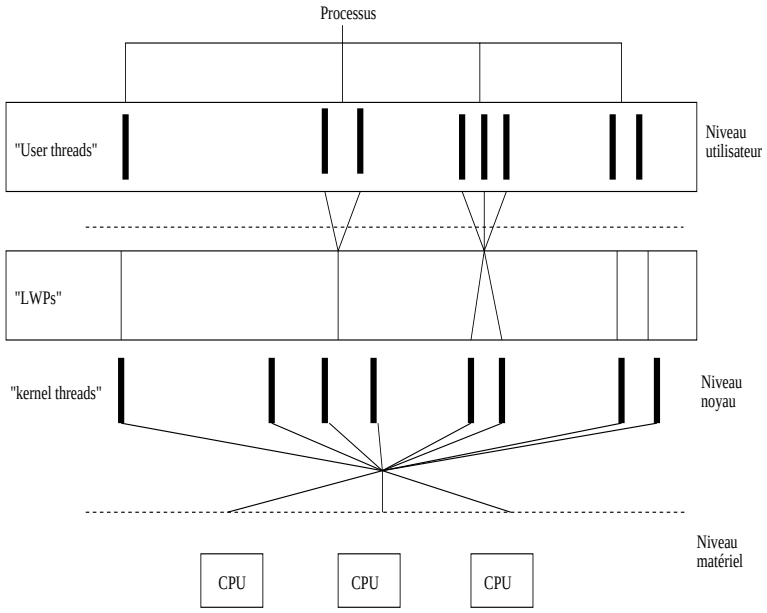


FIG. 3.3: Le modèle processus/thread/lwp de Solaris

Il existe différentes classes de "scheduling" déterminant les politiques et les algorithmes appliqués. Pour chaque classe, il existe une table de valeurs et de paramètres utilisée par le "dispatcher" pour sélectionner un "thread", lui attribuer un processeur, lui donner une priorité basée sur le temps d'attente et le temps passé en exécution sur le processeur. Les classes de "scheduling" sont au nombre de quatre :

- TS : temps partagé : le "dispatcher" attribue des quantums de temps aux "threads" de manière équitable : les "threads" qui ont attendu voient leur priorité augmenter ; les "threads" qui ont utilisé le temps attribué voient leur priorité diminuer (priorités de 0 à 59).
- IA : interactive (les processus du système de fenêtres ou d'OpenWindows ou CDE seront placés dans la classe de scheduling IA) ; priorités de 0 à 59.
- RT : temps réel : priorités de 100 à 159.
- SYS : système : cette classe est utilisée par les daemons de Solaris ou par un "thread" de la classe TS qui possède une ressource critique et se voit alors attribuer une priorité SYS (de 60 à 99).

Les TS, IA et RT classes de scheduling sont implémentées par des modules du noyau dynamiquement chargeables. La classe SYS fait partie du noyau. Par défaut, les classes temps partagé, système et interactive sont chargées. Chaque classe de "scheduling" a une "dispatch table" correspondante contenant des valeurs par défaut pour les priorités et le réajustement de priorité. La "dispatch table" de la classe temps réel contient moins d'informations que pour la classe TS ou IA car un thread temps réel a une priorité fixe, il n'y a donc pas de réajustement de la priorité au cours de la vie des threads temps réels (leur priorité peut juste être modifiée explicitement par un appel système). La classe IA utilise la TS "dispatch table". Il n'y a pas

de "dispatch table" pour la classe SYS ; les "threads" appartenant à cette classe n'ont pas de quantum de temps et leurs priorités ne sont pas recalculées.

Chaque "kernel thread" pointe sur une structure spécifique de la classe de "scheduling" nommée xxproc (xx étant soit ts soit rt) pour que le "dispatcher" connaisse le temps d'exécution, le temps d'attente, la priorité du thread. Les threads de la classe IA ont un lien vers une ts_proc structure ; le champ ts_flags sera TSIA pour les threads interacifs ; le "dispatcher" pourra ainsi distinguer les TS threads et les IA threads. La figure 3.4 montre le lien entre les différentes structures.

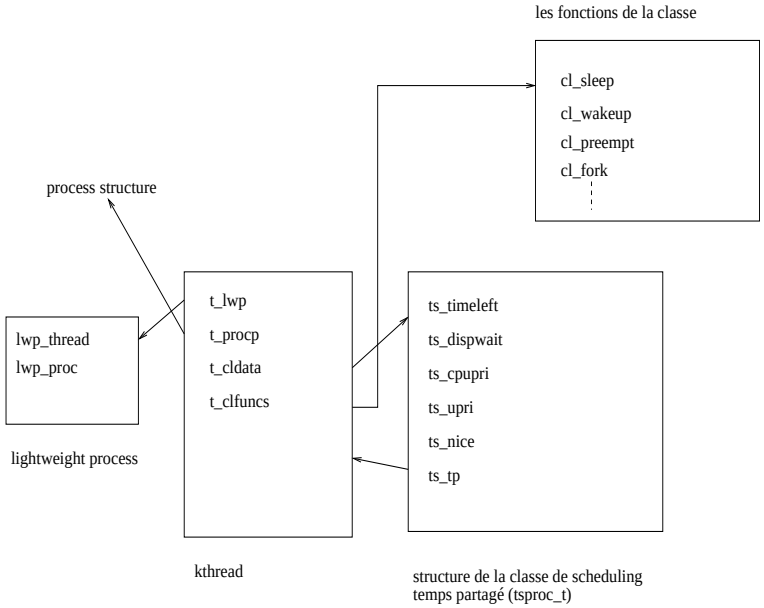


FIG. 3.4: Les structures des classes de "scheduling" de Solaris

Un thread hérite de sa classe de "scheduling" et de sa priorité lors de sa création.

Pour chaque processeur, il y a une "dispatch queue" par niveau de priorité. Il y a par ailleurs une "kernel preemption queue" accessible par tous les processeurs ou une "preemption queue" par partition de processeurs si une partition a été définie. La "kernel preemption queue" est constituée de files de threads par niveau de priorité ; elle contient les threads pouvant provoquer une préemption : threads temps réel ou threads d'interruption. Lorqu'un processus attend un événement, il est rajouté dans la file "waitqueue". La file "waitqueue" de Solaris est une file de files : une file de processus par événement attendu.

La figure 3.5 illustre l'organisation des différentes files ("dispatch queues" et "preempt queue", en supposant qu'il n'y a pas de partition de processeurs).

Le "dispatcher" cherche le thread prêt à tourner qui possède la meilleure priorité, place le thread sur un processeur et gère l'insertion, la suppression de threads dans les "dispatch queues". Le "dispatcher" regarde régulièrement les files : il commence par les "preempt queues"

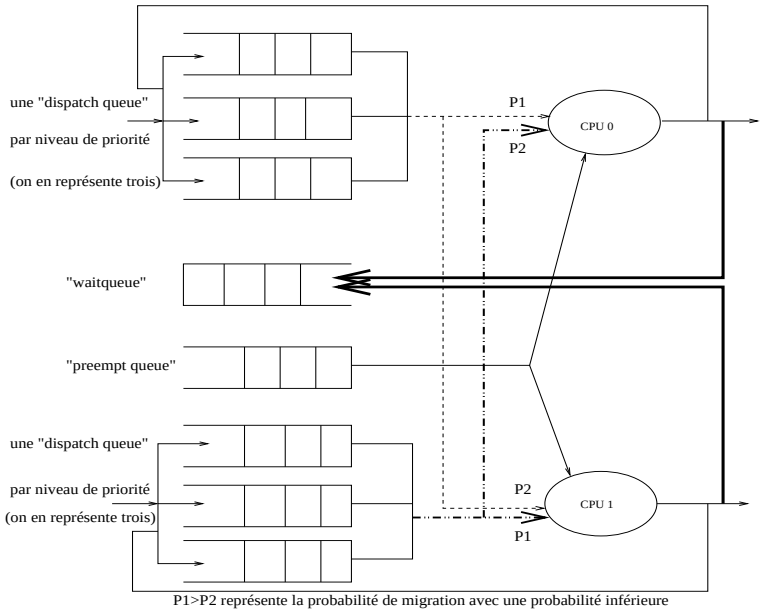


FIG. 3.5: Les différentes files de Solaris

puis il regarde les "dispatch queues" des CPUs. Chaque CPU peut utiliser le "dispatcher" en concurrence pour chercher du travail dans les files. S'il n'y a pas de thread en attente dans les files du processeur, le "dispatcher" ira chercher dans les files d'un autre processeur et il y aura alors migration de thread. Le "dispatcher" vérifie également que la charge dans les différentes files est bien répartie sur tous les processeurs. Solaris tient compte d'un certain niveau d'affinité en plaçant les threads dans une file : un thread sera placé dans une file du processeur sur lequel il a tourné précédemment (cette information est contenue dans le champ t_cpu pour chaque thread). Si cela fait longtemps que le thread a tourné, la probabilité de trouver des données ou des instructions du thread dans le cache diminue, il choisit alors le processeur qui exécute le thread qui a la plus faible priorité. L'exemple suivant illustre cela : un thread quitte le processeur parce qu'il attend une ressource, supposons qu'il obtienne la ressource longtemps après, la probabilité de trouver des données ou des instructions dans le cache est faible, il choisit alors d'insérer le thread réveillé dans une "dispatch queue" du processeur qui exécute le thread de plus faible priorité. Dans le cas de threads pour lesquels un processeur a été explicitement choisi par un appel de fonction, ce processeur sera toujours sélectionné. Une fois le processeur choisi, en fonction de la priorité du thread il détermine la file dans laquelle il est placé.

La sélection du thread pour lui attribuer un processeur est implémentée grâce à un algorithme de sélection et ratification. Le "dispatcher" trouve le thread possédant la plus forte priorité en examinant les "dispatch queues". Pour cela, chaque "dispatch queue" maintient un champ disp_maxrunpri contenant la valeur de la plus forte priorité de la file. Le "dispatcher" cherche tout d'abord dans la "kernel preempt queue", si elle est vide il étudie chaque "dispatch queue" du processeur ; le thread le plus prioritaire est sélectionné. La fonction "ratify()" permet de vérifier que la sélection est bien la meilleure pour le processeur. Si la priorité du thread choisi est supérieure à la priorité maximale dans la "preempt queue" et au champ

disp_maxrunpri de la file du processeur alors le choix est accepté. Sinon un autre thread est cherché. Pendant la ratification, les flags cpu_runrun et cpu_kprunrun du processeur sont mis à jour. Ces flags sont stockés dans la CPU structure de chaque processeur. Le flag cpu_runrun indique une "user preemption" et se produit lorsqu'un thread plus prioritaire que le thread qui s'exécute mais avec une priorité inférieure à kpreemptpri est inséré dans une "dispatch queue". Le flag cpu_kprunrun indique une "kernel preemption" due à un thread de la classe RT ou à un thread d'interruption.

3.2.4 Irix 6.5

IRIX 6.5 est un système d'exploitation de SGI présent dans des machines comme Origin2000, Origin200 [10]. Ce sont des machines multiprocesseurs à mémoire partagée basées sur l'architecture SN0 (Scalable Node 0). Une architecture à mémoire partagée a de nombreux avantages : chaque CPU accède équitablement et directement à toute la mémoire ; mais la scalabilité n'est pas satisfaisante : en effet plus le nombre de CPUs augmente et plus le bus d'accès à la mémoire est saturé. Par opposition, dans les architectures à mémoire distribuée, chaque CPU ne peut accéder directement qu'à sa mémoire locale ; pour accéder à la mémoire distante, des copies doivent être envoyées d'une CPU à une autre. Il n'y a alors plus de problème de scalabilité dû à la bande passante du bus mais la programmation est moins simple.

Dans l'architecture SN0, la mémoire est physiquement distribuée mais l'espace d'adressage global est unifié par le "hardware" ; ainsi la programmation reste aisée et la scalabilité est conservée. L'Origin2000 est un supercalculateur ccNUMA (cache-coherent Non-Uniform Memory Access) utilisant l'architecture à mémoire distribuée globalement. Il est constitué d'un ensemble de noeuds de traitement reliés entre eux par une matrice d'interconnexion appelée CrayLink. Chaque noeud comporte un ou deux processeurs, une partie de la mémoire partagée et deux interfaces : pour les entrées/sorties et pour la liaison avec la matrice d'interconnexion. Des modules appelés Routeurs sont utilisés pour interconnecter les noeuds entre eux dans une topologie d'hypercube à plusieurs niveaux. Cette architecture minimise le temps de latence entre les différents processeurs, offre des chemins redondants entre les noeuds et assure une réelle augmentation de la bande passante avec l'augmentation du nombre de processeurs.

L'algorithme de "scheduling" par défaut d'IRIX est le temps partagé sans priorité. Toutes les 10 ms, pour chaque CPU, le noyau doit prendre une décision de "scheduling" ; cet intervalle de temps est appelé "tick". Chaque CPU est interrompue par un timer à chaque "tick" ; mais comme dans un système multiprocesseurs, les CPUs ne sont pas nécessairement synchronisées, les CPUs peuvent avoir les "ticks" d'interruption à différents moments. Pendant l'interruption, le noyau met à jour certaines valeurs, choisit le prochain processus qui s'exécutera (en principe le processus interrompu sauf si un processus plus prioritaire est prêt à tourner) et vérifie les signaux pendants. Les interruptions tous les "ticks" permettent la préemption ; toutefois il est possible d'annuler les interruptions de "ticks" dans certaines CPUs les rendant ainsi non préemptives. A chaque processus est alloué un "time slice" durant lequel il doit pouvoir s'exécuter sans être préempté. Par défaut, le "time slice" est de 10 "ticks" (100 ms) pour les systèmes multiprocesseurs et de 2 "ticks" (20 ms) pour les systèmes monoprocesseur. A la fin du "time slice", le noyau choisit le processus qui utilisera ce processeur ; le choix se fait selon les priorités des processus mais si deux processus ont la même priorité le noyau les fait tourner à tour de rôle. La durée du "time slice" peut être modifiée pour tous les processus

en utilisant la commande "systune" ; la durée du "time slice" d'un processus donné peut être modifiée grâce à la fonction "schedctl()".

Les applications temps partagé ne sont pas ordonnancées en fonction de la priorité ; tandis que les applications temps réel sont ordonnancées en fonction de la priorité. Il est tenu compte de l'affinité du processus avec le processeur. En effet, quand un processus s'exécute, des données et des instructions sont chargées dans le cache, ce qui crée une affinité entre le processus et le processeur : aucun autre processus ne peut utiliser le processeur plus efficacement et le processus ne peut s'exécuter plus rapidement sur une autre CPU. Le noyau note sur quel processeur le processus s'est exécuté en dernier et mesure l'affinité. Plus le temps d'exécution d'un processus sur un processeur est long et plus son affinité est élevée. Quand un processus abandonne la CPU, soit parce que son temps alloué est terminé soit parce qu'il est bloqué, trois choses peuvent se produire pour la CPU : la CPU exécute le même processus immédiatement, la CPU est oisive en attente de travail ou la CPU exécute un autre processus. Les deux premières actions ne réduisent pas l'affinité du processus mais quand la CPU exécute un autre processus, ce processus voit son affinité augmenter tandis que le précédent voit son affinité diminuer. Tant qu'un processus a une affinité non nulle avec un processeur il est exécuté par cette CPU ; quand son affinité est nulle, il peut être exécuté par n'importe quelle CPU. Ainsi, les processus liés aux entrées/sorties qui s'exécutent par périodes courtes et n'ont pas beaucoup d'affinité sont vite ordonnancés lorsqu'ils sont prêts à s'exécuter tandis que les processus liés aux processeurs qui ont une grande affinité ne sont pas ordonnancés aussi rapidement car ils attendent que "leur" CPU soit libre.

La figure 3.6 montre comment l'accès des processus aux processeurs peut être modélisé pour Irix : aucune distinction n'est faite pour les files en fonction de la priorité et il y a une file par processeur pour tenir compte de l'affinité.

En A les décisions sont prises en fonction de l'affinité. Le processeur CPU0 traite les processus de sa file ou ceux de la file CPU1 qui ont une affinité nulle en faisant du "round-robin" sans priorité.

3.2.5 Le modèle

Si l'ordonnancement des trois systèmes étudiés est comparé, des similitudes apparaissent : tous tiennent compte de l'affinité du processus avec le processeur, Linux et Solaris tiennent compte de la priorité des processus, chaque processus se voit attribuer un quantum de temps, c'est donc du temps partagé.

Pourtant, un modèle le plus simple possible doit être choisi. En effet, plus le modèle sera simple, plus il se résoudra mathématiquement vite et donc il sera possible de déterminer rapidement le temps d'exécution d'une application. Une simplification du modèle a été cherchée ; pour cela, le raisonnement a été le suivant :

- la priorité des processus est négligeable : ce sont les processus d'un utilisateur qui vont de manière globale constituer la charge de la machine, or tous ces processus ont la même priorité car un utilisateur quelconque cherche toujours à avoir une priorité maximum et c'est ce qui lui est attribué par défaut. Il n'y aura donc plus une file par priorité.

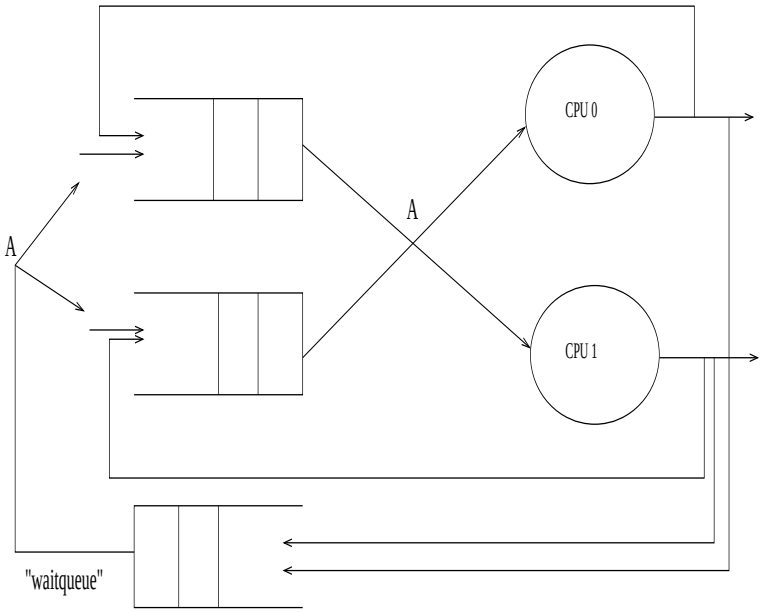


FIG. 3.6: Les différentes files d'Irix

- l'affinité du processus avec un processeur est négligeable : s'il y a beaucoup de processus, globalement au cours du temps ils finiront par accéder à n'importe quel processeur, en moyenne l'affinité avec un processeur influera peu.

- des mesures ont montré qu'il existe une forte corrélation entre le temps d'exécution d'un processus et la taille de la file des processus en attente d'exécution [11]. Seule la file des processus en attente d'exécution ("runqueue") est considérée et pas la file des processus en attente d'événement ("waitqueue").

Finalement, il y a une file d'attente globale à tous les processeurs, et c'est du temps partagé donc après avoir accédé à un processeur, un processus a la possibilité de quitter le système ou de reboucler dans la file d'attente. Le modèle choisi est un modèle "round-robin" multi-serveurs sans priorité [45] (figure 3.7).

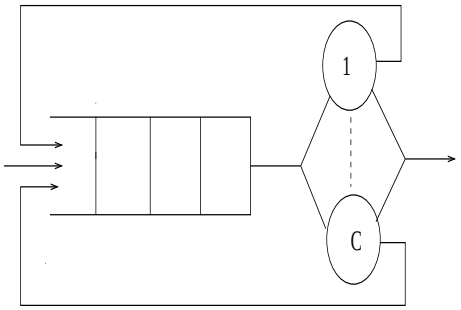


FIG. 3.7: Modèle choisi

3.3 Conclusion

Ce chapitre a présenté trois systèmes d'exploitation : Linux, Solaris et Irix. L'accès aux processeurs est différent mais une modélisation simplifiée et commune a été proposée. Les chapitres suivants s'appuieront sur ce modèle en considérant deux possibilités : une charge déterministe du système ou une charge stochastique.

Chapitre 4

MODELE D'EXECUTION AVEC CHARGE STOCHASTIQUE

4.1 Introduction

De nombreux phénomènes sont décrits par des modèles stochastiques : dans le domaine des télécommunications [95], dans le domaine des réseaux (structures de réseaux et "packet switching") [96] et dans le domaine informatique (systèmes à accès multiples, systèmes d'exploitation) [97] [98]. L'objectif de ces études est d'extraire des informations sur le système dans l'état stationnaire et parfois dans l'état transitoire. Un autre aspect est le comportement de systèmes dont les files d'attente sont à capacité finie [99] ou après un événement spécifique (panne d'un serveur par exemple). Ici il est considéré un événement spécifique qui est l'arrivée d'une charge de travail parfaitement connue (déterministe) dans la charge stochastique du système. L'objectif est d'évaluer le temps de réponse à cet événement. Le système étudié est l'ordonnancement de tâches sur des machines multiprocesseurs avec politique de temps partagé (modèle de files d'attente multiserveurs "round robin") et l'événement est l'arrivée de processus dont les caractéristiques sont connues (applications parallèles régulières). Le modèle de machines multiprocesseurs est d'abord détaillé, puis le problème et ses motivations sont expliqués, dans une troisième partie la chaîne de Markov intégrant les processus déterministes est présentée. Pour finir, les approximations sont décrites.

4.2 Modélisation de machines multi-processeurs

Un placement efficace des processus nécessite une modélisation du support d'exécution : les machines multi-proccesseurs et les réseaux. Le modèle choisi a été expliqué dans le chapitre précédent. La partie concernant les processeurs va être détaillée et les formules déduites vont être expliquées. L'étude de l'influence des processus déterministes (processus pour lequels un placement est cherché) sur la charge supposée stochastique du système a été faite. Les processus qui arrivent sont supposés déterministes et la charge des machines stochastiques.

Le modèle est basé sur les files d'attente : c'est un modèle "round robin". Il est supposé que les processus arrivent dans le système selon une loi de Poisson de taux λ. Le processus à la tête de la file aura accès au premier processeur libre pendant un quantum de temps Q. A la fin du quantum, le processus quittera le système avec une probabilité $1 - \sigma$ ou repartira en fin de file avec une probabilité σ. x est défini comme le nombre de processus dans le système.

La figure 4.1 représente le modèle.

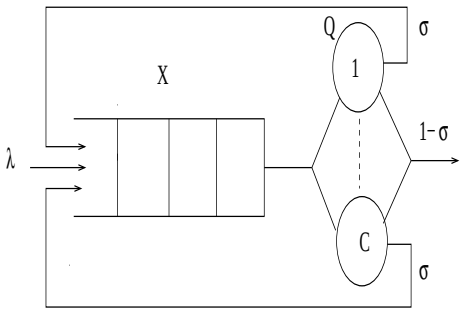


FIG. 4.1: Modèle "round-robin" multi-serveurs

La probabilité g_i pour un processus d'accéder à la ressource processeur i fois et la moyenne de temps passé sur un serveur E(T) sont :

$$g_i = \sigma^{(i-1)}(1-\sigma), \quad E(T) = \sum_{i=1}^{\infty}(iQ)g_i = \frac{Q}{1-\sigma} = \frac{1}{\mu} \tag{4.1}$$

4.3 Présentation du problème et motivation

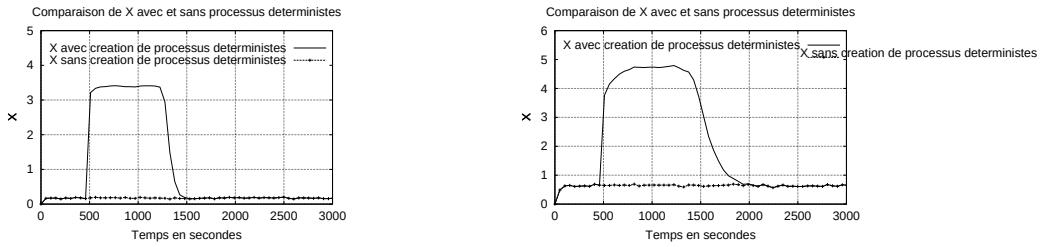
L'objectif de l'étude est de placer une application sur des stations multi-processeurs en optimisant l'utilisation des ressources. Il est supposé que le temps processeur demandé par l'application est connu, l'application est déterministe contrairement aux autres applications qui s'exécutent sur les machines qui constituent la charge stochastique. Le temps d'exécution des processus déterministes doit être prédit. Le comportement des machines après l'insertion des processus déterministes a été observé pour évaluer l'influence des processus déterministes sur la charge stochastique. Le nombre de processus dans le système avec ou sans création des processus déterministes a été comparé. Tous les processus (déterministes et stochastiques) sont ordonnancés avec la politique "round-robin" avec un quantum Q. Un modèle "round-robin" sans priorité et multiserveur a été simulé. Les processus stochastiques sont créés selon une loi de Poisson, ils utilisent les processeurs pendant une durée distribuée selon une loi exponentielle. A une date fixée, les processus déterministes sont insérés. Une liste de travaux constitue la charge de la machine. Les processus sont créés virtuellement : aucun "fork()" n'est fait mais la création d'un processus consiste en l'ajout d'un nouveau processus dans la liste. Les processus déterministes sont distingués des autres avec un principe de marquage.

Deux cas sont présentés dans le tableau 4.1 : le taux d'arrivée est appelé λ, le taux de service μ, le nombre de processus déterministes d, le nombre de serveurs C et le taux d'utilisation ρ (ρ=λ/Cμ). Les processus déterministes sont insérés à la date t=500, ils demandent chacun 500 unités de temps de calcul.

Un exemple de moyenne du nombre de processus dans le système $X(t)$ sur plusieurs simulations avec différentes initialisations du générateur de charge stochastique est décrit sur la figure (4.2).

Paramètres	Cas 1	Cas 2
λ	0.006	0.02
μ	0.03333	0.03333
ρ	0.09	0.3
d	3	3
C	2	2

TAB. 4.1: Etude de l'influence des processus déterministes sur la charge stochastique



(a) Cas 1

(b) Cas 2

FIG. 4.2: Influence des processus déterministes sur la charge stochastique

Dans le premier cas, comme le taux d'utilisation de la machine est faible l'influence des processus déterministes sur les stochastiques n'est pas très importante. Par contre, dans le second cas, la charge est plus élevée donc l'influence est plus importante c'est à dire que la charge n'est pas simplement égale à la somme des processus stochastiques et déterministes. Ce phénomène augmente avec le taux d'utilisation et on peut conclure qu'il ne faut pas le négliger. Dans la prochaine partie, la chaîne de Markov du modèle intégrant les processus déterministes est présentée.

4.4 La chaîne de Markov

Une application est composée de d processus. d est alors le nombre de processus déterministes insérés simultanément et C le nombre de processeurs. Le temps processeur demandé par les processus déterministes est connu, il est appelé t_c (tous les processus ou tâches de l'application ne demandent pas nécessairement le même temps). L'état du système à la date t est donné par le nombre de processus à cette date. L'état n signifie qu'il y a $x = n$ processus dans le système. Le quantum de temps Q est petit comparé aux autres valeurs temporelles du modèle ainsi il est possible de considérer une chaîne de Markov continue. Le nouvel état du système à la date t + Q dépend uniquement de l'état à la date t et de la probabilité d'avoir un départ ou une arrivée. Ainsi, le système est Markovien.

Il ne peut y avoir moins de d processus dans le système après l'arrivée de processus déterministes jusqu'à ce que le premier quitte le système. Ainsi, les états en dessous de d dans la chaîne de Markov ne peuvent pas être atteints. La probabilité d'avoir un nouveau processus pendant un quantum Q est égale à λQ. S'il y a moins de processus que de processeurs alors

tous les processus sont en exécution. S'il y a plus de processus que de processeurs alors C processus sont en exécution et $(n - C)$ processus sont en attente. Le processus à la tête de la file s'exécutera sur le premier serveur libre.

Considérons un état n avec $n < C$, tous les processus ont un serveur et sont susceptibles de quitter le système. Ainsi, la probabilité qu'un des n processus (excepté les déterministes) quitte le système est égal à $(n - d) * \mu Q$. Considérons maintenant un état n avec $n > C$, seulement C processus ont un serveur et seulement les stochastiques peuvent quitter le système. Comme il y a $n - d$ processus stochastiques et n processus au total, la probabilité qu'un processus stochastique quitte le système est égale à $(n-d)/n$ et donc la probabilité d'atteindre l'état n-1 est égale à $C * ((n - d)/n)\mu Q$. Sur la figure 4.3 le cas $d < C$ est présenté et sur la figure 4.4 le cas d $\geq$ C est décrit.

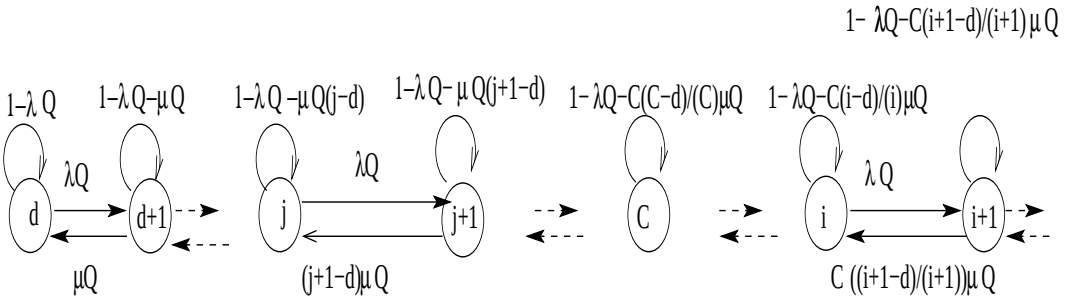


FIG. 4.3: Chaîne de Markov lorsque $d < C$

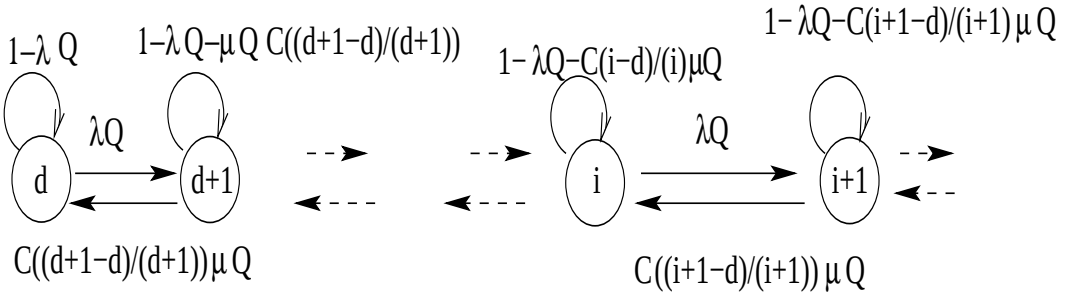


FIG. 4.4: Chaîne de Markov lorsque $d \geq C$

La chaîne de Markov représente l'évolution du nombre de processus dans le système complet après l'insertion des processus déterministes. Les probabilités de transition peuvent être calculées.

$P_i(t)$ est la probabilité d'avoir i processus dans le système à la date t et $\dot{P_i}(t)$ la dérivée de $P_i(t)$. L'équation (4.2) exprime le cas $d < C$ et l'équation (4.3) le cas d $\geq$ C.

$$\begin{array}{rlll}
\text{Pour } d < i < C : & \dot{P_i}(t) = \frac{\partial P_i(t)}{\partial t} = & \lambda P_{i-1}(t) + (i+1-d)\mu P_{i+1}(t) \\
 & & -(\lambda + (i-d)\mu) P_i(t) \\
\text{Pour i=d} : & \dot{P_d}(t) = \frac{\partial P_d(t)}{\partial t} = & \mu P_{d+1}(t) - \lambda P_d(t) \\
\text{Pour i} \geq \text{C} : & \dot{P_i}(t) = \frac{\partial P_i(t)}{\partial t} = & \lambda P_{i-1}(t) + C\mu \frac{(i+1-d)}{i+1} P_{i+1}(t) \\
 & & -(\lambda + C\mu \frac{i-d}{i}) P_i(t)
\end{array} \tag{4.2}$$

$$\begin{array}{rlll}
\text{Pour } d < i : & \dot{P_i}(t) = \frac{\partial P_i(t)}{\partial t} = & \lambda P_{i-1}(t) + C\mu \frac{(i+1-d)}{i+1} P_{i+1}(t) \\
 & & -(\lambda + C\mu \frac{(i-d)}{i}) P_i(t) \\
\text{Pour i=d} : & \dot{P_d}(t) = \frac{\partial P_d(t)}{\partial t} = & \frac{C}{d+1} \mu P_{d+1}(t) - \lambda P_d(t)
\end{array} \tag{4.3}$$

Un état stationnaire est défini comme étant un état infini où les processus déterministes sont encore dans le système. C'est comme un système dans lequel les processus déterministes ne quitteraient jamais les files. En effet, c'est leur temps d'exécution qui est prédit donc le comportement de la machine est uniquement intéressant lorsqu'ils sont encore présents. Ainsi le nombre de processus déterministes d doit être considéré dans la condition de stabilité de la chaîne.

La chaîne de Markov est irréductible car chaque état peut être atteint à partir d'un autre. Ainsi, les probabilités limites existent toujours et sont indépendantes de la distribution initiale. Tous les états sont récurrents non-nul si après un temps infini leur état n'est pas infini. Donc la condition est que le taux d'arrivée doit être plus petit que le taux de sortie (équation 4.4).

$$f(i) = \frac{\lambda}{C\mu(\frac{i+1-d}{i+1})} < 1 \tag{4.4}$$

f(i) est définie pour les états $i \geq d$, elle est décroissante et le maximum est pour $i = d$ sa valeur est : $\lambda(d+1)/C\mu$

Ainsi la condition est satisfaite si la relation (4.5) est vérifiée :

$$\frac{\lambda(d+1)}{C\mu} < 1 \tag{4.5}$$

4.4.1 Le nombre moyen de processus

$X(t)$ est le nombre moyen de clients dans le système et X^{∞} le nombre moyen de clients à l'état stationnaire. X^{∞} peut être calculé à partir de la chaîne de Markov et des équations de balance [100]. La théorie du trafic différentiel [23] [101] va être utilisée ; elle consiste à trouver une équation avec $X(t)$ et sa dérivée.

Lorsque $d < C$:

$$X^{\infty} = (\sum_{i=d}^{C-1} (\frac{\lambda}{\mu})^{(i-d)} * \frac{i}{(i-d)!} + \sum_{i=C}^{\infty} (i(\frac{\lambda}{C\mu})^{(i-C)}) * \frac{i!}{C!(i-d)!} * (\frac{\lambda}{\mu})^{(C-d)}) P_d^{\infty} \tag{4.6}$$

$$P_d^{\infty} = \frac{1}{\sum_{i=d}^{C-1} (\frac{\lambda}{\mu})^{(i-d)} \frac{1}{(i-d)!} + \sum_{i=C}^{\infty} (\frac{\lambda}{C\mu})^{(i-C)} * \frac{i!}{C!(i-d)!} * (\frac{\lambda}{\mu})^{(C-d)}} \tag{4.7}$$

Lorsque d $\geq$ C :

$$X^{\infty} = \sum_{i=d}^{\infty} i(\frac{\lambda}{C\mu})^{(i-d)} * \frac{i!}{d!(i-d)!} P_d^{\infty} \tag{4.8}$$

$$P_d^{\infty} = \frac{1}{\sum_{i=d}^{\infty} (\frac{\lambda}{C\mu})^{(i-d)} * \frac{i!}{d!(i-d)!}} \tag{4.9}$$

$$\dot{X}(t) = \frac{\partial X(t)}{\partial t} = \sum_{i=0}^{+\infty} i\dot{P}_i(t) \tag{4.10}$$

$\dot{X}(t)$ est la dérivée de X(t). A partir des équations (4.2) et 4.10 : en réindexant et en simplifiant l'expression, il est déduit que lorsque $d < C$ $\dot{X}(t)$ obéit à l'équation différentielle (4.11) :

$$\dot{X}(t) = \lambda - C\mu(1 - P_d(t)) + \mu \sum_{i=d}^{C-1} iP_{d+C-i}(t) + C\mu d \sum_{i=C+1}^{\infty} (\frac{P_i(t)}{i}) \tag{4.11}$$

Lorsque d$\geq$C il est déduit de l'équation (4.3) que $\dot{X}(t)$ obéit à l'équation différentielle suivante :

$$\dot{X}(t) = \lambda - C\mu(1 - P_d(t)) + C\mu d \sum_{i=d+1}^{\infty} (\frac{P_i(t)}{i}) \tag{4.12}$$

Si la mesure du nombre de processus i_0 dans le système à la date t_0 est donnée alors la distribution des probabilités à la date t_0 est connue :

$$P_{i_0}(t_0) = 1, \quad P_i(t_0) = 0, \quad i \neq i_0 \tag{4.13}$$

Les équations (4.2) (4.11) ou (4.3) (4.12) donnent l'évolution dans le système pour $t > t_0$. Néanmoins, les équations (4.11) et (4.12) sont difficiles à manipuler en raison des probabilités de transition dans les formules. Au paragraphe 4.5, une approximation sera proposée.

4.4.2 Prédiction du temps d'exécution

Considérons un processus demandant t_c unités de temps de calcul. Il est supposé que le processus arrive à la date t_0. Le processus déterministe quittera le système à une date t_r. $x(t)$ est défini comme le nombre de processus dans le système (les déterministes et les stochastiques) à la date t. Comme le nombre de processus dans le système ne sera pas constant entre t_0 et t_r et que le modèle proposé du système UNIX utilise une politique de time sharing avec Q très petit, l'équation (4.14) peut être donnée.

$$t_c = \int_{t_0}^{t_r} \frac{C}{x(t)} dt \tag{4.14}$$

Dans la prochaine partie, les approximations sont présentées.

4.5 Approximations

Les équations (4.11) et (4.12) ne peuvent pas être utilisées car $P_i(t)$ doit être connu pour t appartenant à l'intervalle $[t_0 ; t_r]$ et pour tout i. De plus il y a des sommes infinies. Cela n'est pas compatible avec une résolution temps réel et une approximation doit être trouvée. $X_a(t)$ est l'approximation de $X(t)$. Dans [44] un modèle pour les machines monoprocesseur a été proposé (modèle M/M/1), la prédiction du nombre moyen de client est donné par l'équation (4.15) :

$$\dot{X}(t) = \lambda - \mu[1 - P_0(t)] \tag{4.15}$$

Le terme $P_0(t)$ a été approximé grâce à une approximation fluide : dans l'état stationnaire d'une file M/M/1 la relation (4.16) est vraie :

$$1 - P_0^{\infty} = \frac{X^{\infty}}{1 + X^{\infty}} \tag{4.16}$$

L'approximation fluide consiste à étendre la relation valable en stationnaire au domaine transitoire (4.17) :

$$1 - P_0(t) = \frac{X(t)}{1 + X(t)} \tag{4.17}$$

Ainsi, l'approximation du nombre moyen de clients est (4.18) :

$$\dot{X}_a(t) = \lambda - \mu \frac{X_a(t)}{1 + X_a(t)} \tag{4.18}$$

Il y a peu de différences entre le modèle étudié dans [44] et le modèle de machines multiprocesseurs étudié ici. Dans le modèle, il y a C serveurs (et plus un seul) et le modèle est "round robin" (et plus un modèle M/M/1). Cependant un modèle "round robin multiserveurs" est équivalent à un modèle M/M/C dans l'état stationnaire. Ainsi, la même approche peut

être utilisée en introduisant le nombre de processeurs C. Dans les équations (4.11) et (4.12) il y a le terme $C\mu(1-P_d(t))$ qui est similaire au terme $\mu(1-P_0(t))$ dans le cas M/M/1. Le terme $C\mu(1-P_d(t))$ est approximé par la fonction $\frac{X(t)}{1+X(t)}$, le terme $C\mu d$ est conservé et un coefficient α est introduit. La valeur de α est choisie pour garantir que lorsque $\dot{X}(t) = 0$ (stationnaire) X_a^∞ tend vers X^∞ donné par les équations (4.6), (4.8).

La relation (4.19) exprime que la variation des processus dans le système est la différence entre le nombre d'arrivées et de départs des processus. Les processus de charge quittent le système mais les processus déterministes ne le quittent pas et sont considérés comme réintroduits dans le système pour être sûrs qu'ils ne sortent pas avant d'avoir utilisé tout leur temps processeur. L'équation (4.19) donne l'approximation :

$$\frac{\partial X_a(t)}{\partial t} = \lambda - C\mu\alpha\frac{X_a(t)}{1+X_a(t)} + C\mu d, \quad \alpha = \frac{(\lambda + C\mu d)(1+X^\infty)}{C\mu X^\infty} \tag{4.19}$$

L'équation (4.14) ne peut pas être utilisée directement, le nombre de processus dans le système $x(t)$ n'est pas connu. Avec l'équation (4.19), le nombre moyen de processus est connu. $x(t)$ ne varie pas beaucoup sur une machine, sa valeur reste proche de sa moyenne. Une approximation de l'équation (4.14) est donnée avec l'équation (4.20).

$$t_c = \int_{t_0}^{t_r} \frac{C}{X(t)} dt \tag{4.20}$$

Il est possible de généraliser au cas de la prédiction du temps d'éxecution de plusieurs processus déterministes insérés à la même date sur la machine. N processus triés par temps de calcul demandé : $t_c^1 \leq t_c^2 \ldots t_c^{N-1} \leq t_c^N$ sont considérés. Chaque processus quittera le système à la date t_r^i et l'équation (4.21) pourra être trouvée (avec $t_r^0 = t_0$).

$$t_c^j = \sum_{i=1}^{j} \left(\int_{t_r^i - 1}^{t_r^i} \frac{C}{X(t)} \right) dt \tag{4.21}$$

4.6 Validation

Cette partie valide le modèle d'accès au processeur et la prédiction du temps d'exécution.

4.6.1 Validation du modèle d'accès au processeur

Pour valider l'équation (4.19) deux tests ont été faits, ils sont expliqués dans les paragraphes 4.6.2 et 4.6.3. Pour les deux tests, les mêmes cas ont été considérés (cas 1 à 6 décrits dans le tableau 4.2) avec différentes charges des machines de 0.12 à 0.6 (groupe 1, groupe 2 et groupe 3). Il est considéré uniquement des taux d'utilisation supérieurs à 0.09 car l'exemple du paragraphe 4.3 a montré qu'en dessous de cette valeur l'influence des processus déterministes était minimale (ceci a été vérifié sur plusieurs exemples). Pour chaque cas d'utilisation des tests ont été faits avec $d < C$ et d $\geq$ C (cas 1 et 2, cas 3 et 4, cas 5 et 6).

La validation de la prédiction du temps d'exécution est présentée au paragraphe 4.6.4.

	Groupe 1		Groupe 2		Groupe 3	
Paramètres	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Cas 6
λ	0.02	0.02	0.02	0.02	0.04	0.04
μ	0.03333	0.03333	0.03333	0.03333	0.03333	0.03333
ρ	0.12	0.12	0.3	0.3	0.6	0.6
d	2	8	1	10	1	10
C	5	5	2	2	2	2

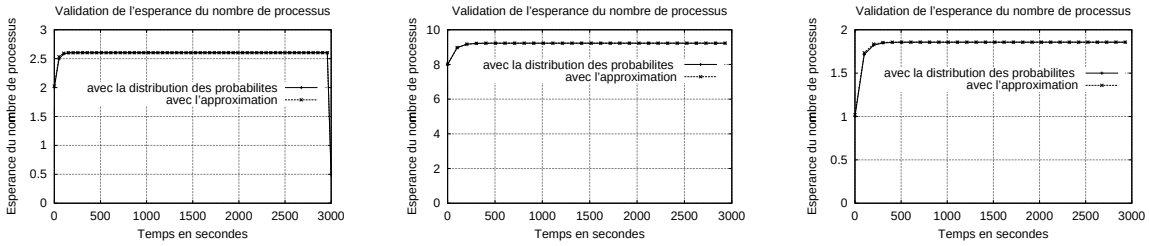
TAB. 4.2: Cas étudiés pour valider l'approximation de la moyenne du nombre de processus

4.6.2 Comparaison de la moyenne du nombre de processus donnée par la distribution des probabilités avec l'approximation.

La moyenne du nombre de processus a été calculée avec la distribution des probabilités tronquée à l'état 100 (équation (4.22)) et comparée avec l'approximation de l'équation (4.19).

$$X(t) = \sum_{i=0}^{100} iP_t(i) \tag{4.22}$$

A la date t=0 les processus déterministes sont déjà dans le système. Les résultats sont décrits sur les figures (4.5, 4.6).



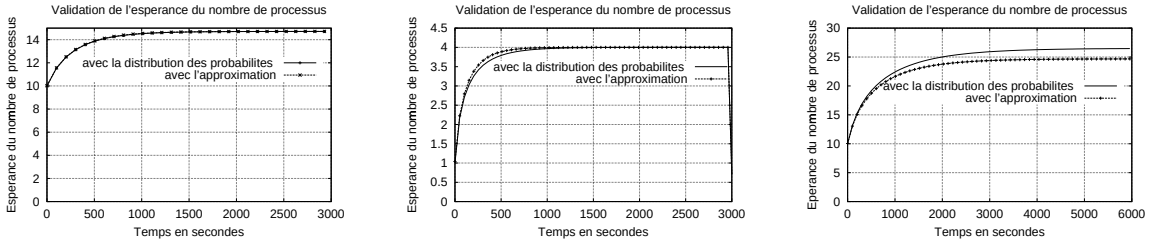
(a) Cas 1 (b) Cas 2 (c) Cas 3

FIG. 4.5: Moyenne du nombre de processus dans le système

Ainsi, l'approximation est bonne dans la plupart des cas. Il y a uniquement une petite différence dans les cas de charge élevée quand il y a beaucoup de processus déterministes insérés.

4.6.3 Comparaison de la moyenne du nombre de processus donnée par la simulation et l'approximation.

Il a été vérifié que l'influence décrite au paragraphe 4.3 avait bien été prédite. Le même simulateur du modèle a été utilisé. L'évolution de la moyenne du nombre de processus dans le système après l'introduction des processus déterministes et la prédiction ont été comparées. La moyenne du nombre de processus a été calculée avec plusieurs simulations (300). A la date t = 1500s un processus déterministe a été introduit. La moyenne du nombre de processus pour

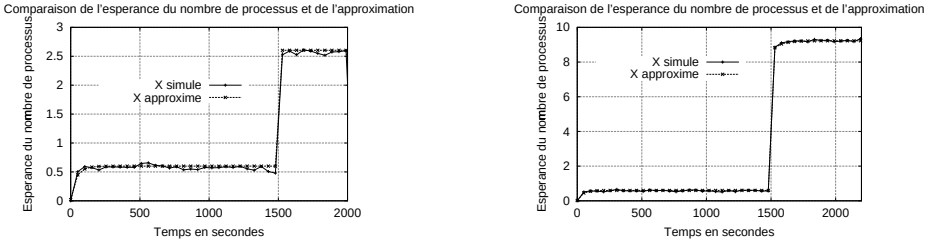


(a) Cas 4 (b) Cas 5 (c) Cas 6

FIG. 4.6: Moyenne du nombre de processus dans le système

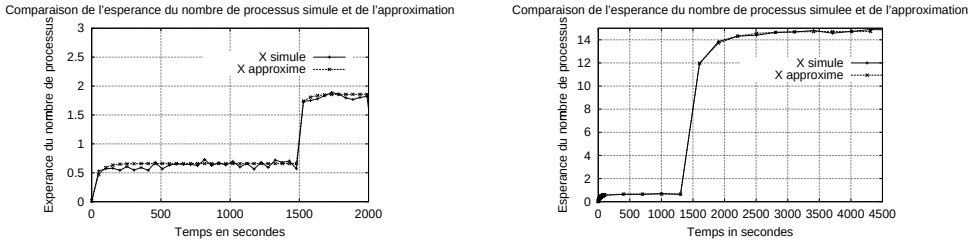
le même taux d'arrivée et le même taux de traitement a été approximée avec l'équation (4.19).

Les résultats sont présentés sur les figures (4.7, 4.8, 4.9).



(a) Cas 1 (b) Cas 2

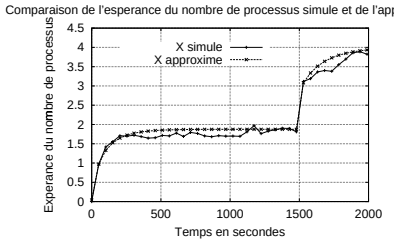
FIG. 4.7: Moyenne du nombre de processus dans le système



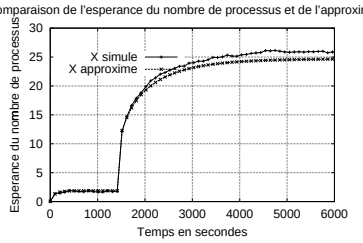
(a) Cas 3 (b) Cas 4

FIG. 4.8: Moyenne du nombre de processus dans le système

L'approximation représente correctement l'évolution de la moyenne du nombre de processus ; pour un taux d'utilisation élevé une petite erreur peut être observée.



(a) Cas 5



(b) Cas 6

FIG. 4.9: Moyenne du nombre de processus dans le système

4.6.4 Validation de la prédiction du temps d'exécution

Des simulations de machines multiprocesseurs avec une politique round robin pour différentes charges ont permis de valider la prédiction du temps d'exécution. Plusieurs tests ont été faits pour différentes charges (0.09, 0.36, 0.63) ; pour chaque charge des tests ont été faits pour $d < C$ et pour d $\geq$ C.

A une date fixe des processus déterministes sont insérés. Les temps d'exécution prédits et mesurés sont comparés. Quand il y a un seul processus déterministe, celui-ci demande entre 20 et 520 secondes de temps de calcul t_c. Quand il y a trois processus insérés, le plus court demande t_c^1 compris entre 20 et 520 secondes, le deuxième processus demande $t_c^2 = 1.5 * t_c^1$ et le processus le plus long demande $t_c^3 = 1.5 * t_c^2$.

L'erreur moyenne pour chaque processus (E^1, E^2 et E^3) et l'écart-type sur l'erreur pour chaque processus (D^1, D^2 et D^3) sont résumés dans le tableau 4.3 et les résultats sont décrits sur les figures (4.10) et (4.11).

Paramètres	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Cas 6
λ	0.006	0.006	0.02	0.02	0.04	0.04
μ	0.03333	0.03333	0.03333	0.03333	0.03333	0.03333
ρ	0.09	0.09	0.36	0.36	0.63	0.63
d	1	3	1	3	1	3
C	2	2	2	2	2	2
t_c^1	20 à 520	20 à 520	20 à 520	20 à 520	20 à 520	20 à 520
t_c^2	–	30 à 780	–	30 à 780	–	30 à 780
t_c^3	–	45 à 1170	–	45 à 1170	–	45 à 1170
E^1	1.6%	3%	6.1%	7.6%	16.5%	13.5%
E^2	–	1.6%	–	1.6%	–	2.8%
E^3	–	1.1%	–	2.5%	–	9.8%
D^1	2.9%	1%	2.3%	2.2%	6.1%	3.7%
D^2	–	2.1%	–	1.9%	–	2.2%
D^3	–	1.6%	–	1.4%	–	2.8%

TAB. 4.3: Les différents cas étudiés pour valider la prédiction du temps d'exécution

Ainsi, c'est pour une charge de la machine petite et moyenne que l'erreur est la plus faible

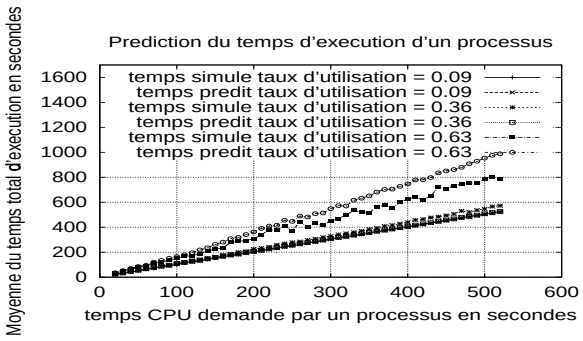
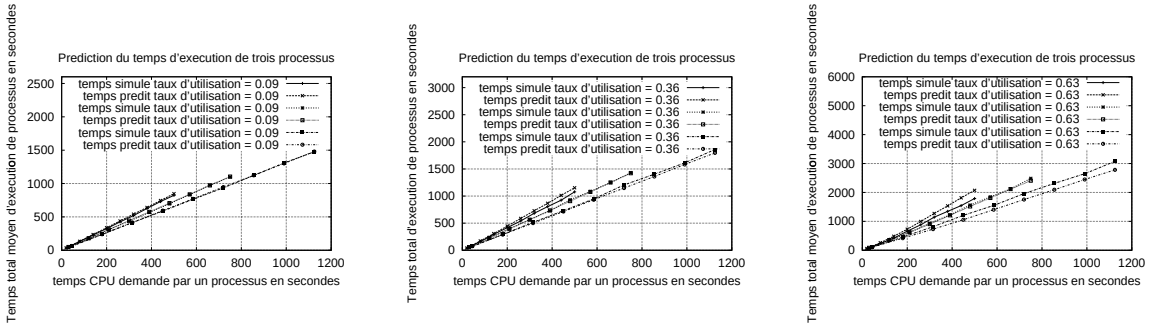


FIG. 4.10: Approximation du temps d'exécution d'un processus (cas 1, 3, 5)



(a) Cas 2 (b) Cas 4 (c) Cas 6

FIG. 4.11: Approximation du temps d'exécution de trois processus (cas 2, 4, 6)

mais de toute manière le placement de tâches est intéressant lorsqu'il y a des machines avec une charge faible ou moyenne, lorsque les machines ont une charge élevée le placement a moins d'influence sur les performances d'une application parallèle.

4.7 Conclusion

L'influence de la charge déterministe sur la charge stochastique a été étudiée. Le phénomène a été traité avec une chaîne de Markov intégrant les processus déterministes. Ensuite la moyenne du nombre de processus et le temps d'exécution ont été prédits. L'originalité de cette approche consiste à calculer les probabilités de transition influencées par l'arrivée des processus déterministes ; cela permet d'étudier l'état stationnaire et transitoire du système. Cette approche peut être généralisée pour différents systèmes de files d'attente. Tout d'abord, les probabilités de transition influencées par les arrivées des processus déterministes doivent être trouvées. Elles sont utilisées pour exprimer la variation de la moyenne du nombre de processus dans le système (théorie du trafic différentiel). Une approximation de cette équation peut ensuite être déduite en utilisant les propriétés en stationnaire (approximation fluide).

Les résultats présentés dans ce chapitre permettent de prédire le temps d'exécution d'applications parallèles régulières en négligeant les communications et en présence de charge aléatoire. L'algorithme de placement du chapitre suivant dans le cas de charge déterministe peut être utilisé directement avec les valeurs prédites ici : la décision de placement peut se faire

avec la prédiction de la date de terminaison présentée ici dans le cas de la charge stochastique.

Chapitre 5

ALGORITHME DE PLACEMENT AVEC QUALITE DE SERVICE ET CHARGE DETERMINISTE

5.1 Introduction

Dans le cadre de l'ASP, il s'agit de placer les applications en respectant le contrat passé avec le client. Le contrat définit les contraintes de l'application (avec deadline, avec date de début imposée, nécessité d'être seul sur la machine lors de l'exécution, contrainte de logiciel...) ainsi que sa priorité. Tout ceci définit des niveaux de qualité de service. Ce problème est encore très peu traité dans la littérature.
Un algorithme de placement mis en oeuvre au niveau domaine d'AROMA et ne tenant pas compte des communications entre les tâches est proposé. En ce qui concerne l'utilisation du processeur, une approche pour fournir de la qualité de service dans le placement d'applications basée sur des classes d'applications est proposée. Tout d'abord le problème du placement avec qualité de service est situé. Puis l'approche utilisée pour l'accès au processeur est explicitée : les différentes classes d'applications et le modèle théorique qui en découle. Enfin, l'implémentation et la validation de l'algorithme sont présentées.

5.2 Le problème du placement avec qualité de service

La figure 5.1 présente le problème de placement.

- **Les modèles d'application :** les modèles d'application sont nécessaires pour obtenir un placement efficace : cela inclut la quantité de ressources demandées (temps processeur, espace mémoire, logiciel spécifique ...), les caractéristiques de l'application (application parallèle ou séquentielle). Si l'application est parallèle, le nombre de tâches ainsi que le modèle de communication sont également nécessaires.
- **Les modèles d'accès aux ressources :** des équations pour la politique d'accès aux ressources sont déduites et utilisées pour prédire le temps d'utilisation de la ressource. Le modèle d'une station multiprocesseur a été expliqué précédemment et dans [45], [102]. Le modèle d'accès au réseau a été présenté et est expliqué dans [104].

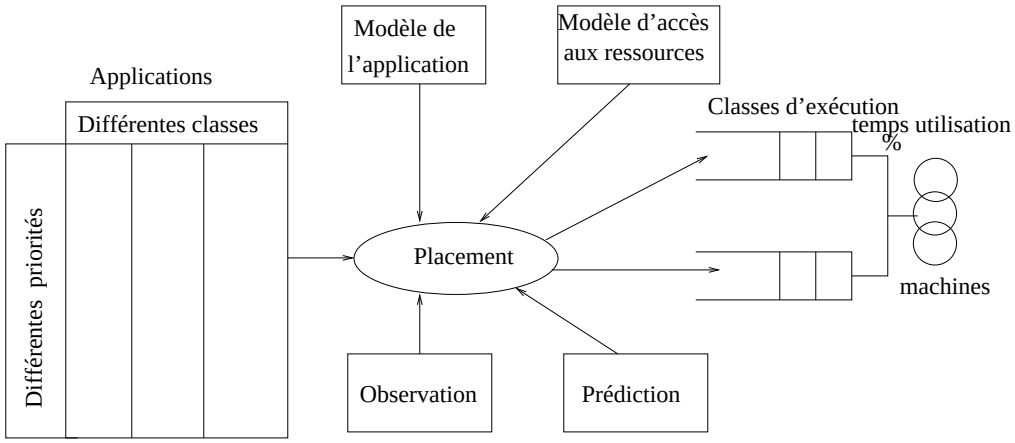


FIG. 5.1: Le problème du placement

- **Les observations :** la charge processeur, réseau, l'espace mémoire disponible et les processus sont observés. D'autres observations peuvent également être ajoutées.
- **Le placement :** à l'instant t, le placement doit respecter le contrat, les contraintes, la décision doit être aussi rapide que possible et ne doit pas introduire de surcharge. Un problème d'ordonnancement doit être résolu avec un critère d'optimisation : la minimisation du temps de complétion ou la minimisation de la facturation (dans le cas où un client reçoit une facture en fonction des ressources utilisées).
  Après la résolution du problème, les applications sont divisées en classes d'exécution (différentes files d'attente). Le paragraphe 5.4 sur l'implémentation détaillera ce point.
  A l'instant t+Δt (après l'arrivée d'une application), le placement doit respecter les contraintes données par le placement à l'instant t. Par exemple, si une application avec une priorité faible est arrêtée, cela introduit une contrainte sur cette application : elle devra être redémarrée sur la même machine. La date de terminaison d'une application déjà placée doit être respectée. Ainsi, une nouvelle demande de placement peut être refusée si le contrat ne peut pas être respecté (application impossible à placer) ou la classe de l'application peut être modifiée au profit d'une application de qualité de service plus faible. Ainsi, un nouveau problème d'ordonnancement doit être résolu avec de nouvelles contraintes.

5.3 La modélisation de l'accès au processeur

5.3.1 Les classes d'applications

Quatre classes d'applications sont définies ; par ordre de priorité [103] :

- classe 1 : les applications avec deadline.
- classe 2 : les applications à exécution immédiate (par définition prioritaires). Une date de terminaison est garantie en fonction de la prédiction calculée lors de la soumission de l'application.
- classe 3 : les applications avec ressources dédiées.

– classe 4 : les applications sans contrainte : elles s'exécutent dès que possible ou ultérieurement avec les ressources disponibles (classe de Best Effort). Les applications irrégulières où aucune prédiction n'est possible se retrouvent ici.

Ces classes d'applications correspondent aux classes de service. Certains niveaux de qualité de service ne sont disponibles que pour certaines catégories d'applications. Dans le cas des applications pour lesquelles un modèle de prédiction du temps d'exécution existe (application régulière par exemple), la date de terminaison peut être garantie. Plus l'ordonnanceur disposera d'informations sur le support d'exécution, le réseau, les applications plus la gestion de la qualité de service sera fine. Il apparaît que la première application de ce type de système est avec un réseau Intranet ou un réseau dédié ce qui permettra de définir une meilleure qualité de service de manière plus simple.
Plusieurs files correspondant chacune à un niveau de priorité (figure 5.2) sont définies. Les files sont considérées dans l'ordre des priorités et les tâches sont affectées aux machines disponibles en tenant compte de ce qui est déjà placé et des contraintes. Pour chaque tâche, la machine attribuée, la date de début, la date estimée de fin d'exécution sont mémorisées de manière à pouvoir agir en cas de préemption ou de panne de la machine.

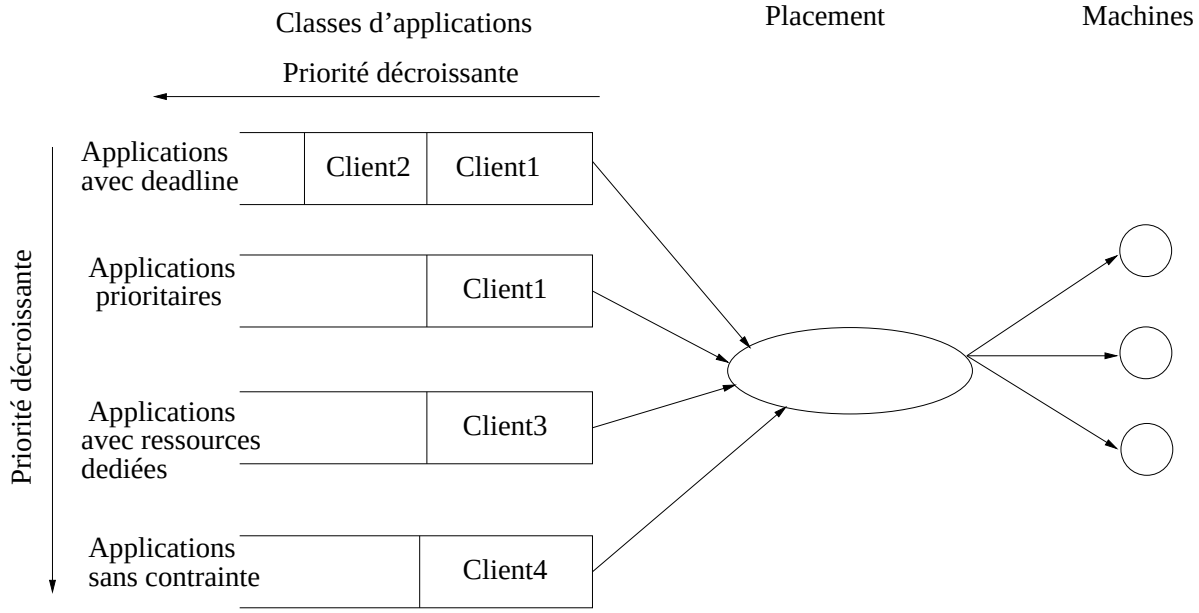


FIG. 5.2: Modèle de files d'attente

5.3.2 Modélisation mathématique du problème

Les notations sont d'abord présentées.

Les variables :

Un événement est un début de tâche ou une terminaison de tâche.

– M : nombre de machines.

– A : nombre d'applications à placer.

- N^a : nombre de tâches d'une application a
- X_m(t) : nombre de processus sur la machine m à l'instant t.
- t_0 : date initiale de recherche de placement.
- $t_b(a, p, m)$: date de début d'exécution de la tâche p de l'application a sur la machine m.
- $t_f(a, p, m)$: date de fin d'exécution de la tâche p de l'application a sur la machine m.
- $t_c(a, p, m)$: temps processeur demandé par la tâche p de l'application a sur la machine m.
- $t_c^k(a, p, m)$: temps de processeur encore demandé par la tâche p de l'application a après le kième événement sur la machine m.
- $t_c(a, p, m) = t_c(a, p)$ * C(m) : équivalence de temps processeur demandé par la tâche p de l'application a en fonction de la machine m. L'utilisateur spécifie un temps $t_c(a, p)$ calculé sur une machine d'une certaine puissance ; la conversion pour déduire le temps nécessaire pour l'exécution sur la machine m est effectuée grâce au coefficient C(m).
- t_m^k : un événement d'arrivée ou de sortie de processus intervient sur la machine m. C'est le kième événement.
- P(a) : ensemble de placements possibles pour une application a.
- $t_f(m, s)$: date à laquelle toutes les tâches prévues sur la machine m sont terminées après le placement s de l'application a avec s$\in$ P(a).
- M(a,p) : machine sur laquelle la tâche p de l'application a est exécutée.
- M_u(m) : mémoire totale utilisée sur la machine m.
- M_t(m) : mémoire totale sur la machine m.
- M_f : constante : coefficient de pondération.

Le critère d'optimisation :

Le problème de placement est un probème d'optimisation, nous devons définir un critère. Plusieurs critères sont connus de la littérature : le "makespan" (minimisation de la date de fin de l'application), "sum-flow" (minimisation de la quantité de ressources utilisées), "max-stretch" (exprime le ralentissement que subit la tâche par rapport à une situation où elle serait seule sur la machine). L'objectif est d'optimiser l'utilisation des ressources du fournisseur de services et de garantir le niveau de qualité de service requis. Ainsi il a été choisi de libérer les ressources le plus rapidement possible. Toutes les applications doivent se terminer le plus tôt possible.

$$\min_{s \in P(a)} (\max_{m} (t_f(m, s) + t_f(m, s) * M_u(m) * M_f / M_t(m))) \tag{5.1}$$

La date de libération des ressources est optimisée ; de plus un second critère consiste à pondérer avec l'espace mémoire utilisé : les machines ayant le plus d'espace mémoire libre sont choisies en premier. En effet, pour certaines tâches l'espace mémoire demandé est inconnu donc le plus simple est de pénaliser les machines ayant le moins d'espace disponible. Le problème est multi-critères (5.1) en utilisant une combinaison linéaire des différents critères. La première partie de l'addition ($t_f(m, s)$) représente la date de libération des ressources de la machine. La

deuxième partie de l'équation $t_f(m,s) * M_u(m) * M_f/M_t(m)$ pénalise les machines qui ont le moins d'espace mémoire libre. $M_u(m)/M_t(m)$ donne une idée de l'utilisation de la mémoire sur la machine. Le coefficient M_f pondère cette partie du critère. La multiplication avec $t_f(m,s)$ positionne les valeurs dans le même ordre de grandeur que la première partie du critère.

L'algorithme de placement :

L'algorithme de placement proposé est expliqué figure 5.3. Un algorithme de listes est utilisé pour les applications, les tâches et les machines pour réduire la combinatoire. Le placement d'une tâche déjà étudiée est revu seulement s'il est impossible de trouver un placement pour l'application a. De plus, pour réduire le temps mis par l'algorithme, la date de libération des machines est sauvegardée et les machines sont triées par date de libération (liste_machines) pour étudier en premier une machine qui donnerait le meilleur placement. Si le placement est possible $t_f(m,s)$ est enregistré, et le placement sur la machine suivante est considéré ; l'étude s'arrête (avec arrêt_itération) dès qu'un t_m^k supérieur ou égal à $t_f(m,s)$ pendant le calcul de $t_f(a,p,m)$ est trouvé.
La qualité de service est toujours garantie : toutes les contraintes induites sont vérifiées par l'algorithme. S'il est impossible de trouver un placement la demande est refusée. Les calculs de $t_b(a,p,m)$ et $t_f(a,p,m)$ dans le cas d'une charge déterministe vont être expliqués.

La date de début de tâche. Calcul de $t_b(a,p,m)$:

Le $t_b(a,p,m)$ est calculé de manière itérative (il est tout d'abord fixé à t_0 et si aucun placement n'est trouvé, une autre date de début est cherchée).

$t_b^0(a,p,m) = t_0$

$t_b^{k+1}(a,p,m) = \min\ t_f(i,j,m)$

avec i $\in [1, a-1]$, j $\in \left[1, N^i\right]$ et $t_f(i,j,m) > t_b^k(a,p,m)$ (seules les tâches qui peuvent se terminer après le dernier t_b étudié sont considérées). Le principe est de rechercher une nouvelle date de début quand le système sera moins chargé, c'est à dire dès qu'un processus se termine.

La date de terminaison de tâche. Calcul de $t_f(a,p,m)$:

Le $t_f(a,p,m)$ est calculé de manière itérative. Pour cela, les dates successives où il y a soit création de processus soit terminaison de processus sont calculées. t_m^0 correspond à l'arrivée du premier processus.

pour toutes les tâches (a allant de 1 à A et p de 1 à N^a, sur m)
avec $t_b(a,p,m) > t_m^k$ et $\delta_m^{M(a,p)}$=1 ($\delta_m^{M(a,p)}$ est le symbole de Kronecker)
Si $X_m(t_m^k) > C$
$t_m^{k+1} = \min_{a,p}\ (t_b(a,p,m), t_c^k(a,p,m) * X_m(t_m^k) + t_m^k)$
sinon
$t_m^{k+1} = \min_{a,p}\ (t_b(a,p,m), t_c^k(a,p,m) + t_m^k)$
Si $t_b(a,p,m) > t_m^k$ et si $X_m(t_m^k) > C$
$t_c^{k+1}(a,p,m) = t_c^k(a,p,m) - C*(t_m^{k+1} - t_m^k)/X_m(t_m^k)$
(c'est donné par la politique de temps partagé : c'est le temps processeur restant)
Si $t_b(a,p,m) = t_m^i$
$t_c^i(a,p,m) = t_c(a,p,m)$

for application a = 1 à A applications classées par priorité **do**
  critère = -1
  while placement optimal non trouvé **do**
    if rejet=calcul de la date de début de l'application $t_b(a,p,m)$ **then**
      exit
    end if
    vérification des constraintes concernant la date de début ; tâche=0
    while tâche< N^a triée par précédence ou par temps processeur demandé ET placement possible **do**
      numéro_machine=0 ; machine_choisie=-1
      while numéro_machine<taille(liste_machines) ET arrêt_itération==faux **do**
        machine=liste_machines[numéro_machine]
        vérification des constraintes concernant les machines
        arrêt_itération = prédiction de la date de fin de la tâche $t_f(a,p,m)$;calcul de $M_u(m)$
        if arrêt_itération==faux **then**
          vérification des constraintes des tâches déjà placées sur la machine
          vérification de la deadline de la tâche ; placement possible sur machine
        end if
        if (machine_choisie ==-1) OU ($t_f(m,S)$+$t_f(m,S)$*$M_u(m) * M_f/M_t(m)$)< *critère* **then**
          machine_choisie = machine ; sauvegarde de la nouvelle valeur du critère
        end if
        numéro_machine++
      end while
      if placement possible **then**
        propagation des conséquences du placement
      end if
    end while
    if placement possible **then**
      sauvegarde de $t_b(a,p,m)$ et $t_f(a,p,m)$ pour toutes les tâches
      modification de l'ordre de la liste liste_machines ; placement optimal trouvé
    else
      placement optimal non trouvé
    end if
  end while
end for

FIG. 5.3: L'algorithme de placement proposé

(c'est le cas lorsqu'un nouveau processus est inséré dans la récurrence)

$X_m(t_m^0) = X_m(t_0)$=nombre de processus à la date initiale

Si t_m^{k+1}correspond à D terminaisons de processus et à A débuts
$X_m(t_m^{k+1}) = X_m(t_m^k)$+A-D

Si $t_c^{k+1}(a,p,m) = 0$, $t_f(a,p,m) = t_m^{k+1}$ la date de terminaison de la tâche a été trouvé.

Les itérations continues jusqu'à ce que toutes les tâches sur la machine soient terminées ceci détermine : $t_f(m)$.

La figure 5.4 illustre les événements ayant lieu au cours du temps : début de tâches, terminaison de tâches qui correspondent aux t_m^k.

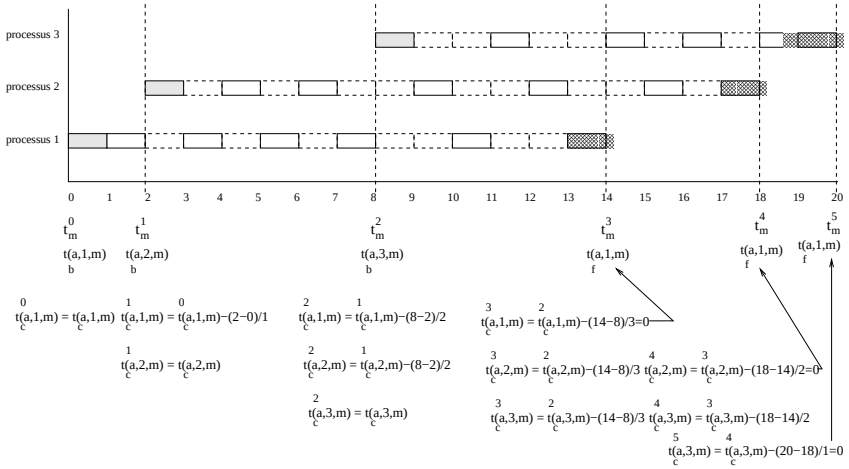


FIG. 5.4: Les différents t_m^k

Lorsque le processus 1 arrive, il est seul et peut donc obtenir tout le temps processeur disponible jusqu'à t=2 correspondant à l'arrivée du processus 2. De t=2 à t=8 il y a deux processus dans le système ils se partagent le processeur. De t=8 à t=14, il y a trois processus. A t=14, le processus 1 a obtenu tout le temps processeur qu'il demandait. Il quitte le système, il ne reste alors plus que les processus 2 et 3. Le processus 2 termine à t=18 et le processus 3 à t=20.

5.4 Processeurs virtuels

Deux possibilités d'implémentation ont été considérées toutes deux étant basées sur les classes de service. Soit les applications à placer sont considérées dans l'ordre d'importance des classes de service avec un risque de famine des applications "best effort". Soit il est considéré que chaque classe de service a son processeur virtuel et que seule cette classe de service s'exécute sur ce processeur (on parlera de classe virtuelle). L'importance entre les processeurs virtuels est donné par des pourcentages (exemple 60% classe 1, 30% classe 2, et 10% classe 4 ; dès qu'un processus appartient à la classe 3 il a 100% du processeur réel). La première implémentation étant triviale, seule l'implémentation basée sur les processeurs virtuels sera détaillée. Plus particulièrement la mise en oeuvre sur un noyau Linux sera décrite.

5.4.1 Modèle logique

La puissance de calcul d'une machine générée par son ou ses processeurs est décomposée en processeurs virtuels. Un coefficient de 1 correspond à la puissance totale de la machine. Les processeurs virtuels correspondent à une sous partie de cette puissance et constituent une partition. Soit P le nombre de processeurs virtuels créés et P_i^v la puissance du processeur virtuel i :

$P_i^v \leq 1$, pour 1≤i≤P

$\sum_{i=1}^{P} P_i^v = 1$

La figure 5.5 illustre le modèle logique utilisé ainsi que la notion de processeur virtuel.

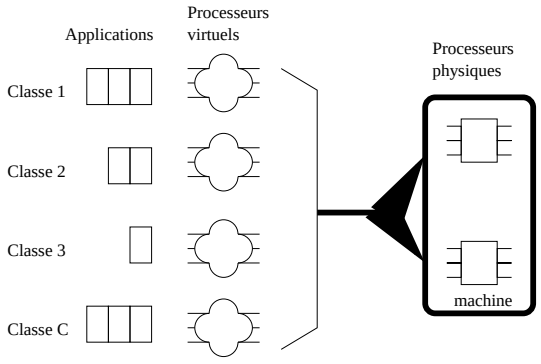


FIG. 5.5: La notion de processeur virtuel

La notion de processeur virtuel est liée à la notion de classe d'applications. Chaque classe utilise un et un seul processeur virtuel.

Ainsi l'ordonnanceur connaît la répartition des puissances dans chaque file et l'utilise pour calculer le temps processeur consommé par chaque processus de chaque file. Les notations suivantes sont introduites :

- $t_c^c(a,p,m)$: temps processeur consommé par la tâche p de l'application a sur la machine m.
- X_m^i : nombre de processus sur la machine m dans la file i.
- $T(a,p,m)$: temps total d'exécution de la tâche p de l'application a sur la machine m.

Soit un processus d'une file i, les équations 5.2 et 5.3 sont connues .

$$t_c^c(a,p,m) = t_c^k(a,p,m) - \frac{C(t_m^{k+1} - t_m^k)}{X_m^i} * P_i^v \tag{5.2}$$

$$t_c(a,p,m) = T(a,p,m) * P_i^v \tag{5.3}$$

Lorsqu'une file ou plusieurs files sont vides les puissances des autres files sont recalculées pour bénéficier de la puissance non utilisée tout en respectant les ratio initiaux. Appelons i, j et k les trois files.
Si deux files sont vides, elles ont une puissance à 0 % et la troisième a une puissance à 100 %.

Si une file est vide par exemple i, elle a une puissance à 0 % et les puissances des files j et k sont calculées grâce aux équations 5.4, 5.5 et 5.6.

$$a = \frac{P_j^v}{P_k^v} \tag{5.4}$$

$$P_j^v = \frac{a}{1+a} \tag{5.5}$$

$$P_k^v = 1 - P_j^v \tag{5.6}$$

5.4.2 Modèle réel

L'accès au(x) processeur(s) sur une machine réelle se fait à travers l'ordonnanceur ("scheduler") du système d'exploitation. Ce dernier est responsable du choix d'un processus pour son exécution pendant un quantum de temps. Le temps partagé implique le passage de tous les processus sur une machine au bout d'un certain temps. La figure 5.6 présente l'ordonnancement des processus sur une machine.

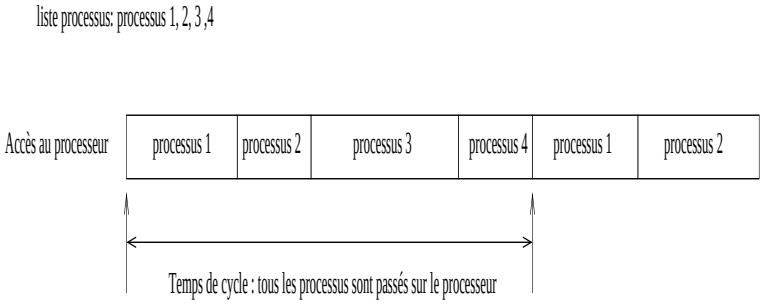


FIG. 5.6: Ordonnancement

A partir de ce constat la notion de cycle est définie. Si il est supposé que les processus restent un certain temps dans le système, le temps d'un cycle Q est l'intervalle de temps nécessaire pour que tous les processus présents à l'instant t (il est supposé qu'il n'y a pas d'arrivée de nouveaux processus) aient eu au moins une fois le temps de consommer leur quantum de temps. Le cycle se termine à la date t+Q.

Soit C le nombre de classes d'applications, J le nombre de processus en exécution, J_i le nombre de processus en exécution de la ième classe et J_i^k le kième processus en exécution de la classe i.

L'application de la notion de processeurs virtuels sur l'architecture réelle va se faire en utilisant les propriétés de l'ordonnanceur du système d'exploitation.

5.4.3 Calcul des priorités sous Linux

Sous linux, l'utilisation de la priorité statique va permettre de mettre en place les classes de services c'est à dire garantir que chaque file n'utilisera pas plus de puissance du processeur réel que celle qui est autorisée.

Les deux propriétés suivantes vont être utilisées :

- le calcul du quantum alloué à un processus dépend de sa priorité statique. Il ne dépend pas de son passé
- tant que tous les processus en attente de processeur n'ont pas consommé leur quantum un processus qui a déjà consommé son quantum ne peut reprendre le processeur.

Un quantum de temps unique sera alloué à chaque classe. Tous les processus d'une même classe auront donc la même priorité statique notée S_i (avec i numéros de classe). Il est noté $Q(S_i)$ le quantum alloué à la classe i et correspondant à la priorité S_i
Q est le quantum total affecté à toutes nos tâches (cela correspond au temps d'un cycle), il se calcule :

$$Q = \sum_{i=1}^{4} Q(S_i) J_i \tag{5.7}$$

Afin de satisfaire la répartition de la puissance entre les différentes classes il doit être trouvé :

$$P_i = \frac{Q(S_i) J_i}{Q} \tag{5.8}$$

avec $-20 \leqslant S_i \leqslant 19$

Le quantum de temps $Q(S_i)$ attribué à chaque processus s'exprime de manière linéaire en fonction de la priorité :

$$S_i : Q(S_i) = c - dS_i \tag{5.9}$$

Pour le scheduler 2.4 : c=11 , d=$\frac{1}{4}$ et pour la version 2.6 : c=$\frac{400}{39}$, d=$\frac{19}{39}$.

Pour résoudre ce problème trois classes uniquement sont considérées car les applications avec ressources dédiées ne sont pas concernées (en effet dès qu'un processus est dans la file des ressources dédiées il a la puissance totale de la machine). Le problème est posé :
A partir de (5.8) :

$$Q(S_i).J_i = Q.P_i \tag{5.10}$$

On inclut (5.7) et (5.9) :

$$(c - d.S_i).J_i = P_i.\sum_{k=1}^{3} Q(S_k).J_k \tag{5.11}$$

$$c.J_i - d.S_i.J_i = P_i.\sum_{k=1}^{3} (c - d.S_k).J_k \tag{5.12}$$

$$c.J_i - d.S_i.J_i = P_i.\sum_{k=1}^{3} c.J_k - d.S_k.J_k \tag{5.13}$$

$$-d.S_i.J_i + P_i.\sum_{k=1}^{3} d.S_k.J_k = -c.J_i + P_i.\sum_{k=1}^{3} c.J_k \tag{5.14}$$

A partir de (5.14) pour :
i=1 :

$$-d.S_1.J_1 + P_1.(d.S_1.J_1 + d.S_2.J_2 + d.S_3.J_3) = -c.J_1 + P_1.(c.J_1 + c.J_2 + c.J_3) \tag{5.15}$$

i=2 :

$$-d.S_2.J_2 + P_2.(d.S_1.J_1 + d.S_2.J_2 + d.S_3.J_3) = -c.J_2 + P_2.(c.J_1 + c.J_2 + c.J_3) \tag{5.16}$$

i=3 :

$$-d.S_3.J_3 + P_3.(d.S_1.J_1 + d.S_2.J_2 + d.S_3.J_3) = -c.J_3 + P_3.(c.J_1 + c.J_2 + c.J_3) \tag{5.17}$$

D'où :

$$\begin{cases} d.J_1.(P_1-1).S_1 + d.J_2.P_1.S_2 + d.J_3.P_1.S_3 = c.J_1.(P_1-1) + c.J_2.P_1 + c.J_3.P_1 \\ d.J_1.P_2.S_1 + d.J_2.(P_2-1).S_2 + d.J_3.P_2.S_3 = c.J_1.P_2 + c.J_2.(P_2-1) + c.J_3.P_2 \\ d.J_1.P_3.S_1 + d.J_2.P_3.S_2 + d.J_3.(P_3-1).S_3 = c.J_1.P_3 + c.J_2.P_3 + c.J_3.(P_3-1) \end{cases}$$

Le système d'équation suivant est obtenu :

$$\begin{cases} a_{11}S_1 + a_{12}S_2 + a_{13}S_3 = b_{11} + b_{12} + b_{13} \\ a_{21}S_1 + a_{22}S_2 + a_{23}S_3 = b_{21} + b_{22} + b_{23} \\ a_{31}S_1 + a_{32}S_2 + a_{33}S_3 = b_{31} + b_{32} + b_{33} \end{cases}$$

avec $a_{ij} = d.P_i.J_j$ pour $i \neq j$, $a_{ii} = d.(P_i - 1).J_i$
et $b_{ij} = c.P_i.J_j$ pour $i \neq j$, $a_{ii} = c.(P_i - 1).J_i$

Cela revient à trouver une solution approchée ou exacte à l'équation matricielle du type $AS = B$, tout en prenant en compte le fait que les priorités S_i doivent être des valeurs entières. Un problème de programmation linéaire en nombres entiers doit être résolu.

Pour résoudre ce système d'équation, le logiciel Xpress est utilisé. Ce logiciel permet de résoudre les problèmes de programmation linéaire. La résolution se décompose en deux grandes phases : la modélisation du problème (définition des équations, contraintes) puis la résolution en minimisant ou maximisant une fonction objectif définie.

L'équation matricielle est modifiée en faisant intervenir un vecteur de résidus, il est donc

obtenu : AS+Y=B. Ce vecteur va permettre de définir la fonction objectif.

La résolution est optimisée en minimisant la somme des résidus. On obtient un programme linéaire en variables mixtes.

$$min \sum_{i=1}^{3} Y_i$$

sous

$$AS + Y = B \tag{5.18}$$

$$-20 \leqslant S_i \leqslant 19 \tag{5.19}$$

S_i entier et Y_i réel avec

$$1 \leqslant i \leqslant 3 \tag{5.20}$$

Ordonnanceur Linux 2.4

Le quantum de temps de la version 2.4 du noyau Linux correspond à une variable "counter" pour un processus. Ce "counter" est le nombre de tops d'horloge affectés à un processus, puis décrémentés par un chronomètre.

Il est calculé à partir de la commande *nice* tel que : $Q = 11 - \frac{nice}{4}$. Donc $c = 11$ et $d = \frac{1}{4}$.

Plusieurs résolutions de ces équations via Xpress sont lancées en faisant varier les puissances souhaitées des deux premières classes entre 10% et 80% (la troisième puissance est déduite par soustraction) afin de ne laisser aucune classe à 0%.
Une classe ayant une puissance nulle ne possèdera jamais de temps processeur et ce n'est pas ce qui est recherché. Puis la somme des erreurs sur les trois classes entre les puissances souhaitées des processeurs virtuels et les puissances obtenues à partir des résultats de priorités est calculée.

$$Erreur = \frac{|P_1 - P'_1| + |P_2 - P'_2| + |P_3 - P'_3|}{P_1 + P_2 + P_3}$$

Avec P_1,P_2 et P_3 puissances souhaitées et P'_1,P'_2 et P'_3 puissances obtenues.

Dans un premier temps les valeurs *nice* sont comprises entre -20 et 19 puis elles sont limitées à l'intervalle 0 et 19 (valeur fixable en simple utilisateur).

La figure 5.7 représente l'erreur entre Xpress et les puissances souhaitées avec nice compris entre -20 et 19 dans le cas de Linux 2.4. Des pourcentages d'erreurs assez élevés (entre 50 et 60%) sont obtenus dans certains cas particuliers mais en moyenne l'erreur se situe autour de 15%. Ces résultats ne sont pas surprenants puisque les contraintes sur les priorités sont assez restrictives et elles créent des erreurs (voir figure 5.7).

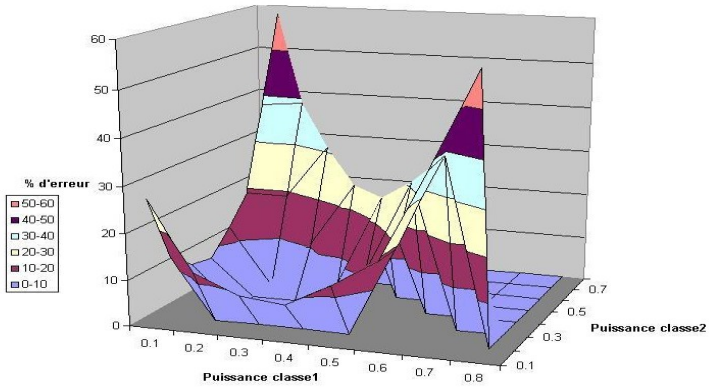


FIG. 5.7: Erreur entre Xpress et les puissances souhaitées (Linux 2.4, nice : -20..19)

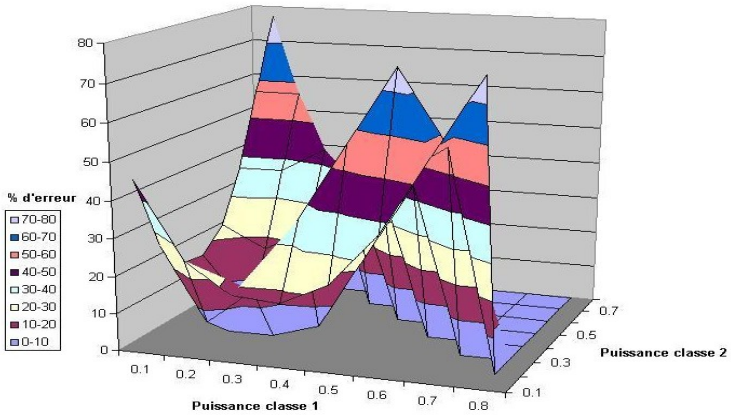


FIG. 5.8: Erreur entre Xpress et les puissances souhaitées (Linux 2.4, nice : 0..19)

La figure 5.8 représente l'erreur entre Xpress et les puissances souhaitées avec nice compris entre 0 et 19 dans le cas de Linux 2.4. Les pourcentages d'erreurs sont plus élevés sur la figure 5.8 , ceci est logique puisque le problème est à nouveau contraint en ôtant les valeurs des priorités administrables uniquement par un super utilisateur.

Scheduler Linux 2.6

Les mêmes tests que précédemment sont exécutés en comparant les puissances souhaitées et celles obtenues.

Les résultats avec le quantum du scheduler Linux 2.6 avec nice entre -20 et 19 (figure 5.9) sont meilleurs que ceux réalisés précédemment. L'erreur n'atteint jamais plus de 8% et la moyenne se situe autour de 4%.

Les résultats, même en contraignant la valeur nice compris entre 0 et 19 de Linux 2.6 (voir

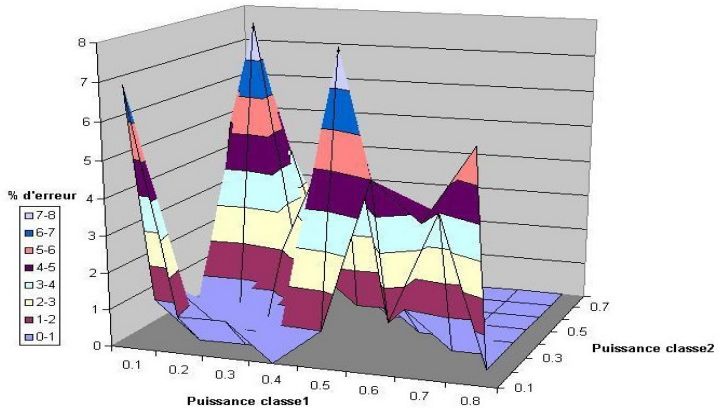


FIG. 5.9: Erreur entre Xpress et les puissances souhaitées (Linux 2.6, nice : -20..19)

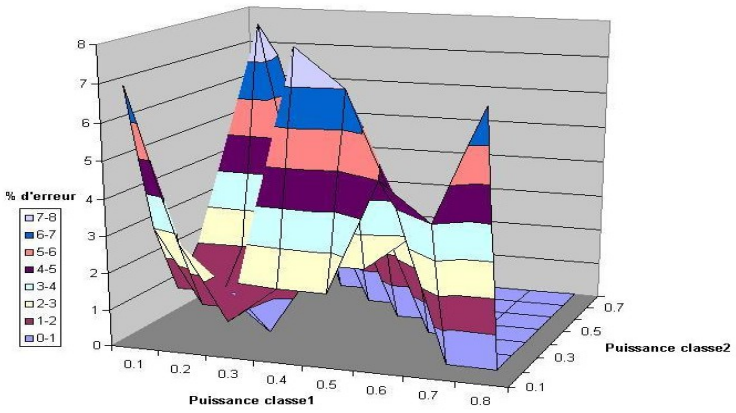


FIG. 5.10: Erreur entre Xpress et les puissances souhaitées (Linux 2.6, nice : 0..19)

figure 5.10), restent très bons (en moyenne 5% d'erreur sur les puissances est obtenu).

L'énorme différence entre les résultats de l'ordonnanceur 2.4 et 2.6 peut être expliquée en comparant les valeurs extrêmes du quantum de temps alloué à partir de la commande nice.

Le scheduler Linux 2.4 n'a pas une tranche de valeur suffisamment élargie contrairement à la version 2.6 (voir Tab 5.1), c'est pour cela que le pourcentage d'erreur sur les puissances est fort. "Counter" et "timeslice" représentant un nombre d'unités de temps attribués aux processus.

	Valeur Nice		
Version	-20	0	19
Scheduler 2.4 (counter)	16	11	6.25
Scheduler 2.6 (timeslice)	20	10.26	1

TAB. 5.1: Différence de quantum selon la valeur nice et l'ordonnanceur Linux

Implémentation et heuristique du calcul des priorités (valeur nice comprise entre 0 et 19)

Après avoir calculé les erreurs entre les puissances Xpress et celles souhaitées, il a fallu implémenter un algorithme pour calculer ces priorités.
La possibilité de coder une méthode de Branch & Bound avec Simplex et Dual Simplex (résolution pratiquée par le logiciel Xpress) a paru trop complexe.

Un algorithme plus simple, prenant comme argument d'entrée le nombre de processus par classe (J1, J2, J3) ainsi que les puissances allouées (P1, P2, P3), a été implémenté.
Pour chaque classe, dans un premier temps le rapport entre la puissance et le nombre de ses processus est calculé.
La classe ayant le rapport le plus fort possèdera le quantum de temps le plus grand (valeur correspondante à une priorité nice 0). Ensuite une règle de trois est effectuée pour calculer les deux autres quantums en fonction des rapports de classe.

Si ces deux quantums sont inférieurs à la valeur minimale pouvant être allouée par l'ordonnanceur (source d'apparition d'erreur) alors le quantum de temps minimal est affecté à la classe ayant le rapport le plus faible.

Le dernier quantum est ensuite calculé en le faisant "tendre" vers les quantums extrèmes possibles (quantum minimal et maximal). L'erreur sur les puissances souhaitées dans ces cas est calculée et celle qui est minimale est conservée.

Exemple (Scheduler Linux 2.4) :

Exemple1

Données d'entrée : $J_1 = 1$; $J_2 = 1$; $J_3 = 3$; $P_1 = 0.3$; $P_2 = 0.1$; $P_3 = 0.6$
Calcul des rapports :

$$R_1 = \frac{P_1}{J_1} = 0.3$$
$$R_2 = 0.1$$
$$R_3 = 0.2$$

La classe 1 sera la plus prioritaire avec un quantum de temps $Q_1 = 11$(priorité $S_1 = 0$).

$Q_2 = \frac{R_2 * Q_1}{R_1} = 3.6667$ et $Q_3 = 7.3333$ donc $S_2 = 19$ et $S_3 = 14$.

La solution trouvée par Xpress est : $S_1 = 0$; $S_2 = 19$; $S_3 = 10$, erreur sur les puissances : 6% ; il s'agit de l'erreur entre ce que calcule XPress et l'heuristique.

Exemple2

Données d'entrée : $J_1 = 1$; $J_2 = 3$; $J_3 = 3$; $P_1 = 0.3$; $P_2 = 0.5$; $P_3 = 0.2$
Calcul des rapports :

$$R_1 = \frac{P_1}{J_1} = 0.3$$
$$R_2 = 0.1667$$
$$R_3 = 0.0667$$

La classe 1 sera la plus prioritaire avec un quantum de temps $Q_1 = 11$(priorité $S_1 = 0$).

$Q_2 = \frac{R_2 * Q_1}{R_1} = 6.1123$ et $Q_3 = 2.445$ donc $S_3 = 19$.
Pour S2 :

- si Q_2 est fixé autour de 6.25 l'erreur sur les puissances est de 23%
- si Q_2 est fixé autour de 11 l'erreur sur les puissances est de 4% donc $S_2 = 0$.

La solution trouvée par Xpress est : $S_1 = 0$; $S_2 = 19$; $S_3 = 10$, erreur sur les puissances : 4% ; il s'agit de l'erreur entre ce que calcule XPress et l'heuristique.

Les mêmes tests que précédemment sont réalisés et comparés aux puissances trouvées par Xpress.

Les valeurs nice sont restreintes à l'intervalle 0 et 19 puisque ce sont les seules valeurs qu'un simple utilisateur peut affecter à une application.

Le calcul de l'erreur se fait entre les puissances calculées avec Xpress et les puissances trouvées par l'heuristique.

$$Erreur = \frac{|P_1 - P_1'| + |P_2 - P_2'| + |P_3 - P_3'|}{P_1 + P_2 + P_3}$$

Le pourcentage d'erreur sur les puissances est assez faible (entre 0 et 10%). Ceci montre que l'heuristique bien que moins précise que Xpress apporte des résultats satisfaisants sur les calculs de priorités pour l'ordonnanceur de Linux 2.4 (figure 5.11).

Les résultats d'erreur sur les puissances du scheduler 2.6 sont sensiblement équivalents à ceux de la version Linux 2.4 (figure 5.12).

5.5 Validation du placement avec qualité de service avec et sans processeur virtuel

Pour valider l'algorithme, un simulateur a été développé. Il fait des simulations de Monte-Carlo ; à partir d'un taux d'arrivée noté λ, d'un taux de traitement μ, il génère les applications et simule leur exécution sur une machine multiprocesseur avec C serveurs selon une politique

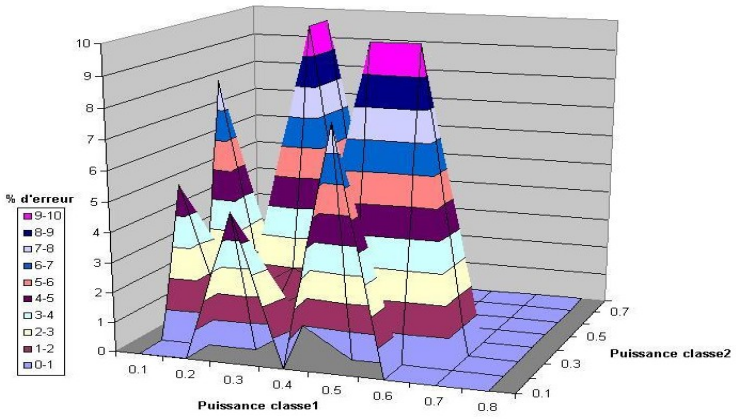


FIG. 5.11: Erreur entre l'heuristique et les puissances calculées avec Xpress (Linux 2.4)

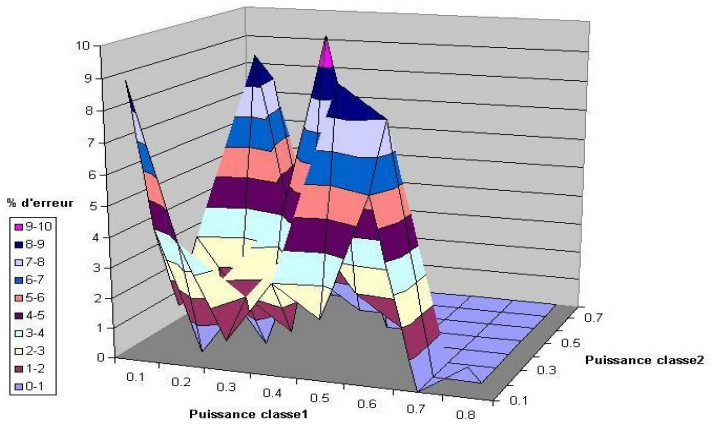


FIG. 5.12: Erreur entre l'heuristique et les puissances calculées avec Xpress (Linux 2.6)

de round-robin sans priorité. Le simulateur peut aussi reproduire des exécutions d'applications à partir de fichiers de logs.

Il est difficile de comparer l'algorithme à d'autres car peu proposent de garantir une qualité de service au processus.

5.5.1 Comparaison du nombre moyen de processus théorique et expérimental.

Une machine SMP avec C processeurs (dans le test C = 12) a été simulée. Toutes les tâches sont indépendantes : il n'y a pas d'applications parallèles et toutes appartiennent à la classe d'applications "best effort". Le nombre moyen de processus dans le système expérimental a été comparé avec celui donné par la formule du "round-robin" avec C serveurs (théorique). Le nombre moyen de processus est noté X. Les arrivées des tâches sont simulées selon une loi de Poisson de taux λ et demandent un temps processeur qui suit une loi exponentielle de taux μ. Le taux d'utilisation ρ également appelé charge est calculé comme dans le modèle M/M/C (en effet, les valeurs en stationnaire sont les mêmes que pour le modèle round-robin) : $\rho =$

$\lambda/(C*\mu)$. 10000 événements sont simulés pour différents λ et μ (donc pour différentes charges de la machine). Pour chaque charge, le nombre moyen de processus est calculé. La figure 5.13 montre la comparaison.

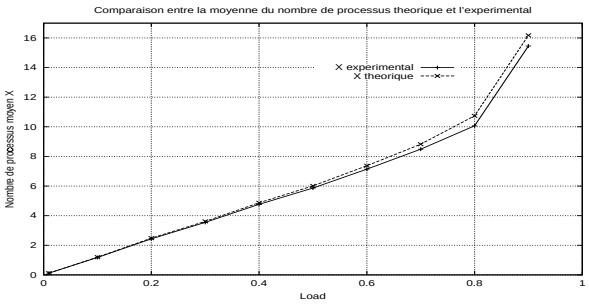


FIG. 5.13: Comparaison entre X expérimental et théorique

Les courbes sont quasiment les mêmes. Ainsi l'algorithme n'introduit pas de perturbation. Lorsque rien n'est connu sur une tâche, le placement donnera des résultats similaires au modèle "round-robin" sans priorité. L'algorithme utilisé ne perturbe pas le modèle "round-robin".

5.5.2 Evolution du temps de placement avec la charge

La même simulation que 5.5.1 est faite pour évaluer le temps mis par l'algorithme de placement. Toutes les tâches appartiennent à la classe "best effort". La figure 5.14 donne l'évolution du temps de placement. Il augmente avec la charge et seulement pour des charges supérieures à 0.8 il est supérieur à 1ms en moyenne.

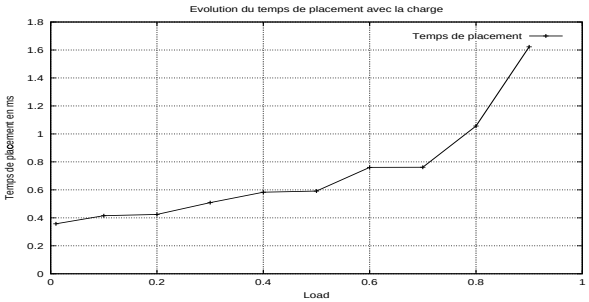


FIG. 5.14: Evolution du temps de placement avec la charge

Ainsi l'algorithme peut être utilisé en temps réel car le temps de réponse est correct.

5.5.3 Comparaison avec NQS

Les fichiers logs de Feitelson (logs réels) ont été utilisés. Ils fournissent la date de soumission de l'application, le temps processeur demandé, le nombre de tâches. Le log (l_sdsc_sp2.swf [105][106], [107],) est utilisé. Le système réel a 128 noeuds et est ordonnancé avec NQS (l'algorithme NQS ne sera pas ici simulé mais l'utilisation d'applications de classe ressources dédiées permet d'avoir une politique FIFO comme NQS). L'exécution des applications a été reproduite, l'algorithme recherche un placement sur 128 noeuds et simule leur exécution. Le temps

d'attente fourni dans les fichiers logs (temps entre la soumission et le début de l'exécution) est comparé à celui trouvé avec l'algorithme.

10000 et 35000 applications ont été simulées, toutes appartiennent à la classe d'application "ressources dédiées". Cela veut donc dire que sur la machine il n'y a plus de temps partagé mais une politique FIFO puisque chaque tâche en exécution sur une machine doit être seule. Toutes les tâches d'une application sont synchronisées c'est à dire démarrent à la même date.

La simulation des 10000 applications est équivalente à une activité du supercalculteur pendant 112 jours. La simulation des 35000 applications correspond à une activité de 249 jours. Les résultats montrent que l'algorithme proposé a un meilleur temps d'attente que NQS. Cela est surtout dû au fait qu'avec notre algorithme une application peut être placée entre deux autres car le temps processeur demandé est connu et la date de fin de chaque tâche placée est prédite. Cela réduit donc significativement les temps d'attente. NQS trie les applications dans des files (en fonction du temps processeur demandé) et mélange les applications lorsqu'un placement est recherché. Cela peut donc augmenter le temps d'attente des tâches courtes qui sont ralenties par des tâches longues. Ce problème est évité avec l'algorithme proposé car les applications sont traitées dans l'ordre de soumission et dans l'ordre des classes (ici toutes les tâches appartiennent à la même classe). Les résultats sont présentés dans le tableau 5.2. Les temps sont en secondes (s).

	10000 événements	35000 événements
temps moyen d'attente avec NQS(s)	10796.74	8979.86
temps moyen d'attente avec l'algorithme proposé(s)	4008.5	4202.09

TAB. 5.2: Comparaison entre le temps d'attente de NQS et celui de l'algorithme proposé

5.5.4 Influence de la qualité de service

L'influence de la qualité de service sur le temps de placement et le temps d'attente a été analysée. Le même fichier log que précédemment a été utilisé : quatre files sont spécifiées : low, normal, high, express et pour chaque application sa file de soumission est connue. Cette information a été utilisée pour faire la correspondance avec nos classes d'applications. Ainsi, la file "low" correspond à la classe "best effort" (classe 4), la file "normal" correspond à la classe "deadline" (classe 1), la file "high" correspond à la classe "ressources dédiées" (classe 3) et la file "express" correspond à la classe d'applications prioritaires (classe 2). Pour les applications de classe 1 leur deadline est : date de soumission + 5*temps cpu demandé. Pour les applications de classe 2 la date de début devant être respectée est : date de soumission + 5 secondes. Les deux possibilités d'implémentation (avec ou sans classe virtuelle) sont considérées. Les résultats présentés avec les classes virtuelles ont été obtenus lorsque les applications avec deadline avaient 70 % du processeur réel, les applications à départ immédiat avaient 20 % du processeur réel et les applications best effort 10 % du processeur réel.

Six cas ont été simulés :

- cas 1 : 10000 événements de toutes les classes sans classe virtuelle.
- cas 2 : 10000 événements de toutes les classes avec classe virtuelle.
- cas 3 : 10000 événements, tous appartiennent à la classe "best effort".
- cas 4 : 35000 événements de toutes les classes sans classe virtuelle.

- cas 5 : 35000 événements de toutes les classes avec classe virtuelle.
- cas 6 : 35000 événements, tous appartiennent à la classe "best effort".

Pour tous les cas, le temps d'attente pour chaque classe, le temps moyen d'attente, le temps de placement pour chaque classe et le temps de placement moyen ont été calculés.

Ainsi, les cas 1, 2 et 3 peuvent être comparés puis 4, 5 et 6 pour voir l'influence de la qualité de service et la différence entre les deux implémentations (sans classe virtuelle ou avec). Les résultats sont présentés dans le tableau 5.3. Les temps d'attente sont en secondes (s) et les temps de placement en millisecondes (ms).

	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Cas 6
temps d'attente classe 1 (s)	30.512	34.94		30.756	41.98	
temps d'attente classe 2(s)	0.0036	0.0069		0.00331	0.0057	
temps d'attente classe 3 (s)	4845.25	5056.33		5182.2	5285.62	
temps d'attente classe 4 (s)	4.5564	4.57		30.567	111.14	
temps d'attente moyen (s)	710.519	742.67	64.68	396.02	445.79	70.3
temps de placement classe 1 (ms)	20.22	60.04		21.12	60.45	
temps de placement classe 2 (ms)	3.63	6.226		3.031	5.69	
temps de placement classe 3 (ms)	269.37	1492.79		334.47	1463.35	
temps de placement classe 4 (ms)	7.25	26.49		28.6	233.31	
temps de placement moyen (ms)	50.39	250.20	13.08	49.19	239.31	76.39
nombre de rejets	21	19	0	154	137	0

TAB. 5.3: Influence de la qualité de service sur les performances de l'algorithme

L'introduction de la qualité de service augmente les temps de placements et d'attente mais les performances restent bonnes (temps de placement de 50ms sans classe virtuelle et de 250ms avec classes virtuelles). Globalement l'augmentation est due aux applications de classe 3 (ressources dédiées), qui doivent attendre longtemps avant de pouvoir être seules sur les machines et l'algorithme étudie toutes les machines et une date de début pour les tâches (cela prend plusieurs itérations avant de trouver le bon t_b). Avec les classes virtuelles les temps d'attentes sont similaires à ceux sans classe virtuelle par contre le temps de placement est plus élevé ; en effet, les calculs sont plus longs. Par contre il y a moins de rejets avec les classes virtuelles (plus de demandes peuvent être satisfaites car toutes les applications ont leur part de puissance du processeur). Les rejets correspondent tous à des applications deadline. Les applications de classe 2 pourraient être refusées mais un placement a toujours pu être trouvé. Les applications de classe 3 et 4 ne peuvent jamais être refusées par contre elles sont retardées.

5.6 Conclusion

Un algorithme de placement d'applications parallèles permettant de garantir une qualité de service a été présenté. Il est basé sur un modèle de file d'attente pour l'accès au processeur et une charge déterministe est considérée. Pour cela les applications sont réparties dans différentes classes d'applications, une priorité est associée à chacune de ces classes. A l'intérieur d'une même classe les applications peuvent être placées en fonction de la priorité des utilisateurs.

Les processus pour lesquels aucune information ne serait connue pourraient être classés dans la file des processus "best effort".

Une implémentation à base de processeurs virtuels a été présentée. Les différents relevés effectués ont tout d'abord permis de valider le système d'équation mis en oeuvre pour calculer les priorités des différentes classes d'applications. Puis dans un second temps, une heuristique a été proposée permettant d'approcher les résultats de Xpress.

Toutefois, l'algorithme de calcul des priorités sera beaucoup plus précis avec la version 2.6 de l'ordonnanceur Linux.
L'imprécision entre l'heuristique et Xpress ne diffère que très peu selon la version Linux. Mais les forts pourcentages d'erreur entre Xpress et les puissances souhaitées avec le noyau 2.4 pénalisent les résultats sur les priorités ; il est difficile de trouver une priorité des processus qui permette de respecter le pourcentage du processeur réel qui leur est accordé.
La nouvelle méthode de calcul du timeslice (version 2.6) permet une résolution beaucoup plus précise et l'obtention d'un meilleur calcul des priorités solutionnant le système.

Enfin le placement avec qualité de service a été validé, les performances sont bonnes, le temps de placement est inférieur à 250ms et une qualité de service est garantie aux applications.

Le chapitre suivant prend en compte l'influence des communications entre les tâches. Ceci est utilisé pour le placement sur une grille de calcul.

Chapitre 6

PLACEMENT SUR UNE GRILLE

6.1 Introduction

Dans ce chapitre, l'objectif est de considérer les communications entre les tâches et de placer les applications sur une grille de calcul constituée de plusieurs sites géographiques. Il peut être souhaitable de répartir la charge sur plusieurs domaines tout en tenant compte des points suivants pour le placement : le même cluster doit être privilégié ; si le placement n'est pas possible, le même domaine doit alors être un choix prioritaire ; si le placement est toujours impossible, il faut tenir compte du débit entre les sites de domaines différents.

Dans AROMA, il existe quatre niveaux dans l'architecture (voir exemple 6.1) :

- la grille (GRU)
- le domaine (DRU)
- le cluster (CRU)
- la machine (HRU)

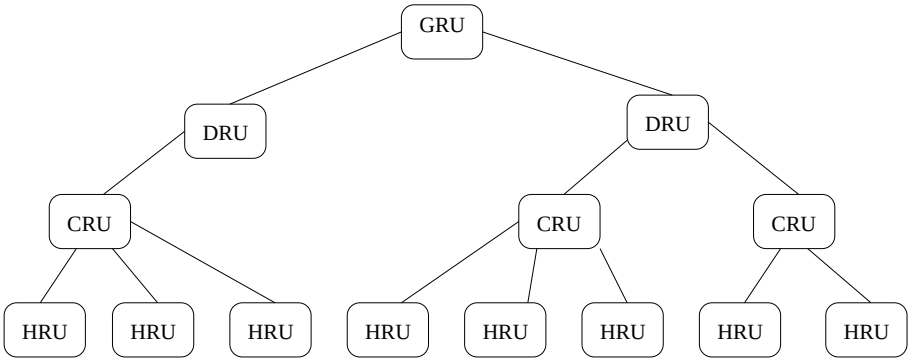


FIG. 6.1: Exemple d'une architecture d'AROMA

L'algorithme de placement proposé au chapitre précédent est mis en oeuvre au niveau d'un domaine et ne tient pas compte des communications entre les tâches. La modélisation de l'accès au réseau sera tout d'abord présentée et validée puis, une méthode de répartition de l'applications sur plusieurs domaines sera proposée. Dans un troisième temps, l'algorithme de placement au niveau grille sera expliqué. Pour finir, la validation sera explicitée.

6.2 Modélisation du réseau

6.2.1 Ethernet

La technologie Ethernet 10Base-T est la plus utilisée dans le domaine des réseaux locaux de stations de travail. Cependant, ses performances sont insuffisantes pour envisager un grain de communication fin dans une interconnexion de plusieurs centaines de noeuds de calcul. Ethernet 100Base-T est une évolution d'Ethernet vers des débits de 100 Mb/s. Cette technologie est surtout utilisée dans le monde du calcul haute performance pour réaliser des réseaux de contrôle (pour le transfert de fichiers, les informations systèmes ...) dans les grappes de stations de travail interconnectées par un réseau Gigabit. Gigabit Ethernet est une technologie full-duplex relativement onéreuse qui offre des débits de 1Gb/s tout en conservant beaucoup de caractéristiques d'Ethernet 10Base-T (protocole CSMA/CD, même format et même taille de trames ...). Cette technologie utilise soit des fibres optiques multimodes (longueur maximum de 550 m) soit des fibres optiques monomodes (longueur 3 km) soit des liens électroniques (longueur 25 m).

Dans notre cas, il faut tenir compte du fait que deux tâches qui émettent un message se partagent la bande passante, il faut donc considérer le réseau comme une file d'attente de messages (les communications) ayant chacun un temps de service à recevoir. Le réseau a donc été modélisé comme une file d'attente sur les processus en état de communication. Le réseau est une ressource commune partagée et utilisée simultanément par l'ensemble des machines connectées [12]. Le modèle est de type "round-robin" car les paquets d'une même communication peuvent être entrelacés avec ceux d'une autre application. La figure 6.2 décrit la modélisation pour deux machines multiprocesseurs reliées à un hub Ethernet 100Mb/s.

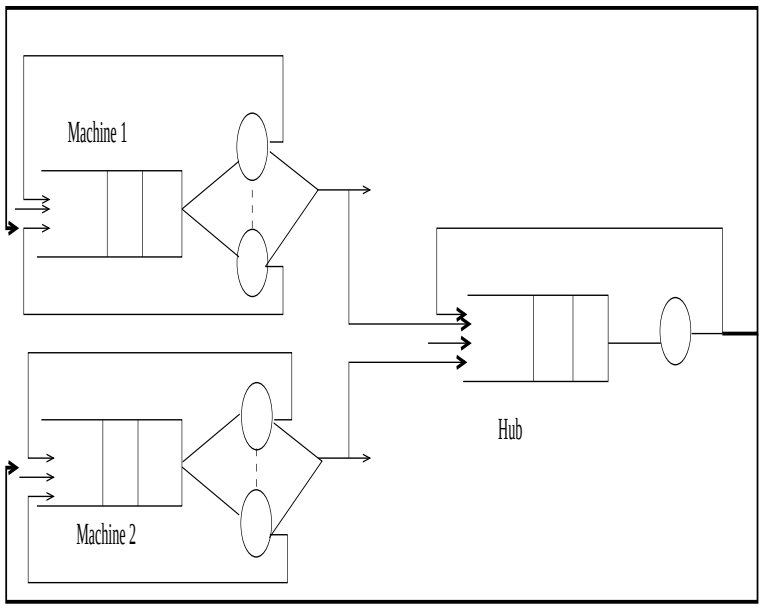


FIG. 6.2: Modélisation de deux machines multi-processeurs et d'un hub Ethernet.

6.2.2 Myrinet

Le réseau Myrinet a été développé par la société Myricom [14], c'est la combinaison de deux projets de recherche. Le premier concernait les systèmes d'interconnexion dans les machines parallèles (projet Caltech Mosaic) [16], le second visait à exploiter les composants de ces dernières pour construire un réseau haut débit (ATOMIC LAN) [17]. Myrinet est une technologie employée pour connecter des clusters, des PC et des serveurs. L'interconnexion se fait à l'aide d'une carte sur l'hôte reliée à un "switch". Les "switch" peuvent être interconnectés entre eux. Ce sont des "switch" de type "crossbar". La liaison est réalisée de différentes manières : câble parallèle sur un octet ou fibre optique. Les cartes d'interfaces permettent d'interagir directement avec les processus communiquants ("OS bypass"). Ceci permet d'éviter les appels système et les commutations de contexte et ainsi d'obtenir des communications de faible latence [18]. La latence est inférieure à 5μs, le débit est proche de 2Gbits/s. La carte réseau dispose de son propre processeur local et de sa propre mémoire RAM. Ceci permet de programmer le protocole de communication directement sur la carte réseau. Il existe d'ailleurs différents protocoles [14] [15] [19]. Dans [20] une librairie de communication pour des clusters de stations interconnectées par un réseau Myrinet est présentée. En effet, TCP n'est pas un protocole optimal pour le calcul distribué. Par ailleurs, Myrinet ne permet pas de faire du multicast ainsi, pour envoyer le même message à N destinataires, l'émetteur doit répéter l'envoi N fois. La librairie décrite dans [20] permet de diminuer cette surcharge en réutilisant le message précedemment envoyé. Toutefois, si le multicast doit concerner un nombre important de récepteurs, il devient alors nécessaire d'utiliser une autre technique : copie et transfert de données dans une structure d'arbre. Programmer le protocole de communication directement sur la carte réseau permet une utilisation très faible du processeur de l'hôte lors de communications. La présence de ce processeur alliée à un programme auto-chargeable assurent le transfert de paquets de données et la gestion du réseau (conversion des adresses stations en routes, planification et contrôle). Le logiciel de Myrinet fait la cartographie du réseau et utilise n'importe quel chemin de communication disponible entre des hôtes. Aucune programmation de "switch", aucune configuration des tables de routage n'est nécessaire. Le routage utilisé est du type "cut-through". L'algorithme de routage utilisé est du type “source routing” : toutes les informations de routage sont contenues dans la trame. Les interfaces des hôtes n'ont pas d'adresse, l'information de routage doit être assurée de manière logicielle. Il n'y a pas de taille maximale de messages, il peut donc y avoir encapsulation de n'importe quel type de paquet sans couche d'adaptation. Myrinet utilise un contrôle de flux basé sur la méthode STOP/GO, il y a une garantie de délivrance des messages et l'ordre des messages transférés par le même chemin est préservé. Les interfaces dans les hôtes Myrinet assurent la flexibilité et la haute performance. Par ailleurs, Myricom fournit une interface standard TCP/IP, UDP/IP et une API (Application Programming Interface) Myrinet [21].

GM est une librairie de communication pour les réseaux Myrinet comprenant un “driver” d'espace noyau, un programme d'exploration réseau et une interface utilisateur. Ainsi, l'accès à l'interface de Myrinet est protégé ; des accès concurrents sont possibles et les caractéristiques de Myrinet sont conservées : fiabilité, garantie d'ordre des messages délivrés, extension à des milliers d'hôtes possible, très faible utilisation CPU des machines et deux niveaux de priorité des messages sont possibles : bas et élevé. Les émissions et les réceptions sont réglementées par un nombre limité de jetons disponibles. Un message ne peut être émis que si l'émetteur possède un jeton pour émettre et un message ne peut être reçu que si le récepteur possède

un jeton pour recevoir. Des jetons peuvent être réservés et dans ce cas, la mémoire ne sera disponible que lorsqu'un message sera reçu.

Les "switch" de Myrinet étant de type "crossbar", le "switch" peut gérer plusieurs communications simultanément en sortie : chaque machine connectée au "switch" en entrée peut établir une communication avec une machine connectée à un port de sortie pourvu que ce port ne soit pas déjà utilisé par une machine en entrée. Si le port est déjà utilisé, il faut attendre la fin de la communication de la machine utilisant ce port de sortie. Ce cas va créer un phénomène d'attente. L'ordre de passage lié au protocole "cut-through" se fait en FIFO.

La modélisation proposée est basée sur la théorie des files d'attente [22] et la théorie du trafic différentiel [23].

Le fait que les "switch" soient "crossbar" et donc permettent de gérer plusieurs communications simultanément implique une modélisation avec plusieurs serveurs. Lorsque deux machines souhaitent communiquer avec la même machine destinataire, l'une des deux doit attendre que la première ait entièrement fini sa communication. Ceci est dû au routage "cut-through" qui impose un ordre de passage de type FIFO. Le modèle proposé intègre cela en mettant une file d'attente par port du "switch" en sortie.

Le modèle proposé pour les communications est donc construit avec un ensemble de files d'attente FIFO mono-serveur associées aux ports de sortie. Une première approximation est faite en considérant des flux poissonniens en entrée λ et des taux de traitement exponentiels μ sur des intervalles de temps. Ceci amène donc à considérer des files M/M/1.
La figure 6.3 représente la modélisation de deux machines multi-processeurs connectées à un "switch" Myrinet.

Les communications modélisées sont bloquantes. Soit un processus placé sur la machine 1 qui demande à faire du calcul, il est dans la file X1, il utilisera dès que possible un processeur. S'il a ensuite besoin de communiquer avec la machine 2 avec le réseau Myrinet, le message à transmettre sera placé dans la file Y2M. Quand il aura fini sa communication le processus sera remis dans la file X1. Sur la figure 6.3 le chemin de ce processus est représenté.

6.2.3 Validation du modèle d'accès au réseau

Pour valider, des tests ont été effectués sur le réseau Myrinet. Le programme utilisé est écrit en C et comporte des appels de fonctions de la librairie MPI (bibliothèque de communication par échange de messages [32]). Trois tâches sont utilisées : les tâches 1 et 2 envoient des messages à des dates tirées aléatoirement suivant une loi exponentielle à la tâche 3. La taille des messages est également tirée aléatoirement. La tâche 3 répond à l'émetteur du message grâce à un acquittement (message de taille un octet). La tâche 1 calcule le temps passé dans le système pour ses messages : temps entre l'envoi du message et la réception de l'acquittement en soustrayant le temps passé en émisssion de la réponse (calculé grâce à la taille de l'acquittement, le débit et la latence du réseau Myrinet). Le débit et la latence du réseau Myrinet ont été mesurés expérimentalement. La tâche 1 fait une moyenne sur plusieurs lancements (T_exp) et compare à l'espérance théorique (T_theo) du temps moyen passé dans le système d'une file M/M/1.

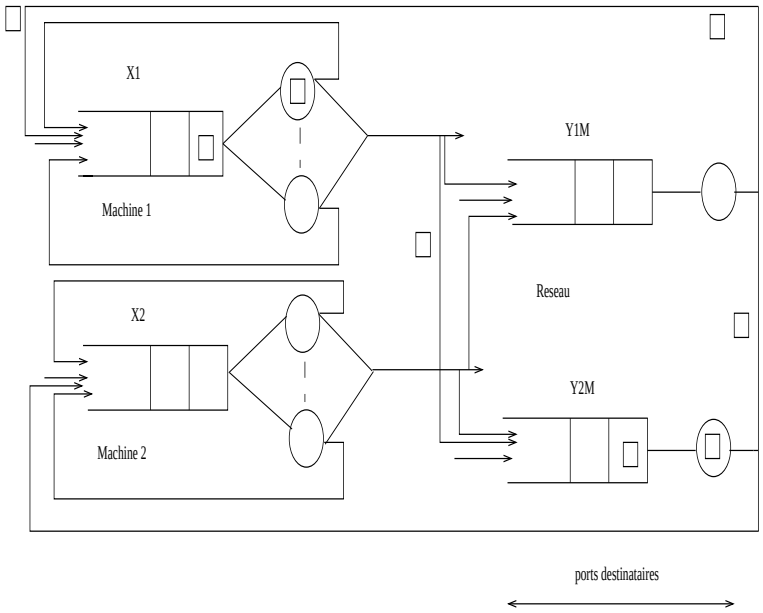


FIG. 6.3: Modèle de communication

Il est connu que :

$$T_{theo} = \frac{1}{\mu - \lambda} \tag{6.1}$$

Expérimentalement, il est calculé de la manière suivante :

$$T_{exp} = \frac{\text{date aller} - \text{date retour} - \text{temps acquittement}}{\text{nombre tours}} \tag{6.2}$$

Deux machines sont donc émettrices et une machine est réceptrice ce qui permet de créer un goulot d'étranglement et de vérifier que le modèle basé sur une file M/M/1 par machine destinataire est correct. Le schéma des communications engendrées est donné figure 6.4. La courbe 6.5 présente les résultats obtenus pour différents taux d'utilisation du réseau : les temps théoriques et expérimentaux sont comparés.

Pour des taux d'utilisation de la file faibles, les temps moyens théoriques et expérimentaux sont proches, le modèle représente donc bien la réalité. Pour des taux d'utilisation de la file importants, une erreur est observée, dans ce cas le modèle n'intégre donc pas tous les paramètres. Toutefois, si des applications gros grain sont considérées, le taux d'utilisation du réseau sera faible donc le modèle convient pour les applications considérées.
Les modèles présentés et validés sont des modèles proches de la réalité pour l'accès au réseau. Toutefois, ils ne seront pas utilisés pour le placement grille mais simplifiés dans un souci de rapidité de résolution de l'algorithme (paragraphe 6.4.2).

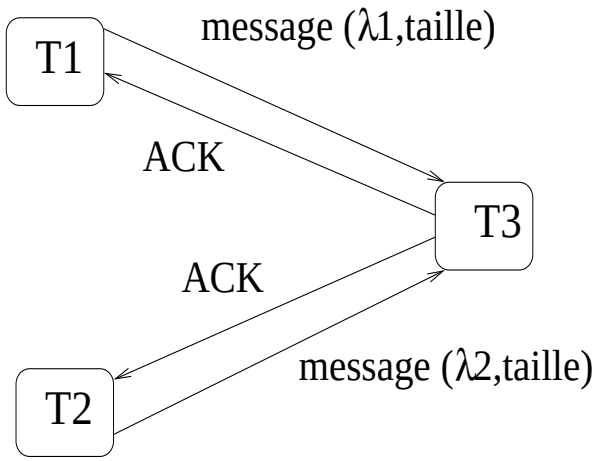


FIG. 6.4: Communications engendrées

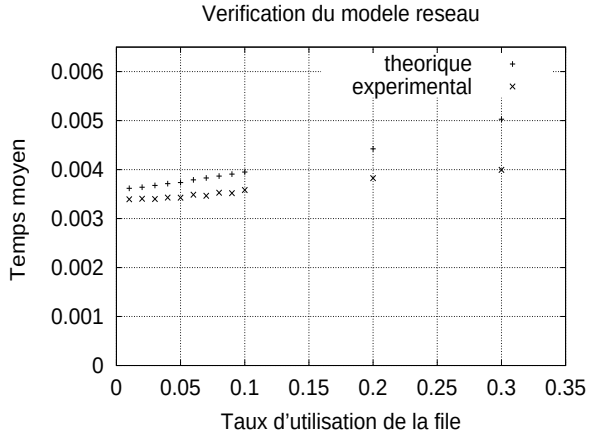


FIG. 6.5: Comparaison temps théorique et expérimental passé dans le système

6.3 Méthode de répartition sur plusieurs domaines

6.3.1 Graphe de tâches

L'application sera modélisée par un graphe de tâches où les sommets représenteront les tâches (et leurs temps de calcul) et les arcs la quantité d'information émise (en octet) ainsi que la "direction" (sur la figure 6.6 la tâche 3 envoie au total au cours de l'exécution 20 koctets à la tâche 2).

6.3.2 Découpe du graphe

L'objectif est de répartir la charge sur différents domaines ; c'est à dire répartir les tâches des applications sur plusieurs domaines si l'exécution en est accélérée. Pour l'algorithme de placement au niveau grille, ce graphe de tâches sera découpé en sous-graphes correspond à des ensembles de tâches dont le placement sera étudié sur différents domaines.

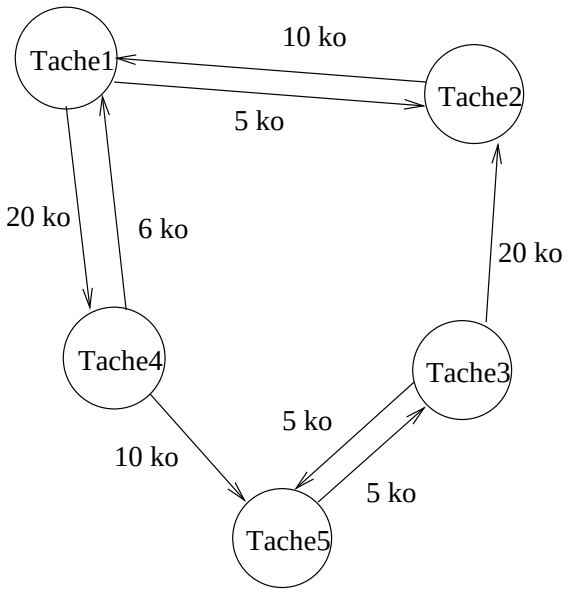


FIG. 6.6: Graphe de communications entre différentes tâches

Pour cela, une méthode proche de celle de Kruskal [109] est utilisée. La méthode de Kruskal permet de réunir deux sommets (ou deux sous-graphes) disjoints par un arc de coût minimal, la même opération sera effectuée sur les arcs de communication maximaux. Ainsi les tâches s'échangeant le plus de données seront regroupées. Le regroupement s'arrête lorsque le nombre maximum de tâches pouvant être placées sur un domaine est atteint.

Cette découpe entraîne trois états possibles pour une tâche :

- non placée, son placement n'a pas encore été étudié
- en cours de placement
- déjà placée

Le graphe de communication précédent est repris pour illustrer le principe du regroupement de tâches (voir exemple 6.7). Dans un premier temps un sous-graphe est créé contenant les tâches 1 et 4 (arc de communication maximum entre ces deux tâches) puis un second sous-graphe avec les tâches 2 et 3.
Par la suite la gestion de cette découpe ainsi que son utilité sera expliquée.

6.3.3 Calcul du nombre de tâches pouvant être placées sur un domaine

La machine

Au niveau des machines, l'information la plus importante concerne la charge de celles-ci au cours du temps. Il faut donc connaître le nombre de tâches en cours d'exécution résultant des placements déjà effectués.

Pour représenter cela, une courbe montrant l'évolution de la charge sur vingt-quatre heures sera utilisée, en étudiant la charge tous les quarts d'heure durant la première heure puis chaque

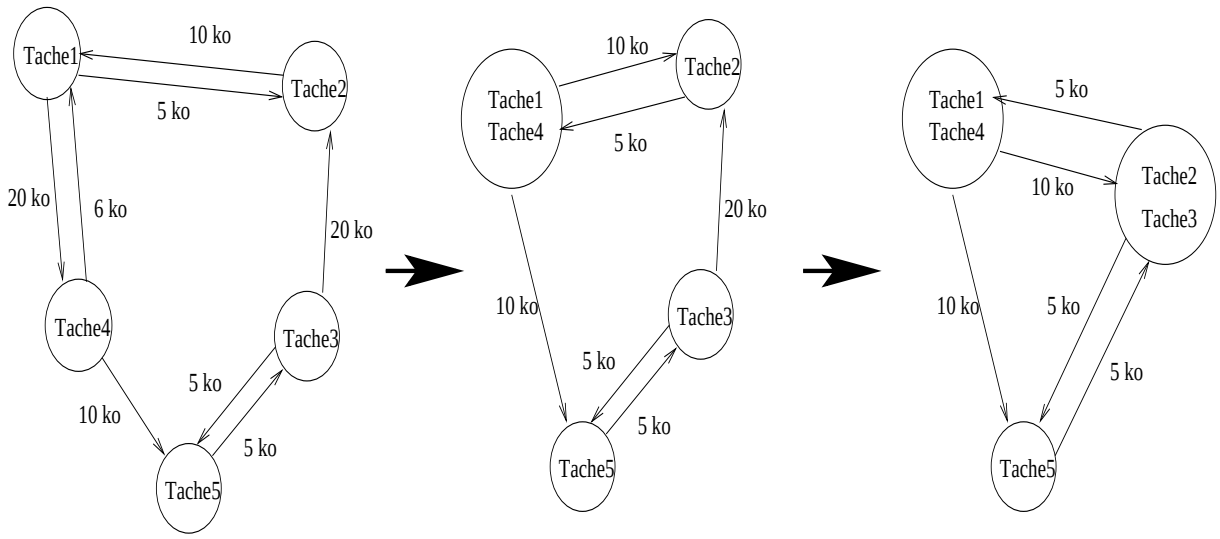


FIG. 6.7: Exemple de regroupement de tâches

heure par la suite. Cette agrégation d'informations est nécessaire pour limiter le flux d'informations sur le réseau et la taille mémoire. Les applications nécessitant l'exécution sur une grille demandent beaucoup de temps de calcul (cela se mesure en heures), l'agrégation est raisonnable.
Dans l'exemple (figure 6.8), aucune tâche n'est exécutée entre la trentième et la quarante-cinquième minutes sur la machine.

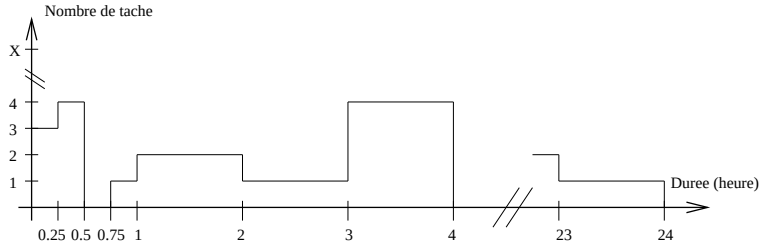


FIG. 6.8: Exemple de courbe représentative d'une machine

Le domaine

Les données utiles au placement au niveau du domaine sont :

- la bande passante du domaine vers l'Internet
- le nombre de machines (donc de processeurs)
- le nombre de tâches pouvant être placées ainsi que la date de début correspondante

Pour calculer ce nombre de tâches et la date de début correspondante, une courbe pratiquement identique à celle des machines sera utilisée. La seule différence se situera au niveau de l'axe des ordonnées qui représentera le nombre de tâches pouvant être placées sur le domaine

au cours du temps. Pour créer cette courbe toutes les machines du domaine sont interrogées afin qu'elles envoient les informations concernant leur courbe.

Puis le nombre de tâches pouvant être placées en fonction du nombre de processeurs du domaine et des tâches s'exécutant sur les machines seront calculés.

Définition de variables :

- $N_{[i,j]}$: nombre de tâches pouvant être placées durant l'intervalle [i,j] de la courbe (tous les quarts d'heure pendant la première heure puis toutes les heures).
- nb : nombre de machines sur le domaine
- $M^m_{[i,j]}$: nombre de tâches en cours d'éxecution présentes sur une machine m durant l'intervalle [i,j]
- n_d : nombre de processeurs sur le domaine d

A partir de ces données, la courbe au niveau du domaine est créée pour tous les intervalles [i,j] tel que :

$$N_{[i,j]} = n_d - \sum_{k=1}^{nb} M^k_{[i,j]} \tag{6.3}$$

Le calcul du nombre maximum de tâches pouvant être placées sur chaque domaine permet de trier les domaines.

Ordonnancement des domaines

Les domaines sont classés en fonction de deux paramètres. Le premier concerne le nombre maximum de tâches pouvant être placées, en cas d'égalité ils sont classés en fonction de la date de début correspondante.

A partir de la courbe d'évolution vue précédement, de la qualité de service de l'application, ces deux paramètres (nombre maximum de tâches pouvant être placées et date de début correspondante) sont cherchés sur tous les domaines de la grille.

Définition de variables :

- Nmax : nombre maximal de tâches pouvant être placées
- d : date de début
- t : temps cpu demandé par l'application (données QoS)
- t_b(a) : date de début de l'application a (données QoS ou calculées par l'ordonnanceur)
- t_f(a) : date de fin de l'application a (données QoS ou calculées par l'ordonnanceur)

L'algorithme parcourt la courbe du domaine tant que la qualité de service de l'application est préservée (date de début et date de fin). En effet, si l'application a une date de début imposée, il ne sert à rien d'étudier la courbe avant cette date ; de même si l'application a une "deadline" il ne sert à rien d'étudier la courbe après cette date. De plus, lorsque le temps cpu demandé est supérieur à un intervalle de la courbe, le minimun des $N_{[i,j]}$ est calculé sur les différents intervalles concernés. Nmax et d sont ainsi calculés.

6.4 Le placement au niveau grille

6.4.1 Graphe de l'application

Dans la réalité, l'exécution d'une application se compose de plusieurs phases : calcul, communication, calcul, communication ... Ainsi, les tâches de l'application sont partagées en plusieurs sous-groupes (un sous-groupe correspond à une phase de calcul). Une tâche réseau virtuelle est introduite entre tout couple de tâches de l'application. C'est une tâche particulière qui ne sera pas placée ; le temps qu'elle "demande au réseau" correspond à la taille des communications. Si l'application utilise des communications de type "all to all", chaque tâche réseau sera utilisée ; sinon, certaines tâches réseau auront une taille de 0. La tâche réseau permet de représenter facilement les précédences qui existent entre une phase de calcul et une phase de communication. Une tâche de calcul de sous-groupe différent de 1 ne pourra commencer son exécution que lorsque toutes les tâches émettrices auront terminé leur calcul et leur émission.

Sur la figure 6.9, les tâches 1', 2' et 3' constituent le premier sous-groupe. Les tâches 1", 2" et 3" constituent le deuxième sous-groupe. Les temps de calcul sont partagés entre les différentes phases. Par exmple, si la tâche 1 demande 500s de processeur, la tâche 1' demande 250s et la tâche 1" demande elle aussi 250s. La tâche 1" doit attendre que toutes les tâches qui la précédent dans le graphe ait terminé pour commencer (c'est la même chose pour les tâhces 2" et 3").

Le découpage de l'application en différents sous-groupes permet de synchroniser les tâches de calcul : cela évite d'avoir toute la phase de calcul puis toute la phase de communication (voir exemple 6.9).

Le nombre de niveaux de découpe en phase de calcul-communication est paramétrable. Toutefois si trop de niveaux sont insérés, il y a une explosion combinatoire.

6.4.2 Calcul des temps réseau

Le calcul du temps passé en communication se fait en fonction de la taille des messages et du débit du lien reliant la machine émettrice et la machine réceptrice. Les modèles précédents ont été simplifiés.

Lorsqu'un domaine reçoit une demande de placement il reçoit en paramètre l'application entière même s'il n'a qu'un sous-groupes de tâches à placer. Ainsi, lorsque le temps de communication est étudié, la tâche émettrice peut ne pas être encore placée, par contre la machine où l'on trouve la tâche réceptrice est potentiellement connue puisque l'on étudie le placement sur cette machine.
Plusieurs cas sont considérés :

- la tâche émettrice est placée sur le même domaine que la tâche réceptrice : le temps réseau est calculé en considérant le débit réel entre les deux machines.
- la tâche émettrice est placée sur un autre domaine que la tâche réceptrice : le temps réseau est calculé en considérant le débit minimal entre les deux domaines.
- la tâche émettrice est à placer sur le même domaine que la tâche réceptrice : une vision optimiste est utilisée : le temps réseau est calculé en considérant le débit du cluster de

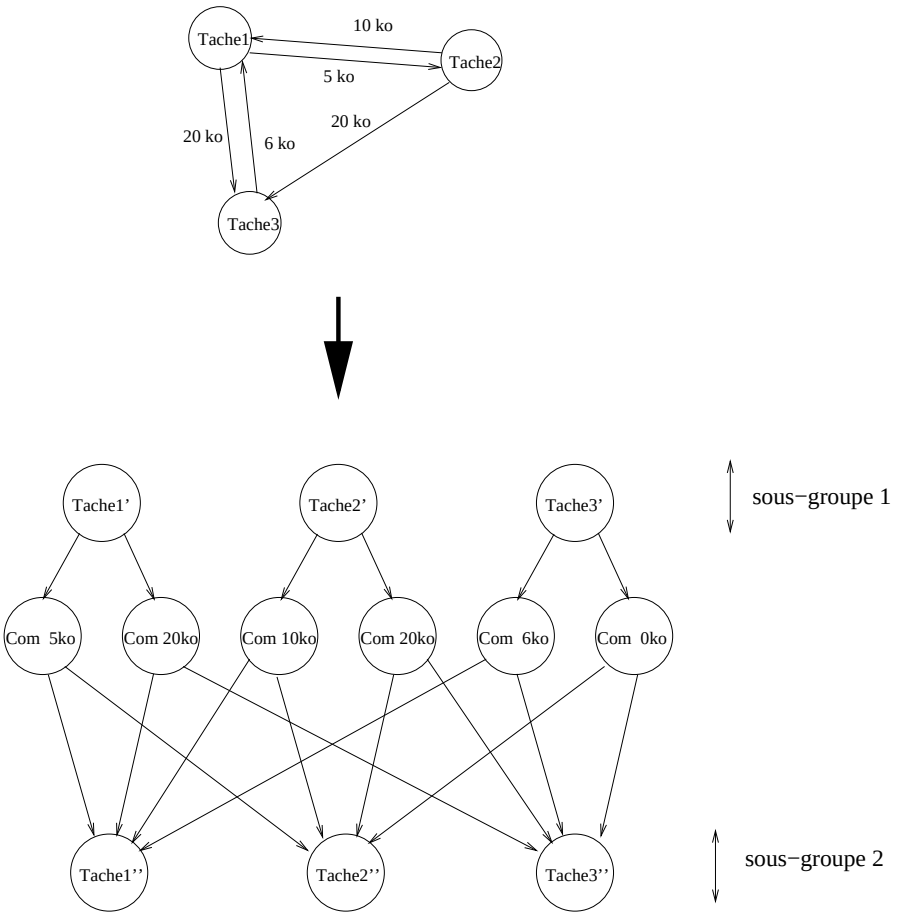


FIG. 6.9: Création de graphe d'application avec précédence à partir d'un graphe de tâches communicantes

la machine réceptrice.

– la tâche émettrice n'est pas placée : le temps réseau est calculé en considérant le minimum des débits entre celui du cluster et celui du domaine vers l'internet (débit en sortie).

6.4.3 Algorithme au niveau grille

L'algorithme de placement au niveau grille consiste à découper l'application reçue en groupe de tâches, à envoyer la demande sur différents domaines puis à choisir le meilleur. Le placement au niveau domaine sera calculé par une autre instance de l'ordonnanceur qui ne connaîtra que les informations relatives à son domaine.

Pour illustrer le principe, un exemple est proposé (voir schéma 6.10). L'ordonnancement des domaines vu précédemment permet dans un premier temps de les classer dans une liste (du plus grand nombre de tâches possibles à placer au plus petit).

Ensuite une boucle est exécutée tant qu'il reste des tâches à placer. A partir des données du premier domaine de la liste, un premier sous-graphe de tâches de l'application est créé.

Ce groupe de tâches est placé sur plusieurs domaines de la liste (requêtes notées 1 et 3 sur l'exemple) et la date de fin d'éxecution (notée 2 et 4) sur chacun des domaines est récupérée (celle-ci est calculée par une simulation du placement).

Le domaine dont la date de libération des ressources est minimale recevra un message de confirmation de placement, les autres, un message leur indiquant de supprimer le groupe de tâches. Pour minimiser l'échange de communication entre la grille et les domaines, ces messages seront envoyés à l'itération suivante de notre boucle.

Le nouveau groupe de tâches à placer est modifié en ôtant le sous-graphe qui vient d'être étudié.
Enfin, la liste des domaines est réorganisée en fonction du nouveau placement.

Il faut également noter que nous réessayons toujours un placement sur le domaine précédemment choisi quelle que soit sa place dans la liste ordonnée. On recherche ainsi à privilégier le placement sur un même domaine.
Dans l'exemple, le premier choix de placement concerne le domaine1 puis le domaine2. On peut noter qu'à chaque itération un placement est redemandé sur le dernier domaine choisi.

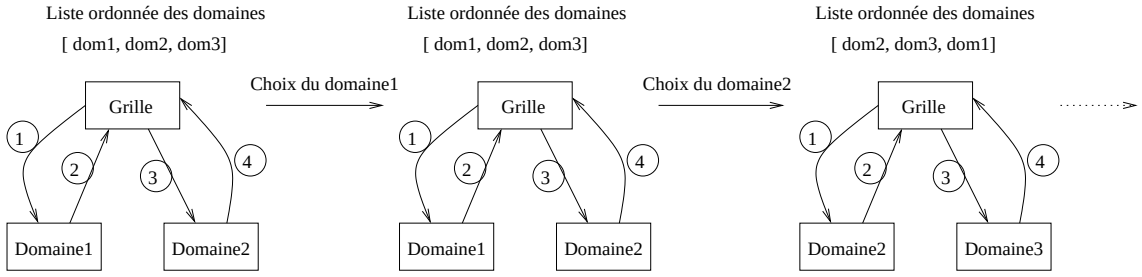


FIG. 6.10: Exemple simple de demande de placement au niveau grille

Définition de variables :

- $etat_d$: état du domaine d
- N_d : nombre de tâches pouvant être placées sur le domaine d
- d_d : date de début où N_d peut être placé
- G : groupe placé sur le domaine d
- R : reste des tâches à placer
- B_d : bande passante du domaine d vers l'Internet
- t_{fd} : date de fin calculée par simulation du groupe de tâches affecté au domaine d

L'algorithme de placement au niveau grille est présenté figure 6.11.

Lorsque le placement est impossible sur chacun des domaines disponibles, l'application est rejetée. Le critère de choix d'un domaine étant basé sur la date de libération des machines, il tient compte du temps passé en communication et donc privilégie autant que possible le même domaine.

```
Initialisation de données
etat_d=debut
Sauvegarde=vide

Liste=Ordonnancer les domaines selon leur N_d

Boucle de Placement
while (R!=vide) do
  // il reste des tâches à placer
  Nmax_d = N_d du premier domaine de la liste
  G=Decouper(R,N_dmax,B_d)

  for domaine d=1 à X où il est choisi les X premiers domaines de la Liste ainsi que
  le domaine sauvegardé do
    Info_d=GetInfo(G,etat_d) // Placement de G sur le domaine d
    // Avec Info_d contenant les données de placement t_fd ainsi que le nouveau calcul
    de N_d et d_d
  end for
  tmin_d=min(t_fd) // minimum de tous le temps

  for Domaine d=1 à X do
    if Domaine d correspond à tmin_d then
      // G déjà placé sur le domaine d
      R=R-G_d
      etat_d=OK
      Sauvegarde=domaine d
    else
      etat_d=NOK
    end if
  end for
  Réorganiser la liste de domaines
end while
etat_d=fin
```

FIG. 6.11: L'algorithme de placement au niveau grille

6.4.4 Algorithme au niveau domaine

Quand une demande de placement est envoyée à un domaine, elle concerne la tâche réelle i c'est à dire l'ensemble des tâches virtuelles i' et i" de tous les sous-groupes. Un placement (figure 6.12) est cherché uniquement pour les tâches du premier sous-groupe. Pour les tâches des sous-groupes supérieurs ou égal à deux, la machine est imposée par le placement de la tâche de calcul équivalente du premier sous-groupe, il suffit de vérifier qu'il est possible (vérification des contraintes). Les tâches 1' et 1" sont virtuelles, en réalité elles représentent la même tâche : tâche 1. Ainsi, lorsqu'un placement pour la tâche 1' a été trouvé, la tâche 1" doit obligatoirement s'exécuter sur la même machine. Cependant, il faut vérifier les contraintes de la tâche 1". La recherche de placement d'une tâche de premier sous-groupe se déroule toujours de la même manière.

```
for sous-groupe=1 à MAXSOUSGROUPES do
  if sous-groupe==1 then
    for tache=1 à nb_taches_a_placer du sous-groupe do
      for chaque machine du domaine do
        Etude du placement
      end for
      Choix de la machine
    end for
    Validation du placement des tâches du premier sous-groupe
    Calcul du temps passé en communication
    Ajout des contraintes des dates de début des tâches de sous-groupe ! =1
    Ajout des contraintes de machines des tâches de sous-groupe ! =1
  else
    for tache=1 à nb_taches_a_placer du sous-groupe do
      Vérification que le placement est possible
    end for
    Validation du placement des tâches du sous-groupe
    if sous-groupe! =MAXSOUS-GROUPES then
      Calcul du temps passé en communication
      Ajout des contraintes des dates de début des tâches de sous-groupe supérieur
    end if
  end if
end for
Validation du placement ou rejet
```

FIG. 6.12: L'algorithme de placement au niveau domaine

6.5 Validation

La validation du placement sur une grille de calcul s'est effectuée par simulation : le simulateur Simgrid [108] a été utilisé.

6.5.1 Présentation de SimGrid

Présentation générale

SimGrid est un logiciel développé conjointement par Henri Casanova et Arnaud Legrand. C'est un simulateur modulaire (écrit en C) permettant de simuler une application distribuée où les décisions d'ordonnancement peuvent être prises par différentes entités. La force de ce simulateur réside dans sa capacité à importer et à simuler aisément des plates-formes réalistes (de la grille à la station de travail).

Le composant le plus important du procédé de simulation est la modélisation des ressources et des tâches. Les ressources peuvent être de trois types :

- machines (host)
- lien réseau (link)
- liste de lien réseau (route)

Les tâches sont de deux types :

- traitement (computing)
- temps réseau (transfert)

Simgrid fournit des mécanismes permettant de modéliser ces caractéristiques soit par un vecteur de valeur soit par une trace.
Il utilisera en priorité les traces, celles-ci permettant la simulation des fluctuations arbitraires d'exécutions (temps partagé par exemple) comme celles observables pour de vraies ressources.

Principe d'utilisation

Pour utiliser SimGrid, il faut toujours commencer par définir la topologie du réseau sur lequel le placement est simulé. Tout d'abord les machines sont créées, puis les liens réseau et enfin la table de routage (liste de liens permettant de relier les stations).
Lorsque le réseau est défini, les tâches peuvent être déclarées et placées. Les tâches dites de "traitement" ne pourront être placées que sur des machines alors que les liens réseau recevront celles dites de "transfert".

Quatre possibilités de simulations existent :

- simulation totale, elle se termine lorsque toutes les tâches ont été accomplies.
- simulation durant une durée d (en seconde).
- simulation jusqu'à ce que la tâche t soit finie.
- simulation jusqu'à ce qu'une tâche (n'importe laquelle) soit terminée.

6.5.2 Interface AROMA/SimGrid

Le but est d'interfacer l'ordonnanceur d'AROMA avec le simulateur SimGrid.
Il faut savoir qu'AROMA est essentiellement codé en Java et que Simgrid doit être utilisé en langage C. Une communication Socket permet d'échanger les données entre l'interface (client Java) et SimGrid (serveur en C).

Echange d'informations (voir figure 6.13) :

L'interface envoie à SimGrid les différentes informations sur les tâches à exécuter (numéros de tâche, application, date de début, machine) ainsi que le temps de simulation (utilisation de Simgrid en mode de simulation sur durée).
SimGrid communique les informations concernant le temps processeur consommé (CPU consommé), la date à laquelle ce temps a été calculé (WatchDate) ainsi que les dates de fin pour les différentes tâches.

Trois types d'évènements associés à des tâches permettent d'arrêter la simulation SimGrid :

- la date de début d'une tâche
- la date de fin d'une tâche
- la date de soumission d'une tâche, date avant laquelle la tâche doit être placée.

Ces trois évènements vont permettre de calculer le temps de simulation à chaque itération de boucle de l'interface Java.

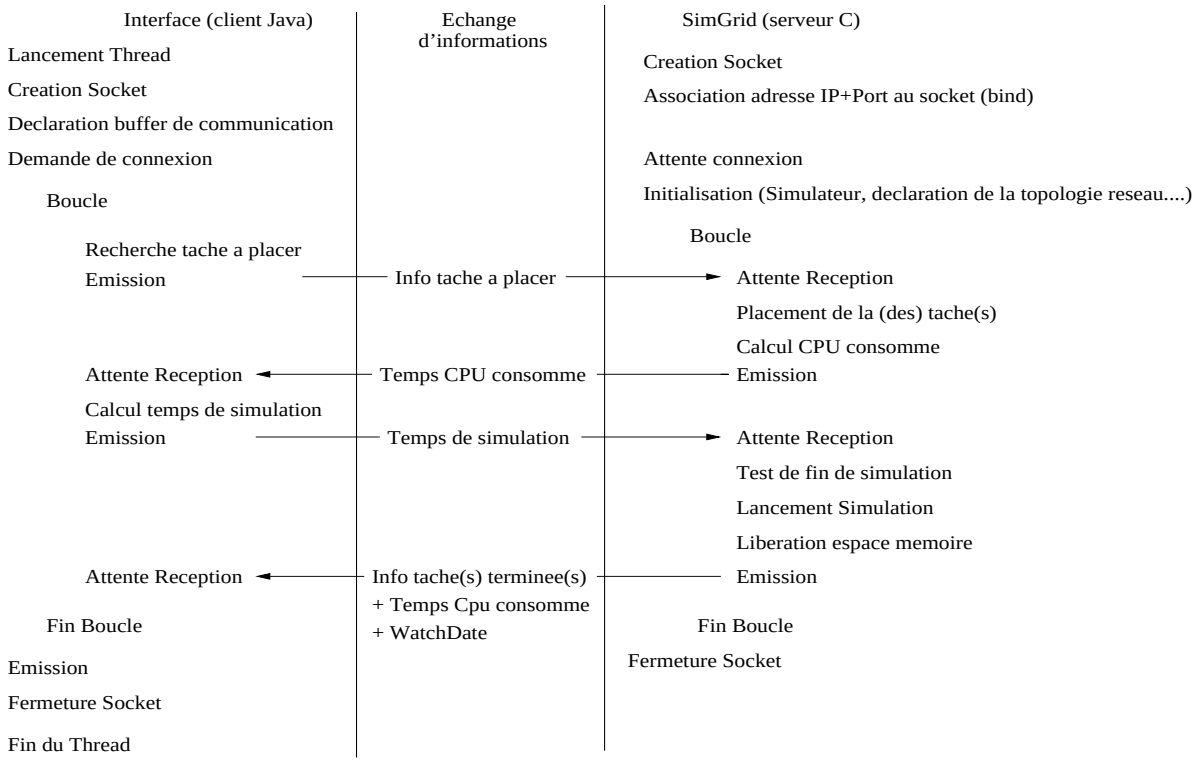


FIG. 6.13: Diagramme d'échange de données entre le client AROMA et le serveur SimGrid

6.5.3 Architecture de test

L'ensemble des tests sont effectués sur une architecture au niveau grille possédant 4 domaines. Chaque domaine est composé de deux clusters de 8 machines (voir schéma 6.14).

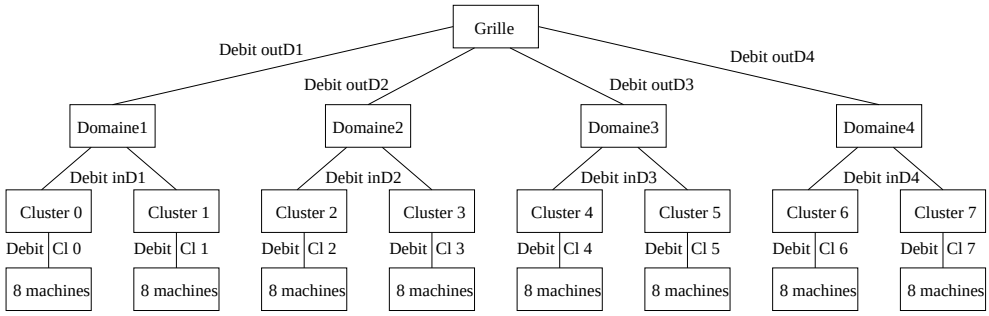


FIG. 6.14: Architecture utilisée pour les tests

Les seules données qui varient sont les différents débits de communication entre les machines (Debit Cl0-7), les clusters (Debit inDom1-4) et les domaines (Debit outDom1-4).

6.5.4 Tests effectués

Deux types de test différents ont été mis en place :

- le premier afin de constater l'influence de la qualité de service sur les temps de placement de l'algorithme.
- le second pour montrer les différences de placement sur les domaines en fonction des débits.

Influence de la QoS

Trois cas de simulation sont étudiés :

- cas 1 : placement de 100 applications (possédant entre 1 et 20 tâches) avec qualité de service, sans processeurs virtuels.
- cas 2 : placement de 100 applications (possédant entre 1 et 20 tâches) avec qualité de service, avec processeurs virtuels.
- cas 3 : placement de 100 applications (possédant entre 1 et 20 tâches) sans qualité de service, sans processeurs virtuels.

Sur ces 100 applications, 60 ont une contrainte de deadline (date de fin maximale fixée), une dizaine à exécution immédiate, 5 avec ressources dédiées et les 25 dernières sans aucune contrainte.

Influence des débits

Pour tester l'influence du débit sur le placement deux tests sont effectués :

- le premier avec des débits sortants des domaines (outD1-4) forts. La valeur fixée est : outD1=300 Mo/s, outD2=150 Mo/s, outD3=200 Mo/s, outD4=250 Mo/s.

– le second avec des débits faibles, on prendra : outD1=20 Mo/s, outD2=5 Mo/s,outD3=10 Mo/s, outD4=10 Mo/s.

Ces deux tests sont lancés sur 20 applications (demandant en moyenne un temps processeur de 6 heures).
Dans un premier temps l'écart entre chaque date de soumission de ces applications est fixé aux environs d'une heure puis dans un second temps à dix minutes.

6.5.5 Résultats

Influence de la QoS

L'influence de la qualité de service dans l'algorithme de placement (voir tableau 6.1) est à noter.
Lorsqu'il n'y a pas de contraintes sur les applications le temps de placement moyen est aux alentours de deux secondes et demi (ce temps était de l'ordre de la milli-seconde lorsque le placement ne s'effectuait que sur un domaine et ne prenait pas en compte les communications réseau). L'ajout de la qualité de service augmente le temps de placement moyen, de plus l'utilisation de processeurs virtuels augmente encore ce temps (19.8 secondes contre 18.2 sans les processeurs virtuels). Cette différence peut être expliquée par le fait que les calculs sont plus importants dans ce cas.

On note que les applications rejetées sont les mêmes dans les cas 1 et 2. Ce sont des applications de classe 1 (Deadline).

	Cas 1	Cas 2	Cas 3
Tps de placement (s) classe 1 (Deadlines)	21.35	22.57	
Tps de placement (s) classe 2 (immédiate)	2.37	0.921	
Tps de placement (s) classe 3 (dédiées)	1.82	3.23	
Tps de placement (s) classe 4	15.09	15.88	
Tps de placement moyen (s)	18.203	19.876	2.466
Nombre de rejets	17	17	0

TAB. 6.1: Influence de la qualité de service sur les performances de l'algorithme

Influence des débits

Des placements différents sont obtenus entre les débits forts et les débits faibles lorsque les écarts entre dates de soumission sont aux alentours d'une heure (voir tableau 6.2).
Pour le test sur les débits forts, 3 applications (sur les 20) ont été réparties sur plusieurs domaines. Par contre pour les débits faibles chaque application a été placée sur un seul domaine. Ceci est cohérent : en effet les communications entre les domaines pénalisent la date de libération des machines. C'est en restant sur un même domaine que le placement est le plus efficace.

On notera également que le domaine ayant le débit sortant le plus fort acceuillera le maximum de tâches (domaine 1 dans ce cas), le domaine avec le débit le plus faible aura le moins de

tâches (domaine 2 ici).

	Domaine 1	Domaine 2	Domaine 3	Domaine 4
Test débit fort (nb de tâches placées)	150	42	60	78
Test débit faible (nb de tâches placées)	162	28	60	80

TAB. 6.2: Influence des débits sur le choix de placement de l'algorithme (écart moyen entre dates de soumission : 1 heure)

Lorsque les applications sont soumises au placeur chaque dix minutes environ (voir tableau 6.3) le placement est identique quels que soient les débits des domaines. Dans les deux cas, 3 applications ont été réparties sur plusieurs domaines. Pour les débits faibles, cette situation provient du fait que l'on charge rapidement les quatre domaines. Dans ce cas, les faibles débits de communications ne pénalisent plus la date de libération des machines. Certaines applications sont donc divisées sur plusieurs domaines.

	Domaine 1	Domaine 2	Domaine 3	Domaine 4
Test débit fort (nb de tâches placées)	80	98	56	96
Test débit faible (nb de tâches placées)	80	98	56	96

TAB. 6.3: Influence des débits sur le choix de placement de l'algorithme (écart moyen entre date de soumission : 10 minutes)

6.6 Conclusion

Ce chapitre a présenté le travail réalisé sur le placement au niveau d'une grille réseau.
Le mise en place de ce placement a necessité dans un premier temps la représentation des entités utiles (graphe de tâches, applications...) puis l'implémentation de l'algorithme. Enfin il a fallu mettre en oeuvre des tests permettant de valider l'efficacité et les performances du placement.

Toutefois comme il a été vu sur le test concernant les applications possédant de faibles écarts de date de sousmission il reste des imperfections dans l'algorithme. Ces erreurs sont dues aux informations concernant l'évolution de la charge sur les machines des domaines. Le fait de vouloir minimiser les échanges de données entre la grille et les domaines a incité à étudier cette charge tous les quarts d'heures (durant la première heure) puis chaque heure. Ceci entraine certaines imprécisions dans le placement.
Il faudrait donc trouver un "juste milieu" entre la taille des informations nécessaires et un temps de placement minimal.

Chapitre 7

MODELE ECONOMIQUE

Dans ce chapitre, la manière dont les modèles économiques issus du marché réel peuvent servir de base pour la gestion des ressources d'une grille de calcul [67] sera présentée. Le fonctionnement général de ces modèles sera étudié. Enfin la mise en place de la facturation dans AROMA sera expliquée.

7.1 Introduction

Dans un contexte de *grid Computing*, où clients et serveurs ont chacun leurs propres objectifs et stratégies, il est nécessaire de mettre en place des politiques de gestion des ressources qui permettent de contenter chacune des parties. Les approches traditionnelles dans ce domaine ne peuvent pas être employées. Elles utilisent des politiques de gestion centralisées qui nécessitent une connaissance de l'état de l'ensemble de la grille. Dans le cadre du *grid computing* il est impossible de détenir de telles informations.

Une approche basée sur les modèles économiques du marché réel peut être employée dans le cadre du calcul sur grille. Ces modèles économiques constitueront une métaphore pour la gestion des ressources et le placement d'applications, permettant ainsi de réguler l'offre et la demande au niveau des ressources de la grille tout en garantissant une certaine qualité de service aux clients.

L'objectif des modèles économiques est de permettre à la fois aux producteurs (les propriétaires des ressources) et aux consommateurs (les utilisateurs des ressources) d'atteindre leurs objectifs. Ils doivent ainsi inciter les producteurs à partager leurs ressources, en leur fournissant une contrepartie, et également inciter les consommateurs à réfléchir aux compromis entre temps de calcul et coût dépendant de la qualité de service désirée.

Ainsi, un gestionnaire de ressources basé sur l'utilisation des modèles économiques aura les avantages suivants :

- il incitera les possesseurs de ressources à autoriser leur utilisation durant leurs périodes d'inactivité,
- il permettra la régulation de l'offre et de la demande pour les ressources,
- il supprime le besoin d'un agent central durant les phases de négociation,

- il offre un traitement uniforme pour toutes les ressources, quelle que soit leur nature,
- il permet le développement de politiques d'ordonnancement basées sur les utilisateurs, et non sur le système,
- il place le pouvoir de décision dans les mains des producteurs et des consommateurs, qui sont libres d'effectuer leurs propres choix pour optimiser leurs profits respectifs.

7.2 Modèles économiques pour la gestion des ressources d'une grille de calcul

7.2.1 Fonctionnement général

Les modèles économiques offrent un moyen de gérer les ressources d'une grille, en mettant en relation leurs propriétaires et leurs utilisateurs potentiels. Les producteurs seront rémunérés à chaque fois qu'un consommateur utilise leurs ressources. Pour cela, il existe deux grands modèles : un modèle basé sur l'échange, et un autre sur les prix. Dans le premier modèle, à chaque fois qu'une ressource est employée, son propriétaire acquiert des droits de consommation. Il pourra ainsi, lorsqu'il en aura besoin, utiliser à son tour les ressources de la grille. Cette façon de voir les choses suppose que tous les participants possèdent des ressources, ce qui n'est pas toujours le cas. Le second modèle permet aux consommateurs d'utiliser des ressources même s'ils n'en possèdent pas. Ils devront alors s'acquitter d'un prix à payer dépendant de la ressource utilisée.

Il est à noter que producteurs et consommateurs ont chacun leurs propres objectifs. Les propriétaires des ressources cherchent à maximiser leurs bénéfices, tandis que les utilisateurs désirent ne pas payer le prix fort, mais plutôt négocier un coût dépendant des caractéristiques de l'opération à effectuer, de leur budget, et de la qualité de service requise. L'établissement du prix d'une ressource dépendra aussi de l'offre et de la demande. Ainsi, dans cette approche, il est important de remarquer que chaque consommateur est en compétition avec les autres pour obtenir le droit d'utiliser une ressource, et chaque producteur est lui aussi en compétition avec les autres afin de remporter le marché.

Une grille utilisant les modèles économiques pour la gestion de ses ressources doit comprendre les éléments suivants :

- un annuaire d'information répertoriant les différentes entités de la grille,
- des modèles permettant d'établir la valeur d'une ressource,
- des mécanismes générant le prix des ressources et les publiant auprès de l'annuaire d'information,
- des protocoles de négociation entre producteurs et consommateurs,
- des agents intermédiaires, agissant en tant qu'agents de régulation, chargés d'établir la valeur des ressources à partir de modèles de prix, de gérer les échanges monétaires, et de gérer au mieux les situations de crise,
- des mécanismes de facturation et de paiement,
- des systèmes d'ordonnancement garantissant le respect de la qualité de service demandée pour les applications qui leur sont confiées.

Pour répondre aux spécifications ci-dessus, plusieurs entités doivent être mises en place :

- un courtier, ou *Grid Resource Broker* (GRB). Il agit comme un médiateur entre l'utilisateur et les ressources. Il est responsable :
  - de la découverte et de la sélection des ressources,
  - du transport des données et des programmes,
  - d'initier les exécutions sur les ressources distantes,
  - de retourner les résultats aux utilisateurs,
  - de s'adapter aux changements au niveau des ressources de la grille,
  - de présenter la grille à l'utilisateur comme étant une seule ressource unifiée.

  Il est à noter que le courtier représente l'utilisateur. Il connaît donc parfaitement ses objectifs, et tente de les satisfaire au mieux lors des négociations avec les fournisseurs de ressources.
- un annuaire, ou *Grid Market Directory* (GMD). Il permet aux propriétaires de ressources de publier leurs tarifs afin d'attirer des clients.
- un serveur d'échange, ou *Grid Trade Server* (GTS). Il s'agit d'un agent associé à un possesseur de ressources, et qui est chargé de négocier avec les clients dans le but de vendre l'accès aux ressources.
- un système de comptabilité. Il est responsable de suivre l'usage des ressources, et de facturer le client selon l'accord préalablement conclu entre le GRB et le GTS. Chaque producteur peut posséder son propre système de comptabilité, ou bien il peut en exister un global (*Grid Bank*).

7.2.2 Les différents modèles économiques existants

Les modèles économiques que nous décrirons successivement sont au nombre de cinq :

- le modèle de marché,
- le modèle des négociations,
- le modèle de l'appel d'offres,
- le modèle de la vente aux enchères,
- le modèle du partage proportionnel des ressources.

Modèle de marché

Dans ce modèle, les fournisseurs de ressources fixent un prix, et les utilisateurs sont facturés à ce prix, proportionnellement à la quantité de ressources consommées. Le prix décidé par les propriétaires de ressources peut être fixe, ou bien varier au cours du temps selon la loi de l'offre et de la demande.

La figure 7.1 illustre le principe de ce modèle.

1. Chaque fournisseur fixe un prix pour ses ressources et le publie auprès de l'annuaire.
2. Le courtier contacte l'annuaire pour obtenir la liste des fournisseurs avec leurs prix.
3. Le courtier identifie le fournisseur qui satisfait au mieux les objectifs du client, et lui confie l'application à exécuter.
4. Le fournisseur facture le client selon la quantité de ressources consommées.

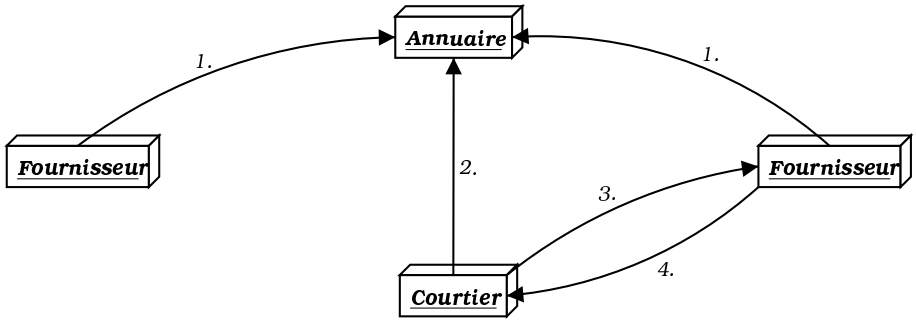


FIG. 7.1: Principe du modèle de marché

Ce modèle peut être étendu par un système de publication d'offres promotionnelles ayant pour but de fidéliser les clients ou bien d'en attirer de nouveaux.

Modèle des négociations

Ce modèle permet à l'utilisateur, contrairement au précédent, d'agir sur le prix des ressources qu'il désire consommer. Ici, producteurs et consommateurs auront leurs propres objectifs, et devront négocier pour les atteindre.

La figure 7.2 illustre le principe d'un tel modèle :

1. Chaque fournisseur s'enregistre auprès de l'annuaire.
2. Le courtier contacte l'annuaire pour obtenir la liste des fournisseurs.
3. Le courtier négocie avec un fournisseur pour obtenir un prix acceptable, le courtier commençant avec un prix faible, et le fournisseur avec un prix fort.
4. Si les deux parties se mettent d'accord sur un prix, l'application est confiée au fournisseur qui pourra ensuite facturer le client. Si aucun terrain d'entente ne peut être trouvé, le courtier s'adresse à un autre fournisseur, et entame une nouvelle négociation.

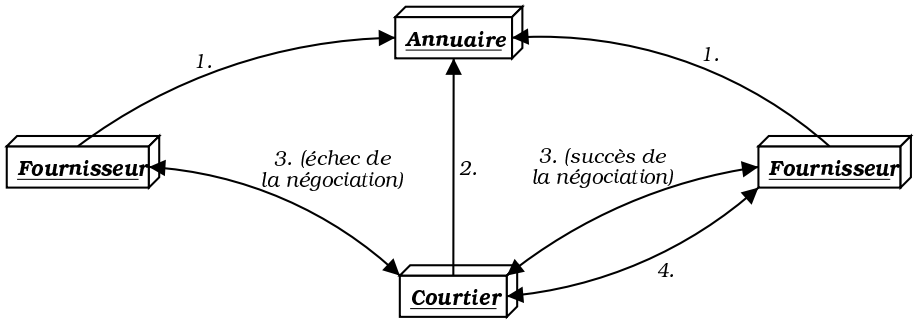


FIG. 7.2: Principe du modèle des négociations

Il est à noter que, grâce à cette approche, un courtier peut prendre des risques pour négocier des prix très bas, et éliminer les ressources trop chères. Ceci peut aboutir à une sous-utilisation

des ressources, obligeant ainsi les fournisseurs à réduire leurs tarifs plutôt que de gaspiller des cycles d'utilisation de leurs ressources.

Modèle de l'appel d'offres

Le modèle de l'appel d'offres est inspiré des mécanismes mis en place dans le monde des affaires pour gouverner les échanges de biens et de services. Il permet au client de trouver le fournisseur de service approprié pour travailler sur une tâche donnée. Dans ce contexte, une nomenclature particulière est adoptée. Le client (ou le courtier qui le représente) est appelé « manager », tandis que chaque fournisseur est un
« contractant » potentiel.

La figure 7.3 illustre le principe de ce modèle :

1. Chaque fournisseur s'enregistre auprès de l'annuaire.
2. Le manager contacte l'annuaire pour obtenir la liste des fournisseurs (contractants potentiels).
3. Le manager lance un appel d'offres auprès des contractants potentiels, en leur fournissant ses exigences.
4. Chaque contractant potentiel l'évalue, et lui soumet une offre s'il est intéressé.
5. Le manager choisit la meilleure offre et lui attribue le contrat.

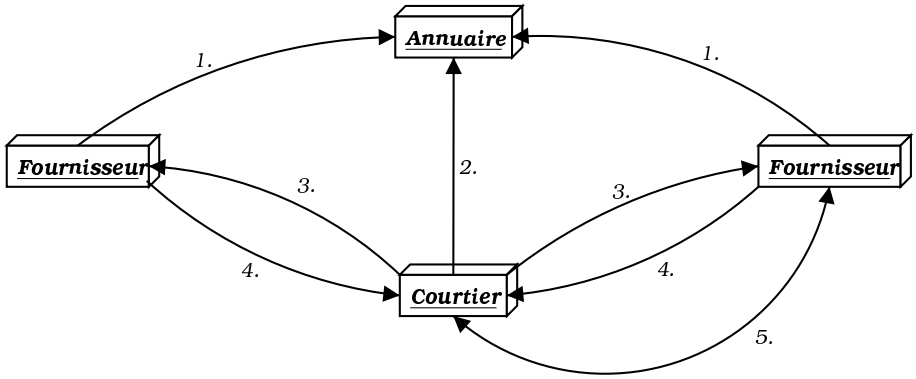


FIG. 7.3: Principe du modèle de l'appel d'offres

Modèle de la vente aux enchères

Ce modèle économique permet d'avoir une négociation entre un fournisseur et plusieurs consommateurs potentiels, qui permet d'aboutir à l'établissement d'un prix et au choix d'un consommateur. Une troisième entité entre également en jeu. Il s'agit du commissaire-priseur dont le rôle est de contrôler le bon déroulement de l'enchère.

La figure 7.4 illustre le principe du modèle des enchères.

1. Le fournisseur publie ses services auprès du commissaire-priseur et invite les utilisateurs à faire des offres.
2. Les différents courtiers font des offres, et peuvent surenchérir.
3. L'enchère s'arrête lorsque plus aucun courtier ne surenchérit. Si le prix minimum fixé par le fournisseur est atteint, la ressource est attribuée à l'utilisateur le plus offrant.

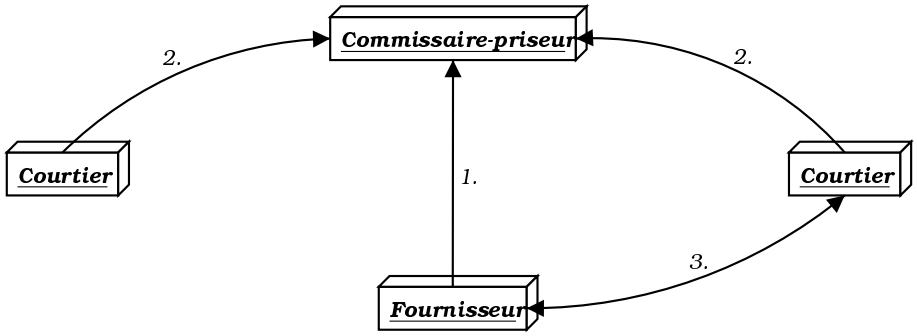


FIG. 7.4: Principe du modèle de la vente aux enchères

Si le type d'enchère décrit ci-dessus est le plus courant, il en existe trois variantes qui peuvent influer sur les stratégies à adopter de la part des courtiers :

- Chaque courtier soumet une et une seule offre sans connaître celles des autres. La meilleure offre l'emporte.
- Le commissaire-priseur part d'un prix très élevé, et le diminue progressivement jusqu'à ce qu'un courtier accepte le prix, ou bien jusqu'à ce que le prix atteigne une valeur seuil minimale.
- Les courtiers et les fournisseurs soumettent à tout moment des offres d'achat ou des ordres de vente. Si à un moment, il y a correspondance entre les deux, un accord est automatiquement passé entre les deux parties.

Modèle du partage proportionnel des ressources

Selon ce modèle, une ressource peut être partagée entre plusieurs utilisateurs. La part de ressource allouée à chaque utilisateur par rapport aux autres est alors proportionnelle à son offre par rapport à celles des autres utilisateurs. Ceci permet à tous les utilisateurs d'avoir accès aux ressources, même s'ils n'ont pas un budget important par rapport aux autres utilisateurs.

7.2.3 Etablissement des prix et mécanismes de paiement

Jusqu'à maintenant, nous avons vu comment des modèles économiques peuvent servir pour gérer au mieux les ressources constituant une grille. Dans la section précédente, nous avons détaillé les différents protocoles d'attribution d'une ressource à un utilisateur. Nous avons ainsi mis en avant deux aspects importants des modèles économiques : le prix des ressources (sur lequel repose leur attribution), et la facturation.

Dans cette section, nous allons étudier quelles sont les ressources qui peuvent être facturées, comment établir leur prix, et ensuite quels sont les mécanismes de paiement qui entrent en jeu lors de la facturation.

Ressources à facturer

Les ressources suivantes pourront être facturées à leurs utilisateurs :

- le temps processeur consommé,
- la quantité de mémoire consommée,
- la quantité de stockage utilisée,
- la bande-passante du réseau consommée,
- les signaux reçus et les changements de contexte,
- les logiciels et librairies utilisés.

Etablissement du prix d'une ressource

Selon le modèle économique que l'on désire appliquer à la gestion des ressources d'une grille, il sera nécessaire de leur donner un prix. Dans les modèles les plus simples, ce prix pourra être fixe. Mais ceci ne reflète pas la réalité. En effet, la valeur d'une ressource variera toujours au cours du temps, et ce en fonction du contexte dans lequel elle se trouve à un instant donné.

Ainsi, le prix d'une ressource dépend des paramètres suivants :

- la durée d'utilisation,
- le moment d'utilisation (notion d'heures pleines et d'heures creuses),
- l'offre et la demande,
- la fidélité de l'utilisateur,
- l'existence d'un pré-contrat entre l'utilisateur et le fournisseur,
- la puissance de la ressource,
- son coût physique,
- la surcharge du service,
- la valeur perçue par l'utilisateur,
- les préférences locales du propriétaire.

Mécanismes de paiement

La facturation est un des aspects fondamentaux des modèles économiques, car c'est sur lui que repose la viabilité de ce système. La notion de paiement suppose qu'il existe une ou plusieurs entités capables de gérer les comptabilités des fournisseurs. Il peut en exister une par fournisseur (chaque fournisseur facture directement ses clients et gère les mécanismes de paiement), ou bien une banque centralisant les comptabilités de chaque fournisseur peut être mise en place.

Le paiement pour l'utilisation de ressources peut s'effectuer à plusieurs moments :

- paiement à l'avance : l'utilisateur achète au préalable des crédits qu'il peut ensuite dépenser pour utiliser les ressources.
- paiement après l'utilisation : le client est facturé conformément à sa consommation de ressources.
- paiement au fur et à mesure de la consommation.

7.3 Modèle économique simple pour la facturation d'AROMA

La facturation d'une application d'AROMA se fera selon les ressources suivantes :

- le temps processeur consommé : il s'agit de la somme du temps réel pendant lequel chaque tâche de l'application aura occupé un processeur,
- le nombre de processeurs utilisés,
- la puissance de chaque processeur utilisé.

Ainsi, selon le coût de chacune de ces ressources, différentes stratégies pourront être adoptées pour minimiser le coût global de l'application. Le problème consistera donc à choisir les machines (leur nombre et leur puissance), de telle sorte que le placement des tâches sur celles-ci offre un bon compromis entre le coût de l'application et sa durée d'exécution.

7.3.1 Définition des variables

Dans cette section, les variables permettant de traiter correctement notre problème sont présentées. Elles sont réparties en deux grandes catégories : les données du problème, et les inconnues, ces dernières fournissant l'ordonnancement trouvé.

Les données du problème

P^R = puissance du processeur de référence. Lorsque l'utilisateur effectue une requête, il doit renseigner la durée totale de l'application, en prenant pour référence cette puissance.

T^R = temps de référence (temps global de l'application demandé par l'utilisateur).

A = nombre de tâches de l'application.

N^P = nombre de processeurs disponibles.

N^P_{min} = borne minimale imposée par l'utilisateur sur le nombre de processeurs à utiliser.

N^P_{max} = borne maximale imposée par l'utilisateur sur le nombre de processeurs à utiliser.

$P(j)_{1 \leq j \leq N^P}$ = puissance de chaque processeur disponible.

$Ch(j)_{1 \leq j \leq N^P}$ = charge de chaque processeur.

C^T = coût correspondant au temps processeur consommé.

C^N = coût correspondant au nombre de processeurs utilisés.

C^P = coût correspondant à la puissance des processeurs utilisés.

$C^P_{coef}(j)_{1 \leq j \leq N^P}$ = coefficient appliqué sur le coût de chaque processeur. Ce coefficient permet de modifier localement le coût d'un processeur particulier.

Les inconnues

$Taches(i,j)_{1 \leq i \leq A, 1 \leq j \leq N^P}$ = matrice associant chaque tâche de l'application à un des processeurs disponibles. Le résultat de l'ordonnancement est donc contenu dans cette matrice. Il s'agit ainsi de l'inconnue principale du problème. Toutes les inconnues suivantes sont déduites de cette matrice.

$U(j)_{1 \leq j \leq N^P}$ = vecteur indiquant, pour chaque processeur, s'il est utilisé ou non.

N = nombre de processeurs utilisés.

$y(j)_{1 \leq j \leq N^P}$ = variables d'écart entrant en jeu dans le calcul du vecteur U. Elles permettent d'avoir des résultats entiers dans ce vecteur.

$Temps(j)_{1 \leq j \leq N^P}$ = temps cumulé occupé par l'application sur chaque processeur. Il s'agit du temps passé réellement par les tâches sur chaque processeur (correspondant à la différence entre la date de fin de la dernière des tâches et la date de début de la première). Il dépend donc de la puissance de ces derniers, ainsi que de leur charge.

C_{App} = coût de l'application.

T_{App} = durée d'exécution de l'application.

Fonction objectif

La fonction objectif est un compromis entre le coût de l'application et sa durée d'exécution. Ces deux paramètres sont pondérés par des coefficients ajustables selon ce qu'il est souhaité de privilégier. La fonction objectif est donc la suivante :

$$Objectif = PoidsCout \times C_{App} + PoidsDuree \times T_{App}$$

Ainsi, il faudra calculer le contenu de la matrice $Taches$ de manière à minimiser cette fonction objectif.

Dans le cas d'une application composée de 5 tâches ($A = 5$). L'utilisateur indique que cette application est exécutée en 100 secondes sur un processeur de puissance 200 ($T^R = 100$ et $P^R = 200$). En prenant pour hypothèse que chaque tâche de l'application est identique, leur durée sur le processeur de référence est de $T^R/A = 100/5 = 20$.

Pour exécuter cette tâche, 6 processeurs sont disponibles ($N^P = 6$). L'utilisateur spécifie qu'il veut en utiliser entre 3 et 5 ($N^P_{min} = 3$ et $N^P_{max} = 5$). La puissance et la charge des processeurs seront contenues dans les vecteurs suivants :

$$P = \begin{bmatrix} 200 \\ 200 \\ 400 \\ 600 \\ 600 \\ 800 \end{bmatrix} \quad Ch = \begin{bmatrix} 0.12 \\ 0.80 \\ 0.51 \\ 0.33 \\ 0.01 \\ 0.99 \end{bmatrix}$$

La charge d'un processeur s'exprime en pourcentage de processeur occupé.

Après minimisation de la fonction objectif dépendant de ces paramètres, des valeurs pour les inconnues seront obtenues. Tout d'abord, le contenu de la matrice $Taches$ sera connu. Elle sera remplie comme suit :

- $Taches(i, j) = 1$ si la tâche i est placée sur le processeur j,
- $Taches(i, j) = 0$ sinon.

Par exemple,

$$Taches = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Le processeur 1 est associé à la tâche 4, le processeur 3 à la tâche 1, et enfin que le processeur 4 exécute les tâches 2, 3 et 5.

Trois processeurs sont donc utilisés ($N = 3$). Le contenu du vecteur U peut être calculé, de la manière suivante :

- $U(j) = 1$ si le processeur j est utilisé,
- $U(j) = 0$ sinon.

Ainsi, on a :

$$U = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Le vecteur $Temps$, contenant la durée pendant laquelle les tâches de l'application auront occupé chaque processeur, sera calculé à partir du nombre de tâches attribué à chaque processeur, de la puissance de ces derniers ainsi que de leur charge. La formule sera donnée dans la section suivante. Cependant, $Temps(2) = Temps(5) = Temps(6) = 0$, puisqu'aucune tâche n'est associée à ces deux processeurs.

7.3.2 Modèle mathématique

Dans cette section, toutes les équations et contraintes régissant le modèle économique seront établis. Tout d'abord des contraintes sur le contenu de la matrice $Taches$ seront posées. Ensuite, les équations permettant de calculer le vecteur U ainsi que le vecteur $Temps$ à partir de la matrice $Taches$ seront précisées. Enfin, les formules permettant de calculer le coût global de l'application ainsi que sa durée d'exécution seront établies.

Contraintes sur la matrice $Taches$

$$\forall i \in \{1, ..., A\} \qquad \forall j \in \{1, ..., N^P\} \qquad Taches(i,j) \in \{0,1\}$$

$$\forall i \in \{1, ..., A\} \qquad \sum_{j=1}^{N^P} Taches(i,j) = 1$$

Calcul du vecteur U

$$\forall j \in \{1, ..., N^P\} \qquad U(j) \in \{0,1\}$$

$$\forall j \in \{1, ..., N^P\} \qquad U(j) = \Big[\big(\sum_{i=1}^{A} Taches(i,j) \big) + y(j) \Big] / A$$

$$\forall j \in \{1, ..., N^P\} \qquad y(j) \in \mathbb{N} \qquad 0 \leq y(j) \leq A - 1$$

$U(j)$ est un entier égal à 1 si la ligne j de la matrice $Taches$ contient au moins un 1, ou égal à 0 sinon. Ainsi, une variable d'écart $y(j)$ est introduite dans le calcul de $U(j)$. Cette variable sera ajustée de telle sorte que $U(j)$ soit entier. On aura ainsi :

- $y(j) = 0$ si $\sum_{i=1}^{A} Taches(i,j) = 0 \qquad (\Rightarrow U(j) = 0)$,
- $y(j) = A - \sum_{i=1}^{A} Taches(i,j)$ sinon $\qquad (\Rightarrow U(j) = 1)$.

Calcul de N

$$N = \sum_{j=1}^{N^P} U(j)$$

$$0 \leq N^P_{min} \leq N \leq N^P_{max} \leq N^P$$

$$N^P_{max} \leq A$$

Plusieurs contraintes sur le nombre de processeurs utilisés entrent en jeu. Ce nombre ne doit pas franchir les bornes fixées par l'utilisateur. D'autre part, ces bornes doivent respecter les limites physiques du système. De plus, l'utilisateur ne peut pas demander plus de processeurs que ce qu'il y a de tâches à exécuter, puisque, dans ce cas-là, des processeurs ne seraient pas utilisés.

Calcul du vecteur $Temps$

Soit T le temps processeur consommé par une tâche sur un processeur de puissance P, on a :

$$T \cdot P = \frac{T^R . P^R}{A} \qquad \Rightarrow \qquad T = \frac{T^R . P^R}{A} \cdot \frac{1}{P}$$

Soit T_i le temps processeur consommé par la tâche i (lorsqu'elle est active), on a :

$$\forall i \in \{1, ..., A\} \qquad T_i = \frac{T^R . P^R}{A} \cdot \sum_{j=1}^{N^P} \frac{Taches(i,j)}{P(j)}$$

Soit T_i^* le temps occupé par la tâche i sur un processeur (qu'elle soit active ou non), on a :

$$\forall i \in \{1, ..., A\} \qquad T_i^* = \frac{T^R . P^R}{A} \cdot \sum_{j=1}^{N^P} \frac{Taches(i,j)}{P(j) \cdot \left(1 - Ch(j)\right)}$$

Le temps pendant lequel l'application occupera ainsi chaque processeur est déduit (que les tâches soient actives ou non) :

$$\forall j \in \{1, ..., N^P\} \qquad Temps(j) = \frac{T^R . P^R}{A} \cdot \sum_{i=1}^{A} \frac{Taches(i,j)}{P(j) \cdot \left(1 - Ch(j)\right)}$$

Calcul du coût de l'application

$$C_{App} = C^T \cdot \frac{T^R . P^R}{A} \cdot \sum_{i=1}^{A} \sum_{j=1}^{N^P} \frac{Taches(i,j)}{P(j)}$$

$$+ C^N \cdot N$$

$$+ C^P \cdot \sum_{i=1}^{A} \sum_{j=1}^{N^P} Taches(i,j) \cdot P(j) \cdot C_{coeff}^P(j)$$

Le coût de l'application dépend du temps processeur consommé par chaque tâche, du nombre de processeurs utilisés, ainsi que de leur puissance.

Calcul de la durée d'exécution de l'application

$$T_{App} = \max_{j=1,...,N^P} Temps(j)$$

7.3.3 Résultats

Le modèle économique a été défini, son comportement va être testé. Pour cela, un exemple d'application et une liste de processeurs seront pris. La fonction objectif sera minimisée pour différentes valeurs des paramètres du modèle. Pour effectuer cette minimisation, le logiciel Xpress sera utilisé.

Définition de l'exemple traité

Une application de vingt tâches ($A = 20$) à placer sur trente processeurs ($N^P = 30$) est considérée. La puissance de chacun des processeurs est donnée dans le vecteur suivant :

$$P = \begin{bmatrix} 100 \\ 200 \\ 300 \\ \vdots \\ 2800 \\ 2900 \\ 3000 \end{bmatrix}$$

La puissance du processeur de référence est de 100 ($P^R = 100$). Le temps global de l'application demandé par l'utilisateur sur le processeur de référence est de 200000 ($T^R = 200000$). L'utilisateur demande à utiliser entre 4 et 20 processeurs ($N^P_{min} = 4$ et $N^P_{max} = 20$).

Minimisation de la durée d'exécution de l'application

Dans un premier temps, la durée d'exécution de l'application sera minimisée, en trouvant l'ordonnancement optimal permettant d'exécuter les tâches au plus vite, quel que soit le prix à payer ($PoidsCout = 0$ et $PoidsDuree = 1$). L'étude est faite avec les paramètres suivants :

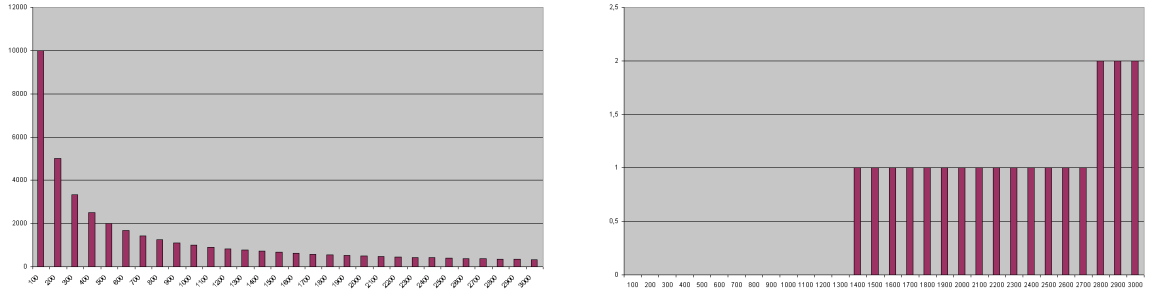
$$C^T = 1 \qquad C^N = 1 \qquad C^P = 50$$

La figure 7.5 montre la répartition des tâches sur les différents processeurs afin d'obtenir un temps d'exécution optimal. Cette étude a été réalisée en supposant que la charge de chacun des processeurs était nulle.

Ce sont les processeurs les plus rapides qui sont utilisés. De plus, le nombre de processeurs mis en jeu est assez important, et ce afin de ne pas les surcharger pour qu'ils puissent exécuter les tâches en un minimum de temps.

Les valeurs suivantes sont obtenues :

$$T_{app} = 714,286 \qquad C_{app} = 2314201,1 \qquad N = 17$$



(a) Temps d'exécution d'une tâche en fonction de la puissance des processeurs

(b) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.5: Minimisation de la durée d'exécution de l'application

La valeur de T_{app} est la durée minimale d'exécution de l'application. Il est totalement impossible, en conservant la configuration du système, de la faire diminuer. Le coût de l'application semble être très élevé.

La durée d'exécution de l'application est maintenant minimisée dans le cas où les processeurs ont une charge initiale (figure 7.6).

L'ordonnancement est affecté par la charge des machines. Même si leur puissance est élevée, aucune tâche n'est placée sur les machines très chargées. Au contraire, ce sont les machines peu chargées qui sont privilégiées, même si elles sont moins puissantes.

Les valeurs obtenues sont :

$$T_{app} = 1360,544 \qquad C_{app} = 2140599,2 \qquad N = 13$$

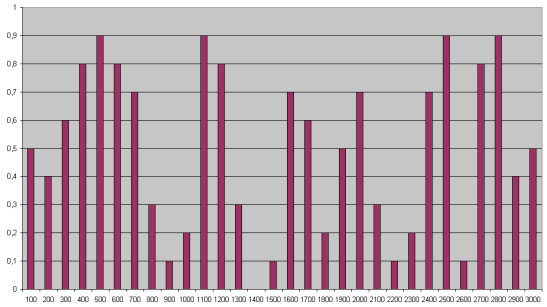
La durée d'exécution de l'application a augmenté avec ce changement d'ordonnancement, ce qui était prévisible. Quant au coût de l'application, il reste très élevé.

Minimisation du coût de l'application

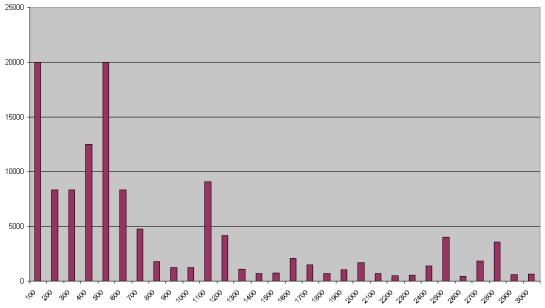
L'approche inverse est maintenant suivie, à savoir minimiser le coût de l'application sans tenir compte de sa durée d'exécution ($PoidsCout = 1$ et $PoidsDuree = 0$).

La charge initiale des différents processeurs est considérée nulle. Les mêmes paramètres de coût que précédemment sont considérés, c'est-à-dire :

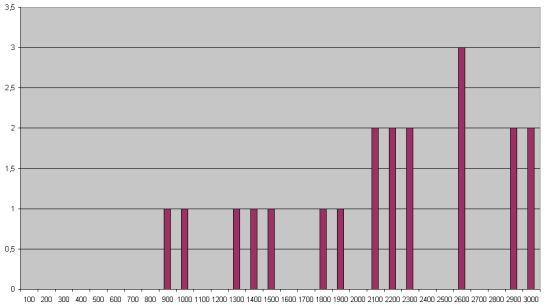
$$C^T = 1 \qquad C^N = 1 \qquad C^P = 50$$



(a) Charge de chaque processeur



(b) Temps d'exécution d'une tâche pour chaque processeur (prise en compte de la charge)



(c) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.6: Minimisation de la durée d'exécution de l'application avec prise en compte de la charge des processeurs

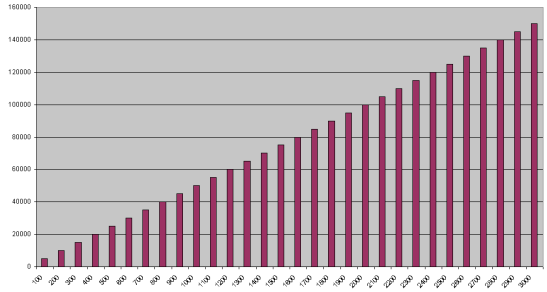
Ces paramètres indiquent que les processeurs de forte puissance sont extrêmement coûteux. La notion des coefficients locaux applicables au coût de chacune des machines n'est pas encore introduite.

La figure 7.7 montre l'ordonnancement choisi pour ces paramètres.

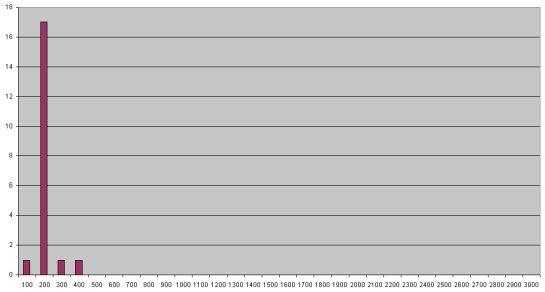
Contrairement à ce qu'il se passait précédemment, ce sont les machines de faible puissance qui sont choisies. Ceci prouve que le coût lié à la puissance est pris en compte pour l'ordonnancement des tâches de l'application.

Désormais les valeurs suivantes sont obtenues :

$$T_{app} = 85000 \qquad C_{app} = 310837,3 \qquad N = 4$$



(a) Coût de chaque processeur (proportionnel à la puissance)

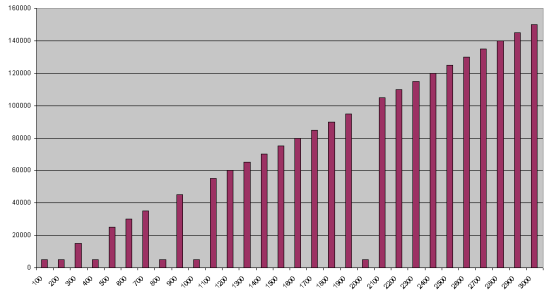


(b) Ordonnancement choisi (nombre de tâches par processeur)

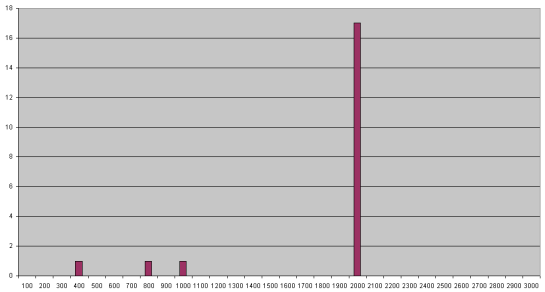
FIG. 7.7: Minimisation du coût de l'application (influence de la puissance des processeurs)

Le coût de l'application a considérablement baissé. Le prix minimal pour les paramètres de coût donnés est atteint. La durée d'exécution de l'application a, quant à elle, extrêmement augmenté. Ceci traduit le fait que l'ordonnanceur n'en tient pas du tout compte pour effectuer le placement de tâches. Il est également à noter que seulement quatre processeurs sont utilisés. Ceci est normal car le nombre de processeurs utilisés est payant, amenant ainsi l'ordonnanceur à en utiliser le moins possible ($N = N^P_{min}$).

La figure 7.8 démontre que, si les mêmes paramètres de coût sont conservés, mais que localement le coût de certaines machines est modifié, ce sont les machines les moins chères qui sont privilégiées.



(a) Coût de chaque processeur (prise en compte des coefficients locaux)



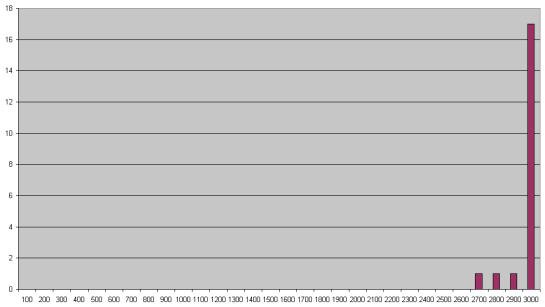
(b) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.8: Minimisation du coût de l'application (utilisation de coefficients de coût locaux)

Maintenant les paramètres de coût sont modifiés, en mettant l'accent sur le prix du temps processeur consommé par les tâches :

$C^T = 50 \qquad C^N = 1 \qquad C^P = 1$

La figure 7.9 donne l'ordonnancement choisi dans un tel cas.



(a) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.9: Minimisation du coût de l'application (influence du temps processeur consommé)

Dans ce cas, ce sont les processeurs de puissance élevée qui sont choisis, et ce en vue de minimiser le temps processeur consommé, et ainsi le coût global de l'application. Seulement quatre processeurs sont utilisés, car le but n'est pas de minimiser la durée globale d'exécution de l'application, mais seulement le temps pendant lequel les tâches, prises individuellement, occupent activement un processeur.

Les valeurs suivantes sont obtenues :

$T_{app} = 5666,661 \qquad C_{app} = 396354,9 \qquad N = 4$

En les comparant avec les résultats précédents, il peut être noté que le coût de l'application conserve le même ordre de grandeur. Il est minimal par rapport aux paramètres de coût qui viennent d'être appliqués. Il est à noter que la durée d'exécution de l'application a considérablement chuté. Elle n'a cependant pas atteint sa valeur minimale.

Minimisation d'un compromis entre coût et durée d'exécution

Maintenant le coût de l'application et sa durée d'exécution vont être minimisés. Les poids de ces deux valeurs dans la fonction objectif sont ajustés.

Les paramètres de coût utilisés sont les suivants :

$C^T = 1 \qquad C^N = 1 \qquad C^P = 50$

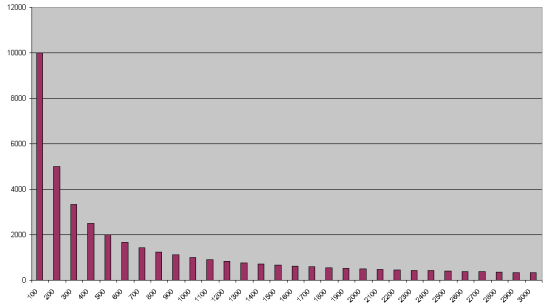
Pour de tels paramètres, il y a :

$$C_{app_{min}} = 310837,3 \qquad T_{app_{min}} = 714,286$$

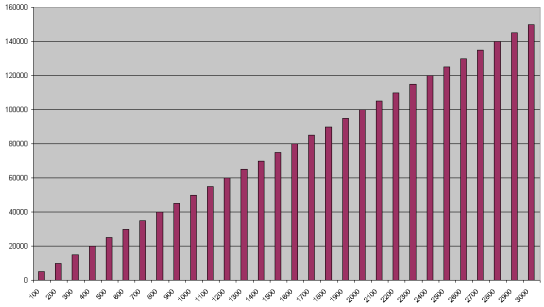
Pour trouver un bon compromis entre C_{app} et T_{app}, il faut multiplier T_{app} par environ 500. Ces deux valeurs auront ainsi le même ordre de grandeur. Il y a donc :

$$PoidsCout = 1 \qquad PoidsDuree = 500$$

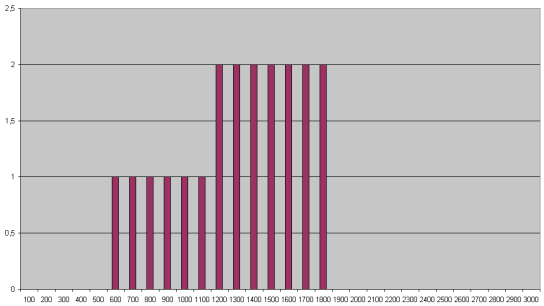
En prenant le cas où chaque processeur a une charge initiale nulle, et où aucun coefficient n'est appliqué sur le coût d'un processeur en particulier, la situation représentée par la figure 7.10 est obtenue.



(a) Temps d'exécution d'une tâche pour chaque processeur



(b) Coût de chaque processeur



(c) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.10: Minimisation d'un compromis entre coût et durée d'exécution

Le compromis choisi se répercute immédiatement sur les machines élues par l'ordonnanceur. En effet, il s'agit de machines comprises dans un domaine de puissances intermédiaires, permettant d'avoir une certaine rapidité d'exécution pour un coût assez faible.

Dans ce cas, les valeurs obtenues sont :

$$T_{app} = 1666.67 \qquad C_{app} = 1321882.981 \qquad N = 13$$

Ici, ni le coût de l'application, ni sa durée d'exécution ne sont optimaux. Ceci est conforme aux résultats attendus, un bon compromis entre les deux était souhaité. De plus, le besoin de rapidité d'exécution force l'ordonnanceur à utiliser plus de processeurs, tout en essayant de ne pas faire trop augmenter le coût.

Pour compléter l'étude, le cas de la minimisation d'un compromis entre le coût et la durée d'exécution de l'application est pris, mais en prenant en compte la charge initiale de chaque machine, et les préférences locales en matière de prix d'un processeur.

La figure 7.11 représente une telle situation.

L'ordonnanceur doit composer avec la charge des machines dans le but de placer les tâches tout en minimisant la fonction objectif. Ainsi, les processeurs sont choisis en fonction de leur puissance, de leur charge et de leur coût.

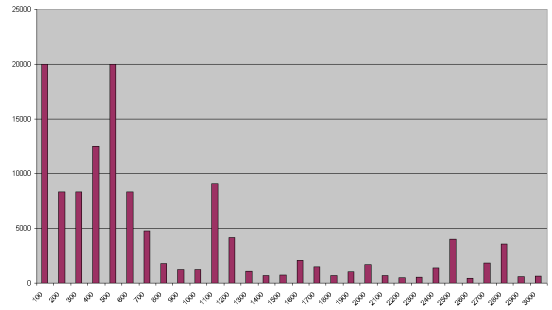
Les valeurs obtenues alors sont les suivantes :

$$T_{app} = 2197.8 \qquad C_{app} = 783117 \qquad N = 9$$

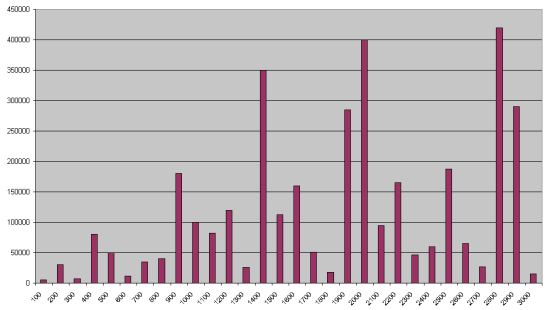
La durée d'exécution de l'application est plus élevée que dans le cas précédent, ce qui est expliqué par la charge des machines. Cependant, le coût de l'application a baissé. Ceci est normal puisque des « réductions » ont été appliquées sur le prix de certaines machines, par le biais des coefficients locaux.

Grâce au logiciel Xpress, le comportement d'un ordonnanceur basé sur le modèle économique a été simulé.

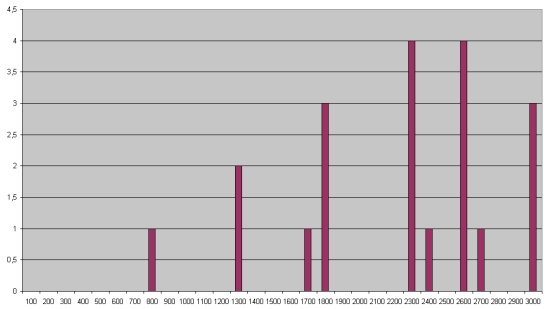
Ainsi il a été observé qu'un tel ordonnanceur possède le comportement attendu. Si la minimisation du coût de l'application est à privilégier, ce sont les machines les moins chères qui sont choisies. Si au contraire, la rapidité d'exécution est à mettre en avant, alors ce sont les machines les plus puissantes et les moins chargées qui sont choisies pour y placer des tâches. Enfin, dans une situation de compromis, l'ordonnanceur doit choisir des processeurs assez puissants, peu chargés, et bien sûr dont le prix reste raisonnable.



(a) Temps d'exécution d'une tâche pour chaque processeur (prise en compte de la charge)



(b) Coût de chaque processeur (prise en compte des coefficients locaux)



(c) Ordonnancement choisi (nombre de tâches par processeur)

FIG. 7.11: Minimisation d'un compromis entre coût et durée d'exécution (cas complet)

7.3.4 Mise en œuvre

Avant de pouvoir facturer une application, il est indispensable de connaître les paramètres de coût à appliquer. Pour cela, ils sont stockés dans une base de données.

Deux types de paramètres sont enregistrés :

- les paramètres communs à toutes les machines (C^T, C^N et C^P),
- les paramètres spécifiques à chacune des machines (C^P_{coeff}).

L'ordonnanceur permet de récupérer les ressources consommées par une application, les facturer, et communiquer un prix au client ; c'est-à-dire :

- le nombre de machines utilisées,
- la puissance de ces machines.

Toutes les autres informations seront fournies par l'utilisateur lorsqu'il donnera les caractéristiques de l'application à exécuter. Quant aux paramètres de coût, ils seront extraits de la base de données.

Ainsi, dans cette partie l'étude des résultats fournis par l'ordonnanceur va démontrer que ces résultats sont conformes aux attentes.

Des tests sur quatre machines seront effectués($N^P = 4$). Ces machines possèdent toutes la même puissance, qui correspond à la puissance de référence :

$$\forall j \in \{1, ..., N^P\} \qquad P(j) = P^R = 502$$

Chaque machine aura son propre coefficient permettant d'ajuster localement le coût lié à la puissance de son processeur. Le tableau suivant donne la valeur de ce coefficient pour chacune des machines :

Nom de la machine	C^P_{coeff}
vaughan	1
manitas	2
fender	5
patricia	10

La valeur choisie pour les paramètres de coût communs à toutes les machines :

$$C^T = 1 \qquad C^N = 1 \qquad C^P = 50$$

Plusieurs applications sont lancées, avec divers paramètres, le coût de chacune d'entre elles est relevé. Les résultats de ces tests sont donnés dans le tableau suivant :

Numéro du test	*Nombre total de tâches*	*Durée d'une tâche*	*Hôtes utilisés*	*Nombre de tâches sur chaque hôte*	*Coût de l'application*
1	1	140	vaughan	1	25241
2	1	140	fender	1	125641
3	1	20	fender	1	125521
4	2	140	fender patricia	1 1	376782
5	5	140	fender patricia manitas vaughan	2 1 1 1	578004

En comparant les deux premiers tests, il est observé que le coefficient appliqué localement à chaque machine est bien pris en compte. En effet, ce coefficient vaut 1 pour *vaughan* et 5 pour *fender*. Ce rapport se retrouve au niveau du prix.

Les tests 2 et 3 montrent qu'en utilisant la même machine, et en faisant varier le temps d'exécution de chaque tâche, le coût ne varie que très sensiblement. Deux conclusions peuvent être tirées : d'une part, le temps d'exécution est bien facturé (sinon, le coût ne varierait pas du tout), et d'autre part, son importance dans le prix à payer est faible (ce qui est prévisible, puisque le coefficient lié au temps d'exécution est très petit devant celui lié à la puissance des processeurs).

Enfin, les derniers tests montrent que lorsque plusieurs machines sont utilisées, chacune d'entre elles est bien facturée au client.

7.4 Conclusion

Dans cette section, l'ordonnanceur d'Aroma prend en compte les aspects liés à la facturation. Les tests effectués permettent de conclure que cet ordonnanceur facture le client conformément au modèle économique établi.

Cependant, cet ordonnanceur ne cherche absolument pas à minimiser le coût d'une application. L'algorithme utilisé permet simplement de garantir à l'utilisateur une certaine qualité de service. Si plusieurs ordonnancements satisfaisant cette qualité de service sont possibles, l'algorithme ne choisit pas forcément celui qui a le coût le plus faible. De même, il reste à traiter l'aspect temporel (heure creuse, heure pleine) et le mélange de plusieurs applications.

Chapitre 8

CONCLUSION

8.1 Bilan

Cette thèse a étudié le placement d'applications avec qualité de service sur des supports de type clusters ou grilles de calcul.

L'étude d'un modèle d'accès aux ressources (processeurs, réseaux) a été faite en étudiant différents systèmes d'exploitation (chapitre 3) et différents protocoles de communication (chapitre 6). Deux modèles d'exécution des processus ont été proposés : en considérant une charge stochastique (chapitre 4) et en considérant une charge déterministe (chapitre 5).

Ces deux modélisations peuvent être utilisées dans les algorithmes de placement proposés. En effet, dans sa thèse Thierry Monteil [30] explique comment obtenir les paramètres du modèle stochastique des machines.

Deux algorithmes de placement ont été expliqués : au niveau domaine (chapitre 5) et au niveau grille (chapitre 6). Ils garantissent la qualité de service demandée par l'utilisateur. Les communications entre les tâches sont prises en compte grâce au modèle d'application proposé au chapitre 6 : une application est découpée en plusieurs sous-groupes et des tâches réseau virtuelles sont introduites.

Enfin, le chapitre 7 propose un modèle économique simple de manière à facturer le client en fonction des ressources utilisées.

La figure 8.1 résume les apports de cette thèse.
Toute l'orginalité de l'étude réside dans l'introduction de la qualité de service et les nouveaux modèles de machines dans le cas d'une charge stochastique.

D'après les tests, les performances de l'algorithme sont très bonnes au niveau domaine (au maximum 250 ms avec qualité de service). Au niveau grille elles sont dégradées (18s avec qualité de service). Cela reste raisonnable dans la mesure où les applications pour lesquelles une exécution sur une grille de calcul est intéressante sont des applications demandant un long temps de calcul (sur 6h d'exécution, il est possible d'attendre 18s pour le placement). Le temps nécessaire au placemen s'explique par l'ensemble des calculs qui sont nécessaires pour

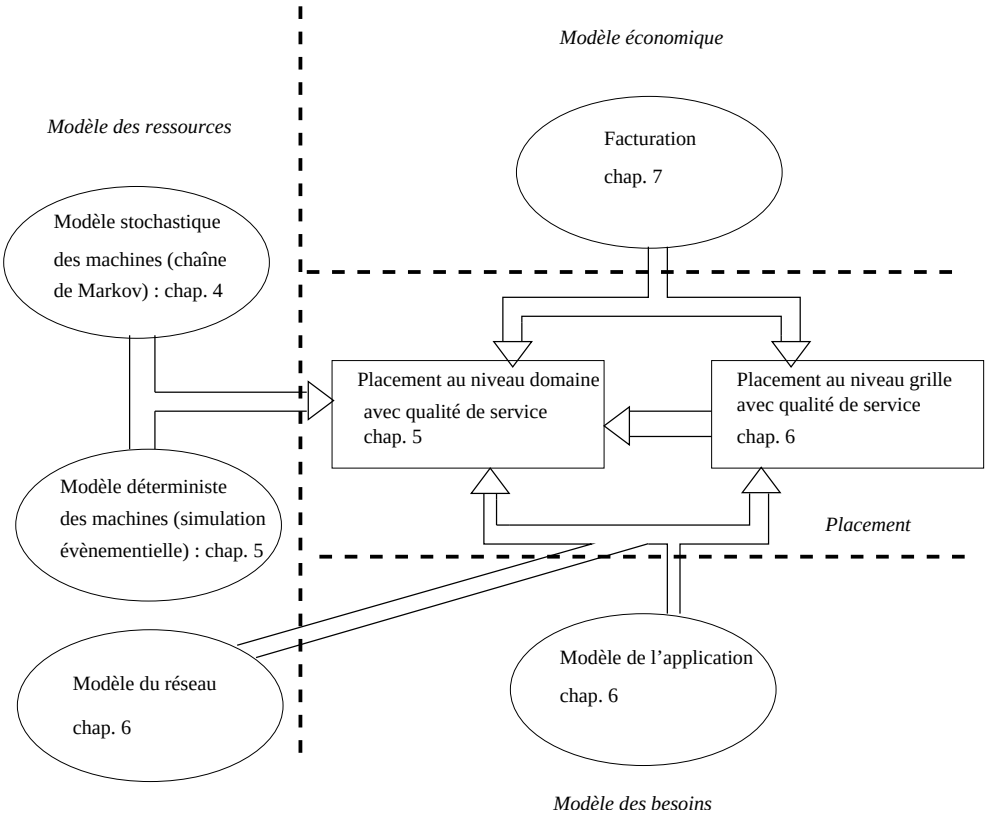


FIG. 8.1: Résumé de l'étude pour la gestion des ressources d'une grille.

étudier le placement et vérifier les contraintes. Au niveau de la grille plus de machines sont considérées, plusieurs domaines sont étudiés et toutes les applications en cours d'exécution susceptibles d'être perturbées par le placement en cours d'étude sont ré-étudiées.

8.2 Perspectives

En perspectives, les performances de l'algorithme de placement au niveau grille peuvent être améliorées. Il pourrait être intéressant d'étudier une meilleure approximation pour la charge des domaines (une analyse sur un quart d'heure pour la première heure puis toutes les heures entraînant des imprécisions de placement).

Un modèle réseau plus précis pourrait aussi être utilisé toutefois il devra rester simpliste pour ne pas dégrader les performances.

Des validations de l'algorithme au niveau grille seraient également intéressantes à faire ; elles permettraient d'étudier un plus grand nombre d'applications et une architecture de grille plus conséquente. Les tests réalisés ici ayant été limités par l'utilisation du simulateur Simgrid.

Enfin, le modèle économique proposé restreint les ressources facturées aux processeurs. Un modèle économique plus complet, prenant en compte toutes les ressources pouvant être facturées telles que la mémoire ou bien les bibliothèques utilisées (cf. section 7.2.3), pourra être développé et inclus dans une future version de l'ordonnanceur.

Par conséquent, il sera nécessaire, par la suite, de créer un nouvel ordonnanceur qui aura pour rôle de respecter le modèle économique créé. Ainsi, il devra être capable de trouver un ordonnancement permettant de minimiser un compromis entre le coût d'une application et sa durée d'exécution, tout en respectant une qualité de service donnée.

Bibliographie

[1] Marcel Soberman. *Les grilles informatiques.* Hermes, 2003.

[2] I.Foster, C. Kesselman.. *The Grid : Blueprint for a new computing infrastructure.* Morgan Kaufmann, 1998.

[3] *Extension d'outils d'observation pour fournir les données pertinentes sur les ressources.* Livrable 2 Projet CASP, sous-projet 1 : Administration, gestion et observation des ressources du cluster, Mai 2003.

[4] P.Pascal, S.Richard, T. Monteil *Architecture of a Grid Resource Manager.* PDPTA02, International Conference on Parallel and Distributed Processing Techniques and Applications, vol 4, pp. 2086-2092, Las Vegas USA, 2002.

[5] P. Bacquet O. Brun J.M. Garcia T. Monteil P. Pascal S. Richard *Telecommunication Network Modeling and Planning Tool on ASP Clusters.* ICCS03, Melbourne Australie,2003.

[6] T. Monteil, J.M. Garcia *Task Allocation Strategies on Workstations Using Processor Load Prediction.* PDPTA97, International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 416-421, Las Vegas USA, 1997

[7] Java Secure Socket Extension. *http ://java.sun.com/products/jsse/*

[8] Java Authentication and Authorization Service. *http ://java.sun.com/products/jaas/*

[9] *http ://www.itworld.com/Comp/2378/UnixInsider*

[10] *http ://techpubs.sgi.com :80/library/dynaweb_bin/ebt-bin/0650/nph-infosrch.cgi/infosrchtpl/SGI_Developer/OrOn2_PfTune/@InfoSearch_BookTextView/466;he=0*

[11] D.Ferrari, S.Zhou. *A Load Index for Dynamic Load Balancing.* Fall Joint Computer Conference, Dallas Texas, pp 684-690, november 1986.

[12] D.Gauchard. *Un Equilibrage de charge sur réseaux de stations de travail.* Rapport LAAS, 1998.

[13] W.T.C.Kramer, J.M. Craw. *Effective use of Cray supercomputers.* Supercomputing'89, pp.721-731, Nov 1989.

[14] *http ://www.myricom.com*

[15] M. Gerla, P. Palnati and S. Walton. *Multicasting Protocols for High-Speed, Wormhole-Routing Local Area Networks* ACM SIGCOMM'96, Palo Alto, CA, Aug. 1996.

[16] Charles L. Seitz and Wen-King Su. *A Family of Routing and Communication Chips Based on the Mosaic* Proceedings of 1993 Symposium on Research on Integrated Systems.

[17] Danny Cohen, Gregory Finn, Robert Felderman, Annette DeSchon. *ATOMIC : A Local Communication Network Created Through Repeated Application of Multicomputing Components* USC/ISI Technical Report ISI/RR-92-291, Sept 92.

[18] H. Chen, P.Wyckoff, K. Moor. *Simulation on Gigabit Ethernet and Myrinet using real application.* Technical report of Sandia National Laboratories.

[19] L. Prylli, B. Tourancheau. *BIP : A New Protocol designed for High-Performance Networking on Myrinet Simulation on Gigabit Ethernet and Myrinet using real application.* PC-NOW IPPS-SPDP'98, Orlando, USA, Mars 1998.

[20] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa. *PM : A High-Performance Communication Library for Multi-user Parallel Environments* Technical report TR-96015,Real World computing Parnership, Nov 96.

[21] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E.Kulawik, Charles L.Seitz, Jakov N. Seizovic, Wen-King Su. *Myrinet A gigabit per second Local area network* IEEE MICRO Février 1995.

[22] B.Baynat. *Théorie des files d'attente.* Hermes, 2000.

[23] J.M. Garcia, F. Le Gall, J.Bernussou. *A model for telephone networks and its use for routing optimization purposes.* IEEE journal on selected areas in communication, special issue on communication network peformance evaluation, vol 4, pp 966-974, Septembre 1986.

[24] Grid computing. *www.gridcomputing.com*

[25] A. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subholk, D. Sutherland. *ReMos : A resource monitoring system for network aware applications.* Carnegie Mellon School of Computer Science,CMU-CS-97-194.

[26] Rich Wolski. *Forecasting network performance to support dynamic scheduling using the Network Weather Service.* Proceedings 6th IEEE Symposium on High Performance Distributed Computing, Portland Oregon, 1997.

[27] Benjamin Gaidioz,Rich Wolski and Bernard Tourancheau. *Synchronising Network Probes to avoid measurement intrusiveness with the Network Weather Service.* Proceedings 9th IEEE International Symposium on High Performance Distributed Computing, HPDC'00. IEEE Computer society.

[28] Rich Wolski, Neil Spring et Jim Hayes. *Predicting the CPU availability of time-shared Unix systems on the computational grid.* Proceedings 8th IEEE International Symposium on High Performance Distributed Computing, HPDC'97, IEEE Computer Society.

[29] T. Monteil, J.M. Garcia. *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, Université P.M˙Curie Paris, pp87-90, Mai 1995.

[30] T. Monteil. *Etude de nouvelles approches pour les communications, l'observation et le placement de tâches dans l'environnement de programmation parallèle LANDA.* Thèse rapport LAAS n 96471, 1996.

[31] V.S. Sunderman, G.A. Geist, J. Dongarra, R. Manchek. *The PVM concurrrent computing system : evolution, experience, and trends.* Parallel computing 20, pp531-545, 1994.

[32] *MPI : A message passing interface standard* Mai 1994.

[33] K.M. Baumgartner et B.W. Wah. *Computer scheduling algorithms : past, present and future.* Information Sciences, vol 57 et 58, pp319-345, Elsevier Science Pub. Co., New York, NY, Sept-Dec 1991.

[34] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik et Parkson Wong. *Theory and Practice in parallel job scheduling.* Proceedings workshop Job Scheduling Strategies for Parallel Processing, Geneva, 1997.

[35] G. Bernard, B. Folliot. *Caractéristiques générales du placement dynamique :synthèse et problématique. Ecole placement dynamique et répartition de charge : application aux systèmes parallèles et répartis.* Presqu'île de Giens, pp115-124, Juillet 1996.

[36] R. Calinescu, D.J. Evans. *A parallel simulation model for load balancing in clustered distributed systems.* Parallel computing 20, pp 77-91, 1994.

[37] G. Muller, P. Sens. *Migration et points de reprise dans les systèmes distribués,Techniques de mise en oeuvre et mécanismes communs.* Ecole Placement dynamique et répartition de charge : application aux systèmes parallèles et répartis. Presqu'île de Giens, pp 67-87, Juillet 1996.

[38] E. G. Talbi. *Allocation dynamique de processus dans les systèmes distribués et parallèles : état de l'art.* Université des Sciences et technologies de Lille. Rapport 162, janvier 1995.

[39] T.L. Casavant, J.G. Kuhl. *Effects of response and stability on scheduling in distributed computing systems.* IEEE Transactions on software engineering, vol 14, N 11, pp 1578-1588, November 1988.

[40] Y.C. Chow, W.H. Kohler. *Models for dynamic load balancing in a heterogeneous multiple processor system.* IEEE Transactions on computers, vol c-28, N 5, pp 354-361, 1979.

[41] C.Y. Lee. *Parallel machines scheduling with nonsimultaneous machine available time.* Discrete applied mathematic North-Holland 30, pp 53-61, 1991.

[42] F. Bonomi, A. Kumar. *Adaptative optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler.* IEEE Transactions on computers, vol 39, N 10, pp 1232-1250, October, 1990.

[43] T.L. Casavant, J.G. Kuhl. *A taxinomy of scheduling in general-purpose distributed computing system.* IEEE Transactions on software engineering, vol 14, N 12, February 1988.

[44] J.M. Garcia, D. Gauchard, T. Monteil, O. Brun. *Process mapping given by processor and network dynamic load prediction.* 5th International Euro-Par Conference (Euro-Par'99), Toulouse, 1999. Lecture notes in computer science 1685, Euro-Par'99 Parallel Processing, pp 291-294.

[45] T. Monteil, P. Pascal. *Task allocation using processor load prediction on multiprocessor cluster.* NATO Advanced Research Workshop Advanced environments, Tools and Applications for Cluster Computing pp 126-135. IEEE Workshop, Mangalia, September 2001.

[46] Maui Scheduler. *http ://www.mhpcc.edu/maui*

[47] Jackson, D., Snell, Q., and Clemen, M. *Core algorithms of the maui scheduler.* In Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing (June 2001).

[48] Jingwen Wang, Songnian Zhou, Khalid Ahmed and Weihong Long. *LSBATCH : A distributed load sharing batch system.* Technical report CSRI-286, avril 93.

[49] PBS. *http ://www.openpbs.org*

[50] R. Henderson and D. Tweten. *Portable batch system : External reference specification.* Technical report, NASA, Ames Research Center, 1996.

[51] M.A. Baker, H.W. Yau. *Cluster Computing Review* NPAC Technical Report, SCCS-748, NPAC, Syracuse University, USA, October 1995.

[52] Mary Papakhian. *Comparing job management systems : the user's perspective.* IEEE Computational science engineering, 1998.

[53] PBSWeb. *http ://www.cs.ualberta.ca/%7Epinchak/PBSWeb*

[54] George Ma and Paul Lu. *PBSWeb :A Web-based Interface to the Portable Batch System* 12 IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), Las Vegas, Nevada, U.S.A., November 6-9, 2000.

[55] Brett Bode, David M. Halstead, Ricky Kendall and Zhou Lei. *The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters.* Proceedings of the 4th annual Linux showcase & conference, Atlanta, 2000.

[56] Warren Smith, Ian Foster and Valerie Taylor. *Scheduling with advanced reservations.* Proceedings of the IPDPS Conference, May 2000.

[57] Silver. *http ://www.supercluster.org*

[58] Dror G. Feitelson and Morris A. Jette. *Improved utilization and responsiveness with gang scheduling.* Proceedings JSSPP 1997 : 238-261. Job Scheduling Strategies for Parallel Processing, IPPS'97 Workshop, Geneva, Switerlang.

[59] Dmitry Zotkin and Peter J. Keleher. *Job-length estimation and performance in backfilling schedulers.* 8th Intl Symp. High Performance Distributed Comput. , Aug 1999.

[60] D. Lifka. *The ANL/IBM SP scheduling system.* Job Scheduling Strategies for parallel processing, pp 295-303, Springer-Verlag, 1995 (LNCS 949).

[61] J.Slovira, W.Chen, H. Zhou, D. Lifka. *The EASY-Loadleveler API project.* Job Scheduling Strategies for parallel processing, pp 41-47, Springer-Verlag, 1996 (LNCS 1162).

[62] C.L. Liu, J.W. Layland. *Scheduling algorithms for multi-programming in a hard real time environment.* Journal of the Association of Computer Machinery (ACM), Vol. 20, No. 1, Jan. 1973, pp. 46-61.

[63] G.C. Buttazzo, J.A. Stankovic. *Technical report.* UM-CS-1993-025. Comp. Science Department. University of Massachusetts, Amherst, July 1993.

[64] H.Chan, T. Lam, K. To. *Non-migratory online deadline scheduling on multiprocessors.* Proceedings of the filfteenth annual ACM-SIAM Symposium on discrete algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004. SIAM 2004.

[65] B. Kalyanasundaram, K. R. Pruhs. *Eliminating migration in multi-processor scheduling.* Journal of Algorithms, 38(1) :2-24,2001.

[66] Atsuko Takefusa, Satoshi Matsuoka, Henri Casanova, Francine Berman. *A study of deadline scheduling for client-server systems on the computational grid.* Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10) San Francisco, California, August 7-9, 2001.

[67] Rajkumar Buyya. *Economic-based distributed resource management and scheduling for grid computing.* Thesis, April 2002.

[68] Mary Papkhian. *Comparing job-management systems : the user's perspective.* IEEE Computational science engineering, 1998.

[69] E.Riedel. *CODINE 4.1.1* Genias Sofware GmbH, Neutraubling, Germany; http ://www.genias.de/geniaswelcome.html.

[70] J.Pasko. *Distributed queuing system 3.1.3 User Guide* Supercomputer Computations Research Inst., Florida State Univ., Mar. 1996.

[71] *LSF User Guide* 4eme édition, décembre 1996.

[72] *Licendes Program Specifications : IBM LoadLeveler Version 1 Release 3.0* IBM Document GH23-0040-03, Aug. 1996.

[73] M. Quinson. *Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué.* Thèse soutenue le 11/12/03.

[74] DIET. *http ://graal.ens-lyon.fr/ diet.*

[75] Legion. *http ://legion.virginia.edu*

[76] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. *Legion : The next logical step toward a nationwide virtual computer.* Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 08 1994.

[77] A.S. Grimshaw, Wm.A.Wulf, and the Legion team. *The legion vision of a worldwide virtual computer.* Communications of the ACM, 40(1), January 1997.

[78] Steve J. Chapin, Dimitrios Katramatos, John Karpovitch and Andrew S. Grimshaw. *The Legion resource management system.* Proceeedings Job Scheduling strategies for parallel processing workshop, IPPS/SPSP'99.

[79] Globus. *http ://wwwġlobus.org*

[80] Ian Foster and Carl Kesselman. *Globus : A metacomputing infrastructure toolkit.* International Journal of Supercomputer Applications, 11(2) :115-128, 1997.

[81] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman and Stuart Martin. *A resource management architecture for metacopmputing systems.* Proceedings 4th workshop on Job Scheduling Strategies for Parallel Processing, pp 62-82 Springer–Verlag LNCS 1459, 1998.

[82] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt and Alain Roy. *A distributed resource management architecture that supports advance reservations and co-allocation.* Proceedings of the International Workshop on Quality of Service, pp 27-36, 1999.

[83] Ian Foster, Alain Roy and Volker Sander. *A Quality of service architecture that combines resource reservation and application adaptation.* Proceedings of the 8th International Workshop on Quality of Service 2000.

[84] Sun GridEngine. *http ://www.gridengine.sunsource.net*

[85] Pascale Vicat-Blanc Primet et Philippe d'Anfray. *Les grilles haute performance et le projet etoile.* Matapli, (71), Mai 2003.

[86] Ganglia : distributed monitoring and execution system. *http ://ganglia.sourceorge.net*

[87] Map center : an open grid status visualization tool. *http ://mapcenter.in2p3.fr*

[88] Citrix Metaframe. *http ://www.citrix.com/products.*

[89] Microsoft. *http ://www.microsoft.com.*

[90] Ninf. *http ://ninf.etl.go.jp.*

[91] H.Nakada, H.Takagi, S.Matsuoka, U.Nagashima, M.Sato and S.Sekiguchi. *Utilizing the metaserver architecture in the Ninf global computing system.* In High-Performance Computing and Networking'98, LNCS 1401, pages 607-616, 1998.

[92] S.Sekiguchi, M.Sato, H.Nakada, S.Matsuoka, U.Nagashima. *Ninf : Network based information library for globally high performance computing.* In Proc. Parallel Object-Oriented Methods and Applications, pages 39-48, Santa Fe, 1996.

[93] NetSolve. *http ://icl.cs.utk.edu/netsolve.*

[94] H. Casanova and J. Dongarra. *NetSolve : a network enabled server for solving computational science problems.* The International Journal of Supercomputer Applications and H igh Performance Computing, 11(3) :212–223, 1997.

[95] D.Bear. *Principles of telecommunication-traffic engineering.* IEE Telecommunications series 2 1976.

[96] J.Walrand and P.Varaiya. *High Performance Communication networks.* Morgan Kaufmann Publishers San Francisco California 2000.

[97] Kleinrock L. and R.R.Muntz. *Processor-Sharing Queueing Models of Mixed Scheduling Disciplines for Time-Shared Systems.* Journal of the Association for Computing Machinery, 19, pp 464-482 1972.

[98] Kleinrock L. *Queueing Systems vol II :Computers applications* Wiley Interscience 1976.

[99] Gravey A., Louvion J.R, Boyer P. *On the Geo/D/1 and Geo/D/1/N queues* Performance Evaluation 11(2), 1990.

[100] A. Tannenbaum. Réseaux, architectures, protocoles, applications, InterEditions, pp.788-792, 1996.

[101] O. Brun, J.M. Garcia *Analytical solution of finite capacity M/D/1 queues* Journal of applied probability, vol 37, N.4, pp 1092-1098, December 2000.

[102] Patricia Pascal and Thierry Monteil. *Influence of Deterministic Customers in Time Sharing Scheduler.* ACM Operating Systems Review, 37(1) :34-45, January 2003.

[103] Patricia Pascal. *Gestion des ressources sur des grappes d'ordinateurs avec qualité de service garantie.* Proceedings 4ème Congrès Ecole Doctorale Système. Mai 2003, Toulouse.

[104] Patricia Pascal and Thierry Monteil. *Modélisation d'un réseau haut débit pour la gestion de ressources en calcul haute performance* Proceedings Journées Doctorales Informatique et Réseaux, pp 251-260, JDIR2002, Mars 2002, Toulouse France.

[105] http ://www.cs.huji.ac.il/labs/parallel/workload/logs.html#sdscsp2

[106] http ://joblog.npaci.edu/

[107] http ://www.cs.huji.ac.il/labs/parallel/workload/

[108] H. Casanova. *SimGrid : A Toolkit for the Simulation of Application Scheduling*

[109] http ://brassens.upmf-grenoble.fr/IMSS/mamass/graphecomp/kruskal.htm