Un mécanisme d'ordonnancement distribué de tâches temps réel

Leila Baccouche

THESE

présentée par

Leïla Baccouche

pour obtenir le grade de **DOCTEUR**
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité **Informatique**

Un Mécanisme d'Ordonnancement Distribué de Tâches Temps Réel

Date de soutenance : 22 Novembre 1995

Composition du jury :

Président : J. MOSSIERE

Rapporteurs : H. GUYENNET

Z. MAMMERI

Examinateur : T. MUNTEAN

M. SILLY

Thèse préparée au sein du Laboratoire de Génie Informatique

A ma mère et mon père pour

tout leur amour et leur soutien.

Je leur dois beaucoup.

A mon frère à qui je ne souhaite pas de faire une thèse.

A hichem…Merci Pour tout.

Et à tous ceux qui m'ont permis d'arriver à ce stade……

Remerciements

Je tiens à remercier très sincèrement Monsieur Jacques MOSSIERE, Professeur à l'INPG pour l'honneur qu'il me fait en étant président du jury de ma thèse. J'ai rencontré Mr MOSSIERE à Tunis lors d'une conférence, il m'a fait découvrir l'ENSIMAG et il m'a convaincue de faire mon doctorat à Grenoble.

Je voudrais temoigner de ma grande gratitude à Monsieur Zoubir MAMMERI, Maître de conférences à l'ENSAM qui a accepté de lire ma thèse en des délais très courts. Les discussions que nous avons eues ainsi que ses conseils très pertinents m'ont beaucoup aidée. Je remercie également Monsieur Hervé GUYENNET, Professeur à l'université de Franche-comté, d'avoir accepté d'être rapporteur de ma thèse. Je le remercie pour le temps qu'il m'a consacré et pour ses conseils.

Mes remerciements vont également à Madame Maryline SILLY, Maître de conférences à l'école centrale de Nantes, pour l'intérêt qu'elle a témoigné à mon travail et pour l'honneur qu'elle me fait en assistant à mon jury.

J'adresse également mes remerciements les plus sincères à Monsieur Traïan MUNTEAN, Professeur à l'université de la méditérannée qui a été mon directeur de thèse.

Je voudrais aussi remercier tous les membres et ex-membres de l'équipe Systèmes massivement parallèles pour leur aide et leur sympathie : Alba BALANIUK, Harold CASTRO, Robert DESPONS, Ahmed ELLEUCH, Leon MUGWANEZA, Irina SMARANDACHE, EL-Ghazali TALBI et Philippe WAILLE.

Une dernière pensée à tous ceux qui m'ont aidée aux derniers préparatifs. Je remercie spécialement Martine PERNICE notre ancienne secrétaire et Fethi BOUNAAS.

Résumé Dans le cadre du contrôle de l'exécution d'applications temps réel, un mécanisme d'ordonnancement de tâches basé sur le critère du temps est indispensable. Le mécanisme se doit de garantir en priorité les tâches périodiques et de maximiser le nombre de celles apériodiques.

Le mécanisme *d'ordonnancement distribué* que nous proposons, traite d'abord l'allocation statique des tâches temps réel, car les tâches périodiques doivent être allouées et ordonnancées avant l'exécution. Nous avons développé un algorithme d'allocation parallèle basé sur l'approche des algorithmes génétiques. Il permet d'obtenir des allocations correctes où le respect des contraintes temporelles qui portent sur les tâches est assuré, en effet l'ordonnancement des tâches est pris en compte lors de la construction du placement.

Dans le cas de systèmes temps réel souples, le mécanisme *d'ordonnancement distribué* met en œuvre deux algorithmes afin de gérer l'ordonnancement et l'allocation des tâches durant l'exécution. Le premier est un algorithme d'ordonnancement local en-ligne, simple et peu coûteux. Le second est un algorithme d'allocation dynamique, indépendant de la taille et de la topologie du réseau. Il se distingue par une heuristique visant à donner davantage de garantie aux tâches apériodiques par l'acceptation de celles-ci et par le transfert de tâches moins urgentes. Une réservation de l'emplacement des tâches sur le processeur désigné pour l'allocation permet de ne pas remettre en cause l'ordonnancement auparavant établi. La mise en œuvre de ces algorithmes dans le noyau du système ParX, nous a permis de montrer qu'un mécanisme d'ordonnancement distribué améliore les performances d'une application temps réel.

Mots clés : systèmes temps réel, ordonnancement et allocation dynamique de tâches, systèmes parallèles

Abstract In the context of the execution control of real-time applications, a scheduling mechanism with time-driven priorities is necessary. The mechanism must offer a great guarantee for the scheduling of periodic tasks and maximize the number of aperiodic ones.

We propose a *distributed scheduling mechanism* that first solves the static allocation of real-time tasks. Indeed, periodic tasks must have their resources reserved and be allocated statically. We developped a parallel genetic algorithm with builds correct allocations by taking into consideration tasks scheduling in the allocation phase.

For more flexibles systems, our mechanism proposes two algorithms for the execution control of real-time applications. The first is a local scheduling algorithm which is dynamic, not expensive and easy to implement. The second is a dynamic allocation algorithm which is independent of the size and the topology of the network. Its principal characteristic is a heuristic that offers more guarantee for critical aperiodic tasks. It proceeds by accepting the critical task and transfering other less critical tasks. A reservation of the task position on the chosen algorithm allows to guaranty the scheduling of the transfered tasks. The mechanism was implemented and integrated into ParX, kernel of the parallel operating system Paros. The results show that a distributed scheduling mechanism improves the performances of a real-time application.

Liste des figures

SOMMAIRE

PARTIE I

Présentation des systèmes temps réel et des approches de conception

Chapitre 1

Introduction

1.1 Problématique générale

Les systèmes temps réel ont depuis toujours acquis une importante place dans le développement des systèmes informatiques. Ils recouvrent un spectre d'applications très étendu qui va du simple contrôle de systèmes de commande, au contrôle du trafic aérien, des télécommunications, en passant par les systèmes de défense militaire, installations nucléaires et bien d'autres. Les systèmes temps réel sont des systèmes de traitement dont la validité est conditionnée non seulement par la correction des résultats mais surtout par les dates auxquelles ceux-ci sont délivrés. En effet, ces systèmes sont essentiellement caractérisés par des contraintes de temps sur les actions à entreprendre, qu'il faut respecter de manière plus ou moins critique.

Grand nombre d'applications temps réel nécessitent pour leur exécution, des systèmes très stricts d'une fiabilité incontestable car la moindre erreur dans le respect des contraintes temporelles peut entraîner la catastrophe. Pour ce faire, aussi bien la machine que son système d'exploitation doivent être adaptés à ce type d'applications. Ainsi, les systèmes qui supportent ce genre d'applications doivent offrir un maximum de *garantie* et de *déterminisme*.

(1) La *garantie* des systèmes temps réel est assurée par le respect des contraintes temporelles lors de l'exécution de l'application. Généralement, une tâche temps réel est décrite par trois paramètres : une durée d'exécution, une date au plus tôt à partir de laquelle la tâche peut être déclenchée et une échéance à ne pas dépasser. Le respect de

ces contraintes aurait été moins complexe s'il s'agissait de paramètres sur l'application dans sa globalité, on se serait alors intéressé à la réduction du temps de réponse de l'application. Cependant les contraintes des applications temps réel portent sur les tâches qui les composent, la garantie des contraintes devient ainsi ponctuelle et doit être assurée pour toutes les tâches de l'application.

(2) Le caractère *déterministe* des systèmes temps réel est imposé par l'enjeu des applications visées. On ne saurait se permettre à bord d'un avion piloté automatiquement de sortir le train d'atterrissage tantôt aussitôt l'ordre émis, tantôt quelques instants ou juste avant l'atterrissage parce que le système ce jour là, aura été occupé à une tâche de refroidissement des moteurs (par exemple). Pour qu'un système soit fiable, il doit être déterministe. Ainsi le comportement du système doit être déterminé et maintenu. Le déterminisme est très caractéristique des systèmes temps réel statiques, qui reproduisent assez souvent le même comportement.

Une solution qui s'impose afin d'assurer la *garantie* des contraintes et le *déterminisme* est l'ordonnancement des traitements. Un ordonnancement par priorités est nécessaire: on ne peut se contenter de partager équitablement la capacité du processeur entre les tâches. Généralement, par ordonnancement temps réel, on entend l'ordonnancement des tâches afin de pouvoir les exécuter dans les délais spécifiés par l'application. Toutefois la notion d'ordonnancement peut également intervenir à d'autres niveaux (communications, allocation, mémoire) afin de garantir l'exécution des tâches. Si une séquence d'exécution des tâches peut être déterminée à l'avance, il ne restera plus qu'à la reproduire lors de l'exécution réelle. Dans un environnement parallèle ou distribué, une séquence d'exécution sera sauvegardée sur chaque processeur ou nœud, ainsi que le code des tâches associées. C'est ainsi que procèdent les systèmes temps réel dits *statiques*. Tous les paramètres des tâches (temps d'exécution, ressources utilisées, communications...) sont connus a priori. Ces systèmes certes d'une fiabilité incontestable, sont très coûteux, utilisent mal les ressources et ils sont non flexibles.

Le coût excessif de ces systèmes est dû au fait que des analyses (simulations) intensives du pire des cas sont effectuées afin de déterminer les paramètres temporels des tâches. Le pire des cas consiste à supposer que toutes les occurrences des tâches et les imprévus possibles ont lieu. Un ordonnancement calculé dans ce cas de figure est fiable dans la mesure où on a tout prévu. Toutefois, ceci conduit à une mauvaise exploitation

des ressources car durant l'exécution, il s'avère souvent que les temps d'exécution sont inférieurs à ceux estimés et les occurrences des imprévus également.

Nécessité de systèmes temps réel dynamiques

Avec les demandes incessantes en applications temps réel plus souples, telles que les communications multimédia, les systèmes multirobots coopératifs ou autonomes, les bases de données temps réel que l'on trouve dans les systèmes de radars etc., la nouvelle génération de systèmes temps réel est *dynamique* plus *flexible* et concerne un spectre plus *étendu* d'applications. Ces systèmes plus souples sont à notre avis adaptés aux besoins de classes plus larges d'applications.

(1) le côté *dynamique* de ces systèmes permet d'ajuster et de recalculer les paramètres durant l'exécution. Il permet aussi de prendre en compte de manière efficace des imprévus qui peuvent surgir et entraîne une meilleure exploitation des ressources. Un modèle dynamique permet également, dans le cas où des tâches seront créées durant l'exécution, de ne pas surcharger les processeurs avec toutes les tâches mais de les créer au fur et à mesure, ce qui implique une utilisation plus optimale de la mémoire.

(2) un spectre plus *étendu* d'applications temps réel est concerné. En effet de nombreuses applications ont des paramètres difficiles à estimer et nécessitent la création de tâches lors de l'exécution (c'est le cas des applications multimédia entre autres).

(3) il peut également arriver dans les systèmes temps réel stricts, des situations où le système doit réagir à des imprévus en un temps très court. Certains cas de figure peuvent être prévus, tel l'exemple d'un accident de la circulation où la voiture est équipée d'un coussin gonflable de sécurité qu'il faudra gonfler quelques millisecondes après que le système ait détecté l'accident. La prise en compte de cet imprévu est triviale comparée à une déviation de trajectoire d'un avion à opérer immédiatement. En effet, ce dernier cas de figure ne peut être traité qu'une fois qu'il a lieu, alors que le premier est prévu. Ainsi, peut-on prévoir une solution à tous les cas de figure dans un système statique ?

L'énoncé du problème posé à présent est le suivant : *Comment préserver la notion de temps dans de pareils systèmes?* Les systèmes temps réel dynamiques sont des systèmes plus souples dans le sens où ils incluent en plus des tâches temps réel à garantir absolument, et qui sont généralement statiques et périodiques, des tâches apériodiques (correspondant à des imprévus) qui sont créées dynamiquement. L'ordonnancement de

ces dernières ne pourra se faire qu'à partir de la date de leur création, et ce en fonction des tâches auparavant ordonnancées. Certaines tâches apériodiques peuvent être critiques (comme les alarmes) et le système se doit de fournir les mécanismes qui garantissent également les contraintes de ce genre de tâches.

Cette thèse est consacrée à l'étude des systèmes temps réel dynamiques dans un environnement distribué. Nous tenons à préciser que le choix d'un environnement parallèle pour notre étude n'est pas uniquement dans un but d'accélérer l'exécution ou de bien exploiter d'éventuelles structures parallèles de l'application, ou même encore de céder à l'alternative attractive que représentent les nouvelles architectures parallèles. Les applications temps réel sont de part leur nature souvent implémentées sur des environnements distribués.

A l'origine, on trouve les trois raisons suivantes : (1) les systèmes temps réel doivent être tolérants aux pannes. Les approches tolérantes aux pannes utilisent souvent la duplication des tâches sur des processeurs différents, ainsi que la multiplication des ressources. (2) certaines applications nécessitent des traitements importants qui dépassent la capacité d'un seul processeur. (3) l'environnement qui commande le système temps réel peut être distribué (avec des capteurs distants). Le parallélisme apparaît ainsi comme une alternative attrayante pour tirer parti de cette distribution naturelle, et améliorer les performances de ces systèmes.

Plusieurs domaines de recherche sont concernés par les systèmes temps réel, depuis les spécifications et validation de ces systèmes au contrôle de l'exécution dans les noyaux des systèmes d'exploitation. Ce dernier aspect consiste en la conception de mécanismes qui assurent une exécution des applications temps réel avec un respect absolu des contraintes. Un contrôle est indispensable à différents niveaux. Cette phase commence par un placement judicieux des tâches sur les processeurs de manière à ce que l'application démarre avec des ensembles de tâches sur les processeurs dont les contraintes sont entièrement respectées. Cette étape est étroitement liée à l'ordonnancement des tâches surtout dans un modèle dynamique. A la différence des systèmes classiques non temps réel, l'allocation et l'ordonnancement ne peuvent être traités séparément pour des systèmes temps réel, même s'ils sont statiques[1]. Par la suite la garantie des contraintes doit être assurée tout au long de l'exécution au niveau de

[1] Nous expliquerons plus en détail au chapitre 2, le lien étroit qui lie l'allocation et l'ordonnancement des tâches temps réel.

l'ordonnancement et de l'allocation des tâches, des communications et de l'allocation de la mémoire, etc. Parmi l'ensemble de ces problèmes, nous nous sommes penchés sur le problème de l'allocation et de l'ordonnancement des tâches.

Dans un système parallèle ou distribué, les tâches désormais ne sont plus ordonnancées uniquement sur un processeur, mais sur tous les processeurs du réseau. Notre modèle autorisant la création dynamique de tâches, dès lors qu'une tâche est créée et que ses contraintes temporelles ne peuvent être garanties sur le processeur, une tentative d'ordonnancement par les autres processeurs est lancée. Cette action est complexe, pour cela le système d'exploitation doit supporter un mécanisme approprié afin de gérer l'ordonnancement des tâches durant l'exécution, la recherche de processeurs pour l'allocation de nouvelles tâches, ainsi que le respect des contraintes tout au long de l'exécution.

1.2 Apport de la thèse

Dans cette thèse, nous proposons un mécanisme pour l'allocation et l'ordonnancement d'un modèle dynamique de tâches temps réel dans un système parallèle. La distribution géographique des tâches et l'aspect dynamique font que l'ordonnancement des tâches sur l'ensemble des processeurs ne peut se faire sans avoir recours à l'allocation.

Le *mécanisme d'ordonnancement distribué* que nous proposons, est composé d'un allocateur statique pour le placement initial des tâches, d'un mécanisme d'ordonnancement local de tâches et d'un mécanisme pour l'allocation dynamique des tâches.

1.2.1 L'allocateur statique

Une caractéristique principale du mécanisme est qu'il doit être capable de résoudre les problèmes d'allocation et d'ordonnancement avant tout pour un modèle statique de tâches. Ainsi le mécanisme propose un allocateur statique de tâches. Il est chargé de la répartition des tâches entre les différents processeurs alloués à l'application. La répartition tient compte de plusieurs critères qui peuvent être stricts tels que le respect des contraintes temporelles ou le respect de la localité physique des ressources utilisées par les tâches. D'autres critères d'optimisation peuvent être pris en compte : on cite la

réduction des coûts de communication, ou l'équilibrage de la charge entre les processeurs. Pour le respect des contraintes strictes, l'allocateur statique opère en collaboration avec le mécanisme d'ordonnancement local afin de pouvoir démarrer l'application en garantissant les contraintes de temps sur chaque processeur alloué à l'application. Nous proposons une approche originale pour l'allocateur statique. Elle consiste à utiliser les algorithmes génétiques : des techniques qui sont inspirées de la théorie de l'évolution des espèces. Notre algorithme génétique est parallèle et adresse à la fois l'allocation et l'ordonnancement.

1.2.2 Le mécanisme d'ordonnancement local

Selon le modèle de tâches visé (statique ou dynamique), le mécanisme d'ordonnancement local applique les algorithmes adéquats afin de déterminer un ordre d'exécution entre les tâches. L'ordre d'arrivée des tâches est un facteur déterminant, car les tâches auparavant ordonnancées ne sont pas remises en cause à la création d'une nouvelle tâche. Elles sont simplement réordonnancées mais uniquement dans le cas où l'algorithme peut accepter la nouvelle tâche. Des phénomènes de blocages peuvent surgir si les tâches communiquent où partagent des ressources, l'algorithme d'ordonnancement doit être capable de les résoudre. Nous avons mis en œuvre un algorithme d'ordonnancement en-ligne (durant l'exécution) qui est peu sensible à la charge du processeur.

1.2.3 Le mécanisme d'allocation dynamique

L'approche que nous avons développée pour ce mécanisme se caractérise par : (1) des algorithmes dont la complexité est indépendante de la taille et de la topologie du réseau. (2) un algorithme d'allocation dynamique qui vise à donner autant de garantie aux tâches apériodiques qu'aux tâches périodiques.

L'efficacité du *mécanisme d'ordonnancement distribué* est donnée par la réduction des délais introduits lors de l'ordonnancement des tâches refusées et par la maximisation du nombre de tâches ordonnancées sur toute la machine. Etant dans un système temps réel souple, nous sommes confrontés à l'ordonnancement de tâches de différentes priorités qui ne doivent pas être traitées avec la même importance par le mécanisme. Certaines tâches sont autorisées à se terminer après leur échéances sans pour autant détériorer considérablement les performances de l'application (comme certaines tâches

apériodiques sans contraintes strictes). Le mécanisme devra être capable de choisir entre les tâches et de minimiser le nombre de tâches refusées.

1.3 Plan de la thèse

La partie I est une introduction aux systèmes temps réel et aux approches de conception de ces systèmes. Le chapitre 2 les définit et présente leurs caractéristiques. Par la suite, les différents modèles de tâches existants sont exposés. Il existe une multitude de modèles de tâches temps réel, selon les contraintes qui portent sur les tâches. Le chapitre termine par une classification des modèles d'allocation et d'ordonnancement existants. En effet, ceux-ci sont toujours dépendants du modèle de tâches étudié.

Le chapitre 3 est structuré en deux parties. Les premières sections exposent l'état de l'art des systèmes d'exploitation temps réel. Nous avons remarqué qu'il y a souvent confusion entre les systèmes temps réel et des systèmes plus rapides tels que les exécutifs temps réel, et certains noyaux pour systèmes embarqués. Les sections suivantes présentent la conception du *mécanisme d'ordonnancement distribué*. Il a été développé dans le cadre du système d'exploitation parallèle Paros et est intégré à son noyau ParX. Il est conçu en tant que machine virtuelle *temps réel* et offre les services de base permettant la conception d'applications temps réel.

Le reste de cette thèse met l'accent sur une distinction entre l'ordonnancement et l'allocation des tâches temps réel, d'un côté, et le modèle statique et dynamique d'un autre côté.

Les fondements de l'ordonnancement statique varient peu de ceux du dynamique, c'est pourquoi ils sont présentés ensemble. Cependant, en ce qui concerne l'allocation, nous présentons les aspects statique et dynamique séparément car la nature du problème à résoudre n'est plus la même, et les solutions également. Il ressort ainsi trois autres parties dans cette thèse.

La partie II concerne l'ordonnancement local. Le chapitre 4 présente les approches utilisées pour la résolution de l'ordonnancement ainsi que les algorithmes appliqués. Dans le chapitre 5, nous présentons un algorithme d'ordonnancement en-ligne qui peut

être appliqué à différents modèles de tâches : tâches périodiques et apériodiques communicantes, avec des relations de précédence...

La partie III est composée des chapitres 6 et 7 et est consacrée à l'allocation statique de tâches temps réel. Le premier chapitre présente l'état de l'art des méthodes existantes et réalisant une allocation qui satisfait les contraintes de temps. Peu de travaux ont effectivement traité ce genre de problème en mettant l'accent sur les contraintes de temps. Dans le chapitre 7, nous proposons un algorithme génétique qui réalise une allocation statique correcte selon les objectifs fixés.

La partie IV est consacrée à l'allocation dynamique. Le chapitres 8 présente les méthodes d'allocation dynamique dans les systèmes distribués et parallèles. Dans le chapitre 9, nous présentons le mécanisme d'allocation dynamique ainsi que les algorithmes que nous avons conçus.

Nous concluons enfin par un résumé des problèmes posés et traités dans cette thèse et nous présentons les perspectives.

Chapitre 2

Présentation des systèmes temps réel

Ce chapitre introduit dans un premier temps les principales caractéristiques des systèmes temps réel. Nous présentons par la suite, les différents modèles de tâches existants. De ces modèles découlent, les classes d'ordonnancement que nous exposerons à la fin de ce chapitre.

2.1 Définition des systèmes temps réel

2.1.1 Définition

On appelle classiquement une application temps réel un système de traitement de l'information ayant pour mission de commander un environnement, en respectant les contraintes de temps (temps de réponse à un stimulus, taux de perte d'information toléré en entrée ...) qui sont imposées à ses interfaces avec cet environnement .

Un système temps réel est un système qui est étroitement lié à son environnement. La propriété du temps réel est attribuée si l'exactitude du système est déterminée par les dates auxquelles les résultats d'exécution sont disponibles. La notion d'information valide ne devient vraie que si le résultat est correct et disponible dans l'intervalle de temps fixé par l'environnement.

Nous pouvons représenter un système temps réel comme étant composé de :

- L'environnement à contrôler,

- Un système de contrôle qui représente le système d'exploitation au dessus duquel l'application sera exécutée.

- l'application

L'interaction entre ces trois composants se traduit par un échange d'informations entre l'application et l'environnement selon des contraintes temporelles imposées par ce dernier. Le système d'exploitation permet de contrôler cette interaction.

2.1.2 Notion d'ordonnancement

Les systèmes temps réel sont conçus pour interagir avec l'environnement extérieur en garantissant le respect des contraintes pour que les réactions ou la capture des données arrivent à bon escient. Le non respect de ces contraintes par les tâches, considéré comme une faute grave du système, résulte le plus souvent de conflits survenus entre les tâches au moment de leur exécution. Ces conflits sont de deux ordres, ils portent sur les processeurs et les ressources :

• conflits sur les processeurs : Plusieurs tâches peuvent réclamer simultanément le processeur afin de respecter leur contraintes. Ces conflits peuvent être résolus par des mécanismes d'ordonnancement gérant l'accès au processeur. Les ordonnanceurs qui s'en chargent n'ont qu'une vision très générale des tâches car ils ne considèrent pas les traitements qu'elles effectuent mais utilisent des paramètres définis à l'avance sur les tâches tels que le temps d'exécution, l'échéance...Ordonnancer des tâches sur un processeur consiste à déterminer une séquence d'exécution des tâches sur le processeur qui garantisse leurs contraintes.

• conflits sur les autres ressources : Ceux-ci surviennent lorsqu'une tâche demande l'accès à une ressource ou donnée non partageable (unité d'entrée/sortie, fichiers, disques...). Des mécanismes de verrouillage peuvent être employés pour résoudre ces conflits. Toutefois, des mécanismes d'ordonnancement de ces ressources avec des considérations de temps doivent également être utilisés, afin d'éviter que pour des besoins d'intégrité de ressources, certaines tâches tardent à y avoir accès, et voient leurs contraintes temporelles violées.

Un mécanisme d'ordonnancement temps réel a pour rôle de mettre en œuvre l'exécution d'un ensemble de tâches sur un ensemble de ressources rattachées à un processeur. Dans un modèle dynamique, ceci s'accompagne d'une *gestion dans le temps*

où le mécanisme d'ordonnancement a pour charge de suivre cette exécution et de la modifier en fonction des précisions obtenues sur certains paramètres et des nouvelles tâches créées.

En univers distribué, un mécanisme d'ordonnancement a deux fonctions principales : une fonction de gestion locale et une de gestion globale. La première est la gestion temporelle. La seconde concerne les processeurs de la machine cible et consiste à allouer les tâches créées dynamiquement aux processeurs, c'est une *gestion spatiale*.

2.1.3 Support d'exécution d'applications temps réel

Afin d'assurer la garantie d'applications temps réel, nous ne pouvons nous contenter d'un système d'exploitation qui fournit un mécanisme d'ordonnancement temps réel. Le système d'exploitation doit être capable de prendre en compte cette notion de temps à tous les niveaux. Les prérequis suivants sont donc indispensables pour un système d'exploitation temps réel [Stan92a], [Stan92b] :

(1) Un modèle qui permet d'exprimer des contraintes temporelles pour toute entité du modèle de processus. Une tâche peut être exprimée en sous-tâches ou en threads (flots d'exécution). Les contraintes d'une tâche ne portent forcément pas de la même manière sur ses composants.

(2) Un langage qui permet une spécification claire de ces systèmes, garantit la conformité de l'exécution à la spécification indépendamment de l'architecture cible et prend en considération les environnement de communications asynchrones.

(3) Des politiques d'allocation et de placement de tâches sur les processeurs qui tiennent compte de contraintes de temps.

(4) Des politiques d'ordonnancement local des tâches au niveau de toutes les ressources rattachées au processeur.

(5) Des protocoles de communication permettant de prendre en compte les contraintes temporelles et de donner la priorité aux messages temps réel afin de garantir l'ordre d'exécution établi entre les tâches ainsi que leurs contraintes.

(6) Des protocoles pour la gestion de la mémoire. Quand on applique l'allocation des pages mémoire, il est nécessaire de privilégier les pages concernant le code de tâches temps réel.

(7) Des mécanismes de synchronisation et de maintien d'une horloge temps réel globale afin que le mécanisme d'allocation et d'ordonnancement utilise une horloge unique dans le système. Un des problèmes majeurs qui se posent dans un système distribué est la gestion d'un temps global. Les horloges des différents processeurs ont souvent un écart, pour cela le recours à des mécanismes de synchronisation est vivement souhaité.

2.2 Modélisation des systèmes temps réel

Une tâche temps réel est associée à des contraintes de temps et de ressources. Selon les contraintes et les caractéristiques des tâches, différents modèles de tâches ont vu le jour afin de répondre aux besoins des applications concernées.

2.2.1 Les contraintes temporelles

Dans un système temps réel, les contraintes temporelles portent essentiellement sur les dates de début et de fin d'exécution des tâches. Les contraintes temporelles sont modélisées par les paramètres suivants [ChMu91], [Chun89], [Herr90], [JaMo86], [LeLa93], [XuPa90]:

Pour chaque tâche T_i on considère :

A_i la date d'arrivée de la tâche au processeur (date de sa création ou éventuellement date à laquelle une tâche transférée par un autre processeur est reçue). Il est désormais possible d'ordonnancer cette tâche.

Suite à l'arrivée ou la création d'une tâche, le mécanisme d'ordonnancement, va rechercher un nouvel ordre prenant en compte la nouvelle tâche.

R_i la date à laquelle une tâche peut commencer son exécution. Certaines tâches sont dépendantes, et ne peuvent commencer leurs exécutions que si les tâches qui les précèdent ont terminé. Ceci car elles sont liées par des relations de précédence (communication d'un résultat en fin

d'exécution) ou de causalité (estampillage des tâches[1]). Souvent R_i et A_i sont confondues. Pour plus de précisions sur ce type de relations, on pourra se référer à [Bacc91], [ChMu91], [PeSh89], [XuPa90].

D_i la date au plus tard à laquelle une tâche peut terminer son exécution, aussi appelée échéance. On distingue les échéances strictes qui si dépassées peuvent sérieusement dégrader les performances de l'application, et les échéances relatives qui peuvent être dépassées (mieux vaut tard que jamais). Un mécanisme dynamique est autorisé à dépasser ces échéances relatives, si cela s'impose. Souvent une priorité utilisateur est rajoutée afin de dissocier une échéance stricte d'une échéance relative.

E_i la durée d'exécution de la tâche. E_i est déterminée par des simulations ou par une étude poussée du code source avant l'exécution. Parfois E_i est difficile à déterminer avant l'exécution (quand le code de la tâche est composé de boucles ou de traitements conditionnels). Le comportement non déterministe d'un pareil code fait que le nombre de boucles peut-être indéterminé, de même pour l'alternative choisie dans l'instruction "si". On détermine dans ce cas un temps d'exécution dit "au pire des cas" qui reflète une borne supérieure de E_i .

Dans un modèle dynamique, E_i ne peut être connu à l'avance puisque la tâche T_i est créée dynamiquement. Il est mis à jour au fur et à mesure de l'exécution de la tâche. On peut par exemple procéder par déduction du temps restant (E_{rest_i}) à partir du temps atteint (E_{att_i}) ;$E_i = E_{att_i} + E_{rest_i}$. Cette technique est appliquée dans [BrFi81], [HaSh89], [PuKo89]. Il existe plusieurs méthodes pour approximer le temps d'exécution d'une tâche, nous ne les citons que brièvement car ceci n'est pas l'objet de notre étude. Ainsi, on trouve Puschner et Koza [PuKo89], Kligerman et Stoyenko [KlSt86], [Stoy87], ainsi que Leinbaugh et Yamini [LeYa86] et d'autres.

P_i la période d'une tâche. Quand une tâche est périodique, elle a une occurrence toutes les P_i unités de temps.

[1] Un estampillage des tâches consiste à leur attribuer un numéro afin que durant l'exécution, cette numérotation soit respectée à travers le réseau.

La connaissance a priori des paramètres des tâches est non seulement assez difficile et coûteuse mais de plus elle entraîne une sous exploitation des ressources car ces paramètres sont calculés dans le pire des cas[2] et correspondent à des bornes supérieures. L'ordonnancement étant réalisé avec des valeurs au pire des cas, la fiabilité du système est assurée. Alors que durant l'exécution, ces paramètres oscillent entre une valeur minimale et la valeur extrême calculée et grand nombre de ressources se retrouvent libres pendant une grande période de temps.

Grâce aux paramètres R_i, E_i, D_i, le mécanisme d'ordonnancement détermine les paramètres suivants :

S_i la date à laquelle une tâche accède à la ressource selon l'algorithme d'ordonnancement déroulé.

C_i la date à laquelle une tâche termine son exécution. Cette date ne correspond pas forcément à $S_i + E_i$, en effet selon l'algorithme d'ordonnancement appliqué, une tâche peut être interrompue ou retardée pour la prise en compte d'une tâche plus urgente.

TR_i le temps de réponse de la tâche. C'est la période entre la date la plus antérieure à laquelle une tâche peut commencer et la date à laquelle elle termine son exécution. TR_i s'exprime par la différence entre C_i et R_i .

TL_i le temps de latence d'une tâche, période pendant laquelle une tâche peut être retardée sans que son exécution ne dépasse son échéance, elle s'exprime par $D_i - R_i - E_i$. La date au plus tard à laquelle la tâche peut commencer son exécution s'appelle la laxité L_i et s'exprime par $D_i - E_i$. Le temps de latence n'est pas constant, plus une tâche est retardée plus son temps de latence diminue.

U_i le taux d'utilisation du processeur. Il vaut E_i/P_i

L'ensemble de ces paramètres doit vérifier :

$A_i < R_i < S_i < C_i - E_i < D_i - E_i$.

La figure 2.1 ci-dessous illustre les paramètres temporels d'une tâche temps réel :

[2] cas où on suppose que toutes les formes possibles du code de la tâche ont lieu (le nombre d'itérations maximum atteint pour toutes les boucles...)

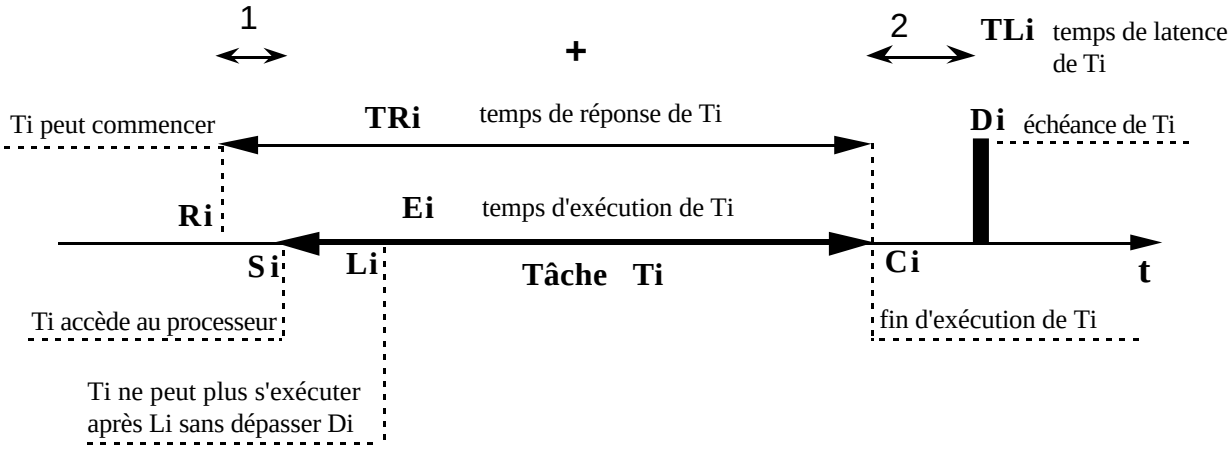


Figure n° 2.1 Les paramètres temporels d'une tâche

Dans certains systèmes temps réel, l'échéance est modélisée par une fonction à valeur dans le temps (FVT) appelée "time value function" (TVF) [Tome89], [ChMu91]. Chaque tâche T_i fournit au moment de sa terminaison, une contribution qui est décrite par une fonction coût $F_i(t)$. La valeur de cette fonction renseigne sur l'utilité de la terminaison de la tâche à l'instant t. $F_i(t)$ a la valeur maximale si T_i termine avant D_i, autrement la valeur de la fonction décroît vers 0, où elle signifie qu'un terminaison à cette date là est inacceptable. Quand la valeur de la fonction décroît brusquement après son échéance, ceci indique une échéance stricte à ne pas dépasser dans n'importe quelle situation. Par contre, si la valeur de la fonction décroît progressivement, cela indique une échéance relative qui peut éventuellement être dépassée de peu. Nous donnons un exemple de ces fonctions dans la figure 2.2.

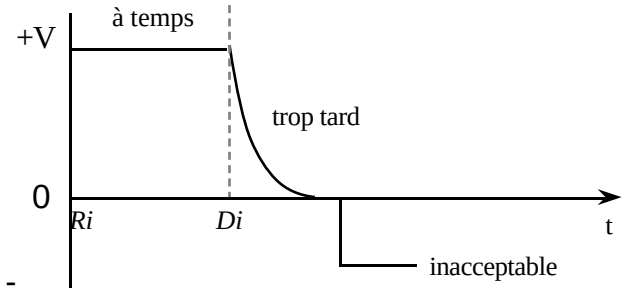


Figure n° 2.2 Terminaison d'une tâche mesurée avec une fonction à valeur dans le temps

L'ordonnancement des tâches, est ainsi ramené à un problème d'optimisation : la maximisation de la somme des contributions au temps d'accomplissement des tâches. L'ordonnancement est déterminé par une série d'évaluations des valeurs des FVT selon la position des tâches dans la séquence d'ordonnancement.

2.2.2 Caractéristiques des tâches

Cette section présente d'autres caractéristiques des tâches, qui peuvent influencer l'ordonnancement : la priorité utilisateur et la périodicité des tâches.

(1) Notion de priorité

Toutes les tâches temps réel n'ont pas la même importance vis à vis du système. On parle alors de priorité attribuée par le mécanisme d'ordonnancement, afin de distinguer les tâches entre elles. Cette priorité est définie par les échéances des tâches. Certaines ont des échéances très proches, leur exécution est par conséquent plus urgente, elles sont donc plus prioritaires que d'autres. Dans un système dynamique, il peut arriver que des tâches ne soient pas garanties, même si cette éventualité est rare. Dans ce cas le système doit être capable de choisir entre les tâches, afin de ne pas écarter des tâches très importantes. Pour ce faire, des priorités peuvent être définies par l'utilisateur. Nous distinguons trois types de tâches selon leurs priorités. Cette classification a été proposée pour le système Spring dans [StRa89] et a été reprise depuis dans de nombreux modèles de tâches :

Les tâches temps réel critiques

Le non respect des échéances pour ces tâches peut entraîner une catastrophe. Leurs contraintes doivent donc être rigoureusement respectées. Nous notons que leur nombre est très faible comparé à la totalité des tâches temps réel dans un système. Elles ont la priorité la plus haute. On observe dans la plupart des systèmes temps réel, un ordonnancement et une allocation statique de ces tâches (leur ressources sont réservées, en effet ces tâches ont leurs paramètres temporels connus avant l'exécution) [StRa89], [ToMe89], [LTCA89]. La réservation des ressources est une garantie supplémentaire nécessaire (mais non suffisante) que ces tâches ne dépasseront pas leur échéances. Cette garantie est non suffisante, car dans un modèle dynamique, des délais imprévus peuvent surgir tels qu'un blocage sur l'établissement d'une communication ou bien un cas de panne d'un processeur.

Les tâches temps réel essentielles

Nous caractérisons d'essentielles les tâches dont les contraintes sont non sévères et peuvent éventuellement être relâchées dans la mesure du possible. Dans certaines

applications, leur échéance est composée d'une échéance stricte et d'une échéance relative. La première partie de l'exécution d'une telle tâche doit être rigoureusement respectée, la seconde (qui peut consister en un affinement du résultat délivré par la première partie) peut ne pas être terminée. Dans ce cas le résultat ne sera pas très précis. Les tâches essentielles peuvent être ordonnancées dynamiquement.

Les tâches non essentielles

Ces tâches peuvent ne pas avoir de contraintes quant à la fin de leur exécution ou la date de leur déclenchement, elles ont donc la priorité minimale. Elles seront exécutées une fois que toutes les tâches à contraintes seront servies et selon leur ordre d'arrivée ou selon un ordre décidé par l'utilisateur.

(2) Notion de périodicité

Les tâches temps réel peuvent ne pas avoir de priorité utilisateur, auquel cas leurs priorités sont définies par leurs échéances. Toutefois, la notion de périodicité que nous introduisons révèle une autre distinction entre les tâches. Le mécanisme doit donc outre les priorités des tâches prendre en compte le fait qu'elles peuvent être : périodiques, apériodiques ou sporadiques.

Les tâches périodiques

Une tâche T_i est dite périodique de période P_i si elle est réexécutée chaque P_i unités de temps. Une telle tâche à ses paramètres R_i et D_i connus. Soit T_{in} la n^{ème} exécution de la tâche T_i , R_{in} la date au plus tôt est donnée par le taux d'interarrivée et D_{in} est déterminé par l'occurrence de la (n+1)^{ème} exécution de T_i (comme illustré dans la figure 2.3). Quand le système doit ordonnancer une tâche périodique, il doit être capable de garantir toutes les futures occurrences de la tâche.

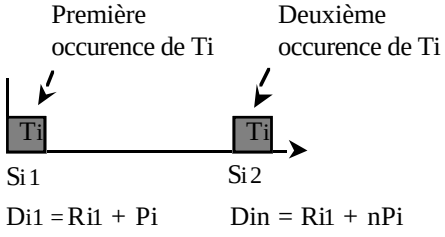


Figure n° 2.3 Notion de périodicité

Dans la plupart des systèmes, une occurrence de tâche périodique a son échéance égale à sa période. Nous précisons toutefois que dans certains modèles l'échéance peut être inférieure à la période [ChAg94].

Les tâches apériodiques

Leur arrivée au processeur suit une loi aléatoire (loi de poisson par exemple), leurs paramètres sont donc inconnus à l'avance. Ces tâches non périodiques sont généralement caractérisées par la moyenne des temps d'arrivée et un standard de déviation (par rapport au taux d'arrivée). Cependant si les occurrences d'une tâche apériodique sont au moins espacées de q unités de temps, on la caractérise de tâche sporadique de période q [ToMe89], [Herr90].

Nous verrons au chapitre 4 que les algorithmes d'ordonnancement sont extrêmement dépendants du modèle de tâches. Ainsi, ils sont conçus pour des tâches périodiques ou apériodiques, c'est pour cette raison que les tâches sporadiques sont en quelque sorte converties en tâches périodiques.

Afin de donner un aperçu sur les traitements que peuvent entreprendre ces deux classes de tâches, nous citons l'exemple d'un système de navigation composé des tâches périodiques suivantes : mise à jour de la position (0.9/2.5)[1] , mise à jour de la vitesse (4/40), émetteur de navigation (20/1000), et de certaines tâches apériodiques telles que : processeur de commande du tableau de bord, et des routines services d'interruption. Un exemple plus détaillé de ces tâches est présenté pour un chasseur F18 dans [ShGa90]. On voit clairement qu'une tache comme la mise à jour de la position est critique et qu'elle doit être exécutée périodiquement afin que la position affichée reflète la réalité.

Une application temps réel est en majorité composée de tâches périodiques. L'occurrence des tâches apériodiques est faible. Nous avons remarqué que les tâches périodiques sont souvent considérées critiques.

[1] Le premier nombre représente le temps d'exécution de la tâche et le second sa période.

2.3 Classification des modèles d'ordonnancement et d'allocation

Dans ce qui précède, nous avons montré la diversité des modèles de tâches existants. Diversité qui découle de celle des applications temps réel et de leur besoins. Tout naturellement, il s'en est suivi une diversité des mécanismes d'ordonnancement temps réel. Nous avons remarqué que dans la littérature, les termes ordonnancement et allocation concernant des domaines étroitement liés, l'allocation des tâches est souvent confondue avec l'ordonnancement des tâches et on parle alors d'ordonnancement local et distribué (ou global). Nous estimons que l'appellation *ordonnancement local* exprime parfaitement le partage du processeur entre les tâches. Cependant, en ce qui concerne *l'ordonnancement global ou distribué*, étant donné qu'il s'agit d'allouer les tâches refusées aux processeurs, nous pensons que le terme *allocation* dynamique convient mieux afin d'éviter toute confusion. Nous avons donc adopté ces termes là.

- **Ordonnancement de tâches** : détermination d'un ordre d'exécution des tâches selon les critères spécifiés. Quand cet ordre est déterminé avant le début de l'exécution, l'ordonnancement est dit statique.
- **Ordonnancement dynamique** : la séquence déterminée au préalable est mise à jour et réordonnancée en fonction des nouvelles tâches créées.
- **Allocation statique** : placement d'un ensemble de tâches sur un réseau de processeurs en respectant et optimisant certains critères.
- **Allocation dynamique** : placement des tâches créées durant l'exécution.

Nous définissons dans ce qui suit, les mécanismes et algorithmes d'ordonnancement :

- **Mécanisme d'ordonnancement local de tâches** (appelé également ordonnanceur) : ensemble de modules nécessaires pour la coordination de l'ordonnancement des tâches sur un processeur. Ils doivent assurer les fonctions suivantes :
  - la vérification de la possibilité d'exécution d'une tâche sur le processeur sans violer ses contraintes. L'énoncé du problème abordé ici est le suivant : *étant donné un ensemble de tâches dont on sait que les contraintes temporelles seront vérifiées, peut-on accepter une nouvelle tâche* ?

Hypothèse 2.1

On dit qu'une tâche est refusée localement si l'ordonnanceur local ne peut garantir ses contraintes avec l'ensemble des tâches préalablement ordonnancées.

• le calcul des priorités des tâches ainsi que leur ordre d'exécution et leurs dates de début et de fin d'exécution. Pour réaliser ceci divers algorithmes d'ordonnancement local existent.

• la mise à jour et l'adaptation (recalcul des priorités, réordonnancement) dans le cas de l'arrivée d'une nouvelle tâche.

En fonction du type d'applications pour lesquelles le système est conçu, et selon le modèle de tâches, chacune de ces trois étapes peut être ou ne pas être présente dans un système. De plus, les frontières entre ces étapes ne sont pas toujours claires (voir chap. 4).

On dit qu'un algorithme d'ordonnancement est un :

• **Algorithme optimal :** si toute configuration T de tâches qui est ordonnançable (par un algorithme donné), l'est aussi par cet algorithme. Une autre définition d'optimalité existe et juge un algorithme optimal s'il détermine la meilleure séquence d'ordonnancement existante (respect des échéances, le cas échéant, minimisation des retards, minimisation du temps d'utilisation des ressources notamment le processeur).

Nous adoptons la première définition d'optimalité et nous dirons également que si un algorithme est optimal et ne peut ordonnancer un ensemble donné de tâches, aucun autre algorithme n'y réussira.

On dit qu'un algorithme d'ordonnancement applique un :

• **Ordonnancement préemptif** : s'il permet d'interrompre l'exécution de la tâche courante pour la prise en compte d'une tâche plus prioritaire. Ceci signifie qu'il fournit les mécanismes nécessaires pour gérer la préemption à savoir :

• la commutation de contexte : elle consiste à sauvegarder le contexte de la tâche en cours d'exécution (registres et zones mémoires) interrompue pour l'exécution d'une autre tâche. Ce contexte sera restauré quand l'exécution de la tâche sera reprise.

Ainsi, une tâche ne pouvant être exécutée en entier est tout de même acceptée si son exécution peut être partitionnée.

Le temps nécessaire à la commutation de contexte est négligeable. Toutefois, ce temps doit être pris en compte lors du calcul des dates de début d'exécution par l'ordonnanceur en effet le cumul de ces temps nécessaires à la commutation de contexte peut rajouter des délais qui pourraient être à l'origine d'une échéance dépassée.

On distingue deux manières d'autoriser la préemption. La première consiste à n'autoriser l'interruption de la tâche en cours que si son exécution peut ultérieurement être reprise sur le même processeur. La seconde envisage le cas échéant de faire migrer la tâche. Ce cas de figure peut se poser si le processeur reçoit une tâche très prioritaire, et qu'il interrompt la tâche en cours car sa priorité est moindre, sans pouvoir reprendre cette exécution par la suite. Dans cette situation, cette tâche sera migrée. Il est déconseillé dans les systèmes temps réel de recourir à la migration de tâches que nous distinguons du transfert de tâches. Le transfert consiste à placer les tâches avant le début de leur exécution par opposition à la migration qui est un déplacement des processus. Etant donné le coût de la migration, peu de systèmes l'appliquent et ce sont généralement des systèmes très souples où les contraintes ne sont pas strictes.

Concernant l'emploi de la préemption dans les systèmes temps réel existants, les avis sont très controversés. Certains la jugent nécessaire pour la prise en compte de tâches plus urgentes [RSSh90], [ScZh92], [StRa89], [Mok83]. De plus dans un mécanisme non préemptif, des tâches peuvent être refusées alors qu'il existe un ensemble d'intervalles de temps suffisants à une exécution de la tâche. D'autres excluent la possibilité de préemption pour deux raisons essentielles [LeYa86]. L'interruption d'une tâche introduit un délai d'ordonnancement supplémentaire qui risque de perturber les tâches déjà acceptées (on peut être amené à recalculer les dates d'ordonnancement S_i). De plus, il peut arriver que la tâche interrompue ne puisse pas être exécutée localement, il faut dans ce cas trouver un autre processeur qui garantisse son exécution. Le problème ne se pose pas pour un modèle dynamique, par contre si le modèle est statique, d'importantes simulations doivent être effectuées afin de borner de manière précise, ces délais.

Nous estimons que dans un modèle dynamique où les contraintes ne sont pas toutes connues a priori, il est intéressant voire même nécessaire d'offrir un ordonnancement

préemptif. D'autres part, de nombreux ordonnancements faisables sont rejetés dans le cas où la préemption n'est pas autorisée comme illustré dans la figure 2.4 qui suit.

Sur cette figure, la séquence T est composée de trois tâches ayant respectivement pour paramètres (R_i, E_i, P_i) les valeurs (0,1,3) (0,1,4) et (0,2,5). Toutes les occurrences de ces tâches ne peuvent être ordonnancées avant leur échéances respectives, sans appliquer de préemption. Nous rappelons que l'échéance d'une tâche périodique est donnée par l'arrivée d'une nouvelle occurrence de la tâche.

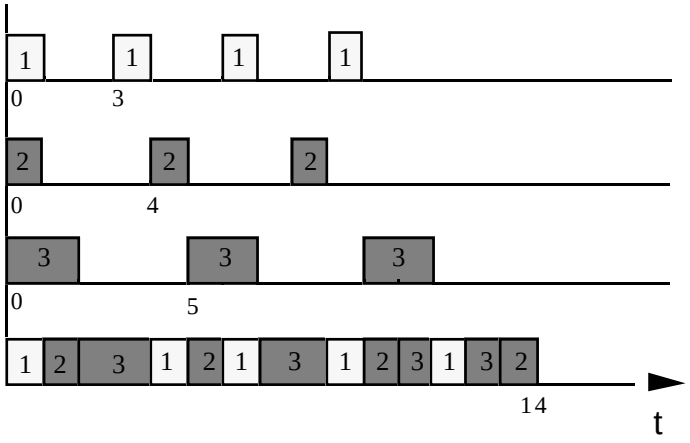


Figure n° 2.4 Exemple d'application de la préemption

Nous donnons à présent quelques définitions relatives à l'allocation des tâches.

• **Allocateur statique de tâches** : il applique un algorithme pour la répartition d'un ensemble de tâches entre les processeurs alloués à la machine. La répartition tient compte de critères qui peuvent être stricts tels que le respect des contraintes temporelles. D'autres critères d'optimisation peuvent également être pris en compte.

• **Mécanisme d'allocation dynamique de tâches** :

Le mécanisme d'allocation dynamique entre en action quand une tâche est refusée localement. Il se charge de la détermination d'un processeur pour exécuter la tâche. Il est également constitué d'un ensemble de modules :

• des politiques de recherche de processeurs pour la tâche refusée.

• des politiques pour le maintien d'un état des processeurs afin de choisir le plus capable d'assurer le respect des contraintes de la tâche.

L'hypothèse 2.1 est appliquée dans la plupart des systèmes temps réel [StRa89], [ScZh92] et autres. Pourtant, il existe une autre manière de faire, à savoir accepter quand même une tâche qu'on ne peut garantir localement pourvu qu'on puisse en écarter une moins urgente. A notre avis, ce choix est fait afin d'éviter de toujours remettre en question l'ordonnancement des tâches. De plus, dans ce cas, il n'existe plus de garantie délivrée par l'ordonnanceur, puisque n'importe quelle tâche peut être retirée du processeur et prise en compte par le mécanisme d'allocation dynamique. D'autant plus que la garantie de cette tâche par un autre processeur ne peut être assurée. Ce cas de figure est à la base d'une heuristique que nous exposons dans le chapitre 9 où nous proposons dans le cas d'une tâche urgente refusée, de choisir une tâche moins urgente, moyennant certaines garanties que la tâche sera acceptée sur un autre processeur.

Pour résumer, nous présentons dans la figure 2.5 qui suit une taxonomie des algorithmes d'ordonnancement et d'allocation. Nous y distinguons au premier niveau, les algorithmes d'ordonnancement monoprocesseur et ceux multiprocesseurs que nous appellerons désormais algorithmes d'allocation. Pour chacune de ces classes d'algorithmes, nous distinguons ceux applicables pour un modèle de tâches statique ou dynamique. Pour chacune de ces catégories, nous avons classé les algorithmes en fonctions des solutions obtenues : ainsi on trouve les algorithmes optimaux et les heuristiques. Le reste de la taxonomie présente les algorithmes que nous allons décrire dans chacun des chapitres sur l'ordonnancement, l'allocation statique et celle dynamique. Nous tenons à préciser que cette taxonomie reflète plutôt le plan que nous allons suivre dans notre rapport, et montre les termnologies qui y sont adoptées. Nous ne la comparons pas à des taxonomies présentées dans d'autres travaux, pour cette raison.

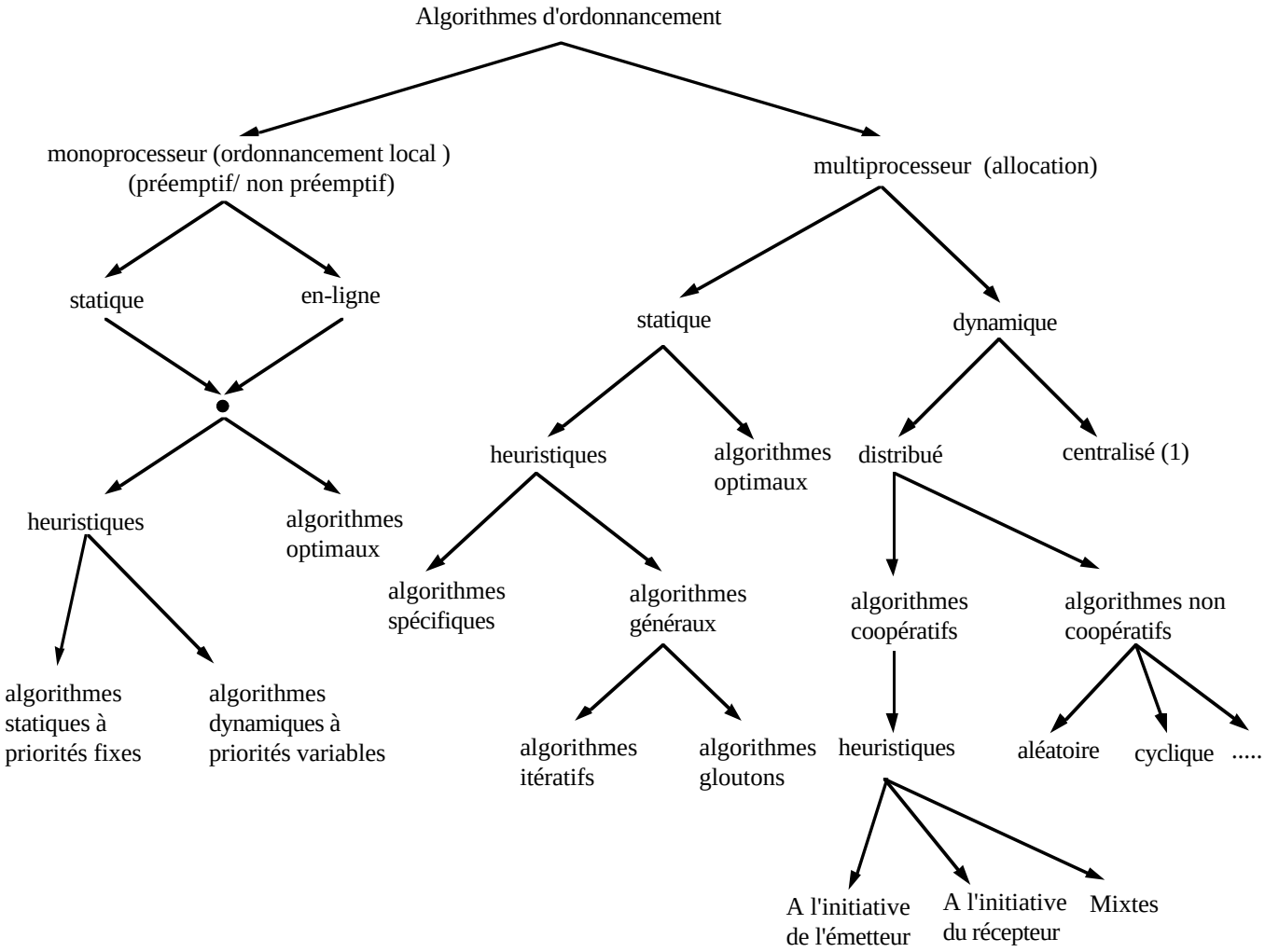


Figure n° 2.5 Une taxonomie des algorithmes d'ordonnancement et d'allocation

(1) Les algorithmes centralisés ne sont pas détaillés car ils ne font pas partie du cadre de notre étude.

Chapitre 3

Présentation de mécanismes d'ordonnancement distribué de tâches temps réel

3.1 Introduction

Nous présentons dans ce chapitre divers mécanismes d'ordonnancement distribué destiné à des applications temps réel. Dans un premier temps, nous nous proposons de discuter du cadre de conception de tels mécanisme. Un mécanisme d'ordonnancement distribué doit-il être intégré dans un système d'exploitation temps réel ou peut-il être supporté par n'importe quel système d'exploitation conventionnel tel Unix ou Mach ?

Nous commençons par exposer les différentes approches envisagées par les concepteurs de systèmes temps réel, et nous les comparons en prenant pour critères l'importance qu'ils accordent aux problèmes de temps et le niveau de garantie qu'ils offrent.

Dans la section 3.3, nous exposons le mécanisme que nous avons conçu dans le cadre du système d'exploitation parallèle Paros. Il est intégré au système et permet de prendre en compte la notion de temps à un niveau bas dans le système.

3.2 Comparaison de systèmes d'exploitation temps réel

Un des domaines les plus étudiés concernant les nouvelles générations d'applications temps réel, est celui du système d'exploitation temps réel adéquat qui supporte ces dernières. Comme nous l'avons introduit au chapitre 2, un système d'exploitation temps réel doit offrir un certain nombre de fonctionnalités qui le rendent capable de prendre en

compte la notion de temps à tous les niveaux du système et de garantir les tâches temps réel. Comme les systèmes conventionnels, ces systèmes temps réel doivent supporter un modèle de processus adéquat, des mécanismes de synchronisation et d'ordonnancement, une gestion de la mémoire appropriée, ainsi que des protocoles de communication et d'entrée/sortie.... Cependant, ils diffèrent des systèmes conventionnels par l'importance accordée à la notion de temps, et par la manière avec laquelle ceci est assuré.

A travers l'étude des systèmes d'exploitation qui prennent en compte la gestion du temps (nous expliquerons par la suite que ce ne sont pas forcément des systèmes temps réel), on distingue trois catégories de systèmes (un état de l'art est présenté dans [MSGh93], [PaDa93], [RaSt92]) : (1) des noyaux commercialisés pour systèmes embarqués de petite taille. Ils se veulent une plate-forme pour le développement d'applications temps réel. Ils offrent généralement un support basé sur le langage Ada ou bien des primitives au niveau du modèle de processus. (2) des extensions temps réel à des systèmes existants qui appliquent le partage de l'utilisation du processeur en quanta de temps. (3) des prototypes de systèmes temps réel développés par des laboratoires de recherche.

3.2.1 Noyaux pour systèmes embarqués

Ce sont des noyaux exécutifs de petites tailles, assez performants et particulièrement appropriés à des systèmes embarqués simples où les besoins sont essentiellement une exécution et un temps de réaction aux événements rapide (alarmes, interruptions...). Parmi les noyaux les plus connus, nous citons VxWorks [FSWi91], pSOS [Tho90] et d'autres [Brow84]. Ils offrent un ordonnancement par priorités (généralement un ensemble de priorités prédéfinies est offert), une horloge globale, des primitives pour la préemption des tâches, pour le retardement ainsi que des mécanismes de synchronisation, etc.

Toutefois, ces noyaux sont inefficaces pour des systèmes complexes ou de taille importante. En effet, dans de pareils systèmes, les contraintes sont plus compliquées et portent à la fois, sur le temps, les ressources, la communication et il est extrêmement difficile de prédire avec de pareils noyaux de base que les contraintes seront vérifiées. La principale cause est due aux délais introduits pour la gestion de ces systèmes. Dans le cas où ces noyaux sont utilisés, on se retrouve aussi avec des phénomènes d'inversion de

priorités[1] car les mécanismes d'ordonnancement utilisés dans ces noyaux sont de très bas niveau. Egalement, des situations où les ressources ne sont pas disponibles à temps, où les messages ne sont pas délivrés à temps, etc. En effet, souvent ces noyaux sont conçus avec le langage Ada qui comporte des contraintes de priorités statiques. Toutefois, de nouvelles versions de Ada sont en cours, et introduisent des priorités dynamiques.

De nombreux concepteurs se sont alors tournés vers des noyaux plus sophistiqués qui permettent à l'utilisateur de concevoir un système dépendant de son application. On a donc observé la naissance de versions temps réel de systèmes d'exploitation conventionnels.

3.2.2 Extensions temps réel pour des systèmes d'exploitation conventionnels

On observe depuis quelques temps que de nombreuses applications temps réel sont exécutées par des systèmes dits "temps réel". Ces derniers sont en fait, des versions optimisées de systèmes d'exploitation appliquant un ordonnancement par quanta de temps[2] . Ainsi, Unix a été étendu à RT-Unix[FGGR91], Posix à RT-Posix, ainsi que Mach [TNRa90] et Chorus [Chor91], [AHLR89].

Ces systèmes sont généralement plus lents et moins prédictibles que les noyaux de systèmes embarqués, mais ils sont munis d'environnements de développement plus puissants, et offrent plus de fonctionnalités. Un autre avantage à souligner, est celui de la portabilité de ces systèmes.

Toutefois certains handicaps subsistent pour l'adoption de tels systèmes dans des applications temps réel souples. Les priorités sont souvent fixes, aucun module n'est offert pour un changement dynamique de priorités (systèmes plus appropriés à des applications temps réel statiques), les queues de processus sont organisées en FIFO, mais également la communication n'est pas prédictible de manière fiable.

Un autre handicap à mettre en évidence réside au niveau de l'approche conceptuelle, la philosophie de ces systèmes conventionnels est de réduire au maximum le contrôle des

[1] Tâche de haute priorité bloquée par une de moindre priorité pour l'accès à une ressource.
[2] Un ordonnancement par quanta de temps consiste à partager la capacité du processeur de manière équitable entre les tâches. Ainsi, une durée d'exécution appelée quantum est fixée et chaque tâche est exécutée jusqu'à la fin du quantum, ensuite elle est interrompue et placée dans une file d'attente des tâches. Elle ne se réexécutera qu'une fois que toutes les autres tâches prêtes aient également consommé chacune un quantum.

ressources par l'application. C'est le système qui décide quand et à qui attribuer les ressources afin d'améliorer les performances de l'application. Même si des niveaux de priorités sont offerts, si le système réajuste celles-ci ne fusse que modestement, il est tout a fait inutile de construire un ordonnancement prioritaire basé sur le temps. La version temps réel de Unix a été conçue pour un public large incluant des utilisateurs temps réel et non temps réel. Ainsi, environ 60 appels systèmes sont déconseillés lors de l'utilisation du système pour des applications temps réel.

Dans la version temps réel de Mach, on remarque des primitives d'ordonnancement assez performantes qui permettent notamment de résoudre le problème très fréquent d'inversion de priorité, toutefois ces mécanismes sont également implémentés au dessus d'un ordonnancement par quanta de temps. Au vu de tous ces handicaps, nous sommes amenés à nous poser la question suivante : doit-on étendre un système d'exploitation performant déjà existant ? Surtout que l'on remarque que les systèmes d'exploitation conçus pour systèmes embarqués sont plus simples et plus rapides que ces versions étendues de systèmes conventionnels.

La réponse est donnée en fonction des applications visées. Cette approche est adaptée aux applications temps réel souples, qui ne nécessitent pas une prise en compte de l'ordonnancement dans la communication des messages ainsi qu'à d'autres niveaux.

3.2.3 Prototypes de systèmes d'exploitation

Le respect des contraintes temporelles, doit se traiter de façon algorithmique et non en proposant du matériel ou des systèmes "assez rapide". On montre facilement [LeLa92], qu'un système rapide doté d'algorithmes d'ordonnancement non adaptés au temps réel peut échouer. Plusieurs concepteurs de systèmes estiment qu'un support direct pour résoudre les problèmes de temps doit être fourni à tous les niveaux du système. Toute la philosophie de conception du système doit être révisée. Ainsi divers systèmes d'exploitation temps réel ont vu le jour.

Dans la suite de cette section, nous allons faire un survol des différents projets de systèmes d'exploitation existants. Ces systèmes seront comparés sur les points suivants : le modèle de système concerné (statique ou dynamique), le type d'ordonnancement fourni, la synchronisation mise en œuvre ainsi que les autres mécanismes prenant en compte la notion de temps. Il existe d'autres noyaux de systèmes temps réel tels que

Chaos [SGBo87], Hartos [KKSh92], Alpha [CJRe92], Rnet [CMMa87] que nous ne détaillerons pas. Ils offrent généralement un support pour la spécification d'applications temps réel ainsi que des algorithmes d'ordonnancement par priorité. Nous ne citerons que les systèmes les plus connus et les plus complets. Le tableau qui suit en résume les principales caractéristiques :

Systèmes d'exploitation	Propriétés du modèle	Propriétés de l'ordonnancement	Autres mécanismes	Architecture cible
Spring [StRa89][1]	- modèle dynamique de tâches - priorités utilisateurs : tâches critiques, essentielles, non temps réel	- analyse du pire des cas - ordonnancement et allocation statique des taches critiques	- primitives de communication synchrones appropriées	- systèmes distribués - un noeud est composé : processeur application, processeurs systèmes, et d'un sous-système d'entrée-sortie
Maruti [LTCA89]	- système orienté objet - modèle dynamique de tâches	- ordonnancement par échéancier	-ordonnancement des communications	- implémenté au dessus du noyau Mach, lui même conçu au dessus de Unix
Mars [DRSK88] [Kope89] [SRGr89]	- modèle de tâches périodiques et apériodiques - limité à des tâches sans conflit de ressources - particulièrement adapté à la tolérance aux pannes	- ordonnancement statique - ordonnancement cyclique pour les tâches périodiques - utilisation des périodes d'oisiveté du processeur pour l'ordonnancement des tâches apériodiques	- algorithmes de synchronisation et de gestion du temps	- systèmes distribués - plusieurs bus synchrones spécifiques pour parer aux pannes
Arts [ToMe89]	- modèle dynamique composé de tâches critiques et de tâches à échéances relatives - noyau orienté objet	- ordonnancement par échéancier - protocole approprié pour résoudre l'inversion de priorité - ordonnancement avec des fonctions de valeurs (FVT)	-ordonnancement des communications	- implémenté sur un réseau de stations SUN3.

Figure n° 3.1 Comparaison de systèmes d'exploitation temps réel

Nous dégageons essentiellement les points suivants de cette présentation :

(1) les systèmes comparés dans le tableau de la figure 3.1 sont les plus complets dans la mesure où le mécanisme d'ordonnancement temps réel est validé par des mécanismes de communication appropriés ainsi que des mécanismes de gestion du temps.

(2) ces systèmes ont été implémentés sur des machines distribuées. Le support de communication de ces machines est souvent un réseau temps réel avec un bus approprié.

[1] Les références citées sont anciennes car il s'agit des premiers travaux ayant décrit ces systèmes d'exploitation.

3.3 Mise en œuvre d'un mécanisme d'ordonnancement distribué pour tâches temps réel

Au vu des approches exposées précédemment, il ressort que le mécanisme d'ordonnancement proposé doit être conçu et faire partie d'un système d'exploitation qui permette de développer les mécanismes nécessaires à la gestion du temps à tous les niveaux où cela sera nécessaire. Nous avons développé notre mécanisme pour le système Paros [CEMW93], qui est un système parallèle multi-couches.

3.3.1 Présentation de PAROS

Paros est un système d'exploitation conçu pour des machines massivement parallèles et développé dans le cadre du projet européen ESPRIT SUPERNODE II (P2528). C'est un système bâti sur la notion de couches et de machines virtuelles. Paros repose sur un noyau adaptable à la fois au matériel utilisé ainsi qu'aux besoins des applications. Il offre tous les services de base afin de construire des machines virtuelles offrant ainsi à l'utilisateur plusieurs interfaces et environnements de programmation. Ainsi on trouve la machine virtuelle à allocation, à diffusion, et celle à mémoire virtuelle partagée. Nous proposons la machine virtuelle temps réel.

Paros est supporté par le noyau ParX [Munt89] qui implémente le modèle de processus et celui de communication. ParX se distingue par un modèle de processus hiérarchique à trois entités : la Ptâche, la Tâche et le Thread. Chaque entité du modèle doit être modifiée afin de permettre l'expression d'un modèle de processus temps réel.

3.3.2 Structure générale de ParX

ParX est un noyau qui vise à fournir un support d'exécution avec plusieurs niveaux d'abstraction sur lequel peuvent être mis en œuvre différents sous-systèmes, chacun offrant à ses utilisateurs une vision de machine virtuelle. Une machine virtuelle est un domaine protégé et géré par le système d'exploitation offrant à l'utilisateur les services systèmes nécessaires et réalisant les modèles d'exécution et de communication entre processus.

La structure globale de ParX est en couches :

• Une première couche "extension matérielle" virtualise une machine parallèle dont les fonctionnalités deviennent accessibles quel que soit le support d'exécution. Elle offre une machine abstraite composée de processeurs entièrement connectés, sans mémoire commune, et disposant d'un support minimal pour les services de base pour la gestion des processus et des communications.

• Au dessus, le micro-noyau met en œuvre le modèle de processus et celui de communication, ainsi qu'un mécanisme de construction générique de protocoles. C'est à partir de ce mécanisme que sont intégrés dans le noyau, en fonction des besoins, des protocoles de communication, des politiques d'ordonnancement et autres (partage de mémoire, etc.)

Au dessus de ParX sont bâtis des outils tels que des systèmes d'exploitation, ou des environnements de programmation ainsi que des bibliothèques.

Egalement au dessus de ParX, Paros gère le découpage de la machine et son partage entre les utilisateurs.

ParX a été implémenté sur des machines Supernode à base de transputers.

Modèle de processus et de communication

ParX offre un modèle de processus structuré en 3 niveaux qui permet de manipuler plusieurs grains de parallélisme et de communication :

• le **thread** : il correspond au grain de parallélisme le plus fin : c'est l'unité élémentaire d'exécution. L'ensemble des threads sur un même processeur s'exécutent en pseudo-parallélisme.

• la **Tâche** : c'est un espace d'adressage protégé dans lequel s'exécutent plusieurs threads. La Tâche est une entité qui définit l'environnement d'exécution des threads et leur espace d'adressage.

• la **Ptâche** : elle regroupe un ensemble de tâches pour une application donnée et en gère aussi le contrôle. Cette entité est bien adaptée à la gestion du parallélisme, en effet elle définit la distribution des tâches, les communications entre elles, la synchronisation, la désignation. La Ptâche représente une application s'exécutant sur une machine virtuelle. Elle correspond au grain de parallélisme le plus gros.

Nous ne développerons pas le modèle de communication, mais nous précisons que ParX offre deux médiums de communication adaptés au modèle de processus :

• le port : permet la communication de plusieurs Tâches vers une. Le port peut être synchrone ou asynchrone et peut être accédé par des Tâches de différentes Ptâches.

• le canal : est une liaison point à point entre différentes Tâches d'une Ptâche.

Un cluster composé de plusieurs processeurs est attribué à chaque Ptâche s'exécutant sur la machine. La figure 3.2 ci-dessous illustre le modèle de processus et celui de communication:

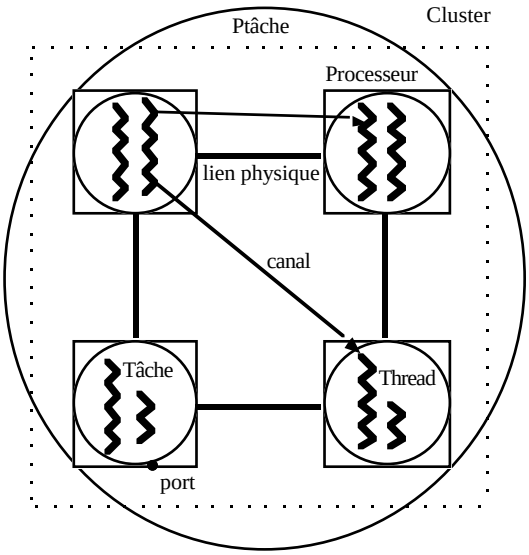


Figure n° 3.2 Modèles de communication et de processus de ParX

Le modèle de processus doit permettre de spécifier des applications temps réel. Afin d'exprimer les contraintes temporelles, et de permettre au mécanisme d'ordonnancement de manipuler le modèle de processus, il est nécessaire de disposer d'un langage unique approprié à ces besoins. Dans cette optique, nous avons conçu un Langage de Description du Système LDS. Il supporte toute information exigée par le mécanisme aussi bien pour la spécification des tâches que pour leur ordonnancement.

En ce qui concerne les spécifications, le LDS intervient au niveau du modèle de processus et du modèle de communication. Ainsi, dans un premier temps, il décrit la tâche en tant qu'entité ordonnançable du modèle et les ressources utilisées par celle-ci.

3.3.3. Prise en compte de l'aspect temps réel : le langage de description du système LDS

(1) Description du modèle de processus temps réel

Le LDS reprend la description du modèle de processus et l'adapte afin d'y incorporer les caractéristiques temps réel de l'application.

La Ptâche

La garantie des systèmes temps réel s'associe souvent à des considérations de tolérances aux pannes. La Ptâche ne variera que de peu dans la mesure où on y rajoute des spécifications pour la correction de cas d'éventuelles pannes. Selon le degré de tolérance visé, ces descriptions peuvent être simples et composées d'un nombre minimum de copies à assurer, ou plus complexes auquel cas, on peut préciser le type de garantie à appliquer :

spécif_tolérance_pannes	: :	desc_copies
		desc_alternatives
		…
desc_alternatives	: :	**Tâches_liste** liste_de_tâches
desc_copies	: :	**Tâche_liste**
		(desc_tache, min_copies, max_copies)
min_gar_copies	: :	**Nombre** nombre_de_copies_min
max_gar_copies	: :	**Nombre** nombre_de_copies_max

Figure n° 3.3 Spécifications de tolérances aux pannes

Quand le type de garantie à fournir est basé sur le nombre de copies (par la donnée de la structure desc_copies), le programmeur établit une liste des copies à assurer pour chaque tâche. Nous envisageons également le cas de figure où il est laissé au mécanisme d'ordonnancement la liberté de fixer le nombre de copies à réaliser ainsi que les tâches concernées. Dans ce cas, le programmeur précise le nombre minimal et maximal de copies à réaliser. La détermination de ce nombre, se fait grâce à l'algorithme d'ordonnancement local qui une fois déroulé, permet de connaître les tâches qui risquent de dépasser leur échéances, et par conséquent de les dupliquer. Nous tenons à préciser que ce type de tolérance aux pannes est particulièrement adapté aux modèles statiques. En effet, en

environnement dynamique, il est difficile de prévoir le nombre de copies nécessaires avant l'exécution ainsi que de connaître les processeurs sur lesquels les tâches seront dupliquées.

Quand le type de garantie est l'alternative, le mécanisme reçoit une liste de tâches à prendre en compte lors de l'ordonnancement. Si l'algorithme d'ordonnancement local échoue lors de l'ordonnancement de la première tâche, la tâche suivante dans la liste donnée pour le paramètre desc_alternatives est considérée.

La Tâche

Dans ParX, une Tâche est principalement décrite par un identificateur, une priorité haute ou basse (le transputer n'offrant que celles-ci [Eldv92]), un état (en cours d'exécution, en attente, ou interrompue...), un espace de travail, un contexte et enfin le premier thread à exécuter. Il est nécessaire de rajouter à cette description, des paramètres plus appropriés à des applications temps réel. Les paramètres décrivant une tâche temps réel de ParX sont donnés dans la définition 1.

Définition 1

Une tâche est décrite par deux ensembles de paramètres *Tâche = (Ordonnancement, Exécution)*

• *Ordonnancement* : dans cet ensemble sont regroupés, les paramètres temporels nécessaires à l'ordonnancement de la tâche.

• *Exécution* : cet ensemble regroupe les caractéristiques nécessaires à l'exécution de la tâche. En plus des paramètres décrivant une Tâche de ParX cités auparavant (descriptif de l'exécution), on y trouve le nom du fichier contenant le code exécutable de celle-ci. Cet ensemble peut éventuellement contenir les ports et canaux utilisés lors des communications ainsi que les ressources utilisées par la tâche.

La grammaire du LDS est illustrée dans la figure 3.4 :

Tâche_desc	: :	(identificateur) {Tâche_attributs}
Tâche_attributs	: :	Ordonnancement_spécif
		Exécution_spécif

Ordonnancement_spécif	: :	**Type** : Périodique/Non périodique **Priorité utilisateur** : (critique, essentielle, non temps réel) **Nombre** période, temps_d'exécution date de début, échéance
Execution_desc	: :	**Structure** descriptif_exécution **Code** nom_fichier **Port** port_liste **Ressource** ressource_liste

Figure n° 3.4 Description d'une Tâche

Le Thread

Le thread est le flot minimal d'exécution et n'a pas d'espace d'adressage propre dans ParX. Son contrôle et réalisé par la Tâche à laquelle il appartient, il partage ainsi l'espace d'adressage de la Tâche avec l'ensemble des threads de celle-ci.

(2) Faisabilité de l'ordonnancement pour ParX

La structure du modèle de processus de ParX en 3 niveaux implique une étude de faisabilité de l'ordonnancement pour chaque entité du modèle. Toutefois au premier abord, l'entité concernée par l'allocation et l'ordonnancement est la Tâche. Dans la suite de ce chapitre nous noterons la tâche avec une initiale en majuscule en référence à la Tâche du modèle de processus de ParX.

Concernant la Ptâche, dans ParX, chaque Ptâche a ses propres clusters qu'elle ne partage avec aucune Ptâche. L'allocation d'une Ptâche consiste à lui allouer un cluster de processeurs. Le nombre de processeurs du cluster doit être optimal. Ceci implique qu'il est à la fois suffisant pour réduire le temps de réponse de l'application mais également réduit afin que des processeurs ne se retrouvent pas libres trop tôt. Allouer trop de processeurs à une Ptâche revient à en priver une autre. Au sein d'une Ptâche, l'allocation des tâches au processeur est réalisée par un algorithme d'allocation statique.

Actuellement, les Ptâches ne partagent pas de clusters (un même processeur ne peut être alloué à deux Ptâches différentes). Au cas où plusieurs Ptâches deviennent autorisées

à partager des clusters (par la donnée des mécanismes de contrôle nécessaires), l'ordonnancement des Ptâches se ramènera à un ordonnancement de Tâches.

L'ordonnancement des Tâches sera réalisé en appliquant des algorithmes d'ordonnancement local.

Paros propose des primitives pour l'ordonnancement des threads. L'algorithme appliqué est le "Round-Robin". Toutefois, il n'est guère possible de distinguer ces threads entre eux, si on n'examine pas l'adresse mémoire. Nous n'envisageons pas d'ordonnancement au niveau d'un thread temps réel, puisqu'on ne peut pas contrôler son exécution de manière indépendante. Le thread n'ayant pas d'espace d'adressage propre, on ne peut pas transférer des threads entre processeurs. Toutefois, on peut envisager d'ordonnancer des threads dans le cadre de la tâche à laquelle ils correspondent. Une fois qu'une tâche est acceptée, on a la garantie que ses threads seront exécutés; on peut alors si chaque thread est doté de paramètre temporels, réaliser un ordonnancement au niveau de ceux-ci.

Un contrôle approprié est nécessaire à l'allocation et l'ordonnancement de ce modèle de processus. Pour cela, le noyau ParX doit supporter un mécanisme approprié aux problème temps réel qui fournit les algorithmes et le contrôle nécessaires. Le mécanisme que nous proposons pour ce faire, est appelé *mécanisme d'ordonnancement distribué*. Le mécanisme doit son nom au fait qu'il gère les problèmes d'ordonnancement sur tous les processeurs du système.

Il est lui-même composé d'un allocateur statique, d'un mécanisme d'ordonnancement local et d'un mécanisme pour l'allocation des tâches temps réel.

(3) Structure du mécanisme d'ordonnancement distribué

Le mécanisme permet de gérer les tâches créées avant et pendant l'exécution. A chaque création dynamique de Tâches, le mécanisme d'ordonnancement local est invoqué. Suite à la tentative d'ordonnancement, et en cas d'échec de cette dernière, le mécanisme d'allocation dynamique est invoqué pour allouer la tâche à un autre processeur.

Le mécanisme d'allocation dynamique doit être pourvu des propriétés suivantes afin d'assurer le respect des contraintes temporelles des tâches :

(1) la complexité des algorithmes appliqués doit être bornée : ici on se réfère aux algorithmes coûteux en communications à savoir ceux d'allocation dynamique employés pour la recherche de processeurs pour les tâches refusées. Si le coût de ces derniers peut être borné, on peut décider à la création d'une tâche si elle peut être garantie ailleurs ou non et prendre les mesures nécessaires à temps le cas échéant. Par exemple, la complexité de ces algorithmes ne doit pas dépendre du nombre de tâches à ordonnancer (surtout que ce dernier n'est pas connu à l'avance).

(2) le mécanisme doit être peu coûteux : les délais introduits par les algorithmes appliqués pour la recherche de processeurs doivent être faibles comparés aux temps d'exécution des tâches.

(3) un mécanisme en-ligne doit améliorer les performances du système en maximisant le nombre de tâches garanties. C'est le principal élément de mesure choisi pour qualifier un mécanisme d'allocation. Certaines tâches ne seront pas exécutées ou dépasseront leur échéances car le mécanisme ne trouvera pas de processeurs capables de les garantir. Il faudra veiller à ce que ces tâches soient non critiques et que leur nombre soit réduit.

Concernant l'implémentation du mécanisme d'ordonnancement distribué, plusieurs solutions sont à envisager: centralisé, semi-centralisé ou distribué.

Un mécanisme *centralisé* est implémenté sur un processeur dédié. Le mécanisme doit avoir des informations sur l'état de tous les autres processeurs du réseau. A chaque création de tâche, un processeur du réseau est désigné en se basant sur ces informations.

Quand le mécanisme est semi-centralisé, le réseau des processeurs est organisé en N groupes chacun rattaché à un serveur. Cette solution a l'avantage de réduire le coût introduit par la gestion de l'état du système.

Un mécanisme distribué est implémenté sur tous les processeurs du réseau. Chaque mécanisme gère les tâches du processeur sur lequel il est implémenté. De cette manière, la gestion de l'état du système est répartie entre les processeurs et on évite les goulots d'étranglement ainsi que des délais supplémentaires, étant donné que notre principale préoccupation est la réduction du temps de l'ordonnancement et de l'allocation des tâches à travers la machine.

La classe d'architectures qui nous intéresse est constituée de processeurs homogènes, fortement couplés et communiquant à travers l'envoi de messages (architectures dites MIMD sans mémoire commune).

Nous pensons qu'un mécanisme entièrement distribué est plus adapté à ces architectures et permet de bien en exploiter le parallélisme. Une gestion semi-centralisée nous semble à écarter étant donné les délais supplémentaires qui seront inévitablement introduits. En effet, il faudrait pour chaque tâche à allouer, rajouter au minimum le délai nécessaire pour communiquer ses paramètres au processeur serveur. Toutefois, avant de se prononcer, il serait plus prudent de mesurer les performances d'une telle implémentation et d'étudier les applications qui pourraient être concernées.

Dans notre implémentation, chaque processeur sera muni de son propre mécanisme d'ordonnancement et d'allocation qui gérera ses propres tâches. Nous retrouvons ce choix d'implémentation également dans les systèmes distribués. Nous citons les systèmes SPRING [StRa89], ARTS [ToMe89], MARUTI [LTCA89] et d'autres encore qui ont été présentés auparavant. Pour cela tous les mécanismes et algorithmes présentés dans cette thèse sont distribués. En revanche, l'allocateur statique ne sera pas résident sur chaque processeur, puisqu'il applique un algorithme statique.

Le mécanisme est implémenté au dessus du noyau ParX, en effet, le mécanisme étant distribué, la communication entre les allocateurs et les ordonnanceurs nécessite une fonction de routage des messages entre les processeurs (car le réseau n'est pas complètement connecté).

La figure 3.5 décrit la structure du mécanisme d'ordonnancement distribué :

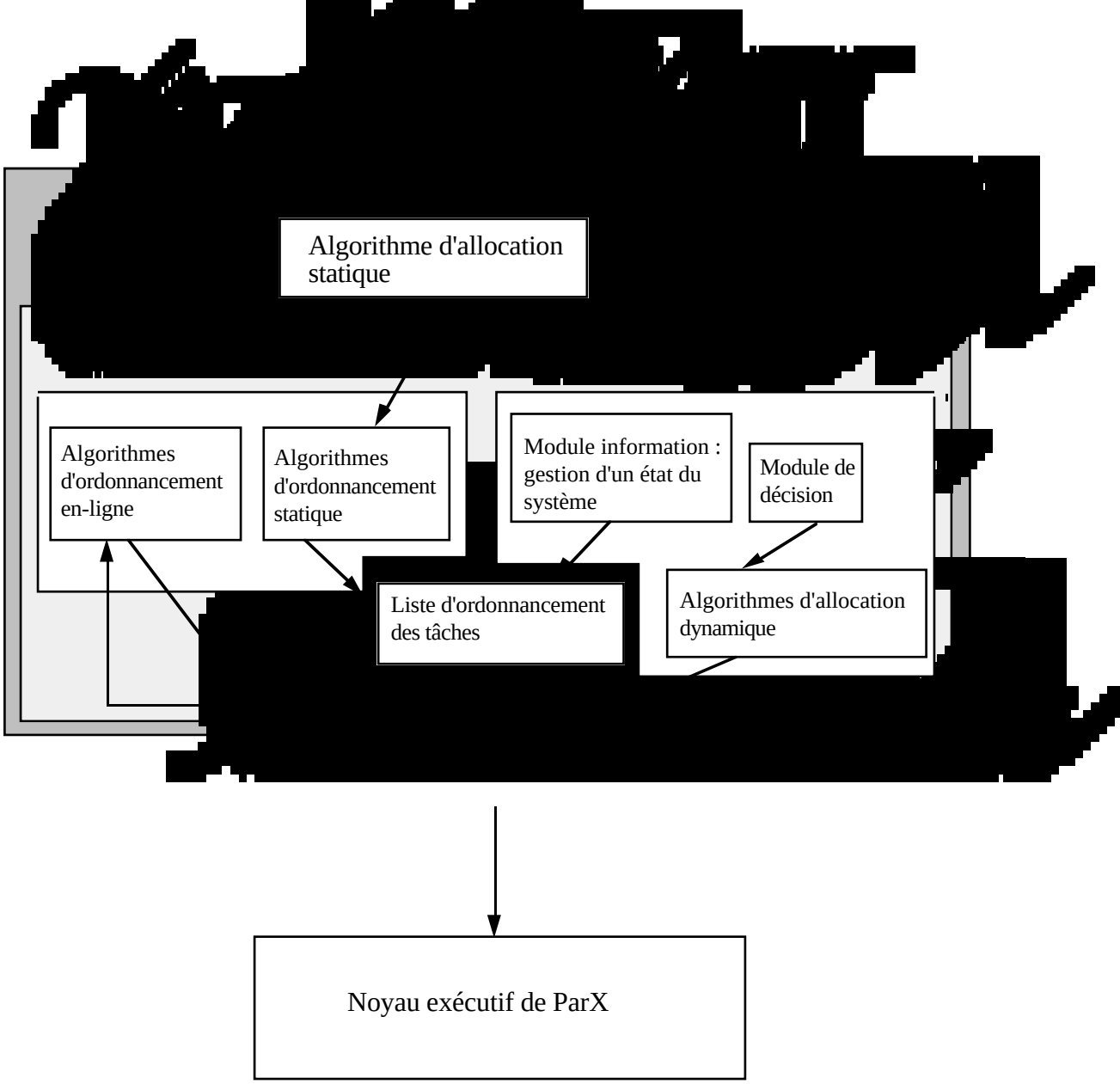


Figure n° 3.5 Structure du mécanisme d'ordonnancement distribué

La figure 3.5, donne le plan des chapitres qui vont suivre. Le reste de ce rapport est organisé en trois partie. Une pour l'allocation statique, une pour l'ordonnancement local, et enfin une dernière pour l'allocation dynamique. Nous commencerons par la partie d'ordonnancement local, en effet la présentation des algorithmes utilisés est essentielle pour la compréhension des algorithmes d'allocation. Chaque partie est composée de deux chapitres. Le premier présente une analyse critique des méthodes existantes et le second notre contribution dans le cadre du mécanisme d'ordonnancement distribué.

Partie II

Ordonnancement local de tâches temps réel

Résumé

L'ordonnancement des tâches est un des domaines les plus étudiés parmi ceux concernés par les systèmes temps réel. Ceci est dû, à notre avis, au fait que l'ordonnancement est nécessaire afin d'exécuter les tâches en respectant les contraintes temporelles. Les modèles de tâches temps réel sont très divers. Un modèle est différent dès lors qu'il comprend des tâches communicantes ou qu'il inclut des tâches apériodiques ou même encore que les tâches périodiques aient des échéances antérieures aux occurences des tâches. Pour chaque type de modèle, ont été conçus des algorithmes d'ordonnancement spécifiques. Le chapitre 4 effectue un survol de tous les types d'algorithmes existants.

Le rôle de ces algorithmes ne se limite pas à l'ordonnancement des tâches sur le processeur, car les tâches peuvent avoir accès à plusieurs ressources. Dans ce cas, un ordonnancement des tâches pour chaque ressource est indispensable. On observe des phénomènes d'inversion de priorité (tâche bloquée pour l'accès à une ressource par une tâche moins prioritaire), des délais de blocages non bornés pour l'établissement des communications. Le chapitre 4 présente également des algorithmes pour la résolution de ces problèmes.

Nous avons conçu et mis en œuvre un algorithme d'ordonnancement en-ligne (durant l'exécution) que nous présentons au chapitre 5. L'algorithme offre un temps d'ordonnancement satisfaisant et un nombre de tâches réordonnancées faible.

Chapitre 4

Techniques d'ordonnancement local de tâches temps réel

Devant l'abondance des travaux dans ce domaine et leur diversité, il nous est impossible de les citer tous, nous avons donc classé ceux-ci en différentes approches selon les modèles de tâches visés.

La section 4.1 définit les principales approches d'ordonnancement. Dans la section 4.2, nous présentons quelques algorithmes afin d'illustrer ces approches.

4.1 Principales approches d'ordonnancement

Dans la section 2.3, nous avons expliqué que l'ordonnancement local consiste à vérifier qu'une tâche peut être ordonnancée sans remettre en question les tâches acceptées et ordonnancées auparavant, et à déterminer une nouvelle séquence d'exécution incluant cette tâche si ce test est concluant. Souvent le test ne consiste pas en la recherche d'un nouvel ordonnancement; cependant quand le test est concluant ceci signifie que l'ordonnanceur trouvera sûrement un ordonnancement correct le moment venu. Ce test de garantie est propre à l'algorithme d'ordonnancement en place et au modèle de tâches visé.

Nous avons classé les approches que nous présentons selon les caractéristiques de ce test : (1) s'il est fait ou non, (2) de manière statique ou dynamique (3) s'il aboutit à la génération d'une séquence d'exécution ou non. De cette vision découle la classification suivante :

(1) L'approche par échéancier statique : un test de garantie est réalisé et un échéancier des tâches est construit et stocké dans une table (divers algorithmes d'ordonnancement peuvent être appliqués). Cette approche est guidée par le fait qu'elle concerne des systèmes temps réel stricts, où la plupart des tâches sont périodiques et critiques et doivent donc être allouées statiquement afin de garantir le déterminisme. La construction de l'échéancier est une assurance en soi. Toutefois, cette approche est non flexible [StRa89].

(2) L'approche à priorités statiques : un test de garantie est réalisé mais aucune séquence d'exécution n'est générée. Une priorité est attribuée aux tâches en fonction de leurs échéances. Lors de l'exécution, la tâche de plus haute priorité est exécutée la première.

(3) L'approche à priorités dynamiques : le test de garantie est effectué durant l'exécution. Il est complété par la détermination d'une séquence d'exécution qui peu à tout moment être modifiée suite à l'acceptation d'une nouvelle tâche.

Dans certains mécanismes, le test revient à rechercher un ordonnancement correct par l'exécution de l'algorithme d'ordonnancement. On pourrait certes penser qu'il est intéressant pour réduire le temps de réponse de confondre test de garantie et ordonnancement, cependant dans le cas où la tâche est refusée, le temps d'ordonnancement devient un délai de plus à rajouter au temps de traitement de la tâche en question par le mécanisme d'allocation dynamique.

Quand le test et l'exécution de l'algorithme d'ordonnancement sont dissociés, on s'appuie sur les paramètres temporels de la tâche T_i et plus particulièrement sur R_i, E_i et D_i pour effectuer ce test. Dans les systèmes statiques, le test consiste plutôt en une condition suffisante et rarement nécessaire qui détermine si un ensemble de tâches peut être ordonnancé par l'algorithme en question ou pas.

4.2 Quelques algorithmes d'ordonnancement

On peut considérer l'ordonnancement de tâches comme la recherche d'un ordonnancement correct parmi un ensemble de combinaisons entre les tâches [Herr90]. L'espace de recherche peut être structuré en un arbre où la racine représente l'ordonnancement vide et où chaque nœud intermédiaire est un ordonnancement partiel.

Les feuilles de l'arbre sont des combinaisons entre toutes les tâches. La figure 4.1 présente l'espace de combinaisons entre les tâches pour un algorithme de parcours en arbre.

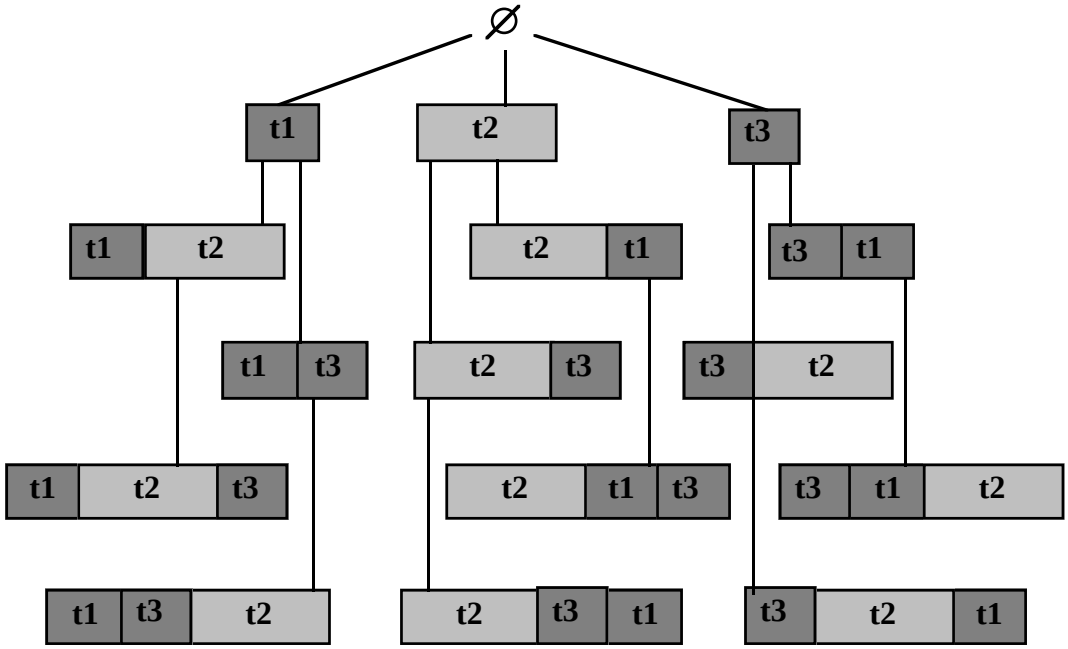


Figure n° 4.1 Arbre des combinaisons entre les tâches

L'ordonnanceur n'a plus qu'à chercher parmi les feuilles celles qui correspondent à un ordonnancement correct. Il est bien sur évident que toutes les feuilles ne peuvent correspondre à des ordonnancement corrects. Cette recherche est exhaustive, en effet pour un processeur sur lequel on se propose d'ordonner *n* tâches sans préemption, la profondeur de l'arbre est de *n* et le nombre de feuilles est (*n*!). Pour un ordonnancement préemptif, le problème se complique davantage.

4.2.1 Algorithmes d'exploration de l'espace de recherche

Les algorithmes de recherche par énumération des solutions sont optimaux mais ils ont une complexité inacceptable. Nous avons remarqué que souvent, les heuristiques appliquées effectuent un parcours en arbre, pour cette raison, l'espace de recherche sera appelé arbre de recherche par la suite.

Les algorithmes de recherche par séparation et évaluation (Branch & Bound) ont été appliqués conjointement à des heuristiques afin de guider la recherche dans l'espace des combinaisons. Le principe est de parcourir l'arbre de recherche en évaluant à chaque niveau les nœuds obtenus afin d'écarter ceux ne pouvant conduire à un ordonnancement

correct (grâce à une fonction d'élimination), et de choisir le meilleur nœud parmi ceux pouvant donner un ordonnancement correct (grâce à une fonction de sélection). Pour guider la recherche, différentes fonctions heuristiques peuvent être appliquées. Le principe est le suivant :

(1) On démarre la recherche avec un ordonnancement vide

(2) une fonction heuristique H est appliquée à l'ensemble T des tâches, et on choisit la tâche qui donne la plus petite valeur de la fonction. L'ordonnancement que l'on se propose de construire est ainsi démarré avec cette tâche disons T_a.

(3) la fonction est ensuite appliquée à l'ensemble des tâches T - $\{T_a\}$. Soit T_b la tâche minimisant les valeurs de H.

(4) la correction de l'ordonnancement déterminé (T_aT_b) est testée,

- si c'est un ordonnancement fortement correct[1] , le nœud est retenu et H est appliquée au reste des tâches (T - $\{T_a,T_b\}$)
- le cas échéant, on revient au nœud précédent T_a et on choisit la tâche qui offre la seconde plus petite valeur de H.
- Les étapes de sélection et de vérification de l'ordonnancement fortement correct sont réitérées.

La recherche se poursuit jusqu'à construction d'un ordonnancement complet. L'algorithme est en O(n^2), en effet la sélection et l'élimination sont appliquées au nombre de tâches restantes qui décroît de 1 à chaque niveau (elles ont donc un coup linéaire).

Parmi les fonctions heuristiques H, on trouve H(T_i) = D_i (en appliquant la petite échéance d'abord), H(T_i) = E_i (en appliquant le plus petit temps d'exécution d'abord)....Plusieurs heuristiques et simulations ont été développées dans [ZRST87b], [ZhRa87], [ZRST87a]. Dans [RSSh89], la complexité de l'algorithme est réduite à *O(nk)*, à chaque étape, on applique H à uniquement *k* tâches parmi celles restantes. Les simulations montrent que *k* est assez faible par rapport à *n*.

Parmi les heuristiques, on distingue de nombreux algorithmes qui sont optimaux quand ils sont appliqués à des tâches avec des caractéristiques particulières [AlDe92],

[1] Un ordonnancement partiel est dit fortement correct si tous les ordonnancements partiels obtenus en l'étendant avec une tâche (prise parmi le reste des tâches à ordonnancer) sont corrects. En effet si T_iT_j est l'ordonnancement partiel, et si $T_iT_jT_k$ (découlant de T_iT_j) est non correct, il est sur que T_iT_j ne peut mener à un ordonnancement complet correct car si T_k ne peut être ordonnancée au plus tôt, elle ne le sera pas au plus tard c'est à dire dans une séquence du genre $T_iT_jT_mT_nT_k$.

[AuBu90]. On distingue des algorithmes d'ordonnancement spécifiques aux tâches périodiques et d'autres appliqués uniquement aux tâches apériodiques. Plusieurs variantes existent selon les caractéristiques des tâches du modèle visé. Nous présentons dans la section 4.2.2 quelques algorithmes d'ordonnancement de tâches périodiques. Le *Rate-monotonic* illustre l'approche statique, *EDF* et *LLF* illustrent l'approche dynamique et peuvent également être appliqués à des tâches périodiques. La section 4.2.3 présente les manières de mettre en œuvre l'ordonnancement des tâches apériodiques.

4.2.2 Ordonnancement préemptif de tâches périodiques

1. L'algorithme à priorités statiques : le *Rate-Monotonic*

L'algorithme du *Rate-monotonic* proposé par Liu et Layland [LiLa73] en 73, est à la base de la théorie sur l'ordonnancement des tâches périodiques. Cette technique se distingue par le fait que les priorités sont fixées statiquement et définitivement. Le *Rate Monotonic* est préemptif et ne s'applique qu'aux tâches périodiques, indépendantes et dont les échéances sont synchronisées avec les périodes (c'est à dire que l'échéance d'une tâche ne peut être antérieure à l'arrivée de sa prochaine occurence, elle doit être égale) [AuBu90], [GMSh94], [SKGo91].

Cette technique consiste à affecter la priorité la plus élevée à la tâche de plus petite période. L'algorithme construit une séquence des tâches caractérisée par une périodicité de longueur L qui a pour valeur le plus petit commun multiple des périodes des tâches déjà acceptées. Ce qui signifie que si la première occurrence de chacune des tâches acceptées est respectée, alors toutes les autres occurrences le seront. La séquence ainsi déterminée peut être reproduite à l'infini.

Le test de garantie du *Rate Monotonic* repose sur la vérification du théorème suivant :

Théorème 1[LiLa73] :

Pour une configuration de n tâches périodiques, une condition suffisante d'acceptation par le Rate-monotonic est :

$$n(2^{1/n} - 1) \geq U \quad \text{où U} = \sum_{i=1}^{n} \frac{E_i}{P_i} \text{ est le taux d'utilisation du processeur}$$

Ce théorème dit qu'un ensemble de n tâches est ordonnançable avec cette technique s'il utilise le processeur avec un facteur de $n(2^{1/n} - 1)$ maximum.

Cette condition résulte d'une analyse dans le pire des cas[1] . Lorsque le nombre des tâches est grand, le facteur d'utilisation tend vers 69% [AuBu90]. D'une façon plus explicite, le processeur reste inutilisé pendant environ 30% de son temps, ceci afin de pouvoir quand même garantir les tâches dans le pire des cas. Cette condition est nécessaire à savoir que toute configuration de tâches qui utilise le processeur pour moins de 69% de son temps peut être garantie par le *Rate-Monotonic*. En revanche, il existe des configurations ordonnançables par l'algorithme mais dont le facteur d'utilisation est supérieur à ce seuil. Dans [LSDi87] on démontre que ce seuil peut être augmenté à 88% si les périodes sont uniformément distribuées et même à 100% si elles sont en harmonie avec la plus petite période.

Une condition nécessaire et suffisante pour garantir l'ordonnancement de tâches périodiques est donnée dans le théorème 2 :

Théorème 2 [ShGo87] [Tind93] :

Une configuration de n tâches périodiques et indépendantes est ordonnançable par le Rate-monotonic si et seulement si :

$$\forall i,\ 1 \le i \le n,\quad \min_{(k,l) \in Yi} \sum_{j=1}^{i} E_j \frac{1}{lP_k} \left\lceil \frac{lP_k}{P_j} \right\rceil \le 1$$

où $Yi = \{(k,l) / 1 \le k \le i \text{ et } l=1,..., \left\lfloor \frac{T_i}{T_k} \right\rfloor\}$

Les crochets entourant les fractions représentent la partie entière de ces fractions. Le *Rate-monotonic* est optimal pour l'ordonnancement de tâches périodiques indépendantes, à savoir que s'il échoue dans l'ordonnancement d'une configuration donnée de tâches, aucun algorithme ne pourra faire mieux. Toutefois, il présente 2 inconvénients majeurs : (1) il ne fournit aucun mécanisme permettant de changer dynamiquement les priorités et de prendre en compte les tâches apériodiques (2) il y a le risque d'inversion de priorité quand les tâches utilisent d'autres ressources que le processeur (tâche de haute priorité bloquée par une autre de moindre priorité pour l'accès à une ressource). Dans 4.2.3 nous

[1] Aussi appelée analyse à l'instant critique, c'est le moment où on suppose que toutes les tâches se déclenchent en même temps.

exposons une adaptation de cette technique pour la prise en compte de tâches apériodiques.

2. L'algorithme de la plus proche échéance d'abord : *EDF (Earliest Deadline First) et ses dérivés*

C'est une technique préemptive qui peut être appliquée de manière statique ou dynamique [ZRSt87b], [DeMo89], [ScZh92]. L'algorithme est aussi connu sous le nom de *Relative Urgency* [Serl72] et *Deadline Driven* [LiLa73]. Le principe est d'exécuter en premier lieu la tâche prête ayant l'échéance la plus proche.

Théorème 3 :

Une condition nécessaire et suffisante d'ordonnancement par EDF de n tâches périodiques indépendantes et dont les échéances sont synchronisées avec les périodes est selon [LiLa73] :

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \leq 1$$

La séquence des tâches périodiques est également déterminée sur une période *L* valant *ppcm(Pi)*. L'algorithme est avantageux par rapport au *Rate-monotonic* car le taux d'utilisation du processeur est de 100%.

EDF étant un algorithme dynamique, il permet outre la garantie des tâches périodiques, de vérifier si les tâches apériodiques peuvent être ordonnancées.

Théorème 4 [ChCh87] [Sill86]

Pour des tâches quelconques ou apériodiques, une condition suffisante est que

$$\sum_{i=1}^{n} \frac{E_i}{R_i} \leq 1$$

Une condition nécessaire et suffisante est que quel que soit Ti, Tj deux tâches d'une séquence T telles que Rj ≤ Ri et Dj ≤ Di, on a:

$$\sum_{\substack{T_k \in T \\ R_j \leq R_k \\ D_k \leq D_i}} E_k \leq R_j$$

Cette dernière condition signifie que dans l'intervalle qui va de la première date de disponibilité (R_j) à la dernière date possible de fin d'exécution (D_i), non seulement T_i et T_j doivent s'exécuter mais également toute tâche dont la priorité est comprise entre celles de T_j et T_i . Il est prouvé que cette technique est optimale pour l'ordonnancement de tâches (avec ou sans préemption) sur un processeur [ZRSt87b], [Herr90], [RSSh90]. La complexité de l'algorithme est en $O(n^2)$.

Une première variante de *EDF* est *LLF* (*Least Laxity first*) qui procède par ordre croissant des temps de latence. A la différence de *EDF*, dans le cas où certaines tâches ont la même échéance, *LLF* peut les différencier. Nous rappelons que les laxités sont dynamiques. Lorsque l'on ne prend en compte que la laxité initiale pour déterminer les priorités, l'algorithme porte alors le nom de *Shortest Slack Time First*.

Une seconde variante optimale est *MUF* (*Maximum Urgency First*) [StKh91] développée afin de guider ces algorithmes dynamiques à distinguer les tâches importantes de celles qui le sont moins. L'urgence est définie par la donnée de deux priorités, une utilisateur (tâche essentielle, critique), et une en fonction de l'échéance. La figure ci-dessous montre un ordonnancement de trois tâches avec EDF et LLF.

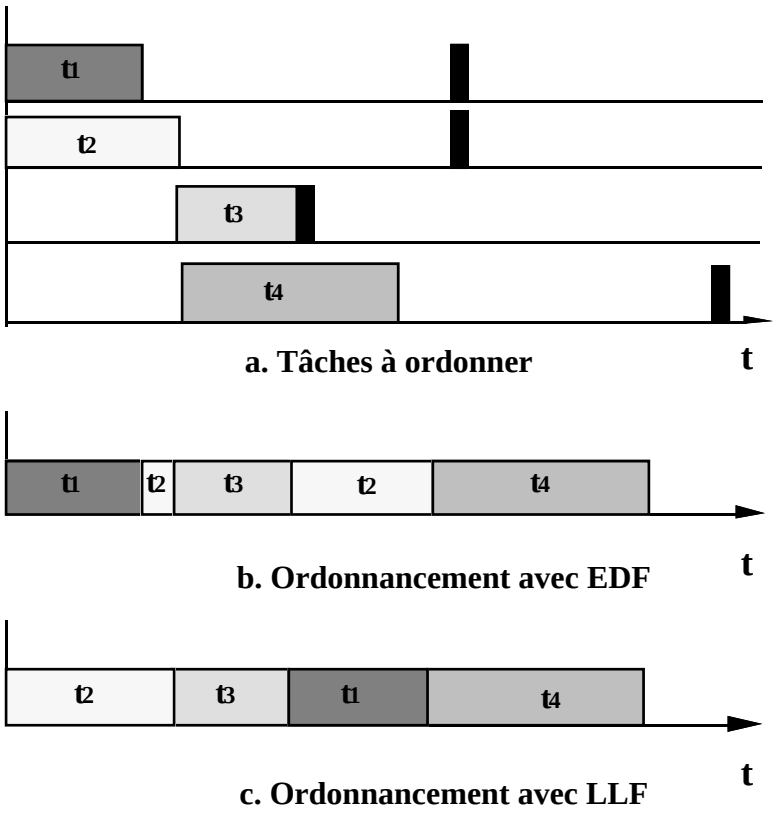


Figure n° 4.2 Comparaison des techniques EDF et LLF avec préemption

Dans la figure 4.2 b, nous présentons un ordonnancement avec la technique EDF des quatre tâches de la figure a. Nous rappelons qu'une tâche en cours d'exécution est interrompue par une autre tâche:

- ayant une échéance plus courte si *EDF* est appliqué
- ayant un temps de latence plus petit si *LLF* est appliqué.

Nous remarquons déjà que bien que la tâche T_2 dispose de moins de temps pour s'exécuter que T_1, c'est T_1 qui est exécutée la première car leurs échéances sont égales et c'est par suite la première arrivée qui sera la première servie. Ensuite à peine T_2 entame son exécution qu'elle est interrompue par T_3 car $D_3 < D_2$. L'exécution de T_2 est reprise sitôt celle de T_3 finie et enfin T_4 est exécutée.

Dans la figure 4.2 c, les mêmes tâches sont reprises avec la technique *LLF*. C'est T_2 qui est exécutée la première puis T_3 dès qu'elle devient prête.

Ces deux techniques donnent toutes deux des ordonnancements corrects. Toutefois, appliquer *LLF* permet d'éviter l'interruption de T_2. Certes, le temps de sauvegarde du contexte est négligeable ainsi que celui de son rétablissement mais pourquoi se rajouterait-on des délais supplémentaires dans un environnement ou un millième de seconde de retard a son influence ?

Les algorithmes présentés dans ce chapitre ont été appliqués et discutés dans divers travaux [Alain92], [BlSc91], [ChCh89], [Clar90], [PDPo93], [ShSa93], [SZGh91]. Il ressort les points suivants de ces algorithmes d'ordonnancement local : (1) le test consiste en la vérification d'une condition ce qui permet d'économiser un ordonnancement inutile, (2) pour les tâches périodiques la détermination de la séquence d'ordonnancement ou de la configuration acceptée sur une durée commune à toutes les périodes est suffisante. Concernant les algorithmes dynamiques, afin de garantir les tâches les plus importantes du système, à savoir celles périodiques, la plupart des systèmes prennent pour hypothèse d'ordonnancer statiquement ces tâches.

4.2.3 Ordonnancement préemptif de tâches apériodiques

La prise en compte de tâches apériodiques est un problème délicat. On s'intéresse désormais à une configuration de tâches périodiques à contraintes strictes dont on doit garantir les contraintes, et de tâches apériodiques à contraintes strictes ou relatives pour

lesquelles on essayera de déterminer un ordonnancement correct ou minimisant les dépassements d'échéances sans pour autant remettre en question l'ordonnancement des tâches périodiques. On distingue deux approches pour mettre en œuvre l'ordonnancement des tâches apériodiques : soit en assurant une prise en compte directe de celles-ci, soit en réalisant un serveur de tâches périodiques.

1. Prise en compte directe des tâches apériodiques

Il s'agit d'utiliser les périodes d'oisiveté du processeur pour essayer d'ordonnancer les tâches apériodiques. Le problème est de bien estimer ces temps d'oisiveté. Nous avons précédemment établi qu'une configuration T de tâches périodiques est ordonnançable sur une période L = ppcm{P_i / i=1,...,n}(avec l'algorithme EDF par exemple). On peut ainsi calculer le temps total d'oisiveté du processeur en le calculant sur chacun des intervalles [kL; $(k+1)L$], k ≥ 0. Nous pensons que dans ce cas, le mécanisme d'ordonnancement local doit commander le déclenchement des tâches, ainsi quand aucune tâche périodique n'est à exécuter, il peut traiter celles apériodiques. L'inconvénient majeur est que le rendement de cette solution dépend du nombre de tâches ordonnancées. Il devient très faible quand de nombreuses tâches périodiques sont ordonnancées avec un temps libre entre les tâches réduit.

2. Utilisation d'un serveur périodique de tâches apériodiques

L'idée est d'adapter le *Rate-Monotonic* de manière à pouvoir gérer des tâches apériodiques. Il s'agit de créer un processus périodique serveur pour gérer l'ordonnancement des tâches apériodiques. Ce serveur ayant toutes les caractéristiques d'une tâche périodique, il est désormais possible d'appliquer le *Rate Monotonic*. Il a été montré dans [Mok83] que *EDF* et ses dérivés restent optimaux en présence de tâches apériodiques. Ils seront donc appliqués par le serveur pour l'ordonnancement des tâches apériodiques.

Plusieurs techniques existent :

(a) Serveur continu

Ce serveur termine sa capacité de service (son temps d'exécution) même s'il n'y a aucune tâche à servir. Ainsi moins de tâches sont différées, en effet si une tâche

apériodique arrive et que le serveur est en cours d'exécution, elle est servie jusqu'à la fin de son temps d'exécution ou jusqu'à la fin du temps alloué au serveur [SSLe89].

L'inconvénient d'un pareil serveur est qu'il peut s'exécuter et ne trouver aucune tâche apériodique à tester, auquel cas son temps de service est perdu. De plus la capacité du serveur est une donnée difficile à déterminer et peut être insuffisante face à de nombreuses créations de tâches apériodiques.

Plusieurs solutions optimisant cette capacité ont été envisagées [LSDi87] :

(b) Serveur scruteur de tâches [SSLe89]

Une fois ce serveur lancé, s'il existe des tâches apériodiques en attente, elles sont exécutées dans la limite de la période du serveur. Le cas échéant, le serveur se suspend jusqu'à sa prochaine période d'occurrence. Cette technique garantit une certaine périodicité dans le traitement des tâches apériodiques. En revanche, le rendement est faible et ceci est dû au fait que le serveur perd sa capacité de service quand il se suspend.

(c) Serveur à échange de priorités [SLSh88], [LSSt87]

Dans ce cas, on attribue au serveur une priorité plus élevée que la plus haute des priorités des tâches périodiques. En quelque sorte c'est comme si ce serveur prenait la main après chaque exécution d'une tâche périodique. Dans le cas où une tâche apériodique survient, elle peut ainsi être testée immédiatement après la tâche en cours d'exécution. Le cas échéant, le serveur échange sa priorité avec la tâche périodique de plus haute priorité. Le serveur va reprendre son temps et la plus haute priorité une fois cette tâche achevée, et ainsi de suite jusqu'à ce que le temps du serveur soit écoulé.

(d) Serveur sporadique [SSLe89], [SGRa88]

Cette solution est particulièrement adaptée aux tâches apériodiques dont le temps de latence est faible. Elle combine les avantages des deux derniers algorithmes. Elle consiste à convertir tous les intervalles de temps CPU libre en tickets et le serveur s'exécute tant qu'il y a des tâches apériodiques à tester. Tandis que dans les précédentes solutions, la capacité du serveur est réinitialisée périodiquement.

Le choix de l'une ou de l'autre des implémentations pour le serveur apériodique est étroitement lié au modèle de tâches considéré. Si les tâches sont peu nombreuses, le

serveur scruteur parait convenable. En revanche, pour des modèles où les tâches ont des temps de latence courts, le serveur à échange de priorité et celui sporadique sont plus adaptés. Un choix définitif nécessite une évaluation des différentes implémentations.

4.3 Ordonnancement multiressources

La question qui se pose pour un ordonnancement de tâches nécessitant d'autres ressources que le processeur est la suivante " *étant donné un certain nombre de demandes en ressources faites par une tâche, peut-on lui attribuer le processeur* ?". Dans cette section, on s'intéresse aux ressources rattachées aux processeurs (fichiers, disques, unités d'entréesortie, etc) mais le processeur n'est pas considéré comme une ressource. L'ordonnancement multiressources est connu pour être NP-complet même dans le cas monoprocesseur [Blaz79]. Certains algorithmes s'exécutant en un temps polynomial existent pour des cas restreints à une ressource.

Par ailleurs, en ce qui concerne l'application des algorithmes précédemment exposés, nous tenons à signaler que EDF n'est plus optimal avec plusieurs ressources, à la différence de LLF qui lui reste optimal, cependant à la condition que les tâches aient les mêmes dates au plus tôt R_i [Herr90]. La figure 4.3 en est une illustration.

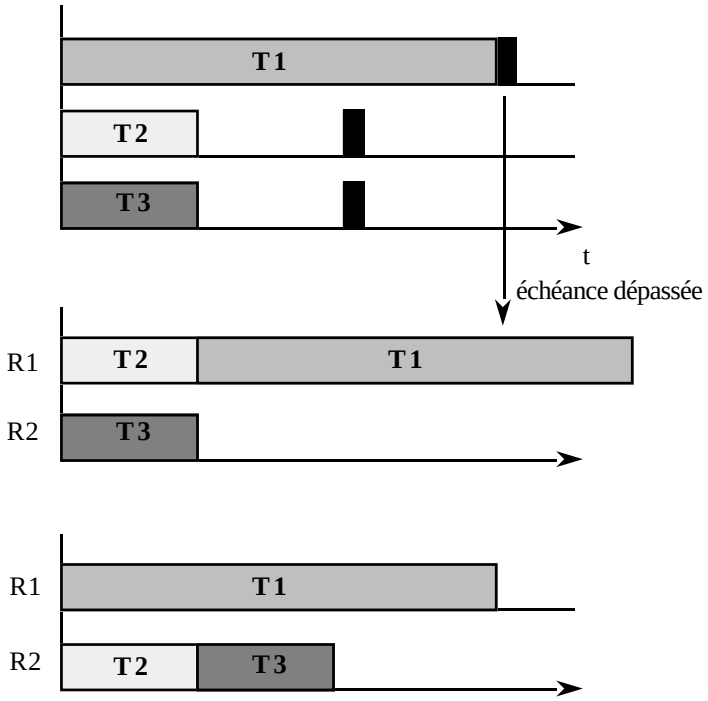


Figure n° 4.3 Ordonnancement avec deux ressources

Le premier ordonnancement est obtenu par *EDF*, le second par *LLF*. On remarque que *LLF* trouve la solution optimale en effet les paramètres R_i sont égaux.

Nous avons remarqué deux manières de procéder dans la littérature :

4.3.1 ordonnancement par quanta de temps

Un ordonnanceur multiressource doit diviser la capacité des ressources entre les tâches de manière à garantir les contraintes. Quand le modèle de tâches est préemptif, le problème revient à répartir l'exécution des tâches sur chaque ressource en quanta de temps et la solution est donnée par la détermination d'un ordre d'exécution de ces quanta de manière à exécuter entièrement une tâche avant son échéance.

Soit Q_k un quantum, associé à QST_k sa date de début et QL_k sa longueur. On note le temps nécessaire à la commutation de contexte. L'hypothèse 2.1 d'ordonnancement correct devient selon [ZRSt87a] :

Un ordonnancement est dit correct si pour toute tâche T_j $j=1,....,n$

$E_j \leq \left(\sum_{Tj \in Qk} QL_k \right)$ - $*K_j$ *où* K_j *est le nombre de fois où* T_j *est interrompue*

et max $(QST_k + QL_k) \leq D_j$ *pour toute Tj contenue dans le quantum* Q_k

à savoir que la somme des quanta finit avant l'échéance de la tâche T_j.

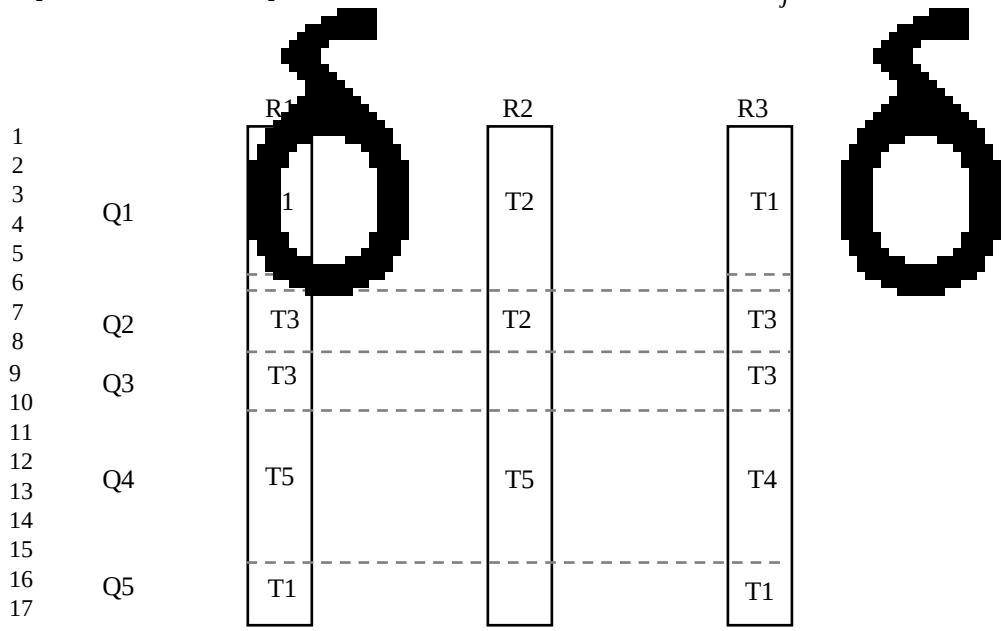


Figure n° 4.4 Un ordonnancement multiressources par quanta de temps

La figure 4.4 illustre un exemple d'ordonnancement préemptif de cinq tâches sur trois ressources : T_1 <R_1,R_3,E_1=7>, T_2 <R_2,E_2=8>, T_3 <R_1,R_3,E_3=4>, T_4 <R_3,E_4=5>, T_5 <R_1,R_2,E_5=5>. p est la durée de la commutation de contexte. On remarque que les quanta sont déterminés après l'ordonnancement des tâches sur les ressources.

4.3.2 Parcours de l'arbre des ordonnancements

Dans [ZRSt87a], est présentée une heuristique afin de guider l'exploration de l'espace des combinaisons par un parcours en arbre, l'algorithme est en O(rn^2) avec *n* tâches nécessitant *r* ressources. De manière à éliminer rapidement les séquences d'ordonnancements partiels non valides, deux vecteurs de mesures sont introduits :

- HD_p : $HD_p(i)$ donne l'heure de disponibilité au plus tôt de la ressource R_i pour un accès en mode partagé.

- HD_e : $HD_e(i)$ donne l'heure de disponibilité au plus tôt de la ressource R_i pour un accès en mode exclusif.

A chaque étape le choix de la tâche à ajouter est guidé par une fonction heuristique H. La tâche retenue est celle qui minimise la fonction *H*. $H(T_i) = E_i + R_i$. Quand une tâche est sélectionnée la valeur des tableaux *HD* est mise à jour par l'addition de $H(T_i)$.

Ainsi, à un niveau donné de l'arbre, en fonction des ressources exigées par les tâches, il est possible de calculer les dates d'activation au plus tôt des tâches (S_i). Des simulations ont été effectuées dans [ZRSt87a] et [ZRSt87b] avec différentes fonctions heuristiques *H* permettant d'appliquer plusieurs algorithmes (par exemple $H(T_j) = D_j$, pour l'algorithme *EDF*). Il en ressort que le taux d'ordonnancement des tâches avoisine 100% quand les temps de latence sont suffisamment grands et essentiellement avec l'application de *EDF* et *LLF*.

4.4. Interférence avec l'ordonnancement d'autres ressources : Correction des problèmes posés par le blocage

Afin d'ordonnancer correctement une tâche nécessitant plusieurs ressources, il est nécessaire de disposer d'une estimation du temps durant lequel elle risque d'être bloquée sur une des ressources auxquelles elle a accès. Dans [Mok83], il est démontré que pour une configuration de tâches périodiques utilisant des ressources en mode exclusif, décider si celle-ci est ordonnançable ou non est un problème NP-complet. En effet, la plupart des

problèmes d'ordonnancement avec des ressources accessibles en mode exclusif le sont également [LRBr77], [GaJo79].

Outre les algorithmes d'ordonnancement multiressources, des mécanismes additionnels doivent être conçus pour pallier à des problèmes pouvant surgir durant l'exécution et entraîner la violation des contraintes temporelles.

Le problème le plus important posé par l'accès aux ressources est le blocage d'une tâche de grande priorité par une autre de moindre priorité pour une durée indéterminée. Soit l'exemple suivant où T_h est de haute priorité, T_b de basse priorité et T_m de priorité moyenne.

Supposons qu'à un moment donné, T_b soit en train de s'exécuter et qu'elle ait accédé en mode exclusif à la ressource R_a. La tâche T_h devient active et préempte T_b , elle ne peut cependant pas avoir l'accès à la ressource R_a car celle-ci est détenue par T_b. On suppose que suite à un temps d'exécution peu précis, la ressource R_a est utilisée plus longtemps par la tâche T_b. Ainsi, T_h va être non seulement bloquée par T_b de moindre priorité, mais également par toute tâche T_m (de priorité comprise entre celles de T_b et T_h) ayant auparavant demandé l'accès à la ressource R_a. Ce phénomène est appelé *inversion de priorité.*

Il existe différentes approches pour minimiser cet effet :

4.4.1 Prévention de l'inversion de priorité

Il s'agit d'empêcher à un moment donné une tâche d'entrer en section critique si une autre tâche de plus haute priorité est susceptible de devenir prête pendant que la première sera en section critique. Pour que cette solution soit applicable, il faut absolument disposer des temps d'exécution et même du temps qu'une tâche va passer dans chaque section critique. Ainsi une tâche T_i ne peut entrer en section critique que si le temps ts_i qu'elle va y passer est inférieur à tl_i (temps libre avant l'occurrence d'une tâche plus prioritaire).

Toutefois, un problème subsiste avec cette approche. En effet, si la condition ci-dessus citée n'est pas vérifiée, la tâche T_i se verra interdire l'accès de la section critique en question et le processeur sera libre pendant $ts_i + tl_i$. Ce scénario peut se répéter plusieurs fois de suite en effet considérons une suite de n tâches $\{T_1, T_2, \ldots T_n\}$ déjà

ordonnancées dont les priorités sont croissantes, chacune devant entrer en section critique. T_1 risque de ne pas entrer en section critique si $ts_1 > tl_1$, de même pour T_2 et ainsi de suite. Arrivé à T_n, il n'y a plus de tâches plus prioritaire, alors T_n *s'exécute* et ensuite T_{n-1}, jusqu'à T_1. Ainsi chaque tâche aura été suspendue pendant au maximum deux fois le cumul des temps d'exécution des tâches plus prioritaires. Cette analyse du pire des cas pour le temps d'utilisation du processeur nous donne ainsi un facteur d'utilisation assez bas (environ 50%).

4.4.2 Protocole à héritage de priorité

Dans le cas où une tâche T_i qui doit entrer en section critique bloque d'autres tâches de plus hautes priorités, ce protocole propose d'attribuer à T_i la priorité de la tâche la plus prioritaire qui sera bloquée par T_i avant de lui céder sa place. A la sortie de la section critique, T_i reprend sa priorité initiale. Cette notion d'héritage de priorité, permet d'une part de ne pas bloquer une tâche hautement prioritaire et d'autre part à T_i de s'exécuter. En effet, si T_i ne faisait que céder sa place, elle pourrait ne jamais s'exécuter à temps si des tâches de moyennes priorités devenaient prêtes. Dans [SRLe90], il est montré que si une tâche peut être bloquée par *m* sections critiques, avec le protocole à héritage de priorité cette tâche sera bloquée au maximum *m* fois. Nous remarquons toutefois que ce protocole ne fournit aucun moyen d'éviter les interblocages (présence de plusieurs sémaphores à l'entrée d'une section critique).

Tous ces inconvénients sont résolus dans le protocole suivant.

4.4.3 Attribution d'une priorité aux sémaphores (the ceiling protocol)

Le principe est que si une tâche T_i verrouille un sémaphore pour l'accès à une section critique, et bloque par conséquent une tâche plus prioritaire T_j , on interdit que d'autres sémaphores (pouvant bloquer T_j) soient verrouillés. Par conséquent une tâche peut être retardée, bien entendu si le sémaphore dont elle a besoin est déjà verrouillé mais aussi si l'accès à un sémaphore (pourtant libre) peut bloquer une tâche plus prioritaire.

Le protocole procède comme suit :

• A chaque sémaphore est associée une priorité qui est celle de la tâche la plus prioritaire qui y aura accès.

• Une priorité *plafond* du système est définie. Elle équivaut à la plus haute priorité de tous les sémaphores bloqués.

• Chaque tâche est dotée d'une autre priorité (dynamique) qui est le maximum de sa priorité initiale et de celles dont elle a hérité (en bloquant des tâches plus prioritaires).

Soit T_i une tâche qui devient prête et qui veut verrouiller le sémaphore *S*. Elle ne pourra y avoir accès que si sa priorité dynamique est supérieure à celle de tous les sémaphores actuellement verrouillés, à savoir la priorité *plafond*. Il est évident qu'un premier sémaphore peut être verrouillé, mais pour qu'un second le soit, il faut absolument qu'il ne risque pas d'être requis par une tâche qui a déjà verrouillé un sémaphore. Ainsi une tâche de haute priorité peut au maximum être bloquée une seule fois (par période) par une tâche moins prioritaire. Dans [PBRa90], il est prouvé que ce protocole empêche la formation d'interblocages.

La figure 4.5 illustre un récapitulatif des algorithmes d'ordonnancement présentés dans ce chapitre:

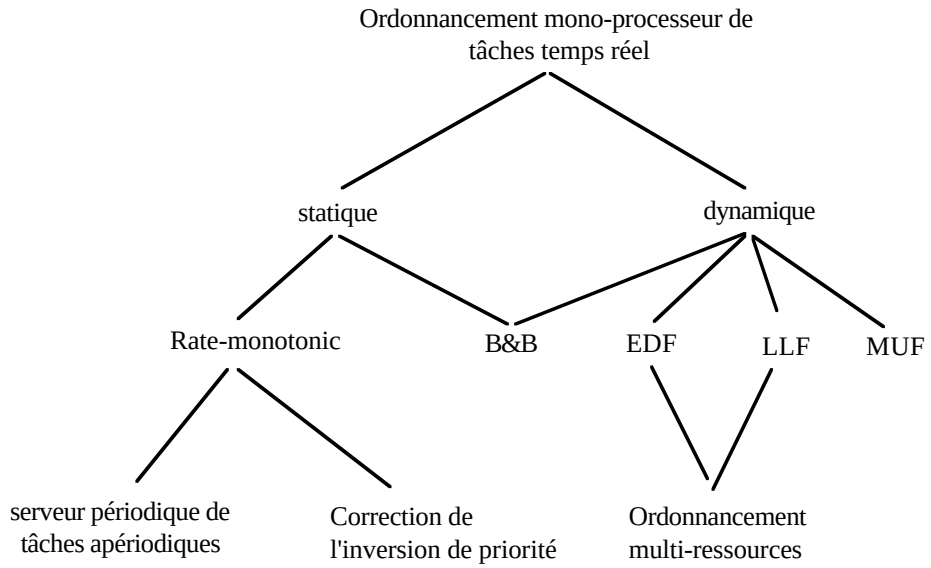


Figure n° 4.5 Une taxonomie des algorithmes d'ordonnancement local de tâches temps réel

4.5 Conclusion

Dans ce chapitre, nous avons présenté divers types d'algorithmes d'ordonnancement local. La diversité de ces algorithmes découle des modèles de tâches pour lesquels ils sont conçus. Il n'existe pas d'algorithme d'ordonnancement local applicable dans l'absolu à n'importe quel modèle de tâches. La philosophie de conception de ces algorithmes est basée sur la vérification d'une condition d'application; ceci permet de ne pas dérouler l'algorithme pour chaque tâche dont il faut tester l'ordonnancement.

Selon que le modèle de tâches est périodique, apériodique et que les tâches communiquent ou nécessitent plusieurs ressources, des problèmes peuvent surgir comme des délais dûs au blocage sur l'attente de l'établissement d'une communication, ou des phénomènes d'inversion de priorités. Les conditions d'acceptation d'un ensemble de tâches par un algorithme donné, sont modifiées en conséquence et les algorithmes sont adaptés afin que la garantie donnée par l'algorithme à travers la condition vérifiée soit toujours assurée.

Chapitre 5

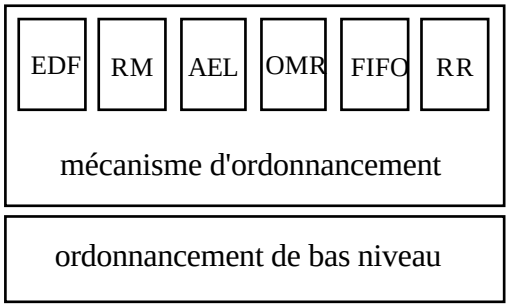
Un algorithme d'ordonnancement local en-ligne

5.1 Structure du mécanisme

Nous présentons dans ce chapitre, la mise en œuvre sur une machine parallèle d'un algorithme d'ordonnancement local de tâches temps réel. L'algorithme a été conçu pour un modèle dynamique, mais il est possible de l'appliquer de manière statique pour ordonnancer un ensemble de tâches avant le début de l'exécution de l'application. L'algorithme est dérivé de la technique LLF, il ordonnance les tâches en fonction de leurs temps de latence.

L'algorithme est le cœur du mécanisme d'ordonnancement local que nous proposons. Le mécanisme doit supporter plusieurs politiques d'ordonnancement afin de choisir la plus appropriée au modèle de tâches appliqué. Ainsi, il supporte des politiques statiques et d'autres dynamiques. Il doit également permettre l'ordonnancement de tâches ayant accès à plusieurs ressources; dans ce but, un algorithme d'ordonnancement multi-ressources est prévu. Il est aussi intéressant de disposer de politiques d'ordonnancement sans contraintes de temps pour des applications où des contraintes de temps ne sont exprimées.

La figue 5.1 présente la structure du mécanisme d'ordonnancement local. L'ordonnancement de bas niveau exploité par le mécanisme est, dans notre implémentation, offert par le processeur. La machine parallèle cible est à base de transputers. La figure 5.1 présente la structure du mécanisme :



Politiques d'ordonnancement

EDF: Earliest Deadline First (première échéance d'abord)
RM : Rate-Monotonic (priorités fixes)
AEL : notre algorithme en-ligne
OMR : Ordonnancement multiressources
FIFO: First In First Out (premier arrivé premier servi)
RR: Round Robin (tourniquet)

Figure n° 5.1 Structure du mécanisme d'ordonnancement local

L'algorithme d'ordonnancement en-ligne est applicable au modèle de tâches suivant : les tâches peuvent être périodiques ou apériodiques, elles peuvent communiquer mais uniquement à la fin de leurs exécutions.

Quand l'application applique un modèle de tâches statique, un ensemble de tâches est présenté au mécanisme, pour celà une routine de garantie est prévue afin de tester l'ordonnancement de cet ensemble. Nous rappellons que ce test consiste à vérifier une équation.

Concernant le cas dynamique, l'algorithme examine les tâches au fur et à mesure qu'elles sont créées ou reçues par le processeur.

5.2 La routine de garantie

Dans le cas où l'ensemble de tâches est uniquement composé de tâche périodiques ou sporadiques et indépendantes, l'algorithme applique une routine de garantie pour vérifier si l'ensemble est ordonnançable. Elle consiste à vérifier l'équation suivante :

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \leq 1$$

Par contre, l'algorithme n'applique pas de routine de garantie si certaines tâches sont apériodiques, auquel cas, la faisabilité de l'ordonnancement est réalisée par le déroulement de l'algorithme. Le temps libre du processeur utilisé pour l'ordonnancement des tâches apériodiques, est déterminé après calcul de la somme des rapports entre les temps d'exécution et les périodes des tâches. Durant cet intervalle de temps, les tâches apériodiques sont ordonnancées et exécutées.

5.3 L'algorithme d'ordonnancement en-ligne

5.3.1 Description

L'intérêt de l'algorithme réside dans l'ordonnancement en-ligne qu'il fournit. Nous précisons que les algorithmes en-ligne présentés dans la littérature sont uniquement décrits par la donnée de la condition à vérifier par les tâches à ordonnancer. En effet, il n'existe par une manière unique pour la mise en œuvre de ces algorithmes.

Notre algorithme fournit un ordonnancement basé sur LLF, et n'implique en aucun cas que les séquences d'ordonnancement construites par l'algorithme seront ordonnées par ordre croissant des temps de latence. Il recherche la meilleure position qui respecte l'approche de conception et qui maximise l'utilisation du processeur. L'algorithme proposé utilise trois listes pour gérer l'ordonnancement et l'exécution des tâches : une première liste de vérification, une seconde liste pour l'ordonnancement et une troisième pour l'exécution. L'algorithme fonctionne selon les étapes suivantes :

(1) enregistrement de la tâche reçue, en tant que tâche à tester. La file d'attente *liste_verification* est prévue pour reccueillir les tâches à tester.Chaque fois que le mécanisme prend la main, la première tâche de la liste, T_i est extraite, et est prise en compte par l'algorithme d'ordonnancement. Celui-ci procède à la :

(2) recherche de l'intervalle de temps [I_debut, I_fin] durant lequel la tâche T_i peut être ordonnancée. Il est clair que l'intervalle est délimité par les paramètres R_i (date au plus tôt) et D_i (échéance). Toutefois, si des tâches ont auparavant été ordonnancées dans cet intervalle, T_i ne pourra commencer à la date R_i et I_debut sera supérieur à R_i . La donnée de l'intervalle de recherche permet outre de connaître S_i, la date à laquelle T_i est ordonnancée pour l'exécution, de calculer le paramètre temps_libre disponible pour l'exécution de la tâche.

Dans le cas où le temps libre donné par l'intervalle [I_debut,I_fin] est insuffisant, une tentative de réorganisation des tâches ordonnancées dans cet intervalle est lancée. Soit T_k, une tâche pour laquelle il y a suffisamment de temps libre et qui est ordonnancée à sa date R_k. Soit la situation, où après avoir accepté T_k (sans pour autant commencer son exécution), l'ordonnanceur reçoit la tâche T_i et que son intervalle de temps ne donne pas suffisamment de temps libre. Si T_k peut être retardée et que ceci libère suffisamment de temps libre pour l'ordonnancement de T_i, peut-on l'accepter ?

Le choix de refuser ou d'accepter une tâche, est assez délicat, car il rentre dans une approche d'ordonnancement qui doit être respectée. Nous avons établi les hypothèses suivantes pour lesquels une tentative de réordonnancement est lancée:

(1) T_i a un temps de latence plus petit que celui deT_k, elle doit donc selon l'approche LLF être ordonnancée avant T_k. L'algorithme vérifie si T_k peut être retardée auquel cas T_i est acceptée et insérée avant T_k.

(2) T_i a une date au plus tôt qui précède celle de T_k. Peu importe les valeurs de temps de latence, si l'exécution de T_k peut être repoussée, T_i est acceptée.

Il est à souligner que ces deux cas de figures ne sont pas exclusifs. Nous voudrions également préciser, que même si le but recherché est un ordonnancement basé sur les temps de latence, la date au plus tôt joue un rôle primordial dans la détermination de la date d'ordonnancement S_i. Le cas de figure n°2 est le seul où une tâcheT_i ayant un temps de latence important est ordonnancée avant une autre tâche T_j plus prioritaire, ceci afin d'éviter que le processeur ne reste inutilement libre.

(3) Une fois une tâche acceptée, un élément est rajouté à *liste_ordonnancement*. L'élément contient l'identificateur de la tâche, son ensemble de paramètres nécessaires à l'ordonnancement, et un couple (S_i, S_i+E_i). La liste contient autant d'éléments que de tâches ordonnancées. Par ailleurs, le fichier contenant le code exécutable de la tâche est inséré à sa position dans une *liste_d'exécution*. Quand la date de début d'exécution d'une tâche est atteinte, le mécanisme la retire de la *liste_exécution* et lance son exécution. La figure 5.2 illustre un exemple de manipulation de la *liste_ordonnancement*. Dans cet exemple, 3 tâches sont ordonnancées dans la liste, et l'algorithme est chargé d'ordonnancer la tâche T_4.

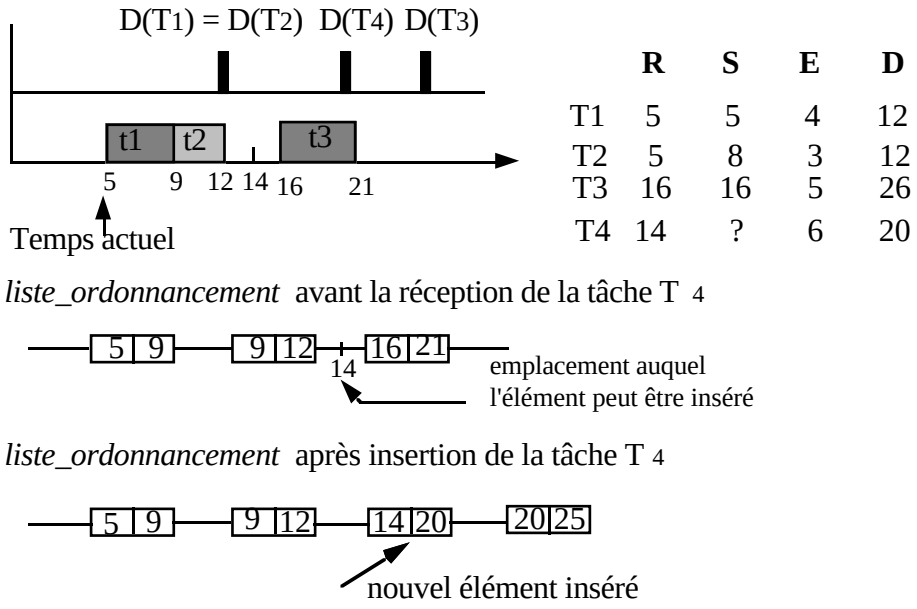


Figure n° 5.2 Exemple d'ordonnancement avec l'algorithme en-ligne

5.3.2 Mise à jour de la *liste_ordonnancement*

Les éléments sont insérés dans *liste_ordonnancement* depuis qu'ils sont pris en compte par le mécanisme et cela jusqu'au début de leur exécution. Afin de réduire les délais apportés par ce mécanisme, il est nécessaire que la recherche dans cette liste ne soit pas trop coûteuse. C'est donc la manière de concevoir et de gérer cette liste qui conditionne la réduction de ces délais.

Actuellement la *liste _ordonnancement* est mise en œuvre à travers une liste chaînée. La recherche d'un élément est en *O(n)* avec n le nombre d'éléments dans la liste. Toutefois, pour une valeur élevée de *n*, il est envisagé que la *liste_ordonnancement* soit gérée par un arbre auquel cas, la recherche se fera en *O(logn)*.

Lors d'un parcours de la liste, les données importantes sont les couples indiquant les dates de début et de fin d'exécution des tâches. En fait, ce qui est intéressant pour une nouvelle tâche à ordonnancer, ce sont les intervalles de temps libres entre les couples. Si certains couples ont des dates de début et de fin d'exécution qui sont les mêmes, il est intéressant de concaténer les éléments correspondants et de réduire ainsi le nombre d'éléments dans la liste. Par conséquent, dans des situations de forte charge (*n* important) deux coûts sont réduits : le coût de recherche dans la liste et le coût mémoire induits par l'allocation de chaque élément de la liste.

Nous signalons que le nombre d'éléments dans la liste augmente considérablement quand l'algorithme est autorisé à interrompre les tâches (version préemptive de l'algorithme).

5.3.3 Présentation de l'algorithme

Cette section présente une version non préemptive de l'algorithme d'ordonnancement en-ligne. Dans les figures 5.3 et 5.4, nous exposons les étapes de calcul intermédiaires. Dans la figure 5.5, nous présentons l'algorithme d'ordonnancement en-ligne. Dans les figures 5.3 et 5.4, nous exposons les tapes de calcul intermédiaires. L'action recherche_I_debut recherche le premier instant à partir duquel la tâche peut être ordonnancée. L'action recherche_temps_libre recherche un intervalle de temps suffisant pour exécuter la tâche en entier. Cette dernière action est appelée par l'algorithme pour chaque tâche à réordonnancer.

```
action recherche_I_debut (tâche_à_garantir)
{
    rechercher le premier élément dont
        I_début est supérieur à R(tâche_à_garantir)
    si R(tâche_à_garantir) < I_fin(tâche_à_garantir)
        I_début(tâche_à_garantir) = I_fin(elm_précédent)
    sinon
        I_début(tâche_à_garantir) = R(tâche_à_garantir)
    si fin liste_ordonnancement atteinte
        temps_libre = D(tâche_à_garantir) - I_début(tâche_à_garantir)
    sinon
        temps_libre = I_début(elm_trouvé) - I_début(tâche_à_garantir)
    si suffisamment de temps libre
        calculer temps de latence (L) de tâche_à_garantir et de elm_précédent
        si R(tâche_à_garantir) < R(elm_précédent) et
        L(tâche_à_garantir) < L(elm_précédent)
            emplacement d'insertion = avant elm_précédent
}
```

Figure n° 5.3 Recherche de l'emplacement d'insertion

```
action recherche_temps_libre (tâche_à_garantir)
{
    si pas suffisamment de temps libre
    Tant que ( temps_libre < E(tâche_à_garantir) ) et
        (I_début(tâche_à_garantir) + E(tâche_à_garantir) < D(tâche_à_garantir))
        rechercher un élément tel que
            R(tâche_à_garantir) < R(élément)
            et L(tâche_à_garantir) < L(élément)
        recalculer temps_libre
        recalculer I_début(tâche_à_garantir)
}
```

Figure 5.4 Calcul du temps libre

L'algorithme d'ordonnancement est présenté dans la figure 5.5 :

```
action ordonnance_tâche (tâche_à_garantir)
{
    recherche_I_debut (tâche_à_garantir)
    recherche_temps_libre (tâche_à_garantir)
    insere un élément pour la tâche_à_garantir
    si non trouvé suffisamment de temps_libre
    {
        sauvegarder dans Sauv_liste l'élément_inséré et son emplacement
        si l'élément_inséré dépasse son échéance
            arret = 1 (arrêter le réordonnancement des éléments
                        après l'élément_inséré)
            défaire
        Tant que (il y a des éléments dans la liste) et
                (arret différent de 1) et
        {
            élément_suivant = élément après l'élément_inséré
            si I_début (élément_suivant) ≥ I_fin (l'élément_inséré)
                arrêt = 1 ( les éléments suivants ne sont pas remis en cause
                            par l'élément inséré)
            sinon
                si élément_suivant dépasse son échéance
                arrêter le réordonnancement et défaire
```

```
                sinon
                  sauvegarder dans Sauv_list  élément_suivant, son emplacement
                  sauvegarder ses anciennes valeurs de I_début et I_fin
                      I_début (élément_suivant) = I_fin (élément_inséré)
                      retirer élément_suivant de liste_ordonnancement
                      calcule_temps_libre (élément_suivant)
                      insere_tâche (élément_suivant)
            }
            si ( défaire = 1)
               Tant que ( il y a des éléments dans Sauv_liste)
                  les restituer à l'emplacement sauvegardé
                  restituer les valeurs de I_début et I_fin
    }
```

Figure 5.5 L'algorithme d'ordonnancement en-ligne

Concernant la concaténation des éléments, à chaque parcours de la liste quand l'ordonnancement d'une tâche se pose, l'algorithme compare les dates de début et fin des couples de chaques éléments consécutifs, et procède quand cela est possible à la concaténation des éléments.

L'algorithme étant dynamique, certaines tâches risquent de dépasser leur échéances sitôt après la tentative d'ordonnancement voire même durant celle-ci. Afin de réduire ce risque, les tâches en attente d'ordonnancement sont ordonnées dans *liste_verification* selon leurs échéances. Ainsi, les tâches les plus urgentes sont traitées les premières. Toutefois, le maintien de cette sécurité supplémentaire nécessite un parcours en *O(n).*

(1) Complexité de l'algorithme

La compléxité de l'algorithme d'ordonnancement nécessite le calcul de compléxité des différentes actions appelées par l'algorithme. Commençons par la recherche de I_début. L'algorithme parcourt la *liste_ordonnancement* en un temps en *O(n)* maximum pour chaque tâche. Une fois l'emplacement de la tâche approximé, le calcul de I_début et temps _libre se fait certes après maintes vérifications, mais sans parcours de liste.

L'action recherche_temps libre se déroule en *O(k)* avec *k* le nombre maximum d'éléments à vérifier. Nous précisons que cette recherche commence à partir de l'élément trouvé par l'action précédente et prend fin dès qu'un élément avec un I_ début supérieur à l'échéance de la tâche à ordonnancer, est trouvé. La valeur de *k* augmente quand les

éléments de la liste ont des temps d'exécution très courts et des dates de début très proches, ainsi il peut y avoir un nombre considérable d'éléments à vérifier avant l'échéance de la tâche. Nous dirons donc que la recherche est en *O(n)* mais est en pratique assez faible.

Concernant l'action de réordonnancement, le temps nécessaire à la réorganisation des tâches et en *O(nlogn)* si toutes les tâches doivent être réordonnancées. Toutefois, ici également, le temps réel est bien inférieur en effet, seules les tâches suivant la tâche insérée risquent d'être réordonnancées. Ainsi, l'algorithme à une complexité en *O(n) + O(n) +O(nlogn) = O(nlogn)* dans le pire des cas.

(2) Preuve d'optimalité

Notre algorithme applique la technique LLF dont l'optimalité a été prouvée dans [ZRSt87b]. Nous donnons donc une brève démonstration d'optimalité. Nous rappelons la définition (cf. chap.2) où un algorithme est optimal si face à l'échec lors de l'ordonnancement d'un ensembe de tâches, aucun autre algorithme ne peut y reussir.

Soit un ensemble T : {T_1, T_2, ..., T_{i-1}}qui sont ordonnancées. Supposons que l'algorithme échoue dans l'ordonnancement d'une tâche T_i , deux cas de figure se posent : dans le premier il n'existe aucun intervalle de temps libre avant l'échéance de T_i. Dans ce cas, il est évident qu'un autre algorithme ne pourra en aucun cas faire mieux. Dans le second cas de figure la valeur de temps_libre est insuffisante. Les tâches étant ordonnancées au plus tôt et n'étant retardées que si une tâche plus prioritaire se présente, l'intervalle de temps_libre sera le même quel que soit l'algorithme d'ordonnancement appliqué, seul l'ordre des tâches pourra changer. Nous en concluons qu'aucun algorithme ne pourra ordonnancer T T_i; Par conséquent notre algorithme est optimal.

5.4 Mise en œuvre et évaluation de l'algorithme

La version présentée de l'algorithme a été implémentée sur le SuperNode, une machine à base de trasputers, massivement parallèle à mémoire distribuée. Un mécanisme d'ordonnancement local est présent sur chaque transputer. Nous précisons que la version actuelle est non préemptive pour des raisons dues à l'architecture du transputer. Le transputer est doté de deux niveaux de priorités pour l'exécution des processus dans l'ordre de leur arrivée : haute et basse. Une file d'attente est conçue pour accueillir les

tâches de chaque priorité. Les tâches de basse priorité ne s'exécutent qu'une fois qu'il n'y a plus aucune tâche de haute priorité [Eldv92].

Afin d'implémenter une version préemptive de l'algorithme, il est nécessaire que l'ordonnanceur manipule lui-même les files du transputer. Il sera implémenté en haute priorité et chaque fois qu'il s'exécutera, il sélectionnera la tâche à exécuter et la lancera en basse priorité. Dans [CSLa93], [AdBo90a], [AdBo90b] cette intervention au niveau des files du transputer est réalisée. Les résultats présentés sont très satisfaisants. Un délai d'environ 1,5 % d'un quantum de temps (2048 μs) est rajouté. D'autres implémentations sur des machines à base de transputers sont discutées dans [VeTh90], [CKMP90], [PaCa91], [Welc90], [SCLa92].

Nous avons appliqué l'algorithme d'ordonnancement à des ensembles de tâches variant de six à 50 tâches. La moyenne du temps d'interarrivée des tâches a été choisie afin d'exprimer trois états du processeur : un état faiblement chargé dans la mesure où les tâches sont suffisamment espacées pour que l'algorithme trouve l'emplacement d'insertion sans difficulté, un état moyennement chargé où l'algorithme a recours au réordonnancement des tâches de temps en temps, et un état hautement chargé où les tâches ont des dates au plus tôt R_i assez proches.

La figure 5.6 présente une moyenne du temps nécessaire à l'ordonnancement d'une tâche en fonction du nombre de tâches dans la *liste_ordonnancement*. L'intervalle séparant l'occurence de deux tâches est un nombre aléatoire *deviation* qui varie entre 0 et 30 ms. $R_{i+1} = R_i + déviation$. Pour un ensemble de 50 tâches, la moyenne des interarrivée entre les tâches était de 14 ms.

Moyenne du temps d'ordonnancement par tâche (μs)	nombre de tâches	nombre de tâches réordonnancées au total	nombre maximum de tâches réordonnancées pour une tâche donnée
360	10	1	1
412	20	3	1
437	30	4	1
454	40	5	1
468	50	7	1

Figure n° 5.6 Moyenne du temps d'ordonnancement pour un état de faible charge

La figure 5.7 présente ces mêmes paramètres pour un processeur moyennement chargé. L'écart entre les arrivées des tâches a été fixé à 2 ms plus un nombre aléatoire *déviation* variant de 0 à 30 ms. $R_{i+1} = R_0$ + (i+1)*2ms + *déviation*.

Moyenne du temps d'ordonnancement par tâche (µs)	nombre de tâches	nombre de tâches réordonnancées	nombre maximum de tâches réordonnancées pour une tâche donnée
364	10	1	2
483	20	8	2
548	30	13	3
649	40	19	5
690	50	25	5

Figure n° 5.7 Moyenne du temps d'ordonnancement pour un processeur moyennement chargé

Le dernier tableau correspond à un système très chargé. $R_{i+1} = R_0$ + *déviation*.

Moyenne du temps d'ordonnancement par tâche (µs)	nombre de tâches	nombre de tâches réordonnancées	nombre maximum de tâches réordonnancées pour une tâche donnée
501	10	4	3
740	20	11	7
874	30	19	7
1009	40	28	5
1089	50	36	8

Figure n° 5.8 Moyenne du temps d'ordonnancement pour un processeur fortement chargé

Les résultats montrent que le temps d'ordonnancement est satisfaisant même en cas de forte utilisation du processeur. Ainsi des tâches dont le temps d'exécution est en moyenne de 10 ms peuvent être ordonnançées par l'algorithme en-ligne avec un délai supplémentaire de 10 % maximum (quand le processeur est très chargé).

On observe également que le nombre de tâches réordonnancées est faible (colonne de droite dans chaque tableau). Ce qui réduit le temps d'exécution de l'algorithme et le rend bien inférieur en pratique au temps théorique.

Concernant la solution adoptée pour l'implémentation du mécanisme d'ordonnancement, actuellement nous avons mis en œuvre le mécanisme en tant que

serveur à échange de priorité. A savoir que l'algorithme en-ligne s'exécute jusqu'à ce que la date d'exécution d'une tâche donnée est atteinte. Auquel cas, il échange sa priorité avec la tâche et reprend la main, une fois l'exécution de celle-ci terminée. Il serait intéressant de pouvoir comparer des implémentations en serveur apériodique et serveur scruteur de tâches.

PARTIE III

Allocation statique de tâches temps réel

Résumé

Un des premiers problèmes qui se posent quand on veut exécuter une application temps réel ou non sur un système parallèle, est celui du placement des tâches de l'application sur les processeurs de la machine cible. Outre la complexité de cette phase, les performances de l'application dépendent fortement de ce placement.

Divers algorithmes d'allocation statique ont été utilisés, avec des critères tels que la réduction du coût des communications, l'équilibrage de la charge entre les processeurs et autres. Quand l'application est sujette à des contraintes de temps, le critère principal pour une allocation correcte devient l'obtention d'ensembles de tâches sur les processeurs qui soient ordonnançables. Des critères tel que la réduction du temps de réponse de l'application ou du coût des communications peuvent alors être pris en compte par la suite afin d'affiner les allocations correctes obtenues.

Les algorithmes d'allocation classiques ne peuvent être appliqués au problème d'allocation de tâches temps réel. Ces derniers doivent être adaptés et associés à des algorithmes d'ordonnancement local afin d'atteindre le but recherché.

Dans le chapitre 6, une analyse critique des différents travaux réalisés dans ce domaine est présentée. Nous proposons dans le chapitre 7 une approche originale pour résoudre ce problème. Elle consiste à utiliser des algorithmes génétiques inspirés de la théorie d'évolution des espèces. Un algorithme génétique parallèle est proposé et évalué [BaMu95].

Chapitre 6

Techniques d'allocation statique de tâches temps réel

6.1 Introduction

Nous nous intéressons à présent à une phase antérieure à l'ordonnancement des tâches et qui concerne la répartition d'un ensemble initial de tâches sur un ensemble de processeurs.

Il est nécessaire de réaliser une allocation de ces tâches de manière judicieuse en effet cette étape conditionne fortement les performances de l'application. L'étape d'allocation obéit à plusieurs critères, certains doivent être rigoureusement respectés (limitation de la taille mémoire d'un processeur, contraintes temporelles, localité physiques des ressources...), d'autres doivent être considérés afin que l'allocation soit peu coûteuse (équilibrage de la charge de travail des processeurs et des communications...).

Divers algorithmes sont proposés dans la littérature pour résoudre ce problème. Nous estimons que l'allocation de tâches temps réel peut être résolue avec les mêmes approches appliquées pour l'allocation classique de tâches en effet, le problème est le même, c'est uniquement la formulation du modèle et des contraintes qui change. Que ce soit en environnement temps réel ou non, il s'agit d'allouer un ensemble de tâches à un ensemble de processeurs en tenant compte de certains critères. Toutefois, étant donné le fait que les critères changent, la philosophie adoptée pour appliquer ces approches et concevoir les algorithmes doit être révisée. Une allocation de tâches temps réel n'est considérée correcte que si le résultat suivant est vérifié :

Hypothèse 6.1

Une fois l'allocation terminée, on obtient sur chaque processeur un ensemble de tâches pour lesquelles un ordre d'exécution peut être déterminé en respectant toutes les contraintes temporelles.

Le problème d'allocation statique d'un ensemble de tâches sur un ensemble de processeurs est connu pour être NP-complet [GaJo79], également le problème de l'ordonnancement de ces tâches sur un processeur et celui de l'ordonnancement des communications [TBWe92]. Il ne faut pas pour autant considérer ces trois problèmes séparément afin d'éviter de rendre leur résolution plus complexe. Dans le cas précis de systèmes temps réel, ces problèmes sont étroitement liés, et il est souvent difficile d'obtenir des solutions optimales (voire même faisables), si la résolution de ces problèmes se fait de manière successive.

Il est nécessaire à notre avis d'aborder le sujet avec une vue globale et de réaliser à la fois l'allocation et l'ordonnancement en essayant de réduire ou d'équilibrer certains coûts comme la communication, principal handicap des systèmes parallèles et distribués (nous précisons toutefois que les unités de mesures des coûts dans chacun de ces systèmes sont très différentes). De plus, la prise en compte de l'ordonnancement permet de guider l'allocation et d'éliminer des solutions absolument infaisables (soit l'exemple de tâches ayant les mêmes paramètres R_i, E_i et un temps de latence réduit allouées au même processeur), et voire même de ne pas écarter des solutions partielles qui peuvent aboutir à des solutions correctes.

Ce chapitre présente un état de l'art des techniques utilisées. Les algorithmes exposés sont regroupées en fonction du modèle considéré. On distingue :

• le modèle de tâches périodiques indépendantes où chaque instance d'une tâche est exécutée une fois durant sa période avant l'occurrence de la prochaine instance. C'est le modèle de base où chaque tâche doit être ordonnancée après sa date R_i et finie avant P_i (sa période). Le paragraphe (1) de la section 6.2.2 expose une solution à ce problème.

• le modèle de tâches périodiques reliées par des contraintes de précédence et éventuellement des communications et des duplications. On a alors un graphe de tâches. Ce modèle est plus complexe en effet il introduit en sus des contraintes temporelles, des contraintes de synchronisation et de localité. Diverses heuristiques sont présentées dans

la section 6.2.2 (paragraphes (2) (3) (4)) ainsi que des algorithmes optimaux (section 6.2.1).

• le modèle de tâches périodiques communicantes où on introduit une nouvelle forme de contraintes temporelles : un intervalle minimum entre deux occurrences d'une tâche, ainsi que des communications asynchrones et cycliques. Une heuristique est présentée au paragraphe (3) de la section 6.2.2.

6.2 Méthodes d'allocation statique de tâches temps réel

Parmi les techniques appliquées pour l'allocation statique de tâches temps réel, on distingue:

• les algorithmes optimaux ou exacts qui trouvent toujours la solution optimale mais qui sont très coûteux en temps CPU et en mémoire.

• les heuristiques qui permettent d'obtenir rapidement des solutions satisfaisantes en réduisant la dimension de l'espace de recherche.

Nous tenons à préciser que nous ne présentons que les travaux appropriés au modèle de tâches avec contraintes temporelles. Il existe une multitude de travaux sur l'allocation de tâches sans contraintes. Un état de l'art est présenté dans [TaMu91]. Certains se sont également penchés sur les problèmes de temps, mais les solutions sont généralement spécifiques au problème ou bien consistent essentiellement à réduire le temps de réponse de l'application ainsi que le coût des communications et non de respecter les contraintes de temps. Nous citons [BDWe86], [Blak92], [CaDo94], [ChYu90], [ChLa87], [Hous87], [Lo88], [XHXi88],...

6.2.1 Les algorithmes optimaux

Parmi les algorithmes exacts, on trouve les techniques issues de la théorie des graphes, les algorithmes de programmation mathématiques (B&B (Branch and Bound) aussi appelés *procédures par séparation et évaluation*, programmation dynamique,...)

De nombreux travaux dans la littérature ont traité le problème de l'allocation temps réel par des techniques issues de la théorie des graphes [HZCa88], [Hous87], [Huan85], [ChLa87], mais, ils avaient pour objectifs de réduire le temps de réponse de l'application ainsi que le coût des communications et non de respecter des contraintes

telles que celles sur les échéances. Nous remarquons que pour appliquer ces algorithmes à notre problème, il est nécessaire de pouvoir exprimer les contraintes de temps dans une modélisation en graphe. Cette technique ne fournit aucun mécanisme permettant ceci. L'extension de l'algorithme afin d'inclure de telles contraintes est très complexe. De plus, les techniques de théorie de graphes n'appliquent pas de vue globale en essayant de réaliser à la fois l'allocation et l'ordonnancement des tâches.

En ce qui concerne les techniques issues de la programmation mathématique. La formulation du problème en tant que problème d'optimisation combinatoire permet à travers une fonction coût d'exprimer les contraintes de temps. La résolution du problème se fait pas la suite en appliquant des algorithmes tels que : la programmation dynamique [HaGi86], les procédures par séparation et évaluation (B&B) [ChYu90]. L'avantage de cette technique est qu'elle permet de mieux formaliser le problème, toutefois ses besoins en temps mémoire et CPU croissent exponentiellement avec la taille du problème.

Toutefois les algorithmes B&B ont été appliqués avec succès pour la résolution de l'allocation statique de tâches temps réel en prenant en compte l'ordonnancement des tâches.

Nous rappelons que les algorithmes B&B consistent à explorer l'espace de recherche composé des combinaisons des tâches sur les processeurs (voir definition section 4.2.1).

Souvent l'espace de recherche et représenté par un arbre, nous l'appellerons désormais arbre des combinaisons entre les tâches. La recherche commence par la racine qui correspond à l'ensemble vide, et à chaque niveau dans l'arbre, on explore les différentes branches. Les algorithmes B&B sont souvent associés à une fonction heuristique utilisée pour l'évaluation des nœuds de l'arbre afin d'éliminer ceux qui ne peuvent pas conduire à une allocation finale correcte. En effet l'arbre comporte m^n assignations possibles pour n tâches à placer sur m processeurs (avec un ordre quelconque des tâches), et l'évaluation de celles-ci peut nécessiter des jours entiers. Le parcours de l'arbre est guidé à chaque niveau de profondeur par les valeurs minimales de cette fonction (le successeur d'un nœud donné est l'allocation de coût minimal). La figure 6.1 illustre un parcours d'arbre pour le placement de trois tâches sur deux processeurs. Les noeuds représentés par des cercles non hachurés concernent la recherche d'une solution pour des tâches dont l'ordre d'exécution est quelconque sur le processeur. Les

nœuds hachurées sont à rajouter à l'arbre car les tâches doivent être ordonnancées sur chaque processeur.

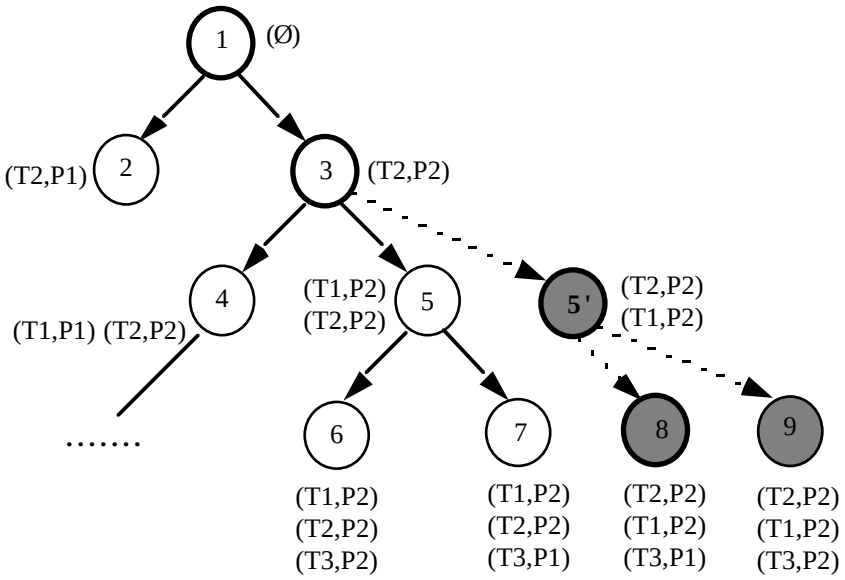


Figure n° 6.1 Arbre de recherche pour le placement de trois tâches sur deux processeurs

L'arbre à explorer pour le placement de trois tâches sur deux processeurs, est constitué des nœuds de 1 à 7. Comme les tâches doivent également être ordonnées, d'autres nœuds sont rajoutés, ainsi le nœud 3 va donner naissance au nœud 5' qui lui même va donner naissance aux nœuds 8 et 9. Ainsi, utiliser des algorithmes B&B pour parcourir cet arbre peut donner le chemin (1,3,5',8)

Dans [PeSh89], il est présenté une approche pour la résolution de l'allocation statique de tâches temps réel périodiques basée sur deux algorithmes B&B. Le premier **B&BA** (allocation) détermine une allocation optimale des tâches aux processeurs. Le second **B&BS** (scheduling) détermine un ordonnancement optimal des tâches allouées aux processeur.

Dans leur graphe de tâches, Peng & Shin utilisent les contraintes de précédence pour représenter les communications entre les tâches, en effet les tâches sont composées de modules de calcul et de communication. Parmi ces derniers on trouve un module pour l'émission d'un message et un pour la réception. A chaque module est associé un temps d'exécution. Les communications peuvent être synchrones ou asynchrones. Pour la détermination de la solution optimale, l'algorithme utilise l'algorithme B&BS pour calculer le coût d'une allocation complète et ce dernier est utilisé par le B&BA pour dériver une allocation optimale.

Dans [HoSh94] deux algorithmes B&B sont appliqués à un modèle de tâches périodiques communicantes et nécessitant des duplications. L'originalité de ces travaux réside dans une méthode pour la détermination des tâches à dupliquer ainsi que le nombre des duplications. Les auteurs se basent sur le temps de latence d'une tâche pour ce choix.

Les algorithmes B&B ont l'avantage de trouver la solution optimale, cependant le temps d'obtention des solutions est trop important. De plus, l'heuristique utilisée pour guider la recherche dans l'arbre des combinaisons doit être bien définie afin de trouver la solution optimale. Alors pourquoi chercherait-on la solution optimale alors qu'une allocation correcte est définie par un ordonnancement correct sur tous les processeurs ? Des heuristiques ont donc été proposées.

6.2.2 Heuristiques

Nous définissons les heuristiques comme des méthodes permettant de choisir parmi différentes orientations celle qui est la plus efficace vu les objectifs à atteindre. On distingue une classification en *algorithmes spécifiques* et *généraux.* Ces derniers sont applicables à une grande variété de problèmes d'optimisation. Parmi les algorithmes spécifiques qui ont été appliqués à l'allocation de tâches temps réel, on distingue le groupement de processus. Parmi les algorithmes généraux, les techniques les plus utilisées sont le recuit simulé, les algorithmes itératifs de recherche locale, les réseaux de neurones [CaMa93] et les algorithmes génétiques.

Les heuristiques peuvent également être classées en :

• *algorithmes gloutons* qui démarrent avec une solution partielle (uniquement une partie des tâches est allouée) qu'ils cherchent à étendre à chaque étape de manière à obtenir une solution complète. Un choix est définitif, ces algorithmes n'autorisent pas les retours en arrière. Ainsi, les solutions obtenues sont très dépendantes de l'ordre de considération des tâches.

• *algorithmes itératifs* qui sont initialisés par une solution complète (toutes les tâches sont allouées) qu'ils cherchent à améliorer à chaque étape jusqu'à l'obtention d'une solution complète. La complexité de ces algorithmes dépend de la solution initiale. Leur complexité est en général plus grande que celle des algorithmes gloutons mais les résultats sont meilleurs.

Nous présentons dans ce qui suit des algorithmes spécifiques l'un basé sur l'exécution d'un algorithme d'ordonnancement afin de minimiser le nombre de processeurs nécessaires à l'allocation, l'autre sur le groupement de processus afin de réduire le coût des communications. Ensuite nous exposons deux algorithmes généraux itératifs, le premier appliquant le recuite simulé et le second les algorithmes génétiques.

(1) Utilisation d'un algorithme d'ordonnancement local

Le problème de l'allocation statique peut également être résolu par l'exécution d'algorithmes d'ordonnancement local afin de construire les listes d'ordonnancement sur chaque processeur [DaDh86]. Il s'agit de classer des tâches périodiques selon leur facteur d'utilisation $U_i = (E_i / P_i)$ et de former des ensembles de tâches en se basant sur la condition d'ordonnancement du *Rate-Monotonic*.

Soit $[T_i]$ i = 1.., n l'ensemble des tâches à allouer, et $[U_i]$ i= 1..n, leur facteurs d'utilisation respectifs. L'ensemble des tâches est divisé en M classes.

Une tâche T_i appartient à la classe-k si $(2^{1/(k+1)} - 1) < U_i \leq (2^{1/k} - 1)$ et appartient à la classe M si $0 < U_i \leq (2^{1/M} - 1)$. On obtient ainsi la décomposition suivante:

Classe-k	Intervalle du facteur d'utilisation
1	$[2^{1/2} - 1, 1]$
2	$[2^{1/3} - 1, 2^{1/2} - 1[$
3	$[2^{1/4} - 1, 2^{1/3} - 1[$
. .	
M	$[0, 2^{1/M} - 1[$

L'ensemble des processeurs est également partitionné en M classes. Un processeur est dit de classe-k s'il va recevoir des tâches de la classe-k. Le principe est donc d'attribuer à un processeur des tâches d'une même classe.

L'algorithme présenté (Next-Fit-Monotonic dérivé du *Rate-monotonic*) pour le groupement des tâches a pour objectif d'allouer les tâches d'une classe-k donnée par groupes de k à chaque processeur classe-k. Supposons que la classe-4 contienne dix tâches, il y aura regroupement des tâches par quatre, et il leur sera attribué trois processeurs (au dernier seront allouées deux tâches). Ce regroupement des tâches par k permet de garantir leur exécution, en effet leurs facteurs d'utilisation vérifient l'équation

$$k(2^{1/k} - 1) \geq \sum_{i=1}^{k} U_i \qquad (1)$$

Le nombre final de processeurs nécessaire pour exécuter les n tâches est la somme des processeurs classe-k (déterminé par l'algorithme) pour toutes les classes. Toutefois, ce nombre est optimisé afin de diminuer le nombre de processeurs auxquels sont allouées moins de k tâches. Les tâches allouées à ces processeurs sont redistribuées entre les processeurs. Comme aucun processeur n'est exploité à 100 %, (puisque l'algorithme Rate-monotonic atteint un taux d'utilisation de 69 % maximum cf. chap. 4) il suffit de comparer le taux d'utilisation du processeur pour une tâche donnée avec la capacité du processeur non exploitée par l'ensemble des tâches allouées. Dans [DaDh86], la complexité de l'algorithme en *O(n)* est prouvée et il est recommandé de l'appliquer avec $4 \leq M \leq 12$.

La figure 6.2 illustre l'allocation de 16 tâches avec le *Rate-monotonic*. Ces tâches sont organisées en quatre classes. Chaque tâche de la classe-1 est allouée à un processeur différent. La classe-3 nécessite deux processeurs puisqu'elle contient plus de trois tâches, la classe-4 également. Ce regroupement en classes nécessite sept processeurs. Cependant, on a pu réduire ce nombre à six en plaçant la tâche 9 non allouée de la classe-3 avec la tâche 12, et on vérifie bien que $2(2^{1/2}-1) = 0,82 \geq 0,33 + 0,25$. On a ainsi réalisé une allocation correcte puisque sur chaque processeur, l'équation (1) est vérifiée.

Ti	Ei	Pi	Ui
T1	1	6	0,16
T2	1	6	0,16
T3	1	6	0,16
T4	1	3	0,33
T5	1	6	0,16
T6	1	3	0,33
T7	1	6	0,16
T8	1	6	0,16
T9	1	4	0,25
T10	2	2	1
T11	2	8	0,25
T12	3	6	0,33
T13	1	4	0,25
T14	1	6	0,16
T15	1	4	0,25
T16	3	17	0,17

classe-1 {T10, T12}
classe-2 {T4, T6}
classe-3 {T9, T11, T13, T15}
classe-4 {T1, T2, T3, T5, T7, T8, T14, T16}

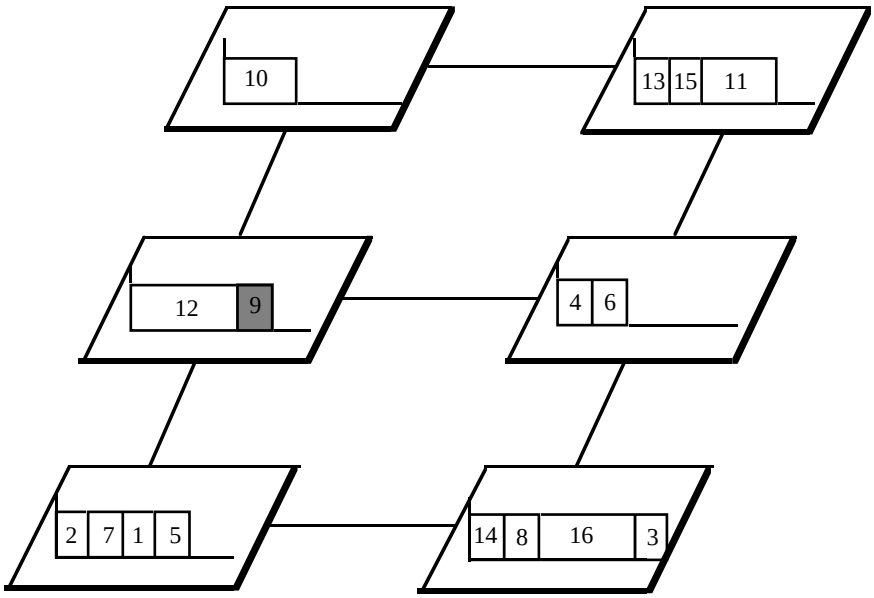


Figure 6.2 Exemple d'allocation en appliquant le *Rate-monotonic*

Cette heuristique permet d'allouer des tâches indépendantes mais elle n'est pas applicable aux tâches communicantes ni à celles soumises à des contraintes de précédence.

(2) Groupement de processus (groupement par paires de tâches)

Dans [Rama90], Ramamritham présente une heuristique pour l'allocation statique de tâches temps réel avec des considérations de tolérances aux fautes (par duplication de tâches) et d'ordonnancement des communications. Le regroupement des tâches est déterminé par la quantité d'information échangée entre deux tâches et par le temps d'exécution de chacune d'elles. Les délais de communication sont prévisibles. Le médium utilisé est un réseau à accès multiple et applique le protocole TDMA (Time Division Multiple Access).

Les tâches du modèle sont périodiques et chacune est constituée de sous-tâches. Ces dernières ne communiquent que pour envoyer des résultats à la fin de leur exécution. Les communications n'ont lieu qu'à l'intérieur des tâches. Les sous-tâches peuvent être dupliquées pour des raisons de tolérance aux pannes, avec la contrainte d'être placées sur des sites différents. L'algorithme applique l'heuristique LST/MISF (Latest Start Time/ Maximum Immediate Successors First) et a pour but d'allouer les sous-tâches et de construire un ordonnancement de longueur L = PPCM(P_i).

Le principe de l'heuristique pour l'allocation est le suivant :

- Construire le graphe des tâches sur la période L
- Construire le graphe des communications
- répéter
  - à chaque niveau dans le graphe
    - regrouper les sous-tâches non communicantes ou dont le coût de communication est faible et les allouer au même nœud
  - vérifier les contraintes temporelles

jusqu'à obtention d'une allocation correcte

Pour la construction du graphe des communications, on calcule pour chaque paire de sous-tâches communicantes le rapport entre la somme de leurs temps d'exécution et la quantité d'information à échanger entre ces sous-tâches. On définit CF un facteur de

communication, si ce rapport est inférieur à CF, l'arc reliant ces deux sous-tâches sera pondéré de 0 et elles seront allouées au même site. Il est évident que le placement le moins coûteux en communication sera le placement initial où toutes les sous-tâches sont allouées à un même processeur et où la valeur de CF est le le maximum des rapports calculés. Toutefois, les contraintes de temps ne seront pas vérifiées.

Les sous-tâches sont allouées par ordre croissant de leur date LST (Last Start Time) qui est calculée comme suit : échéance de la tâche de laquelle on retranche la somme des coûts de calcul et de communication du chemin menant de la sous-tâche à sa période dans le graphe. Dans le cas où il y a conflit entre deux sous-tâches, celle qui a le plus de successeurs est choisie. L'heuristique propose d'allouer les sous-tâches par paires si l'arc les reliant est pondéré de 0.

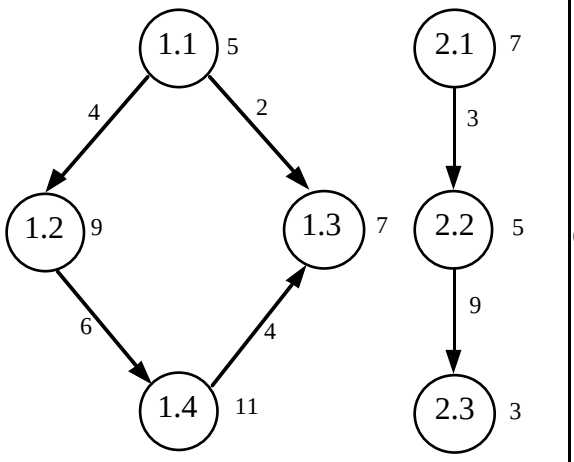
Concernant l'ordonnancement des sous-tâches, un test permet de vérifier si une sous-tâche est ordonnancée à une date postérieure à sa date LST, auquel cas l'allocation courante est écartée.

Nous tenons à préciser qu'à l'origine cet algorithme est dynamique mais il peut également être appliqué de manière statique tel que nous l'avons décrit. Dans un modèle dynamique, les modifications essentielles sont au niveau de l'ordonnancement. En effet une sous-tâche donnée ne peut être allouée à un nœud donné si deux cas de figure se présentent :

1- le temps courant sur le nœud en question est supérieur à sa date LST

2- l'intervalle de temps entre le temps courant et la fin de la période L est insuffisant pour terminer les prédécesseurs de la sous-tâche ainsi que leurs communications (celles-ci étant synchrones).

Dans la figure 6.3 ci-dessous, nous présentons un exemple avec deux tâches périodiques. Le nombre à droite de chaque sous-tâche représente son temps d'exécution, celui rattaché à l'arc reliant deux sous-tâches représente la quantité d'informations à communiquer (L'auteur Ramamritham définit le coût de la communication comme étant égal à la quantité d'information). Dans la figure 6.3, la $k^{ème}$ instance de la $j^{ème}$ sous-tâche de la tâche périodique i est notée i.j.k. Le dernier nombre indique le numéro de la duplication. Dans ce cas, uniquement la sous-tâche 1.2.1 nécessite une duplication. Le rectangle à droite de chaque sous-tâche représente sa date LST. Par exemple LST(1.2.1.3) = 50 - 11 - 6 - 9 = 24.



Tâche périodique 1
Période : 50
Sous-tâches : 1.1, 1.2, 1.3, 1.4
Sous-tâche 1.2 à dupliquer 3 fois

Tâche périodique 2
Période : 25
Sous-tâches 2.1, 2.2, 2.3

Figure n° 6.3 Structure de deux tâches périodiques

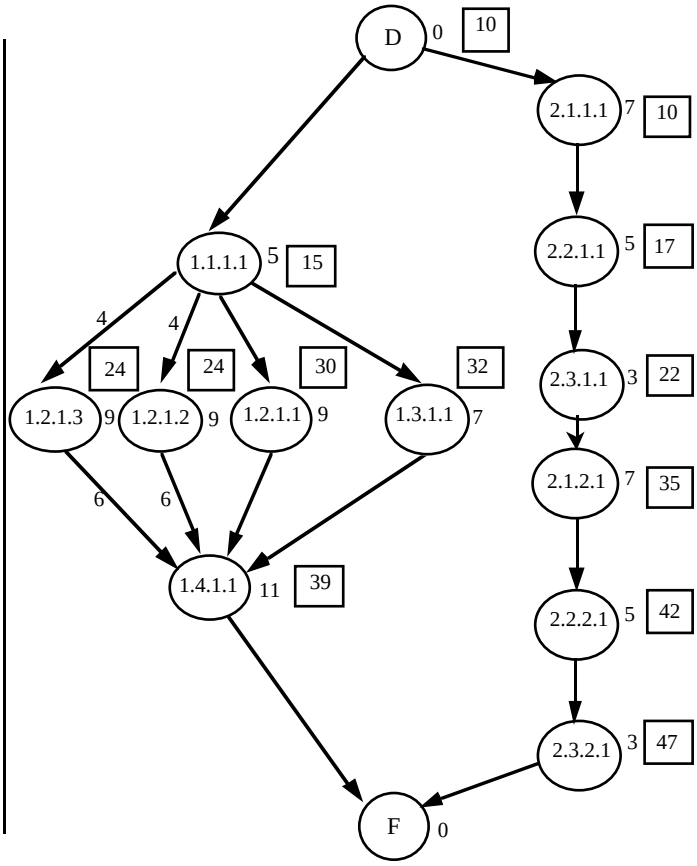


Figure n° 6.4 Graphe de communications

L'algorithme déroulé sur cet exemple donne l'ordonnancement suivant pour les sous-tâches et les communications associées :

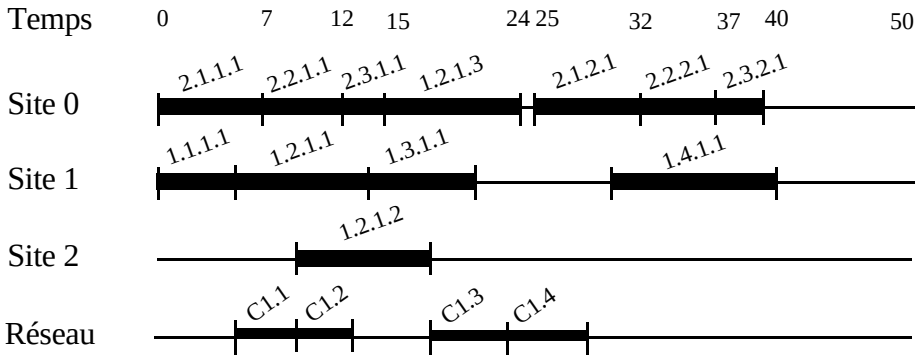


Figure n° 6.5 : allocation et ordonnancement de T_1 et T_2

C1.1 représente la communication entre les sous-tâches 1.1.1.1 et 1.2.1.2

C1.2 représente la communication entre les sous-tâches 1.1.1.1 et 1.2.1.3

C1.3 représente la communication entre les sous-tâches 1.2.1.2 et 1.4.1.1

C1.4 représente la communication entre les sous-tâches 1.1.1.3 et 1.4.1.1

Cet algorithme a été appliqué dans [Rama90] à une centaine de graphes comprenant de 100 à 150 sous-tâches à placer sur 6 nœuds. Le pourcentage d'allocations correctes trouvées est élevé, mais il est étroitement dépendant de la période des tâches, des quantités d'informations à communiquer et par conséquent du paramètre CF.

(3) Le recuit simulé

Cette méthode est inspirée des techniques de cristallisation dans la sidérurgie. Le recuit simulé (RS), introduit par [KGVe83], a été appliqué avec succès dans divers travaux pour résoudre le problème de l'allocation de tâches aux processeurs [BoMi88], [Talb91]. Dans [TBWe92], [BNTZ93] et [ChAg94] il est appliqué pour des tâches temps réel.

Pour la recherche de la solution optimale, le recuit simulé procède à travers la génération de divers états d'énergie (chacun correspondant à une solution) commandés par la réduction de la température jusqu'à l'obtention du plus faible point d'énergie qui représente la meilleure solution. L'algorithme commence par générer la configuration initiale *conf_i* qui représente un point de l'espace des allocations possibles. A la configuration initiale sont associées une température et une énergie élevées. On définit le voisinage d'un point de l'espace comme l'ensemble des solutions qui peuvent être obtenues en déplaçant une tâche d'un processeur à un autre. L'algorithme procède par la génération et l'évaluation (grâce à une fonction coût) de configurations voisines. Cette politique d'exploration des solutions avoisinantes permet d'éviter le blocage dans des minima locaux. L'algorithme du récuit simulé est présenté ci-dessous :

- E_0 : énergie de *conf_i*
- T_0 : T_{max} (température de démarrage)

Le but du RS est de minimiser l'énergie (celle-ci est calculée par une fonction coût qui modélise les objectifs). Le processus suivant est réitéré jusqu'à l'équilibre thermique :

- génération d'une configuration voisine *conf_v*
- estimation de *conf_v* : une configuration est acceptée suivant une probabilité P_{RS} qui dépend de son coût et de la température courante : $(E_{cour} - E_{conf\_v}) / T_{cour} \geq P_{RS}$
- *Tcour* est mise à jour et le processus est réitéré à un autre pallier de température

L'équilibre est atteint une fois qu'un nombre suffisant de générations est atteint.

Dans [TBWe92], l'algorithme est appliqué à un modèle de tâches périodiques ayant les caractéristiques suivantes :

• certaines tâches sont dupliquées. Les duplications doivent nécessairement être placées sur des processeurs différents.

• l'architecture cible est à base de processeurs reliés par un bus dans [TBWe92]. Dans [BNTZ93] les processeurs sont reliés par un réseau d'interconnexion point à point DIA (Data interaction architecture)

• les délais de transmission sont bornés. Le protocole de communication est à jeton.

La fonction d'énergie appliquée pour évaluer une allocation est la suivante :

$$E = k_1 E_{replica} + k_2 E_{mem} + k_3 E_{sched} + k_4 E_{bus}$$

où :

- $E_{replica}$ représente le nombre de tâches dupliquées et placées sur le même processeur,

- E_{mem} représente la somme des dépassements de mémoire sur chaque processeur (utilisations mémoire supérieures à 100%),

- E_{sched} représente la somme des pénalités sur chaque processeur. Une pénalité de x est attribuée à une tâche qui dépasse son échéance de x unités de temps,

- E_{bus} représente l'utilisation du bus.

Les poids k_1, k_2, k_3, k_4 mesurent l'importance de certains paramètres par rapport à d'autres. En effet, on peut aboutir à une allocation infaisable mais avec une faible utilisation du bus, ce qui n'est pas l'objectif à atteindre.

Le recuit simulé a également été appliqué dans [ChAg94] pour un modèle de tâches avec des contraintes particulières :

• le modèle est destiné à des applications où sont exprimés des intervalles de temps minimum entre l'ordonnancement de deux occurrences d'une tâche,

• le modèle autorise l'exécution de plus d'une occurrence d'une tâche durant sa période,

• les communications peuvent être cycliques c'est à dire par exemple que la j^{ème} occurrence d'une tâche T_i soit T_i^j peut envoyer un message à T_k^j et que cette dernière

renvoie un message à T_i^{j+1} l'occurrence suivante de T_i . On parle alors d'émission dans la première communication et de réception dans la seconde (celles-ci sont toujours considérées par rapport à une même tâche).

La fonction d'énergie choisie modélise les écarts des temps de fin d'exécution par rapport aux échéances, ainsi que les pénalités accordées aux occurrences ne respectant pas l'intervalle minimal imposé entre les tâches, et à celles qui sont ordonnancées alors que toutes les réceptions et émissions n'ont pas été achevées.

L'algorithme appliqué procède à l'ordonnancement de toutes les réceptions, des occurrences et enfin des émissions. Une fois l'allocation réalisée, l'algorithme propose différents modes de génération d'autres configurations (transfert de tâches en vue d'augmenter sensiblement l'énergie et de quitter les minima locaux, échange de tâches afin d'accélérer la convergence quand le système a une énergie assez basse...)

Les résultats présentés dans [ChAg94] montrent que l'algorithme peut être appliqué à des modèles de tâches de taille importante. 624 tâches avec 1580 communications ont été placées sur 6 processeurs en équilibrant le facteur d'utilisation du processeur. Cependant le recuit simulé appliqué de manière séquentielle délivre des résultats en des temps assez importants, notamment 21h pour cet exemple.

Le RS permet de qualifier une solution de bonne ou de moins bonne, mais ne fournit pas de mécanisme pour déterminer une bonne solution rapidement. On peut donc envisager de l'associer à d'autres techniques de résolution de problèmes d'optimisation (recherche tabou [CTGe93], [PoRi93], algorithmes génétiques).

(4) Utilisation d'algorithmes génétiques

Les algorithmes génétiques (AG) se sont révélés comme une voie très prometteuse pour la résolution de problèmes d'optimisation [Gold89]. Ce sont des techniques de recherche stochastiques introduites par Holland [Holl75] qui sont basées sur le principe de l'évolution biologique des espèces et qui sont exploitées par sélection et survie de l'espèce la plus adaptée au milieu. Le principe est de générer une population initiale d'individus où chacun représente une solution au problème à savoir une allocation possible. Les individus se reproduisent entre eux grâce à des opérateurs génétiques et donnent ainsi naissance à d'autres allocations.

Le principe fondamental des AG est "meilleur est un individu, plus grande est sa probabilité d'être sélectionné". La reproduction se fait dans le but de générer une autre population avec des individus meilleurs, en leur transmettant certaines de leur caractéristiques. Une fonction coût est utilisée afin d'évaluer les individus et de sélectionner les meilleurs pour constituer une autre population à laquelle on appliquera de nouveau les opérateurs. Ce processus est réitéré jusqu'à ce qu'un critère d'arrêt soit atteint.

Le codage des individus et le choix des opérateurs est fondamental pour une convergence rapide des AG. Pour le problème d'allocation de tâches aux processeurs, généralement les individus sont codés par un ou plusieurs vecteurs où une position dans le vecteur indique sur quel processeur va être placée une tâche donnée [Gold89]. Parmi les opérateurs les plus utilisés on retrouve le croisement et la mutation. Le croisement consiste à couper deux individus en un point de coupure qui peut être choisi aléatoirement, et à échanger les parties coupées. La mutation, quant à elle, échange deux positions dans un même individu. La figure 6.6 illustre une représentation binaire pour les individus.

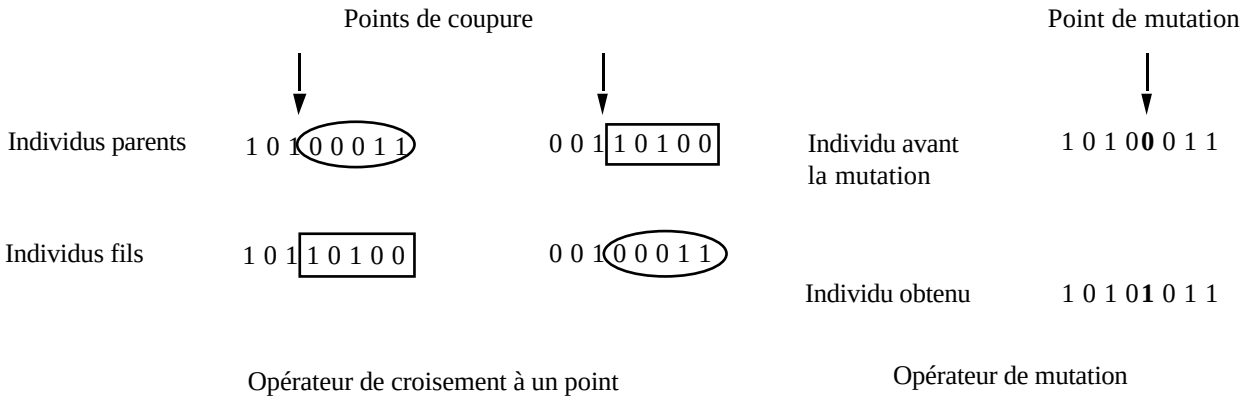


Figure n° 6.6 Application des opérateurs génétiques de croisement et mutation

L'algorithme génétique standard est le suivant :

• générer aléatoirement une population initiale d'individus
• évaluer chaque individu (lui affecter une valeur de la fonction coût)
• tant qu'une bonne solution n'a pas été trouvée
faire
étape de sélection : sélectionner un ensemble des meilleurs individus
étape de reproduction : appliquer les opérateurs aux individus sélectionnés
étape d'évaluation : calcul du coût de chaque individu
étape de remplacement : génération d'une nouvelle population par remplacement des individus sélectionnés par les meilleurs parmi ceux obtenus
fin faire

Les AG ont été appliqués avec succès à l'allocation statique de tâches. Ils ont été l'objet d'une thèse dans notre équipe [Talb93], [TaMu92] cependant les algorithmes présentés ne peuvent être appliqués à la résolution du problème de l'allocation avec des contraintes de temps. Comme nous l'avons mentionné auparavant, l'approche des AG peut être appliquée à de nombreux domaines. Ce qui distingue un algorithme d'un autre, c'est la manière d'élaborer le codage et d'adapter les opérateurs. Nous avons d'un côté un codage simple mais qui ne reflète pas les ordonnancements construits sur les processeurs, ce qui ne permet pas de guider l'algorithme à travers les opérateurs génétiques. D'un autre côté, quand les opérateurs sont appliqués de manière aléatoire, comme c'est le cas dans [Talb93], [Kidw93], [EaMa93], l'ordonnancement n'est pas pris en compte lors de la génération des allocations. On s'oriente plutôt dans ce cas, vers une phase d'allocation, suivie de l'exécution de l'algorithme d'ordonnancement afin de vérifier si les contraintes sont respectées. Les tâches ayant des contraintes individuelles, l'algorithme peut s'éxecuter assez longtemps avant de trouver une solution correcte.

En ce qui concerne l'application des AG au problème d'allocation de tâches avec contraintes de temps, peu de travaux existent [Kidw93], [EaMa93], [HRAn94], [HRAn92]. Uniquement dans les deux dernières références, les auteurs s'intéressent à la fois à l'allocation et à l'ordonnancement des tâches temps réel. Cependant, l'algorithme est appliqué pour l'ordonnancement de tâches non communicantes et soumises à des contraintes de précédence. Une hauteur est attribuée aux tâches en fonction de leur niveau de profondeur dans le graphe de tâches. Le codage employé est de un vecteurs par processeur, où chaque vecteur représente les tâches allouées au processeur selon les

relations de précédence. Afin d'éviter la génération d'individus illégaux, les points de coupure sont choisis de manière à ce que les nouveaux individus aient une hauteur croissante. Ceci permet de respecter la relation prédécesseur suivi de successeur et d'éviter qu'une tâche ne soit ordonnancée alors que l'exécution de ses prédécesseurs n'est pas terminée. Dans la figure 6.7, un exemple de ce codage est représenté, ainsi qu'un exemple de l'application de l'opérateur de croisement.

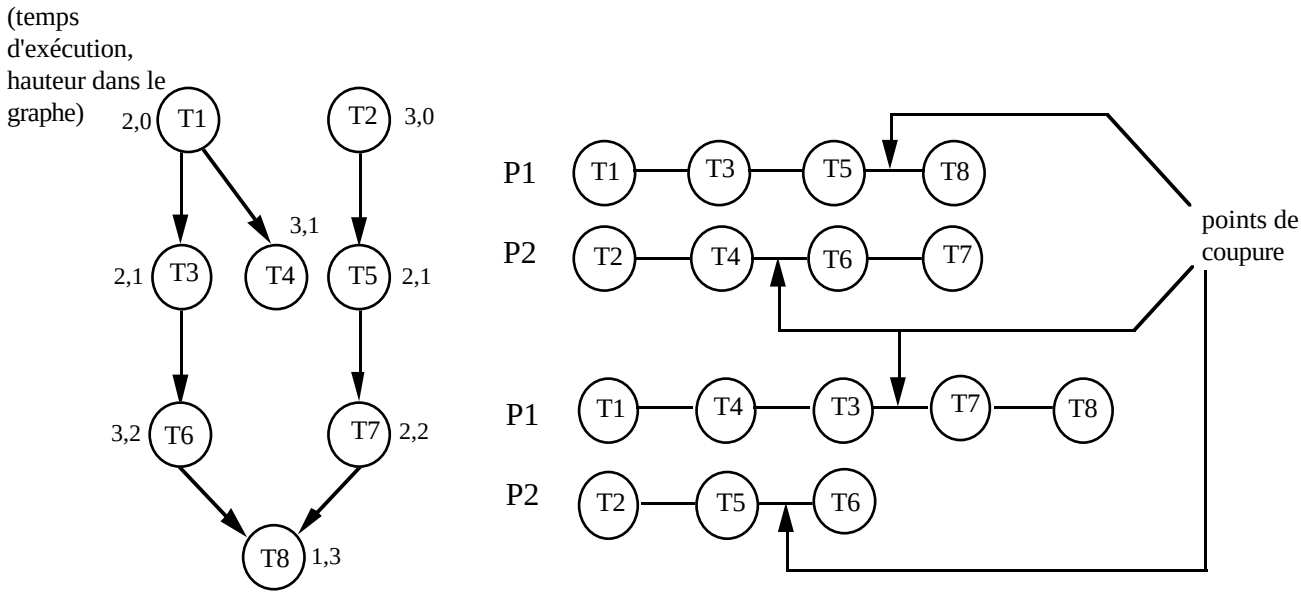


Figure n° 6.7 Exemple de codage avec application du croisement

On peut voir clairement que si le premier point de coupure sur le premier individu avait été choisi entre les tâches T_3 et T_5 à la place de T_5 et T_8, on aurait généré un individu comportant deux tâches T_3. Le choix de placer les points de coupure entre des tâches de hauteurs croissantes permet d'éviter de tels cas.

L'algorithme a été testé avec des graphes de 20 à 90 tâches dans [HRAn94]. La population initiale ne dépassait pas 20 individus. L'algorithme converge rapidement vers la meilleure solution après une moyenne de 1000 générations. De nombreuses simulations ont été effectuées sur des graphes de tâches dont la solution optimale est connue tels que le manipulateur de stanford et celui d'elbow [Desn93], l'algorithme donne des résultats avec une moyenne de 5% et un écart de 10% maximum par rapport à l'optimum.

6.3 Conclusion

On peut souligner le manque de travaux qui traitent l'allocation statique de tâches temps réel, en mettant l'accent sur les contraintes de temps individuelles des tâches et sur la nécessité d'un ordonnancement correct sur tous les processeurs. On remarque également que tout comme pour l'ordonnancement des tâches, les algorithmes sont conçus pour un modèle de tâches bien précis. Nous considérons les algorithmes itératifs, prometteurs pour ce genre de problème. En effet, à chaque itération, il est possible de vérifier si l'ordonnancement correct a été obtenu sur tous les processeurs.

Chapitre 7

Un algorithme génétique pour l'allocation statique de tâches temps réel

7.1 Introduction

Nous proposons dans ce chapitre un allocateur statique de tâches temps réel, basé sur l'application des algorithmes génétiques. Comme nous l'avons déjà mentionné, certains travaux ont déjà appliqué les AG pour des problèmes similaires mais ceux-ci ne sont pas appropriés au modèle de tâches temps réel communicantes et soumises à des contraintes de duplication que nous considérons.

Notre approche est basée sur une vue englobant l'allocation et l'ordonnancement des tâches, avec un codage du problème qui reflète la vraie nature des objectifs souhaités. Nous présentons également des opérateurs génétiques adaptés à la spécificité du problème et dont l'application diffère des opérateurs standards de croisement et de mutation [BaMu95].

7.2 Modélisation

Nous nous proposons d'allouer un ensemble T de n tâches communicantes à m processeurs. Nous rappelons l'hypothèse 6.1 où une allocation est considérée correcte dès qu'un ordonnancement correct est obtenu sur chaque processeur. Ainsi le premier critère pris en compte par l'allocation est celui du temps (à savoir le respect des contraintes temporelles). Il peut exister plusieurs allocations correctes pour un ensemble de tâches à allouer. D'autres critères sont donc pris en compte afin de les comparer et

d'affiner le résultat souhaité. Nous citons la réduction des coûts de communication ou du temps de réponse.

7.2.1 Présentation des hypothèses du modèle

Les hypothèses du modèle sont décrites ci-dessous. Nous avons adopté les même notations que celles utilisées au chapitre 2 pour représenter les paramètres temporels des tâches :

Hypothèse 1: chaque tâche T_i est décrite par le 4-uplet suivant :

T_i : < R_i, E_i, D_i, Com(T_i) > où R_i est la date à partir de laquelle T_i peut commencer son exécution, E_i le temps d'exécution, D_i l'échéance et Com(T_i) l'ensemble des tâches avec lesquelles T_i communique ainsi que les quantités d'informations à échanger. Com(T_i) = {(T_j, $c_{i,j}$) avec $c_{i,j}$ les quantités d'information à communiquer entre T_i et T_j}. Le modèle inclut des tâches périodiques et d'autres apériodiques.

Hypothèse 2 : On suppose que les tâches ne communiquent qu'à la fin de leur exécution et qu'elles ne sont pas liées par des relations de précédence. Un programmeur désirant exprimer une précédence entre les tâches, le fera à travers les dates R_i auxquelles les tâches sont prêtes. Ainsi les paramètres R_i auront des valeurs qui reflètent l'ordre d'exécution des tâches selon le graphe de précédence.

Hypothèse 3 : Nous supposons également que certaines tâches peuvent être dupliquées pour parer à des pannes matérielles. Ceci rajoute la contrainte que deux tâches dupliquées ne doivent pas être placées sur le même processeur.

Hypothèse 4: Vu notre modélisation, nous ne pouvons nous contenter du test de garantie, en effet, une pénalité sera accordée aux tâches qui dépassent leur échéances. Elle aura pour valeur le nombre d'unités de temps dépassées. Nous sommes donc dans l'obligation de dérouler un algorithme d'ordonnancement local. Nous appliquons l'algorithme d'ordonnancement que nous avons développé au chapitre 5.

Nous adopterons les notations suivantes pour la suite :

fin(P_j) date à laquelle la dernière tâche ordonnancée sur le processeur P_j finit son exécution,

Long_ord(A)	longueur de l'ordonnancement correspondant à l'allocation A réalisée. Elle correspond au maximum des *fin(P_j)*,
Pen(T_i)	pénalité de la tâche T_i. Elle vaut le nombre d'unités de temps par lesquels T_i dépasse son échéance, soit C_i - D_i ,
Pen(P_j)	pénalité du processeur P_j. Nous la définissons comme le maximum des pénalités des tâches dépassant leurs échéances sur P_j. En effet on suppose que si une tâche T_i dépasse son échéance et que par la suite, toutes les tâches ordonnancées après T_i dépassent les leurs à cause de T_i, leur pénalité est due à T_i. Il peut suffire de retirer T_i du processeur pour que ces tâches n'aient plus de pénalité. Pour cette raison, nous estimons inutile de faire la somme des pénalités sur un processeur mais de prendre le maximum des pénalités. *Pen(P_j)* = max (*Pen(T_i)*),
Pénalité(A)	nombre d'unités de temps dépassées pour les tâches qui dépassent leurs échéances avec une allocation donnée. Elle est composée de la somme des pénalités sur tous les processeurs, *Pénalité(A)* = somme(*Pen(P_j)*),
Réplication(A)	nombre de duplications d'une tâche placées sur le même processeur,
nb_T(Pj)	nombre de tâches allouées à P_j,
nb_Pen(P_j)	nombre de tâches pénalisées sur le processeur P_j,
T_Pen(P_j)	la première tâche ayant une pénalité sur P_j,
cl(T_i)	classe d'une tâche (l'ensemble des tâches sera réparti en classes selon les valeurs des échéances)
classe(cl)	ensemble de tâches de classe cl,
cardinal(cl)	nombre de tâches de la classe cl,
com_delai(P_i, P_j)	somme des délais de toutes les communications entre les 2 processeurs,
com_delai(T_i ,T_j)	délai de la communication entre T_i et T_j. Il dépend de la distance séparant les processeurs sur lesquels elles sont placées et de la quantité d'information communiquée,
coût_com(A)	somme des coûts des communications entre toute paire de processeurs,
q(T_k,T_j)	quantité d'information échangée entre T_i et T_k,

$d(P_i, P_j)$ distance entre les processeurs P_i et P_j. Elle est définie comme étant le nombre de processeurs intermédiaires pour un chemin de P_i à P_j moins 1,

7.2.2 Génération des individus

Le codage des individus et la population initiale générée en conséquent sont un facteur déterminant pour la rapidité de la convergence vers de bonnes solutions. Un bon fonctionnement des algorithmes génétiques est assuré si tous les points de l'espace de recherche représentent des individus valides pour le problème à résoudre et si la taille de la population et le nombre de générations est infini (ce qui est impossible en pratique). Dans [HRAn94], un individu est valide s'il respecte les contraintes de précédence entre les tâches. Cette notion d'individu valide doit être redéfinie dans notre cas en effet, par correspondance, un individu valide serait un individu qui respecte les contraintes temporelles des tâches et par conséquent qui réalise un ordonnancement correct sur tous les processeurs.

Il est difficile à notre avis de générer de pareils individus, en effet ceux-ci représentent la solution au problème. Nous rappelons que nous recherchons une allocation où un ordonnancement correct est obtenu sur chaque processeur. L'algorithme que nous proposons permet de trouver ces allocations et de déterminer parmi elles celle qui réduit le temps de réponse et les coûts de communication.

Hypothèse 7.1

Nous supposons qu'un individu est valide si chaque tâche y est présente et une seule fois (individu complet sans duplication).

Avant de détailler la codification utilisée, nous tenons à préciser que meilleur est le codage quand il est très dépendant du problème à résoudre. Notre but étant de construire des ordonnancements faisables sur chaque processeur, nous estimons qu'il est intéressant que ces ordonnancements apparaissent dans le codage. Pour cela nous avons choisi de représenter chaque individu, non par un vecteur mais par m vecteurs, où chaque vecteur représente un ensemble de tâches allouées à un processeur (ordonnées selon leur échéances). Soit l'exemple suivant d'un individu représentant le placement de six tâches sur deux processeurs. Au processeur P1 sont allouées les tâches T_1, T_2, T_3, T_5. L'ordre représenté est obtenu après déroulement de l'algorithme d'ordonnancement :

P_1 $T_1 \mid T_3 \mid T_5 \mid T_2 \mid$
P_2 $T_6 \mid T_4 \mid$

Génération de la population initiale

Dans [HRAn94], les tâches sont ordonnées en fonction de leur hauteur (selon les niveaux dans le graphe de précédence entre les tâches). Ceci permet de générer des individus avec des hauteurs croissantes et par conséquent d'éviter qu'une tâche ne soit ordonnancée alors que l'exécution de ses prédécesseurs n'est pas terminée. Dans notre modèle, nous désirons exprimer dans chaque individu, des tâches ordonnées en fonction de leur échéances. Pour ce faire, l'ensemble des n tâches est ordonné selon les échéances et découpé en classes.

la classe 0 contient les tâches dont l'échéance appartient à l'intervalle [0, d] (avec d la plus petite échéance de l'ensemble des tâches). La valeur *écart* est choisie en fonction de la dispersion des échéances.

La classe 1 celles dont l'échéance est entre]d, d + *écart*]
La classe 2 celles dont l'échéance est entre]d + *écart* , d + 2*écart*]
La classe 3 celles dont l'échéance est entre]d + 2*écart* , d + 3*écart*]
…

jusqu'à la dernière échéance de l'ensemble des tâches.

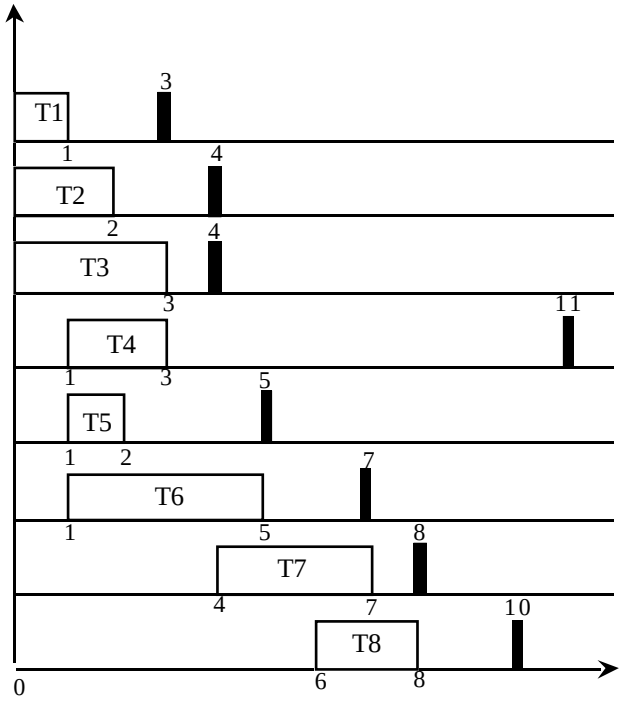
L'algorithme appliqué pour générer la population initiale choisit un nombre aléatoire de tâches parmi chaque classe et l'alloue à un processeur. Il est décrit dans ce qui suit :

- répartir les tâches en classes
- déterminer *cardinal(cl)* le nombre de tâches dans chaque classe.
- répéter pour chaque processeur P_j avec j = 1 à m-1
  - pour chaque classe cl_k k=1 à nombre de classes
    faire
    générer un nombre aléatoire *nb_tâches* entre 0 et *cardinal(cl_k)*.
    prendre *nb_tâches* de *classe(cl_k)*, les retirer de cette classe et les allouer au processeur P_j

fin faire

• allouer les tâches restantes dans chaque classe au processeur P_m

La figure 7.1 représente un ensemble de huit tâches à placer sur deux processeurs. Sur la colonne de gauche, on trouve les différentes classes établies et leur tâches respectives. Plus bas nous trouvons, trois exemples d'individus générés. La notation Individu 1 {1, 2, 2, 1} signifie que le paramètre *nb_tâches* choisi aléatoirement pour chaque classe est de 1 pour la classe 0, 2 pour la classe 1 ....et 1 pour la classe 3. Ici d et écart valent 3.



classe(0) { T1}

classe(1) { T2, T3, T5}

classe(2) {T6, T7}

classe(3) {T8, T4}

Individu 1 { 1, 2, 2, 1}
P1 T1 | T2 | T3 | T6 | T7 | T8 |
P2 T5 | T4 |

Individu 2 { 1, 1, 1, 1}
P1 T1 | T2 | T6 | T8 |
P2 T3 | T5 | T7 | T4 |

Individu 3 { 0, 1, 2, 1}
P1 T3 | T6 | T7 | T8 |
P2 T1 | T2 | T5 | T4 |

Figure n° 7.1 Exemple de génération d'individus

7.2.3 Fonction coût

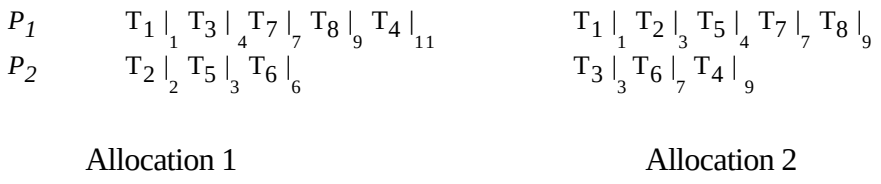
Afin de réaliser les objectifs du problème, nous considérons la fonction coût suivante:

$$F(A) = k_1 Long\_ord(A) + k_2 Coût\_com(A) + k_3 Pénalité(A) + k_4 Réplication(A)$$

Nous cherchons à minimiser la fonction F. Cela consiste à obtenir des pénalités et des réplications nulles et à réduire la longueur de l'ordonnancement ainsi que les coût des communications.

calcul de la longueur de l'ordonnancement : *Long_ord(A)* est obtenue après que les tâches allouées aient été ordonnées sur tous les processeurs. Nous estimons nécessaire de minimiser cette valeur, en effet il peut exister des allocations qui respectent les contraintes de temps mais qui n'équilibrent pas l'utilisation du processeur. Certes notre modélisation ne prend pas en compte l'utilisation du processeur, mais ceci fait partie des objectifs à atteindre à travers cette minimisation de la longueur de l'ordonnancement. S'il existe des allocations avec des longueurs plus courtes que celle trouvée, c'est qu'un autre agencement des tâches existe et non que certaines tâches ont été démarrées plus tôt dans une autre allocation (puisqu'un algorithme d'ordonnancement local, les ordonnance toujours à la date au plus tôt à laquelle elles peuvent démarrer). Par conséquent, s'il existe une longueur d'ordonnancement plus courte, c'est qu'une utilisation plus équilibrée des processeurs a été trouvée.

Soit l'exemple de la figure 7.2 qui représente l'allocation des 8 tâches de la figure 7.1 à deux processeurs, la figure représente 2 allocations possibles. Sur chaque processeur P_1 et P_2, les tâches sont ordonnancées :



Allocation 1 Allocation 2

Figure n° 7.2 Comparaison d'ordonnancements

Les chiffres en bas à droite des tâches représentent leur date de fin d'exécution. C'est l'allocation 2 qui sera gardée car elle offre la longueur d'ordonnancement la plus courte.

calcul des coûts de communication : on commence par estimer la quantité d'information échangée entre les processeurs, une fois l'allocation faite.

Le délai de communication suite à l'attente du résultat d'une tâche donnée est défini comme suit :

• $com\_delai(T_i,T_j) = q(T_i,T_j)*d(P_i, P_j)$ avec T_j émettant vers T_i une information de quantité $q(T_i,T_j)$, T_j est placée sur P_j et T_i sur P_i.

• $com\_délai(P_iP_j)$ = somme des $com\_delai(T_i, T_j)$ pour toute tâche T_i et T_j communiquant entre P_i et P_j.

Les coûts de communication vont permettre de calculer les dates de début d'exécution des tâches selon l'algorithme d'ordonnancement local déroulé. L'algorithme se base sur les paramètres R_i, E_i et D_i pour calculer C_i. Par conséquent, une première phase de recalcul des R_i est nécessaire afin d'y inclure les temps de communication.

Pour calculer R_k pour une tâche T_k donnée, on calcule $com\_delai(T_k,T_j)$ pour chaque tâche T_j communiquant un résultat à T_k.

R_k = max { R_k ; max{$R_j + E_j + com\_delai(T_k, T_j)$} pour toute tâche T_j communicant avec T_k}

<u>*calcul de la pénalité*</u> : afin de qualifier une solution de bonne ou mauvaise, nous attribuons une pénalité pour chaque tâche dont l'échéance est dépassée dans une allocation donnée. Nous rappelons l'équation :

Pénalité(A) = $\sum_{j=1}^{m} \max\{(Ci - Di)\}$ où chaque processeur calcule la différence (C_i-D_i) pour toutes les tâches qui lui ont été allouées. Les dates C_i ne sont pas fournies par le programmeur mais calculées lors du déroulement de l'algorithme d'ordonnancement local.

<u>*calcul de la réplication*</u> : on dit qu'il y a réplication si deux tâches dupliquées sont placées sur le même processeur. Dans ce cas pour toute tâche répliquée, on attribue une pénalité de 1. Ainsi *réplication(A)* vaut le nombre de tâches dupliquées placées sur le même processeur. Tout comme pour la pénalité, la réplication doit être nulle afin qu'une allocation soit qualifiée de correcte.

Les poids k_1, k_2, k_3, k_4 sont associés à chaque composant de la fonction coût, afin d'exprimer l'importance de certains par rapport aux autres. Si les valeurs de ces poids sont proches, les composants de la fonction coût seront équilibrés. Nous sommes intéressés par des pénalités et des réplications nulles, pour cela les poids k_3 et k_4 seront les plus élevés.

7.3 Définition de nouveaux opérateurs génétiques

La principale fonction des opérateurs génétiques est la création de nouveaux individus en combinant ou en réarrangeant des individus parents de la population actuelle. Cette étape permet de générer une nouvelle population. Il est important de bien choisir les opérateurs les plus adaptés au problème à résoudre afin que les nouveaux individus générés soient légaux et qu'on ne perde pas de temps à générer et évaluer des individus qui sont moins bons que leurs parents.

Nous proposons d'utiliser pour la reproduction des individus, les opérateurs de croisement et mutation standards que nous avons modifié afin qu'ils soient plus appropriés au codage utilisé et à l'objectif visé.

La mutation

Notre codage étant sur *m* vecteurs, il s'agira d'appliquer la mutation entre deux positions appartenant à des vecteurs différents. En effet, dans un même vecteur, ceci n'est pas nécessaire car l'algorithme d'ordonnancement détermine le meilleur ordre pour l'ensemble des tâches du processeur en question. Le but de la mutation est de réduire les pénalités sur les vecteurs d'un individu donné. Pour cela, nous avons adopté les hypothèses suivantes :

• la mutation n'est appliquée qu'aux individus ayant une pénalité non nulle.

• elle est régie par des règles afin de réduire le nombre de pénalités. Pour ce faire chaque individu sauvegarde pour chaque processeur P_j, en plus de $Pen(P_j)$ la tâche qui correspond à cette pénalité $T\_Pen(P_j)$ et le nombre de tâches pénalisées $nb\_Pen(P_j)$.

Le principe est de réduire les pénalités en allégeant les processeurs très pénalisés. Pour cela on définit *min_pen* comme la valeur à partir de laquelle les tâches sont transférées. Dans le cas où un processeur est très pénalisé, il y a transfert de la première tâche ayant une pénalité. Le cas échéant on échange des tâches entre les processeurs très pénalisés et ceux qui le sont moins. Nous avons choisi comme tâche à transférer la première tâche ayant une pénalité, en effet, le retard de celle-ci peut entraîner un retard des tâches ordonnancées après et par conséquent être à l'origine de leur pénalité.

L'algorithme suivant décrit l'application de la mutation :

• déterminer le processeur de plus haute pénalité P_{hp}

• déterminer le processeur de plus faible pénalité P_{fp}

• si $nb\_Pen(P_{hp}) > min\_pen$

prendre $T\_Pen(P_{hp})$ et la transférer de P_{hp} vers P_{fp}

sinon

prendre $T\_Pen(P_{hp})$ et l'échanger avec une tâche T_e de P_{fp} tel que $cl(T_e) = cl(T\_Pen(P_{hp}))$ (ou supérieure le cas échéant)

si réplication(A) différente de 0

choisir deux processeurs au hasard

choisir une tâche au hasard sur le premier processeur et la transférer vers le second

Cet algorithme permet d'alléger P_{hp} s'il contient beaucoup de tâches pénalisées, le cas échéant d'échanger sa tâche la plus pénalisante avec une autre telle que son échéance soit plus grande afin de ne pas tomber dans le cas où on procéderait au remplacement de $T\_Pen(P_{hp})$ par une tâche qui risque d'être aussi pénalisante.

Les deux critères de réduction de la pénalité et de la réplication étant antagonistes, le test sur la réplication nulle permet d'éviter à l'algorithme de tomber dans un minimum local quand la pénalité est non nulle. Ainsi, de nouveaux individus peuvent être générés.

Soit l'exemple de la figure 7.3 où on compare l'application de l'échange et du transfert de tâches lors de la mutation :

P1 $T_1 \|_1 T_2 \|_3 T_3 \|_6 T_4 \|_8$	$T_1 \|_1 T_2 \|_3 T_3 \|_6 T_7 \|_9$	$T_1 \|_1 T_2 \|_3 T_3 \|_6 T_7 \|_9 T_4 \|_{11}$
P2 $T_5 \|_2 T_6 \|_6 T_7 \|_9 T_8 \|_{11}$	$T_5 \|_2 T_6 \|_6 T_8 \|_8 T_4 \|_{10}$	$T_5 \|_2 T_6 \|_6 T_8 \|_8$
$Pen(P_1) = Pen(T_3) = 2$	$Pen(P_1) = 2 = \max(2, 1)$	$Pen(P_1) = 2 = \max(2, 1)$
$Pen(P_2) = 1 = \max(1,1)$	$= \max(Pen(T_3), Pen(T_7))$	$=\max(Pen(T_3), Pen(T_7))$
$= \max(Pen(T_7),Pen(T_8))$	$min\_pen = 2$	$min\_pen = 1$
$nb\_Pen(P_1) = 1$	$nb\_Pen(P_1) = 2$	$nb\_Pen(P_1) = 2$
$nb\_Pen(P_2) = 2$		

Figure n° 7.3 Comparaison du transfert et de l'échange dans la mutation

La colonne de gauche représente l'individu sur lequel on va appliquer la mutation. La colonne du centre l'individu obtenu après échange de la première tâche pénalisante sur P_2 à savoir T_7 avec une de classe supérieure T_3 (puisqu'il n'y a pas de tâches de même classe sur le processeur P_1). La colonne de droite montre l'individu obtenu après transfert de cette même tâche.

L'opérateur de croisement

La préoccupation majeure lors de l'application de cet opérateur est la génération d'individus complets sans duplication (individus valides). Pour plus de clarté, la figure 7.4 illustre un exemple où le croisement est appliqué en choisissant des points de coupure aléatoirement :

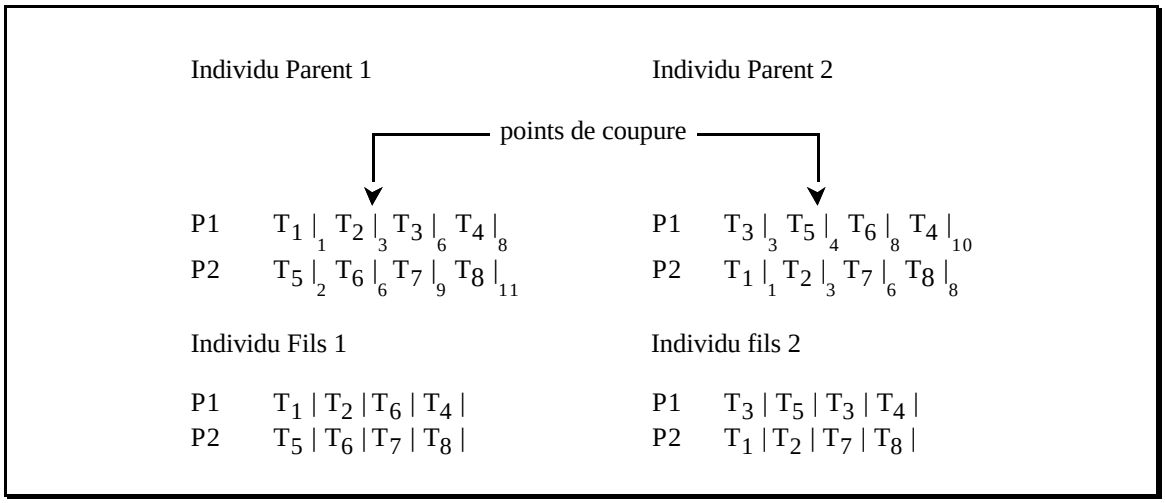


Figure n° 7.4 Exemple de points de coupure aléatoires

Afin d'éviter de pareilles situations, nous choisissons des points de coupure qui vérifient la relation suivante :

$$cl(T_i) < cl(T_j)\ ,\ cl(Ti') < cl(T_j')\ ,\ cl(T_i) = cl(T_i') \qquad (1)$$

P_1 $T_k \mid T_i \mid T_j \mid T_k \mid$ P_1 $T_k \mid T_{i'} \mid T_{j'} \mid T_k \mid$

Individu Parent 1 Individu Parent 2

L'équation (1) garantie qu'à droite de chaque point de coupure, les tâches auront des classes supérieures et par conséquent le risque de duplication de tâches au sein d'un individu est exclus. Un point de coupure est choisi sur chaque vecteur composant les individus à croiser. Le croisement est appliqué entre deux vecteurs représentant le même

processeur. Les points de coupure ne doivent pas nécessairement être à la même position sur les deux vecteurs. L'exemple qui suit illustre une application correcte du croisement :

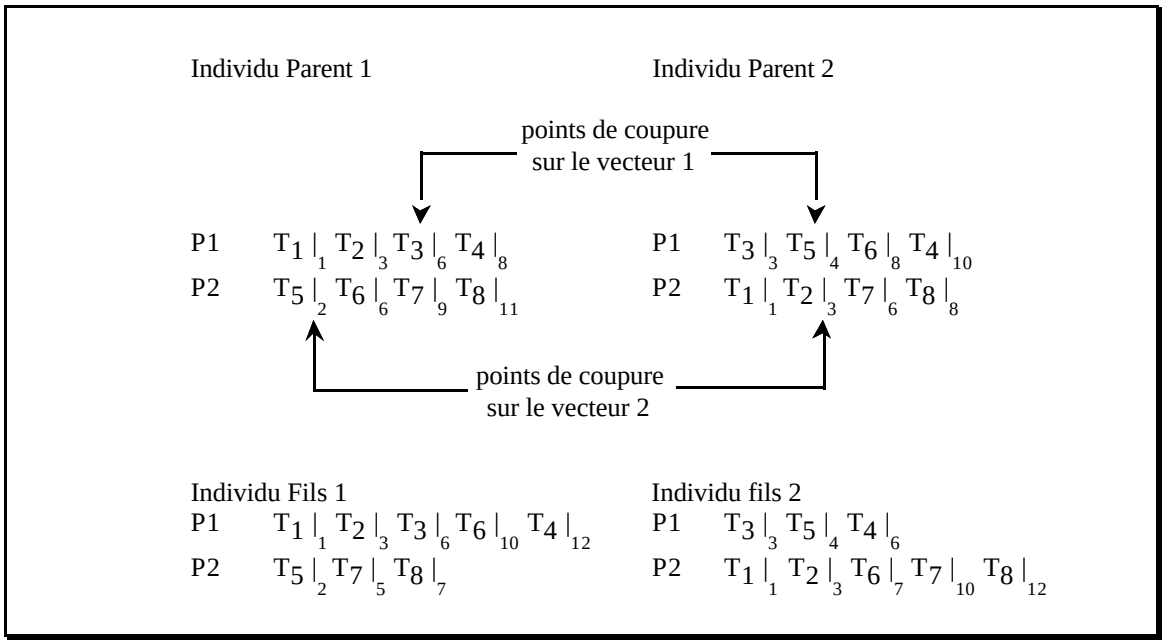


Figure n° 7.5 Exemple de croisement correct

Ces individus sont plus mauvais que leurs parents si on compare les paramètres *long_ord* et *pénalité*

	long_ord	*pénalité*
Individu parent 1	11	3
Individu parent 2	10	1
Individu fils 1	12	3
Individu fils 2	12	2

Toutefois, en appliquant la mutation aux individus fils, on obtient :

P1 $T_1 |_1 T_2 |_3 T_6 |_7 T_4 |_9$ P1 $T_3 |_3 T_5 |_4 T_7 |_7$

P2 $T_3 |_3 T_5 |_4 T_7 |_7 T_8 |_9$ P2 $T_1 |_1 T_2 |_3 T_6 |_7 T_8 |_9 T_4 |_{11}$

En revanche l'individu fils 1 a une pénalité nulle et une longueur d'ordonnancement minimale.

Pour cela nous proposons d'appliquer d'abord le croisement, de réajuster les individus (en les ordonnant), de les réévaluer et ensuite si nécessaire d'appliquer la mutation.

7.4 Modèles parallèles pour l'exécution de l'algorithme génétique

Plusieurs applications des AG aux domaines de la robotique, de l'intelligence artificielles ont été étudiées. Cependant leur coût d'exécution important reste un obstacle majeur pour leur développement. L'apparition des architectures parallèles a permis d'obtenir des résultats très satisfaisants en parallélisant ces algorithmes.

On distingue plusieurs modèles pour une exécution en parallèle de l'algorithme génétique :

(1) Le modèle *ferme de processeurs*, où un processeur maître effectue la sélection des paires d'individus et les envoie aux processeurs esclaves qui se chargent de la reproduction et de l'évaluation. Ce modèle est assez coûteux en communications, et n'est pas entièrement parallèle en effet l'étape de sélection est souvent séquentielle. Il est peu adapté à des machines parallèles, où les communications doivent être réduites si l'on veut garantir de bonnes performances.

(2) Le modèle à *décomposition* où la population est divisée en sous-populations, chacune placée sur un processeur. A notre avis, ce modèle est plus adapté que le précédent aux machines parallèles. Il est intéressant quand le nombre de processeurs est inférieur à la taille de la population. Toutefois, le parallélisme inhérent à une sous-population n'est pas exploité.

(3) Le modèle *massivement parallèle* où un individu est placé sur un processeur. Cette approche nous paraît la meilleure pour des machines parallèles où le nombre de processeurs ne cesse de croître. Cette approche est adoptée dans [Talb93]. Ainsi toutes les étapes se font en parallèle. La sélection est faîte après échange des individus entre processeurs.

Afin d'éviter que l'échange ne soit coûteux en communications, il est restreint au voisinage immédiat du processeur, à savoir les processeurs avec lesquels il est directement connecté.

La description de l'algorithme génétique exécuté par chaque processeur est la suivante :

• Générer individu initial local
• ordonner et évaluer pénalité individu local
• Tant que (nombre d'itérations ≤ maximum-itérations)
faire
phase de communication (chaque processeur envoie son individu et attend ceux des autres)
sélection du meilleur individu parmi ceux reçus
phase de reproduction
appliquer le croisement à (individu local et individu choisi)
ordonner et évaluer pénalité individu local
appliquer la mutation aux individus fils générés
ordonner et évaluer la fonction coût si la pénalité est nulle
phase de remplacement
fin faire

Figure n° 7.6 L'algorithme génétique parallèle

Il est à signaler que la fonction coût n'est pas souvent évaluée, en effet nous sommes particulièrement préoccupés par des allocations à pénalités nulles. La longueur de l'ordonnancement et le coût des communications ne deviennent des éléments de mesure qu'une fois qu'une pénalité nulle a été trouvée.

7.5 Evaluation des performances de l'algorithme

L'algorithme génétique a été implémenté sur la machine SuperNode. C'est une machine à mémoire distribuée composée de transputers reliés par un réseau d'interconnexion. L'algorithme a été implémenté sur une topologie en grille torique composée de 120 processeurs. Des mesures ont été prises pour des ensembles composés de huit et 30 tâches.

L'évaluation de cet algorithme est une tâche délicate, en effet la fonction coût est formée de plusieurs critères antagonistes. Prenons les paramètres pénalités accordés pour les tâches dépassant leurs échéances et celles répliquées. L'algorithme peut rapidement trouver une pénalité nulle et la perdre en essayant de réduire la réplication. Pour mettre en évidence l'influence des réplications sur les performances de l'algorithme, nous

présentons des mesures du nombre d'itérations moyen pour trouver le résultat quand l'ensemble de tâches n'est pas soumis à des duplications et par la suite quand certaines tâches doivent être dupliquées.

Les cas de figure pour lesquelles l'algorithme a été appliqué sont résumés dans les quatre benchmarks qui suivent :

• benchmark1 : un graphe de huit tâches sans duplication à placer sur deux processeurs,

• benchmark2 : un graphe de huit tâches dont trois ont été dupliquées à placer sur deux processeurs,

• benchmark3 : un graphe de 30 tâches sans duplication à placer sur deux processeurs,

• benchmark4 : un graphe de 30 tâches dont cinq ont été dupliquées à placer sur deux processeurs.

Pour chaque benchmark, 20 exécutions de l'algorithmes ont été réalisées.

7.5.1 Influence de la duplication des tâches

L'ensemble des tâches des benchmarks 3 et 4 ont des dates R_i assez proches, en effet la moyenne de l'écart entre les dates au plus tôt est de 3 ms. Les temps d'exécution varient entre 1ms et 40 ms. Ceci pour préciser que l'algorithme d'ordonnancement appliqué se retrouve fréquemment à réordonnancer les tâches. Les durées d'exécution de l'algorithme sont données en secondes.

La figure 7.7 résume les temps de calcul pour les quatre benchmarks suivants :

un ensemble de 8 tâches sans duplication			un ensemble de 8 tâches dont 3 sont dupliquées		
	itérations	temps		itérations	temps
min	1	0,5	min	1	0,6
max	2	1	max	20	1
moyenne	1	0,75	moyenne	9	0.85

un ensemble de 30 tâches sans duplication			un ensemble de 30 tâches dont 5 sont dupliquées		
	itérations	temps		itérations	temps
min	2	0,8	min	4	1
max	33	3	max	100	15
moyenne	13	1.25	moyenne	40	9

Figure n° 7.7 Temps de calcul en fonction de la duplication des tâches

Pour les deux ensembles de tâches on observe que l'algorithme converge rapidement vers une allocation correcte quand aucune tâche n'est dupliquée. Par contre, le nombre d'itérations est plus important quand certaines tâches sont dupliquées. Ceci s'explique par le fait que l'algorithme n'a aucun moyen de préserver le résultat obtenu quand il reste des tâches répliquées. Que ce résultat soit une pénalité nulle ou une réplication améliorée, l'opérateur de mutation va échanger deux tâches choisies aléatoirement (tout en veillant à créer des individus légaux). Même dans le cas où l'algorithme est modifié pour déplacer la tâche qui cause la réplication, rien ne garantit que l'individu obtenu aura une pénalité nulle.

On remarque également que le coût d'exécution ainsi que le nombre d'itérations croissent proportionnellement avec la taille du problème. En effet la qualité du résultat dépend de la taille de la population. Une taille de 120 individus est encore raisonnable pour l'ensemble des 30 tâches à placer, ce qui explique la rapidité de convergence. Toutefois, il faut s'attendre à ce que les performances de l'algorithme se dégradent pour des ensembles de tailles beaucoup plus importantes.

7.5.2 Influence des paramètres de l'algorithme

En ce qui concerne les probabilités d'application des opérateurs de croisement et de mutation, seule la mutation est appliquée avec une certaine probabilité et uniquement dans le cas où la pénalité et la réplication sont non nulles. L'opérateur de croisement que nous avons proposé pour notre algorithme ne nécessite pas de probabilité pour son application, en effet la recherche de points de croisement peut échouer (il n'existe pas de points de coupure permettant de générer des individus légaux).

En revanche, nous avons remarqué que l'opérateur de mutation a une influence sur la rapidité de convergence. Elle doit être fréquemment appliquée. Ceci s'explique par la manière dont il a été conçu. Dans la section 7.2.1, nous avons expliqué que la pénalité d'un processeur peut être considérablement réduite en retirant la première tâche qui dépasse son échéance. On observe d'ailleurs que dès la deuxième itération de l'algorithme, la pénalité est réduite considérablement.

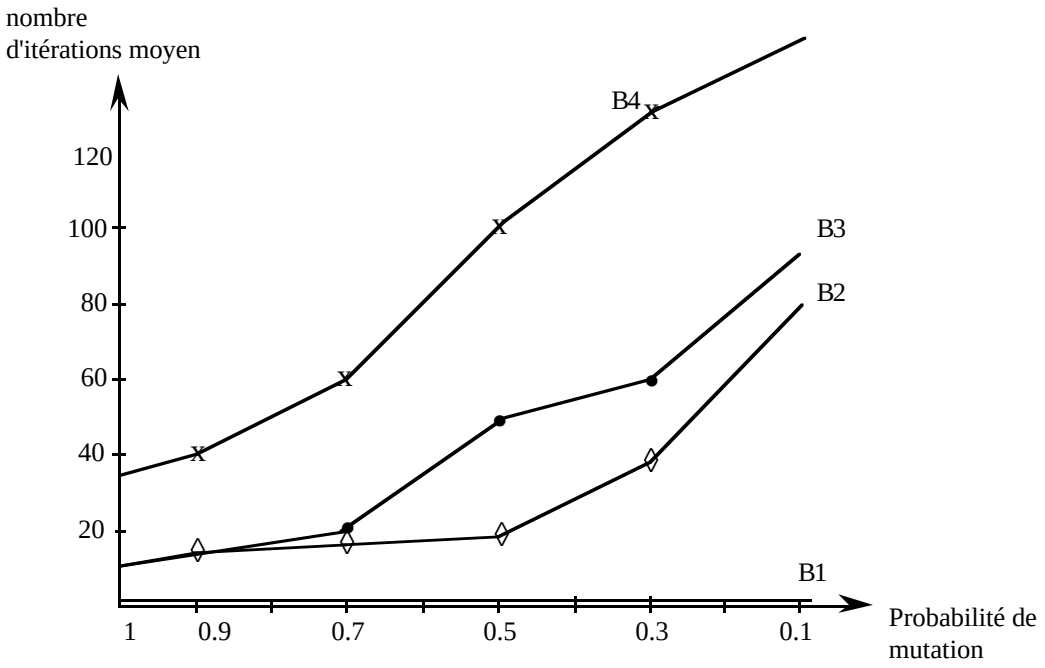


Figure n° 7.8 Influence de la probabilité de mutation sur la qualité de la solution

7.6 Comparaison avec le recuit simulé

Dans cette section, les performances de notre algorithme génétique sont comparées à celles obtenues en appliquant le recuit simulé à la résolution du problème de l'allocation statique de tâches temps réel. Nous rappelons qu'une description détaillée de cette approche a été présentée au chapitre 6.

Dans notre implémentation, les configurations sont exprimées de la même manière que dans l'algorithme génétique, à savoir qu'une configuration donnée est décrite par la donnée d'une matrice où chaque ligne indique l'ordre d'exécution des tâches sur le processeur, ayant pour numéro l'indice de la dite ligne. La génération de configurations voisines s'est faite par la sélection d'une tâche sur un processeur et son déplacement vers un autre processeur. Le choix de la tâche et du processeur s'est fait de manière aléatoire.

Le recuit simulé a été implémenté de manière séquentielle. Afin que les comparaisons soient raisonnables, nous avons également développé une version séquentielle de l'algorithme génétique.

L'algorithme a été appliqué avec une température initiale de 10, et une température minimale de 0.1. Le taux de décroissance de la température est de 0.9. Pour les mêmes

benchmarks que ceux présentés à la section 7.5 et dans les mêmes conditions d'exécution, on obtient les mesures suivantes :

	Un ensemble de huit tâches sans duplication		Un ensemble de huit tâches dont trois sont dupliquées	
Algorithme appliqué	Moyenne des itérations	Moyenne du temps	Moyenne des itérations	Moyenne du temps
Recuit simulé	72	0.8	366	2.5
Algorithme génétique séquentiel	5	0.7	30	2.9

Figure 7.9 Comparaison entre le recuit simulé et l'AG séquentiel pour un ensemble de huit tâches

	Un ensemble de 30 tâches sans duplication		Un ensemble de 30 tâches dont cinq sont dupliquées	
Algorithme appliqué	Moyenne des itérations	Moyenne du temps	Moyenne des itérations	Moyenne du temps
Recuit simulé	945	9	5300	53
Algorithme génétique séquentiel	420	11	1200	27

Figure 7.10 Comparaison entre le recuit simulé et l'AG séquentiel pour un ensemble de 30 tâches

Au premier abord, on remarque le faible nombre d'itérations faites par l'AG même s'il est séquentiel, ceci s'explique par la réduction du nombre de réplication et de pénalité qui est faite dès les premières itérations (comme mentionné auparavant). Pour les quatre benchmarks, l'AG se comporte mieux. Ceci est dû aux opérateurs que nous avons proposé, car ils permettent d'identifier la tâche pénalisante et de la déplacer. Alors que dans le recuit simulé, une configuration voisine est générée de manière aléatoire.

Nous avons également remarqué que la température initiale n'influe pas sur le temps nécessaire pour trouver une première solution correcte. Une température élevée implique un temps d'exécution plus important, ce qui permet à l'algorithme de trouver parmi les allocations correctes déterminées, celle qui minimise la valeur de la fonction coût.

D'autres mesures sont en cours afin d'étudier le comportement de l'algorithme du recuit simulé en fonction de la vitesse de décroissance de la température. Nous

envisageons également de guider le déplacement des tâches entre les processeurs comme c'est le cas dans l'AG.

7.7 Conclusion

Le côté itératif des algorithmes génétiques combiné avec la puissance des opérateurs génétiques constitue un attrait particulier. Les algorithmes génétiques peuvent être appliqués à de nombreux modèles de tâches avec des contraintes diverses. Le codage et les opérateurs étant spécifiques au problème à résoudre, l'élaboration de ces derniers est un facteur déterminant pour une bonne vitesse de convergence.

Dans ce chapitre, nous avons proposé un codage et de nouveaux opérateurs pour un algorithme génétique qui réalise à la fois l'allocation et l'ordonnancement des tâches. En effet, quand l'algorithme se termine, on obtient sur chaque processeur un ensemble de tâches ordonnançables correctement. Le modèle de tâches visé est assez large dans la mesure où il considère la communication et la duplication des tâches.

La mise en œuvre de l'algorithme a permis d'observer, une convergence rapide qui passe par une réduction sensible de la pénalité et de la réplication dès les premières itérations.

L'algorithme possède une vitesse de convergence presque idéale. Elle est définie par le rapport entre le temps d'exécution de l'algorithme sur un seul processeur et le temps de son exécution sur plusieurs processeurs. Ceci s'explique par le fait que le coût de communication de l'algorithme est indépendant de la taille de l'architecture (un individu communique uniquement avec ses voisins).

La comparaison de l'algorithme génétique proposé avec celui du recuit simulé, a montré un net avantage de notre algorithme. Les temps de recherche sont nettement plus courts. Nous envisageons à l'avenir de comparer notre algorithme aux algorithmes de recherche tabou. Il serait également intéressant de combiner certaines de ces approches afin d'exploiter les avantages de chacune. Ainsi, la recherche tabou pourrait guider l'algorithme génétique quand la convergence devient lente, à savoir quand l'algorithme n'arrive pas à annuler à la fois la pénalité et la réplication.

PARTIE IV

Allocation dynamique de tâches temps réel

Résumé

Dans un système temps réel, où peuvent être créées des tâches durant l'exécution, un mécanisme d'allocation dynamique capable de fournir un haut degré de garantie à ces tâches est d'une importance capitale pour les performances du système. Afin qu'il soit efficace, le mécanisme doit à la fois garantir les contraintes d'un maximum de tâches, et dépenser un minimum de temps pour la recherche d'un processeur pour chaque tâche à allouer.

Le coût excessif dû au maintien d'un état global du système et le besoin de réduire les temps de recherche font que la plupart des algorithmes appliqués dans ces mécanismes sont des heuristiques.

La plupart des algorithmes appliqués dans la littérature sont spécifiques au problème d'allocation sans contraintes de temps. Ils ont été adaptés afin d'être appliqués au problème qui nous intéresse. Ils sont présentés dans le chapitre 8.

Dans le chapitre 9, nous exposons la structure du mécanisme d'allocation dynamique proposé [BMTa94]. Il est composé de deux modules : le module information et le module décision. Dans un premier temps, nous détaillons le module information chargé du maintien de l'état du système. Nous y proposons un algorithme pour l'échange des informations entre les processeurs indépendants de la taille et de la topologie du réseau. Le module décision applique une heuristique pour l'allocation des tâches apériodiques urgentes qui vise à leur donner davantage de garantie qu'aux tâches périodiques. Elle procède par un retrait d'une tâche moins urgente auparavant garantie pour laquelle l'algorithme d'allocation est appliqué, et par l'acceptation de la tâche apériodique. Afin que la garantie obtenue par le mécanisme d'ordonnancement local puisse être assurée, des dispositions de réservation de l'emplacement de la tâche au niveau du processeur choisi sont mis en place.

Le mécanisme est intégré au système d'exploitation Paros. Il est supporté par le noyau ParX qui implante le modèle de processus et offre le support pour la communication. Une implémentation du mécanisme a été mise en oeuvre sur une machine à base de transputers. Des résultats préliminaires sont présentés.

Chapitre 8

Algorithmes d'allocation dynamique de tâches à contraintes de temps

8.1 Introduction

Contrairement aux tâches périodiques, les tâches apériodiques ne peuvent être allouées statiquement sans entraîner une mauvaise exploitation des ressources et un déséquilibre entre les utilisations des processeurs. Encore faut-il qu'elles soient sporadiques afin qu'un intervalle minimum entre les tâches soit connu. Il est donc nécessaire de prévoir une allocation au fur et à mesure que ces tâches se présentent.

Ainsi, plusieurs algorithmes d'allocation dynamique ont vu le jour. Tout comme pour l'ordonnancement, à chaque processeur est associé un mécanisme d'allocation appelé *allocateur* qui se charge désormais d'allouer les tâches refusées à d'autres processeurs.

Nous tenons à préciser que ces algorithmes sont valables aussi bien pour les systèmes distribués que parallèles, ces solutions en fait sont conçues pour des systèmes sans mémoire commune.

Nous distinguons deux approches pour l'allocation dynamique des tâches temps réel. La première regroupe des politiques non coopératives où l'allocateur agit seul pour le choix du processeur pour le transfert. La seconde regroupe celles coopératives, où certains allocateurs coopèrent afin de déterminer le processeur qui est le plus apte à recevoir la tâche.

8.2 L'approche non coopérative

Dans le cas des politiques non coopératives, le rôle de l'allocateur consiste à chaque fois qu'il prend la main à dérouler l'algorithme d'allocation choisi. Aucun contrôle global n'est nécessaire puisque le processeur est choisi en faisant abstraction de toute information sur les autres processeurs.

8.2.1 L'algorithme aléatoire

Soit T_r la tâche refusée localement sur le processeur P_{ref}, AP_{ref} (l'allocateur du processeur P_{ref}) envoie T_r vers un processeur choisi aléatoirement parmi les autres. Dans [RSZh89] cette technique a été appliquée et les résultats obtenus sont plutôt satisfaisants. Ceci s'explique par le fait qu'en cas de petite charge, les processeurs disposent d'un temps libre important et peuvent eventuellement accepter la tâche refusée sur P_{ref}. Cet algorithme a également été comparé à des techniques coopératives en cas de forte charge. Les résultats sont considérés satisfaisants comparés aux techniques coopératives car ces dernières sont gourmandes en communication, et doivent être appliquées avec beaucoup de précautions en cas de forte charge. Ceci n'empêche que cette technique est très risquée pour les systèmes temps réel. Etant donné que cet algorithme n'exploite aucune information sur l'état du système, il existe le risque que le processeur choisi ne puisse pas exécuter T_r. Par conséquent T_r peut visiter plusieurs processeurs avant d'en trouver un, au risque de dépasser son échéance.

8.2.2 L'algorithme cyclique

Il existe d'autres techniques non coopératives, comme celles où le processeur choisi appartient à un groupe de processeurs. Le groupe est servi de manière cyclique à chaque allocation de tâche refusée localement [WaMo85]. Nous pensons qu'il serait tout de même judicieux vu le caractère urgent des tâches de constituer ces groupes de processeurs en se basant sur des critères de temps appropriés. Un allocateur pourrait ainsi former un groupe avec des allocateurs qui ont déjà garanti des tâches pour lui, et choisir le processeur de manière cyclique.

8.3 L'approche coopérative

En ce qui concerne les techniques coopératives, compte tenu de l'absence de mémoire commune aux processeurs du réseau, le contrôle sera pris en charge par la coopération des allocateurs situés sur chaque processeur du réseau.

8.3.1 Propriétés des modules réalisant l'allocation dynamique

Nous recensons les caractéristiques suivantes :

(1) Les allocateurs coopèrent de manière à réaliser l'allocation des tâches refusées dans le système.

(2) Chaque allocateur prend ses propres décisions. Il n'y a ni contrôle centralisé ni information globale sur l'état des autres processeurs.

8.3.2 Principales politiques

Les algorithmes présentés dans cette section, sont des algorithmes d'allocation classique qui ont été adaptés afin que le critère du temps soit primordial. A l'origine, on distingue les deux raisons suivantes :

• dans les systèmes sans contraintes de temps, l'état de ce dernier est mesuré par un facteur de charge qui vaut généralement le nombre de tâches sur un processeur. La première raison est que ce facteur de mesure peut dans certaines situations ne pas être valable car un processeur disposant que de deux tâches n'est pas forcément peu chargé. En effet, le temps d'exécution peut être assez long.

• la seconde raison est qu'une bonne adaptation des paramètres utilisés par ces algorithmes peut conduire à des résultats plus intéressants. C'est la manière d'adapter ces algorithmes qui est un facteur déterminant de la qualité des résultats, que nous présentons dans ce qui suit.

Nous distinguons deux approches pour le transfert des tâches : la première applique des politiques à l'initiative de l'émetteur, c'est à dire que ce dernier dispose d'informations sur les autres processeurs qui lui permettent de choisir le plus apte à garantir la tâche, la seconde applique des politiques à l'initiative du destinataire, où un

processeur interroge régulièrement les autres processeurs à la quête de tâches refusées qu'ils pourraient lui transférer.

8.3.2.1. Les techniques à l'initiative de l'émetteur

Dans ce cas l'allocateur est chargé de gérer une information sur les processeurs du réseau afin de pouvoir sélectionner le meilleur processeur pouvant garantir les contraintes d'une tâche refusée.

Il est principalement composé des modules suivantes :

- un module pour la gestion de l'échange de l'information entre les processeurs
- un module pour le choix de l'algorithme de transfert à appliquer.

La figure 8.1 illustre la coopération entre les allocateurs et leur interaction avec l'ordonnanceur. Quand une tâche est reçue d'un autre processeur, deux cas de figure possibles se présentent : soit le processeur a été choisi pour exécuter cette tâche auquel cas c'est l'ordonnanceur qui se charge de vérifier si elle peut être garantie localement. Il en est de même pour les tâches créées sur le processeur. Le deuxième cas de figure correspond à une tâche pour laquelle un allocateur est en train de rechercher des processeurs capables de la garantir, dans ce cas, la réponse est retournée vers cet allocateur.

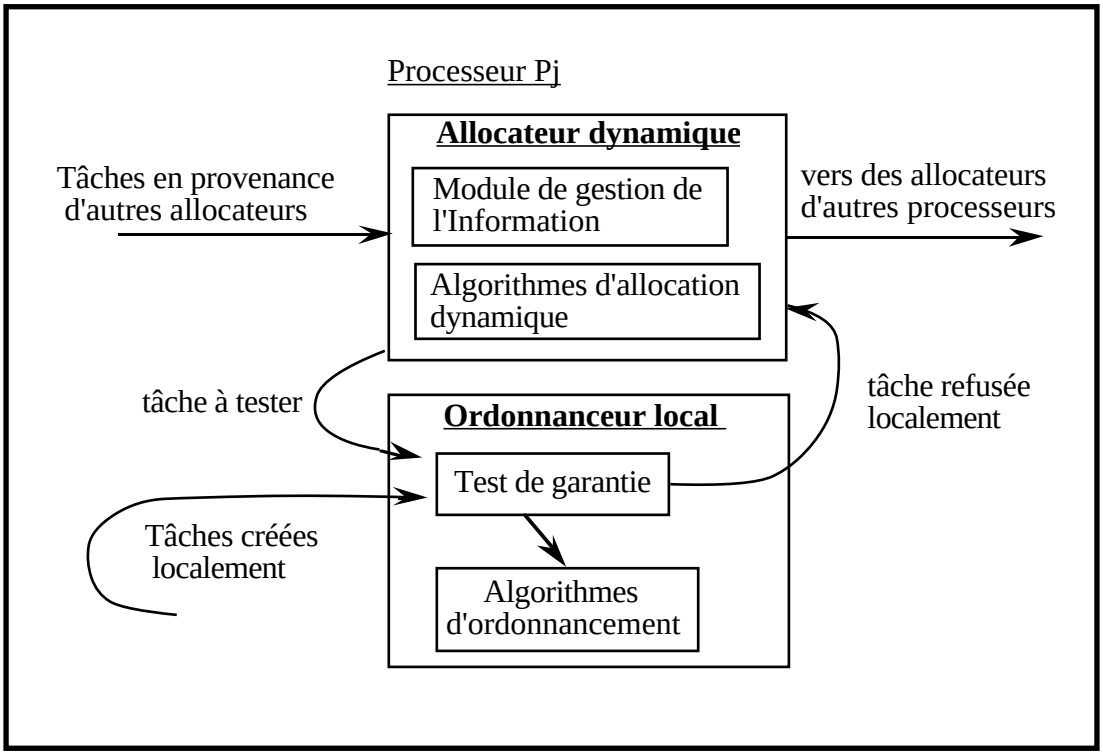


Figure n° 8.1 Coopération entre les allocateurs

Dans les systèmes classiques d'allocation dynamique, l'information échangée entre les processeurs est la charge du processeur. Dans un système temps réel, la charge ne renseigne pas sur la capacité d'un processeur à garantir des dates d'exécution. Quand les tâches à allouer ont la caractéristique temps réel, on s'intéresse plutôt au temps d'oisiveté des processeurs. Ainsi, l'information qui nous intéresse sera le temps libre du processeur. Il s'agit pour chaque processeur de calculer la valeur de son *surplus* et de le communiquer.

Pour pouvoir exploiter cette information de manière appropriée, le surplus est donné par rapport à un intervalle dans le temps qui renseigne sur le cumul des périodes de temps libres entre les tâches. Plus le système est souple et contient des tâches apériodiques, plus le surplus est difficile à calculer. En effet la plupart des tâches étant périodiques, les séquences d'exécution des tâches se réitèrent indéfiniment sur une période L valant le plus petit commun multiple entre les périodes des tâches. C'est pourquoi dans [ShSi91], [RSZh89], [Stan85], [CaKu88], le surplus est donné dans le passé. Les auteurs supposent donc que le comportement du système variant peu, on peut se baser sur le passé pour prédire le futur.

Selon la taille du système, le surplus sera échangé entre tous les processeurs ou uniquement entre groupes de processeurs. Ainsi, chaque allocateur dispose d'une table où il met à jour les surplus reçus. Les techniques d'envoi de surplus seront détaillées dans le chapitre 9 et comparées à la solution que nous proposons afin que cet échange de surplus donne une information cohérente et ne soit pas un handicap, vu le coût élevé des communications dans les systèmes parallèles.

Grâce à la notion de *surplus*, AP_{ref} l'allocateur du processeur où se trouve T_r , la tâche non garantie, dispose d'une estimation sur la capacité des autres processeurs à garantir T_r. Il peut calculer la garantie que peut un processeur fournir pour T_r . On définit cette dernière comme étant le nombre d'instances de T_r qui peuvent être garanties par le processeur :

• Soit gar(T_r , P_j) = SP_j/E_r la garantie donnée à la tâche T_r par le processeur P_j où SP_j est le surplus du processeur P_j et E_r le temps d'exécution de la tâches T_r.

Dans le cas où T_r nécessite plusieurs ressources :

• gar(T_r , *Res_r*) = PPCM ($SP_{j,\mathrm{m}}$)/E_r, avec $SP_{j,\mathrm{m}}$ le surplus pour la ressource *m* rattachée au processeur P_j et *Res_r* l'ensemble des ressources utilisées par T_r.

Il existe différentes politiques pour exploiter le surplus afin de déterminer un processeur pour T_r .

(1) L'algorithme du meilleur surplus (The focused addressing algorithm)

Le principe de cette technique est que l'allocateur se base sur les surplus déjà recueillis pour le choix du processeur. Une garantie pour T_r est calculée pour l'ensemble de ces surplus.

La tâche Tr est envoyée au processeur P_{dest} pour lequel gar(T_r ,P_{dest}) est la plus élevée. Dans [RSZh89] un paramètre système GM (garantie minimale) est fixé en dessous duquel aucun processeur n'est sélectionné et l'algorithme ne peut être appliqué. Une fois que T_r est reçue par P_{dest}, le test de garantie est exécuté. Si la tâche ne peut être ordonnancée, elle est rejetée. Ce cas de figure peut avoir lieu, en effet un certain temps s'est écoulé entre le dernier surplus envoyé par P_{dest} à P_{ref} et la réception de T_r . Afin de limiter les risques de rejet d'une tâche chez le processeur choisi par l'algorithme du meilleur surplus, il est nécessaire que le module qui gère les échanges de surplus entre les processeurs, garde une cohérence forte pour les surplus. Ainsi si cet algorithme est

adopté, on optera pour une fréquence d'échange des surplus assez élevée. C'est à notre avis une manière efficace de maintenir un degré de cohérence élevé pour les surplus. Cependant cette solution qui initialement n'introduit aucun délai (puisque le choix d'un processeur pour T_r ne nécessite aucune communication préalable propre à cet algorithme) risque de devenir coûteuse par suite de l'échange des surplus. la figure 8.2 regroupe les messsages utilisés par l'algorithme du meilleur surplus.

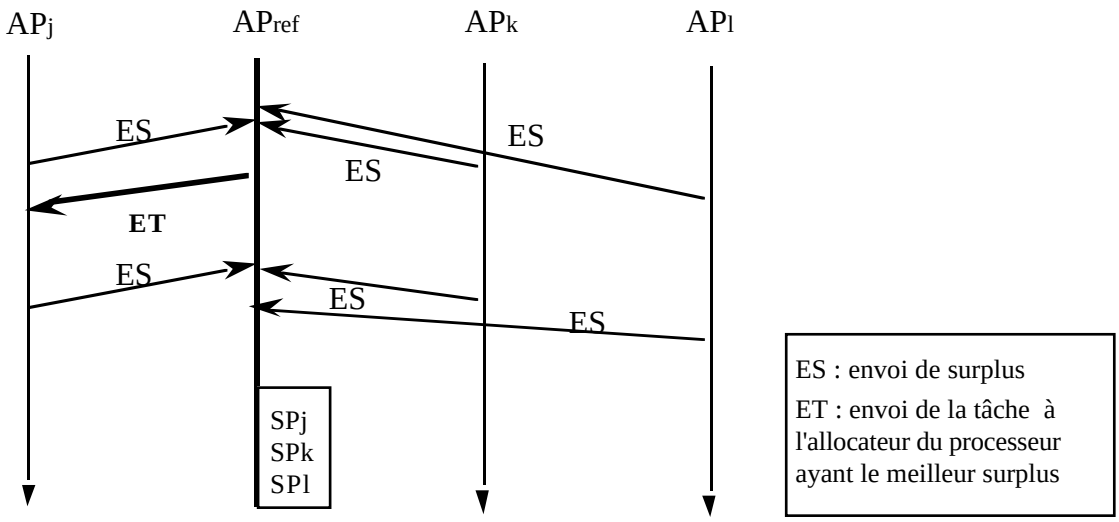


Figure n° 8.2 Les échanges de messages pour l'algorithme du meilleur surplus

(2) L'algorithme des enchères (bidding algorithm)

Cette technique se base sur l'approche de la vente aux enchères afin de prendre les décisions concernant le transfert. L'allocateur d'un processeur qui ne peut assurer l'ordonnancement d'une tâche T_r donnée diffuse une demande d'enchères à un groupe de processeurs et attend une offre sur la garantie qu'ils proposent pour T_r . Le processeur choisi est celui qui propose la meilleure offre.

Une demande d'enchères est propre à la tâche, le message envoyé par AP_{ref} a la structure suivante :

< demande d'enchères (DE) ; T_r (R_r , E_r , D_r) ; P_{ref} >

où AP_{ref} précise les paramètres de la tâche et l'identificateur du processeur auquel il faut renvoyer la réponse. L'offre donnée par les allocateurs des processeurs concernés AP_k est de la forme suivante :

< retour d'offre pour enchères (ROE) ; T_r ; gar(T_r, P_k) >

La figure 8.3 illustre les envois de messages pour l'algorithme des enchères.

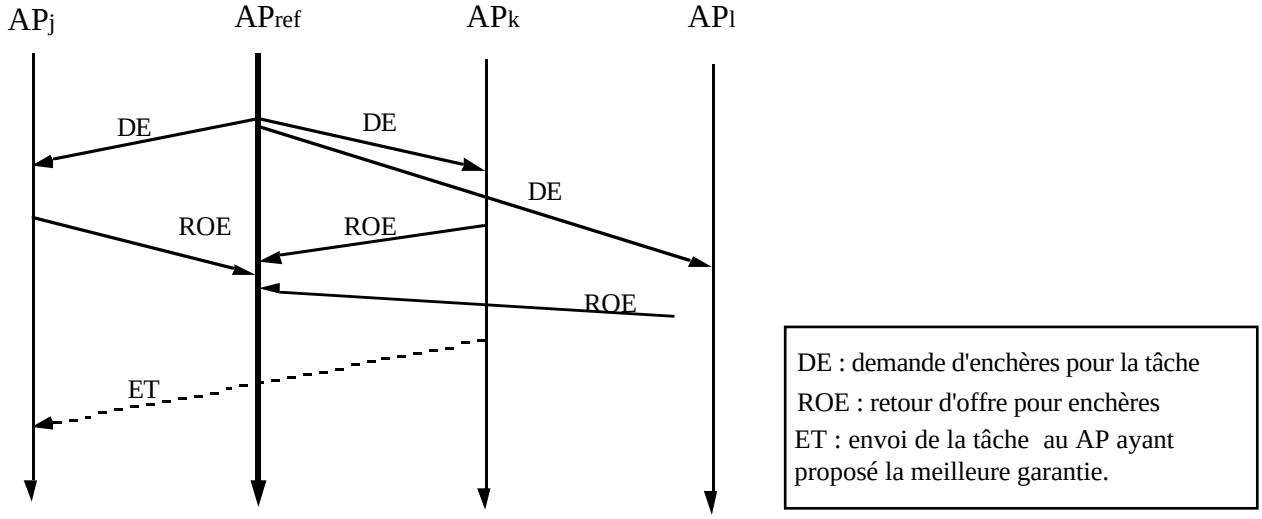


Figure n° 8.3 Les échanges de messages de l'algorithme des enchères

Un allocateur peut émettre plusieurs appels d'offres pour différentes tâches refusées, en effet cet algorithme n'est pas bloquant.

On remarque à ce stade que tout comme l'algorithme du meilleur surplus, il y a calcul d'une garantie pour la tâche. Sauf que dans ce cas ce sont les processeurs qui sont des destinataires potentiels qui calculent cette garantie, ce qui reflète ainsi la réalité au niveau de la charge des processeurs en tâches.

Choix du nombre de processeurs participant aux enchères

Trois cas de figures sont à considérer :

1- T_r est envoyée à tous les processeurs de la table des surplus (mis a part ceux ne pouvant la garantir)

2- Un paramètre système est fixé, soit GM (garantie minimale) en dessous duquel le processeur ne participera pas à l'appel d'offre.

3- Dans [RSZh89], un groupe de k processeurs est sélectionné de manière à ce que la somme des surplus des processeurs soit supérieure à SGS (surplus de garantie du système) un paramètre à fixer.

Une manière d'offrir plus de garantie à la tâche à travers cette technique pourrait être que chaque processeur participant à l'offre insère momentanément la tâche dans sa liste d'ordonnancement. Elle sera ensuite retirée (dans le cas où un autre processeur aura proposé une meilleure offre) après une durée ayant pour valeur au minimum la moyenne du temps nécessaire pour : l'émission de la demande d'offre, la réception de toutes les offres, le choix d'un processeur et l'envoi d'une annulation pour les places auparavant réservées.

Cependant, ce choix ne peut être retenu car un allocateur donné propose des offres pour plusieurs allocateurs, et cette manière de faire va entraîner des offres pessimistes et écarter des cas d'allocation potentiels.

Cette technique est nettement plus coûteuse en communications que celle du *meilleur surplus*, cependant elle offre plus de garantie. Un compromis doit être trouvé entre le nombre de processeurs participant à l'offre afin de réduire le coût des communications et la garantie supplémentaire fournie par cette technique [Stan85], [CaKu88].

(3) L'algorithme mixte

Dans le but d'augmenter le nombre de tâches garanties par l'allocateur, un algorithme mixte a été proposé [RSZh89]. Le principe de l'algorithme est le suivant :

- Il applique dans un premier temps l'algorithme du *meilleur surplus*, à savoir que T_r est envoyée au processeur de la table des surplus de AP_{ref} qui donne la meilleure garantie. Nous le notons P_{dest}.
- Parallèlement, l'algorithme des *enchères* est exécuté à savoir qu'une requête d'offres est envoyée par AP_{ref} aux autres processeurs du groupe sélectionné.
- Les offres des différents processeurs sont retournées à P_{dest}, puisque la tâche lui a été transférée.
- Quand T_r arrive à P_{dest}, le test de garantie est exécuté. Les offres qui seront reçues ne sont prises en compte par AP_{dest} que dans le cas où la tâche est refusée par P_{dest}. Comme il y a eu anticipation dans l'envoi de la tâche, AP_{dest} peut si la tâche est refusée localement retransmettre la tâche sans procéder à une nouvelle élection.

La figure 8.4 illustre les envois de messages pour l'algorithme mixte :

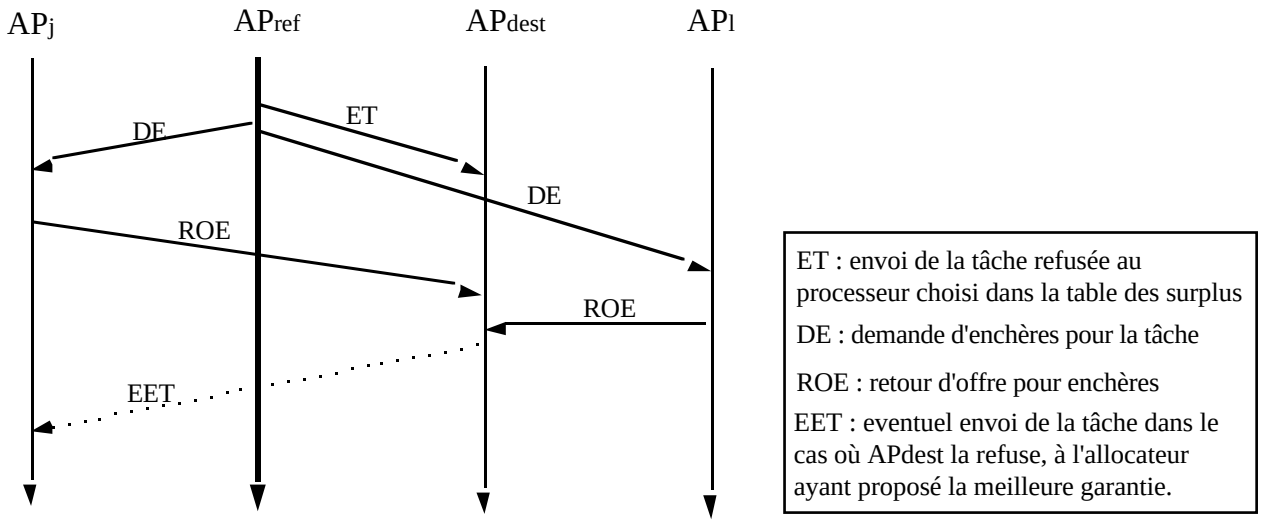


Figure n° 8.4 Les échanges de messages pour l'algorithme mixte

Cette politique d'allocation permet pour le cas d'une tâche dont la laxité est faible (le choix de l'algorithme du meilleur surplus est donc imposé), d'avoir plus de chance de lui trouver un processeur puisqu'un processeur est choisi immédiatement tandis que de meilleures allocations sont recherchées.

Dans [RSZh89], plusieurs simulations ont comparé cet algorithme à celui du *meilleur surplus* et des *enchères*. Le système comportait six processeurs reliés par des canaux (aucun routage n'était nécessaire), chacun exécutant une seule tâche périodique avec les paramètres suivants E_i = 400 unités de temps et P_i = 2000. Les tâches apériodiques arrivent avec une loi de poisson et sont au maximum de six sur 400 unités de temps. Les études faites comparent ces algorithmes en fonction des temps de communication d'un message d'un processeur à l'autre, et des paramètres systèmes fixés pour le surplus SGS et la garantie GM. On observe que l'algorithme *mixte* se comporte mieux que les autres dans tous les cas. Il atteint un taux d'acceptation des tâches apériodiques de 87% quand les délais de communications sont faibles et avoisine les performances de l'algorithme du *meilleur surplus* quand ces délais sont importants (alors que l'algorithme des *enchères* devient inapplicable). Nous précisons que dans l'implémentation de [RSZh89], l'allocateur et l'ordonnanceur sont exécutés sur un coprocesseur.

8.3.2.2 Les techniques à l'initiative du récepteur

L'idée de ces politiques est que les allocateurs sur les processeurs peu chargés prennent l'initiative en proposant leur aide aux processeurs chargés.

(1) L'algorithme d'interrogation

Dans [ChLi86], un algorithme basé sur l'équilibrage de charge en tenant compte de critères de priorités est proposé. Chaque tâche peut avoir trois états : *garantie* si elle est garantie localement, *critique* si elle est à transférer ou en *retard* si son échéance est dépassée. Chaque processeur peut être dans l'un des trois états suivants: *sous-chargé* s'il a moins de $n1$ tâches, *chargé* s'il a entre $n1$ et $n2$ tâches et *surchargé* s'il a plus de $n2$ tâches, ou s'il existe au moins une tâche dans l'état critique ou en retard.

Les auteurs proposent de faire de l'intérrogation sur tous les autres processeurs uniquement dans le cas où le processeur est dans l'état *sous-chargé*. L'algorithme a les étapes suivantes :

- intérrogation des sites par l'allocateur AP_j
- chaque allocateur interrogé
  - recherche et transfère ses tâches critiques ou en retard
  - transfert des tâches garanties vers P_j si

    délai transfert + temps d'exécution de la tâche < échéance

Les auteurs cherchent à travers cette dernière étape de l'algorithme à anticiper *l'effet d'avalanche* en équilibrant la charge des processeurs. Ce cas de figure survient en cas d'alarme où plusieurs tâches apériodiques doivent être exécutées rapidement. Toutefois, le transfert des tâches déjà garanties parait extrêmement douteux dans un système temps réel.

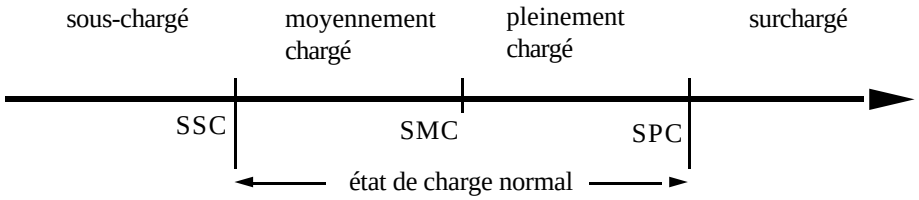
Nous pensons que cet algorithme appliqué tout seul est insuffisant pour garantir d'éventuelles tâches à échéance proche quand le système est chargé, en effet un allocateur doit patienter qu'il y ait un processeur sous-chargé pour que ses tâches refusées soient prises en compte. D'un autre côté, dans un système temps réel, nous estimons risqué de se contenter d'un nombre de tâches pour mesurer la charge d'un processeur. Mesurer la charge avec le taux d'utilisation du processeur et appliquer un algorithme tel que le *Rate-*

monotonic nous paraît plus raisonnable. Des mesures ont été faîtes par les auteurs et montrent que l'algorithme se comporte bien en cas de forte charge (supérieure à 75%).

(2) L'algorithme de diffusion des d'états

Cet algorithme cherche à minimiser les délais introduits par la coopération des allocateurs en diffusant les changements d'états. Notons qu'à chaque processeur est associé un processeur de communication. L'idée est de se à limiter à une vue partielle du système pour chaque allocateur. Chacun conserve ainsi l'état d'une dizaine de ses processeurs voisins.

Ensuite l'algorithme procède par une diffusion des états dans chaque groupe. Chaque allocateur garde une liste; si le message reçu indique que le processeur est surchargé, il est retiré de la liste. Par contre s'il est sous chargé, il y est rajouté. En sus de ces deux états, les auteurs utilisent les états moyennement et pleinement chargé comme l'indique la figure qui suit [ShCh89]:



SSC: seuil sous-chargé, SMC : seuil moyennement chargé, SPC : seuil pleinement chargé

Figure n° 8.5 Les états de charge

La diffusion des états se fait quand la charge du processeur évolue entre sous-chargé, normalement chargé ou surchargé. La liste des allocateurs reflète ainsi la charge des processeurs puisque l'information échangée n'est pas ponctuelle dans le temps, mais correspond à un état.

8.4 Conclusion

On peut remarquer, que tous les algorithmes de l'approche coopérative utilisent une connaissance de l'état des autres processeurs par une diffusion totale ou partielle de cette information. Ils appliquent tous une coopération pour répartir la charge du système soit de manière directe comme les techniques à l'initiative du récepteur soit indirecte puisqu'un allocateur qui a des tâches refusées et qui cherche à les placer peut être qualifié de chargé.

Toutefois, la mesure de la charge dans les techniques à l'initiative du récepteur est à notre avis, loin d'être satisfaisante pour des systèmes temps réel où les contraintes doivent être considérées individuellement. Nous optons plutôt si de telles techniques sont appliquées, pour une mesure de charge basée sur U_i l'utilisation du processeur.

Les résultats de simulations présentés par les divers auteurs concernent des systèmes distribués où le nombre de processeurs est faible (un maximum de dix), ce qui ne renseigne pas sur les délais qui peuvent être introduits lors de l'implémentation de tels algorithmes sur des machines parallèles. Il ne nous est donc pas possible de comparer leurs performances à celles obtenues lors de notre implémentation de certains de ces algorithmes sur une machine parallèle. Toutefois pour une implémentation sur des machines distribuées, les coûts présentés par les auteurs sont satisfaisants dans la mesure où une allocation nécessite un délai qui représente un faible pourcentage du temps d'exécution de la tâche allouée.

Chapitre 9

Un mécanisme pour l'allocation dynamique de tâches temps réel

9.1. Introduction

Nous présentons dans ce chapitre, un mécanisme d'allocation dynamique de tâches temps réel. Les algorithmes appliqués pour la gestion de l'état du système et pour la recherche de processeurs, sont distribués et indépendants de la taille et de la topologie du réseau. Nous avons veillé à trouver un compromis entre la réduction du temps nécessaire à la recherche d'un processeur (ce qui permet d'augmenter le nombre de tâches allouées avant leur échéances) et le degré de cohérence de l'information sur l'état du système.

Dans un premier temps, ce chapitre présente la structure du mécanisme. Ensuite, nous décrivons les modules d'information et de décision. Les résultats de l'implémentation sont présentés à la fin du chapitre.

9.2 Le module information

Ce module est chargé de maintenir un état partiel du système. Nous avons discuté dans le chapitre 8, de la manière dont le surplus est calculé. En fait, il est prédit en se basant sur les valeurs précédemment reçues car la plupart des tâches sont périodiques. En revanche, quand le système est plus souple et contient plusieurs tâches apériodiques, le surplus est moins évident à calculer.

Pour ces raisons, le module d'information que nous proposons, calcule son surplus par rapport à l'instant courant dans un intervalle dans le futur. Pour cela, la taille de la fenêtre est fixée *tail_fen* et le surplus est calculé dans l'intervalle [t, t+*tail_fen*].

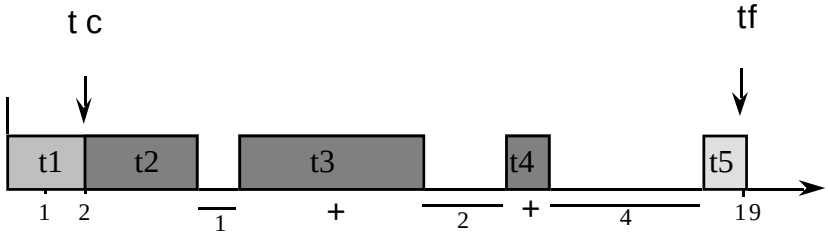
A la réception d'un tel surplus, un allocateur donné AP_{ref} ayant une tâche refusée T_r, peut juger si le processeur émetteur du surplus est qualifié pour la garantie de la tâche refusée. Il peut donc comparer les paramètres de la tâche aux bornes de la fenêtre et calculer la garantie que pourrait avoir Tr sur ce processeur.

9.2.1 Calcul du surplus

Chaque allocateur évalue son surplus et l'envoie à un ensemble de processeurs du réseau. Une définition plus précise du surplus est donnée ci-dessous :

Définition 1 : Un surplus SP_j est un vecteur où la première composante représente le surplus du processeur P_j, et où les autres composantes représentent le surplus pour chaque ressource r_i rattachée au processeur P_j :

Un exemple de calcul de surplus est donné dans la figure 9.1.



Surplus de la ressource processeur à l'instant t = 19 vaut 7

Figure n° 9.1 : Exemple de calcul du surplus

9.2.2 Maintien d'un état du système

Nous nous intéressons à présent aux méthodes pour sélectionner le groupe des processeurs auxquels sera envoyé le surplus. On distingue deux approches : la première est empirique et suppose qu'un processeur ayant déjà garanti des tâches refusées peut en accepter d'autres. La seconde est une approche semi-centralisée dans la mesure où on désigne certains processeurs pour assurer la gestion du réseau. Nous notons, P_{rec} le processeur auquel vont être envoyés les surplus, AP_{rec} l'allocateur sur ce processeur et P_i un processeur qui va envoyer un surplus à P_{rec}.

- gestion empirique : un groupe est formé avec tous les processeurs qui ont déjà garanti des tâches pour le processeur P_{ref} et le surplus est envoyé aux processeurs de ce groupe. Dans un système parallèle, où le nombre de processeurs peut être très élevé, cette gestion ne peut être appliquée sans adaptation. En effet, la distribution géographique des processeurs du groupe introduit un surcoût de communication. D'un autre côté, un problème se pose pour fixer le nombre maximum des processeurs du groupe. Doit-on former le groupe en mettant l'accent sur la garantie à fournir, ou doit-on se préoccuper principalement du coût des communications et ne garder que les processeurs dont la localité géographique est proche? Un compromis est difficile à trouver. Toutefois, appliquer cette approche dans un système de faible taille, peut être efficace. C'est le cas dans [StRa89] où le réseau est composé de 6 processeurs.

- gestion semi-centralisée : Dans cette solution, on définit un allocateur serveur pour un ensemble de processeurs qui gardera les surplus de ces derniers. Un allocateur ne pourra décider d'envoyer une tâche vers un autre processeur à partir d'informations locales, il devra consulter l'allocateur serveur auquel il appartient qui pourra à son tour consulter d'autres serveurs jusqu'à obtenir un processeur avec un surplus suffisant pour garantir la tâche en question. Un sérieux problème de tolérance aux pannes se pose si un ou plusieurs serveurs tombent en panne. Faut-il dupliquer les surplus gardés par un serveur sur un autre processeur et le dédier à cette tâche? Quel processeur choisir pour cette duplication? Une gestion sécurisée de cette centralisation introduit des coûts importants en mémoire et communications. D'un autre côté, ce choix est plus adapté à une machine à mémoire partagée et ne permet pas d'exploiter pleinement le parallélisme du système.

Nous avons développé une gestion plus appropriée aux systèmes parallèles. Chaque processeur applique le même algorithme et gère lui-même ses surplus:

- gestion parallèle : nous estimons que disposant d'un système parallèle, il est intéressant d'exploiter au maximum le parallélisme, en adoptant un même algorithme pour le maintien des surplus à exécuter par tous les processeurs. Celui-ci est basé sur la notion de voisinage, à savoir qu'un processeur reçoit le surplus de tous ses voisins physiques. On vise à ne garder sur chaque processeur P_i que de bons surplus, pour cela on définit SM (surplus minimal) un paramètre de mesure de la qualité des surplus, en dessous duquel le surplus n'est pas envoyé. Une alternative est fournie pour le cas où le nombre de voisins est faible. Si plusieurs voisins ne peuvent envoyer leurs surplus, l'allocateur

risque de se retrouver avec un nombre réduit de surplus, et le choix du processeur pour une tâche refusée est ainsi limité.

Ainsi, nous proposons dans le cas où P_i , un des voisins de P_{rec} a un surplus faible, qu'il examine les surplus de ses voisins à la recherche d'un processeur avec un surplus suffisant. Cette recherche est rapide puisque P_i dispose dans sa table des surplus de ses voisins. Une fois un processeur avec un bon surplus déterminé, son surplus est envoyé au processeur P_{rec}.

La figure 9.2 présente l'algorithme *parallèle pour l'échange des surplus* :

En parallèle
- recevoir
  recevoir un surplus soit SP_k
  rajouter l'identificateur du processeur P_k et SP_k dans Table_surplus
- émettre
  Soit SP_j le surplus du processeur courant
  Si SP_j < SM alors
    rechercher dans Table_surplus un surplus SP_k qui soit > SM
    envoyer SP_k à tous mes voisins
  sinon envoyer SP_j à tous mes voisins

Figure n° 9.2 L'algorithme de l'échange des surplus

Cette approche basée sur le voisinage devrait limiter le nombre de surplus gardés sur un processeur donné (nombre de voisins physiques), et par suite réduire les chances de déterminer un processeur pour la tâche non garantie. Toutefois, la modification apportée par l'envoi des surplus des voisins remédie à cet inconvénient.

Concernant les techniques utilisées pour l'échange des surplus, l'échange peut être périodique, relatif à la variation du surplus, ou à la demande. Toutefois, cette éventualité est écartée dans des systèmes temps réel pour cause des délais qu'elle introduit.

- l'échange périodique : Le problème reste à déterminer une période appropriée, de manière à éviter les deux extrêmes où le réseau est inondé par les communications et celle où l'information ne reflète pas la réalité. Le choix de cette stratégie est étroitement lié à la

politique de transfert adoptée. Pour une politique qui envoie la tâche au processeur qui offre le meilleur surplus, il est nécessaire que ce dernier soit fréquemment mis à jour.

- l'échange relatif : dans ce cas, l'échange se fera chaque fois que le surplus variera. Nous rappelons que cette situation a lieu quand le processeur reçoit de nouvelles tâches et qu'elles sont acceptées par l'ordonnanceur local. Il est cependant nécessaire à notre avis, d'imposer un seuil pour le cas où le surplus varierait trop souvent.

Ainsi on peut aussi envisager des solutions, où on impose une valeur minimale à la variation du surplus. Dans ce cas le surplus est envoyé quand ce seuil est dépassé. Le choix d'une stratégie pour l'échange dépend de la politique d'allocation en vigueur.

9.3 Le module Décision

Nous exposons dans cette section une heuristique pour le choix d'un processeur quand une tâche est non garantie localement. Parmi les politiques présentées au chapitre 8, laquelle est la mieux adaptée à un système parallèle temps réel? Dans un système parallèle, un handicap majeur est le coût des communications. Ces dernières sont nombreuses et coûteuses. Par conséquent la politique d'allocation des tâches ainsi que l'élément d'information ne doivent pas être trop coûteux en communications. D'un autre côté, parmi les techniques présentées au chapitre précédent, certaines sont plus appropriées à des problèmes temps réel que d'autres. Ainsi, les techniques à l'initiative du récepteur nous paraissent à appliquer avec certaines précautions, étant donné le fait que la charge du processeur est mesurée en nombre de tâches et non selon les valeurs du taux d'utilisation du processeur. A notre avis, un processeur peut avoir uniquement deux tâches à ordonnancer et ne pas être peu chargé pour autant..

Les techniques à l'initiative de l'émetteur nous paraissent bien adaptées au problème d'allocation sous contraintes de temps critiques. L'algorithme mixte ainsi que celui des enchères ont été implémentés par notre mécanisme, ceci afin de les évaluer et de comparer les performances dans un système parallèle.

Le module de décision implémente deux algorithmes, celui mixte et celui des enchères. Il choisit entre les deux en fonction de la valeur de la garantie calculée. Cette dernière est comparée à un paramètre système GM (Garantie minimale) en dessous duquel

il est inutile d'allouer la tâche à un processeur, mais plutôt nécessaire de rechercher d'autres allocations par l'algorithme des enchères :

```
• tant que il y a des tâches refusées
faire
        calculer la garantie gar(Tr , Pj ) = SPj/Er pour tous les processeurs
        de table_surplus
        soit Gar(Tr) cette valeur et Pk le processeur associé
        si gar(Tr,Pk) > GM ( le nombre d'exécutions est suffisant pour garantir Tr)
            alors l'algorithme mixte est appliqué
        sinon l'algorithme des enchères est appliqué
fin faire
```

Figure n° 9.3 L'algorithme d'allocation dynamique

9.3.1 Une heuristique pour l'allocation dynamique des tâches

Le mécanisme propose une heuristique pour les tâches refusées dont le temps de latence est très court. Elle consiste à retirer de la *liste_ordonnancement* une tâche auparavant acceptée dont le temps de latence permet de l'allouer dynamiquement, et d'accepter la tâche refusée.

(1) Garantie des tâches à faible temps de latence

Outre la validation du mécanisme, son implémentation nous a permis d'estimer les délais nécessaires à la recherche d'un processeur pour une tâche refusée. En nous basant sur ces délais, il est désormais possible pour une tâche refusée donnée de juger s'il sera possible de l'allouer à un autre processeur.

L'heuristique que nous proposons se base sur ces délais pour augmenter le ratio de garantie du mécanisme. Soit le cas d'une tâche T_r refusée localement pour laquelle le temps de latence est insuffisant pour appliquer un des algorithmes d'allocation. L'idée est la suivante : T_r ayant un temps de latence court, elle est considérée comme une tâche urgente. Toutefois, elle risque d'être non garantie sur aucun processeur alors que d'autres tâches reçues auparavant sont moins urgentes dans la mesure où elles ont un temps de latence plus grand.

Nous soulignons au passage un des principaux inconvénients du modèle dynamique, en effet dans un modèle statique, l'ordre de traitement des tâches importe peu, c'est la finalité de l'ordonnancement et de l'allocation qui est visée. Cependant, dans un modèle dynamique, des tâches plus urgentes peuvent être reçues ou créées après d'autres tâches moins urgentes et dépasser ainsi leur échéances.

Dans un but de garantir le maximum de tâches reçues, et de réduire le nombre d'ordonnancements corrects qui sont ainsi écartés, nous proposons de revoir l'hypothèse d'ordonnancement local (2.1) qui interdit à une tâche de remettre en question les tâches déjà ordonnancées sur le processeur, en sélectionnant une autre tâche à transférer et en acceptant T_r localement. Nous veillerons toutefois à garantir l'ordonnancement de la tâche choisie.

L'acceptation de T_r localement implique le déroulement de l'algorithme d'ordonnancement local afin de déterminer l'emplacement auquel un élément représentant la tâche sera inséré. Ensuite, l'algorithme recherche les tâches qui dépassent leurs échéances, soit ST_{ref} cet ensemble (les tâches de ST_{ref} sont appelées des tâches échangées puisqu'elles échangent leur places avec une tâche apériodique plus urgente qu'elles). A l'ensemble ST_{ref}, sera appliquée l'heuristique d'allocation dynamique. Toutefois, afin d'accepter T_r , l'ensemble des tâches qui deviennent refusées doit satisfaire les conditions suivantes :

- chaque tâche de l'ensemble doit avoir un temps de latence suffisant pour dérouler l'algorithme d'allocation dynamique et déterminer un processeur capable de la garantir. Nous définissons le paramètre *délai_garantie* qui est une valeur majorée de la moyenne du temps nécessaire à l'allocation.

- aucune des tâches ne doit avoir la priorité critique définie par le programmeur. Nous rappelons que les tâches critiques sont allouées statiquement afin d'éviter tout risque de non exécution ou de dépassement d'échéance.

Nous avons jugé inutile et surtout coûteux en temps de partager la mise en œuvre de l'heuristique entre le mécanisme d'ordonnancement local et celui d'allocation dynamique. En effet, le mécanisme d'ordonnancement local pourrait se charger de déterminer l'ensemble des tâches à transférer à la place de T_r soit ST_{ref}, tandis que le mécanisme d'allocation dynamique vérifierait que les tâches de cet ensemble peuvent être transférées. De nombreux problèmes se posent dans ce cas de figure :

(1) doit-on suspendre le mécanisme d'ordonnancement local jusqu'à ce que l'ensemble ST_{ref} soit vérifié ? Le cas échéant, la présence de T_{ref} dans *liste_ordonnancement* peut perturber l'ordonnancement qui va suivre puisque T_{ref} risque d'être retirée par la suite.

(2) Dans ce cas une *liste_sauvegarde* doit être mémorisée et mise à jour en fonction des tâches acceptées entre temps afin de pouvoir reconstituer la *liste_ordonnancement.*

Au vu de ces raisons, nous estimons trop complexe cette manière de procéder. Pour cela, nous avons modifié l'algorithme d'ordonnancement local afin qu'il procède à la détermination de ST_{ref} et à la vérification de la faisabilité de l'allocation pour ses tâches.

(2) Adaptation de l'algorithme d'ordonnancement local

Dans l'action ordonnance_tâche, un élément est inséré pour la tâche à garantir même si on ne dispose pas de suffisamment de temps libre. Au lieu de défaire l'ordonnancement réalisé dès qu'une tâche réordonnancée dépasse son échéance, nous proposons de constituer l'ensemble ST_{ref}. Par la suite, chaque tâche de l'ensemble est testée, si elle n'est pas critique et que son temps de latence est suffisant pour l'allouer (le paramètre *délai_garantie* indique le temps moyen nécessaire à l'allocation d'une tâche), elle est placée dans une *liste_tâches_refusées* (la liste pour laquelle l'algorithme d'allocation est déroulé). Nous précisons que *liste_tâches_refusées* contient les tâches non ordonnancées localement, celles reçues d'autres processeurs (à travers l'exécution de l'algorithme d'allocation) et celles échangées.

L'action *ordonnance_tâche* doit être modifiée de la manière suivante afin que l'heuristique d'allocation puisse être appliquée.

Début
retirer une tâche de *liste_verification*
rechercher I_debut
calculer temps_libre
insérer un élément pour la tâche
si non trouvé suffisamment de temps libre
    parcourir *liste_ordonnancement* à la recherche des tâches qui dépassent

leurs échéances une fois réordonnancées.
Soit ST_{ref} l'ensemble de ces tâches
Pour chaque tâche T_{ref} de ST_{ref} ,
si temps de latence est supérieur à *délai_garantie* et
priorité(T_{ref}) différente de critique
placer T_{ref} dans *liste_tâches_refusées*
sinon arrêter le parcours de ST_{ref} , défaire et
reconstituer *liste_ordonnancement*
Fin

(3) L'algorithme d'allocation dynamique

L'algorithme des enchères a été adapté afin que les tâches appartenant à ST_{ref} soient garanties et qu'elles ne cèdent pas leur places pour rien. Cette adaptation est basée sur la réservation sur le processeur destinataire.

Lors de l'envoi du message d'appel d'offres, l'algorithme précise qu'une réservation doit être réalisée, ce qui signifie que l'allocateur du processeur qui reçoit le message devra réserver le temps nécessaire à cette tâche en l'insérant dans sa *liste_ordonnancement*. De cette manière, une fois que l'allocateur ayant lancé l'appel d'offres a reçu une offre, il est sur que la tâche en question sera garantie.

La libération des emplacements réservés sur les autres processeurs se fera suite à la réception d'un message d'annulation. Une autre manière de faire afin d'éviter de surcharger le réseau de messages serait d'annuler l'emplacement après l'écoulement d'un délai de garde. Le choix d'une des alternatives se fera après estimation du délai de garantie par le mécanisme.

Nous devons également imposer une limite sur le nombre des tâches échangées à transférer. Même si ces tâches seront garanties ailleurs, il ne faut pas néanmoins ignorer les messages supplémentaires dus à ces tâches, aussi bien ceux pour la recherche d'un processeur que ceux pour l'annulation de l'emplacement réservé sur les processeurs le moment venu. D'un autre côté, le traitement de ces tâches par l'allocateur retarde celui d'autres tâches non échangées, et réduit ainsi leur chances d'être allouées et exécutées avant leur échéances. Le nombre maximum de tâches échangées peut être fixé mais ce

paramètre doit être approximé par simulation, en effet il dépend des délais de recherche d'un processeur et également de la charge de l'allocateur.

Il existe une autre manière de garantir une tâche refusée en invoquant la migration de la tâche, si l'allocateur peut garantir l'exécution d'une partie de la tâche.

(4) Cas où l'allocateur ne peut garantir l'exécution de T_r en entier

Dans le cas où la tâche refusée peut être exécutée en partie sur le processeur, on peut envisager d'appliquer l'algorithme d'allocation afin de rechercher un processeur capable de garantir la partie de code de la tâche qui ne peut être exécutée localement. L'idée est qu'il est plus probable de garantir une partie de la tâche que la tâche en entier.

Ainsi au lieu d'échanger des tâches, il serait intéressant de rechercher par la politique d'allocation en cours, un temps plus réduit sur les processeurs. En effet, supposons que le processeur courant puisse exécuter T_r pendant E_{cour}, l'algorithme d'allocation devra donc chercher un processeur pouvant garantir T_r pour une exécution de durée E_r - E_{cour}ce qu'il est plus probable de trouver. Toutefois quel est le prix de commencer l'exécution de la tâche et de l'interrompre par la suite pour la migrer? Est-il raisonnable d'utiliser la migration dans les systèmes temps réel?

La migration introduit un surcoût important pour le transfert du processus et la gestion qui s'en suit. Le coût du transfert est important car il est proportionnel à la taille du processus transféré. Il comprend le contexte du processus y compris son espace d'adressage (contenu des piles, des registres, descripteurs des fichiers ouverts, etc). Dans certains systèmes temps réel, la plupart des créations de tâches qui ont lieu dynamiquement sont connues avant l'exécution, c'est le moment de création qui ne l'est pas. Ainsi le code des tâches peut être placé sur certains processeurs voire même dupliqué sur plusieurs processeurs. Une fois la tâche créée dynamiquement, on a accès au fichier contenant le code de la tâche.

Avoir recours à un mécanisme de migration pour l'allocation du processus rajoute outre le coût effectif de transfert, un coût pour la gestion du processus migré. En effet, une des plus importantes propriétés de la migration est la transparence. Elle est réalisée si tous les processus communicants avec le processus migré s'exécutent et communiquent exactement comme si le processus n'avait pas migré. Elle consiste essentiellement au rétablissement des communications.

Le recours à la migration dans les systèmes temps réel est un domaine peu exploré [Gait90]. Rares sont les travaux qui l'ont effectivement appliquée. Une thèse sur la migration dans les systèmes parallèles a été réalisée dans notre équipe par [Elle94]. A notre avis, le coût de l'algorithme d'allocation est déjà assez important pour le surcharger avec celui de la migration. Toutefois, quand la granularité des processus est fine, on peut supposer le coût de migration constant et disposer ainsi d'une estimation sur les délais à rajouter.

9.4 Mise en œuvre du mécanisme d'allocation dynamique

9.4.1 Modélisation

Nous avons mis en œuvre le mécanisme d'allocation dynamique, sur une machine parallèle composée de 16 transputers T800, reliés par un réseau d'interconnexion. La topologie du réseau est une grille torique. L'implémentation de ce mécanisme a nécessité l'exécution de quatre processus en parallèle.

Un premier processus implémente l'algorithme de l'échange des surplus, un second l'heuristique d'allocation dynamique proposée. Un troisième, gère la communication. Trois types de messages peuvent être reçus : un message contenant un surplus, un message indiquant une requête d'offres, et un dernier contenant une offre. La réception des tâches transférées est assurée par un quatrième processus, en effet, la taille d'un message de transfert de tâches est bien plus importante que celle de ceux gérés par le troisième processus. Nous rappelons que dans certaines applications, le code des tâches à créer dynamiquement peut être stocké en mémoire sur différents processeurs, auquel cas, ce code est transmis par message au processeur choisi par l'algorithme d'allocation. Cette manière de procéder, permet d'observer l'influence de la circulation de ces messages de différentes tailles.

Un cinquième processus implémente le mécanisme d'ordonnancement local. Enfin, un dernier processus est chargé de lancer les tâches, une fois leur dates d'exécution atteinte. L'exécution réelle est en fait simulée par une attente d'une durée égale au minimum au temps d'exécution de la tâche.

Dans un premier temps, nous avons mesuré le temps moyen nécessaire à la recherche d'un processeur lors de l'allocation. Il est évident que ce coût n'est introduit que quand

l'algorithme des enchères (AVAE) est appliqué. Nous évaluons ce coût par la mesure des paramètres suivants :

Coût_AVAE = *délai_émission_req + délai_réception_offres + délai_transfert_tâche*

Pour mesurer le délai d'émission d'une requête d'offres, nous avons mesuré le temps moyen d'acheminement à travers le réseau, d'un message contenant les paramètres d'une tâche pour laquelle l'algorithme est déroulé. Le protocole de communication adopté est le "port asynchrone" offert par le noyau ParX. Les mesures que nous avons effectuées montrent qu'il faut en moyenne entre 2 et 4 ms pour acheminer un message de requête dans un système où tous les processeurs reproduisent le même comportement.

Concernant le *délai_réception_offres* , ce dernier comporte le temps d'acheminement du message d'offre mais également le temps nécessaire au traitement du message de requête et au calcul de l'offre. Les processus s'exécutant en parallèle, il s'en suit un certain indéterminisme dans leur ordre d'exécution, qui fait qu'il est difficile d'estimer le temps de la prise en compte du message de requête. Pour éviter de trop longues attentes qui peuvent être à l'origine de dépassement d'échéances, nous avons instauré un délai de garde. Sa mise en œuvre repose sur la valeur de l'offre et sur celle du temps de latence dynamique de la tâche. Ainsi, si le mécanisme reçoit une garantie qualifiée de bonne, la tâche est allouée, autrement on attend l'expiration du délai de garde. Dans ce cas, ce dernier est un pourcentage du temps de latence de la tâche. Un paramètre du système Garantie Suffisante (GS) est utilisé en plus de la garantie minimale (GM) introduite auparavant, afin de qualifier une garantie de "suffisamment bonne" pour être exploitée.

9.4.2 Mesures de performances

Pour les mesures réalisées, GS a été fixée à 4 et GM à 2. Nous avons observé des temps d'allocation de tâches qui varient entre 10 ms et 40 ms pour un nombre de tâches non ordonnancées sur toute la machine qui varie entre 10 et 48. Ces temps, sont relativement importants en partie, à cause des communications. Une implémentation de la communication qui exploite directement les liens du transputer devrait permettre de réduire considérablement ces temps. Toutefois, le routage des messages ne serait pas fourni. Ces temps nous permettent de décider à quelles tâches appliquer une allocation dynamique. Ainsi, dans le cas où on aurait le choix entre une allocation statique et une création et une allocation dynamique de certaines tâches, il est prudent de ne pas allouer

dynamiquement des tâches dont les temps de latence sont inférieurs à 30 ms. D'un autre côté, ces mesures permettent également de choisir quand appliquer l'heuristique que nous avons proposée à la section 9.3.1.

L'implémentation du mécanisme d'allocation dynamique a permis de mesurer les performances du mécanisme d'ordonnancement distribué par la donnée du ratio de garantie (noté R). Ce dernier est défini comme le rapport entre les tâches ordonnancées dynamiquement et le nombre total de tâches créées.

Sur chacun des processeurs du réseau, nous avons placé 10 tâches. La moyenne de création des tâches apériodiques varie entre 1.6 ms et 7.8 ms. La moyenne des temps d'exécution des tâches est fixée à 4 ms. Dans la figure 9.4, nous illustrons une comparaison du ratio de garantie du mécanisme d'ordonnancement distribué en fonction de l'algorithme d'allocation appliqué pour différents états de charge du processeur :

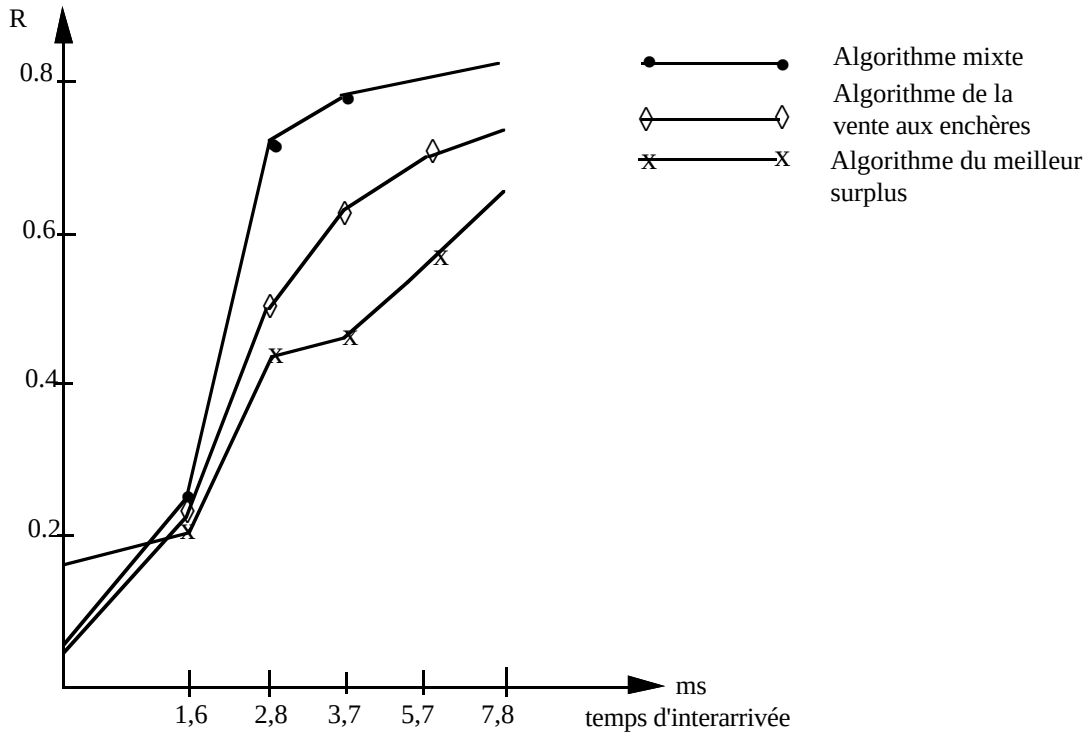


Figure n° 9.4 Performances du mécanisme en fonction des politiques d'allocation

On remarque un fonctionnement proche pour les trois algorithmes en cas de forte charge (temps d'arrivée inférieur à 2 ms) à savoir que le ratio n'excède pas les 25%. Il ressort toutefois que l'algorithme du meilleur surplus se comporte mieux que les autres. En effet, c'est celui qui introduit le meilleur rapport : nombre de tâches acceptées par rapport au coût des communications.

Dans des situations de charge moyenne, l'algorithme mixte donne un ratio avoisinant les 80%. Même si les communications sont importantes en charge moyenne, cet algorithme se comporte bien, en effet il applique l'algorithme du meilleur surplus qui peut tout de suite allouer la tâche. De plus, l'exécution des enchères donne davantage de garantie au cas où la tâche est refusée après cette allocation.

Nous avons également comparé les ratios donnés par ces algorithmes en fonction du temps de latence des tâches. Les trois courbes représentent des mesures pour une moyenne du temps de latence qui vaut respectivement : 8 ms, 5,3 ms et 2,5 ms. La figure 9.5 en est une illustration :

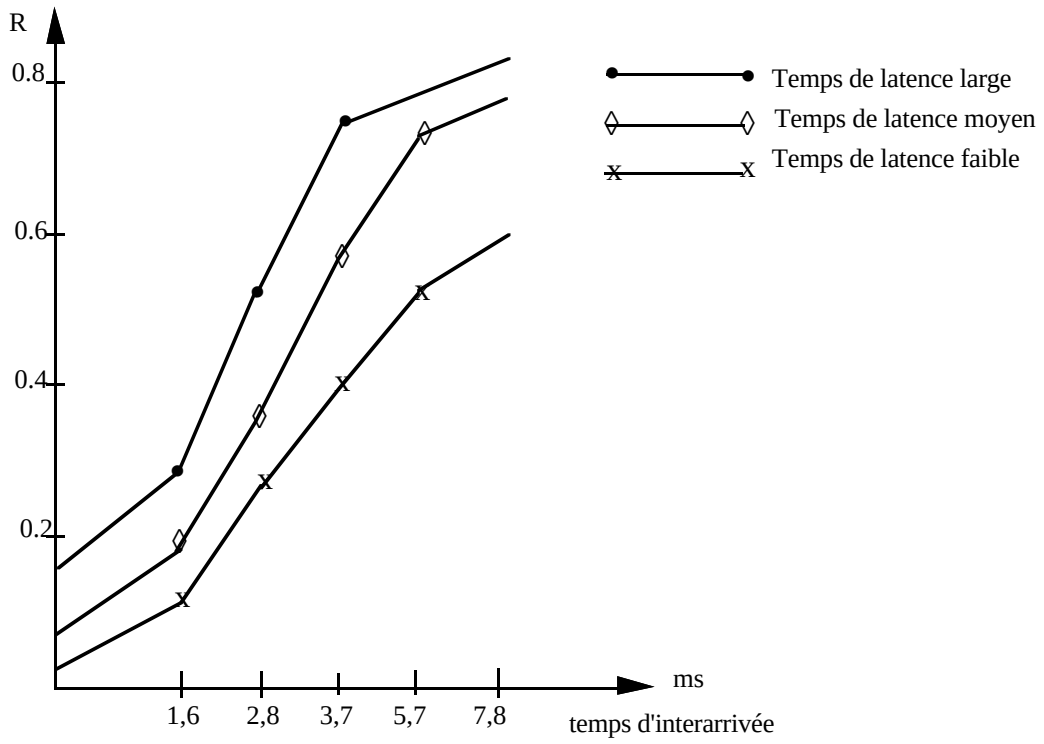


Figure n° 9.5 Influence du temps de latence

Il ressort que les performances du système ne sont pas une fonction linéaire du temps de latence. Toutefois, un temps de latence court dégrade le ratio de garantie, en effet si les tâches ne disposent que de peu de temps pour s'exécuter, quel que soit l'algorithme appliqué, les performances du système s'en ressentent.

D’autres mesures des performances du mécanisme d’ordonnancement distribué sont en cours. Elles consistent à comparer les politiques présentées à des politiques non

coopératives tel que l'algorithme aléatoire afin de voir le comportement de ce dernier dans des situations de moyenne et forte charge. Il serait également intéressant de voir le comportement des algorithmes mis en œuvre face à des processeurs de charges différentes. Nous prévoyons également d'étudier l'impact de temps d'exécution peu précis sur le ratio de garantie (des tâches qui libèrent le processeur plus tôt ou plus tard).

Conclusion et perspectives

Dans cette thèse, nous avons considéré une grande variété de problèmes qui se posent dès lors qu'on aborde le domaine de l'ordonnancement dans des systèmes temps réel. La plupart des problèmes rencontrés proviennent du fait que les applications temps réel nécessitent un respect absolu des contraintes de temps et des ressources qui les caractérisent. Concernant les systèmes qui supportent ce genre d'applications, nous avons remarqué qu'il y a souvent confusion entre les systèmes temps réel et des systèmes offrant des primitives de bas niveau afin d'accélérer les calculs et s'exécutant sur des machines très performantes. Des systèmes d'exploitation adéquats doivent être conçus pour supporter de telles applications.

A la différence des systèmes conventionnels (non temps réel), une prise en compte de la notion de temps est indispensable à plusieurs niveaux d'abstraction dans un système d'exploitation temps réel. Ainsi, un mécanisme d'ordonnancement doit être intégré à un système d'exploitation qui permet de le valider par des mécanismes de communication où l'acheminement des messages temps réel peut être traité avec priorité, également des mécanismes de gestion de la mémoire qui allouent en priorité les pages nécessaires selon le niveau de priorité, ainsi que des mécanismes de gestion du temps et autres.

Dans cette thèse, nous avons conçu et mis en œuvre, un mécanisme *d'ordonnancement distribué* de tâches temps réel. Ce mécanisme d'ordonnancement peut être décrit par quatre mots clés : *statique, dynamique, local* et *distribué*.

Les aspects *statique* et *dynamique* sont dus aux classes d'application temps réel existantes. Souvent les paramètres décrivant les tâches sont connus à l'avance, et on parle alors de systèmes temps réel statiques. De nombreuses applications temps réel sont statiques et incluent des tâches périodiques, ce qui implique un certain déterminisme dans leur exécution. Le mécanisme d'ordonnancement que nous avons présenté garanti ce déterminisme par la donnée d'algorithmes appropriés à ce modèle de tâches statique.

Notre mécanisme peut également être appliqué à des modèles dynamiques. Depuis quelques années, on observe un réel besoin en systèmes temps réel capables de gérer des applications appliquant des modèles dynamiques et plus souples. D'un côté, il existe des applications temps réel qui n'ont pas de contraintes strictes et qui nécessitent la création dynamique des tâches. D'un autre côté, ce besoin de systèmes plus souples découle entre autres, du fait qu'il est à la fois difficile et coûteux de connaître tous les paramètres des tâches avant le début de l'exécution. Le problème se pose essentiellement lors de la présence de tâches apériodiques, dont on ne connaît avec précision ni le nombre d'occurrences ni les paramètres d'exécution.

Ces tâches, tout comme celles périodiques, peuvent avoir des contraintes strictes. L'efficacité du mécanisme se traduit par la donnée d'algorithmes capables de traiter ces tâches dès leur création et de garantir le respect de leurs contraintes.

Il faut tout de même rester vigilant car même dans ces systèmes temps réel souples, certaines tâches sont plus urgentes que d'autres, à savoir les tâches périodiques, et le mécanisme doit avant tout garantir celles-ci. Autrement dit, il est intéressant de disposer d'un mécanisme capable d'ordonnancer les tâches apériodiques durant l'exécution, mais il faut se rendre à l'évidence que les tâches apériodiques sont rares dans un système temps réel.

Ainsi, le mécanisme est capable de résoudre les problèmes d'ordonnancement aussi bien quand le modèle de tâches est statique ou dynamique. Les problèmes d'ordonnancement de tâches se présentent au niveau monoprocesseur et au niveau multiprocesseurs puisque nous nous intéressons à des systèmes distribués.

On distingue de la sorte les aspects *local* et *distribué* traités par le mécanisme d'ordonnancement distribué que nous proposons. Le mécanisme *d'ordonnancement distribué* offre les fonctionnalités suivantes :

• L'allocation des tâches aux processeurs avant le début de l'exécution. Les tâches ayant la caractéristique temps réel, l'ordonnancement de ces dernières doit être pris en compte dans la phase d'allocation. En effet, une allocation correcte est définie par l'obtention d'ensembles de tâches ordonnançables sur chacun des processeurs.

• L'ordonnancement des tâches créées durant l'exécution.

• L'allocation durant l'exécution, des tâches ne pouvant être ordonnancées sur un processeur vers d'autres processeurs capables de garantir leurs contraintes temporelles.

Le mécanisme est intégré à un système d'exploitation parallèle Paros qui est supporté par le noyau ParX, développé par notre équipe de recherche. Nous avons commencé par reprendre et modifier les spécifications du modèle de processus de ParX afin de l'adapter au problème temps réel.

Pour chacune des parties présentées dans cette thèse nous avons proposé une contribution au problème traité. Dans la partie traitant l'ordonnancement local, nous avons présenté divers algorithmes d'ordonnancement local. Chaque algorithme est approprié à un modèle de tâche donné. Nous avons proposé et mis en œuvre un algorithme d'ordonnancement de tâches en-ligne. Le modèle de tâches visé est réaliste dans la mesure où les tâches peuvent être périodiques ou apériodiques, et peuvent communiquer entre elles. Des résultats préliminaires sont présentés. Les délais rajoutés dans un modèle d'ordonnancement dynamique sont dus au fait que les tâches préalablement acceptées sont souvent réordonnancées à chaque création de tâche. L'algorithme fournit des temps satisfaisants même dans les cas de figure où le processeur est en présence d'un nombre important de tâches à ordonnancer.

Dans la partie concernant l'allocation statique des tâches temps réel, nous avons mis en évidence la nécessité d'une approche qui considère à la fois les problèmes d'allocation et d'ordonnancement. L'algorithme génétique proposé est parallèle . Il est caractérisé par un codage des individus spécifique au problème, qui permet de manipuler directement l'ordonnancement obtenu sur chaque processeur. L'algorithme est appliqué à un modèle de tâches communiquantes et soumises à des contraintes de duplication.

Les algorithmes génétiques étant itératifs, ils sont une alternative attrayante pour réaliser à la fois l'allocation et l'ordonnancement. Cependant une convergence rapide est conditionnée par la manière dont le codage est fait et par la définition de nouveaux opérateurs afin de guider l'algorithme dans sa construction d'allocations. Les contraintes temporelles qui portent sur les tâches d'une application temps réel font que les opérateurs génétiques standards ne peuvent conduire rapidement à une solution.

Les résultats montrent que dès les premières itérations, l'algorithme réduit le nombre de tâches dépassant leurs échéances sur chaque processeur. L'algorithme trouve

des allocations correctes rapidement. Une comparaison de notre algorithme avec celui du recuit simulé a mis en évidence un net avantage du notre. Le coût d'exécution est plus réduit ainsi que le nombre d'itérations. Nous envisageons de comparer notre algorithme à des algorithmes itératifs de recherche locale.

La dernière partie de cette thèse s'est intéressée à l'allocation dynamique des tâches temps réel. De nombreux algorithmes ont été proposés dans la littérature pour l'allocation de tâches non temps réel. Certains algorithmes appliquent l'équilibrage de charge pour résoudre le problème de l'allocation. Ces approches, nous paraissent risquées pour des systèmes temps réels, d'autant plus que la charge du processeur n'est pas définie par son taux d'utilisation comme c'est souvent le cas dans les systèmes temps réel mais par le nombre de tâches sur le processeur. Toutefois, il serait intéressant de les implémenter afin de les comparer sous différents états de charge à des approches qui prennent davantage en compte la notion du temps.

Nous avons proposé un mécanisme d'allocation dynamique de tâches temps réel. Il est caractérisé par des politiques de maintien de l'état du système et de transfert des tâches assez simples. Un premier module information gère l'état du système, il applique un algorithme peu coûteux basé sur le voisinage pour la collecte des informations. Un second module décision applique une heuristique pour l'allocation des tâches. Elle se distingue par un traitement particulier pour les tâches apériodiques urgentes.

L'heuristique vise à donner davantage de garantie à ces tâches qu'aux tâches périodiques ordonnancées de manière statique. Face à une tâche apériodique refusée, elle procède par un retrait d'une tâche moins urgente auparavant garantie pour laquelle l'algorithme d'allocation est appliqué, et par l'acceptation de la tâche apériodique. Afin que la garantie délivrée par le mécanisme d'ordonnancement local puisse être assurée, des dispositions de réservation de l'emplacement de la tâche au niveau du processeur choisi, pour la tâche moins urgente sont mis en place.

La mise en œuvre du mécanisme a permis la validation des algorithmes proposés et l'estimation du délai moyen pour l'allocation. Ce délai permet de choisir quand il est nécessaire d'appliquer l'heuristique d'allocation proposée. Les résultats obtenus montrent que des tâches dont le temps de latence n'est pas court peuvent être alloués avec ce mécanisme et être garanties. Dans les situations de charge moyenne, où le temps d'interarrivée entre les tâches est réduit, le mécanisme donne de bons résultats; le

taux d'acceptation des tâches dans ce cas est d'environ 80 %. Ce taux peut être augmenté dans le cas d'utilisation de coprocesseurs sur lesquels seront implémentés les mécanismes d'ordonnancement et d'allocation. Le processeur dans ce cas, aura davantage de temps libre pour l'acceptation et l'exécution des tâches.

Les coûts introduits par le mécanisme sont non négligeables et nous sommes amenés à regretter l'avantage qu'on pourrait acquérir quand des protocoles de communication temps réel sont utilisés. En effet, la prise en compte de la priorité due au temps se fera à un bas niveau, et permettra de réduire ces coûts. Nous envisageons d'exploiter le routage offert par ParX. Durant l'envoi d'une tâche refusée à un processeur parallèlement à l'envoi de l'appel d'offres, nous proposons d'essayer d'ordonnancer la tâche sur les processeurs intermédiaires du chemin allant du processeur sur lequel la tâche a été refusée au processeur choisi par l'algorithme d'allocation. Cette solution, nécessite la modification des primitives de communication de ParX, afin qu'un message soit dédoublé avant son acheminement à un autre processeur. La copie du message pourra être prise en compte par l'allocateur pendant que le message est acheminé vers le processeur choisi.

Du travail reste à faire dans ce domaine afin que le mécanisme d'ordonnancement distribué soit efficace. Il serait intéressant d'étudier les mécanismes de synchronisation et de gestion du temps, en effet la dérive des processeurs dans une machine parallèle peut être importante. Une horloge globale est nécessaire pour l'allocation dynamique des tâches, aussi bien afin d'assurer la pertinence des surplus échangés entre les processeurs, mais également afin d'assurer le respect des contraintes sur les tâches.

Toujours dans le but de réduire les coûts introduits par le mécanisme, il serait intéressant de voir quelles fonctionnalités pourraient être réalisées par le matériel. D'un autre côté, en virtualisant la machine, le mécanisme devient indépendant du processeur et la suppression des interfaces implémentées pour combler le manque de fonctionnalités de base permet de réduire les coûts d'ordonnancement.

Références bibliographiques

[AdBo90a] J-M. Adamo, J. Bonneville, "**Manipulation des listes d'ordonnancement de processus sur Transputer : une technique sure**", La lettre du Transp. p5-15, Juin 1990.

[AdBo90b] J.M. Adamo, J. Bonneville, " **Scheduling and communication on the transputer: mechanisms and their accurate specification**", RT 90-10, LIP, Fevrier 1990.

[AHLR89] F. Armand, F. Herrmann, J. Lipkis, M. Rozier, " **Multi-threaded processes in CHORUS-MIX** ", Rapport technique Chorus systèmes, Octobre 1989.

[Alain92] J-M Alain, " **Real-Time scheduling of periodic tasks**", RT INRIA-Sophia Antipolis, Octobre 1992.

[AlDe92] M. Alabau, T. Dechaize, " **Ordonnancement temps réel par échéance**", Techniques et Sciences Informatiques, Vol. 11, n° 3 p59-123, 1992.

[AuBu90] N. Audsley, A.Burns, " **Real time system scheduling**", RR YCS 134 Univ. de York GB, Janvier 1990.

[Bacc91] L. Baccouche, " **techniques de mise en ïuvre d'une simulation à événements discrets sur machines multiprocesseurs** ", Rapport DEA, Labo. Modélisa. Calc., Juin 91.

[BDWe86] J. Blazewicz, M. Drabowski, J. Weglarz, "**Scheduling multiprocessor tasks to minimize schedule length**", IEEE Trans. on Comp. Vol C-35 n°5 p389-393, Mai 86.

[Berr89] R. Berry, " **Real time programming : Special purpose or general purpose languages**", Proc. IFIP 89 world Comp. Congress, San francisco.

[Blak92] B.A. Blake, " **Assignment of independent tasks to minimize completion time**", Soft. Pract. and exper., Vol 22(9) p723-734, Septembre 1992.

[Blaz79] J. Blazewicz, " **Deadline scheduling of tasks with ready times and resource constraints**", Inform. Proc. Letter, Vol n° 8 p60-63, Février 1979.

[BlSc91] Ben A. Blake, K. Schwan, " **Experimental evaluation of a real-time scheduler for a multiprocessor system**", IEEE trans. on soft. eng. Vol 17 p34-44, Janvier 91.

[BMTu94] L. Baccouche, T. Muntean, E-G. Talbi, " **Ordonnancement dynamique de tâches sous contraintes temporelles dans les systèmes parallèles**", Actes des 6èmes Rencontres francophones du parallélisme p458-462, ENS Lyon, Juin 1994.

[BaMu95] L. Baccouche, T. Muntean, " **Efficient static allocation of real-time tasks using genetic algorithms**", A paraître dans 14th workshop of the UK planning and scheduling SIG, Colchester, Novembre 1995.

[BNTZ93] A. Burns, M. Nicholson, K. Tindell, N. Zhang " **Allocating and scheduling hard real-time tasks on a point-to-point distributed system**", Proc. of the workshop on Par. and Dist. real-time Syst, p346-360, Avril 93.

[BoMi88] S.W. Bollinger, S.F. Midkiff, " **Processor and link assignment in multicomputers using simulated annealing**", Proceedings of the 11th int. conf. on para. proc., The Penn. State Univ. Press, Août 1988.

[BrFi81] R. M. Bryant, R. A. Finkel, " **A stable distributed scheduling algorithm**", IEEE computers p314-323, 1981.

[Brow84] F. Browaeys & al., "**SPECTRE : Proposition de noyau normalisé pour les exécutifs temps réel**", TSI 1984.

[CaDo94] B. M. Carlson, L. W. Dowdy, " **Static processor allocation in a soft real-time multiprocessor environment**", IEEE trans. on Par. and Distr. Syst. p316-320, Mars 1994.

[CaKu88] T. L. Casavant, J. G. Kuhl, " **A taxonomy of scheduling in general-purpose distributed computing systems**", IEEE trans. on Soft. Engi., Vol 2 Février 88.

[CaMa93] C. Cardeira, Z. Mammeri, "**Ordonnancement de tâches temps-réel par utilisation de réseaux de neurones**", RI Crin 93-R-281, Centre de Rech. en Infor. de Nancy, Septembre 1993.

[ChAg94] S-T. Cheng, A. K. Agrawala, "**Allocation and scheduling of real-time periodic tasks with relative timing constraints**", Tech. Report University of Maryland 1994.

[ChCh87] H. Chetto, M. Chetto, " **How to insure feasability in distributed systems for real-time control**", Int. Symp. on High-performance Comp. Sys. Décembre 1987.

[ChCh89] H. Chetto, M. Chetto, " **Some results of the Earliest Deadline Scheduling Algorithm** ", IEEE trans. on Soft. Eng. Vol 15 p1261-1269, Octobre 1989.

[ChLa87] W.W Chu, L.M Lan, " **Task allocation and precedence relations for distributed real-time systems**", IEEE trans. on Comp. p667-679, Juin 1987.

[ChLi86] H. Y. Chang, M. Livny, "**Distributed scheduling under deadline constraints : a comparison of sender-initiated and receiver initiated approaches**", Proc. IEEE Int. Conf. on Dis. Comp. Sys. 1986.

[ChMu91] K. Chen, P. Muhlethaler, " **A family of scheduling algorithms for real-time systems using time value functions**", RR Inria-Rocquencourt n° 1530 Sept embre 1991.

[Chor91] Chorus Kernel v3 r4.0" **Programmer's reference manual"**. Technical report CS/ TR-91-71, Chorus systèemes, 1991.

[Chun89] J-Y.Chung " **Scheduling hard real-time jobs that allow imprecise results**", These Univ. Illinois, 1989.

[ChYu90] G-H. Chen, J-S. Yur, " **A branch-and-bound with understimates algorithm for the task assignment problem with precedence constraint**", IEEE Computer p494-501, 1990.

[CJRe92] R.K. Clark, E.D. Jensen, F.D. Reynolds, " **An architectural overview of the Alpha Real-Time distributed Kernel**", Proc. USENIX Workshop on Microkernels and Other Kernel Architectures, Seattle, Avril 1992.

[CKMP90] O. Caprani, J. E. Kristensen, C. Mork, H. B. Pedersen, " **Implementation of real-time scheduling algorithms in a transputer environment**" , Real-time systems with transputers, H. Zedan p186-214, 90.

[Clar90] R.K. Clark, " **Scheduling dependent Real-Time activities**", thèse, Carnegie Melon Univ., Août 90.

[ClJR92] R.K. Clark, E.D. Jensen, F.D. Reynolds, " **An architectural overview of the Alpha Real-Time distributed Kernel**", Proc. USENIX Workshop on Microkernels and Other Kernel Architectures, Seattle, Avril 1992.

[CMMa87] M. F. Coulas, G. H. Macewen, G. Marquis, " **RNet : A hard real-time distributed programming system**", IEEE trans. on comp., Vol 8 Aout 1987.

[CSLa93] M.H. Cheung, K.M. Shea, F.C.M. Lau, " **Preemptive scheduling of multi-priority processes in transputer**", Transp. Applic. and Syst.'93, R. Grebe et al. IOS Press p877-889, 1993.

[CTGe93] T.G. Crainic, M. Toulouse, M. Gendreau, " **Towards a taxonomy of parallel tabu search algorithms**", CRT-933, centre de recherche sur les transports, Université de Montréal, 1993.

[DaDh86] S. Davari, S. K. Dhall, " **An on-line algorithm for real-time tasks allocation**", IEEE Computerp194-200, 1986.

[DeMo89] M.L Dertouzos, A. K-L. Mok, " **Multiprocessor on-line scheduling of hard real-time tasks**", IEEE Trans. on Softw. Engin., n°12 p1497-1506, Décembre 1989.

[Desn93] N. Desni, " **Generating random tasks with known optimal schedule for multiprocessing scheduling**", Master's project, Newark, NJ, 1993.

[DRSK88] A. Damm, J. Reisinger, W. Shwabl, H. Kopetz, " **The real-time operating system of MARS**", Institut für Technique Informatik, Austria, Octobre 1988.

[EaMa93] F. F. Easton, N. Mansour, "**A distributed genetic algorithm for employee staffing and scheduling problems**", Proc. of the 5th Intern. Conf. on Gene. Algo. San Mateo 1993.

[Eldv92] J.D. Eldvidge, " **PE second implementation scheduler**", DWP2-3, ESPRIT project 2528, THORN EMI, Mai 1992.

[Elle94] A. Elleuch, "**Migration de processus dans les systèmes massivement parallèles**", Thèse INPG, Novembre 1994.

[FGGR91] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, M. Roberts, " **Real-time Unix systems**", Kluwer academic publishers, Norwell, Massachusetts, 1991.

[FSWi91] J. Fiddler, E. Stromberg, D. Wilner, "**Software considerations for real-time risc**". The SPARC technical papers p305-313, 1991.

[Gait90] J. Gait, " **Scheduling and process migration in partitioned miltiprocessors**", Journ. of Par. and Dist. comp. p274-279, 1990.

[GAJo79] M.R. Garey, D.S. Johnson, " **Computers ans intractability : a guide to the theory of NP_completeness**", freeman, san francisco, 1979.

[GMSh94] K. Ghosh, B. Mukherjee, K. Schwan, " **A survey of real-time operating systems - Draft**", TR GIT-CC-93/18, Georgia Inst. of Techn., Février 1994.

[Gold89] D.E. Goldberg, " **Genetic algorithms in search, optimization and machine learning**", Addison-wesley, 1989.

[HaSh89] D. Haban, K. Shin, " **Application of real-time monitoring to scheduling tasks with random execution times**", 6th IEEE Workshop on Real-time Oper. Syst.and Soft. p1-27, Mai 1989.

[Herr90] R. G. Herrtwich, " **An introduction to real-time scheduling**", RT-90-035 Int. Comp. Science Instit. california, Juillet 1990.

[Holl75] J.H. Holland, "**Adaptation in natural and artificial systems**", Ann Arbor : Univ.of Michigan Press, 1975.

[HoSh94] CH. J. Hou, K.G. Shin, " **Replication and allocation of task modules in distributes real-time systems"**, 24th symposium on fault-tolerant computing p26-35, Austin Juin 1994.

[Hous87] C. Houstis, " **Allocation of Real-Time applications to distributed systems**", Proc. Int. Conf. in Paral. Processing p863-866, Août 1987.

[HRAn92] E. S. H. Hou, H. Ren, N. Ansari, " **Efficient multiprocessor scheduling based on genetic algorithms**", Dynamic, Genetic and chaotic programming, John Wiley & Sons,ê1992.

[HRAn94] E. S. H. Hou, H. Ren, N. Ansari, " **A genetic algorithm for multiprocessor scheduling**", IEEE trans. on par. and distr. syst. Vol 5 Février 1994.

[Huan85] J.P. Huang, " **Modeling of software partition for distributed real-time applications**", IEEE Trans. on Soft. Engin. n°10 p1113-1126, Octobre 1985.

[HZCa88] X.Huang, H. Zhang, X. Cai, " **Heuristic software partitioning algorithms for distributed Real-Time applications**", 10th Int. Conf. on Soft. Engin., Singapore, Avril 1988.

[JaMo86] F. Jahanian, A. K. L. Mok, "**Safety analysis of timing properties in real-time systems** ", IEEE Trans. on Soft. Eng. Sept embre 1986.

[KGVe83] S. Kirkpatrick, C.D. Gelatt, M.P. Vecci, " **Optimization by simulated annealing**", Science, Vol 220 p671-680, Mai 1983.

[Kidw93] M. D. Kidwell, " **Using genetic algorithms to schedule distributed tasks on a bus-based system**", Proc. of the 5th Intern. Conf. on Gene. Algo. San Mateo 1993.

[KKSh92] D. Kandlur, D. Kiskis, K. Shin, " **A real-time operating system for HARTS**", *mission critical systems*, ed. A. Agrawala, K. Gordon, P.Hwang, IOS Press, 1992.

[KlSt86] E. Kligerman, A.D. Stoyenko, "**Real-time Euclid : A language for reliable real-time systems**", IEEE Trans. on Soft. Engin. SE-12, Septembre 1987, p941-949.

[Kope89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, " **Distributed Fault-Tolerant Real-Time Systems : The Mars Approach**", IEEE Micro, Février 1989.

[LeLa92] G. Le Lann, " **Designing real-time dependable distributed systems**", Computer communications Vol 15 n°4 p44-51, Mai 1992.

[LeLa93] G. Le Lann, " **Proposition de taxonomie du probleme de l'ordonnancement Temps-Réel**", Dossier Temps-Réel, Tribunix n° 47, Janvier/Février 1993.

[LeYa86] D. W. Leinbaugh, M.-R. Yamini, " **Guaranteed response times in a distributed hard real-time environment**", IEEE trans. on Soft. Engi., Vol 12 p1139-1144, Décembre 1986.

[LiLa73] C. L. Liu, J. W. Layland, " **Scheduling algorithms for multiprogramming in a hard real-time environment**", Journ. of the Assoc. for Comp. Machinery., Vol 20 n° 1 p46-61, Janvier 1973.

[Lo88] V.M. Lo, " **Heuristics algorithms for task assignment in distributed systems**" Proc. Int. Conf. on Dist. Comp. p30-39, San Francisco, Novembre 1988.

[LRB77] J. K. Lenstra, A. H. G. Rinnooy, P. Brucker, " **Complexity of machine scheduling problems**", Ann. Discrete Math. 1977.

[LSDi87] J. P. Lehoczky, L. Sha, V. Ding, " **The Rate-monotonic scheduling algorithm : exact characterization and average case behavior**", TR, Departm. of Statistics, Univ. of Carnegie-mellon 1987.

[LSSt87] J. P. Lehoczky, L. Sha, J. K. Strosnider, " **Enhancing aperiodic responsiveness in hard real-time environment**", Proc. 8th IEEE Real-time Systs. Symp., San Jose, Calfornia, Décembre 1987.

[LTCA89] S. T. Levi, S. K. Tripathi, S. D. Carson, A. K. Agrawala, " **The MARUTI hard real-time operating system**", Oper. Syst. Review, Vol 23, Juillet 1989.

[Ma84] R. P-Y Ma, " **A model to solve timing critical application problems in distributed computer systems**", IEEE Computers p62-68, 1984.

[Mok83] A. K. Mok, " **Fundamental design problems of distributed systems for hard real-time environment**", PHD thesis, Lab. for Comp. Scien. (MIT), MIT/LCS/TR-297, 1983.

[MSGh93] B. Mukherjee, K. Schwan, K. Ghosh, " **A survey of real-time operating systems - Preliminary draft**", RT GIT-CC-93/18, College of Computing, Georgia Inst. of Techn. Mars 9193.

[PaCa91] O. Pasquier, J.P. Calvez, " **Utilisation du transputer pour les applications temps -réel**", La Lettre du Transputer p7-33, Juin 1991.

[PaDa93] F. Panzieri, R. Davoli, " **Real time systems : a tutorial**", RT UBLCS-93-22, Laboratory of Computer Science, Univ. of Bologna, Octobre 1993.

[PBRa90] M. Pilling, A. Burns, K. Raymond, " **Formal specification and proofs of inheritance protocols for real-time scheduling**", Softw. Engi. Journal, 5(5), 1990.

[PDPo93] F. Panzieri, L. Donatiello, L. Poretti, " **Scheduling real time tasks : a performance study**", RT UBLCS-93-10, Laboratory of Computer Science, Univ. of Bologna, Mai 93.

[PeSh89] D-T. Peng, K. G. Shin, " **Static allocation of periodic tasks with precedence constraints in distributed real-time systems**", IEEE Proc. of the 10th Int. Conf. on Distrib. Comp. p190-198, Juin 1989.

[PoRi93] S.C.S. POrto, C.C. Ribeiro, " **A tabu search approach to task scheduling on heterogeneous processors under precedence constraints**", Départ. d'infor. Université catholique de Rio de Janeiro, Mars 1993.

[PuKo89] P. Puschner, CH. Koza, " **Calculating the maximum execution time of Real-time programs**", Journ. of Real-Time Syst. Février 1989, p159-176.

[PyWa79] I. C. Pyle, I. C. Wand, " **Real-time programming languages for industrial and scientific progress control** ", Rapport technique University of York, Angleterre, Mai 1979.

[Rama90] K. Ramamritham, "**Allocation and scheduling of complex periodic tasks**", Proc. Int. Conf. on Dist. Comp. Syst. p108-115, Mai 1990.

[RaSt92] K. Ramamritham, J. A. Stankovic, " **Scheduling algorithms and operating systems support for real-time systems**", RR Dep. of Comp. Science Univ. of Massachusetts, Amherst, MA 01003, 1992.

[RSSh89] K. Ramamritham, J.A. Stankovic, P-F. Shiah," **O(n) scheduling algorithms for real-time multiprocessor systems**", 1989 International Conference on Parallel Processing Vol III, p143-152.

[RSSh90] K. Ramamritham, J.A. Stankovic, P-F. Shiah," **Efficient scheduling algorithms for real-time multiprocessor systems**", IEEE trans. on Par. and Dist. Syst. Vol 2 p184-194, Avril 1990.

[RSZh89] K. Ramamritham, J. A. Stankovic, W. Zhao, " **Distributed schedulings of tasks with deadlines and resource requirements**", IEEE trans. on Comp. Vol 8, p1110-1123, Août 1989.

[SBWT87] K. Schwan, T. Bihari, B. W. Weide, G. Taulbee, " **High performance operating systems primitives for robotics and real-time control systems**", ACM trans. on comp. syst. Vol 5, Aout 87, pp 189-231.

[SCLa92] K.M. Shea, M.H. Cheung, F.C.M. Lau, "**An efficient multi-priority scheduler for the transputer**", Proceedings of the 15th World Occam and Transp. User Group Techn. Meeting, p139-152, Mars 1993.

[ScZh92] K. Schwan, H. Zhou, " **Dynamic scheduling of hard Real-Time tasks and Real-Time threads**", IEEE Trans. on Soft. Engi., Vol 18, p736-748, Août 1992.

[Sevc93] K.C. Sevcik, " **Application scheduling and processor allocation in multiprogrammed parallel processing systems**", RT CSRI-282, Univ. of Toronto, Computer System Research Institute, Mars 93.

[SGBo87] K. Schwan, P. Gopinath, W. Bo, " **CHAOS- Kernel support for objects in the real-time domain**", IEEE Trans. on Comp. Vol c-36, p904-916, Août 1987.

[SGRa88] L. Sha, J. B. Goodenough, T. Ralya, " **An analytic approach to real-time software engineering**", Softw. Engin. Inst. Draft Report, 1988.

[ShCh89] K. G. Shin, Y-C. Chang, " **Load sharing in distributed real-time systems with broadcast of state changes**", IEEE Trans. On Comp. Vol 38 n° 8, Août 1989.

[ShGa90] T. Shepard, M. Gagne, " **A model of the F18 mission computer software for Pre-Run-Time scheduling**", Proceed. 10th IEEE Int. Conf. on Distr. Comp. Syst. p62-69, Mai 1990.

[ShGo87] L. Sha, J.B. Goodenough, " **Real-time scheduling theory and Ada**", IEEE computer, Avril 1987.

[ShSa93] L. Sha, S. S. Sathaye, " **Distributed real-time system design : Theoretical concepts and aplications**", RT CMU/SEI-91-TR-2, Soft. Engin. Inst Carnegie Mellon Univ. Mars 1993.

[ShSi91] N. G. Shivaratri, M. Singhal, " **A transfer policy for global scheduling algorithms to schedule tasks with deadlines**", IEEE Int. Conf. on Distr. Comp. Syst. 248-2551991.

[Sill86] M. Silly, " **La tolérance aux fautes dans un système temps réel à contraintes strictes**", RR 512, INRIA, Mars 1986.

[SKGo91] L. Sha, M.H. Klein, J.B. Goodenough, " **Rate monotonic analysis for real-time systems**", RT CMU/SEI-91-TR-6, Soft. Engin. Inst Carnegie Mellon Univ. Mars 1991.

[SLSh88] B. Sprunt, J. P. Lehoczky, L. Sha, " **Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm**", Proc. 9th IEEE Real-time Syst. Symp., p251-258, Hunstville, Décembre 1988.

[SRGr89] W. Schwabl, J. Reisinger, G. Grünsteidl, " **A survey of MARS**", Tech. report, ICSC, Octobre 1989.

[SRLe90] L. Sha, R. Rajkumar, J. P. Lehoczky, " **Priority inheritance protocols : an approach to real-time synchronization**", IEEE Trans. On Comp. Vol 39 n° 9 Septembre 1990.

[SSLe89] B. Sprunt, L. Sha, J. P. Lehoczky, " **Aperiodic task scheduling for hard real-time systems**", Journal of real-time syst. 1989.

[Stan85] J. A. Stankovic," **Stability and distributed scheduling algorithms**", IEEE transactions on software engineering, Vol 10 p1141-1152, Oct 1985.

[Stan88] J. A. Stankovic," **Misconceptions about real-time computing**", IEEE Computer, Oct 1988.

[Stan92a] J. A. Stankovic," **Editorial: Real-Time kernel interfaces**", Multi-dimensional systems and signal processing, Kluwer Academic Publishers p1-13, Boston, 1992.

[Stan92b] J. A. Stankovic, " **Distributed real-time computing : the next generation**", tech. Rept. University of Massachussets Amherst janvier 1992.

[StKh91] D. B. Stewart, P. K. Khosla, " **Real-time scheduling of sensor-based control systems**", 8th IEEE Work. on real-time OPer. Sys. and Softw. Mai 1991.

[Stoy87] A.D. Stoyenko, " **A real-time language with a schedulability analyzer**", TR CSRI-206, Comp. Syst. Resear. Instit. Université de toronto, Décembre 1987.

[StRa89] J. A. Stankovic, K. Ramamritham, " **The SPRING kernel : A new paradigm for Real-Time operating systems**", Oper. Syst. Review, Vol 23, Juillet 1989.

[SZGh91] K. Schwan, H. Zhou, A. Geith, " **Real-time threads**", ACM Oper. Syst. Rewiew, Vol 25, p35-46, Octobre 1991.

[Talb91] E-G. Talbi, " **Un algorithme d'allocation dynamique de processus sur un réseau de transputers**", La lettre du Transputer, p7-20, Septembre 1991.

[Talb93] E-G Talbi, " **Allocation de processus sur les architectures paralèles à mémoire distribuée**", thèse de l'INPG, grenoble, Mai 1993.

[TaMu91] E-G. Talbi, T. Muntean, " **Méthodes de placement statique de processus sur architectures parallèles**", Techn. et Scienc. Infor. TSI Vol 10 n° 5, Nov 1991.

[TaMu92] E-G. Talbi, T. Muntean, " **Evaluation et étude comparative d'algorithmes d'optimisation combinatoire : application au problème de placement de processus**", Rapport de Recherche RR-886-I, LGI/IMAG, Avril 1992.

[TBW92] K. Tindell, A. Burns, A. Wellings " **Allocating hard real-time tasks : (An NP-hard problem made easy)**", Journal of Real-time systems, 1992.

[Tind93] K. Tindell, " **Fixed priority scheduling of hard real-time systems**", PHD thesis Univ. of York, Comp. Science Depart. 1993.

[TNRa90] H. Tokuda, T. Nakajima, P. Rao, " **Real-time Mach : Towards a predictable real-time system**". Proc Usenix Mach Workshop, Octobre 1990.

[ToMe89] H. Tokuda, C. W. Mercer, " **ARTS : A distributed Real-Time kernel**", Oper. syst. review, Vol 23 (3), p29-53, juillet 1989.

[VeTh90] E. Verhulst, H. Thielemans, " **Predictable response times and portable hard real-time systems with TRANS-RTXc on the transputer**", Real-time Syst. with Transp. H. Zedan, Ed. 1990, p232-240, IOS Press.

[WaMo85] Y. T. Wang, J. T. Morris,, "**Load sharing in distributed systems**", IEEE Trans. on Comp. Vol C-34, Mars 1985.

[Welc90] P.H. Welch, " **Multi-priority scheduling for transputer-based real-time control**", Real-time Syst. with Transp.1990, IOS Press.

[XHXi88] H. Xin, Z. Hong, C. Xiyao, " **Heuristic software partitioning algorithms for distributed real-time applications**", IEEE trans. on Comp., p116-121, 1988.

[XuPa90] J. Xu, D. L. Parnas, " **Scheduling processes with release times, deadlines, precedence and exclusion relations**", IEEE trans. on Soft. Eng. Vol 3, p360-369, Mars 1990.

[ZhRa87] W. Zhao, K. Ramamritham, " **Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints**", Journal of Syst. ans Softw, p195-25, 1987.

[ZRST87a] W. Zhao, K. Ramamritham, J. A. Stankovic, "**Scheduling tasks with resource requirements in hard real-time systems**", IEEE Trans. on Softw. Eng., p564-576, Mai 1987.

[ZRSt87b] W. Zhao, K. Ramamritham, J. A. Stankovic, " **Preemptive scheduling under time and resource constraints**", IEEE trans. on comp.,Vol 8, p949-960, Août 87.