Contribution à l'algorithmique distribuée : arbres et ordonnancement

Franck Butelle

N° d'ordre :

Mémoire d'Habilitation à Diriger des Recherches

présentée à
L'UNIVERSITÉ PARIS 13

spécialité :
INFORMATIQUE

par

Franck BUTELLE
Laboratoire d'Informatique de Paris Nord

Sujet :
Contribution à l'algorithmique distribuée : arbres et ordonnancement

Présentée le 17 décembre 2007 devant le jury composé de :

MM.	**Yves**	**MÉTIVIER**	Rapporteur
	Jean-Frédéric	**MYOUPO**	Rapporteur
	Nicola	**SANTORO**	Rapporteur
	Ivan	**LAVALLÉE**	Examinateur
	Christian	**LAVAULT**	Examinateur
	Gérard	**PLATEAU**	Examinateur

Remerciements

Je tiens à remercier les membres du jury pour l'intérêt qu'ils ont manifesté à ce travail. Je suis très reconnaissant envers Messieurs les Professeurs Jean-Frédéric Myoupo, Yves Métivier et Nicola Santoro d'avoir accepté d'être rapporteurs.

Les Professeurs Ivan Lavallée de l'EPHE et Christian Lavault de l'Université Paris Nord, ont toute ma sympathie et mon amitié, ils ont été parmi les premiers à me pousser à me lancer d'abord dans la recherche puis maintenant dans ce travail.

J'ai été l'élève de maîtrise du Professeur Gérard Plateau, il a, depuis, toujours été un ami et un homme de bon conseil. Je le salue encore une fois très respectueusement.

***

J'aimerai saluer l'ensemble du LIPN, je ne peux citer ici tous ses membres, mais l'ambiance de recherche y est vraiment chaleureuse et c'est certainement grâce à ces directeurs, Jaqueline Vauzeilles puis Christophe Fouqueré pour ceux que j'ai bien connu.

Parmi les personnes que je voudrais aussi remercier, j'ai une pensée particulière à ceux que j'ai encadré, depuis les étudiants de DUT jusqu'à des étudiants en troisième cycle. En particulier, Mlle Lélia Blin et M. Mourad Hakem dont j'ai eu le plaisir de co-encadrer les thèses. Depuis ce sont devenus des amis et je continue à avoir de fructueux échanges avec eux.

Résumé

Nous présentons dans ce mémoire de thèse d'habilitation une étude sur des algorithmes distribués asynchrones de contrôle et d'ordonnancement.

Un algorithme de contrôle établit une structure virtuelle sur un réseau de sites communicants. Nous faisons le choix de faire un minimum d'hypothèses sur les connaissances de chaque site. De même, nous évitons autant que possible d'utiliser des mécanismes conduisant à des attentes qui peuvent être pénalisantes comme, par exemple, l'utilisation de synchroniseurs. Ces choix conduisent à privilégier les modes de fonctionnement essentiellement locaux. Nous introduisons toutefois une limite à cette démarche, dans ce travail, nous ne considérons que des algorithmes déterministes.

Dans ces circonstances, un problème essentiel de l'algorithmique distribuée est l'établissement d'une structure de contrôle couvrant la totalité du réseau, dans laquelle chaque site distingue certains de ses voisins de façon spécifique.

Après avoir rappelé des notions fondamentales en partie I, nous présentons dans la première partie, trois de nos algorithmes de construction d'arbre couvrant avec contraintes, ces dernières apportant une plus grande efficacité à la structure de contrôle établie. En particulier, nous considérons la contrainte de poids total minimum qui caractérise plutôt une recherche économique, celle de diamètre minimum qui concerne l'efficacité à la fois en temps mais aussi évidemment en messages et la contrainte de degré minimal qui permet par exemple d'utiliser des équipements d'interconnection moins coûteux.

Dans la troisième partie nous présentons deux de nos heuristiques pour la résolution du problème de l'ordonnancement distribué en ligne, avec arrivées sporadiques, d'abord de tâches indépendantes puis de tâches avec dépendances non cycliques. Nous montrons que là encore, la structure d'arbre peut être utilisée de façon bénéfique. En particulier, dans des réseaux de taille arbitrairement grande, des arbres de plus courts chemins limités aux voisins relativement proches peuvent être utilisés pour définir un concept nouveau et prometteur : la Sphère de Calcul. Cette Sphère de Calcul limite le nombre de messages échangés et le temps de calcul.

Tout au long de ce mémoire nous présentons des algorithmes nouveaux, voire pionniers dans leurs domaine. De nombreux développements sont possibles, certains déjà réalisés par nous-même ou par d'autres auteurs, d'autres sont des problèmes ouverts (recherche d'algorithmes optimaux par exemple).

Table des matières

Introduction générale

Des ordinateurs qualifiés d'hyper-parallèles sont déjà apparus sur le marché et des réseaux internationaux existent. Mais, même si comme M. Daniel Hillis, « père » de la connection machine, nombreux sont ceux qui pensaient qu'en multipliant le nombre de processeurs on pourrait multiplier la puissance de calcul, le développement de logiciels adéquats s'avère être une tâche difficile. Le rêve d'une puissance démesurée se dissout face à des problèmes de « grande taille ». Il s'agit de comprendre, maîtriser et diriger les communications entre des milliers de processeurs. Même si, dans certaines applications (comme par exemple le traitement d'image), des résultats probants ont été obtenus, le problème reste très difficile pour la plupart des autres applications et conduit les chercheurs, du mathématicien au programmeur, à de nouveaux modes de pensée.

En effet, depuis l'enfance, l'apprentissage ainsi que la résolution de problèmes sont habituellement décomposés en petites tâches exécutées en séquence, les unes après les autres. Cette technique ne peut pas, en général, être conservée dans l'utilisation de l'hyper-parallélisme ou parallélisme massif.

Un secteur actif de la recherche s'intéresse de près aux algorithmes distribués. Ces algorithmes sont, par définition, des algorithmes destinés à fonctionner en milieu distribué, c'est à dire sur des machines disposant de processeurs sans mémoire partagée, ou encore sur des réseaux de sites communicants. Ces environnements peuvent être caractérisés par un graphe $G = (V, E)$, où V est l'ensemble des sommets représentants les sites et E est l'ensemble des arêtes représentant les liens de communication entre ces sites. Un site étant soit un processeur, soit plusieurs, voire un réseau de communication local entre machines distinctes. La seule contrainte, et qui d'ailleurs s'avère généralement vérifiée dans la pratique, est l'existence d'une liaison unique entre le réseau (principal) et le site, par arête du graphe des communications.

Ce mémoire présente un bref tour d'horizon de plusieurs années de recherche sur des réseaux de sites sans mémoire, ni variables, ni horloge globale. Les sites travaillent en parallèle et de façon asynchrone [1].

1. Pour pouvoir considérer la complexité temporelle de ces algorithmes, on utilise un modèle

Toute action d'un site est conditionnée par la réception de message et de ceux précédemment reçus (on parle alors d'algorithme « message-driven »). Par souci d'homogénéité dans la rédaction de ce mémoire, nous ne présentons que des algorithmes déterministes, pour lesquels les sites disposent d'identités distinctes bien que nous ayons pu relâcher cette contrainte dans un de nos travaux.

L'un des problèmes majeurs des réseaux de sites sans mémoire partagée, est celui de sa structuration virtuelle. Nous avons cherché à établir des algorithmes utilisant les hypothèses les plus faibles possible au sens mathématique du terme. C'est-à-dire que ces algorithmes sont, en grande partie, indépendants du matériel. Ils sont asynchrones (il n'y a pas nécessité d'existence d'une synchronicité totale) et ne présupposent pas de connaissance sur la topologie du réseau (à part peut-être sa connexité) : que ce soit sur sa taille, son diamètre ou encore son degré. Nous ne formulons pas non plus d'hypothèse sur l'homogénéité du réseau : les machines qui le composent peuvent être de capacités et de puissance différentes, seul le protocole de communication doit être le même pour toute machine. De plus, nous n'avons pas besoin de l'existence d'un site ayant une particularité connue de tous les autres ainsi que de lui-même (parfois appelé initiateur [2]). Nous ne formulons pas non plus, en général, de restriction sur le nombre de sites qui démarrent les algorithmes distribués considérés. Ils forment un sous-ensemble quelconque non vide de l'ensemble des sites.

La justification de ces choix tient en ce que l'augmentation prévisible de la taille des réseaux et des machines distribuées, la combinaison des moyens physiques de connexion (câbles coaxiaux, fibres optiques, ondes Hertziennes, etc.) conduit à des topologies physiques imprévisibles, voire même dynamiques. Nous présentons d'ailleurs un algorithme qui n'utilise que des ressources (en termes de sites) à voisinage limité, permettant un fonctionnement sur des graphes de taille arbitrairement grande.

La structuration virtuelle d'un réseau consiste en la construction de structures de contrôle sur la totalité des processus, brisant ainsi la symétrie d'origine.

Avec l'établissement de la structure virtuelle, chaque site acquiert des connaissances supplémentaires sur le réseau, connaissances liées à la structure de contrôle établie. C'est-à-dire que chaque site peut alors distinguer certains de ses voisins selon leurs rôles et/ou privilèges dans la structure.

Les structures généralement utilisées sont l'anneau virtuel ou l'arbre couvrant voir par exemple [Awer87, GaHS83, HeMR87, KoKM90, LaLa89a].

Un arbre couvrant (AC) du réseau permet la résolution de nombreux problèmes en algorithmique distribuée tels l'exclusion mutuelle [TrNa87] ou la synchronisation [Awer85c] ; il est lié aussi aux problèmes de calculs distribués de base

affaibli : le réseau est dit asynchrone à délais bornés (voir Partie I, chapitre 2)

2. On parle aussi de rupture de symétrie dans ce cas

tels la recherche d'extremum, l'élection et la terminaison distribuée. Nous avons rappelé comment ces problèmes se réduisent à la construction d'AC dans [Bute94].

Le mémoire s'articule de la façon suivante :

Partie I

Nous précisons, dans cette première partie, d'une part les diverses notations et définitions classiques de la théorie des graphes ainsi que quelques propriétés essentielles et d'autre part le modèle de système distribué qui nous intéresse.

Partie II

Cette deuxième partie décrit trois de nos algorithmes de construction d'arbre couvrant contraint. Ces contraintes apportent une plus grande efficacité à cette structure de contrôle. En particulier, l'accent sera mis sur les contraintes de poids total minimal, qui caractérise plutôt une recherche économique, celle de diamètre minimal, qui permet de minimiser les communications en moyenne, et celle de degré minimal, contraintes issues par exemple du choix d'équipements d'interconnexion à coût réduit ou encore de limiter des phénomènes de type « goulet d'étranglement ».

Partie III

Les applications modernes sont particulièrement coûteuses en temps de calcul, que ce soit la prédiction météorologique ou toute autre simulation. Ces besoins en calculs ont poussé les chercheurs à paralléliser les programmes et les données pour répartir la charge sur les moyens de calcul. Les applications parallèles (ou parallélisées) sont souvent représentées sous forme de tâches indépendantes ou en graphes de dépendances de tâches. Très généralement, l'ordonnancement et le placement des tâches composant ces applications parallèles sur une architecture matérielle n'est souvent envisagée que d'un point de vue centralisé (même dans les architectures et technologies de type grille). Dans un contexte réaliste, l'architecture matérielle n'est même pas forcément régulière ni de taille connue. Nous allons tenter de nous approcher le plus possible de cette situation en distribuant l'ordonnanceur : c'est le propos de la troisième et dernière partie de ce travail dans lequel nous présentons deux de nos heuristiques d'ordonnancement distribué en ligne. La première porte sur l'ordonnancement de tâches indépendantes à arrivées sporadiques sur des sites quelconques du réseau. La seconde traite le problème des tâches avec dépendances non cycliques, dans le même contexte.

Nous terminons ce mémoire par un bilan et une synthèse des perspectives qui s'en dégagent.

Première partie

Notations et généralités

Chapitre 1

Définitions et Notations

1.1 Introduction

Nous présentons dans cette partie quelques définitions et notations de la théorie des graphes pour éviter certaines ambiguïtés puis le modèle de système distribué asynchrone qui nous intéresse.

Les algorithmes distribués étant destinés à fonctionner sur des réseaux, ces réseaux sont associés à des graphes, les liens de communications étant les arêtes et les sites les nœuds du graphe. Nous allons reprendre des propriétés essentielles de la théorie des graphes et préciser celles qui nous intéressent. Nous discutons en particulier de la notion de distance et de ses dérivés.

Ensuite, nous présentons le modèle général de système distribué asynchrone, modèle repris par de nombreux auteurs avec certaines nuances. Nous précisons quelles sont les hypothèses précises qui forment le cadre de notre travail. Ces notions essentielles ajoutées à la terminaison et à la mesure de la complexité des algorithmes distribués sont évoqués et discutés.

1.2 Définitions et notations de la théorie des graphes

Nous utilisons dans ce mémoire les notations et définitions usuelles de la théorie des graphes (voir par exemple C. Berge [Berg70]) plus quelques autres (ou des définitions qui pourraient être ambiguës) que nous rappelons dans les paragraphes qui suivent ou avant leur utilisation.

Un *graphe simple* est un graphe dans lequel tout arc relie deux *sommets* distincts, et étant donné deux sommets de G, il existe au plus un arc les reliant. Tous les graphes considérés dans ce document sont des graphes simples.

Il existe deux terminologies distinctes suivant que le graphe est orienté ou non. Nous utilisons les deux terminologies indépendamment car nous ne considérons que des graphes non-orientés et il est évident que ses derniers sont des cas particuliers des graphes orientés et ainsi l'abus de langage qui s'en suit n'est pas très conséquent, du moins pour ce qui nous concerne.

Soit $G = (V, E)$ un graphe *non-orienté* (ou *symétrique*) tel que $|V| = n$ et $|E| = m$. V est l'ensemble des nœuds (*Vertices*) et E est l'ensemble des arêtes (ou arcs, *Edges*). Nous associons au graphe G, une fonction de coût ω (ou de poids, ou encore de délai de communication entre sites pour un réseau) de E vers $\mathbb{R}^+$ (dans la dernière partie de ce travail, nous définirons aussi une fonction de coût pour les nœuds).

Nous utilisons les notations communes suivantes :

$\boldsymbol{\Gamma_G(x)}$ Ensemble des *voisins* de x.

$\boldsymbol{\delta(x)}$ *Degré* d'un sommet x, égal ici à $|\Gamma_G(x)|$.

$\boldsymbol{[x_1, x_2, \ldots, x_p]} = (e_1, \ldots, e_{p-1})$ *Chemin* contenant les sommets x_1 à x_p, les arêtes $e_i = (x_i, x_{i+1})$, $(i \in \{1, \ldots p-1\})$ appartiennent toutes à E.

Nous ne considérons dans cette étude que des chemins *élémentaires*, c'est à dire ne comportant pas deux fois le même sommet (et donc pas deux fois la même arête) [1]. Un *cycle* (ou *circuit*) est un chemin dont les extrémités coïncident.

Notions de longueurs, distances, etc.

Nous étendons la définition de la fonction de « poids » ω aux ensembles E d'arêtes : $\omega(E)$ étant alors la somme des poids des arêtes composant cet ensemble, ainsi la *longueur* d'un chemin $\mu = [x_1, x_2, \ldots, x_p]$ est notée $\omega(\mu)$. Ceci introduit la notion de « longueur valuée » et influence directement les définitions de distance, rayon, diamètre etc., qui vont suivre. Il est à noter que tous les auteurs n'utilisent pas ces définitions qui se voient ici généralisées aux graphes valués. Nous étendons la définition de longueur sur les graphes non-valués en faisant l'amalgame graphe non-valué, graphe valué uniformément avec des poids égaux à 1. De cette façon, cette définition permet de conserver les définitions usuelles sur les graphes non valués où la distance entre deux sommets est égale au nombre d'arêtes d'un plus court chemin reliant ces sommets.

Cette notion de longueur permet d'introduire des définitions et notations généralisées, c'est à dire définies aussi bien sur des graphes valués que sur des graphes

1. Dans la terminologie des graphes non-orientés, il est en fait plus correct de parler de *chaîne*, mais nous gardons la terminologie chemin, et nous gardons l'idée de direction associée dans la mesure où des messages sont émis d'un site pour arriver à un autre.

non-valués, comme suit :

$d_G(x, y)$ *Distance* de x à y, égale à la longueur d'un plus court chemin de x à y. Cette longueur est choisie égale à $+\infty$ si un tel chemin n'existe pas et égale à 0 pour la distance d'un nœud à lui même : $d_G(x, x)$.

$ecc_G(v)$ *Écartement* (ou *excentricité*, ou *séparation*) d'un sommet v défini comme suit : $ecc_G(v) = \max_{k \in V} d_G(v, k)$. Tant que les graphes considérés sont connexes, l'excentricité est une valeur finie.

$D(G)$ *Diamètre* du graphe G. Il peut être défini par la longueur du plus long chemin ou par la plus grande excentricité : $D(G) = \max_{x,y \in V} d_G(x, y) = \max_{v \in V} ecc_G(v)$. Par extension, nous utilisons le terme de *chemin diamétral* de G pour désigner un plus court chemin entre deux nœuds de longueur égale à $D(G)$.

$\rho(G)$ *Rayon* du graphe. Le rayon d'un graphe est égal à l'excentricité minimale soit : $\rho(G) = \min_{v \in V} ecc_G(v)$.

Un *centre* v_0 du graphe est un sommet d'excentricité minimum, son excentricité $ecc_G(v_0)$, est donc égale au rayon du graphe. Si $\rho(G)$ est fini alors il existe au moins un centre.

$ACC_G(x)$ Ensemble des arbres couvrants de G de plus court chemin enracinés en x.

Nous utilisons aussi parfois la version orientée de l'arbre couvrant : une *arborescence couvrante* est définie comme un arbre couvrant muni d'une racine x, c'est à dire qu'à partir de x il existe un chemin vers tous les autres nœuds de l'arborescence. Il est à noter que cette notion n'est utilisée que d'un point de vue logique : c'est l'algorithme qui fournit une orientation aux arêtes, pas le réseau.

Dans la suite de ce travail, le G en indice est omis dès que cela n'entraîne pas d'ambiguïté.

1.3 Quelques propriétés utiles

Nous allons rappeler maintenant quelques propriétés de la théorie des graphes qui nous seront utiles dans la suite de cette thèse.

1.3.1 Propriétés des distances

– *Inégalité triangulaire* (ou Eulérienne) :

$$\forall i, j, k \in V,\ d(i, j) + d(j, k) \geq d(i, k)$$

– Un *sous-chemin* d'un plus court chemin est aussi un plus court chemin (Bellman) :

$$\forall \mu = [x_1, \ldots, x_p],\ x_i \in V,\ \forall i \in \{1, \ldots p\},\ \omega(\mu) = d(x_1, x_p)$$

$$\Rightarrow d(x_1, x_j) = \omega([x_1, x_j])\ \forall j \in \{1, \ldots, p\}$$

C'est sur l'exploitation de ces deux propriétés que sont basés tous les algorithmes de plus court chemin.

1.3.2 Propriétés sur les rayons et les diamètres

Dans ce paragraphe, nous allons exprimer quelques inégalités sur les diamètres et les rayons. Nous omettons la plupart des preuves car elles peuvent être retrouvées facilement dans la littérature sur la théorie des graphes.

Lemme 1 $\rho(G) \leq D(G) \leq 2\rho(G)$

Si T' est un *Arbre Couvrant de Rayon Minimal* sur G, alors

$$\rho(T') = \rho(G) \text{ et } D(T') \leq 2\rho(G)$$

Si T^* est un *Arbre Couvrant de Diamètre Minimal* sur G alors

$$D(G) \leq D(T^*) = D^* \leq D(T')$$

La borne inférieure est atteinte par exemple lorsque G est un arbre et la borne supérieure est atteinte lorsque G est un graphe complet.

Finalement, nous obtenons l'inégalité suivante :

$$\rho(G) \leq D(G) \leq D^* \leq D(T') \leq 2\rho(G) \tag{1.1}$$

Nous pouvons maintenant énoncer quelques propriétés sur les arbres valués.

Par définition, si $T \in ACC_G(x)$, nous avons :

$$D(T) = \max_{x,y \in V}(d_T(x, y)) \leq \max_{i,j \in V}(d_G(i, x) + d_G(x, j))$$

Lemme 2 *Un centre x de G est aussi un centre de tout arbre $T \in ACC_G(x)$.*

Si toutes les arêtes sont de poids 1, nous pouvons reprendre un très vieux résultat de Camille Jordan, daté de 1869, repris par C. Berge dans [Berg70], chapitre 4, théorème 8.

Théorème 1 *Si G est un arbre, et si $D(G)$ est pair, alors G a un centre unique, par lequel passent tous les chemins de longueur maximum ; en outre on a :*

$$\rho(G) = \frac{1}{2}D(G)$$

Si $D(G)$ est impair, alors G admet exactement deux centres, avec une arête qui les relie et par laquelle passent tous les chemins de longueur maximum ; en outre on a :

$$\rho(G) = \frac{1}{2}\left(D(G) + 1\right)$$

Par conséquent, avec nos notations, nous avons $\rho(T) = \left\lfloor \frac{D(T)+1}{2} \right\rfloor$ pour tout arbre T.

Dans le cas plus général où les arêtes sont de poids positif non nul nous obtenons le résultat suivant :

Lemme 3 *Tout arbre T a exactement un ou deux centres ; s'il a exactement un centre alors :*

$$D(T) = 2\rho(T)$$

Si il a exactement deux centres :

$$D(T) < 2\rho(T) < D(T) + 2\beta$$

ou β est le poids de l'unique arête joignant les deux centres.

Preuve : Si l'arbre T n'a qu'un centre, alors il existe au moins deux chemins distincts, de ce centre à deux feuilles de l'arbre, de poids identiques et maximum. Donc le rayon de l'arbre est ce poids et le diamètre est obtenu par la plus longue chaîne égale à $2\rho(T)$.

Si l'arbre a deux centres c_1 et c_2. Démontrons d'abord qu'il ne peut y avoir qu'une seule arête entre ces deux centres, puis nous démontrons l'inégalité prévue par le lemme. Supposons donc, par contradiction, qu'il existe un chemin de longueur minimale avec au moins deux arêtes entre c_1 et c_2 et donc un ou deux centres peuvent être trouves dans ce chemin en lieu et place de c_1 et c_2.

Soit β le poids de l'arête joignant c_1 et c_2, α la longueur du plus long chemin de c_1 a une feuille de son sous-arbre ne contenant pas c_2 et, de même, soit α' la longueur du plus long chemin de c_2 a une feuille de son sous-arbre (ne contenant pas c_1). Supposons par exemple que $\alpha \geq \alpha'$.

Nous obtenons alors la relation suivante :

$$D(T) = \alpha + \beta + \alpha' \text{ et } \rho(T) = \alpha + \beta$$

mais

$$\alpha' + \beta > \alpha \ \ (car\ \beta > 0)$$

donc

$$\rho(T) < \frac{\alpha + 3\beta + \alpha'}{2} = \frac{D(T)}{2} + \beta$$

et

$$\rho(T) > \frac{\alpha + \beta + \alpha'}{2} = \frac{D(T)}{2}$$

Il ne nous reste plus qu'à montrer qu'un arbre ne peut avoir plus de deux centres, c'est une démonstration par récurrence analogue à celle décrite par C. Berge dans [Berg70] pour démontrer le théorème de C. Jordan où les poids sont de 1 et les graphes sont orientés. □

Nous présentons dans le chapitre suivant le modèle du système distribué.

Chapitre 2

Le modèle du système distribué

Nous allons décrire maintenant, dans les paragraphes qui suivent, le cadre de notre travail à savoir ce que peut bien représenter un système distribué (ou réparti) et la notion d'algorithme dans ce système. Ces définitions sont très couramment employées et proviennent, entre autres, des articles et ouvrages de C. A. R. Hoare, G. Le Lann, I. Lavallée, C. Lavault et M. Raynal [Hoar87, LaLa86, Laval90, Lavau87, Lela77, Rayn87, Rayn91].

2.1 Le modèle

Le modèle standard de système distribué $\mathbb{M}$ considéré dans la suite est une structure logicielle et matérielle répartie sur un réseau *asynchrone point à point de processus séquentiels communicants*. Tout système distribué est composé d'un ensemble de *sites* reliés entre eux par un *système de communication*. Cette structure est modélisée par un graphe connexe, simple et symétrique.

Chaque site possède une mémoire locale non partagée de capacité bornée, et au moins un processeur. Les seules informations échangées entre les sites sont des messages véhiculés par le réseau. Nous admettrons aussi que chaque site possède un numéro d'identification unique le distinguant de tout autre : son *identité*. L'ensemble des identités dans $\mathbb{M}$ est muni d'un ordre total strict et donc les identités sont, dans ce modèle, bien distinctes.

Dans un but de généralisation, nous utilisons aussi bien la notion de site que de processeur, processus, ou encore nœud du graphe des canaux de communication. Un site étant soit un processeur, soit plusieurs, voire un réseau de communication local entre machines distinctes. La seule contrainte, et qui d'ailleurs s'avère généralement réalisée dans la pratique, est l'existence d'une liaison unique entre le réseau (principal) et le site, par arête du graphe. Ce site, s'il représente un

réseau local, devra distinguer un unique processeur, parmi ceux composant son réseau, pour les communications avec « le monde extérieur » *c'est à dire.* le réseau principal.

Nous devons insister sur le fait que dans un tel système distribué *asynchrone* $\mathbb{M}$, il n'y a pas de mémoire partagée ni d'horloge globale ce qui entraîne l'absence de toute *variable* ou *état global* accessible par les sites à un instant donné. Ce concept d'instant donné est inutilisable et même paradoxal au vu de la définition du modèle asynchrone. Les seuls événements perceptibles par un processus quelconque sont soit produits par le processus lui-même (envoi de message, calcul local) soit des événements générés par d'autres processus (réception de messages). Nous détaillons plus loin les hypothèses liées à l'asynchronisme de $\mathbb{M}$.

Les messages sont traités dans leur ordre d'arrivée. Si plusieurs messages sont reçus simultanément, ils sont traités dans un ordre arbitraire ou bien suivant l'ordre total sur l'identité des émetteurs de ces messages. Dans la suite nous utilisons indifféremment, et tant qu'il n'y a pas d'ambiguïté possible, les termes de *site*, *processus*, *processeurs* et *sommet*, pour désigner les composants de $\mathbb{M}$ et les *liens de communications* sont aussi appelés *lignes*, *arêtes* ou encore *arcs*. Cette synonymie est rendue possible dans la mesure ou ils se confondent dans l'analyse des algorithmes distribués. Usuellement, deux sommets quelconques reliés par une arête sont dits *voisins*.

Nous considérons essentiellement des réseaux de type *point à point*, c'est à dire vérifiant les trois caractéristiques suivantes :

- *Orientation locale* : tout processus est capable de faire la distinction entre ses *portes* d'entrée-sortie. Ceci ne présume en rien de la connaissance des identités des voisins qui sont *a priori* inconnues. Uniquement dans un but de simplification du code des algorithmes présentés, nous associons une porte d'Entrée/Sortie avec l'identité du voisin correspondant. Notons que la connaissance réelle des identités des voisins est une connaissance en quelque sorte globale, elle peut amener une grande diminution du nombre de messages nécessaires à la résolution de tous les problèmes de l'algorithmique distribuée. Notons encore que pour obtenir une telle information il faut que chaque arête soit parcourue deux fois donc conduit à l'échange de $2m$ messages.
- *Communication sans perte de message* : tout message envoyé à un voisin est reçu si ni la ligne, ni le récepteur ne sont fautifs (très précisément exempts de toute faute) durant la transmission du message.
  Rappelons que nous ne pouvons pas prévoir le délai de transmission du message, ainsi cette hypothèse ne fait que stipuler qu'au bout d'un temps fini, en l'absence de faute, tout message envoyé est correctement reçu.
- *Taille bornée des messages* : nous supposons que tout message parcourant le réseau G contient au plus t_G bits, où t_G est une constante prédéfinie du réseau. Plus précisément, cette taille est en $O(f(m, id))$ ou en $O(f(n, id))$

(voir $O(1)$ dans certains cas) où n est le nombre de sites et m le nombre de liens. id est la taille, en nombre de bits, de l'identité maximum d'un site de G.

2.2 L'algorithmique distribuée

Un *algorithme distribué* $\mathbb{A}$ sur un système distribué $\mathbb{M}$ est la suite des *transitions locales* à effectuer séquentiellement sur chaque processus p de $\mathbb{M}$ suivant la réception de tel ou tel message à partir d'un état déterminé de p. $\mathbb{A}$ peut être considéré comme une collection de processus qui, par échange d'informations, établissent un calcul ou plus généralement collaborent vers un but commun tout en conservant leur autonomie et leur indépendance. On peut alors définir précisément chaque processus par une *suite d'événements*, un ensemble fini d'états et un ensemble fini de *transitions atomiques* entre événements. Nous schématisons les transitions atomiques de la façon suivante :

$$\langle Q_i,\, E_j\rangle \longrightarrow \langle Q_{i+1},\, E_{j+1}\rangle$$

où Q_i est le i^{e} état du processus et E_j une j^{e} *action atomique* du processus. Nous classons les événements en trois catégories : les événement d'envoi, les événement de réception et les événements internes. Un *événement interne* ne produit qu'un changement d'état, un *événement d'envoi* provoque l'envoi d'un message asynchrone et un *événement de réception* donne lieu à la réception d'un message positionné alors dans un tampon. Si le tampon était vide alors cet événement est lié à un événement interne : la mise à jour de l'état local suivant le contenu du message.

Le concept d'algorithme distribué est donc clairement distinct de celui d'algorithme séquentiel et n'en est pas non plus une variante. Un algorithme distribué est fonction essentiellement de notions spécifiques aux environnements distribués tels la communication, la causalité entre événements, la connaissance acquise par apprentissage etc.

2.3 Démarrage d'un algorithme distribué

Le système $\mathbb{M}$ se comporte selon les règles suivantes :

– Un processus est toujours dans l'un des deux états fondamentaux : *actif* ou *passif*.

– Seuls des processus actifs peuvent émettre des messages.

– Un processus décide de passer de l'état actif à l'état passif suite à un événement interne.

– Un processus passe de l'état passif à l'état actif uniquement sur réception de message ou suite à un événement extérieur.

La dernière règle exprime la notion de démarrage : plus précisément, dans le modèle général, nous ne formons pas d'hypothèse particulière sur l'identité du ou des sites qui démarrent l'algorithme. De même nous ne fixons aucune connaissance sur le nombre de ces sites que l'on qualifie *d'initiateurs*. L'unique exigence est qu'au moins un démarre ; c'est à dire passe de l'état passif à l'état actif.

Il n'existe que deux conditions d'éveils d'un site, la réception d'un message ou un stimulus extérieur. Ce dernier représente une intervention de type mise en circuit ou ordre d'exécution que ce soit d'un utilisateur ou bien d'un mécanisme extérieur quelconque.

Nous devons toutefois accepter une autre hypothèse que l'on retrouve sous-entendue dans la quasi-totalité des algorithmes distribués connus sans pour autant qu'elle soit précisée systématiquement : le démarrage d'un site consiste à l'exécution de la première instruction de l'algorithme distribué. Cette hypothèse semble fort naturelle mais elle peut être discutable surtout lorsque l'on considère les problèmes de reprise sur panne. On peut consulter à ce sujet l'article de M. Fischer *et al.* [FMRT90].

Avec l'hypothèse précédente, il est clair que l'on pourra distinguer éveil spontané et éveil par réception de message par un simple test sur le tampon de réception.

2.4 Terminaison d'un algorithme distribué

Avec les règles précédentes, la *terminaison* d'un algorithme distribué peut survenir suivant deux schémas distincts exprimant tous deux la notion de stabilité globale. Le premier respecte le sens usuel : l'algorithme termine si tous les processus passent à l'état passif et restent passifs et savent que tous les autres finissent au bout d'un délai fini par devenir passifs – on utilise alors le terme de *terminaison par processus*. Cette terminaison est très contraignante et peut être difficile à obtenir.

La *terminaison par message* se caractérise par l'absence de communication — elle n'implique pas que l'information de fin d'exécution de l'algorithme soit connue de tous les processus. Ce dernier schéma de terminaison est celui de certains algorithmes de routage (voir par exemple A. Segall [Sega83]) : tant que le réseau reste stable, l'algorithme de routage ne provoque pas d'échange de messages de mise à jour, c'est à dire que l'état global du système est stable et reste stable — autrement dit il s'agit d'une terminaison par défaut de messages.

Dans une terminaison par processus, il est nécessaire qu'un site qui a terminé l'exécution de l'algorithme ne reçoive plus de messages ayant un rapport avec cet

algorithme. Dans le cas contraire, il pourrait alors être réveillé par ce message et ne pourrait déterminer si le message fait partie de cet algorithme ou d'une nouvelle exécution.

De nombreux travaux ont été réalisés autour du problème de la détection de la terminaison d'algorithmes distribués, comme le montre F. Mattern dans [Matt87] et la thèse de G. Tel [Tel91].

Nous pouvons maintenant résumer et préciser les hypothèses que nous ferons sur $\mathbb{M}$.

2.5 Hypothèses : résumé et précisions

1. Les communications sont asynchrones non bloquantes - Chaque processus dispose d'un tampon borné. (Il est possible d'utiliser un réseau de communication synchrone a condition de pouvoir exécuter plusieurs processus par nœud – voir par exemple la thèse de M. Bui [Bui89]).
2. Les liaisons sont bidirectionnelles (les messages peuvent se croiser sur une ligne).
3. Les délais de transmission sont finis (mais non bornés). Lorsque nous cherchons à calculer la complexité temporelle d'un algorithme, nous nous plaçons en fait dans un modèle asynchrone à délai borné.
4. Si des messages arrivent simultanément, ils sont pris en compte séquentiellement.
5. Il n'y a pas de panne de processus.
6. Il n'y a ni perte, ni duplication, ni modification de message.
7. Un sous-ensemble non vide de l'ensemble des processeurs s'éveille spontanément.
8. Les messages arrivent en séquence — les messages ne peuvent se doubler sur la ligne ; c'est la discipline PAPS : Premier Arrivé, Premier Servi dite aussi FIFO « First In, First Out » (on peut toujours se ramener à ce cas par l'ajout d'estampilles, voir l'article de L. Lamport [Lamp78]).
9. Les seules connaissances accessibles pour un processus sont son identité et les portes d'Entrée/Sortie (distinctes) qui mènent vers ses voisins (en fait, pour simplifier l'écriture des algorithmes, on ne fait pas dans ce travail de distinction entre la porte qui mène vers un voisin et l'identité du voisin. Cette simplification est possible car les algorithmes que nous présentons ont la particularité d'être insensibles à cette hypothèse).
10. Toutes les identités des processus sont distinctes.
11. Le graphe représentant le réseau des processus est connexe.

L'hypothèse 1 pose différents problèmes, en fait, la plupart des algorithmes asynchrones ne fixent pas de borne à la taille des tampons. Malgré tout, les algorithmes que nous allons présenter échangent généralement un nombre fini borné de messages, par conséquent une borne sur le nombre de messages échangés est aussi une borne pour la taille des tampons.

2.6 Évaluation de la complexité

Pour comparer deux algorithmes séquentiels, l'analyse s'appuie sur le calcul du nombre d'opérations élémentaires nécessaires à l'obtention du résultat en fonction de la taille des données. Les opérations élémentaires sont souvent les opérations arithmétiques, les opérations d'accès à la mémoire etc. En fait les mesures de complexité des algorithmes sont la mesure de quantité de ressources utilisées. Les notations utilisées dans ce mémoire sont les classiques notations de Landau voir par exemple [FrGS90].

En algorithmique distribuée, le concept de mesure de quantité de ressource utilisée est le même et l'on peut résumer comme suit les ressources utilisées par $\mathbb{A}$:

- le temps de calcul local en chaque processus,
- le temps de communication entre processus pour échanger l'information nécessaire à l'exécution des instructions globales de $\mathbb{A}$,
- les messages (ou le nombre de bits) échangés,
- l'information transmise de processus à processus, et l'information disponible sur l'état des processus le long de chaque ligne de communication et en chaque site,
- l'occupation en mémoire.

Dans un environnement distribué asynchrone, l'exécution d'un algorithme est non seulement dépendante de l'entrée comme dans un algorithme séquentiel mais en plus deux exécutions différentes sur les mêmes entrées pourront produire des résultats différents, donc utiliser des quantités de ressources différentes. Cette dernière remarque ajoutée à la caractérisation d'une nouvelle ressource essentielle : la transmission de l'information, explique la difficulté de l'analyse des algorithmes distribués en particulier asynchrones.

La complexité se mesure suivant deux paramètres essentiels : la complexité en *quantité d'information échangée* (voire en nombre de messages dans certains cas) et le temps total d'exécution de l'algorithme. La quantité d'information échangée se calcule en additionnant le nombre de bits de tous les messages échangés. En première analyse elle est donc fortement liée à la complexité en nombre de messages (multiplié par t_G la taille maximum d'un message).

La complexité en temps d'un algorithme distribué $\mathbb{A}$ dans le modèle $\mathbb{M}$ est une mesure paradoxale car le temps lui-même est une notion inutilisable en théorie

dans un réseau asynchrone. Toutefois, il est d'usage de définir (uniquement à des fins d'analyse) soit une borne maximum sur le délai d'acheminement d'un message, soit une fonction de cette borne et de la taille du message véhiculé. Il est évident que cela revient à plonger l'algorithme dans un milieu asynchrone à délai borné et ceci peut provoquer des effets pervers sur la mesure de ces algorithmes. En effet, un algorithme synchrone exécuté sur un réseau synchrone est en général plus performant qu'un algorithme asynchrone exécuté sur un réseau synchrone.

Par ailleurs nous formons l'hypothèse que les temps de calcul locaux ainsi que les temps d'attente des messages dans les tampons des sites sont négligeables face aux coûts de communication (délais de transmission et nombre de messages transmis).

Dans un réseau local ou dans une machine parallèle à mémoire partagée virtuelle les délais de communication sont très faibles (ou tout du moins très peu fluctuants, donc bornés par des fonctions linéaires de la taille des messages véhiculés) par rapport aux temps de calcul local. Notre hypothèse nous place donc au contraire dans le cadre de réseaux non locaux comportant de nombreux sites géographiquement très dispersés.

Chapitre 3

Conclusion de la première partie

Un des problèmes les plus exploré de l'informatique distribuée est de briser la symétrie des processus en donnant à l'un d'entre eux un privilège suivant différentes techniques. Ce sont des algorithmes d'élection et d'exclusion mutuelle dans le sens le plus général du terme (lecteurs-rédacteurs, producteur-consommateurs, exclusion mutuelle simple, etc...).

Le problème de l'élection et, plus généralement encore, l'élaboration en commun d'une structure de contrôle couvrant la totalité du réseau jouent un rôle essentiel en algorithmique distribuée. Nous étudions des algorithmes de construction de ces structures dans la deuxième partie.

Nous avons précisé dans cette partie le cadre de notre travail en illustrant des notions pour la plupart très connues sur la théorie des graphes et en détaillant nos hypothèses et nos choix. En particulier, nous avons délibérément choisi de ne nous intéresser qu'aux algorithmes distribués asynchrones disposant d'au moins une symétrie de texte. Ce choix est guidé par la réalité des réseaux et machines distribuées comme nous le montrons dans le chapitre suivant. La symétrie de texte permet non seulement une grande souplesse d'utilisation, mais il est inconcevable, pour pouvoir exécuter un seul et même algorithme, de programmer des milliers de processeurs avec des codes distincts !

D'autre part, ces algorithmes se caractérisent par leur difficulté d'appréhension : en effet, le comportement d'algorithmes distribués asynchrones nécessite à la fois une étude théorique longue et difficile, mais aussi de nombreuses exécutions pour véritablement saisir toute la valeur d'un algorithme par rapport à un autre.

Deuxième partie

Construction distribuée d'Arbres Couvrants contraints

Chapitre 1

Introduction à la construction distribuée d'Arbres Couvrants contraints

Dans la littérature de l'algorithmique distribuée, le problème de l'élection est souvent lié au problème de la construction d'un Arbre Couvrant (noté AC par la suite) sur le réseau. Usuellement, le processus élu est la racine de l'AC et est d'identité maximum (ou minimum).

Dans cette partie nous présentons trois de nos algorithmes distribués asynchrones de construction d'arbres couvrants contraints sur des graphes quelconques (connexes toutefois).

Note premier apport concerne un algorithme de construction d'AC de poids total maximal très rapide, plus précisément linéaire en temps de calcul. Plus encore, cet algorithme est plus rapide que tous les algorithmes linéaires qui l'on précédé.

Notre deuxième apport est un algorithme qui construit un Arbre Couvrant de Diamètre Minimum, problème relativement nouveau en algorithmique distribuée et dont l'intérêt réside dans le gain potentiel que représente cette structure. En effet, la contrainte de diamètre minimal minimise dans cette structure les délais de communication entre sites ainsi que les messages. Nous considérons le diamètre D d'un graphe sous sa forme « valuée », c'est à dire qui est la somme des poids des arêtes du plus long des plus courts chemins. Clairement, si les poids représentent des délais de communications, il existe toujours au moins deux sites du réseau qui nécessitent au moins D unités de temps pour le transfert d'informations d'un site à un autre. Si les poids sont tous identiques à 1 alors il faudra au moins D messages pour ce même transfert.

Le dernier chapitre de cette partie concerne la construction d'arbre couvrant de degré minimal. Nous verrons que la complexité temporelle d'une telle construction est NP-difficile, mais qu'il existe un algorithme de résolution approché que nous avons adapté au contexte distribué, créant ainsi le premier algorithme distribué de résolution de ce problème.

Chapitre 2

Arbre couvrant de poids total minimal

Dans ce chapitre, nous présentons succinctement un travail réalisé avec Lélia Blin (actuellement maîtresse de conférences à Evry). Pour de plus amples détail voir [BlBu01].

Résumé

Nous présentons un nouvel algorithme distribué asynchrone de construction d'un Arbre Couvrant de poids total minimal sur un graphe $G = (V, E, \omega)$ valué (non négativement) quelconque. Notre algorithme est le plus rapide des algorithmes de complexité linéaire connus, il calcule l'ACM en $|V|/2$ unités de temps sans connaissance structurelle préalable (taille du graphe, topologie, etc.)

2.1 Introduction

Nous présentons ici de façon succincte un algorithme distribué pour la construction d'un Arbre Couvrant de poids (total) Minimal (que l'on notera par la suite ACM). Premier cas d'Arbre Couvrant contraint de cette partie.

Ce problème est un fondamental pour l'algorithmique distribuée et a été parmi les premiers problèmes d'Arbre Couvrant a avoir été étudié. Les arbres sont une structure essentielle dans de nombreux protocoles de communication, par exemple la synchronisation d'horloge, la recherche en largeur d'abord et la résolution d'inter-blocages dans des problèmes d'exclusion mutuelle. Pour diffuser l'information dans le réseau, il est avantageux d'utiliser un ACM car l'information va être délivrée à chaque nœud avec un coût total minimal (si les poids des arêtes du graphe de communication représentent le « coût » d'utilisation).

Le problème de trouver un initiateur/*leader* est réductible au problème de la recherche d'un AC. Parmi toutes les applications possibles, l'élection de leader est utilisée par exemple pour remplacer un coordinateur central défaillant, ou encore pour trouver un site initial dans un système de fichiers distribué.

Pour résumer, la construction d'un AC ou l'élection de leader apparaît comme une brique essentielle de tout protocole réseau complexe et est liée intimement à de nombreux problèmes de calcul distribué. En effet, nous avons montré dans [Bute94] que le problème de l'élection, de la terminaison distribuée et de la recherche d'extremum sont polynomialement réductibles à la recherche d'AC et inversement.

Ici nous considérons le problème de la recherche de l'AC de poids minimal, défini comme suit (voir par exemple [Tel94]) : soit $G = (V, E, \omega)$ un graphe valué non négativement. $\omega(e)$ représente le poids de l'arête $e \in E$ ($n = |V|, m = |E|$). Le poids d'un AC T de G est égal à la somme des poids des $n-1$ arêtes de T, et T est alors appelé un *Arbre Couvrant Minimal* si aucun autre AC n'a un poids inférieur à T.

On suppose dans ce qui suit (comme dans [GaHS83]) que chaque arête à un poids unique, c'est à dire que des arêtes distinctes ont des poids différents. Dans ce cas, il est évident que l'ACM est unique. Cette hypothèse n'est pas très contraignante car en fait on peut toujours « créer » des poids distincts grâce au fait que les identités des nœuds sont déjà supposés distincts (on considère alors le triplet *(poids de l'arête, extrémité de plus faible identité, extrémité de plus forte identité)* et on retrouve une relation d'ordre totale).

Dans un algorithme distribué, chaque noeud de V est associé à son propre processeur/site et les sites peuvent communiquer par échange de messages. Ici l'objectif est la construction de l'ACM.

2.2 Quelques travaux sur l'Arbre Couvrant Minimal

Gallager, Humblet et Spira ont introduit en 1983 [GaHS83] les bases d'un grand nombre d'articles sur le sujet. Citons par exemple [Awer87, ChTi85, FaMo95, GaKP98, KuPe98]. Tous ces algorithmes calculent l'ACM en se basant sur la notion de fusion de fragments qui sont des sous-arbres de l'ACM.

A l'initialisation, chaque nœud est un fragment réduit à ce seul nœud. Les nœuds d'un même fragment coopèrent pour trouver l'arête sortante de poids minimal. Lorsque cette arête est trouvée, le fragment est fusionné avec le fragment se trouvant à l'autre extrémité de l'arête. L'algorithme termine lorsqu'il n'y a plus qu'un seul et unique fragment.

Ces étapes définissent la technique générale utilisée pour calculer l'ACM, mais

les différences entre les articles résident sur l'organisation de la construction à chaque étape. Par exemple, dans l'article pionnier [GaHS83], les tailles des fragments doivent doubler à chaque étape ce qui introduit un nombre d'étapes optimal mais génère des phénomènes d'attente qui peuvent être gênants.

Les algorithmes de [Awer87, ChTi85, FaMo95, GaHS83] cherchent à obtenir à la fois un temps et un nombre de message échangés optimaux. Au contraire [GaKP98] se focalise sur la complexité temporelle et ignore les coûts de communication de leur algorithme. Dans les articles dont le principal intérêt est la complexité temporelle, l'économie de message passe en second, les deux mesures étant en partie corrélées. Cependant, obtenir un algorithme rapide est une tâche complexe, l'algorithme de [GaKP98] nécessite un temps sous-linéaire (c'est à dire inférieur à $O(n)$), mais leur solution apparaît comme artificielle, comme le souligne [Tel98]. En effet l'article de [GaKP98] est plus une preuve d'existence d'un algorithme sous-linéaire (mais plus que linéaire si on ne considère plus n mais le diamètre du réseau). Dans un graphe quelconque, un paramètre peut être plus réaliste pour décrire la complexité temporelle est le diamètre de l'ACM résultant.

2.3 Nos résultats

Notre algorithme asynchrone est du type recherche de la meilleur complexité temporelle tout en conservant un nombre de messages échangés raisonnable (de l'ordre de n^2 dans le pire des cas, rappelons que dans un graphe complet il y a presque n^2 arêtes). Pour un graphe quelconque, le problème de la construction d'un ACM distribué requiert au moins $O(m + n \log n)$ messages. La transmission d'un message sur une arête requiert une unité de temps. La complexité temporelle de [GaHS83] est en $O(n^2)$ et sa complexité en nombre de messages échangés est de $2m + 5n \log n$.

[Awer87, ChTi85] ont amélioré la complexité temporelle (respectivement $\simeq 4n$ et $n \log^* n$), mais pas la complexité en nombre de messages échangés. Nous proposons un algorithme qui s'exécute en $n/2$ unités de temps avec $O(n^2)$ messages.

2.4 Description de l'algorithme

L'algorithme présenté comporte trois étapes principales. Nous supposons qu'un sous-ensemble non vide de sites démarrent l'algorithme spontanément. Les autres sites s'éveillent sur réception de message.

Étape initiale

Initialement, chaque site x qui démarre l'algorithme se considère responsable de son fragment. Un fragment porte cette identité initiale. x diffuse ensuite à tous

ses voisins immédiats le message $< NoChild >$ (non fils) excepté pour le voisin se trouvant à l'autre extrémité de l'arête de poids minimum sortante de x pour lequel il envoie le message $< Child >$ (fils).

A la fin de cette étape, chaque nœud connaît qui est son père et ses éventuels fils dans le futur arbre couvrant (mais le sens père-fils pourra changer dans la suite de l'algorithme). De plus, chaque nœud sait s'il est une feuille de l'arbre ou non. Notons qu'un fragment peut donc avoir deux « racines », chacune étant le père-fils de l'autre. Ces « racines » seront appelées par la suite des « Centres de Décision » ou CD.

Étape 2 : récolte des informations

Dans cette deuxième étape, chaque feuille envoie la liste (message $Data$) des ces arêtes sortantes avec leurs identités vers leurs parents respectifs. Un nœud interne reçoit les informations de ses fils, les fusionne et les fait suivre jusqu'au CD par son père.

A la fin de cette étape, le CD calcule, à partir des informations reçues l'arête sortante du fragment de poids minimal. Cette arête devient l'arête de fusion avec un autre fragment. S'il y a deux CD, ils font le même calcul et donc découvrent dans quelle direction se trouve l'arête sortante. Ils décident alors qui est celui des deux qui doit démarrer l'étape de connexion.

La direction dans l'arbre est arbitraire, nous pouvons donc la changer. Si un nœud reçoit des informations de son père et qu'il manque une information d'un des fils, ce fils peut devenir son nouveau père. L'idée est de récupérer l'information sans attendre. Cette amélioration diminue la complexité temporelle surtout lorsque l'ACM final est de faible degré (au pire il s'agit d'une chaîne).

Étape 3 : connexion

Dans cette dernière étape, les différents fragments fusionnent. Quand un CD trouve l'arête sortante de poids minimum, il envoie un message $< down >$ à l'extrémité de cette arête sortante. Ce message descend dans le fragment par la branche conduisant à cette extrémité et est envoyé sur l'arête sortante de poids minimum.

Quand un nœud reçoit ce message d'un fragment qui n'est pas le sien il envoie un message $< up >$ à son CD via la suite de ses pères. Le CD qui reçoit ce message peut alors calculer la nouvelle arête sortante de poids minimum.

Cette étape se termine lorsque le fragment est unique, c'est à dire lorsqu'il n'y a plus d'arête sortante.

Le lecteur intéressé pour obtenir l'algorithme détaillé dans [BlBu01].

2.5 Analyse

2.5.1 Rappel des propriétés de base des ACM

Les propriétés suivantes sont nécessaires pour les preuves de correction. Voir [GaHS83] pour des preuves détaillées.

1. Si tous les poids des arêtes d'un graphe connexe G distincts, alors G a exactement un ACM.

2. Nous supposons que tous les poids des arêtes sont distincts. Soit F un fragment de l'ACM et soit e l'arête de poids minimum sortant de F. L'ajout de l'arête e à F crée un nouveau fragment sans création de cycle. Ce nouveau fragment est toujours un fragment de l'ACM.

Corollaire 1 *Alors, (avec les mêmes hypothèses) pour un fragment F, il n'y a plus d'arête sortante, la construction est terminée et F est l'ACM.*

2.5.2 Complexité

Théorème 2 *Le nombre total de messages échangés dans le pire des cas pour un graphe $G = (V, E)$ est $2m + \frac{n^2}{8} + O(n)$ où $n = |V|$ et $m = |E|$.*

Preuve : Le pire des cas est atteint lorsque le graphe est un graphe complet et que l'ACM est réduit à une chaîne (et ceci est le pire des cas pour tout algorithme de construction utilisant les arêtes de l'ACM pour les échanges d'information).

Calculons le nombre de messages échangés durant chaque étape. Nous supposerons, pour simplifier l'exposé, que tous les nœuds démarrent simultanément l'algorithme.

Dans l'étape initiale, tous les nœuds envoient les messages $< Child >$ et $< noChild >$, donc $2m$ messages échangés. Le nombre de fragments créés alors est au plus de $\lfloor \frac{n}{2} \rfloor$, chaque fragment étant composé d'au moins deux nœuds (cela peut être 3 si n est impair).

Ensuite pour l'étape de récolte des informations, nous supposons, pire des cas, que le nombre de fragments est maximal. Dans ce cas, chaque nœud de chaque fragment envoie ses informations vers l'autre nœud. Notons que le nombre de message *Data* échangés peut être plus grand avec de plus grands fragments mais il y a un équilibre entre le nombre de fragments et leur taille. Dans cette étape, au plus n messages sont échangés.

Finalement dans l'étape de connexion, nous allons considérer des niveaux. Un niveau peut être défini par le nombre de fragments. Nous allons du niveau $i + 1$ au niveau i quand le nombre de fragments est divisé par 2 (pour simplifier l'analyse nous alors supposer que $n = 2^k$). Pour aller du niveau $i + 1$ au niveau i, les messages $< down >$ doivent traverser leurs fragments et ensuite passer au

travers des meilleures arêtes sortantes. Les messages $< up >$ doivent traverser les fragments pour « chasser » le CD. Par construction et puisque nous nous intéressons au pire des cas, un fragment va absorber tous les autres. Cela donne un grand nombre de messages $< up >$ plus une oscillation des messages $< down >$ le long de la chaîne de l'ACM. Nous définissons le niveau 1 tel qu'il ne reste qu'un fragment. Donc nous démarrons au niveau $n/2 - 1$ avec au plus $n/2$ fragments de taille 2. Cela donne $\Sigma_{i=1}^{n/2-1}(i+1) = \frac{n^2}{8} + \frac{n}{4} - 1$.

En sommant les étapes, on obtient le nombre de messages donné dans le théorème.

Notons que si la croissance des fragments est au contraire bien équilibrée, le nombre de messages échangés est lié essentiellement aux messages $< down >$ et le total est de l'ordre de $2m + \frac{1}{2}n\log(n) + O(n)$. □

Théorème 3 *Si tous les nœuds se réveillent spontanément en même temps, la complexité temporelle de cet algorithme est* $\frac{n}{2} + O(1)$.

Preuve : Le pire des cas est le même que dans la preuve précédente. La complexité temporelle découle alors du fait que les messages doivent traverser la chaîne de l'ACM de bout en bout. Quand ils arrivent au milieu, toute l'information a été collectée et l'algorithme est terminé. □

2.5.3 Terminaison

Cet algorithme est de type « terminaison par message ». Pour obtenir une terminaison par processus il est facile d'ajouter un nouveau type de message qui sera diffusé sur l'ACM par le CD à la fin du calcul. Ceci n'ajoute que $n - 1$ messages et $H(n)$ unités de temps, où $H(n)$ est la hauteur de l'ACM.

2.6 Conclusion

Dans ce chapitre, nous avons présenté un algorithme distribué asynchrone de construction d'Arbre Couvrant de poids total Minimum. Notre algorithme propose une solution de calcul de l'ACM en $n/2$ unités de temps avec $O(n^2)$ messages échangés (où n est le nombre nœuds du graphe). Le nombre total de messages dans le pire des cas est légèrement supérieur aux autres algorithmes, mais en pratique nos résultats sont meilleurs.

Ce travail a été produit durant l'encadrement de la thèse de Lélia Blin (à 50% avec Ivan Lavallée).

Chapitre 3

Arbre couvrant de diamètre minimal

Nous présentons dans ce chapitre un extrait des travaux réalisés avec Messieurs Marc Bui (professeur à l'Université de Paris 8) et Christian Lavault (professeur à l'Université de Paris 13) sur la construction d'Arbres Couvrants de Diamètre Minimum et de calcul des plus courts chemins.

Pour plus de détails le lecteur intéressé pourra consulter [BuBL04]. Notons que nous avons aussi proposé une version avec sites sans identités (réseau dit anonyme), voir [BuLB95a].

Résumé

Nous présentons un nouvel algorithme de construction d'un Arbre Couvrant de Diamètre Minimal de façon distribuée sur un graphe $G = (V, E, \omega)$ valué (non négativement) quelconque. Comme étape intermédiaire, nous présentons, pour la recherche d'un centre absolu de G, un nouvel algorithme de calcul des plus courts chemins qui s'exécute en temps linéaire. L'algorithme distribué résultant est asynchrone, il fonctionne pour des réseaux arbitraires non anonymes et s'exécute en $O(V)$ unités de temps et en $O(|V|\,|E|)$ pour la complexité en nombre de messages échangés.

3.1 Introduction

De nombreux réseaux de communications requièrent de leurs sites une diffusion d'information à tous les autres ne serait ce que pour des besoins de contrôle du réseau lui-même. Ceci peut être fait efficacement en émettant les messages suivant un arbre, couvrant le réseau de communication. De plus, pour optimiser

le temps de propagation de l'information sur un arbre couvrant, il faut que son diamètre soit minimal.

L'utilisation d'une structure de contrôle couvrant l'ensemble du réseau est un problème fondamental en systèmes distribués et dans les réseaux d'interconnexion.

Étant donné un réseau, un algorithme distribué est dit *total* si et seulement si tous les nœuds (sites) participent au calcul. Tous les algorithmes distribués totaux sont en complexité temporelle en $\Omega(D)$, où D est le diamètre du réseau (c'est à dire en termes de nombre de sauts *hops*, ou suivant le sens étendu donné par Christophides dans [Chri75], en terme de longueur de chemin, voir en partie I, le chapitre 1).

Ainsi, ayant un AC de diamètre minimal sur un réseau arbitraire, il est possible de concevoir une grande variété d'algorithmes distribués efficaces.

Pour construire un tel arbre, il est nécessaire (en l'état actuel des connaissances) de calculer l'ensemble des plus courts chemins pour toute paire de nœuds. Plusieurs algorithmes distribués existent pour résoudre ce problème sous diverses hypothèses. Cependant, nos hypothèses de travail sont plus générales que celles communément utilisées. Par exemple, nous avons besoin d'un algorithme disposant d'une terminaison par processus pour des réseaux valués en l'absence de connaissance commune.

3.2 Modèle, notations et définitions

Nous rappelons ici notre modèle standard : réseau asynchrone, point à point, constitué de n sites communicants par m canaux bidirectionnels. Chaque processus dispose d'une mémoire locale non partagée et communique par échange de messages avec ses voisins immédiats. Un site peut transmettre et recevoir de plus d'un site à la fois.

Le réseau est un graphe simple, fini, connexe, non orienté, valué $G = (V, E, \omega)$. Nous supposons que, pour tout $(u, v) \in E$, $\omega(u, v) = \omega(v, u)$ et pour simplifier la notation $\omega(u, v) = \omega(e)$ représente le poids de l'arête $e = (u, v)$. $|V| = n$, $|E| = m$.

Un **centre absolu** de G est défini comme le nœud (pas forcément unique) réalisant l'excentricité minimale sur G (voir chapitre 1, partie I).

$D = D(G)$ représente le *diamètre* de G, défini par $D = \max_{v \in V} ecc(v)$ (voir [Chri75]), et $R = R(G)$ le *rayon* de G, défini par $R = \min_{v \in V} ecc(v)$.

Finalement, $\Psi(u) = \Psi_G(u) \in ACC_G(u)$ représente l'*Arbre des plus Courts Chemins* (ACC) de G enraciné en u : $(\forall v \in V)\ d_{\Psi(u)}(u, v) = d(u, v)$. $\Psi(u)$ est choisi *de façon unique* parmi les arbres de plus courts chemins de G enracinés en u. Lorsqu'il y a le choix entre deux chemins de même longueur, nous choisissons le chemin tel que le deuxième nœud rencontré soit d'identité minimale. Nous

noterons $\Psi = \Psi(G)$ l'ensemble des ACC de G.

Voir la section 1.3.2 du chapitre 1, partie I, pour quelques propriétés sur les rayons, diamètres et centres.

3.3 Le problème

Soit un graphe valué $G = (V, E, \omega)$, le **problème de l'ACDM** est de trouver un arbre couvrant G de diamètre minimum.

Notons que ce problème suppose que G est valué/étiqueté avec des valeurs non négatives (c.-à-d., $\forall e \in E \quad \omega(e) \in \mathbb{R}^+$). En effet, le problème de l'ACDM est connu pour être NP-difficile si l'on autorise des cycles de longueur négative (voir [CaGM80]).

En dépit du fait que ce problème requière des poids d'arêtes non négatifs réels, notre algorithme termine par processus. Ceci n'est en général pas le cas avec ce type de contraintes. Quand les arêtes sont supposés de poids réel, une hypothèse (supplémentaire) est souvent nécessaire sur la connaissance d'au moins une borne sur la taille du réseau (voir par exemple [Awer85c, BeGa92, Lamp82]). Au contraire, notre algorithme ne nécessite aucune « information structurelle », qu'elle soit topologique : taille ou borne sur la taille du réseau, ni sens de la direction etc.

Les autres hypothèses sont celles déjà formulées dans le chapitre 2, partie I.

3.4 État de l'art

Le peu de littérature traitant du problème d'ACDM est lié soit à des problèmes de graphe dans le plan Euclidien (Arbre Couvrant de Diamètre Géométrique Minimum), ou à la construction de l'arbre de Steiner (voir [HLCW91, IhRW91])

Le problème de l'ACDM est clairement une généralisation du problème de l'ACDGM. Le problème de l'ACDM en séquentiel a été traité par quelques auteurs (voir par exemple [Chri75]).

Étonnamment, en dépit de l'importance d'avoir un ACDM dans un système distribué arbitraire, peu d'articles ont traité ce problème. Le problème de maintenir un diamètre faible a toutefois été traité dans [ItRa94], et le problème en distribué par [BuBu93b, BuLB95].

3.5 Notre apport

Notre algorithme de construction des Arbres de plus Court Chemins est une généralisation des algorithmes d'ACC sur des graphes non valués (ou valués avec des poids unitaires). A notre connaissance, notre algorithme de

construction d'ACDM est aussi le premier qui résout ce problème de façon distribuée [BuBu93b]. L'algorithme ACDM fonctionne sur des topologies arbitraires avec des communications asynchrones. Il s'exécute avec une complexité temporelle en $O(n)$ et une complexité en quantité d'information échangée en $O(nm(\log n + \log W))$ bits (W est le maximum sur les poids des arêtes).

Dans la section suivante nous présentons une description succincte de notre algorithme, ensuite nous verrons son analyse.

3.6 L'algorithme

3.6.1 Description

D'abord, rappelons que le problème de l'ACDM sur un graphe valué est polynomialement réductible au problème de la recherche d'un centre absolu de G [BuBL04]. La recherche d'un tel centre peut être fait en calculant les tables de routage des ACCs (voir lemme 5).

Voici les étapes principales de notre algorithme :

1. Calcul des ACCs de G ;
2. Calcul d'un centre absolu de G
3. Construction de l'ACDM et transmission de cet ACDM à tous les nœuds du réseau G.

Nous allons voir dans la section suivante ce qu'est un centre absolu d'un graphe et comment l'on peut en déduire un ACDM. Ensuite nous verrons comment diffuser la connaissance acquise (fin de l'étape 3). Enfin nous traiterons l'étape 1 par un nouvel algorithme (la procédure ACC) en section 3.6.5.

3.6.2 Du centre absolu à l'ACDM

La définition de l'excentricité peut être généralisée comme suit : nous considérons une arête (u, v) de poids ω comme un intervalle continu de longueur ω, et pour tout $0 < \alpha < \omega$ nous autorisons l'insertion d'un « nœud artificiel » γ et remplaçons l'arête (u, v) par les arêtes : (u, γ) et (γ, v) de poids respectivement α et $\omega - \alpha$.

Suivant la définition, l'excentricité $ecc(\gamma)$ d'un *nœud généralisé* γ (c'est à dire soit un vrai nœud de V soit un nœud artificiel) est clairement définie par $ecc(\gamma) = \max_{z \in V} d(\gamma, z)$. Un nœud γ^* tel que $ecc(\gamma^*) = \min_\gamma ecc(\gamma)$ est appelé *centre absolu* du graphe. Rappelons que γ^* existe toujours dans un graphe connexe et qu'il n'est pas unique en général. De plus, un centre absolu de G est souvent un des nœuds artificiels.

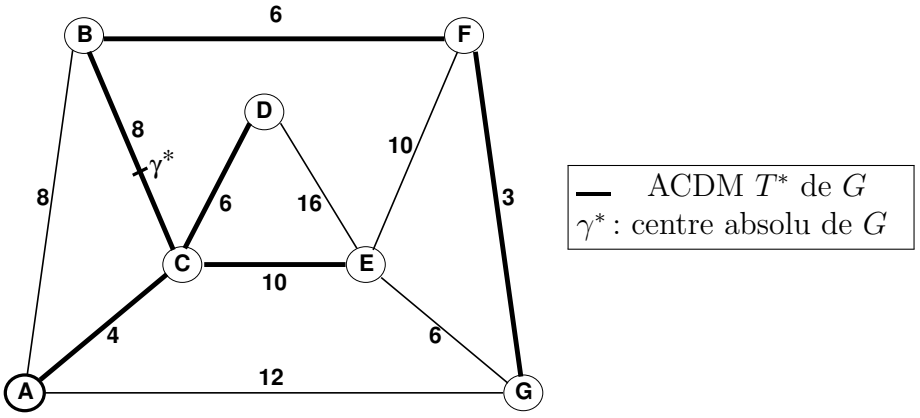

FIGURE 3.1 : Exemple d'ACDM T^* ($D(G) = 22$ et $D(T^*) = 27$). Notons que T^* est ni un Arbre des plus Courts Chemins, ni un arbre couvrant de poids total minimum.

De même, la définition de $\Psi(u)$ est aussi généralisée pour prendre en compte les nœuds artificiels. Chercher un ACDM ce réduit alors à chercher un centre absolu γ^* de G : l'ACC enraciné en γ^* est un ACDM de G. C'est le propos du lemme suivant :

Lemme 4 [CaGM80] *Étant donné un graphe valué G, le problème de l'ACDM sur G est (polynomialement) réductible au problème de la recherche d'un centre absolu de G.*

3.6.3 Calcul d'un centre absolu d'un graphe

L'idée du calcul des p-centres absolus a été introduite par Hakimi, voir par exemple [HaPS78]. Ici nous nous intéressons au cas où $p = 1$. Suivant les résultats de [Chri75], nous avons besoin du lemme suivant (dite méthode de Hakimi) pour chercher un centre absolu de G.

Lemme 5 *Soit $G = (V, E, \omega)$ un graphe valué. Pour chaque arête $e \in E$, soit γ_e l'ensemble de tous les nœuds généralisés de G d'excentricité minimale pour e. Un nœud qui atteint l'excentricité minimale parmi tous les nœuds de $\bigcup_{e \in E} \gamma_e$ est un centre absolu. Trouver un centre absolu de G est donc calculable en temps polynomial.*

Preuve : (La preuve est constructive)
(i) Pour tout arête $e = (u, v)$, soit $\alpha = d(u, \gamma)$. Puisque la distance $d(\gamma, z)$ est la

longueur d'un chemin $[\gamma, u, \ldots, z]$ ou d'un chemin $[\gamma, v, \ldots, z]$,

$$ecc(\gamma) = \max_{z \in V} d(\gamma, z) = \max_{z \in V} \min(\alpha + d(u,z), \omega(u,v) - \alpha + d(v,z)). \qquad (3.1)$$

Si nous traçons $f_z^+(\alpha) = \alpha + d(u,z)$ et $f_z^-(\alpha) = -\alpha + \omega(u,v) + d(v,z)$ en coordonnées cartésiennes pour $z = z_0$ fixé, les fonctions à valeurs réelles $f_{z_0}^+(\alpha)$ et $f_{z_0}^-(\alpha)$ (séparément dépendantes de α dans l'intervalle $[0, \omega(e)]$) sont représentées par deux segments $(S_1)_{z_0}$ et $(S_{-1})_{z_0}$, de pente $+1$ and -1, respectivement. Pour un $z = z_0$ fixé, le plus petit des deux termes $f_{z_0}^+(\alpha)$ et $f_{z_0}^-(\alpha)$ dans (3.1) défini une fonction linéaire par morceaux $f_{z_0}(\alpha)$ composée de $(S_1)_{z_0}$ et $(S_{-1})_{z_0}$.

Soit $B_e(\alpha)$ la *borne supérieure* ($\alpha \in [0, \omega(e)]$) de tous les $f_z(\alpha)$ ($\forall z \in V$). $B_e(\alpha)$ est une courbe formée de segments linéaires, qui passent par plusieurs minimums locaux. (voir Figure 3.2). Un point γ réalisant le minimum global de $B_e(\alpha)$ est un centre absolu γ_e^* de l'arête e.

(ii) De par la définition de γ_e^*, $\min_\gamma ecc(\gamma) = \min_{\gamma_e^*} s(\gamma_e^*)$; et γ^* réalise l'excentricité minimale. Donc, un centre absolu γ^* du graphe est trouvé en tout point tel que le minimum de tous les $ecc(\gamma_e^*)$s est atteint. □

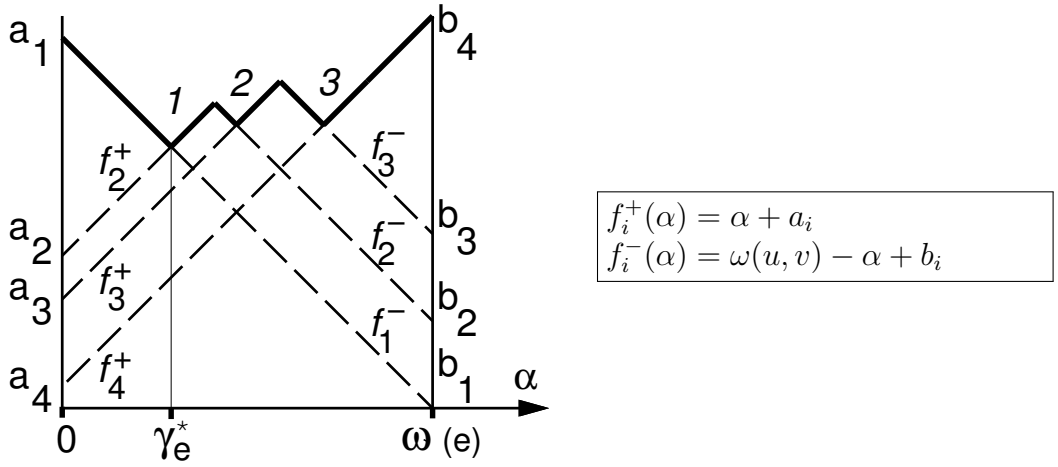

FIGURE 3.2 : Exemple de borne supérieure $B_e(\alpha)$

De la vision géométrique à l'algorithme

Par le lemme 5, nous pouvons considérer cette méthode d'un point de vue algorithmique :

Pour chaque $e = (u, v) \in E$, soit

$$\mathcal{C}_e = \{(d_1, d_2) \mid (\forall z \in V) \quad d_1 = d(u,z) \text{ et } d_2 = d(v,z)\}.$$

Une paire (d_1', d_2') *domine* une paire (d_1, d_2) ssi $d_1 \leq d_1'$ et $d_2 \leq d_2'$; plus précisément, la fonction $f_{z'}(\alpha)$ définie par (d_1', d_2') est au dessus de $f_z(\alpha)$ définie par (d_1, d_2). Toute paire dominée par une autre sera ignorée. Les minimums locaux de la borne supérieure $B_e(\alpha)$ (numérotés de 1 à 3 dans la Figure 3.2) sont à l'intersection des segments $f_i^-(\alpha)$ et $f_{i+1}^+(\alpha)$, quand toutes les paires dominées sont retirées. Le tri dans l'ordre descendant l'ensemble $\mathcal{C}_e$ par rapport au premier terme de chaque paire restante (d_1, d_2), donne la liste $L_e = ((a_1, b_1), \ldots, (a_{|L_e|}, b_{|L_e|}))$ constituée de toutes les paires dominantes. Alors, le plus petit minimum local de $B_e(\alpha)$, pour une arête e donnée fournit un centre absolu γ_e^* par rapport à cette arête.

Par le Lemme 5, une fois que tous les γ_e^* sont calculés, nous pouvons obtenir un centre absolu γ^* du graphe G. Finalement, par le Lemme 4, trouver un ACDM de G se réduit au problème du calcul de γ^*.

3.6.4 Construction et diffusion de la connaissance d'un ACDM

A la fin du protocole *ACC*, tous les nœuds ont connaissance du nœud u_{min} avec la plus petite identité et d'un plus court chemin dans G qui mène vers lui.

Maintenant considérons la collection de chemins $[u, \ldots, u_{min}]$ (calculés par *ACC*). Cette collection forme un arbre enraciné en u_{min} et puisqu'il s'agit d'un ACC de G, cette information est échangée « optimalement » sur l'ACC $\Psi(u_{min})$ [1].

Ainsi, le nombre de messages nécessaire à la recherche d'un extremum sur l'arbre $\Psi(u_{min})$ est au plus en $O(n)$ (avec des messages de longueur en $O(\log n + \log W)$).

Lorsque le calcul d'un centre absolu γ^* de G est réalisé, l'extrémité d'identité minimale de l'arête portant γ^* émet un message vers u_{min} portant l'identité de γ^*. Sur réception de ce message, u_{min} diffuse cette information sur l'arbre $\Psi(u_{min})$ (ajoutant ainsi le même coût en termes de temps et de messages échangés).

Finalement, chaque nœud de G connaît alors un chemin vers γ^*, et l'ACDM est virtuellement construit de par l'ensemble des connaissances des sites. Le lecteur intéressé pourra consulter l'algorithme détaillé dans [BuBL04].

Nous allons présenter maintenant la procédure *ACC*.

1. Sur $\Psi(u_{min})$, cette information est transmise *« optimalement »* en terme de temps et de messages, dans le sens que chaque poids arête peut être considérée comme le délai de transmission d'un message sur un canal.

3.6.5 Algorithme des plus courts chemins (protocole *ACC*)

En section 3.6.3, nous considérons les distances $d(u,z)$ et $d(v,z)$, pour tout $z \in V$ et pour toute arête $e = (u,v) \in E$. Ces distances doivent être calculées par un algorithme de routage distribué (avec terminaison par processus) ; le protocole ACC est conçu à cette fin.

Notre algorithme est basé sur l'algorithme Netchange (voir sa preuve dans [Lamp82]), l'algorithme de Bellman-Ford [BeGa92] et le synchroniseur α décrit dans [Awer85c]. Ces trois algorithmes terminent « proprement » (terminaison par processus) si et seulement si une borne supérieure sur n est connue. Cependant, une terminaison par processus peut être obtenue sans aucune connaissance supplémentaire en utilisant la technique décrite dans [ChVe90]. Nous allons maintenant décrire succinctement l'algorithme (du point de vue d'un site u, dont l'identité est id_u).

Le protocole ACC est divisé en phases après une étape d'initialisation des variables. Les variables utilisées sont les suivantes : le site sélectionné fixé initialement à id_u, la distance à ce site fixée ici à 0, les autres distances estimées (tables de routage) sont initialisées à $+\infty$ et l'ensemble *Updatenodes* initialisé à $\emptyset$. Ensuite chaque phase de l'algorithme est constituées de trois étapes comme suit :

Étape 1. Émission à tous les voisins immédiats de l'identité du site sélectionné avec sa distance à u.

Étape 2. Attente de réception du même nombre de message que le nombre d'envois de l'étape 1. moins le nombre de voisins inactifs (voir paragraphe suivant). Sur réception d'un message, mise à jour des tables de routage. Si l'estimation d'une distance à un site change, ajouter ce site à l'ensemble *Updatenodes*. Si un message ⟨**Inactif**⟩ est reçu d'un voisin, le marquer *inactif*. Lorsque le nombre de messages attendu est atteint, démarrer l'étape 3.

Étape 3. Choisir un site actif de l'ensemble *Updatenodes* de distance à u minimale et aller à l'étape 1. S'il n'existe pas un tel site, envoyer un message ⟨**Inactif**⟩ à tous les voisins actifs et devenir inactif.

Nous avons besoin de quelques règles pour que l'algorithme termine par processus.

1. Un et un seul message ⟨**Inactif**⟩ est émis de u à un voisin v et ce message est le dernier du protocole ACC de u à v.

2. (de la règle précédente) Un site termine uniquement quand deux messages ⟨**Inactif**⟩ sont émis (chacune dans un sens) pour toutes ses arêtes adjacentes.

Ce nouvel algorithme d'ACC a une très bonne complexité en nombre de messages échangés ($2mn$).

3.7 Analyse

Pour l'analyse de complexité, soit $W \in \mathbb{R}^+$ le poids de l'arête de poids maximum dans E : le nombre de bits pour coder W en binaire est $\lceil \log_2 W \rceil$. Donc envoyer dans un message le poids d'une arête coûte $O(\log W)$ bits. Les preuves sont disponibles dans [BuBL04].

Lemme 6 *Le protocole ACC termine par processus. Il s'exécute en $O(n)$ unité de temps et échange de l'ordre de $O(nm)$ messages pour le calcul des tables de routage en chaque nœud de G. La taille des messages échangés est au plus en $O(\log n + \log(nW))$.*

Théorème 4 *L'algorithme ACDM résout le problème de l'ACDM pour tout graphe/réseau valué positivement, asynchrone en $O(n)$ unités de temps. Sa complexité en quantité d'information échangée est en $O(nm(\log n + \log(nW)))$ bits, et sa complexité spatiale est en au plus $O(n(\log n + \log(nW)))$ bits (pour chaque nœud). Le nombre de bits utilisés pour l'identité d'un nœud est $O(\log n)$, et le poids d'un chemin ayant ce nœud pour extrémité est $O(\log nW)$.*

La preuve découle du lemme précédent et des remarques faites sur la construction de l'arbre.

3.8 Conclusion

Sur tout graphe connexe positivement valué G, notre algorithme construit un ACDM sur G et répartit la structure de contrôle sur le réseau G. Cet algorithme est nouveau, simple et naturel. Il est aussi efficace en temps et en messages : ses mesures de complexité $O(n)$ et $O(nm)$ respectivement, sont, en un certain sens, a peu près les meilleures possibles (bien que non optimales) en distribué. Nous avons par ailleurs décrit, comme procédure initiale, un nouvel algorithme de calcul des plus courts chemins sans aucune connaissance structurelle sur le graphe et terminant par processus avec une très bonne complexité temporelle ($2m$ messages échangés).

La complexité spatiale de l'algorithme de construction d'ACDM peut paraître non satisfaisante. Ceci est en fait une conséquence de nos hypothèses très générales, en particulier l'hypothèse de construction de tables de routage complètes sur un réseau arbitraire. Notre algorithme nécessite en tout $O(n^2(\log n + \log(nW)))$ bits pour stocker ces tables. Ceci étant dit, il a été montré que tout schéma de routage raisonnable requière au moins $\Omega(n^2)$ bits [FrGa95]. Ceci n'est finalement qu'à un facteur logarithmique près la complexité spatiale de notre algorithme.

Chapitre 4

Arbre couvrant de degré minimal

Nous présentons dans ce chapitre un extrait d'un travail réalisé avec Lélia Blin (actuellement maîtresse de conférences à Evry) [BlBu04].

Résumé

Dans cette article nous présentons le premier algorithme distribué sur graphe quelconque pour le problème de l'Arbre Couvrant de Degré Minimal. La contrainte de degré minimal traduit un besoin en termes d'équipement d'interconnexion (éviter leur surcharge, limiter leur taille en nombre de ports etc.).

Le problème est NP-difficile en séquentiel. Notre algorithme trouve un arbre couvrant dont le degré est à au plus 1 de l'optimal.

L'algorithme distribué résultant est asynchrone, il fonctionne pour des réseaux quelconques non anonymes et s'exécute en $O(|V|)$ unités de temps et nécessite au plus $O(|V|\,|E|)$ messages.

4.1 Introduction

Dans ce chapitre, nous portons notre attention sur le problème de la construction d'un Arbre Couvrant de Degré Minimum Approché que nous noterons **ACDegM** pour le différentier du problème du chapitre précédent.

Nous avons déjà évoqué l'intérêt de la construction d'un arbre couvrant, la particularité de celui-ci vient de l'observation pratique de l'activité d'un algorithme distribué ou de tout protocole réseau en général. Si le degré d'un nœud est important cela peut une entraîner une surcharge locale (goulot d'étranglement) De même, si, lors d'une diffusion par exemple, on veut limiter la quantité de travail par nœud, son degré doit être minimum. Le problème peut aussi être

d'ordre économique : des éléments d'interconnexion de faible taille (nombre de ports d'entrée/sortie) sont souvent moins chers et plus faciles à remplacer.

4.1.1 Le problème

Dans cet article nous considérons le problème de la recherche d'un ACDegM. Soit $G = (V, E)$ un graphe avec $n = |V|$ nœuds/sites et $m = |E|$ arêtes/liens de communication. Un Arbre Couvrant $T = (V, E_T)$ de G est un arbre tel que $E_T \subset E$. Parmi tous les AC de G, nous cherchons un de degré minimal (le degré d'un graphe est égal au degré maximum de ces nœuds). Ce problème a été prouvé comme étant NP-difficile car généralisation du problème de la recherche d'un chemin hamiltonien [GaJo79]. Nous chercherons donc un algorithme distribué approché.

4.1.2 État de l'art

Le problème du ACDegM a déjà été étudié mais essentiellement d'un point de vue séquentiel. A notre connaissance, il y a uniquement un article sur ce problème en distribué : il s'agit de [KoMZ87] qui traite ce problème sur un graphe complet. Ils montrent que tout algorithme qui construit un arbre couvrant de degré au plus k utilise au moins, dans le pire des cas, $O(\frac{n^2}{k})$ messages.

En séquentiel, [FuRa92] propose une heuristique pour l'ACDegM dont le degré est à 1 près de l'optimal. Leur travail part d'un AC quelconque puis leur algorithme améliore l'arbre (du point de vue de son degré) par échange d'arêtes.

Ils montrent que la meilleure borne atteignable, en temps polynomial, est le degré optimal plus 1. Leur heuristique est simple, elle utilise un propriété locale d'optimalité : aucune arête (n'appartenant pas à l'AC) ne doit permettre de réduire le degré maximal des nœuds sur le cycle induit par cette arête.

Dans ce chapitre, nous présentons un algorithme distribué basé sur les idées de [FuRa92]. Au départ, nous avons besoin d'un AC. Pour construire un tel arbre, de nombreux algorithmes existent tels ceux construisant un AC de poids total minimal [GaHS83, Awer87, ChTi85, FaMo95, GaKP98, BlBu01].

La complexité de notre algorithme est fonction du degré initial k du premier AC et du degré k^* de son AC, localement optimal, correspondant. Nous obtenons un algorithme en $O((k - k^*)m)$ messages et $O((k - k^*)n)$ unités de temps. Ces résultats montrent que nous ne sommes pas très éloignés de l'optimal [KoMZ87].

4.2 Notre algorithme

4.2.1 Méthode

Nous allons décrire la méthode que nous utilisons pour construire un ACDegM. Nous supposons qu'un AC a déjà été construit et par ailleurs la plupart des algorithmes de construction d'AC fournissent en même temps la racine de cet arbre et une orientation virtuelle (père-fils) des arêtes de cet arbre.

Notre algorithme est une boucle qui comporte les étapes suivantes (l'ensemble des étapes qui constituent cette boucle sera appelée une itération) :

SearchDegree Chercher le nœud de l'AC de degré maximum k et d'identité minimale. Soit p ce nœud. Par la suite, pour simplifier la description de la méthode, supposons que p est le seul nœud de degré k.

MoveRoot Déplacer la racine de l'abre vers p.

Cut Couper virtuellement les fils x de p dans l'AC. Chaque x est maintenant la racine d'un sous-arbre appelé fragment.

BFS Chaque x démarre un parcours en largeur d'abord (*Breadth First Search*) (ou vague BFS) jusqu'au arêtes sortantes du fragment. Une fois la vague revenue, x envoie à p sa meilleure arête sortante.

Choose p choisit une arête sortante pour faire un échange d'arêtes. Ce faisant, il diminue son degré k (voir Figure 4.1).

L'algorithme exécute les étapes précédentes jusqu'à ce qu'aucune amélioration ne puisse être trouvée ou que $k = 2$ (l'arbre est alors une chaîne).

Lorsqu'il y a plus d'un nœud de degré k, ces nœuds se comportent comme des racines excepté qu'ils doivent envoyé un message vers la vraie racine de l'arbre, seule habilitée à faire le choix de l'arête sortante.

4.2.2 Description détaillée de l'algorithme

Nous supposons un AC déjà construit par un algorithme disposant d'une terminaison par processus.

Nous allons décrire chaque étape de l'algorithme du point de vue d'un nœud x.

SearchDegree

Chaque nœud coopère à la recherche du nœud de degré maximum dans l'AC. A la fin de cette étape, la racine de l'AC est informée du degré maximum de l'AC.

– Chaque feuille x de l'AC émet un message avec son degré (1 donc) vers son père.

– Chaque nœud x recevant les degrés de tous ces fils conserve celui de degré maximal k. Il le compare aussi à son propre degré. Si deux nœuds, ou plus, ont le degré k alors x choisit celui d'identité minimale y (x peut être égal à y).

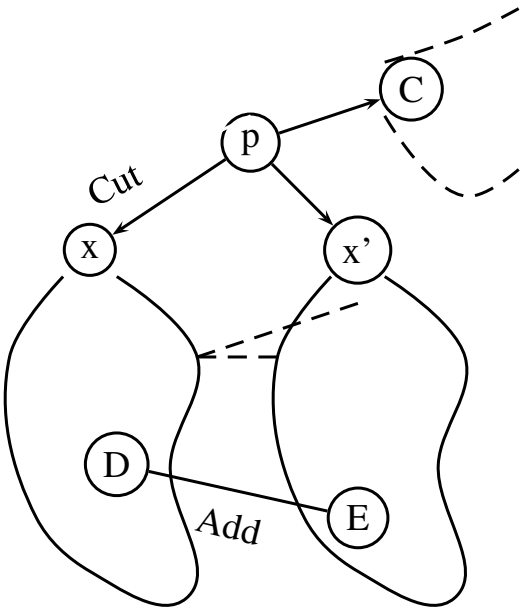

FIGURE 4.1 : Réduction du degré maximal de l'AC

Ensuite, x envoie le degré maximal et y à son père (tout en les conservant en mémoire).

– Ce processus est répété jusqu'à ce que les messages arrivent à la racine de l'arbre. La racine choisit de même entre les identités et degrés reçus.

MoveRoot

La racine est déplacée vers le nœud de degré maximal p (et d'identité minimale). Durant ce déplacement, le chemin est « retourné » (voir la notion classique de *Path Reversal* par exemple dans [Lavau95]). A la fin de cette étape, la nouvelle racine est p.

Cut

Dans cette étape, la racine p coupe virtuellement les liens vers ses fils dans le but de diminuer son degré. Le message est simplement $< cut, k, p >$.

BFS

Chaque nœud x recevant le message précédent, démarre une vague BFS sur ses fils par un message $< BFS, k, p, x >$. Dans la figure 4.2, les flèches en pointillé

représentent des messages BFS qui se croisent, signe que la vague a rencontré une arête interne au fragment mais non incluse dans l'AC.

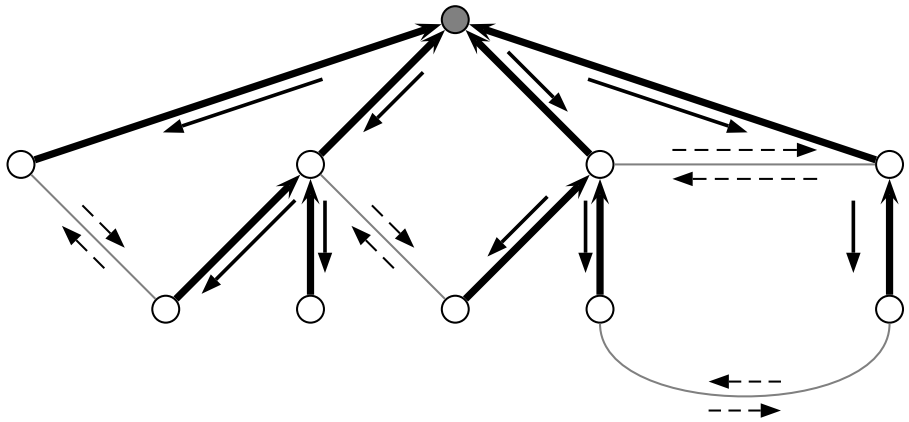

FIGURE 4.2 : Vague BFS

procédure BFS

– Un nœud x recevant un message $< BFS, k, p, p' >$ de son père, note son identité de fragment (p, p') et ensuite diffuse ce message à tous ses voisins excepté son père. Si des réponses sont en attente d'émission, les réponses sont envoyées suivant les règles suivantes :

– Lorsqu'un nœud x reçoit un message $< BFS, k, r, r' >$ d'un voisin y (il ne peut venir d'un fils) Trois cas sont possibles :

1. x n'a pas déjà été touché par une vague BFS de son père alors la réponse doit attendre que x apprenne son identité de fragment.

2. x a déjà reçu le message $< BFS, k, p, p' >$ de son père et $(r, r') < (p, p')$ (c'est à dire $r < p$ ou $(p = r$ et $r' < p')$). Dans ce cas, x peut répondre immédiatement à y par un message $< BFSBack, r, r', deg(x), () >$ (les parenthèses vides de la fin du message sont utilisées pour porter une identification d'arête potentiellement intéressante de fils à père).

3. x a déjà reçu le message $< BFS, k, p, p' >$ de son père mais $(r, r') > (p, p')$. Dans ce cas, x émettra ou a déjà émis un message de type BFS à y et alors il peut ignorer celui-là.

procédure BFS-Back

– Lorsqu'un nœud x reçoit une réponse de tous ses voisins :

– x choisit une arête sortante parmi les arêtes récoltées dans les messages « BFSBack » (une par message). Ces arêtes sont susceptibles de diminuer le degré de p. Remarquons que les nœuds de degré $k-1$ ne peuvent être pris en compte car ils ne permettraient pas la diminution du degré de l'AC.
– x émet un message « BFS-Back » avec l'arête choisie (parfois aucune) à son père (voir la figure 4.2, les flêches en gras représentent le message BFS-Back)

– Cette procédure est répétée jusqu'à revenir au nœud initiateur de la vague BFS.
– Ce nœud émet un message à p avec l'arête choisie.

Choose

S'il n'y a plus d'arête sortante candidate, c'est que la construction est terminée et que le degré de l'arbre est (localement) minimal. Dans le cas contraire, p choisit une arête parmi celles récupérées dans l'étape précédente. Le fils qui lui a envoyé la meilleur arête sortante est alors supprimé de la liste de ses fils et la meilleure arête sortante est ajoutée à l'arbre pour relier les deux sous-arbres.

Plus d'un nœud de degré maximum

L'AC peut inclure plus d'un nœud de degré maximum k. Dans ce cas, lorsque la vague BFS rencontre un de ces nœuds x, il se comporte alors comme une racine. A la fin d'une boucle, ce nœud peut se trouver dans un des deux cas suivants :

– Il existe aucune arête sortante. Cela signifie que le degré maximum de cet AC ne peut être amélioré, ce nœud émet un message « stop » à ses parents. Ce message est propagé jusqu'à la vraie racine.
– Sinon, il existe deux types d'arêtes sortantes, la première relie deux éléments du sous-arbre issu de x et la seconde relie deux sous-arbres de racines différentes x et y (il faut alors faire attention de ne pas choisir cette arête par ces deux noeuds de degré k). Dans le cas de deux racines différentes (donc avec des identités de fragments différents), on choisit celle d'identité inférieur : elle aura le droit de proposer cette arête pour l'échange.
La racine d'identité supérieure peut aussi profiter de la connaissance d'une éventuelle deuxième meilleure arête sortante e' si elle existe.
Un message « update » est émis vers le fils qui améliore le sous-arbre et le message « BFS-Back, e » est envoyé à son père. e est soit l'arête sortante dans le cas du premier type ou si x est d'identité inférieure à y. Si x est d'identité supérieure à y, $e = e'$ si e' existe, sinon le message « BFS-Back » est vide.

4.3 Analyse

Dans [FuRa92], le problème a été traité en séquentiel et leur principal résultat est énoncé dans le théorème suivant :

Théorème 5 *Soit T un AC de degré k d'un graphe G. Soit Δ^* le degré d'un AC de degré minimum. Soit S l'ensemble des nœuds de degré k. Soit B un sous-ensemble arbitraire des nœuds de degré $k-1$ dans T. $S \cup B$ supprimé du graphe, casse l'arbre T en une forêt F. Supposons que G satisfasse la condition telle qu'il n'y a pas d'arêtes entre les arbres de F. Alors $k \leq \Delta^* + 1$.*

Lorsqu'il n'y a pas d'arêtes entre les arbres de F, l'arbre T trouvé à ce point de l'algorithme est appelé un Arbre Localement Optimal.

La preuve de notre algorithme découle de ce théorème : la vague BFS, démarrée depuis les voisins de chaque nœud de degré k, trouve toutes les arêtes entre les arbres de F. Les arêtes sortantes des nœuds de degré $k-1$ sont simplement ignorées.

4.3.1 Analyse de Complexité

Dans cet algorithme tous les messages sont de taille en $O(\log n)$ où n le nombre de nœuds (au plus 4 identités par message).

Une des définitions classique de la complexité temporelle est la longueur de la plus longue chaîne de dépendance causale. Cette définition signifie que chaque transmission de message est au plus égale à une unité de temps.

Nous allons d'abord calculer, étape par étape, le nombre de messages échangés.

SearchDegree Puisque nous avons déjà un AC, nous l'utilisons pour la recherche donc seulement $n-1$ messages sont nécessaires.

MoveRoot Au plus $n-1$ messages sont utilisés pour déplacer la racine (dans le pire des cas le ST est une chaîne).

Cut Nous allons l'inclure pour le calcul avec l'étape suivante.

BFS Une vague BFS doit couvrir l'ensemble du graphe. Chaque arêtes verra passer au plus deux messages : un pour le BFS (ou CUT) et un pour le BFS-Back (voir la figure 4.2). Donc dans cette étape (cumulée à la précédente) au plus $2m$ messages sont émis.

Choose p dois envoyer un message vers une des extrémités de l'arête sortante choisie. Au plus $n-1$ messages sont utilisés pour tous les nœuds de degré maximal.

En tout, $O(m+n) = O(m)$ messages sont émis durant une boucle avec une complexité temporelle en $O(n)$.

Le nombre d'itérations dépends du degré de l'AC initial, soit k ce degré et soit k^* le degré de son Arbre Localement Optimal (ALO) correspondant. Il y a

$k - k^* + 1$ boucles. La valeur maximale de k est égale à $n - 1$ (lorsque l'AC est une étoile) et la valeur minimale de k^* est 2 (l'ALO est une chaîne). Ceci donne dans le pire des cas $O(mn)$ messages échangés. Dans les perspectives de recherche, nous pouvons tenter de changer un peu l'algorithme de construction de l'AC initial pour obtenir (par heuristique) une valeur de k pas trop éloignée de k^*.

4.4 Conclusion

Nous avons présenté dans ce chapitre le premier algorithme approché, distribué, de construction d'un ACDegM et qui fonctionne sur un graphe quelconque.

Il utilise la construction d'un AC comme étape initiale et échange un nombre de messages qui est relativement raisonnable. En effet, il nécessite $O(mn)$ messages (de taille $O(\log n)$) et il a été prouvé qu'un algorithme ne peut échanger moins de $O(\frac{n^2}{k})$ pour construire un arbre couvrant de degré au plus k [KoMZ87].

Chapitre 5

Conclusion de la deuxième partie

Dans cette partie, nous avons considéré trois types de contraintes pour la construction d'arbres couvrants : poids total minimal, diamètre minimal et degré minimal. Pour chacune de ces contraintes, nous avons proposé un algorithme distribué asynchrone permettant de résoudre ces problèmes sur des graphes quelconques.

Chacun de ces algorithmes a fait l'objet de publications internationales.

Les algorithmes de construction d'arbre couvrant de diamètre minimal et de degré minimal sont des algorithmes pionniers dans ce domaine de recherche. Les complexités temporelles et en nombre de messages échangés sont proches de l'optimal pour un au moins de ces paramètres.

Notons que le cumul des contraintes précédentes est envisageable par algorithme d'approximation, notamment, le problème de l'AC de degré minimal et de poids total minimal a été traité très récemment par [LaVa07].

Divers développement sont encore possibles, la prise en compte des pannes par exemple.

Troisième partie

Ordonnancement distribué

Chapitre 1

Introduction à l'ordonnancement distribué

Les applications modernes sont de plus en plus exigeantes en termes de temps de calcul, que l'on évoque les calculs de prédiction météorologique ou autre simulation numérique d'importance. Ces calculs ont poussé les chercheurs à paralléliser les programmes et les données pour répartir la charge sur les moyens de calcul. L'exploitation du parallélisme entraîne de multiples difficultés de gestion. Une représentation classique des applications parallèles (ou parallélisées) est faite en tâches (entités élémentaires) indépendantes ou en graphes de dépendances de tâches où l'on représente le fait qu'une tâche ne peut être exécutée qu'après la terminaison de ses tâches prédécesseurs dans le graphe. Très généralement, l'ordonnancement et le placement des tâches composant ces applications parallèles sur une architecture matérielle sont souvent envisagés que d'un point de vue centralisé (même dans les architectures et technologies de type grille).

Dans un contexte réaliste, l'architecture matérielle n'est même pas forcément régulière ni de taille connue. Nous allons tenter de nous approcher le plus possible de cette situation en distribuant l'ordonnanceur.

Cette partie est divisée en trois chapitres. Le premier décrit le contexte et la problématique de l'ordonnancement, ses caractéristiques, définitions et notations tout en précisant quel est notre cadre de travail.

Le deuxième chapitre traire de l'ordonnancement distribué de tâches indépendantes. nous montrons que les techniques classiques de l'ordonnancement séquentiel peuvent être appliquées moyennant quelques aménagements. Nous proposons de plus dans ce chapitre une nouvelle heuristique d'ordonnancement de tâches à arrivées sporadiques complètement distribuée sur un réseau quelconque.

Le troisième chapitre de cette partie concerne l'ordonnancement distribué de

tâches avec graphe de dépendances valué par les durées des communications inter-tâches. Nous présentons dans ce chapitre un nouvel algorithme qui fonctionne sur un graphe quelconque de taille arbitrairement grande avec des arrivées sporadiques de graphes de tâches.

1.1 Contexte et motivations de l'ordonnancement distribué

Pour les calculs intensifs que nous avons déjà évoqués, les architectures et technologies en grille sont très en vogue, elles sont basées sur une connaissance *a priori* des besoins, ce qui permet de gérer ces besoins par des mécanismes de réservations (en terme de temps et de nombre de processeurs) et priorités. Dans un contexte de réservations des besoins, il est évident alors qu'une structure avec gestion centralisée est plus simple à mettre en oeuvre, mais est-elle vraiment réaliste ? Une dérive possible peut être de surréserver les besoins pour assurer la fin du calcul dans de bonnes conditions. Il faut alors envisager des mécanismes de coût associé à l'usage, pour contraindre les utilisateurs. Nous pensons qu'il est possible de distribuer la gestion des tâches, c'est à dire leur ordonnancement.

L'ordonnancement consiste à attribuer une tâche à un processeur avec une date d'exécution dite au plus tôt suivant certains critères (temps-réel, temps total d'exécution, équilibrage de charge, etc.). Toujours dans une tentative d'aller vers plus de réalisme, nous pensons que cet ordonnancement ne peut être fait *a priori* comme proposé dans la plupart des articles sur le sujet. Au contraire, de nombreuses applications génèrent de nouvelles tâches qui elles-mêmes peuvent appeler d'autres — imposer une connaissance assez précise du nombre, de la fréquence des tâches, de leur durée d'exécution etc. nous semble illusoire. Il faut pouvoir s'adapter à l'émergence de nouvelles tâches pas forcément indépendantes.

La qualité d'un ordonnancement, voire sa validité pour les systèmes temps-réel, se mesure en termes de temps total d'exécution de l'application ou encore en termes de taux d'acceptation et exécution des tâches avant leurs échéances. Le nombre de processeurs utilisés mais aussi le temps de calcul de l'ordonnancement lui-même sont des paramètres fondamentaux dans la recherche d'implémentations parallèles efficaces.

Dans cette partie de la thèse, nous nous sommes intéressés aux problèmes d'ordonnancement distribués. Pour finir ce chapitre nous allons rappeler quelques hypothèses et modèles classiques d'ordonnancement pour nous situer par rapport à ce vaste domaine de recherche. Les deux chapitres suivants seront dédiés à la présentation de nos heuristiques d'ordonnancement distribué de tâches indépendantes puis dépendantes.

1.2 Notations et définitions

Nous présentons dans cette section les notations et les définitions de base utilisées dans cette partie du document. Dans ce qui suit, les termes nœud, sommet et tâche sont considérés comme équivalents.

1.2.1 Tâches et Graphes de tâches

– Un **graphe de tâches** (Graphe de flots de données) est la modélisation d'une application devant s'exécuter sur des machines à mémoire distribuée. Un graphe de tâches $\mathcal{G}$ est un graphe acyclique orienté et étiqueté défini par un triplet $(\mathcal{V}, \mathcal{E}, \mathcal{W})$. Tout élément de $\mathcal{E}$ est une paire d'éléments de $\mathcal{V}$. $\mathcal{V}$ est appelé l'ensemble des sommets et $\mathcal{E}$ l'ensemble des arcs. Chaque sommet représente une tâche. Il y a un arc de la tâche t_i à la tâche t_j s'il y a une dépendance entre t_i et t_j (les dépendances sont induites par les échanges de données entre les tâches). Cela signifie que la tâche t_j doit être exécutée après la fin de la tâche t_i.
– $\mathcal{W}(t_i, t_j)$ (le **poids d'un arc**) est la durée de communication entre t_i et t_j. Elle est considérée comme nulle lors de l'exécution si les deux tâches sont exécutées par le même processeur.
– La notation précédente est étendue au **poids des sommets** (tâches), $\mathcal{W}(t)$ est la durée d'exécution de la tâche t,
– Le **poids d'un chemin** est défini comme la somme des arcs de ce chemin à laquelle on ajoute la somme des poids des sommets de ce chemin.
– **Prédécesseur** : Dans un graphe de tâches orienté, une tâche t est un prédécesseur d'une tâche t' si et seulement si, il existe un chemin allant de t à t'. t est un prédécesseur immédiat de t', si et seulement si $(t, t') \in \mathcal{E}$.
– **Successeur** : Dans un graphe de tâches orienté, une tâche t est un successeur d'une tâche t' si et seulement si, il existe un chemin allant de t' à t. t est un successeur immédiat de t', si et seulement si $(t', t) \in \mathcal{E}$.
– Nous notons $\Gamma^-(t)$, l'ensemble des tâches prédécesseurs immédiats de t et $\Gamma^+(t)$ l'ensemble des tâches successeurs de t.
– Une **tâche d'entrée** est une tâche sans prédécesseurs.
– Une **tâche de sortie** est une tâche sans successeurs.
– Une tâche t d'un graphe $\mathcal{G}$ a deux niveaux :
  – **Niveau supérieur** $L^-(t)$ (*top level*) : Il est défini comme étant le poids du chemin le plus long depuis une tâche d'entrée quelconque dans le graphe $\mathcal{G}$ jusqu'à la tâche t, la durée d'exécution de t incluse.
  – **Niveau inférieur** $L^+(t)$ (*bottom level*) : Il est défini comme étant le poids du chemin le plus long depuis la tâche t, sa durée d'exécution incluse jusqu'à une tâche de sortie quelconque du graphe $\mathcal{G}$.

– **Granularité** : La granularité $g(\mathcal{G})$ d'un graphe de tâches $\mathcal{G}$ (dit aussi *Grain* du graphe), est définie comme suit (définition issue de [GeYa93]) :

$$g(\mathcal{G}) = \min_{i=1..v} \left(\min \left\{ \frac{\mathcal{W}(t_i)}{\max\limits_{t_j \in \Gamma^-(t_i)} W(t_j, t_i)}, \frac{\mathcal{W}(t_i)}{\max\limits_{t_k \in \Gamma^+(t_i)} \mathcal{W}(t_i, t_k)} \right\} \right)$$

Si $g(G) \geq 1$, le graphe de tâches est dit à gros grain, sinon, il est dit à grain fin. Clairement, pour un graphe à gros grain, les durées d'exécution des tâches sont plus importantes que les communications.

1.2.2 Processeurs et Architectures multiprocesseurs

– Nous distinguons trois types d'architectures multiprocesseurs (ou *plates-formes*).

– Processeurs **identiques** (*identical*) : dans un système parallèle, les processeurs sont dits identiques lorsqu'ils ont la même puissance de calcul, plus précisément, la durée d'exécution d'une tâche est la même sur chaque processeur.

– Processeurs **uniformes** (*uniform*) : les processeurs sont dits uniformes lorsqu'ils diffèrent les uns des autres par la vitesse mais de façon constante, plus précisément si un processeur est deux fois plus rapide qu'un autre, il en sera ainsi pour toutes les tâches que l'on pourrait lui faire exécuter.

– Processeurs **indépendants** (*related*) : les processeurs sont dits indépendants lorsque la durée du calcul d'une tâche peut varier d'un processeur à l'autre et cela de façon indépendante d'une tâche à l'autre.

– Une **grappe** (*cluster*) de tâches est un graphe de dépendance de tâches (ou sous-graphe) qui doit être exécuté sur un même processeur.

– Le **regroupement** (*clustering*) des tâches consiste à regrouper un ensemble de tâches dans un cluster dans le but d'économiser les coûts de communication entre ces tâches.

– La **réplication** d'une tâche est une technique qui consiste à exécuter cette même tâche par plusieurs processeurs dans le but d'économiser les communications ou à des fins de tolérance aux pannes.

– Le **surplus** (*idle time*) I_k du site k est la somme des durées des périodes d'inactivité d'un processeur sur une fenêtre temporelle donnée. Il est également appelé *temps d'oisiveté*.

– L'**efficacité** est définie comme étant le taux d'occupation processeur, plus précisément la somme des durées d'occupation divisée par le temps total d'observation (fenêtre temporelle).

– Le **Makespan** est le temps total d'exécution d'un graphe de tâches ordonnancé, c'est à dire la date de fin d'exécution de la dernière tâche.

1.2.3 Ordonnancement

Ordonnancer un graphe de tâches consiste à assigner pour chaque tâche du graphe un processeur d'exécution et une date de début tout en vérifiant les contraintes suivantes :

– A un instant donné, un processeur ne peut exécuter au plus qu'une tâche
– L'exécution d'une tâche ne peut commencer qu'après l'achèvement des communications issues de ses prédécesseurs immédiats.

Pendant le processus d'ordonnancement nous distinguons trois états possibles pour une tâche :

– **Ordonnancée** : la tâche est affectée à un processeur.
– **Libre** : tous les prédécesseurs immédiats de cette tâche sont ordonnancés.
– **Partiellement libre** : au moins un prédécesseur immédiat de cette tâche n'est pas ordonnancé.

1.2.4 Modèle d'exécution

Le modèle de base d'exécution pour les graphes de tâches est appelé *macro data flow*. Il s'agit d'un modèle d'exécution dirigé par les données où chaque tâche suit le protocole suivant :

– avant de s'exécuter sur son processeur la tâche reçoit toutes les données nécessaires à ses calculs,
– la tâche s'exécute séquentiellement et sans interruption,
– elle transmet ensuite les données aux tâches successeurs immédiats.

1.3 Brefs rappels sur les contextes d'ordonnancement

1.3.1 En ligne (On-Line) ou déconnecté (Off-line)

Dans la problématique de l'ordonnancement, il existe deux approches principales : les techniques/heuristiques dites en ligne (On-line) qui caractérisent le fait que les décisions d'ordonnancement doivent être faites à l'arrivée des tâches (à la volée) et les techniques/algorithmes déconnectés (off-line) qui travaillent sur un scénario d'arrivée de tâches prédéfini. Cette dernière approche permet parfois un calcul exact de l'ordonnancement optimal. L'ensemble des tâches à calculer est généralement au moins évalué par un test dit de satisfiabilité dans les systèmes temps-réel (voir section suivante).

Nous nous intéressons dans ce travail uniquement à l'approche en ligne, nous n'avons pas de connaissance *a priori* des tâches ou de leur date d'arrivée. Ce contexte est évidemment le plus difficile et l'on ne peut alors prétendre obtenir un

algorithme optimal. Par conséquent, nous nous intéressons à des heuristiques de calcul. Ces heuristiques doivent toutefois être « raisonnables » en temps d'exécution (c.-à-d. ne pas avoir d'impact sur les échéances). En effet, les cas d'application concrets que nous considérons sont souvent temps-réel.

1.3.2 Temps-réel

Les systèmes informatiques temps-réel sont des systèmes pour lesquels l'exactitude du résultat ne suffit pas, il faut aussi respecter les délais impartis. Ces systèmes sont souvent liés à des processus industriels (systèmes de contrôle de procédé, telles les chaînes de montage) ou encore au traitement d'informations qui ne pourrait supporter d'interruption (audio, vidéo).

Pour garantir le respect des contraintes temporelles, il est nécessaire que les différents services et processus s'exécutant sur un processeur s'exécutent en un temps borné. De plus, les différents enchaînements possibles des traitements doivent respecter leurs limites temporelles. On parle alors de test de satisfiabilité (voir la section suivante).

Temps-réel strict ou souple (hard/soft real-time) On distingue le temps-réel strict du temps-réel souple suivant l'importance accordée aux contraintes temporelles. Le temps-réel strict ne tolère aucun dépassement de contrainte. *A contrario*, le temps-réel souple peut tolérer quelques dépassements dans certaines limites au delà desquelles le système ne peut plus être considéré comme utilisable.

1.3.3 Test de satisfiabilité

Le test de satisfiabilité est utilisé dans les systèmes temps-réel strict, pour vérifier qu'aucun dépassement des contraintes temporelles ne pourra survenir. Cette vérification, appelée aussi analyse de faisabilité ou encore contrôle d'admission se fait généralement off-line sur un scénario d'arrivée de tâches.

Dans un contexte temps-réel strict on-line, le test de satisfiabilité est réduit à accepter ou non la tâche (ou le graphe de dépendance de tâches) qui survient. Ce test est alors lié intimement à l'heuristique d'ordonnancement utilisée et doit donc s'exécuter dans un temps « raisonnable ». Pour simplifier le raisonnement et les heuristiques, il est d'usage de considérer en fait un coprocesseur chargé uniquement du travail d'ordonnancement pour ne pas interférer avec le traitement proprement dit des tâches. Mais même avec ce coprocesseur si le temps d'exécution de l'heuristique n'est pas à peu près linéaire en fonction de la taille du graphe de tâches, le respect des échéances peut être compromis.

1.3.4 Préemptif ou non-préemptif

Dans un système d'exploitation, l'ordonnanceur se charge de répartir le temps de calcul d'un processeur entre les différentes tâches. Un ordonnanceur est dit préemptif s'il peut interrompre une tâche pour en faire exécuter une autre (avec souvent un autre contexte : variables d'environnement, descripteurs de fichiers ouverts, etc.).

Une tâche qui ne peut être interrompue (tâche liée au noyau du système d'exploitation par exemple) est dite non-préemptible. Par extension on parle d'ordonnancement non-préemptif si l'on ne s'autorise pas à interrompre des tâches dont l'exécution a commencé. Évidemment, le cas de l'ordonnancement non-préemptif est plus complexe à traiter car il s'apparente alors à des problèmes d'optimisation combinatoire en nombre entiers.

1.4 Notre cadre de travail

Nous nous intéressons dans cette partie du mémoire qu'au modèle on-line temps-réel non-préemptif sur architecture multiprocesseurs (que l'on supposera identiques dans un premier temps pour simplifier l'exposé, mais l'extension au modèle multiprocesseurs uniforme est possible). Nous ne nous autorisons pas la réplication de tâches.

Dans ce cadre, le problème de l'ordonnancement est NP-difficile [GaJo79], néanmoins c'est ce modèle qui nous semble le plus proche de la réalité. Nous nous efforçons donc de trouver des heuristiques. Dans la section suivante nous présentons un rapide tour d'horizon de quelques résultats proches dans ce contexte ou dans des contextes relativement proches.

Il existe de nombreuses variantes du problème de l'ordonnancement, mais nous nous limiterons à l'objectif qui consiste à placer judicieusement les différentes tâches composant l'application sur l'architecture cible (les processeurs) afin de minimiser le Makespan.

1.5 Quelques résultats sur l'ordonnancement multiprocesseurs temps-réel

Il existe principalement deux catégories d'heuristiques : Les algorithmes de liste et les algorithmes basés sur le chemin critique.

1.5.1 Algorithmes à gestion de liste

Voir par exemple [HCAL89, ReLe90, YaGe93, SiLe93, MeGh94, Sorel94, KwAh95].

Les heuristiques à gestion de liste les plus connues sont : ETF (*Earliest Time First*) [HCAL89], DLS (*Dynamic Level Scheduling*) [SiLe93] et l'algorithme de Sorel appelé σ [Sorel94]. Cette catégorie d'algorithmes consiste à établir une liste de tâches ordonnées selon une mesure de priorité. La liste peut être statique ou dynamique. Elle est dite statique si elle est inchangée pendant le processus d'ordonnancement et elle dite dynamique si les priorités des tâches non encore ordonnancées peuvent varier d'une étape à une autre pendant l'ordonnancement du graphe de tâches. Un mauvais choix dans la gestion des priorités peut retarder l'exécution des tâches les plus urgentes.

Les algorithmes à gestion de liste sont appelés également *BNP* (*Bounded Number of Processors*).

Par ailleurs, les auteurs de l'algorithme ETF ont donné une fonction de garantie des algorithmes de liste qui est une extension du résultat de Graham :

$$\mathcal{M} \leq \left(2 - \frac{1}{P}\right) \mathcal{M}_{opt} + \max \left\{ \sum_{(t_i,t_j)\in\pi} W(t_i, t_j) \right\}$$

Où, $\mathcal{M}_{opt}$ est le Makespan minimum du problème sans délais de communication, P est le nombre de processeurs et π est un chemin du graphe de dépendance.

1.6 Algorithmes basés sur le chemin critique

Citons entre autres [GeYa92, GeYa94, KwAh95, Kwok96, LaSo93, Sarkar89] et notre algorithme [HaBu05a].

Le chemin critique d'un graphe de tâches (DAG) est défini comme le chemin de poids maximal d'une tâche d'entrée à une tâche de sortie. Rappelons que ce poids inclut la durée des tâches (poids des nœuds) et les durées des communications (poids des arcs). Dans cette catégorie, on parle de problème de regroupement (clustering) qui consiste à former des groupes de tâches qui seront exécutées sur un même processeur. L'idée de ces algorithmes est de réaliser d'abord un partitionnement du graphe de tâches en clusters disjoints. Puis, chaque cluster est ordonnancé sur un processeur. Si le nombre de processeurs est borné et si le nombre de clusters est supérieur au nombre de processeurs disponibles, un autre ordonnancement est alors effectué jusqu'à avoir un nombre de clusters égal au nombre de processeurs.

La première heuristique qui a été proposée dans cette catégorie est celle de Sarkar [Sarkar89]. Il considère le processus de regroupement comme une phase préliminaire de l'ordonnancement sur un nombre borné de processeurs. Cette phase est appelée *Internalisation Prepass*. L'idée de cette heuristique est de regrouper de manière itérative les tâches qui donnent lieu à la communication la plus importante (ceci est appelé *edge zeroing*) jusqu'à ce qu'un tel regroupement

n'améliore plus la longueur partielle du regroupement.

Kwok et Ishfaq ont proposé un algorithme appelé DCP (*Dynamic Critical Path*) [Kwok96]. L'idée de cet algorithme est que les dates de début des tâches ne sont pas fixées tant que les tâches ne sont pas placées sur les processeurs. Cela permet d'insérer les tâches examinés dans les étapes ultérieures entre les tâches déjà examinées. La sélection processeur pour une tâche donnée est effectuée en regardant les dates de début des successeurs critiques de cette tâche sur le processeur. Les tâches candidates à l'ordonnancement sont celles qui sont sur le chemin critique. Elles sont identifiées en vérifiant l'égalité des deux paramètres : AEST(*Absolute Earliest Start Time*) et ALST (*Absolute Latest Start Time*). Ces paramètres sont calculés pour chaque tâche en traversant le graphe en entier à chaque étape du processus d'ordonnancement. Cette opération de calcul coûte $O(v^2)$ pour les v tâches du graphe. À la différence de DCP, notre algorithme de regroupement, présenté dans [HaBu06], peut calculer la longueur du chemin critique du graphe partiellement ordonnancé de façon incrémentale, sans reparcourir le graphe dans son intégralité.

À notre connaissance, l'algorithme qui était considéré comme le meilleur à ce jour est DSC (*Dominant Sequence Clustering*) de Gerazoulis et Yang [GeYa94]. Le principe de cet algorithme est inspiré de celui de Sarkar. À chaque étape du processus d'ordonnancement, il fusionne deux groupes de tâches de la séquence dominante. La séquence dominante représente le chemin le plus long du graphe partiellement ordonnancé. Cette heuristique à l'avantage d'avoir une faible complexité temporelle ($O((e+v)\log v)$) et une bonne performance surtout pour les graphes à gros grain.

Cette catégorie est connue également sous le nom UNP (*Unbounded Number of Processors*). Une taxonomie des algorithmes appartenant à cette catégorie peut être trouvée dans [GeYa92].

1.7 Comparatif de quelques algorithmes distribués d'ordonnancement de tâches indépendantes

1.7.1 Structure d'un ordonnanceur distribué

Nous distinguons deux composants principaux pour l'ordonnancement réparti :

1. L'ordonnanceur local à un site, qui est chargé de l'allocation du processeur selon une certaine politique prédéfinie. C'est lui qui prend la décision finale d'accepter ou de refuser d'exécuter localement une tâche.

2. L'ordonnanceur global qui gère l'interface avec l'ordonnancement local, et qui a pour rôle, en cas de surcharge (tâches refusées localement), de choisir l'emplacement d'exécution des tâches. Il assure quatre fonctions principales :

(a) Échange des informations sur la charge des sites
(b) Initiation de la phase de migration
(c) Choix des tâches à faire migrer
(d) Localisation des sites d'exécution.

1.7.2 Principales approches de résolution

1.7.2.a L'approche non coopérative

Ce type d'approche ne prend pas en compte l'état courant du système. Elle ne possède donc pas de module d'information : ce sont des méthodes aveugles. Principalement ce sont des méthodes aléatoires ou de type tourniquet (*round-robin*). Nous ne détaillerons pas plus avant ces techniques dans ce mémoire.

1.7.2.b L'approche coopérative

Compte tenu de l'absence de mémoire partagée des sites du réseau, le contrôle est pris en charge par la coopération des sites de ce réseau. Chaque site prend ses propres décisions au vu des informations recueillies.

Les algorithmes présentés dans cette section, sont des algorithmes d'ordonnancement classiques qui ont été adaptés pour deux raisons :

La première raison est que dans les systèmes sans contraintes de temps, l'état d'un processeur est mesuré par un facteur de charge généralement égal au nombre de tâches sur ce processeur. Ce facteur de mesure peut dans certaines situations ne pas être valable car un processeur ne disposant que très peu de tâches n'est pas forcément peu chargé. Nous préférerons donc la notion de surplus définie précédemment. Il s'agit pour chaque site de calculer la valeur de son surplus et de le communiquer. Grâce à la notion de surplus, le site ayant des tâches non garanties, dispose d'une estimation sur la capacité des autres sites à garantir ses tâches refusées.

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.

Approches à l'initiative de l'émetteur

L'algorithme du meilleur surplus (*focused addressing*) L'algorithme se base sur les surplus diffusés régulièrement pour le choix des sites pouvant garantir les tâches refusées. Une tâche refusée t_r est envoyée au site $\mathcal{S}_{dest}$ pour lequel la

fonction G$arantie(t_r, \mathcal{S}_{dest})$ est la plus élevée. Dans [RaSZ89, StRC85] un paramètre système GM (Garantie Minimale), est fixé en dessous duquel aucun site n'est sélectionné et l'algorithme ne peut être appliqué. Une fois que t_r est reçue par $\mathcal{S}_{dest}$, un test de garantie est effectué. Si la tâche ne peut être ordonnancée, elle est rejetée. Ce cas de figure peut avoir lieu dans le cas ou un certain temps s'écoule entre le dernier surplus envoyé par $\mathcal{S}_{dest}$ à $\mathcal{S}_{ref}$ et la réception de t_r. Donc l'information n'est plus à jour.

Pour limiter les risques de rejet d'une tâche par le site choisi, il est nécessaire que le module d'information garde une forte cohérence des surplus. Ainsi pour maintenir un degré de cohérence élevé, on optera pour une fréquence d'échange des surplus très élevée. Cependant cette solution risque de devenir très coûteuse en communications (ne peut être raisonnablement supportée que par un réseau à diffusion, ce qui n'est pas le cas de figure qui nous intéresse).

L'algorithme des enchères (*bidding*) L'idée développée dans [RaSZ89, StRC85], est basé sur l'approche de la vente aux enchères afin de prendre les décisions concernant le transfert. Un site qui ne peut assurer l'ordonnancement d'une tâche sporadique t_r, diffuse un appel d'offre aux autres sites et attend une offre sur la garantie qu'ils proposent pour t_r. Le site choisi est alors celui qui propose la meilleure enchère. Le message de demande d'enchères a la structure suivante :

$$< \ APPEL;\ \mathcal{P}t(t_r);\ \mathcal{S}_{ref} >$$

Où $\mathcal{S}_{ref}$ précise les paramètres $\mathcal{P}t$ de la tâche et l'identificateur du site auquel il faut renvoyer la réponse. L'offre donnée par les sites invoqués est de la forme suivante :

$$< \ ENCHERE;\ t_r;\ \text{G}arantie(t_r, \mathcal{S}_i)\ >$$

Par rapport à l'algorithme du meilleur surplus, la garantie est calculée par les sites destinataires à l'instant de la demande. Ceci implique un temps d'attente de(s) réponse(s) (on peut attendre que les k premières etc.) qui peut conduire au rejet de la tâche mais elle donne une information de meilleur qualité/fraicheur. Un compromis doit être trouvé entre le nombre de sites participants à l'offre afin de réduire le coût des communications et la garantie supplémentaire fournie par cette technique.

L'algorithme mixte Dans le but d'augmenter le nombre de tâches garanties dans le système, des algorithme mixtes ont été proposés [BRPM01, RaSZ89, StRC85] et le notre [HaBu04].
Principe : A priori l'algorithme du meilleur surplus est appliqué, en envoyant la tâche sur un site foyer (site qui donne la meilleure garantie), s'il existe. Parallèlement, et dans un but préventif, l'algorithme des enchères est exécuté. Les offres

sont retournées au site foyer, puisque la tâche lui a été transférée. Si ce site ne peut garantir la tâche (test de garantie négatif) il peut encore retransmettre la tâche au site qui a la meilleure offre sans avoir à procéder à une nouvelle élection.

Vu l'anticipation dans l'envoi de la tâche, ce type d'algorithme a un gain non négligeable, surtout pour les applications temps réel. Il permet dans le cas de tâches dont l'échéance est très proche, d'avoir plus de chance de trouver un site puisqu'un site d'exécution est choisi immédiatement tandis que de meilleures allocations sont recherchées en parallèle.

Approches à l'initiative du récepteur L'idée de ces méthodes est de faire en sorte que les sites peu chargés proposent leur aide aux sites surchargés.

L'algorithme d'interrogation [ChLi86] Cet algorithme est basé sur l'équilibrage des charges. Chaque tâche est caractérisée par un état qui peut prendre les valeurs suivantes :

- *Garantie* : la tâche est garantie localement,
- *Critique* : la tâche ne sera garantie qu'en cas de transfert,
- *En retard* : l'échéance est dépassée.

Les auteurs proposent de faire une « proposition de service » vers tous les sites uniquement si le site fournisseur de service est dans l'état *sous-chargé* par rapport à un certain seuil s_{min} basé sur le nombre de tâches dans la file d'attente de l'ordonnanceur. Un site est dans l'état surchargé si le nombre de tâches qui sont dans la file d'attente dépasse un second seuil s_{max}, ou s'il y a au moins une tâche dans l'état critique ou en retard.

Les auteurs justifient l'idée de l'algorithme par une sorte d'anticipation des effets d'avalanche qui se produisent en cas d'alarmes où plusieurs tâches sporadiques doivent être exécutées rapidement ; ils pensent les prévenir en équilibrant la charge.

Nous pensons que cette solution appliquée toute seule est insuffisante pour garantir d'éventuelles tâches temps réels (surtout lorsque l'échéance est proche), en effet un site doit patienter jusqu'à ce qu'il y ait un site sous-chargé pour que ses tâches refusées soient prises en compte. De plus, dans un système temps réel, nous estimons insuffisant de se contenter du nombre de tâches en attente pour mesurer la charge d'un site.

L'algorithme de diffusion d'états [ShCh89] Cet algorithme cherche à minimiser les temps de gestion occasionnés par la coopération des sites en diffusant l'information de changement d'état. L'idée est de se restreindre à une vue partielle du système pour chaque site. Chaque site conserve l'état d'une dizaine (en fonction de la taille du système) de ses voisins. Ensuite l'algorithme procède par une diffusion des états dans chaque groupe.

Les auteurs évaluent la charge des sites par une technique de seuils là encore par rapport à la taille de la file d'attente de l'ordonnanceur pour moduler les conditions de transfert.

1.8 Conclusion sur les algorithmes distribués d'ordonnancement

Les algorithmes de l'approche coopérative utilisent la connaissance de l'état des autres sites grâce à une diffusion totale ou partielle de cette information. Ils coopèrent tous pour répartir la charge du système soit de manière directe ou indirecte. La mesure de charge utilisée par ces approches est à notre avis, loin d'être satisfaisante pour des systèmes temps réel. Nous optons, si de telles stratégies sont appliquées, pour une mesure de charge basée sur le taux d'utilisation du processeur. De plus, les résultats expérimentaux présentés par les divers auteurs concernent des systèmes où le nombre de processeurs est faible (un maximum de dix), ce qui ne renseigne pas sur les délais qui peuvent être introduits par les communications et surtout par la migration des tâches. L'une des difficultés introduites par l'ordonnancement réparti est l'adéquation nécessaire entre le maintien de la connaissance de l'état global du système pour prendre de meilleures décisions et la minimisation des coûts de communications inter-sites. Donc si on veut améliorer la connaissance globale, on augmente d'autant le nombre de communications, il faut trouver des compromis. Nous avons proposé deux heuristiques sur architecture multiprocesseurs [HaBu04, HaBu05a]. Ces heuristiques ont des temps d'exécution de meilleure complexité ($O(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$ au lieu de $O((|\mathcal{E}| + |\mathcal{V}|) \log |\mathcal{V}|)$), tout en ayant en moyenne, en pratique, des Makespans similaires [Hakem06].

Chapitre 2

Ordonnancement distribué de tâches indépendantes

Nous présentons dans ce chapitre, un des algorithmes issu des travaux réalisés avec M. Mourad Hakem dans le cadre de sa thèse de doctorat soutenue en décembre 2006 [Hakem06]. Pour plus de détails sur cet algorithme, le lecteur pourra consulter [HaBu04].

Résumé

Dans ce chapitre, nous présentons un nouvel algorithme en ligne pour l'ordonnancement de tâches apériodiques sur un système distribué temps-réel. De nombreux algorithmes distribués ont été proposés pour ordonnancer des tâches avec échéances. Ces algorithmes considèrent, pour un site donné, le temps d'oisiveté (ou surplus) obtenu des autres sites du réseau comme une mesure pour transférer des tâches lorsque leur date au plus tard ne peut être respectée localement (ou quand la charge locale semble trop forte). Nous montrons qu'il est préférable, pour ces algorithmes, de prendre en compte aussi les distances entre sites pour limiter le nombre de tâches avec dépassement d'échéance.

Notre algorithme a deux points forts : il prend en compte le calcul des distances et il utilise une nouvelle politique de gestion des surplus pour minimiser la surcharge du réseau lors des diffusions de surplus.

2.1 Introduction

L'ordonnancement des tâches dans un univers réparti est un problème de contrôle qui, compte tenu de l'absence de mémoire commune, peut être pris en

charge par la coopération des processus communicants situés sur chaque site actif. Plusieurs algorithmes d'ordonnancement dynamique en ligne (ordonnancement global) ont vu le jour. L'impossibilité d'avoir un état global cohérent d'un système totalement reparti asynchrone fait que la plupart des algorithmes proposés ne permettent d'avoir une solution optimale, seules des heuristiques peuvent être utilisées. Tous ces algorithmes se basent sur l'hypothèse suivante : l'ordonnancement des tâches périodiques est toujours assuré localement, mais l'arrivée d'une tâche apériodique peut provoquer une phase de migration si elle ne peut être garantie.

Cette étude diffère des travaux précédents de part la prise en compte des distances dans le réseau ainsi que des estimations de surplus des autres sites du réseau.

L'information dite de surplus est composée de la somme des temps d'oisiveté dans la fenêtre temporelle d'observation. Elle est bon indicateur de la capacité d'un site à satisfaire l'échéance d'exécution d'une tâche arrivant sur ce site.

2.2 État de l'art

Pour un environnement distribué, multiprocesseur, l'ordonnancement est un problème NP-difficile [GLLR79]. La nature faiblement couplée des systèmes distribués rend le problème encore plus difficile. Clairement, dans un cadre réaliste, seules des heuristiques peuvent être considérées.

Le problème de l'ordonnancement est évidemment un problème important dans la conception de systèmes distribués. Cependant, la plupart des travaux sur l'ordonnancement de tâches avec contrainte temps-réel sont réduits à l'utilisation de l'information de surplus, provenant des autres sites du réseau, pour choisir vers quel site envoyer une tâche dont on ne peut satisfaire l'échéance [ChLi86, ShSi91, ZhRa85]. Le but de ces approches est de maximiser le nombre de tâches garanties avant leur échéance. Malheureusement, elles n'utilisent que l'information de surplus, la connaissance d'un plus court chemin du receveur vers le site destinataire est pourtant d'un avantage immédiat.

2.3 Modèle

L'architecture type dans un système distribué faiblement couplé consiste en n sites interconnectés par un réseau de communication. Le réseau sera vu comme un graphe $G = (V, E)$ avec $|V| = n$ sites $|E| = m$ liens bidirectionnels entre ces sites. Nous ne supposons pas la préexistence d'un protocole de communication sous-jacent. Voir en partie I, le chapitre 1 pour les références de la théorie des graphes et en partie III, le chapitre 1 pour les questions concernant l'ordonnancement.

Rappelons que nous sommes dans un modèle on-line temps-réel non-préemptif sur architecture multiprocesseurs identiques.

La charge du système est composée de groupes de tâches périodiques et apériodiques indépendantes. Une tâche apériodique T est caractérisée par le quadruplet (a_T, l_T, e_T, d_T), où a_T est sa date d'arrivée, l_T est sa date d'exécution au plus tôt (**date de début**), e_T représente sa durée d'exécution et d_T est son échéance (date au plus tard de fin d'exécution).

Les tâches apériodiques ne sont pas connues *a priori*, leurs caractéristiques ne sont connues que lors de leur arrivée.

Les tâches périodiques sont celles liées à la gestion du système lui-même et ont supposera leurs caractéristiques connues *a priori* ainsi que leur site d'affectation. Elles sont supposées prioritaires.

Puisque le but de ce travail est d'évaluer la performance des composants non locaux d'un système d'ordonnancement distribué, nous supposerons par exemple que l'ordonnanceur local est de type EDF *Earliest Deadline First* [ChCh89, ScZh92]. Ce choix est justifié par la nécessité d'avoir un maximum de flexibilité dans l'ordonnancement. Le lecteur intéressé pourra consulter [StRC85] pour la fonction de garantie.

A contrario des précédents travaux dans le domaine de l'ordonnancement distribué temps-réel, le modèle que nous utilisons dans cette étude inclut le problème du calcul des plus courts chemins. Ce calcul est basé sur les hypothèses faites en partie I, chapitre 2, section 2.5.

2.4 Schéma de l'ordonnanceur

Chaque site dans le système distribué a un ordonnanceur. Chaque ordonnanceur consiste en cinq processus de gestion plus un test de garantie, comme suit :

2.4.1 Éléments constitutifs de l'ordonnanceur

1. L'ordonnanceur local qui reçoit les tâches apériodiques et qui décide si la nouvelle tâche peut être exécutée localement (test EDF local). Si la tâche ne peut être garantie localement, la tâche est soit rejetée soit transmise au processus d'enchères.

2. Le répartiteur *dispatcher* qui détermine si la prochaine tâche à exécuter est une tâche périodique ou apériodique.

3. Le module d'enchères qui sert à déterminer vers quel site envoyer une tâche qui ne peut être garantie localement. Il est invoqué soit soit par une requête locale ou externe au site.

4. Le processus d'initialisation de tâches périodiques : il est exécuté une seule fois lors de l'initialisation du système. Un ensemble (éventuellement vide) de

tâches périodiques existe sur chaque site. Ce processus doit vérifier que la capacité de calcul locale est au moins suffisante pour assurer l'exécution de ces tâches, voir [LiLa73].

5. Le processus de calcul des plus courts chemins. Il est exécuté lors de l'initialisation du système. Nous utilisons celui que nous avons conçu pour [BuLa98] et qui est décrit sommairement dans la partie II, chapitre 3, section 3.6.4.

2.4.2 Ordonnancement de tâches

On suppose les étapes d'initialisations effectuées (calcul des plus courts chemin et test de garantie des tâches périodiques).

Quand une tâche T, arrive en un site x, l'ordonnanceur local est appelé. Si la tâche peut être garantie localement, elle est placée dans le plan d'exécution. Dans le cas contraire, l'ordonnanceur doit interagir avec les ordonnanceurs des autres sites pour trouver un site, le plus proche possible et de surplus suffisant, vers lequel envoyer la tâche.

Cette interaction est basée sur une méthode qui combine l'algorithme des enchères et l'adressage ciblé (algorithme , voir chapitre précédent).

2.4.3 L'algorithme mixte

Pour déterminer si un site est susceptible d'être un candidat pour l'adressage ciblé, nous effectuons le calcul de qualité suivant pour tous les sites y autres que x, site qui a reçu la tâche :

$Q(T, y) = Surplus_y - d(x, y)$ où $Surplus_y$ représente la connaissance disponible (d'après les messages émis par y et reçus par x) du surplus de y et $d(x, y)$ est la distance de x à y telle que calculée par l'algorithme des plus courts chemins.

Les $Q(T, y)$ sont triés par ordre décroissant ce qui donne un ordre sur les sites y. Soit k le nombre de sites y tels que $S_D(T, y) * (d_T - a_T) > F.e_T$ (F est un paramètre ajustable du système).

Si $k > 0$ alors la tâche T est immédiatement envoyée vers le site y^* de meilleure qualité. Ce site sera la cible de l'adressage ciblé. En parallèle, x envoie un appel d'offre (enchère) aux $k - 1$ autres sites au cas où T ne pourrait finalement pas être garantie en y^*. Dans cet appel d'offre, il est précisé que les réponses doivent être envoyées à y^*.

Si $k = 0$, alors l'algorithme se réduit à l'algorithme d'enchères qui suit :

2.4.4 L'algorithme d'enchères

Si l'algorithme précédent échoue à trouver un site cible pour l'adressage ciblé, l'algorithme d'enchères est lancé :

Le site x récepteur de la tâche T diffuse un message d'appel d'offre avec son adresse d'émetteur et les caractéristiques de T. Un site y recevant un tel appel d'offre, calcule une enchère qui va préciser la possibilité de garantir localement T. Ensuite, il émet cette enchère (en rappelant les caractéristiques de T) vers le site émetteur.

Lorsqu'un site reçoit une tâche que lui a fait suivre x, il tente de la garantir localement, si c'est le cas, il ignore les enchères qui lui parviendront pour cette tâche. Dans le cas contraire, il renverra la tâche vers le site proposant le meilleur rapport enchère/distance. Il est possible qu'aucune offre n'arrive dans un délai raisonnable, dans ce cas la tâche est finalement rejetée.

2.5 Conclusion et autres résultats sur l'ordonnancement

Dans ce chapitre, nous avons présenté une nouvelle heuristique d'ordonnancement complètement distribuée. Cette heuristique est simple, a un très bon comportement en pratique et est dynamique dans le sens qu'elle cherche à maximiser le nombre de tâches apériodiques temps-réel acceptées par le système, tout en satisfaisant des tâches périodiques critiques qui doivent être toujours exécutées avant leurs échéances. Un des aspects particuliers de notre technique est l'utilisation des distances en plus des surplus pour limiter le nombre de tâches n'arrivant pas à échéance. A notre connaissance, cet algorithme est le premier à prendre en compte cette notion, pourtant essentielle.

Une perspective intéressante est l'intégration d'une certaine forme de tolérance aux pannes, avec Mourad Hakem, nous avons commencé une série de travaux sur ce problème dit bicritère dans [HaBu06b, HaBu07a].

Dans le cadre de l'ordonnancement multiprocesseurs nous avons aussi quelques résultats [HaBu05a, HaBu05b, HaBu06].

Nous nous sommes aussi intéressés à des tâches avec dépendances, c'est le propos du chapitre suivant.

Chapitre 3

Ordonnancement distribué de tâches avec dépendances

3.1 Introduction

L'approche présentée dans le chapitre précédent, ainsi que dans la plupart des articles, se placent dans le cadre d'hypothèses de travail très simplificatrices car ils ne traitent que de tâches *indépendantes* et ils supposent que les coûts de *migration* des tâches sont négligeables. Le problème de prise en compte des *contraintes de précédence* en univers réparti est très mal résolu à l'heure actuelle. En effet, il faut connaître la localisation de toutes les tâches du graphe de précédence et vérifier l'ordonnancement en tenant compte des délais de communication induits par les tâches, ce qui amène à construire un graphe de localisation.

Nous présentons dans ce chapitre, une partie des travaux réalisés avec MM. Lucian Finta et Mourad Hakem sur une approche répartie d'ordonnancement en ligne des graphes de tâches temps-réel sporadiques (pour plus de détails voir [BuFH07]).

Un graphe de tâches ($\mathcal{G}$) est modélisé par un graphe dirigé acyclique. Des graphes de tâches arrivent concurremment à tout moment et sur n'importe quel site du réseau. L'algorithme d'ordonnancement développé dans ce chapitre est basé sur un nouveau concept de calcul : les sphères, qui permet de déterminer les voisins potentiels qui peuvent coopérer et participer à l'exécution des graphes de tâches s'ils ne peuvent être garantis localement sur leurs sites d'origine. Les points forts de ce mécanisme sont :

– Permettre à l'algorithme d'être exécuté sur un réseau quelconque de taille arbitraire puisqu'il utilise un nombre limité de sites et de liens de communications pour l'exécution distribuée du graphe de tâches ;

– Permettre de calculer le test de faisabilité distribuée du graphe de tâches (qui peut déterminer si le graphe est ordonnançable ou pas) avant la migration des tâches.

On fait les hypothèses suivantes :

– Le réseau est quelconque et le graphe de liaisons inter-sites est connexe (en fait l'algorithme peut fonctionner sur chaque composante connexe).
– Les communications se font en point à point.
– Pour simplifier la présentation nous supposons que les sites sont identiques (mais avec identités différentes).
– Chaque site connaît les délais de communication vers ses voisins immédiats
– Chaque site peut recevoir sporadiquement des graphes de tâches
– Chaque site à son propre ordonnanceur
– Les tâches apériodiques peuvent migrer
– L'ordonnancement local suit la règle *Earliest Deadline First* ;
– Un graphe de tâches est dénoté par $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ où $\mathcal{V}$ est l'ensemble des tâches et $\mathcal{E}$ est l'ensemble des relations de précédences entre tâches. Chaque graphe $\mathcal{G}$ est caractérisé par son échéance $\mathcal{D}$, son temps d'arrivée $\mathcal{A}$ et $\mathcal{R}$ sa date d'exécution au plus tôt. Une tâche $t_i \in \mathcal{G}$ a une durée d'exécution notée $e(t_i)$.

3.2 Description de l'algorithme

Ci-dessous, nous présentons une description succincte des étapes principales de notre algorithme. Elle est donnée pour un site initiateur k.

– Construction de la Sphère de Calcul Potentielle (SCP) issue de k. Le calcul est effectué une seule fois à l'initialisation du système (avec l'hypothèse qu'aucune panne matérielle (lien/site) ne se produise pendant le processus d'ordonnancement).
– À la réception du graphe de tâches ($\mathcal{G}$)

1. Test d'acceptation pour vérifier si le DAG peut être garanti localement
Si le graphe ($\mathcal{G}$) est garanti
insérer les tâches du graphe dans la liste de l'ordonnanceur
Sinon faire les étapes suivantes.
2. Construction (calcul) de la Sphère de Calcul Disponible (SCD)
3. Construction des séquences par le Processus d'Assignation (*Mapper*)
4. Validation des séquences précédentes
5. Exécution des séquences après réception de la permutation calculée à l'étape précédente et des codes des tâches.

3.3 L'ordonnanceur local à un site

Lorsqu'un nouveau graphe de tâches arrive sur le site k, l'ordonnanceur local tente de garantir son exécution localement. Ce test consiste à vérifier si toutes les tâches du graphe peuvent être ordonnancées avant l'échéance $\mathcal{D}$ du graphe de tâches sans remettre en cause les tâches déjà acceptées. Si cela est possible, les tâches sont ordonnancées parmi les tâches déjà garanties. Sinon le processus de distribution du graphe de tâches dans la sphère est invoqué. Un test similaire est effectué pendant le processus de validation des séquences (voir section 3.10).

3.4 Sphère de Calcul Potentielle (SCP)

Si le graphe de tâches ne peut être garanti sur le site k, une tentative d'exécution du graphe par un ensemble de voisins potentiels de k appelé sphère de calcul potentielle (SCP) peut commencer. Les sites de la sphère sont choisis proches de k en terme de sauts (hops) et de délais de communication. Une Sphère de Calcul Potentielle est caractérisée par les propriétés suivantes :

- Les sites ont un seul plus court chemin vers k en terme de délai de communication
- Le diamètre de la sphère en terme de sauts est borné par une constante h et les délais de communication entre tout couple de sites de la sphère k sont bornés par un certain délai maximal.
- Tous les membres de la sphère connaissent un chemin vers tous les autres.

3.5 Algorithme de construction distribué de la SCP

La sphère de calcul potentielle (SCP) du site k est un ensemble de sites qui peuvent participer à l'exécution du graphe de tâches non garanti sur k. Elle est construite une seule fois à l'initialisation du système. Il est à noter que pour deux sites initiateurs k et j, il peut y avoir une intersection non vide de leurs SCP. Par conséquent, un site de cet ensemble d'intersection pourrait participer à l'exécution des graphes de tâches arrivant sur k et j.

Plusieurs algorithmes distribués classiques peuvent être adaptés pour la construction des SCPs. Nous présentons dans ce qui suit un algorithme de construction distribué des sphères basé sur [BeGa87]. Cet algorithme est interrompu avant terminaison pour limiter l'inondation du réseau. Nous donnons une brève description de cet algorithme puis nous montrons comment l'adapter.

- Conditions de départ : chaque site commence par émettre un vecteur de distances (délais) vers tous les sites voisins immédiats. Chaque site

maintient une table de routage qui a la forme <*destination, distance, site intermédiaire*>.

– Étape d'envoi : chaque site envoie périodiquement sa table de routage à tous ses voisins immédiats. Les mises à jour sont envoyées quand l'entrée « *destination* » des vecteurs de la table change.

– Étape réception : mise à jour des lignes de la table de routage avec les tuples reçus.

Nous construisons les SCPs en arrêtant l'algorithme précédent après un certain nombre de phases. D'abord, nous l'adaptons pour l'organiser en phases logiques. Le site k commence la construction de sa SCP en émettant sa table de routage à tous ses voisins. Une phase est composé de deux étapes : une étape d'envoi et une étape de réception de toutes les tables de routages de ses voisins immédiats. Ceci nous assure que chaque nouvelle phase effectue une mise à jour des distances à un saut (hop) de plus. Ainsi, si l'algorithme est arrêté après h phases, les distances calculées jusqu'ici par l'algorithme sur chaque site, représenteront les distances minimales exactes vers tous les sites (la distance = h *sauts*, représente le rayon de la sphère). En fait, l'algorithme est arrêté après $2h$ phases, pour que chaque site de la sphère de k puisse découvrir tous les autres sites de la même sphère y compris les chemins vers ces sites. Mais seuls les sites découverts dans les h premières phases sont pris en compte comme membres de la SCP.

À chaque étape, chaque site envoie à tous ses voisins un message, donc nous avons $2m$ messages échangés ($|E| = m$ est le nombre d'arêtes). La construction de la SCP s'effectue en $2h$ étapes. Par conséquent, le nombre de messages échangés est au plus $4mh$.

3.6 Construction de la Sphère de Calcul Disponible (SCD)

Quand un graphe de tâches ne peut être garanti sur le site d'origine k, la construction de la SCD peut commencer. Les membres de la SCD sont dynamiquement sélectionnés dans l'ensemble des sites appartenant à la SCP de k. Cette construction est assurée par un processus de *marquage/recrutement* qui utilise la table de routage de la SCP. L'ensemble des sites recrutés est appelé SCD.

Comme évoqué précédemment, un site peut appartenir à plusieurs SCPs. Par conséquent, il peut participer à l'exécution de plusieurs graphes de tâches en même temps. Le processus de recrutement utilise donc un mécanisme d'*exclusion mutuelle* : une variable verrou sur chaque site permet de s'assurer que les sites recrutés n'appartiennent qu'à la SCD de k pendant la phase de distribution du graphe de tâches.

Chaque site recruté envoie un message de confirmation, ainsi que son surplus

calculé, au site initiateur k. Les sites recrutés gèrent leurs variables verrous associées à l'initiateur k. Les messages de demande de recrutement reçus des autres sites sont ignorés jusqu'à réception d'un message de déverrouillage du site initiateur k.

3.7 Construction des séquences par le Processus d'Assignation

Le processus assignation (ou *Mapper*) dispose des entrées suivantes :

- le graphe de tâches sporadiques $\mathcal{G}$ à ordonnancer ;
- la liste des sites de la SCD triés par ordre décroissant de leurs surplus respectifs.

Le Mapper décide si le graphe de tâches $\mathcal{G}$ est rejeté ou, dans le cas contraire, construit le partitionnement. Le partitionnement M est un ensemble de trois applications :

- $S : T \rightarrow U$, tel que $U = 1, \ldots, |U|$ est l'ensemble des sites logiques ($S(t)$ est le site logique auquel tâche t est assignée) ;
- $r : T \rightarrow \mathcal{R}^+$, tel que $r(t)$ est la date d'exécution au plus tôt de la tâche t ;
- $d : T \rightarrow \mathcal{R}^+$, tel que $d(t)$ est la date d'échéance de t.

Le partitionnement M est construit par le Mapper en se basant sur les informations de la SCP et de la SCD : le surplus de chaque site $j \in$ SCD et les distances entre chaque couple de sites de la SCD.

3.8 Validation des séquences

Les séquences (partitionnement M) générées par le Mapper doivent être validées par les différents sites de la SCD, car le Mapper connaît uniquement les surplus des sites de la SCD mais pas les dates exactes de début et de fin des périodes d'oisiveté de chaque site.

Donc le partitionnement M est diffusé dans la SCD. À la réception de M, chaque site $j \in$ SCD tente de valider toutes les tâches assignées au site logique i pour chaque $i \in U$.

Soit T_i l'ensemble des tâches affectées au site logique i, $T_i = \{t \in T \,/\, S(t) = i\}$. L'ensemble T_i est localement satisfiable par l'ordonnanceur local si et seulement si chaque tâche $t \in T_i$ peut être exécutée dans sa fenêtre d'exécution respective $[r(t), d(t)]$. La liste des sites i, pour lesquels l'ensemble T_i est localement satisfiable est envoyée à l'initiateur k.

Quand l'initiateur k reçoit toutes les listes des sites de la SCD, il tente de réaliser un *couplage maximum* (problème classique de la théorie des graphes résolu en temps polynômial voir [Berg70]). Soit c un couplage, si $|c| < |U|$ alors

aucune combinaison ne satisfait l'ensemble des séquences T_i ($i \in U$) de M. Par conséquent, le graphe $\mathcal{G}$ est rejeté et les membres de la SCD sont libérés. Si en revanche un couplage maximum de cardinalité $|c| = |U|$ est trouvé, le graphe $\mathcal{G}$ est accepté et une permutation de l'ensemble des sites pouvant exécuter le graphe de tâches non garanti par l'initiateur k est calculée.

3.9 Exécution distribuée

L'initiateur k envoie la permutation calculée avec le code des tâches aux sites pouvant valider les séquences T_i ($i \in U$). Les sites non invoqués dans la permutation sont libérés.

Soit i l'identité du site logique assigné au site physique j. Le verrou de j est immédiatement libéré après avoir inséré toutes les tâches $t \in T_i$ dans son plan d'ordonnancement (*schedule*). Une fois que toutes les séquences du partitionnement M sont ordonnancées, l'exécution distribué du graphe de tâches peut commencer.

3.10 L'ordonnanceur local – Test d'ordonnançabilité

Le test de garantie local est invoqué pour vérifier si toutes les tâches du graphe sporadique $\mathcal{G}$, peuvent être exécutées localement avant l'échéance $\mathcal{D}$ de $\mathcal{G}$. Soit $\mathcal{L}$ la liste des tâches garanties sur le site initiateur k, $s(t_i)$ la date de début d'exécution d'une tâche $t_i \in \mathcal{G}$ et $f(t_i)$ la date de fin d'exécution de t_i est $f(t_i) = s(t_i) + e(t_i) \leq d(t_i)$. Deux cas de figures sont à prendre en compte :

1. $\mathcal{L} = \emptyset$: la solution la plus évidente pour le test est de vérifier l'inégalité :

$$\sum_{i=1}^{|T|} e(t_i) \leq \mathcal{D} - \nabla$$

2. $\mathcal{L} \neq \emptyset$: le test d'acceptation le plus adapté à notre modèle de tâches est de chercher pour chaque tâche du graphe $\mathcal{G}$ une période d'oisiveté suffisamment grande pour traiter la tâche dans son intégralité, sans préemption. Pour cela, le graphe est parcouru du bas vers le haut et les tâches sont testées avec un calcul des dates de début au plus tard. Nous supposons que $\mathcal{L} = \{t_{i1}, t_{i2}, \ldots, t_{iq}\}$ et que la liste des périodes d'oisiveté entre les tâches de $\mathcal{L}$ est :

$$[0, s(t_{i1})], [f(t_{i1}), s(t_{i2})], \ldots, [f(t_{i(q-1)}), s(t_{iq})], [f(t_{iq}), \infty[$$

En parallèle nous parcourons la liste des périodes d'oisiveté de droite à gauche à partir de l'instant $\mathcal{D}$ et pour chaque tâche $t_i \in \mathcal{G}$ sélectionnée (la sélection

est guidée par la politique EDF [LiLa73]), nous cherchons le premier interval de temps $[f(t_{ik}), s(t_{i(k+1)})]$ qui satisfasse l'inégalité suivante :

$$\min\left\{s(t_{i(k+1)}), d(t_i)\right\} - f(t_{ik}) \geq e(t_i)$$

Ainsi, la date de début au plus tard de t_i est donnée comme suit :

$$s(t_i) = \min\left\{s(t_{i(k+1)}), d(t_i)\right\} - e(t_i)$$

Où, l'échéance $d(t_i)$ de chaque tâche $t_i \in \mathcal{G}$ représente le minimum des dates de début au plus tard des tâches successeur. Elle est calculée comme suit :

$$\forall\ t_i \in T,\quad d(t_i) = \begin{cases} d & \text{si } \Gamma^+(t_i) = \phi \\ \min\limits_{t_j \in \Gamma^+(t_i)} \{s(t_j)\} & \text{sinon} \end{cases}$$

Si la date de début au plus tard de la dernière tâche est dans le futur ou le présent alors le graphe de tâches $\mathcal{G}$ est garanti. Sinon le graphe est rejeté et le processus de distribution du graphe sur la sphère de calcul peut commencer.

Remarque : Pour le cas préemptif, la *condition nécessaire et suffisante* d'acceptation du graphe G est $\sum\limits_{i=1}^{|T|} e(t_i) \leq \Omega(\mathcal{R}, \mathcal{D})$ où $\Omega(\mathcal{R}, \mathcal{D})$ est la durée totale du temps d'oisiveté du site entre les dates $\mathcal{R}$ et $\mathcal{D}$.

Test d'ordonnançabilité – Validation des séquences

À la réception du partitionnement $M = \left\{T_1, T_2, \ldots, T_{|U|}\right\}$, un test de garantie est effectué pour chaque séquence $T_j \in M$. Soit $\mathcal{P}$, le site recevant le partitionnement M, t_1^j la première tâche de la séquence T_j $(j = 1 \ldots |U|)$, $\mathcal{L} = \{t_{i1}, t_{i2}, \ldots, t_{iq}\}$ la liste des tâches déjà acceptées pour exécution sur $\mathcal{P}$. L'ensemble des périodes d'oisiveté entre les tâches de $\mathcal{L}$ sur $\mathcal{P}$ est :

$$[0, s(t_{i1})], [f(t_{i1}), s(t_{i2})], \ldots, [f(t_{i(q-1)}), s(t_{iq})], [f(t_{iq}), \infty[$$

Le test de garantie consiste à chercher pour chaque tâche $t_i \in T_j$ un interval de temps suffisamment grand pour traiter la tâche dans son intégralité, sans préemption. Pour cela, chaque séquence T_j $(j = 1 \ldots |U|)$ est parcourue de gauche à droite à partir de l'instant $r(t_1^j)$. Si toutes les tâches d'une séquence donnée peuvent être insérées, la séquence est considérée comme garantie sinon la séquence est rejetée.

La tâche t_i peut être insérée dans la période d'oisiveté qui satisfait l'inégalité suivante :

$$s(t_{i(k+1)}) - \max\{r(t_i), f(t_{ik})\} \geq e(t_i)$$

$s(t_i)$ satisfait les conditions suivantes : $(s(t_i) \geq r(t_i))$ et $(s(t_i) \geq f(t_{ik}))$
Par conséquent la date d'exécution au plus tôt est donnée par

$$s(t_i) = \max\{r(t_i), f(t_{ik})\}$$

Après la vérification de toutes les séquences de M, un vecteur de booléens de taille $|U|$ représentant les séquences garanties et non garanties sur le site $\mathcal{P}$ est envoyé au site initiateur k.

3.11 Conclusion

Nous avons présenté dans ce chapitre, un nouvel algorithme totalement distribué pour l'ordonnancement en ligne des graphes de tâches temps-réel sur des réseaux de taille arbitraire. Les graphes de tâches sont caractérisés par leurs échéances et ils arrivent concurremment sur n'importe quel site de manière sporadique. Le nouveau concept de calcul des sphères introduit dans ce chapitre a pour but, d'une part, de réduire les coûts des communications et de migration des tâches, et, d'autre part, de permettre à l'algorithme distribué d'utiliser un nombre limité de sites et de liens de communication.

Chapitre 4

Conclusion de la troisième partie

Nous avons présenté dans cette partie du mémoire nos résultats sur un domaine jusqu'alors peu exploré : l'ordonnancement distribué temps-réel en ligne. Ce contexte est évidemment le plus difficile et l'on ne peut alors prétendre obtenir un algorithme optimal. Par conséquent, nous nous sommes intéressés à des heuristiques. Les heuristiques que nous avons présenté sont raisonnables en temps de calcul, c'est à dire que dans le meilleur des cas la tâche ou le graphe de tâches est accepté immédiatement, et dans le pire des cas, la réponse est donnée comme réponse à un consultation d'un sous-ensemble des sites du graphe. En effet, les cas d'application concrets que nous considérons sont souvent temps-réel et l'inondation du graphe pour la recherche d'un site permettant l'acceptation est inutile. Nos heuristiques prennent en compte les distances, permettant de ne pas avoir d'échange de message à faire avec des sites trop éloignés qui n'auraient aucune chance de répondre dans l'échéance. Le nombre de sites consultés à chaque arrivée de tâche ou de groupe de tâches est d'ailleurs paramétrable.

Le sujet fut lancé pour le mémoire de DEA de Mourad Hakem qui a été suivi d'une thèse de doctorat [Hakem06] soutenue en décembre 2006 à l'université de Paris 13. Nous n'avons présenté ici que la partie distribuée des travaux dont les points de départ sont dans cette thèse [HaBu04, BuFH07]. D'autres travaux ont été effectués avec Mourad Hakem, notamment l'ordonnancement sur architectures multiprocesseurs, avec nombre de processeurs bornés [HaBu05b] ou non [HaBu05a, HaBu06] et l'ordonnancement bi-critère dans lequel on s'attache à prendre en compte non seulement l'aspect minimisation du Makespan mais aussi limiter la probabilité de panne [HaBu07a].

Développement logiciel

Durant ces années de recherche nous avons conçu un simulateur (EVADA pour EValuation d'Algorithmes Distribués Asynchrones) qui de par sa réalisation nous a apporté une expérience certaine dans la programmation de machines parallèles. Ce simulateur écrit en C (7000 lignes) permet de prendre en compte le modèle classique distribué à échange de messages mais aussi le modèle « link-register » et le modèle à mémoire partagée. Nous avons intégré dans ce simulateur un suivi des changements d'états des processus par matrice stochastique – qui peut être vu comme des chaînes de Markov, et a permis d'illustrer certains résultats, voir [Bui90a].

Il nous a aussi permis de mettre en évidence certains comportements à occurrences rares dans certains algorithmes distribués ainsi que des comportements en moyenne que l'analyse théorique n'avait pas révélés. En particulier des phénomènes de surcharge locale. Nous avons explicités ces comportements et apporté des solutions simples. Une fois codés dans un langage de description simple et intuitif, les algorithmes ont été simulés sur des graphes divers et nous ont permis d'établir des distinctions entre algorithmes de construction d'arbre couvrant établissant ainsi un classement suivant différents critères. Ce simulateur à aussi été utilisé intensivement pour la thèse de Félix Francisco Ramos Corchado (CINVESTAV, Mexique) soutenue à l'Université de Technologie de Compiègne.

Dans un tout autre domaine, la combinatoire, notamment la combinatoire algébrique, manipule de nombreux objets indexés par le groupe symétrique. De fait, les calculs sur ces objets sont très difficiles à réaliser et très lourds en termes de temps de calcul et de mémoire. Une première tentative réussie de distribution du calcul [NeTB05] nous incite à aller plus loin. En outre, en nous appuyant sur le programme SCHUR [Wybo], développé initialement par B.G. Wybourne (Professeur à Torun, Pologne, décédé) et maintenant par Frédéric Toumazet (Université Paris 13), Ronald C. King (Professeur émérite de l'Université de Southampton, Royaume-Uni) et moi-même, nous essayons de proposer des méthodes de calcul simples et rapides. Le programme interactif SCHUR est le fruit de plus de 20 ans de recherches, c'est une sorte de super calculatrice qui peut aider aussi bien

les étudiants que les chercheurs. En effet, il est à l'origine d'un grand nombre de conjectures sur les fonctions symétriques (certaines ont été prouvées depuis). Après avoir fait un gros travail sur le code (nettoyage des 50 000 lignes de code C sans commentaires, en tout plus de 160 commandes) nous l'avons désormais rendu disponible sur Sourceforge [1] en « open source » (GNU Public Licence) pour la communauté scientifique. Nous y avons apporté de nombreuses corrections et quelques commandes supplémentaires. Nous avons développé aussi une batterie de tests pour développeurs et nous le proposons sous forme de paquet prêt-à-installer (*package*) pour linux (RPM, deb) et pour Windows 98/XP.

1. http ://sourceforge.net/projects/schur

Bilan et perspectives

Nous avons présenté dans ce mémoire un extrait de plusieurs années de recherche dans le domaine de l'algorithmique distribuée, dans ce qu'elle a de plus original mais aussi de plus difficile à appréhender, son caractère *asynchrone* ou *asynchrone à délais bornés*.

Le concept d'algorithmique distribuée asynchrone conduit à la recherche de structures de contrôle sur la totalité du réseau. Nous nous sommes efforcés de définir un modèle précis et simple très peu exigeant en matière de connaissances globales : pas d'information structurelle *a priori* sur le réseau de communication (taille, degré, diamètre, topologie, etc.), pas d'horloge globale ni de mémoire partagée.

La taille d'un réseau et l'arbitraire de sa topologie conduisent à écarter des techniques de type routage point à point et anneau virtuel pour l'établissement de structure de contrôle. Nous avons donc présenté dans un premier temps des algorithmes de construction d'arbre couvrant ayant chacun des particularités intéressantes.

Une structure de contrôle distribuée se doit d'être efficace. Nous avons présenté trois contraintes de construction :

- le poids total minimal qui traduit plutôt une recherche économique (travail qui a été effectué dans le cadre du co-encadrement de la thèse de Lélia Blin).
- le diamètre minimal qui traduit la recherche d'efficacité aussi bien en messages qu'en temps lorsque les liens de communications sont valués par leurs délais de transmission exacts (ou du moins par un majorant de ces délais) et
- le degré minimal qui apporte un réponse à des besoins en termes de coût en équipements d'interconnexion et de résolution de goulot d'étranglement, entre autres.

Tous ces algorithmes sont nouveaux, voire pionniers de leurs domaines et nous avons par ailleurs proposé des versions très détaillées de ces algorithmes qui sont directement implémentables.

Utilisant nos connaissances sur l'algorithmique distribuée, il nous a paru inté-

ressant de lancer divers travaux sur un sujet de recherche qui semblait un peu négligé : l'ordonnancement distribué. Malgré un intérêt évident dans des domaines de recherche très en vue, tels les grilles de calculs, très peu de travaux étaient parus sur le sujet. Persuadés que le domaine valait une réelle exploration, nous avons proposé le sujet et encadré le mémoire de DEA puis la thèse de M. Mourad Hakem (encadrement à 99%). Ce travail a été fructueux : pas moins de quatre publications internationales ont confirmé l'intérêt de la communauté scientifique envers ce sujet.

Comme nous l'avons vu dans ce mémoire, la plupart de nos algorithmes sont des algorithmes pionniers. Dans le domaine de l'algorithmique distribuée, le problème de la recherche d'arbre couvrant, structure de contrôle de prédilection, a été source de nombreuses publications. Pourtant certaines de ses variantes contraintes, telles l'arbre couvrant de diamètre minimal ou celui de degré minimal n'étaient pas traitées dans un cadre aussi général que celui que nous nous sommes posé : pas de mémoire partagée, pas d'horloge globale, pas d'informations structurelles sur le graphe sous-jacent du réseau de communication. Pourtant, nous avons pu proposer des algorithmes de résolution simples et efficaces, certes pas optimaux mais permettant encore de nombreux développements.

En particulier nous souhaitons désormais nous intéresser à des aspects laissés (en partie) de côté ; ainsi, entre autres, toute la problématique de la tolérance aux fautes, en particulier l'auto-stabilisation, et de l'algorithmique probabiliste.

Nous avons commencé une démarche de suppression et de relaxation des hypothèses de fonctionnement. L'information structurelle *a priori* (taille du graphe, topologie, etc.) n'est plus directement pertinente (sauf en ce qui concerne la complexité des algorithmes) ; il nous faut plutôt orienter notre travail vers les caractéristiques des identités des processus. En effet, dans un environnement distribué asynchrone et *anonyme* – c.-à-d. les identités des sites ne sont pas nécessairement distinctes (ce qui équivaut à l'absence d'identité) –, il n'existe plus d'algorithme déterministe de construction d'un arbre couvrant (ou, plus généralement, de structure de contrôle couvrante). Créer des algorithmes probabilistes (ou randomisés) est alors la seule solution envisageable.

Une autre branche de cette recherche concerne les *réseaux sans fils*, réseaux mobiles, réseaux ad hoc (ou *Manet*) et réseaux radio ou de capteurs. Les réseaux sont alors modélisés par des graphes complètement dynamiques, soit du fait de la mobilité des sommets, soit du fait de la portée limitée des signaux (radios, optiques ou autres) en transmission/réception, soit du fait même des fautes inhérentes à ces modèles. Nous devrons confronter notre expérience à cette approche, certes différente, mais déjà très riche et dont l'intérêt va croissant.

Un dernier domaine de recherche extrêmement prometteur a trait à l'*analyse*

en moyenne des algorithmes distribués. Les analyses de complexité des algorithmes distribués se sont jusqu'ici principalement limitées au pire des cas. Leur « analyse perturbée » (*smoothed analysis* voir [BaBM03]) apporte une vision enrichie du problème. Il s'agit d'une mesure de complexité intermédiaire entre le pire des cas et la complexité en moyenne qui semble adéquate pour quantifier la « difficulté » d'un problème. Il paraît ainsi naturel de s'intéresser aux nombreux problèmes classés NP ou assimilés (d'optimisation et de combinatoire). En effet, être « worst-case-NP » est fort différent d'être « average-case-NP », lui-même fort différent d'être « average-case-NP » sur une région suffisamment grande.

Du côté de l'ordonnancement en ligne, nous avons ouvert un certain nombre de brèches avec la thèse de Mourad Hakem. Nous avons évoqué dans ce mémoire, l'aspect ordonnancement distribué de tâches indépendantes ou non. Ici encore, la tolérance aux pannes est encore à développer. Mais nous avons aussi par ailleurs développé d'autres heuristiques pour l'ordonnancement de graphes de tâches sur architectures multiprocesseurs qui s'avèrent très efficaces en pratique [HaBu05a, HaBu05b, HaBu06, HaBu07a]. Ces heuristiques sont pour certaines bicritères, prenant en compte non seulement le temps total d'exécution d'un graphe de tâches ordonnancé (*Makespan*) mais aussi les probabilités des pannes de sites.

Bibliographie

[Awer85c] Awerbuch (Baruch). – « Complexity of network synchronization ». *Journal of the ACM*, vol. 32, n° 4, 1985, pp. 804–823.

[Awer87] Awerbuch (Baruch). – « Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems ». *ACM Symposium on Theory Of Computing*, pp. 230–240. – 1987.

[BaBM03] Banderier (C.), Beier (R.) et Mehlhorn (K.). – « Smoothed analysis of three combinatorial algorithms ». *Proc. of Mathematical Fundations of Computer Science (MFCS'03)*. pp. 198–207. – Springer-Verlag, 2003.

[BeGa87] Bertsekas (Dimitri P.) et Gallager (Robert G.). – *Data networks*. – Prentice-Hall International, 1987.

[BeGa92] Bertsekas (Dimitri P.) et Gallager (Robert G.). – *Data networks*. – Prentice-Hall International, 1992, 2nd édition.

[Berg70] Berge (Claude). – *Graphes et Hypergraphes*. – Paris, Dunod, 1970, *Monographies universitaires de mathématiques*. English translation : Graphs and Hypergraphs (North-Holland, Amsterdam 1973).

[BlBu01] Blin (Lélia) et Butelle (Franck). – « A very fast (linear time) distributed algorithm, in general graphs, for the minimum spanning tree ». *Studia Informatica Universalis, Hors Série OPODIS'2001*, 2002, pp. 113–124.

[BlBu04] Blin (Lélia) et Butelle (Franck). – « The first approximated distributed algorithm for the minimum degree spanning tree problem on general graphs ». *Int. Jour. on Foundations of Computer Science*, vol. 15, n° 3, 2004, pp. 507–516.

[BRPM01] Bhattacharjee (Anup K.), Ravindranath (K.), Pal (A.) et Mall (R.). – « Ddsched : a distributed dynamic real-time scheduling algorithm ». *Progress in computer research*, 2001, pp. 170–184.

[BuBL04] Bui (Marc), Butelle (Franck) et Lavault (Christian). – « A distributed algorithm for the minimum diameter spanning tree problem ». *Journal of Parallel and Distributed Computing*, vol. 64, n° 5, 2004, pp. 571–577.

[BuBu93b] Bui (Marc) et Butelle (Franck). – « Minimum diameter spanning tree ». *OPOPAC, Int. Workshop on Principles of Parallel Computing*. pp. 37–46. – Hermès & Institut National de Recherche en Informatique et en Automatique, novembre 1993.

[BuFH07] Butelle (Franck), Finta (Lucian) et Hakem (Mourad). – « Real-time distributed scheduling of precedence graphs on arbitrary wide networks ». *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium International Parallel and Distributed Processing Symposium*. p. 220 (article sur CDROM). – IEEE Computer Society, 2007.

[Bui89] Bui (Marc). – *Etude Comportementale d'algorithmes distribués de contrôle*. – Thèse de doctorat, Université Paris XI, Orsay, 1989.

[Bui90a] Bui (Marc). – « On optimal management of resources in distributed networks ». *Optimization*, vol. 21, n° 5, 1990.

[BuLa98] Butelle (Franck) et Lavault (Christian). – « Distributed algorithms for all-pairs shortest paths and minimum diameter spanning tree problems ». *Distributed Computing : OPODIS'98 (Proceedings of the 2nd Int. Conf. On Principles Of DIstributed Systems)*. pp. 77–88. – Hermès, janvier 1999.

[BuLB95] Butelle (Franck), Lavault (Christian) et Bui (Marc). – *A Uniform Self-Stabilizing Minimum Diameter Spanning Tree Algorithm*. – RR n° 95-07, Institut Galilée, Université Paris Nord, avril 1995.

[BuLB95a] Butelle (Franck), Lavault (Christian) et Bui (Marc). – « A uniform self-stabilizing minimum diameter spanning tree algorithm ». *Proceeding of the 9th International Workshop on Distributed Algorithms*. pp. 257–272. – Springer-Verlag, 1995.

[Bute94] Butelle (Franck). – *Contribution à l'algorithmique distribuée de contrôle : arbres couvrants avec et sans containtes*. – Thèse de doctorat, Paris 8 – St Denis, mars 1994.

[CaGM80] Camerini (P. M.), Galbiati (G.) et Maffioli (F.). – « Complexity of spanning tree problems : Part I ». *European Journal of Operationnal Research*, vol. 5, 1980, pp. 346–352.

[ChCh89] Chetto (Houssine) et Chetto (Maryline). – « Some results of the earliest deadline scheduling algorithm ». *Proc. IEEE trans. on Soft. Eng.*, pp. 1261–1269. – october 1989.

[ChLi86] Chang (Hung-Yang) et Livny (Miron). – « Distributed scheduling under deadline constraints : a comparison of sender-initiated and receiver initiated approaches ». *Proc. IEEE Int. Conf. on Distr. Comp. Syst.*, pp. 175–180. – 1986.

[Chri75] Christophides (Nicos). – *Graph Theory : An algorithmic approach*. – Academic press, 1975, *Computer Science and Applied Mathematics*.

[ChTi85] Chin (Francis Y.) et Ting (H. F.). – « An almost linear time and $o(nlogn+e)$ messages distributed algorithm for minimum-weight spanning trees ». *IEEE Symposium on Foundations Of Computer Science*, pp. 257–266. – 1985.

[ChVe90] Chandrasekaran (S.) et Venkatesan (S.). – « A message optimal algorithm for distributed termination detection ». *Journal of Parallel and Distributed Computing*, vol. 8, n° 3, 1990, pp. 245–252.

[FaMo95] Faloutsos (Michalis) et Molle (Mart). – « Optimal distributed algorithm for minimum spanning trees revisited ». *Symposium on Principles of Distributed Computing*, pp. 231–237. – 1995.

[FMRT90] Fisher (M. J.), Moran (Shlomo), Rudich (S.) et Taubenfeld (G.). – « The wakeup problem ». *ACM Symposium on Theory Of Computing*, pp. 106–116. – 1990. (Extended abstract).

[FrGa95] Fraigniaud (P.) et Gavoille (C.). – *Memory requirement for universal routing schemes*. – RR n° 95-05, LIP, 1995.

[FrGS90] Froidevaux (Christine), Gaudel (Marie-Claude) et Soria (Michèle). – *Types de données et algorithmes*. – McGraw-Hill, 1990.

[FuRa92] Fürer (Martin) et Raghavachari (Balaji). – « Approximating the minimum degree spanning tree to within one from the optimal degree ». *ACM-SIAM Symposium on Discrete Algorithms*, pp. 317–324. – 1992.

[GaHS83] Gallager (Robert G.), Humblet (Pierre A.) et Spira (Paul M.). – « A distributed algorithm for minimum weight spanning trees ». *ACM Transactions on Programming Languages and Systems*, vol. 5, n° 1, 1983, pp. 66–77.

[GaJo79] Garey (Michael R.) et Johnson (David S.). – *Computers and intractability : A guide to the theory of NP-completeness*. – W. H. Freeman, 1979.

[GaKP98] Garay (Juan A.), Kutten (Shay) et Peleg (David). – « A sublinear time complexity distributed algorithm for minimum-weight spanning trees ». *SIAM Journal on Computing*, vol. 27, n° 1, 1998, pp. 302–316.

[GeYa92] Gerasoulis (Apostolos) et Yang (Tao). – « Pyrros : static schedulinng and code generation formessage passing multiprocessors ». *Proc. of the 6th International Conference on Supercomputing*, éd. par ACM, pp. 428–437. – 1992.

[GeYa93] Gerasoulis (Apostolos) et Yang (Tao). – « On the granularity and clustering of directed acyclic task graphs ». *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n° 6, 1993, pp. 686–701.

[GeYa94] Gerasoulis (Apostolos) et Yang (Tao). – « DSC : Scheduling parallel tasks on an unbounded number of processors ». *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, 1994, pp. 951–967.

[GLLR79] Graham (Ronald L.), Lawler (E. L.), Lenstra (J. K.) et Rinnooy Kan (A. H. G.). – « Optimization and approximation in deterministic sequencing and scheduling : A survey ». *Annals of Dis. Math.*, vol. 5, 1979.

[HaBu04] Hakem (Mourad) et Butelle (Franck). – « A new on-line scheduling algorithm for distributed real-time system ». *Proc. of the 3rd Int. Symp. and School on Advanced Distributed Systems*, pp. 241–251. – 2004.

[HaBu05a] Hakem (Mourad) et Butelle (Franck). – « Dynamic critical path scheduling parallel programs onto multiprocessors ». *Proc. of the 19th IEEE International Parallel & Distributed Processing Symposium (IPDPS'05 – APDCM'05)*. – 2005. CDROM, 10 pages.

[HaBu05b] Hakem (Mourad) et Butelle (Franck). – « Efficient critical task scheduling parallel programs on a bounded number of processors ». *Proc. of the 17th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'05 – IASTED)*. pp. 139–144. – Acta Press, 2005.

[HaBu06] Hakem (Mourad) et Butelle (Franck). – « Critical path scheduling parallel programs on an unbounded number of processors ». *Int. Jour. Foundations Comp. Sci.*, vol. 17, n° 2, april 2006, pp. 287–301.

[HaBu06b] Hakem (Mourad) et Butelle (Franck). – « A bicriteria scheduling algorithm for minimizing makespan and failure probability of applications in heterogeneous distributed computing systems ». *Rencontres Francophones du Parallélisme (RenPar'17)*. – 2006. A paraître.

[HaBu07a] Hakem (Mourad) et Butelle (Franck). – « Reliability and scheduling on systems subject to failures ». *Proc. of Int. Conf. on Parallel Programming (ICPP)*. – 2007. A paraï¿œtre.

[Hakem06] Hakem (Mourad). – *Ordonnancement multiprocesseurs et distribué temps-réel*. – Thèse de PhD, Université Paris 13, décembre 2006.

[HaPS78] Hakimi (S. L.), Pierce (J. G.) et Schmeichel (E. F.). – « On p-centers in networks ». *Transportation Sci.*, vol. 12, 1978, pp. 1–15.

[HCAL89] Hwang (Jing-Jang), Chow (Yuan-Chieh), Anger (Frank D.) et Lee (Chung-Yee). – « Scheduling precedence graphs in systems with interprocessor communication times ». *SIAM Journal on Computing*, vol. 18, n° 2, 1989, pp. 244–257.

[HeMR87] Hélary (Jean-Michel), Maddi (Aomar) et Raynal (Michel). – « Calcul réparti d'un extrémum et du routage associé dans un réseau quel-

conque ». *RAIRO Informatique théorique et applications*, vol. 21, n° 3, 1987, pp. 223–.

[HLCW91] Ho (J.-M.), Lee (D. T.), Chang (C.-H.) et Wong (C. K.). – « Minimum diameter spanning trees and related problems ». *SIAM Journal on Computing*, vol. 20, n° 5, octobre 1991, pp. 987–997.

[Hoar87] Hoare (Charles A. R.). – *Processus séquentiels communicants*. – Masson, 1987.

[IhRW91] Ihler (E.), Reich (G.) et Wildmayer (P.). – « On shortest networks for classes of points in the plane ». *Int. Workshop on Computational Geometry – Meth., Alg. and Appli.*, pp. 103–111. – mars 1991.

[ItRa94] Italiano (G. F.) et Ramaswani (R.). – « Maintaining spanning trees of small diameter ». *Proc. ICALP'94*, pp. 227–238. – 1994.

[KoKM90] Korach (Ephraïm), Kutten (Shay) et Moran (Shlomo). – « A modular technique for the design of efficient distributed leader finding algorithms ». *ACM Transactions on Programming Languages and Systems*, vol. 12, n° 1, 1990, pp. 84–101.

[KoMZ87] Korach (Ephraïm), Moran (Shlomo) et Zaks (Shmuel). – « The optimality of distributed constructions of minimum weight and degree restricted spanning trees in complete network of processors ». *SIAM Journal on Computing*, vol. 16, n° 2, 1987, pp. 231–236.

[KuPe98] Kutten (Shay) et Peleg (David). – « Fast distributed constructiono f small k-dominating sets and applications ». *Journal of Algorithms*, vol. 28, 1998, pp. 40–66.

[KwAh95] Kwok (Yu-Kwong) et Ahmad (Ishfaq). – « Bubble scheduling : A quasi dynamic algorithm for static allocation of tasks to parallel architectures ». *Proc. 7th IEEE Symp. on Parallel and Distr. Processing*, pp. 36–43. – 1995.

[Kwok96] Kwok (Yu-Kwong) et Ahmad (Ishfaq). – « Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors ». *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, n° 5, 1996, pp. 506–521.

[LaLa86] Lavallée (Ivan) et Lavault (Christian). – « Scheme for efficiency performance measures of distributed and parallel algorithms ». *Proc. of the 1st Int. Workshop on Distributed Algorithms, Distributed algorithms on graphs*, éd. par Gafni (E.) et Santoro (N.). pp. 69–103. – Carleton University Press, 1986.

[LaLa89a] Lavallée (Ivan) et Lavault (Christian). – *Yet another distributed election and (minimum-weight) spanning tree algorithm*. – RR n° 1024,

Institut National de Recherche en Informatique et en Automatique, 1989.

[Lamp78] Lamport (Leslie). – « Time, clocks and the ordering of events in a distributed system ». *Communications of the ACM*, vol. 21, n° 7, 1978, pp. 558–565.

[Lamp82] Lamport (Leslie). – « An assertional correctness proof of a distributed algorithm ». *Sci. Computer Programming*, vol. 2, 1982, pp. 175–206.

[LaSo93] Lavarenne (C.) et Sorel (Y.). – « Performance optimization of multiprocessor real-time applications by graphs transformations ». *Proc. Parallel Computing*. – Grenoble, 1993.

[LaVa07] Lavault (Christian) et Valencia-Pabon (Mario). – « A distributed approximation algorithm for the minimum degree minimum weight spanning trees ». *Journal of Parallel and Distributed Computing*, a paraître. – JPDC-06-72.

[Laval90] Lavallée (Ivan). – *Algorithmique parallèle et distribuée*. – Hermès, 1990.

[Lavau87] Lavault (Christian). – *Algorithmique et complexite distribuées - application*. – Thèse d'état, Univeristé Paris-Sud Orsay, 1987.

[Lavau95] Lavault (Christian). – *Evaluation des algorithmes distribués*. – Hermes, 1995.

[Lela77] Le Lann (Gérard). – « Distributed systems : towards a formal approach ». *Proc. of IFIP congress 77*. IFIP Congress, pp. 155–160. – décembre 1977.

[LiLa73] Liu (C. L.) et Layland (James W.). – « Scheduling algorithms for multiprogramming in a hard real-time environment ». *Journ. of the Assoc. for Comp. Machinery*, 1973, pp. 46–61.

[Matt87] Mattern (Friedemann). – « Algorithms for distributed termination detection ». *Distributed Computing*, vol. 2, n° 3, 1987, pp. 161–176.

[MeGh94] Neelima Mehdiratta (Kanad Ghose). – « A bottom-up approach to task scheduling in distributed memory multiprocessors ». *Proc. Int. Conf. on Parallel Processing*, pp. 151–154. – 1994.

[NeTB05] Nzeutchap (Janvier), Toumazet (Frédéric) et Butelle (Franck). – « Kostka numbers and littlewood – richardson coefficients : Distributed computation ». *Symmetry, Spectroscopy and Schur (Proc. of Prof. Brian G. Wybourne Commemorative Meeting)*. pp. 211–221. – Nicolaus Copernicus University Press, 2006.

[RaSZ89] Ramamritham (Krithi), Stankovic (John A.) et Zhao (Wei). – « Distributed schedulings of tasks with deadlines and resource requirements ». *Proc. IEEE trans. on Computers*, pp. 1110–1123. – august 1989.

[Rayn87] Raynal (Michel). – *Systèmes répartis et réseaux.* – Eyrolles, 1987.

[Rayn91] Raynal (Michel). – *La communication et le temps dans les réseaux et les sytèmes répartis.* – Eyrolles, 1991.

[ReLe90] Rewini (H. El) et G.Lewis (T.). – « Scheduling parallel program tasks onto arbitrary target machines ». *Journal of Parallel and Distributed Computing*, vol. 9, 1990, pp. 138–153.

[Sarkar89] Sarkar (V.). – *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors.* – MIT Press, 1989.

[ScZh92] Schwan (Karsten) et Zhou (Hongyi). – « Dynamic scheduling of hard real-time tasks and hard real-time threads ». *Proc. IEEE Trans. on Softw. Eng.*, pp. 736–747. – august 1992.

[Sega83] Segall (Adrian). – « Distributed network protocols ». *IEEE Transactions on Information Theory*, vol. IT-29, n° 1, janvier 1983, pp. 23–35.

[ShCh89] Shin (K. G.) et Chang (Y. C.). – « Load sharing in distributed real-time systems with state-change broadcasts ». *IEEE Trans. on Software Engineering*, vol. 38, n° 8, 1989, pp. 1124–1142.

[ShSi91] Shivaratri (Niranjan G.) et Singhal (Mukesh). – « A transfer policy for global scheduling algorithms to schedule tasks with deadline ». *Proc. IEEE Int. Conf. on Distr. Comp. Syst.*, pp. 248–255. – 1991.

[SiLe93] Sih (G.C.) et Lee (E.A.). – « A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures ». *IEEE Trans. on Parallel and Dist. Systems*, vol. 4, n° 2, 1993, pp. 75–87.

[Sorel94] Sorel (Yves). – « Massively parallel computing systems with real-time constraints : the "algorithm architecture adequation" ». *Proc. of Massively Parallel Comput. Syst., MPCS.* – 1994.

[StRC85] Stankovic (John A.), Ramamritham (Krithivasan) et Cheng (Shengchang). – « Evaluation of a flexible task scheduling algorithm for distributed hard real time systems ». *Proc. IEEE trans. on Computers*, pp. 1130–143. – december 1985.

[Tel91] Tel (G.). – *Topics in distributed computing.* – Thèse de PhD, Utrecht, 1991.

[Tel94] Tel (Gerard). – *Introduction to Distributed Algorithms.* – Cambridge University Press, 1994.

[Tel98] Weiss (G.) (édité par). – *Distributed Control for AI*, chap. Distributed Artificial Intelligence, Ch. 14. – MIT Press, 1998.

[TrNa87] Tréhel (Michel) et Naïmi (Mohammed). – « Un algorithme distribué d'exclusion mutuelle en $\log(n)$ ». *Technique et Science Informatiques*, vol. 6, n° 2, 1987, pp. 141–150.

[Wybo] Wybourne (Bryan G.). – Schur, an interactive programme for calculating properties of lie groups and symmetric functions. Manuel du logiciel SCHUR, 250pp.

[YaGe93] Yang (Tao) et Gerasoulis (Apostolos). – « List scheduling with and without communication delays ». *Parallel Computing*, vol. 19, n° 2, 1993, pp. 1321–1344.

[ZhRa85] Zhao (Wei) et Ramamritham (K.). – « Distributed scheduling using bidding and focussed adressing ». *Proc. IEEE Real-time Systems Symposium*, pp. 103–111. – december 1985.