Placement et Validation dans les Systèmes Temps Réel Distribués

M. Richard P. Richard F. Cottet
{richardm,richardp,cottet@ensma.fr}

Laboratoire d'Informatique Scientifique et Industrielle
École Nationale de Mécanique et d'Aérotechnique
Téléport 2 – BP 40109 F-86961 Chasseneuil Futuroscope Cedex, France

Résumé : Lorsque l'architecture matérielle est distribuée, l'ordonnancement des tâches est fortement dépendant du placement de celles-ci sur les différents processeurs. Nous considérons ici le problème du placement statique et de l'ordonnancement à priorités fixes. Nous proposons dans cet article une méthode de placement et d'affectation des priorités permettant la validation d'une application distribuée. Le placement des tâches et l'affectation des priorités aux tâches et aux messages sont réalisés simultanément.

Mots clés : Temps-réel, Placement, Ordonnancement à priorités fixes, Affectation des priorités, Système distribué.

1 Introduction et problématique

Lors de la conception des systèmes distribués temps réel, la structure logicielle est affectée sur l'architecture matérielle disponible. L'étape de validation temporelle est alors totalement dépendante du résultat de l'allocation des différentes fonctions aux calculateurs. La validation temporelle a fait l'objet de nombreuses études dans un contexte monoprocesseur. Dans le cas d'un système distribué, elle ne dépend plus uniquement des carcatéristiques des tâches, mais aussi de leur répartition sur les différents processeurs. Nous proposons une méthode de placement et d'ordonnancement des tâches telle que leurs spécifications temporelles soient respectées. L'originalité de notre méthode, vis-à-vis des travaux déjà effectués sur le sujet, est d'une part, d'effectuer le placement et l'ordonnancement simultanément ; et d'autre part, le cas échéant, d'établir l'inexistence d'une solution validable par une analyse holistique.

Nous présentons ci-dessous les architectures logicielles et matérielles étudiées ainsi que les hypothèses et les contraintes à considérer.

1.1 Architecture logicielle

Chaque site (i.e. processeur) présent dans l'application exécute un système d'exploitation temps réel [Ose97] ordonnançant les tâches à priorités fixes réalisant les fonctions de l'application. Chaque tâche τ_i est définie comme un quadruplet (C_i,D_i,T_i,π_i) où :

- C_i est le pire temps d'exécution,
- D_i est l'échéance relative au réveil de τ_i,
- T_i est la période des activations de τ_i,
- π_i est la priorité de τ_i, locale au processeur sur lequel elle s'exécute.

Si notre méthode supporte les tâches dont les échéances sont non reliées aux périodes, nous considérons dans la suite, afin de simplifier les formules, des tâches à échéances reliées aux périodes (i.e. $D_i < T_i$ et $D_i = T_i$).

Les tâches sont ordonnancées en fonction de leurs priorités par un algorithme d'ordonnancement préemptif fourni en standard dans le système d'exploitation. À chaque instant, la tâche possédant le niveau de priorité le plus important obtient le processeur. Nous rappelons que la priorité π_i d'une tâche τ_i est fixe durant toute la vie de l'application. La valeur 0 désigne le plus fort niveau de priorité. Notons enfin que les tâches sont toutes périodiques et activées au départ de l'application.

Nous faisons l'hypothèse que les réceptions des messages s'effectuent au début de l'exécution d'une tâche et les émissions s'opèrent à la fin de l'exécution

de celle-ci. Ceci implique qu'aucune communication ne peut avoir lieu dans le corps d'une tâche.

Les tâches peuvent communiquer et/ou se synchroniser au moyen de deux mécanismes. Si deux tâches s'exécutent sur le même processeur, la communication s'effectuent via la mémoire associée localement au processeur. Cette communication est alors modélisée par une contrainte de précédence. Dans le cas contraire, les deux tâches échangent des messages via le réseau. Notons qu'une tâche peut recevoir et envoyer plusieurs messages. De plus, un même message peut être émis vers plusieurs destinataires. Un message m_i est défini comme un triplet (n_i,D_i,T_i) où n_i est le nombre d'octets constituant le message, D_i l'échéance relative à la date d'envoi du message et T_i la période d'envoi du message.

Nous définissons les notations suivantes :

- J_i est la gigue sur l'activation de la tâche τ_i (différence entre l'activation et le réveil de τ_i) liée à une communication entrante s'effectuant sur le réseau,
- Tr_i est le pire temps de réponse de la tâche τ_i, c'est-à-dire la durée entre l'activation de la tâche et sa fin d'exécution,
- $[i]$ est le numéro de la tâche affectée au niveau de priorité i sur le processeur sur lequel elle s'exécute.
- Γ_i^- est l'ensemble des différents messages reçus par la tâche τ_i via un réseau et nécesaire à son exécution. Cet ensemble dépend du placement des tâches puisque si l'émetteur et le récepteur sont placés sur le même processeur, le message les liant ne transite pas par le réseau.
- $\prec$ dénote une relation d'ordre partiel entre deux tâches. Le graphe des communications est un graphe orienté acyclique,
- n est le nombre de tâches et de messages.

1.2 Architecture matérielle

Nous considérons des architectures multiprocesseurs sans mémoire partagée interconnectées par un ou plusieurs réseaux. Ce modèle d'architecture englobe le cas simple d'un unique processeur (i.e. contexte monoprocesseur). Chaque site dispose d'un processeur Pr_i et d'une mémoire locale autorisant les communications entre deux tâches s'exécutant sur le site. Deux tâches ordonnancées sur deux sites différents ne communiquent que par messages transitant sur le réseau. La Figure 1 présente un exemple d'architecture matérielle multi-réseaux supportée par notre méthode. Dans le cas d'architecture multi-réseaux, le site connecté aux deux réseaux (exemple du processeur Pr_6 sur la Figure 1) assure le rôle de passerelle entre ceux-ci.
Afin de pouvoir considérer les contraintes de placement, nous définissons ci-

dessous la notion de *pool de processeurs.*

Définition 1 *Un pool de processeurs Pl_i est défini par un couple (PR_i,Θ_i) où :*

- *PR_i est un ensemble de m_i processeurs identiques,*
- *Θ_i est un ensemble de n_i tâches à placer sur les processeurs $Pr_i \in PR_i$.*

Dans le cas d'un pool de processeurs Pl_i, l'ensemble tâches Θ_i est fixe (i.e. le nombre de tâches le composant est fixe et connu).

La figure 1 montre le découpage d'une architecture distribuée, composée de deux réseaux (CAN et VAN), et de trois pools de processeurs.

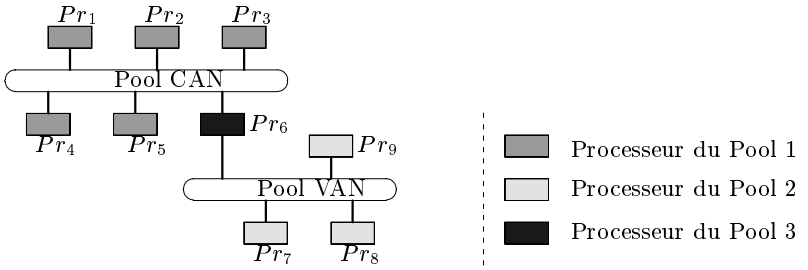


FIG. 1 – *Type d'architecture étudiée*

Les réseaux considérés dans la suite (Cf. Figure 1) sont les réseaux CAN (Controller Area Network) et VAN (Vehicle Area Network) [ISO94a, ISO94b], correspondant à la sous-couche MAC (Medium Access Control) et basés sur le protocole CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance). Ces deux réseaux ordonnancent les trames selon leurs priorités fixes. D'autres réseaux pourront être modélisés et intégrés dans la méthode.

1.3 Validation

Le nombre de messages, de contraintes de précédence, la charge des processeurs sont autant de paramètres dépendant de la superposition de l'architecture logicielle sur l'architecture physique disponible. Le respect des contraintes temporelles des tâches et messages de l'application est donc fonction du résultat de cette phase d'allocation. Les problèmes d'allocation des tâches dans un système distribué peuvent être classés en deux catégories selon le contexte d'étude [KA99, Bea96] : le problème du placement statique et le problème du placement dynamique.

Dans la littérature, les travaux sur le problème du placement statique sont très nombreux. Une classification rigoureuse et complète s'avère délicate. Ces travaux diffèrent à la fois par le type de problème traité et par les méthodes employées. Nous pouvons classer les problèmes en deux grandes catégories : les problèmes de validation [Ram95, Bea96, SMM99, AH97, TBW92, ML95, SFOC97, OCSF97] et les problèmes d'optimisation [BT83, SVC98, OS95, ST85, PSA97, JS97]. De façon générale, le problème d'allocation a été montré $\mathcal{NP}$-difficile [LW82, Law83].

Dans la suite, nous considérons le problème du placement statique. De plus, nous faisons l'hypothèse qu'une fois affectées à un processeur, toutes les instances d'une tâche τ_i s'exécutent sur ce même processeur. En d'autres termes, nous interdisons toute migration de tâche en cours d'exécution. La validation temporelle d'une application implique que toutes les instances de toutes les tâches et de tous les messages respectent leurs échéances.

Nous proposons une méthode effectuant conjointement l'allocation d'une tâche sur un processeur et l'affectation d'une priorité fixe à cette tâche. Cette technique permet la prise en compte de l'interdépendance forte entre le problème de l'allocation et celui de l'ordonnancement. De plus, si l'espace de recherche reste le même pour l'ensemble des méthodes, le nombre de solutions valides détectables par notre méthode sera bien plus important. En effet, cette technique ne contraint pas le placement par une méthode d'affectation des priorités spécifique telle que RM et DM.

2 Placement et affectation

Notre méthode de placement et d'affectation des priorités aux tâches et messages est basée sur le principe des *méthodes par séparation et évaluation*. Ces procédures de recherche s'appuient sur une structure arborescente permettant la mémorisation des solutions potentielles.

2.1 Principes généraux de la procédure de recherche

L'arbre de recherche contient l'ensemble des permutations possibles des priorités aux tâches pour toutes les affectations possibles des tâches aux processeurs. Un nœud de l'arbre représente donc le placement d'une tâche τ_i sur un processeur Pr_j et l'affectation de la priorité π_i à τ_i. En pratique, l'arbre de recherche est mémorisé dans une liste contenant les nœuds de l'arbre non encore explorés. La construction des nœuds se fait durant la recherche. Nous présentons ci-dessous l'algorithme général de la *procédure par séparation et évaluation*.

`Procédure Allocation & Affectation`
Début

- `Initialisation de l'ensemble Actif` A `avec les noeuds`

$\{1..N\}$ correspondant aux tâches ordonnançables à la plus forte priorité sur un processeur libre du premier pool
TantQue $A \neq \emptyset$ **Faire**
- Sélection d'un noeud dans A suivant la règle de *sélection*
- Génération d'un ensemble B de noeuds fils en fonction de la règle de *branchement*
- Calcul de la borne inférieure pour chaque noeud $\in B$ selon le principe d'évaluation
- Suppression des noeuds de B selon la règle d'élimination

Si il existe une feuille valide $\in B$
- Stop

Sinon
- Copie de tous les éléments de B dans A

FinSi

FinFaire

Fin

Nous détaillons maintenant la structure de l'arbre de recherche et les différentes règles utilisées dans l'algorithme ci-dessus.

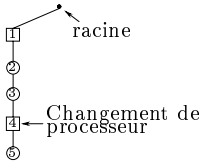
2.2 Structure de l'arbre de recherche

Dans la procédure de recherche, l'énumération se fait pool de processeurs par pool de processeurs. Nous présentons dans un premier temps les règles de construction à l'intérieur d'un pool de processeurs, puis la structure générale de l'arbre de recherche. Nous appelons *processeur courant* le processeur sur lequel à un instant t de la procédure, les tâches ordonnançables seront placées. Cette définition est étendue à la notion de pool de pool courant Pl_c.

2.2.1 Structure arborescente d'un pool

La structure arborescente utilisée pour la recherche et l'affectation des priorités dans un pool de processeurs est basée sur les travaux présentés dans [BH91, Vig97]. Ces auteurs présentent une façon d'énumérer, sans répétition, tous les arrangements possibles de n tâches sur m processeurs identiques. Ils utilisent pour cela deux types de nœuds : les nœuds ronds et les nœuds carrés. Si le chemin passe par un nœud rond, la tâche est placée sur le processeur courant. Si, en revanche, le chemin passe par un nœud carré, alors la tâche correspondante est affectée au processeur suivant, ce dernier devenant alors le processeur courant. La table de la figure 2 donne le placement et l'affectation des tâches correspondant à l'exemple du chemin présenté sur cette même figure 2. Les priorités sont affectées dans l'ordre décroissant des niveaux de priorités. Ainsi, les tâches modélisées par un nœud carré possèdent le niveau de priorité le plus fort sur le nouveau processeur. Notons que l'affectation des priorités est locale à un processeur. Ceci implique qu'à chaque changement de processeur

(i.e. chaque nœud carré dans la branche) les priorités sont réinitialisées. Ainsi, une tâche modélisée par un nœud carré possède le niveau de priorité le plus fort, c'est-à-dire la priorité 0. La construction de l'arbre de recherche repose sur le respect de l'ensemble de règles. Ces règles permettent d'une part de parcourir l'ensemble des solutions potentielles, et d'autre part d'éviter les phénomènes de redondances.



Tâches τ_i	Pr_p	π_i
1	1	0
2	1	1
3	1	2
4	2	0
5	2	1

FIG. 2 – *Exemple d'une branche de l'arbre*

Intéressons nous aux règles de constructions :

- 1. Le niveau 0 de la structure arborescente contient un unique nœud fictif représentant la racine de l'arbre.
- 2. Le niveau 1 est formé de $n - m + 1$ nœuds carrés. Ceci correspond au placement des $n - m + 1$ premières tâches sur le premier processeur. Ces tâches seront ordonnancées au niveau de priorité le plus fort.
- 3. Un chemin partant du niveau 0 et se terminant au niveau i, $1 \leq i \leq n$, peut être agrandi au niveau $i + 1$ par n'importe quel nœud rond parmi n ou n'importe quel nœud carré parmi n, si les règles 4, 5 et 6 sont respectées.
- 4. Le numéro k d'une tâche τ_k n'apparaît qu'une seule fois dans tout le chemin de la racine à une feuille de l'arbre. Ceci assure qu'une tâche ne sera placée qu'une seule fois.
- 5. Un nœud carré k ne peut être utilisé pour agrandir un chemin contenant déjà un nœud carré l tel que $l > k$. Ceci évite les redondances d'énumération, c'est-à-dire la construction de chemins différents mais modélisant le même placement des tâches et la même affectation des priorités.
- 6. Aucun chemin ne peut être étendu par un nœud carré quelconque si ce chemin contient déjà m nœuds carrés. Cette règle assure le respect du nombre de processeurs.
- 7. Aucun chemin ne se termine s'il contient moins de m nœuds carrés, sauf si $n < m$. Ceci assure que tous les processeurs du pool sont utilisés.

Le placement de deux tâches en précédence sur un même processeur induit une règle de construction supplémentaire. Un ordonnancement de tâches à priorités

fixes sur un processeur respecte les contraintes de précédence si :

$$\tau_i \prec \tau_j \Rightarrow \pi_i < \pi_j \qquad \text{(i.e. } \tau_i \text{ est plus prioritaire que } \tau_j\text{)} \qquad (1)$$

Le placement des deux tâches τ_k et τ_l sur la même machine (i.e. insertion entre le dernier nœud carré et le nœud courant) implique le respect de la règle 8 :

- 8. Soient deux tâches en précédence, $\tau_k \prec \tau_l$; si τ_k et τ_l sont placées sur le même processeur, alors il n'existe pas de sous chemin débutant par un nœud carré l ou un nœud rond l et se terminant par un nœud rond k ne contenant que des nœuds ronds entre l'origine et la fin du sous chemin. Ceci assure que la relation [1] est vérifiée pour les tâches τ_k et τ_l.

Remarquons que s'il existe un mécanisme de synchronisation entre deux tâches dépendantes sur un unique processeur, cette technique n'est pas optimale. Précisément, l'ensemble des affectations des priorités respectant la condition [1] n'est pas un ensemble dominant. Nous proposons dans [RRGC02] une méthode de validation en présence d'un mécanisme de synchronisation.

La charge engendrée par l'ensemble de tâches placées sur un processeur doit être inférieure ou égale à 1. Cette condition implique une nouvelle règle de construction :

- 9. Un sous chemin débutant par un nœud carré k et composé de l nœuds ronds ne peut être étendu par un nouveau rond r si la charge des tâches composant le sous chemin, notée U_s, additionnée à la charge engendrée par τ_r est strictement supérieure à 1. Si $U_s + U_{\tau_r} > 1$ et $r > k$ (Cf. règle 5), alors un nœud carré r sera créé.

Deux nouvelles règles de construction sont ajoutées, afin d'éviter la construction de nœuds menant à des branches structurellement incorrectes. Ces règles trouvent leur justification dans la combinaison des règles 5 et 7. Précisément, l'idée est de vérifier, avant l'insertion d'un nœud rond ou carré, si le nombre de tâches non encore placées est suffisant par rapport au nombre de processeurs restants et non utilisés. La figure 3 présente l'influence des règles 10 et 11 sur la construction de l'espace des solutions parcouru. Sur la figure 3.a, le nœud carré τ_5 est créé et ne mène à aucune solution respectant les règles d'énumération. À la construction du nœud carré τ_5 (figure 3.b), le nombre de tâches restant à affecter dont l'insertion future dans un nœud carré respectera la règle 5 est inférieur au nombre de processeurs non utilisés. Ainsi, aucune sous-branche respectant la règle 7 et issue du nœud carré τ_5 ne pourra être construite dans la suite de la procédure de recherche. La règle 10 évite la construction du nœud carré τ_5. Dans le même esprit, la figure 3.c enumère le nœud rond τ_4. Or, en plaçant τ_4 à ce niveau, comme précédemment, le nombre de tâches restant à affecter et dont le numéro est supérieur au numéro de la tâche insérée dans le dernier nœud carré (i.e. τ_3) est inférieur au nombre de processeurs non utilisés. Aucun placement utilisant l'ensemble des processeurs (i.e. respect de la règle 10) ne peut être réalisé en plaçant le nœud rond τ_4 à ce niveau. La règle 11 évite la construction de ce nœud.

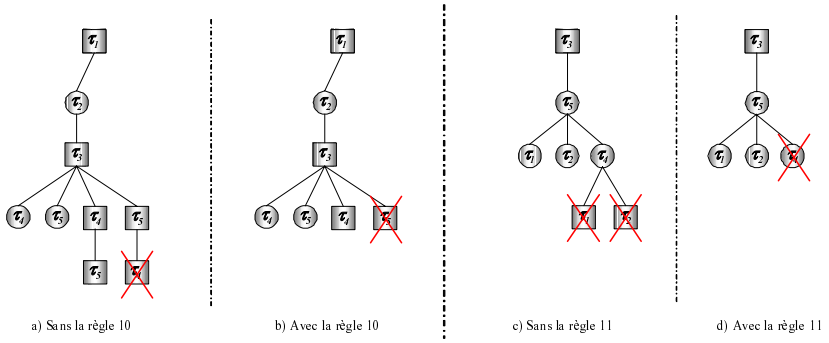


FIG. 3 – *Influence des règles 10 et 11 sur la construction de l'arbre*

- 10. Soit $\tau_{k'}$ la tâche à insérer. On note nb_τ le nombre de tâches τ_r non encore placées et telles que $r \geq k'$. Soit nb_{Pr} le nombre de processeurs non utilisés : si $nb_\tau < nb_{Pr}$ alors le nœud carré k' n'est pas créé.
- 11. Soit $\tau_{k'}$ la tâche insérée dans le dernier nœud carré. Si $nb_\tau < nb_{Pr}$ alors le nœud rond k' n'est pas créé.

2.2.2 Structure arborescente globale

La figure 4 présente la structure arborescente globale sur laquelle s'appuie notre méthode. Les différents sous arbres, modélisant les pools de processeurs et les réseaux, sont obtenus par l'application des règles présentées ci-dessus. Cette structure de sous arbre permet de modéliser un ensemble de m processeurs identiques. La modélisation de p ensembles de m_k processeurs identiques, $k \in \{1, \ldots, p\}$, différents entre eux est réalisée par la juxtaposition de p sous-arbres. Chaque sous-arbre modélisant un pool de processeurs est relié au suivant par un nœud fictif. Ceci sous-tend que toutes les feuilles d'un sous-arbre sont reliées au *même* sous-arbre inférieur.

Lorsque l'architecture comporte un ou r réseaux, chaque réseau et l'ensemble des messages circulant sur celui-ci sont modélisés par un sous arbre représenté en bas de la figure 4. On parle alors de pool réseau ; chaque pool réseau est composé d'un unique réseau et de l'ensemble des messages circulant sur ce dernier. Notons que l'ensemble des messages d'un réseau n'est totalement défini que lorsque toutes les tâches de l'application sont placées. En conséquence, les sous arbres modélisant les réseaux sont situés après tous les sous arbres représentant les pools de processeurs. Dans un pool réseau, la méthode de placement et d'affectation revient à uniquement affecter les priorités aux différents messages.

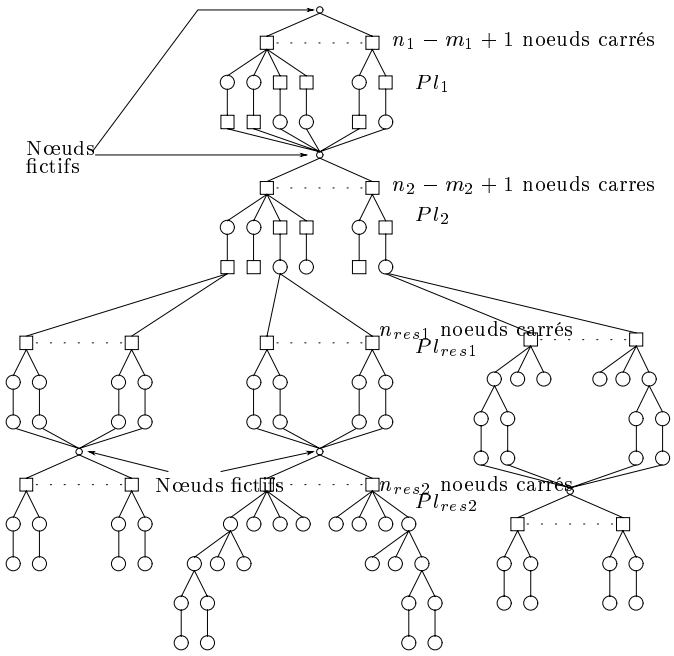


FIG. 4 – *Structure générale de l'arbre de recherche*

2.3 Règles de branchement

Les règles de branchement déterminent l'ordre de parcours des pools, des processeurs au sein des pools, et des tâches sur chaque processeur.

Dans le but de faciliter et d'améliorer la recherche, c'est-à-dire augmenter les chances de trouver un placement et une affectation valide rapidement, nous affinons la structure de l'arbre de recherche. Les heuristiques de construction mises en place concernent les pools de processeurs et les tâches.

2.3.1 Ordre des pools de processeurs

Les pools de processeurs sont classés selon leur *coefficient de charge globale*. Nous définissons le *coefficient de charge globale d'un pool de processeur* Pl_i comme la somme des charges des n_i tâches composant l'ensemble Θ_i, soit :

$$U_{Pl_i} = \frac{1}{m_i} \sum_{k=1}^{n_i} \frac{C_k}{T_k} \tag{2}$$

Nous avons choisi d'ordonner les pools selon la charge décroissante. Dans ce cas, le pool de plus forte charge globale est positionné au début de la structure

arborescente. S'agissant du pool le plus chargé, le nombre de feuilles de ce sous arbre susceptibles de contenir une solution valide est plus faible que dans le cas d'un pool de processeurs de charge plus faible. Ainsi, puisqu'il existe peu de placement valide, le nombre de coupes effectuées est grand. Or, une coupe proche du sommet de la structure élimine de l'arbre de recherche tout le sous arbre correspondant. Le nombre de solutions à explorer diminue donc d'autant.

2.3.2 Ordre des tâches

Les priorités sont affectées aux tâches dans l'ordre des niveaux de priorité croissant. Ainsi, entre deux nœuds carrés de niveau k et l, $k < l$, une tâche modélisée par un nœud de niveau k possède un niveau de priorité supérieur à une tâche représentée par un nœud de niveau l (cf. exemple de la figure 2). Un nœud carré correspond à la tâche la plus prioritaire sur le nouveau processeur courant (i.e. le passage à un nouveau processeur est modélisé par un nœud carré).

Les tâches d'un pool de processeurs Pl_i sont triées en fonction de la charge décroissante. Durant un parcours en *profondeur d'abord* de l'arbre de recherche, les tâches de charge importante sont plus difficiles à ordonnancer. Ainsi, le nombre de coupes dans les premiers niveaux de chaque sous arbre est plus grand. Ces coupes sont également plus bénéfiques puisqu'elles diminuent de manière plus importante l'espace des solutions à explorer.

2.4 Règles de sélection : Parcours

La règle de sélection permet de choisir le prochain nœud à traiter. Un nœud de l'arbre représente le placement d'une tâche sur un processeur et l'affectation à un niveau de priorité. Nous distinguons deux niveaux dans la notion de parcours : le parcours au sein d'un pool de processeurs et la stratégie de parcours globale de l'arbre de recherche.

2.4.1 Parcours au sein d'un pool de processeur

À ce niveau, nous avons implémentés deux types de parcours différents : un parcours dit *"de remplissage"* et un parcours dit *"d'équilibrage"*.

- *Parcours de remplissage :* dans ce cas, les nœuds favorisés sont les nœuds ronds. Ainsi tant que la règle de charge (i.e. règle 9) est respectée, et que les tâches modélisées par les nœuds ronds sont ordonnançables, elles sont ajoutés sur le même processeur. Les solutions obtenues sont alors déséquilibrées en terme de répartition des tâches sur les processeurs du pool.
- *Parcours d'équilibrage :* afin d'estomper le phénomène précédent, nous avons implémenté un parcours visant à construire en premier lieu les solutions les plus équilibrées. Précisément, nous ajoutons un critère de

construction permettant de créer le plus rapidement possible des solutions pour lesquelles le placement des tâches sur les processeurs du pool est équilibré. Ainsi, pour un pool Pl_i composé de n_i tâches, m_i processeurs et possédant une charge globale U_{pl_i} deux stratégies sont envisagées :

1. *fonction du nombre de tâches :* les nœuds ronds sont favorisés (i.e. construit en premier) tant que le nombre de tâches placées sur le processeur en cours de traitement est inférieur à :

$$\left\lceil \frac{n_i}{m_i} \right\rceil$$

   Lorsque cette borne est atteinte, les nœuds carrés sont construits en premier permettant ainsi de changer de processeur.
2. *fonction de la charge :* de la même manière que précédemment, les nœuds ronds sont favorisés tant que la charge induite par les tâches déjà placées sur le processeur en cours de traitement est inférieure à :

$$\frac{U_{Pl_i}}{m_i}$$

2.4.2 Parcours global de l'arbre de recherche

Nous avons montré dans [Ric02] que l'arbre de recherche est fortement déséquilibré. Précisément, le nombre de solutions potentielles à examiner est plus grand dans le premier sous-arbre (gauche) que dans le dernier (droite). Lors du traitement d'une instance de taille importante, un *parcours en profondeur simple* ne parcourera que la partie gauche de l'arbre sans jamais visiter la partie droite de l'arbre. Dans le but de construire des solutions équitablement réparties dans l'arbre, nous avons implémentés trois types de parcours de l'arbre de recherche (i.e. empilement de pool de processeurs), tous basés sur le *parcours en profondeurs d'abord*.

- *parcours en profondeur simple :* dans chaque pool, la branche la plus à gauche (i.e. première branche construite dans le pool en fonction du parcours local au pool choisi) est parcourue en premier.
- *parcours en parallèle :* afin d'éviter le problème rencontré dans le premier type de parcours, nous divisons le parcours de l'arbre. Au début de la procédure de recherche, l'arbre est découpé en k parties où k est le nombre de nœuds carrés du premier niveau du premier sous-arbre de l'abre global, c'est-à-dire $n_1 - m_1 + 1$. Ces différents arbres sont alors parcourus séquentiellement de manière équitable. Ceci revient à alterner un parcours en profondeur et un parcours en largeur. Ce type de parcours se heurte à une complexité en espace plus importante que le parcours précédent et qui est due au parcours en largeur.
- *parcours en parallèle auto adaptatif :* dans l'optique de diminuer la complexité en espace du parcours ci-dessus, est d'augmenter les chances de

trouver une solution valide rapidement, nous conservons le même découpage en $n_1 - m_1 + 1$ arbres, et modifions la séquence de parcours de ceux-ci. Précisément, ils sont toujours parcouru séquentiellement, mais nous favorisons maintenant un des $n_1 - m_1 + 1$ arbres. Nous entendons par favoriser un arbre, l'action de parcourir plus longtemps cet arbre que les autres. Deux politiques peuvent alors être tenues en fonction du nombre de solutions en cours de construction dans l'arbre. Nous avons choisi de privilégier l'arbre contenant le plus de solutions en cours de construction. En effet, c'est dans l'arbre contenant le plus de solutions que la probabilité de trouver un solution valide est la plus forte.

Notons enfin que le dernier parcours présenté est fortement parallélisable, puisque l'arbre globale est découpé en $n_1 - m_1 + 1$ parties totalement indépendantes. Il est simple de montrer que tous les parcours présentés ci-dessus sont polynomialement borné en espace mémoire.

2.5 Évaluation

À chaque création d'un nœud, il faut pouvoir décider si la tâche est ordonnançable au niveau de priorité choisi sur le processeur courant. Or, il n'existe pas de condition nécessaire et suffisante d'ordonnançabilité pour les systèmes distribués à priorités fixes. Dans la suite, après avoir brièvement rappeler les principes de la méthode pire cas sur laquelle s'appuie notre méthode, nous présentons les règles d'évaluation de la solution en cours de construction dans les différents contextes pouvant se présenter lors de la procédure de recherche arborescente.

La technique d'évaluation est basée sur l'analyse holistique présentée dans [Tin94, Ric02]. Remarquons que celle-ci fait l'hypothèse que les tâches sont indépendantes sur chaque processeur. Nous levons cette hypothèse en assurant une affectation des priorités aux tâches respectant l'ordre partiel. Le mécanisme des priorités assure le respect des contraintes de précédence.

2.5.1 Évaluation d'une borne inférieure des pires temps de réponse

Lorsqu'une tâche est placée à un niveau de priorité, un nœud est créé. Afin de déterminer si le placement est valide, nous évaluons une borne inférieure $LB(Tr_i)$ du pire temps de réponse Tr_i. Pour cela nous reprenons le système de recherche de point fixe de l'analyse holistique. La fonction d'évaluation des temps de réponse, *Evaluer*, est modifiée ainsi que la fonction de mise à jour des gigues sur l'activation, *Propager*.

$$1 \leq i \leq n \quad \left\{ \begin{array}{lcl} LB\left(Tr_i^{(k)}\right) & = & Evaluer\left(LB\left(J_i^{(k-1)}\right)\right) \\ LB\left(J_i^{(k)}\right) & = & Propager\left(LB\left(Tr_j^{(k)}\right)\right) \end{array} \right. \tag{3}$$

Le point fixe est obtenu lorsque pour le plus petit $k \in \mathbb{N}$:

$$LB(Tr_i) = LB\left(Tr_i^{(k-1)}\right) = LB\left(Tr_i^{(k)}\right), 1 \leq i \leq n \quad (4)$$

L'évaluation des bornes inférieures des temps de réponse est appliquée sur un nœud donné, c'est-à-dire pour un placement des tâches et une affectation des priorités incomplets. Les calculs dépendent donc du contexte du nœud analysé, c'est-à-dire :

- du placement : le placement des différentes tâches sur les processeurs va influer sur le graphe des communications, et donc sur le calcul des bornes inférieures des gigues sur l'activation.
- du traitement : si le placement et l'affectation des priorités ont été effectués ou non.
- du type de pool : si le pool est régi par une politique d'ordonnancement préemptive (processeur), ou non préemptive (réseau).

Le principal intérêt du système (3) est de propager les résultats, c'est-à-dire les bornes inférieures, de la fonction *Evaluer* aux tâches et aux messages via la mise à jours des gigues sur l'activation, et ce, de manière immédiate, par l'intermédiaire de la fonction *Propager*. À chaque fois qu'un nœud est inséré dans l'arbre de recherche, le système (3) est résolu pour l'ensemble des tâches et messages de l'application. Or, le nombre de messages circulant sur le réseau dépend totalement du placement des tâches. La valeur n peut non seulement être différente entre deux branches de l'arbre, mais aussi au cours du parcours d'une même branche. Ainsi, au début de la procédure de recherche, la valeur n correspond au nombre de tâches puisqu'on ne connaît pas le nombre de messages sur le réseau.

Nous détaillons ci-dessous le fonctionnement des fonctions *Propager* et *Evaluer* dans les différents contextes rencontrés au cours de la construction d'une branche de l'arbre de recherche. La mise à jour est différente dans le cas d'une tâche ou d'un message.

Fonction *Propager* : Le rôle de la fonction *Propager* est de mettre à jour les gigues sur l'activation des tâches et des messages de l'application. Nous rappelons que la gigue sur l'activation d'une tâche est nulle si cette tâche est indépendante.

- *cas d'une tâche τ_i :* si $\Gamma_i^- = \emptyset$, alors la tâche τ_i ne subit aucun retard dû à un message. Ainsi la borne inférieure de la gigue sur l'activation est nulle : $LB(J_i) = 0$. Dans le cas contraire, une borne inférieure de la gigue sur l'activation est obtenue par l'équation (5) :

$$LB(J_i) = \max_{k \in \Gamma_i^-} \{LB(Tr_k)\} \quad (5)$$

- *cas d'un message m_i :* si le message m_i est traité dans le système itératif (3), alors l'émetteur et le récepteur du message m_i ne sont pas sur le même processeur. Ainsi, il existe $\tau_k \in \Gamma_i^-$. La gigue sur l'activation de m_i est donc égale au pire temps de réponse de l'émetteur τ_k, qui est unique.

$$LB(J_i) = Tr_k \tag{6}$$

Nous présentons maintenant le fonctionnement de la fonction *Evaluer* dans les différents contextes rencontrés lors de la procédure de recherche.

Fonction *Evaluer* **:** La fonction *Evaluer* calcule une borne inférieure du temps de réponse d'une tâche ou d'un message. Nous rappelons que du point de vue de la validation, c'est-à-dire de l'ordonnancement, un message est considéré comme une tâche ordonnancée sur un processeur non préemptif. Nous distinguons ces deux cas dans le fonctionnement de la fonction *Evaluer*.

D'une manière générale, le calcul du temps de réponse d'une tâche repose sur des résultats classiques de la littérature [LL73, Leh90, Tin94, GRS96]. Précisément, le calcul du temps de réponse d'un tâche est basée sur la plus grande période de temps où le processeur est pleinement occupé par des tâches de priorités supérieures ou égales à i. C'est le concept de ("*i-level Busy Period*")[Leh90]. La longueur maximale de la période d'activité est obtenue lorsque les premières instances des tâches plus prioritaires sont activées simultanément lors de la réception de leurs premiers messages et que toutes les instances suivantes des tâches plus prioritaires sont exécutées sans attente [Tin94] (instant critique).

Ordonnancement préemptif (cas d'une tâche) : La fonction de charge Int dépend donc des tâches affectées à des niveaux de priorité supérieurs à celui de la tâche τ_i. Deux cas doivent être distingués : la tâche étudiée est placée et possède une priorité, ou la tâche analysée n'est pas placée et ne possède donc pas de priorité (i.e. tâche non encore traitée par la procédure de recherche).

- Si la tâche τ_i analysée est placée et possède une priorité l, l'ensemble des tâches plus prioritaires du processeur sur lequel elle s'exécute est connu. Précisément, il est formé des tâches modélisées par les nœuds compris entre le premier nœud carré supérieur (inclus) et le nœud modélisant la tâche étudiée ($l^{ième}$ nœud rond suivant le dernier nœud carré). Dans ce contexte, calculer le pire temps de réponse Tr_i revient à examiner l'instance de τ_i s'exécutant durant la période d'activité de niveau l. Lorsque la tâche analysée est placée et possède une priorité, la fonction *Evaluer* calculant une borne inférieure du pire temps de réponse est définie par le point fixe de l'équation (7).

$$\begin{aligned} Int^{(k+1)} &= C_i + \sum_{j=0}^{l-1} \left\lceil \frac{LB\left(J_{[j]}\right) + Int^{(k)}}{T_{[j]}} \right\rceil C_{[j]} \\ LB\left(Tr_i\right) &= Int + LB\left(J_i\right) \end{aligned} \tag{7}$$

Le point fixe est atteint lorsque:

$$Int = Int^{(k+1)} = Int^{(k)} \tag{8}$$

- Si la tâche τ_i n'est pas placée et ne possède pas de priorité, nous dissocions deux nouveaux cas en fonction du processeur en cours. Soit Pr_c le processeur courant appartenant à l'ensemble PR_c du pool courant Pl_c.
  - Si Pr_c est le dernier processeur de Pl_c et si $\tau_i \in \Theta_c$, on connaît un sous-ensemble minimum des tâches plus prioritaires que τ_i. En effet, puisque Pr_c est le dernier processeur de Pl_c, le seul placement possible pour la tâche analysée est Pr_c. Ainsi un sous-ensemble minimum de tâches plus prioritaires que τ_i est constitué des tâches modélisées par les nœuds compris entre le dernier nœud carré (inclus) de la branche et le dernier nœud rond de cette même branche. Soit l le niveau du dernier nœud rond. Il est alors possible de déterminer une borne inférieure de l'interférence induite par les tâches de plus forte priorité par la méthode de recherche de point fixe utilisée plus haut. La fonction de charge est donnée par l'équation (9).

$$Int^{(k+1)} = \sum_{j=0}^{l} \left\lceil \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rceil C_{[j]} + C_i \tag{9}$$

    Le point fixe est atteint lorsque :

$$Int = Int^{(k+1)} = Int^{(k)} \tag{10}$$

    La borne inférieure du temps de réponse de τ_i est calculée par l'équation (11) :

$$LB\left(Tr_i\right) \quad = \quad Int + LB\left(J_i\right) \tag{11}$$

  - Si Pr_c n'est pas le dernier processeur de Pl_c ou si $\tau_i \notin \Theta_c$, on ne connaît pas le processeur sur lequel sera placé τ_i dans la suite de la procédure de recherche. Il n'est donc pas possible d'établir une borne inférieure de l'interférence des tâches plus prioritaires. Dans ce cas, la borne inférieure du temps de réponse de τ_i se résume donc à l'équation (12) :

$$LB(Tr_i) = LB(J_i) + C_i \tag{12}$$

Ordonnancement non préemptif (cas d'un message) : les résultats obtenus dans un contexte préemptif s'étendent aisément au contexte non préemptif. L'unique différence provient du décalage pouvant être engendré par un message moins prioritaire, utilisant le processeur à l'instant critique. La préemption n'étant pas autorisée, l'instant critique est décalé d'au maximum la plus longue durée de propagation parmi les messages moins prioritaires. Ainsi,

le pire cas survient lorsque ce message moins prioritaire débute sa propagation sur le réseau à l'instant -1 en imposant que l'instant critique survienne à la date 0 [GRS96]. Là encore, nous distinguons le cas d'un message traitée ou non.

- le message m_i est placé et possède une priorité: toutes les tâches sont affectées et possèdent une priorité. L'ensemble des messages est alors complètement connu. De plus, les messages de niveau de priorité inférieur à celui de m_i sont connus. En effet, l'ensemble des messages ne possédant pas de priorité à cet instant seront assignés, dans la suite de la procédure, à un niveau de priorité inférieur à celui de m_i. Soit $\Theta_r^\star$ cet ensemble. Une borne inférieure du temps de blocage est alors obtenue en considérant le plus long message parmi ceux appartenant à l'ensemble $\Theta_r^\star$. La fonction de charge dans ce cas est donnée ci-dessous (13).

$$Int^{(k+1)} = C_i + \sum_{j=0}^{i-1} \left(\left\lfloor \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rfloor + 1 \right) C_{[j]} + \max_{k \in \Theta_r^\star}(C_k) \quad (13)$$

- le message m_i ne possède pas de priorité: pour faciliter la lecture, nous présentons dans une première approche un premier calcul de borne en dissociant deux cas: le pool courant correspond au pool réseau ou non. Puis, nous apportons une amélioration valable pour les deux cas présentés ci-dessous.

  1. Première approche:
     Soit Pl_r le pool réseau auquel est associé le message m_i étudié.
     - $Pl_c = Pl_r$: dans ce cas, toutes les tâches sont placées et possèdent une priorité. Comme dans le cas préemptif, nous pouvons déterminer un ensemble minimum de messages qui seront ordonnancés à une priorité plus forte que celle de m_i dans la suite de la procédure de recherche. Une borne inférieure de l'interférence induite par ces messages est obtenue par l'équation (14). Le message analysé ne possédant pas encore de priorité, il peut, par la suite, être ordonnancé à la priorité la plus faible. Dans ce cas le facteur de blocage dû à la politique non préemptive sera nul.

$$Int^{(k+1)} = C_i + \sum_{j=0}^{l} \left(\left\lfloor \frac{LB(J_{[j]}) + Int^{(k)}}{T_{[j]}} \right\rfloor + 1 \right) C_{[j]} \quad (14)$$

     - $Pl_c \neq Pl_r$: l'ensemble des messages considérés à cet instant n'est pas totalement connu puisque toutes les tâches ne sont pas placées. En conséquence, les messages considérés ici ne possèdent pas de priorités. Dans ce contexte, la meilleure borne inférieure du temps de réponse est obtenue en considérant que le message sera affecté au niveau de priorité le plus élevé. Une borne inférieure de l'interférence des messages plus prioritaires est obtenue en considérant que le message sera ordonnancé à ce

niveau de priorité. Dans ce contexte, on obtient :

$$Int = C_i \tag{15}$$

2. Extension :
   Nous nous intéressons de manière plus précise au message m_i. Dans les deux cas précédents, nous avons considéré qu'il n'était pas possible d'estimer une borne inférieure du facteur de blocage induit par la politique d'ordonnancement non préemptive du réseau. Or, si l'on considère $\Theta_c^\star$ l'ensemble des messages traités par le système itératif (3) à cet instant (qui dépend du placement des tâches émettrices et réceptrices) et ne possédant pas de priorité, il est possible d'obtenir une borne inférieure du facteur de blocage par la formule 16 :

$$\max_{k \in \Theta_r^\star - \{m_i\}} (C_k) \tag{16}$$

   Nous devons prouver qu'il s'agit bien d'une borne inférieure afin de rester optimale vis-à-vis du critère d'évaluation. Plaçons nous pour cela dans les différents cas pouvant se produire lors de la suite de la procédure de recherche. Soit m_b, de priorité π_b, le message produisant le temps de blocage le plus long.
   Si m_i se voit affecter un niveau de priorité tel que $\pi_i > \pi_b$, le message m_b sera bien considéré dans l'ensemble $\Theta_r^\star$ lors de la résolution de l'équation (13). Il s'agit donc bien dans ce contexte d'une borne inférieure.
   Considérons maintenant que m_i obtient le plus faible niveau de priorité ou un niveau de priorité vérifiant $\pi_i < \pi_b$. Ici, le message m_b ne sera pas inclus dans l'ensemble $\Theta_r^\star$ lors de la résolution de (13). Or m_b possédant à cet instant un niveau de priorité supérieur à celui de m_i, il sera considéré dans l'interférence induite par l'ensemble des messages plus prioritaire. De plus, une borne inférieure de l'interférence engendré par m_b est C_b. À nouveau, la formule (16) est bien une borne inférieure. Nous étendons donc les formules (14) et (15) en y ajoutant le terme (16) afin d'améliorer la qualité du critère d'évaluation.

Les bornes inférieures présentées ci-dessus sont non décroissantes durant la construction d'une branche de l'arbre de recherche. Cette propriété assure l'optimalité vis-à-vis de la méthode de validation.

2.6 Règle d'élimination

Les bornes inférieures de temps de réponse obtenues par la méthode présentée dans le paragraphe précédent sont utilisées pour tester la validité du placement et de l'affectation des priorités associés au nœud courant. Pour ce faire, la règle suivante est appliquée.

Règle 1 : *S'il existe $i \in 1..n$ tel que $LB(Tr_i) > D_i$ alors le placement et l'affectation des priorités en cours ne peut mener à une solution valide. La branche associée au nœud est coupée, et un retour arrière (Backtracking) est effectué.*

La méthode s'arrête dès qu'une branche complète de l'arbre conduit à un placement et une affectation des priorités permettant de valider l'application par la méthode d'évaluation présentée dans le paragraphe précédent.

3 Expérimentation

Nous avons créé un générateur aléatoire d'applications temps réel. Le nombre de sites, le nombre de réseaux, la charge globale et le nombre de communications sont des paramètres réglables du générateur. Les tâches générées sont à échéance sur requête, et le nombre de pools de processeurs est fixé à 1. Afin de produire des configurations réalistes, la génération des durées d'exécution et des périodes suit deux lois normales différentes. La loi utilisée pour les durées d'exécution est paramétrée par : $m = 1$, et $\sigma = 12$. Concernant les périodes, les paramètres m et σ sont dépendant du facteur de charge globale désiré. Dans le contexte distribué, la génération des communications est réalisée par un tirage aléatoire de l'émetteur, du récepteur et de la taille de la section de données du message. Période et échéance sont héritées des tâches émettrice et réceptrice. Enfin, dans le but d'obtenir des résultats objectifs, nous produisons aléatoirement 50 configurations différentes pour chaque taille de problème. Si le temps de calcul est supérieur à une heure, la recherche est stoppée.

3.1 Architecure distribuée

La figure 5 présente l'évolution du nombre de configurations validées en fonction du nombre de sites pour une charge fixe variant entre 0,5 et 0,6. Le nombre de tâches est fixe (40) et le nombre de messages varie entre 5 et 20. Quelle que soit la taille du problème, le nombre d'instances validées est supérieur ou égale à 43, soit 86%. Ce résultat est obtenu pour des problèmes composés de 5 ou 10 processeurs et d'au moins 10 messages. Lorsqu'au moins 20 sites sont disponibles, toutes les instances sont validées. Le pourcentage d'instances validées pour des problèmes de taille réaliste montre l'intérêt de notre méthode.

Les courbes de temps de calculs de la figure 6 correspondent aux traitements des instances de la figure 5. Ces valeurs sont des moyennes et, pour les trois premières courbes, intègrent les configurations non résolues en moins d'une heure. Le temps de recherche moyen, dans ce cas, est toujours inférieur à 1000 secondes quelle que soit la taille du problème traité. En réalité, le temps moyen (i.e. réel cf. figure 6) permettant de valider une configuration ne dépasse pas 35 secondes pour cette taille d'instance. La méthode supporte donc des applications de taille réaliste.

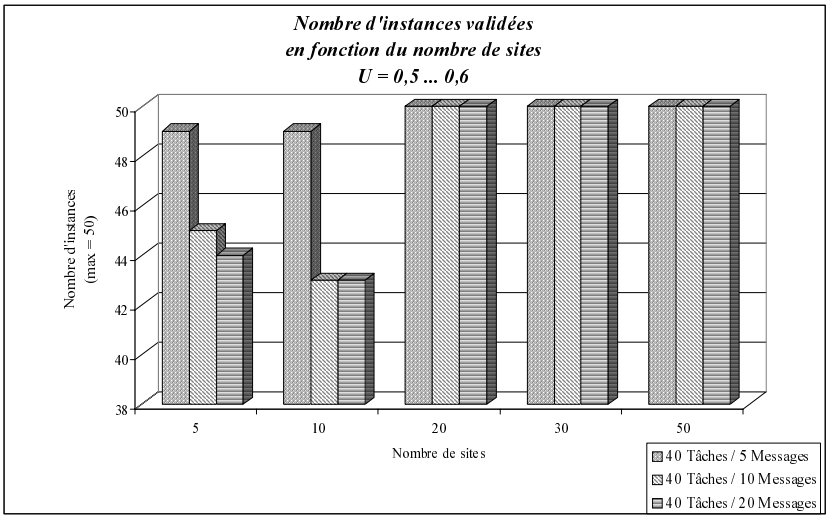


FIG. 5 – *Influence du nombre de sites et de messages sur le nombre d'instances validées*

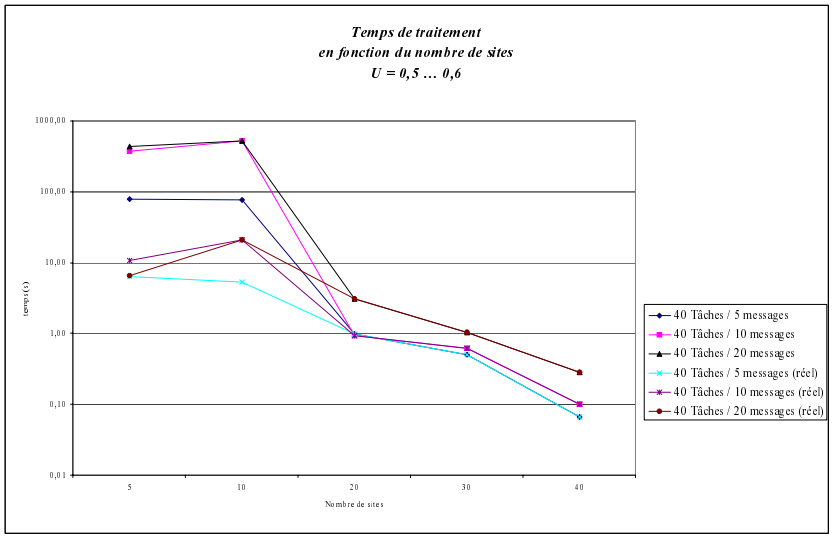


FIG. 6 – *Évolution du temps de traitement en fonction du nombre de sites et de messages*

3.2 Comparaison des différents parcours

Nous proposons dans ce paragraphe une comparaison des performances des différents types de parcours présentés. Précisèment, chaque parcours est l'association d'une stratégie de parcours au sein d'un pool de processeur et d'une stratégie de parcours global de l'arbre de recherche :

- *le parcours 1 :* est l'association d'un parcours de remplissage et d'un parcours en profondeur simple de la structure arborescente globale,
- *le parcours 2 :* est l'association d'un parcours d'équilibrage en fonction du nombre de tâches et d'un parcours en profondeur simple de la structure arborescente globale,
- *le parcours 3 :* est l'association d'un parcours de remplissage et d'un parcours en parallèle auto adaptatif de la structure arborescente globale. L'arbre contenant le plus de solutions en cours de contruction est privilégié,
- *le parcours 4 :* est l'association d'un parcours d'équilibrage en fonction du nombre de tâches et d'un parcours en parallèle auto adaptatif de la structure arborescente globale. L'arbre contenant le plus de solutions en cours de construction est provilégié.

La figure 7 présente les temps de traitement moyens en fonction du nombre de sites, pour 50 configurations traitées, de ces différents parcours. La configuration est composée de 15 tâches et 5 messages. Les parcours 3 et 4, qui effectuent parcours en paralèlle auto adaptatif, affichent des performances nettement supérieures aux parcours 1 et 2. La stratégie de parcours au sein d'un pool de processeurs améliore encore les temps de calculs (parcours 2 et 4) quel que soit le type de stratégie de parcours de la structure arborescente globale.

4 Conclusion

Nous avons présenté une méthode réalisant conjointement le placement et l'affectation des priorités des tâches et des messages d'une application temps réel distribuée. Cette méthode permet de modéliser des architectures physiques très différentes, ainsi que des architectures logicielles complexes. Les tests de la méthode montrent qu'elle peut être appliquée pour valider des systèmes réels.

La mise en œuvre actuelle impose que l'émetteur soit plus prioritaire que le récepteur dans le cas d'une communication locale. Dans [RRC01], nous avons étudié le cas général qui relaxe cette contrainte dans un contexte monoprocesseur. L'intégration de ce travail dans la méthode de placement présentée dans cet article permettra de valider un plus grand nombre de systèmes distribués.

Les perspectives de ce travail sont d'étendre la méthode pour considérer des critères techniques et économiques :

– *optimisation du nombre de processeurs dans chaque pool (critère écono-*

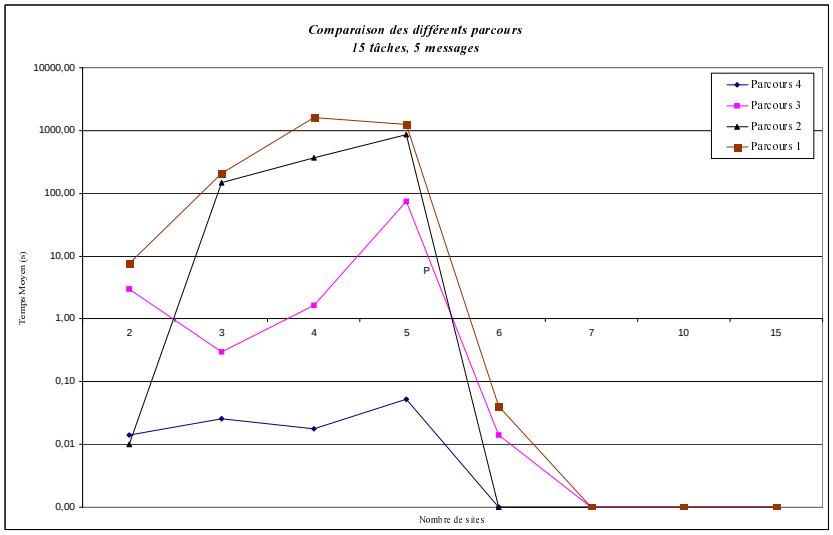


FIG. 7 – *Temps de calcul pour les différents types de parcours*

mique): le but est de trouver un solution valide minimisant le nombre de processeurs utilisés.

– *ajout de contraintes locales à un pool (critères techniques) :* parmi ces contraintes nous pensons modéliser des processeurs de vitesses différentes, prendre en compte des tâches dédiées à des processeurs spécialisés, ou encore, considérer des contraintes d'espace mémoire.

Références

[AH97] P. Altenbernd and H. Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Journal of Real-Time Systems*, 13(2):103–130, Septembre 97.

[Bea96] J. P. Beauvais. *Etude d'Algorithmes de Placement de Tâches Temps Réel Périodiques Complexes dans un Système Réparti.* PhD thesis, École Centrale de Nantes, Juin 1996.

[BH91] S. A. Brah and J. L. Hunsucker. Branch and bound algorithm for the flow shop with multiple processors. *European Journal of Operations Research*, 51:88–99, 1991.

[BT83] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report 2966, INRIA, Septembre 1996.

[ISO94a] Road vehicles – low-speed serial data communication – part 2: low-speed controller area network (can). ISO International Standard 11519-2, 1994.

[ISO94b] Vehicle area network, serial data communication – road vehicle, serial data communication for automotive application. ISO International Standard 11519-3, 1994.

[JS97] J. Jonsson and K. G. Shin. A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP'97)*, 1997.

[KA99] U.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[Law83] E. L. Lawler. Recent results in the theory of machine scheduling. *Mathematical Programming: The State of the Art*, pages 202–233, 1983.

[Leh90] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of Real-Time Systems Symposium*, pages 166–171, 1990.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, pages pp. 237–250, 1982.

[ML95] M. W. Mutka and J.-P. Li. A tool for allocating periodic real-time tasks to a set of processors. *Journal Systems Software*, 29:135–148, 1995.

[OCSF97] J. Orozco, R. Cayssials, J. Santos, and E. Ferro. Precedence constraints in hard real-time distributed systems. In *Proceedings of the* 3^{rd} *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 33–38, 1997.

[OS95] Y. OH and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time System Journal*, 9(3):207–239, Novembre 1995.

[Ose97] Osek operating system, version 2.0r1. http://www-iiit.etec.uni-karlsruhe.de/ osek/, 1997.

[PSA97] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12):745–758, 1997.

[Ram95] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, April 1995.

[Ric02] M. Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes et Placement.* PhD thesis, École Nationale Supérieure de Mécanique et d'Aérotechnique – Université de Poitiers, Novembre 2002.

[RRC01] M. Richard, P. Richard, and F. Cottet. Affectation optimale des priorités des tâches et des messages dans les systèmes distribués temps réel. In Teknea, editor, *Real Time and Embedded Systems*, pages 107–122, mars 2001.

[RRGC02] M. Richard, P. Richard, E. Grolleau, and F. Cottet. Contraintes de précédences et ordonnancement mono-processeur. In Teknea, editor, *Real Time and Embedded Systems*, pages 121–138, 26-28 mars 2002.

[SFOC97] J. Santos, E. Ferro, J. Orozco, and R. Cayssials. A heuristic approach to the multitask-multiprocessor assignment problem using the empty-slots method and rate monotonic scheduling. *Journal of Real-Time Systems*, 13(2):167–199, 1997.

[SMM99] I. Santhoshkumar, G. Manimaran, and C. S. R. Murthy. A pre-run-time scheduling algorithm for object-based distributed real-time systems. *Journal of Systems Architecture*, 45:1169–1188, 1999.

[ST85] C. C. Shen and W.H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transaction Computers*, 34(3):41–47, Jan 1985.

[SVC98] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *Proceedings of the 10^{th} Euromicro Workshop on Real Time Systems*, pages 53–60. Berlin, Germany, 17-19 June 1998.

[TBW92] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks (an np-hard problem made easy). *Real-Time Systems*, 4(2):145–165, 1992.

[Tin94] K. W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems.* PhD thesis, Univeristy of York, 1994.

[Vig97] A. Vignier. *Contribution à la résolution des problèmes d'ordonnancement de type monogamme, multimachine ("Flow-shop hybride").* PhD thesis, École d'Ingénieurs en Informatique pour l'Industrie, Tours, 1997.