Formalisation et Optimisation Statique d'Implantations Parallèles

Deuxièmes Journées Francophones de Recherche Opérationnelle
Sousse, Tunisie 6-8 Avril 1998

Annie Vicard et Yves Sorel

Résumé *Dans ce papier, nous allons tout d'abord présenter un formalisme relationnel, reposant sur la théorie des graphes, permettant de décrire l'ensemble des implantations possibles d'un algorithme d'application sur une machine parallèle. Pour ce faire, nous présentons les modèles de graphes d'algorithme d'application, d'architecture et d'implantation choisis. Le terme implantation correspond ici, non seulement à la distribution et à l'ordonnancement des calculs et à la distribution des communications inter-processeur, mais aussi, et c'est le point important, à l'ordonnancement des communications inter-processeur éventuellement routées. Nous présentons ensuite une heuristique gloutonne et avec retour arrière, qui choisit parmi l'ensemble précédent une implantation sous-optimale en termes de durée d'exécution, en prenant en compte l'allongement du chemin critique et la marge d'ordonnancement. Cette heuristique est statique c'est-à-dire que les choix d'implantation sont effectués à la compilation et lors de l'exécution temps-réel. Nous la comparons ensuite sur quelques exemples d'algorithmes d'application à des heuristiques proposées dans la littérature.*

Mots-clefs : Formalisation, Implantation Parallèle, Heuristique de Distribution et Ordonnancement, Ordonnancement glouton de type liste.

Abstract *In this paper, we present a relational formalism based on graphs theory to build all the possible implementations of an algorithm onto a parallel architecture. Here, the term implementation means, not only as usual, the distribution and the scheduling of the computations, as well as the distribution of the inter-processor communications, but also, and this is a crucial issue, the scheduling of the inter-processor communications, possibly routed. We develop a greedy heuristic with back-tracking, which chooses among the previous set, a sub-optimal implementation taking into account the critical path lengthening and the schedule flexibility. This heuristic is static, i.e. the implementation choices are made during the compilation and not during the real-time execution. Then we compare our heuristic with others presented in the literature on several application algorithm examples.*

Keywords : Formalization, Parallel Implementation, Distribution and Scheduling Heuristic, Greedy List Scheduling Algorithm.

1 Introduction

Nous nous intéressons à l'implantation d'algorithmes d'applications parallèles [6] complexes qui interagissent avec leur environnement et sont soumises à des contraintes temps-réel [3].

En utilisant la méthodologie d'aide à l'implantation [1] A^3 [16], nous cherchons une implantation de durée d'exécution minimale. Pour ce faire, il faut exploiter à la fois le parallélisme potentiel de l'algorithme d'application et le parallélisme disponible de l'architecture. Ces parallélismes sont

[1] Adéquation Algorithme Architecture

caractérisés à l'aide d'un formalisme commun reposant sur la théorie des graphes. Le graphe de l'algorithme peut être obtenu soit par spécification directe, soit par compilation d'un programme écrit à l'aide d'un langage flot de données tel que le langage SIGNAL [4]. C'est un graphe orienté qui induit un ordre partiel sur des opérations à exécuter et caractérise ainsi le parallélisme potentiel de l'algorithme. Le graphe de l'architecture est non orienté et représente un réseau d'unités de calcul et d'unités de communication qui caractérise le parallélisme disponible de la machine.

L'implantation désigne ici la distribution des calculs et des communications sur l'architecture comme c'est toujours le cas, mais aussi l'ordonnancement des communications, ce qui est plus rarement le cas.

Dans une première partie, nous présentons les modèles d'algorithme, d'architecture et d'implantations choisis. A l'issue de cette partie, l'ensemble des implantations possibles d'un algorithme sur une architecture donnée a été décrit. Choisir parmi cet ensemble la meilleure implantation solution, en termes de durée d'exécution, est un pb NP-difficile [5, 7]. Nous présentons dans la seconde partie du papier une variante plus efficace avec retour-arrière de l'heuristique proposée dans [13], choisissant une solution sous-optimale parmi l'ensemble des implantations.

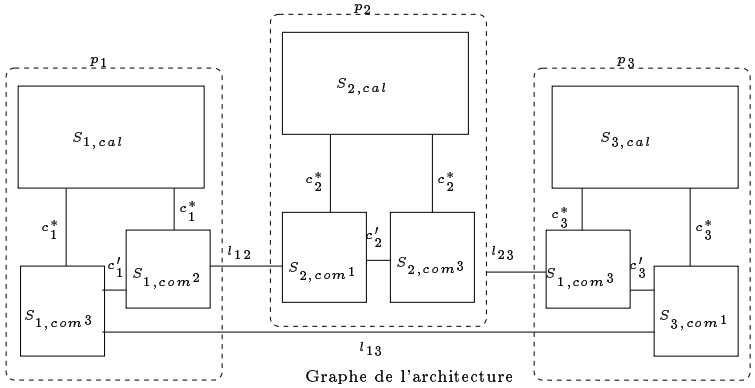
2 Modèle d'architecture

Le modèle d'architecture choisi est le modèle développé défini dans [2] (cf. schéma ci-après). C'est un réseau d'unités de calcul et d'unités de communication connectés par des liens point à point qui sont des médias physiques de communication. Dans ce réseau, il n'existe pas de lien entre les unités de calcul. Pour communiquer, ces unités passent par l'intermédiaire des unités de communication. Nous supposons que l'architecture est homogène, c'est-à-dire que toutes les unités de calcul ont les mêmes caractéristiques ainsi que toutes les unités de communication et les liens inter-unité. Nous supposons aussi que l'architecture n'est pas nécessairement complètement connectée, ce qui aura pour conséquence des coûts de *transfert de données*, entre opérations du graphe de l'algorithme affectées à des unités de calcul différentes, variables suivant le nombre de liens utilisés pour un même transfert. Enfin nous supposons qu'il y a une seule unité de calcul par processeur.

p_i : processeur
$S_{i,cal}$: unité de calcul de p_i
S_{i,com^j} : unité de communication de p_i pour communiquer avec p_j.

c_i^* : lien de p_i entre l'unité de calcul et l'unité de communication
c_i' : lien de p_i entre les unités de communication.

l_{ij} : lien entre l'unité de communication de p_i et l'unité de communication de p_j



Graphe de l'architecture

Beaucoup d'heuristiques prennent comme modèle d'architecture simplement un réseau de processeurs. Nous n'avons pas retenu ce modèle car il ne permet pas d'ordonnancer les communications inter-processeur et comme dans notre cas, l'ordonnancement est suivi de génération d'exécutif distribué et temps-réel, nous devons les modéliser finement.

3 Modèle d'algorithme d'application

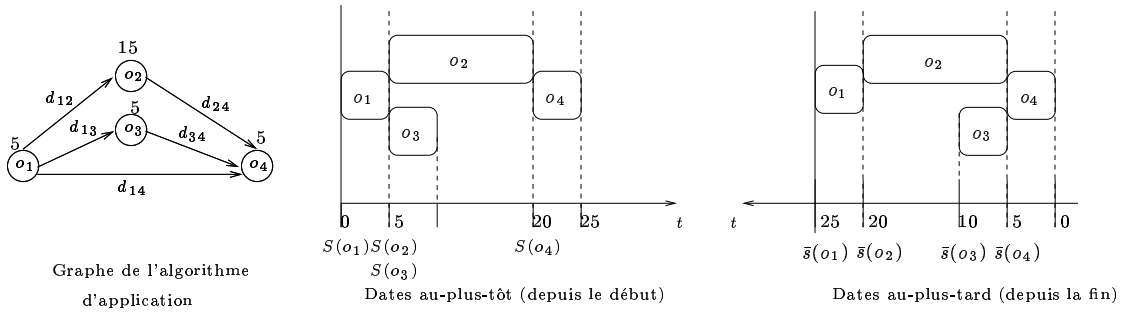
Les applications réactives numériques interagissent avec leur environnement de manière discrète, donc répétitive (répétition d'une séquence : lecture capteur-calculs-écriture actionneur). On les modélise

par un graphe acyclique de *dépendances de données* entre *opérations*, constitué d'un motif infiniment répété, dont la factorisation [14] correspond à un graphe flot de données. Chaque dépendance de données inter-motif apparaît dans la forme factorisée du graphe sous forme d'un cycle, qu'on marque d'un sommet spécial appelé *retard* (correspondant au z^{-1} des automaticiens), afin que le cycle apparent ne soit pas interprété comme un cycle de dépendance de données.

Notation Soit $G_{al} = (O, D)$, le graphe de l'algorithme. O est l'ensemble des Opérations du graphe comprenant les opérations de calcul et les opérations retard et D est l'ensemble des Dépendances de données entre ces opérations. L'ensemble des arcs privé des arcs issus ou provenant d'opérations retard, induit un ordre partiel sur les opérations de calcul d'un même motif et l'ensemble des arcs restant induit un ordre partiel entre opérations appartenant à des motifs différents.

Etiquetage du graphe Une fois que l'on a caractérisé l'architecture cible, on associe une durée d'exécution à chaque opération de G_{al}. Les arcs sont alors étiquetés par le type et la quantité de données du transfert qu'ils modélisent, mais pas par la durée car on ne la connaît pas a priori. En effet, si par exemple, ce transfert est effectué sur la même unité de calcul (transfert direct entre mémoires de la même unité de calcul), sa durée sera négligée, par contre si ce transfert est effectué entre mémoires d'unités de calcul différentes, par l'intermédiaire de un ou plusieurs liens (à cause du routage éventuel), alors sa durée dépendra du nombre de liens utilisés.

Calcul de dates Une fois que l'on a étiqueté les opérations par leur durée, on définit les dates au-plus-tôt (depuis le début) et au-plus-tard (depuis la fin) pour chacune de ces opérations (cf. schéma ci-après). Ces calculs de dates ne prennent pas en compte les durées des transferts car, comme on vient de le voir, on ne les connaît pas a priori. Les transferts seront pris en compte par l'heuristique d'optimisation de la distribution et de l'ordonnancement décrite plus loin.



Graphe de l'algorithme d'application

Dates au-plus-tôt (depuis le début)

Dates au-plus-tard (depuis la fin)

4 Modèle d'implantations

L'implantation d'un algorithme sur une architecture est formalisée comme la composition de trois relations : le *routage*, la *distribution* et *l'ordonnancement* que nous allons présenter successivement.

4.1 Le routage

Le routage est une relation qui modifie le graphe de l'architecture afin de rendre complètement connectées entre elles les unités de communication de ce graphe. Pour ce faire, on construit l'ensemble des routes possibles pour aller d'une unité de communication à une autre unité de communication. Une première conséquence du routage est l'augmentation du parallélisme effectif au niveau des liens inter-unités de communication et une deuxième conséquence du routage est que la durée d'un même transfert de données est variable suivant le nombre de liens utilisés pour effectuer ce transfert s'il y a plusieurs routes possibles pour effectuer le transfert. Il existe un unique graphe routé pour un graphe d'architecture donné.

4.2 La distribution

La distribution, souvent appelée *placement*, répartit les opérations de calcul et les retards sur les unités de calcul et répartit les dépendances de données sur les unités de calcul (dans le cas où les opérations dépendantes sont distribuées sur la même unité de calcul), et sur les unités de communication et les liens (dans le cas où les opérations dépendantes sont distribuées sur des unités de calcul différentes). Pour un algorithme d'application donné, il existe un nombre fini de graphes distribués sur une architecture donnée. Lors de la distribution, des opérations, appelées *opérations de communication*, sont rajoutées au graphe de l'algorithme distribué, entre les opérations qui sont dépendantes et qui sont distribuées sur des unités de calcul différentes. Ainsi une dépendance de données, dans le cas où les opérations sont situées sur des unités de calcul différentes séparées par un seul lien inter-processeur par exemple, a comme support deux unités de communication et trois liens dont un inter-processeur et deux intra-processeur.

L'intérêt de rajouter ces opérations est de pouvoir modéliser avec précision les communications inter-processeur et surtout de pouvoir les ordonnancer ce qui n'est pas faisable si l'on ne considère qu'un réseau de processeurs.

4.3 L'ordonnancement

A l'issue de la distribution, un ensemble d'opérations (de calcul et de retard) et d'arcs de dépendances entre ces opérations (sous-graphe du graphe de l'algorithme d'application initial qui est un ordre partiel), est associé à chaque unité de calcul du graphe de l'architecture. De même, à chaque unité de communication est associé un ensemble d'opérations de communication. Chaque unité de calcul (resp. de communication) est un automate purement séquentiel, l'ordre d'exécution des opérations doit donc y être total.

L'ordonnancement consiste à renforcer chaque ordre partiel affecté à chaque unité pour en faire un ordre total en ajoutant au graphe distribué des arcs d'ordonnancement. A un graphe distribué correspond en général plusieurs, mais un nombre fini, de graphes ordonnancés.

Les arcs d'ordonnancement correspondent, d'une part, à la fermeture transitive des arcs du graphe de l'algorithme initial, et d'autre part à des arcs ajoutés entre opérations indépendantes. Pour ces opérations indépendantes, l'ordonnancement est déterminé par les dates et les durées de ces opérations comme on le verra dans la section suivante.

A l'issue de l'ordonnancement, le parallélisme disponible de l'algorithme est réduit au parallélisme effectif de l'architecture. Plus précisément, le parallélisme intra-unité est supprimé et le parallélisme restant se situe entre les unités (de calcul ou de communication).

En composant les trois relations précédemment définies, on décrit l'ensemble des implantations possibles d'un algorithme sur une architecture donnée. Choisir parmi cet ensemble l'implantation de durée minimale est un problème NP-difficile. Nous proposons donc dans la suite une heuristique de distribution/ordonnancement qui donne une solution approchée au problème de l'implantation.

5 Heuristique de distribution et d'ordonnancement

Pour choisir parmi l'ensemble des implantations précédemment décrit, on construit une solution approchée à l'aide d'une fonction de coût. Contrairement à la formalisation où toutes les opérations sont d'abord distribuées avant d'être ordonnancées, avec la méthode approchée chaque opération est distribuée puis ordonnancée avant de distribuer et d'ordonnancer l'opération suivante. Ceci est réalisé en respectant l'ordre partiel induit par les précédences du graphe de l'algorithme d'application.

La méthode approchée choisie est statique : les choix de distribution et d'ordonnancement sont réalisés à la compilation et non à l'exécution. Ceci permet de générer à l'aide du logiciel d'aide à

l'implantation SynDEx [17] [2] supportant la méthodologie A^3, un exécutif [12] principalement statique et donc à faible coût. Ceci est possible, car comme on l'a vu à la section 3, la structure du graphe de l'algorithme est connue a priori et ce graphe est étiqueté par les durées d'exécution des opérations et par les transferts de données.

L'heuristique que nous allons présenter est gloutonne et de type liste [19] enrichie par une méthode de retour arrière. L'heuristique de type liste construit rapidement une solution de bonne qualité [15] et cette solution est ensuite améliorée par une méthode de retour arrière.

A chaque étape de l'heuristique de type liste, on construit une liste ordonnée d'opérations ordonnançables auxquelles on a assigné des priorités. Une opération ordonnançable, ou encore candidate à la distribution et à l'ordonnancement, est une opération dont tous les prédécesseurs sont déjà distribués et ordonnancés. Si cette opération est un retard alors tous ses successeurs doivent aussi avoir été distribués et ordonnancés, ceci afin de respecter l'ordre partiel de l'algorithme. La priorité d'une opération est déterminée par une fonction de coût qui évalue l'urgence de distribuer et d'ordonnancer cette opération. Une fois la liste construite, les trois macro-étapes suivantes sont répétées jusqu'à ce que la liste soit vide : (1) sélectionner le meilleur processeur pour chacune des opérations ordonnançables, (2) sélectionner le couple (opération, processeur) ayant la plus haute priorité pour l'ordonnancement et (3) mise à jour de la liste. Afin d'évaluer notre heuristique, nous allons la comparer à d'autres heuristiques gloutonnes de type liste.

5.1 Algorithme proposé

Nous présentons tout d'abord l'heuristique gloutonne et sans retour-arrière utilisée dans la méthodologie A^3 et définie dans [13] qui construit la solution initiale d'implantation puis nous présentons la méthode de retour-arrière qui améliore la solution initiale.

5.1.1 Distribution/Ordonnancement initial

Soit $O^{(n)}_{cand}$, l'ensemble des opérations candidates à la distribution et à l'ordonnancement à l'étape n de l'heuristique. Comme l'heuristique est gloutonne, le nombre d'étapes est égal au cardinal de l'ensemble des opérations de G_{al}.

Soient $o_i \in O^{(n)}_{cand}$, $p_j \in P$. Pour tout couple $(o_i, p_j) \in O^{(n)}_{cand} \times P$, on calcule la fonction de coût (appelée *pression d'ordonnancement*) suivante : $\sigma^{(n)}(o_i/p_j) = S^{(n)}(o_i/p_j) + \bar{s}(o_i) - \mathcal{R}^{(n-1)}$. $S^{(n)}(o_i/p_j)$ est la date de début au-plus-tôt (depuis le début) de o_i sur p_j. Cette date prend en compte les communications entre o_i et ses prédécesseurs éventuellement situés sur des processeurs différents. $\bar{s}(o_i)$ est la date de début au-plus-tard (depuis la fin) (définie dans la section 3). $\mathcal{R}^{(n-1)}$ est la longueur partielle d'ordonnancement à l'étape $n-1$. La pression d'ordonnancement mesure à la fois la marge d'ordonnancement et l'allongement du chemin critique, correspondant à la longueur finale d'ordonnancement, et induit une priorité d'ordonnancement pour chaque opération.

Pour chaque o_i, on recherche le meilleur processeur p_i^{best} (macro-étape (1) de l'heuristique de liste): $\forall o_i \in O^{(n)}_{cand} \quad \sigma^{(n)}_{opt}(o_i/p_i^{best}) = \min_{p_k} \sigma(o_i/p_k)$. On obtient alors un meilleur couple $= (o_i, p_i^{best})$.

Parmi ces meilleurs couples (opération, processeur), on choisit celui qui a la pression la plus élevée (macro-étape (2) de l'heuristique de liste) : $\sigma^{(n)}_{best}(o_./p_.^{best}) = \max_{o_i \in O^{(n)}_{cand}} \sigma^{(n)}_{opt}(o_i/p_i^{best})$

$o_.$ est alors distribuée et ordonnancée sur $p_.^{best}$. S'il existe plusieurs couples (o_i, p_i^{best}) tels que $\sigma^{(n)}_{opt}(o_i/p_i^{best}) = \sigma^{(n)}_{best}(o_./p_.^{best})$, alors un couple est choisi au hasard parmi les couples équivalents pour être ordonnancés à l'étape n et les autres couples sont mémorisés et serviront à déterminer la profondeur du retour-arrière.

[2] http://www-rocq.inria.fr/syndex

L'opération ordonnancée est retirée de la liste et on ajoute à la liste les successeurs ordonnançables (macro-étape (3) de l'heuristique de type liste) et on passe à l'étape $n+1$. Le procédé est itéré jusqu'à ce que toutes les opérations de G_{al} aient été distribuées et ordonnancées.
Remarque La longueur partielle d'ordonnancement n'est en pratique jamais calculée car on compare des pressions et donc les $\mathcal{R}^{(n-1)}$ s'annulent.

5.1.2 Amélioration de l'implantation par retour-arrière

L'algorithme de distribution/ordonnancement glouton a réalisé un premier distribution/ordonnancement qu'on appelle solution initiale. A une étape donnée de l'heuristique, il peut exister plusieurs solutions partielles équivalentes que la fonction de coût, i.e. la pression d'ordonnancement, n'a pas permis de départager. Un choix aléatoire est alors effectué et ce sont ces choix qui conduisent à une solution initiale. Le retour arrière permet de supprimer l'aléatoire dans l'algorithme de distribution/ordonnancement en évaluant, en termes de durée d'exécution, des solutions initiales construites à partir de chacun des couples équivalents. Ces derniers vont constituer le voisinage que le processus de retour-arrière va explorer. Le voisinage est ici déterminé au fur et à mesure du déroulement du processus d'ordonnancement, ce qui n'est pas le cas de l'algorithme FAST décrit dans [11] qui détermine avant l'ordonnancement les nœuds sur lesquels il va falloir revenir.

5.2 Comparaison d'heuristiques sur des exemples d'application

Dans [1], Ahmad et Kwok distinguent trois catégories d'heuristique :

1. **BNP** -Bounded Number of Processors-. Le nombre de processeurs est supposé fini, l'architecture complètement connectée et le routage et l'ordonnancement des communications ne sont pas pris en compte. Un exemple d'algorithme d'ordonnancement de cette catégorie est l'algorithme ETF (Earliest Time First) qui choisit comme nœud le plus prioritaire celui qui a la plus petite date de début. Nous comparons cet algorithme avec d'autres de la catégorie suivante sur les exemples fig 1 et fig 2.

2. **UNC** -Unbouded Number of Clusters- [18]. Le nombre de processeurs est supposé infini, l'architecture est complètement connectée et ni le routage ni l'ordonnancement des communications inter-processeur ne sont effectués. DCP (Dynamic Critical Path), EZ (Edge Zeroing) et MD (Mobility Directed) [10] appartiennent à cette catégorie et nous allons les comparer sur les exemples fig 1 et fig2. DCP distribue et ordonnance en priorité le nœud à la fois le plus critique et qui engendre la communication la plus élevée vis à vis de ses successeurs. EZ ordonne les opérations par coûts de communication décroissants. Enfin à chaque étape MD distribue et ordonnance un nœud critique sur le processeur qui a le temps libre suffisant pour l'accueillir. Au besoin ce processeur peut décaler les opérations pour intercaler le nœud si l'ordre partiel initial est maintenu.

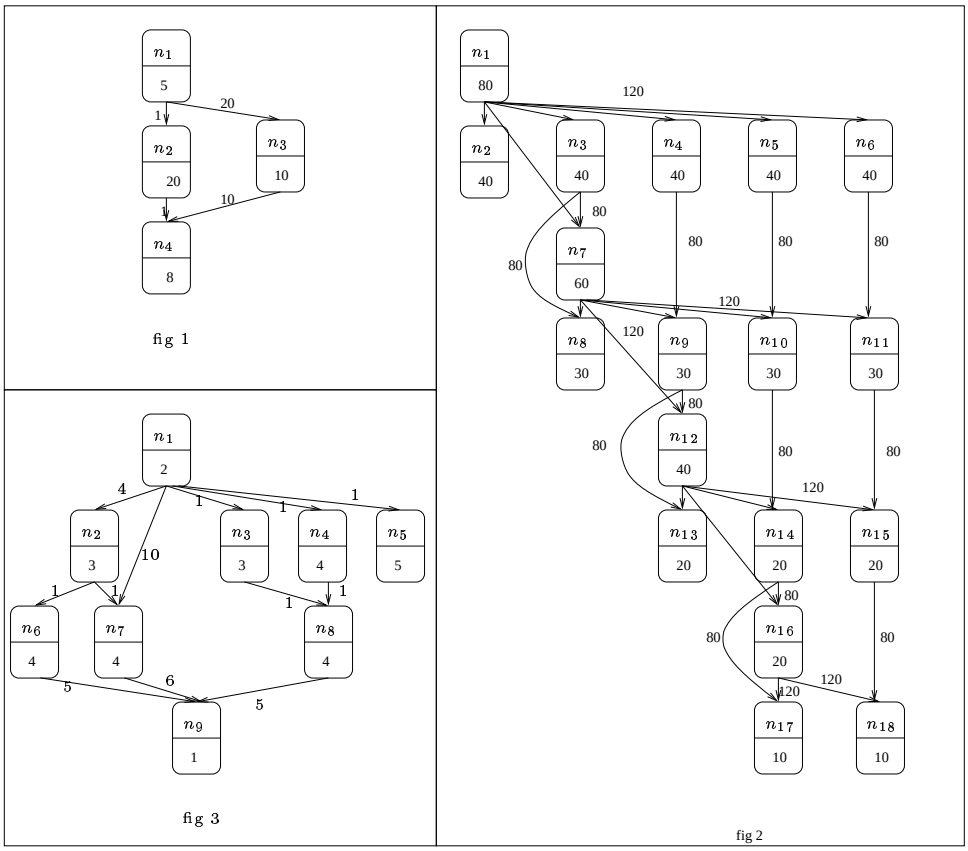
3. **APN** -Arbitrary Processor Network-. La topologie du réseau est arbitraire. Il existe très peu d'algorithmes appartenant à cette catégorie. On distingue les heuristiques : (a) routant et distribuant les communications sur les liens, et (b) routant, distribuant et ordonnançant les communications. L'heuristique que nous allons étudier appartient à cette dernière catégorie avec une hypothèse de plus, à savoir que les communications routées ont des coûts de transfert de données variables suivant le nombre de liens empruntés pour effectuer la communication, alors que les autres heuristiques de cette catégorie supposent des transferts constants quel que soit le nombre de liens utilisés. Nous comparerons sur l'exemple 3 notre heuristique à trois de cette catégorie : les algorithmes BSA (Bubble Scheduling Algorithm), BU (Bottom Up), et MH (Mapping Heuristic) présentés [8, 9]. BSA consiste à recueillir les opérations ordonnançables sur un processeur pivot qui les redistribue ensuite. BU distribue et ordonnance les nœuds critiques

sur un même processeur et distribue et ordonnance les nœuds restants sur les autres processeurs en optimisant l'équilibrage de la charge sur chacun de ces processeurs. Enfin pour l'algorithme d'ordonnancement MH, le nœud ordonnançable prioritaire est celui qui a le plus grand static b-level, i.e. la plus grande somme des durées d'exécution de ce nœud inclus jusqu'à un nœud de sortie (nœud sans successeur).

Nous allons comparer sur des exemples notre heuristique avec celles décritent dans la littérature et présentées brièvement ci-dessus. Bien que certaines n'appartiennent pas tout à fait à la même classe que la nôtre, nous restreindrons nos hypothèses pour se retrouver dans le même cas qu'elles.

Soit σ, notre heuristique initiale et $\sigma + RA$, l'heuristique avec retour arrière. Pour éviter la confusion entre durée d'exécution de l'algorithme d'application et durée d'exécution de l'algorithme de distribution/ordonnancement, nous utiliserons dans la suite, le terme longueur d'ordonnancement pour désigner la durée d'exécution de l'algorithme d'application.

Exemple 1 (cf. fig. 1)



Voici les résultats des longueurs d'ordonnancement du graphe de l'algorithme d'application de la figure 1 en utilisant les données de [10]. Les algorithmes d'ordonnancement comparés sont de catégorie BNP et UNC, ils supposent donc une architecture complètement connectée et ne prennent pas en compte l'ordonnancement et le routage des communications. Le nombre de processeurs est égal à deux.

Type d'algorithmes	ETF	EZ, MD	σ
longueur d'ordonnancement	43	35	34

On constate que notre heuristique donne la plus petite longueur d'ordonnancement pour cet exemple.

Exemple 2 (cf. fig. 2) Les algorithmes d'ordonnancement comparés dans [10] appartiennent aux deux premières catégories : BNP et UNC. Ni le routage, ni l'ordonnancement des communications ne sont donc effectués. L'architecture est un réseau de quatre processeurs, elle est complètement connectée. Pour ne pas être pénalisée par le routage et l'ordonnancement des communications qu'effectuent σ, nous avons choisi une architecture avec deux liens entre chaque processeur.

Type d'algorithmes	EZ	ETF	MD	DCP	σ
longueur d'ordonnancement	600	520	460	440	450

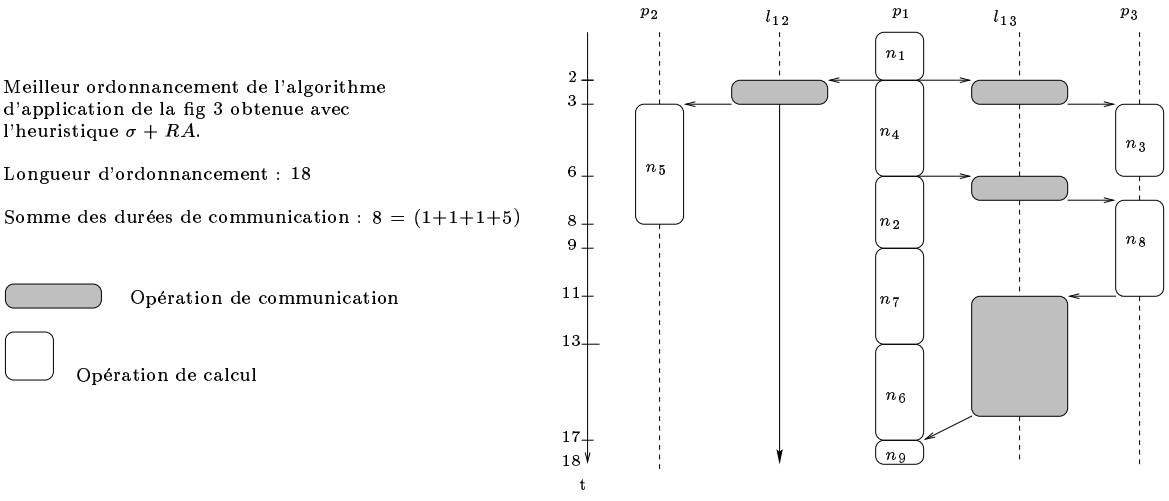
Dans les conditions ainsi définies, notre algorithme arrive en deuxième position pour cet exemple.

Exemple 3 (cf. fig. 3) Les algorithmes d'ordonnancement comparés dans cet exemple appartiennent à la catégorie APN : l'architecture n'est pas forcément complètement connectée, les communications peuvent donc être routées et enfin les communications sont ordonnancées.

La topologie choisie dans l'exemple 3 présenté dans [8, 9] est un anneau à quatre processeurs. Le routage est donc nécessaire mais contrairement à l'algorithme σ, les trois autres algorithmes d'ordonnancement comparés BSA, BU et BH, supposent des coûts de transfert constants quel que soit le nombre de liens utilisés pour ce transfert. Voici les résultats issus de [9, 8] :

Type d'algorithmes	Bubble	BU	MH	σ	$\sigma + RA$
longueur d'ordonnancement	16	24	20	20	18
Somme des durées des communications	11	27	16	14	8
Total	27	51	36	34	26

Bubble est le meilleur algorithme pour la minimisation de la longueur de l'ordonnancement de l'exemple 3, avec une longueur de 16. $\sigma + RA$ donne une longueur d'ordonnancement égale à 18, ce qui lui confère la place de deuxième. Si l'on compare les algorithmes d'ordonnancement non pas sur la longueur d'ordonnancement mais sur la somme de la longueur d'ordonnancement et des durées des communications, l'algorithme $\sigma + RA$ est le meilleur pour cet exemple (cf. schéma ci-après). On peut aussi constater que l'algorithme σ sans retour arrière donne des résultats honorables. En effet, il se place en troisième position pour la longueur de l'ordonnancement et il se place également troisième pour la somme de la longueur d'ordonnancement et des durées des communications.



Exemple 4 L'exemple 4 permet de comparer σ et $\sigma + RA$. Cet exemple est un graphe comprenant des retards et donc comprenant des cycles. Le tableau suivant donne en colonne, l'étape de l'heuristique où des opérations sont équivalentes en termes de fonction de coût, et en ligne, le nombre

d'opérations équivalentes à l'étape correspondante. Ainsi, à l'étape 24, il existe quatre opérations qui ont même résultat par la fonction de coût et c'est le retour arrière sur la quatrième opération équivalente qui donne la plus petite longueur d'ordonnancement. Ainsi, sur cet exemple, le retour-arrière améliore la longueur d'ordonnancement de 22 %.

Etapes de l'heuristique/no de cand eq	1	2	3	4
5	77959	77959	77959	
6	77959	77959		
24	77959	75725	72671	**60383**
25	77959	72671	78815	
28	77959	65671		
31	77959	77959		
32	77959	77959		

6 Conclusion

Grâce à une formalisation précise des modèles d'algorithme d'application et d'architecture, nous avons obtenu un modèle d'implantation où à la fois les calculs et les communications sont distribués et ordonnancés. Le modèle d'implantation ainsi obtenu prend en compte des architectures réalistes, ce qui est important pour la génération d'exécutif temps-réel qui peut découler de l'implantation. A l'issue de la formalisation, nous avons contruit l'ensemble des implantations possibles d'un algorithme d'application donné sur une architecture donnée. Pour faire un choix d'implantation parmi cet ensemble, nous avons proposé une extension avec retour-arrière d'une heuristique gloutonne rapide. La rapidité de l'heuristique gloutonne permet de faire du prototypage rapide d'applications soumises à des contraintes temps-réel. Les performances des heuristiques de distribution/ordonnancement avec et sans retour-arrière ont été évaluées en comparant, avec d'autres heuristiques de distribution/ordonnancement, les longueurs d'ordonnancement obtenues de quelques exemples d'algorithmes d'application. Ces performances sont satisfaisantes sur ces exemples, il reste maintenant à étendre ces résultats sur d'autres types de graphes.

References

[1] I. Ahmad and Y. K. Kwok. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *International Symposium on Parallel Architecture, Algorithms and Networks*, pages 207–213, Beijing China, June 1996.

[2] C. Aiglon, C. Lavarenne, Y. Sorel, and A. Vicard. Utilisation de SynDEx pour le traitement d'images temps-réel. Rapport de Recherche 2968, INRIA, Septembre 1996.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-times systems. Rapport de recherche 1445, INRIA, June 1991.

[4] A. Benveniste, M. LeBorgne, and P. LeGuernic. SIGNAL as a Model for Real-Time and Hybrid Systems. Rapport de recherche 1608, INRIA, Feb 1992.

[5] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement.* Masson, 1988. Etudes et Recherches en informatique.

[6] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterEditions, PARIS, 1993.

[7] Garey and Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[8] Y. K. Kwok and I. Ahmad. Bubble Scheduling : An Efficient Algorithm For Compile-Time Scheduling of Parallel Computations on Messages-Passing Architectures. *Journal of Parallel and Distributed Computing*. to appear.

[9] Y. K. Kwok and I. Ahmad. Bubble scheduling : A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proc of 7th IEEE symposium on Parallel and Distributed Processing*, pages 36–43, Octobre 1995.

[10] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, volume 7, pages 506–521. May 1996.

[11] Y. K. Kwok, I. Ahmad, and J. Gu. FAST : A Low-Complexity Algorithm For Efficient Scheduling of DAGs on Parallel Processors. In *Proceedings of the 25th International Conference on Parallel Processing*, volume 2, pages 150–157, Aout 1996.

[12] C. Lavarenne and Y. Sorel. Documentation du logiciel SynDEx d'aide à l'implantation d'algorithmes sur architectures multi-processeurs sous contraintes temps-réel. http://www-rocq.inria.fr/syndex/.articles/doc/doc/SynDEx42.

[13] C. Lavarenne and Y. Sorel. Performance Optimization of Multiprocessor Real-Time Applications by Graph Transformations. In *Parallel Computing*, Grenoble, Septembre 1993.

[14] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, vol 14, n.6, 1997.

[15] Z. Lui and C. Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. *Proc. of PARLE'93, 5th international PARLE conference, Munich, Germany, June 14-17*, pages 452–463, Nov. 1993.

[16] Y. Sorel. Massively parallel systems with real time constraints. the "Algorithm Architecture Adequation Methodology". In *Proc. Massively Parallel Computing Systems*, Italy, May 1994.

[17] Y. Sorel. Real-Time Embedded Image Processing Applications using the A^3 Methodology. In *Proc.IEEE International Conference on Image Processing*, Switzerland, Sep. 1996.

[18] T. Yang and A. Gerasoulis. DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–957, 1994.

[19] T. Yang and A. Gerasoulis. List Scheduling with and without Communication Delays. Rutgers University, August 92.