HAL
open science

Problématique du placement de tâches dans MeDLey

Carlos Gamboa dos Santos

To cite this version:

Carlos Gamboa dos Santos. Problématique du placement de tâches dans MeDLey. [Rapport de recherche] RR-3256, INRIA. 1997, pp.44. inria-00073433

HAL Id: inria-00073433

https://inria.hal.science/inria-00073433

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Problématique du placement de tâches dans MeDLey

Carlos Gamboa Dos Santos

Carlos.Gamboa@loria.fr

N■ 3256

Septembre 1997

THÈME 1

Rapport de recherche

ISSN 0249-6399

Problématique du placement de tâches dans MeDLey

Carlos Gamboa Dos Santos
Carlos.Gamboa@loria.fr

Thème 1 — Réseaux et systèmes
Projet RESEDAS

Rapport de recherche n° 3256 — Septembre 1997 — 44 pages

Résumé : La programmation d'applications parallèles basées sur le modèle d'échange de messages est simple à apprendre, mais nécessite un investissement considérable pour développer des applications complexes et obtenir de bonnes performances. Afin d'aider le programmeur dans ce contexte, nous proposons un environnement de développement et d'exécution, partiellement réalisé, composé de quatre parties principales: un module gérant la communication des messages, un placeur de tâches, un analyseur de traces et un module lié à l'administration de réseaux. Ce document a pour but de présenter les problèmes spécifiques au placement au sein de cet environnement en effectuant d'abord un survol général du domaine de placement de tâches, puis une description plus détaillée de différents modules de l'environnement.

Mots-clé : Placement de tâches, environnement, échange de messages, adminstration de réseaux, performances

(Abstract: pto)

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Task placement problems related to MeDLey

Abstract: Programming parallel applications using the message passing paradigm is easy to learn, but requires far more resources to develop complex applications and to achieve good execution performances. To help the programmer, we propose a development and run-time environment, which is partly realised, built upon four major components: a message management module, a task placement module, a trace analysis module and a network management module. The aim of this document is to present the placement problems inherent to our environment. Therefore we give a general survey of task placement, followed by a detailed description of the four modules.

Key-words: Task placement, environment, message passing, network management, performances

Table des matières

1 Introduction **5**
1.1 Architecture matérielle 5
1.2 Modèle de programmation 7

2 Principes du placement **9**
2.1 Introduction 9
2.2 Placement statique 11
2.2.1 Algorithme optimal 12
2.2.2 Algorithme sous-optimal 13
2.3 Placement dynamique 14
2.3.1 Politique d'information 15
2.3.2 Politique de transfert 15
2.3.3 Politique de localisation 17
2.3.4 Algorithmes « classiques » 18
2.3.5 Outils de placement 20
2.4 Conclusion 21

3 Environnement **22**
3.1 Architecture 22
3.2 MeDLey 24
3.2.1 Modèle M-SPMD 24
3.2.2 Langage de spécification 25
3.2.3 Graphe des communications 26
3.2.4 Relation de précédence 27
3.3 Administration de réseaux 28
3.4 Analyse 29
3.5 PlaTo 30
3.5.1 Politique de transfert 31
3.5.2 Politique d'information 31
3.5.3 Politique de localisation 32

4 Perspectives **33**
4.1 Pré-placement 33
4.2 Schémas de communication 34

4.3 Outils d'administration 35
4.4 Migration 37

5 Conclusion **38**

1 Introduction

La programmation d'applications parallèles basées sur le modèle d'échange de messages est simple à apprendre, mais nécessite un investissement considérable pour développer des applications complexes et obtenir de bonnes performances.

Afin d'aider le programmeur dans ce contexte, nous proposons un environnement de développement et d'exécution. Cet environnement, partiellement réalisé, se compose de plusieurs parties : `MeDLey` – un langage de spécification des communications qui facilite l'écriture de l'application et optimise les communications, PLATO – un outil de placement de tâches (affectation à des processeurs) afin d'assurer de bonnes performances (temps d'exécution les plus courts possibles) en tenant compte des communciations entre tâches et un analyseur de traces permettant de mieux connaître l'application et ainsi de trouver de meilleurs placements.

Ce document a pour but de présenter les problèmes spécifiques au placement de tâches d'applications parallèles au sein de notre environnement et plus particulièrement PLATO. Pour cela, nous effectuons un survol du domaine de placement de tâches suivi d'une description plus détaillée, du point de vue placement, des modules composant l'environnement.

Mais avant d'aborder la partie principale du document, il est nécessaire de définir le contexte dans lequel s'insère notre travail. La présentation de l'architecture matérielle et du modèle de programmation visés font donc l'objet de ce premier chapitre.

1.1 Architecture matérielle

Dans le domaine du parallélisme il existe actuellement deux approches au niveau de l'architecture matérielle utilisée afin de traiter des applications nécessitant une grande puissance de calcul. Les deux types de machines que nous distinguons sont :

1. **les machines dites parallèles** constituées de plusieurs processeurs – en général identiques – fortement couplés par des communications et synchronisations très rapides. Du point de vue de l'utilisateur une telle

machine est perçue comme étant mono-application, c.-à-d. qu'il semble que seule l'application utilisateur s'exécute sur la machine.

Notons également qu'il existe deux grandes familles de machines parallèles : les massivement parallèles (MPP – *Massively Parallel Processors*) à mémoire distribuée et les multiprocesseurs symétriques (SMP – *Symmetric MultiProcessor*) à mémoire partagée. De plus, ce genre de systèmes possède une quantité de mémoire physique élevée ainsi que des entrées/sorties très performantes ;

2. **les systèmes répartis** formés par des machines autonomes (faiblement couplées), le plus souvent hétérogènes au niveau matériel et réseau, avec des communications plus ou moins lentes. Ces systèmes sont en général multi-utilisateurs et multi-applications, car non-dédiés à un utilisateur ou une application particulière.

   De plus, la définition donnée dans [Tannenbaum 95] implique qu'un utilisateur doit percevoir un système réparti comme une seule machine. Cependant, les systèmes d'exploitation usuels ne fournissent pas encore cette transparence totale aux utilisateurs, forçant ces derniers à lancer les tâches d'une application parallèle sur plusieurs machines de manière explicite.

Notre travail se place dans le contexte des systèmes répartis basés principalement sur des stations de travail reliées par des réseaux. L'utilisation d'une telle architecture pour exécuter des applications parallèles est très répandue, car un tel système est moins cher et plus souple qu'une machine parallèle.

Un réseau de stations possède une mémoire distribuée physiquement, c.-à-d. qu'il existe plusieurs espaces d'adressage : les données ne sont donc pas directement partageables par les tâches d'une même application parallèle. L'échange d'informations (données) s'effectue au moyen de mécanismes logiciels, qui sont le plus souvent indépendants du système d'exploitation. La figure 1 montre un ensemble hétérogène de machines reliées par des réseaux de types différents. Notons qu'il est possible de concevoir un système réparti formé de stations de travail et de machines parallèles, bien qu'il soit assez difficile de programmer efficacement pour une telle configuration.

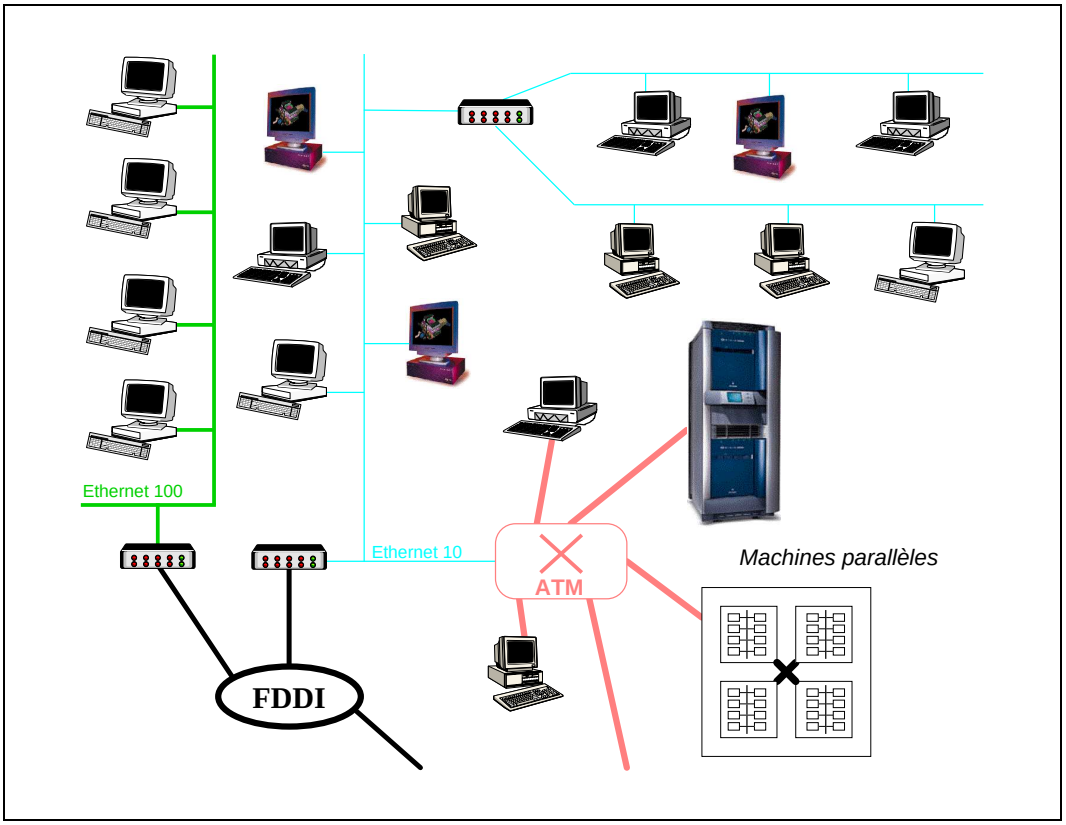


FIG. 1 – *Exemple de réseaux et machines hétérogènes*

1.2 Modèle de programmation

Le modèle de programmation le plus utilisé dans le cadre d'un système réparti est celui du parallélisme de contrôle qui est basé sur des tâches communiquant par échange explicite de messages (*message-passing*). Un échange se caractérise par l'association de deux opérations de base : l'envoi (côté émetteur) et la réception (côté récepteur). Ce modèle a l'avantage d'être relativement simple à apprendre par des utilisateurs non informaticiens. En général, sa mise en œuvre utilise une bibliothèque de communication, comme par exemple PVM [Geist 94] ou le nouveau standard MPI [Gropp 94], offrant des services d'échange de messages entre processus.

Cependant, le modèle de programmation par échange de messages n'a pas que des avantages. Parmi les problèmes qui subsistent, les plus importants sont les suivants :

1. *gestion des messages*
   l'utilisateur doit gérer lui-même le contenu des messages de manière à garantir leur cohérence, c.-à-d. que les types des données contenues dans le message envoyé correspondent bien aux types définis dans le message attendu ;

2. *délais de transfert des messages*
   les bibliothèques de communication étant par définition générales, le temps d'échange d'un message peut être très élevé. Ce délai est dû aux nombreuses recopies mémoire (tampons intermédiaires) effectuées lors de l'envoi et de la réception d'un message ;

3. *placement des tâches*
   pour obtenir un bon niveau de performances pour une application parallèle (exécution en un temps minimum), il faut placer ses tâches sur un ensemble de machines, de telle sorte que les tâches ne soient pas en concurrence pour des ressources (p.ex. CPU ou réseau) au même moment. En général, cela signifie que chaque tâche s'exécute sur un processeur différent.

Pour résoudre ces problèmes, nous proposons une solution en trois étapes regroupées au sein d'un environnement de développement et d'exécution. La première étape (points 1 et 2 ci-dessus) passe par un langage de spécification de messages, nommé `M`*e*`DL`*ey* [Dillon 96b], qui permet de décrire la structure des messages échangés entre les tâches et d'optimiser les délais de transfert en minimisant les recopies mémoire. La seconde étape est constituée par le placement des tâches d'une application parallèle (module nommé PLATO) en tenant compte, outre les critères classiques tels que la charge machine ou la présence d'activité console, des communications entre ces tâches et la topologie du réseau. La prise en compte de ces deux derniers types d'information se fait en lien avec l'administration de réseaux. La troisième étape consiste dans la quantification de l'application en terme de données échangées afin d'améliorer

le placement. Pour cela nous proposons d'effectuer une analyse *post-mortem* des traces d'exécution.

L'objectif de ce document est de présenter la problématique du placement de tâches d'une application parallèle dans le contexte de `MeDL`*ey* et de son environnement. Pour cela, nous présentons tout d'abord les principes du placement au chapitre 2, suivi au chapitre 3 par l'environnement dont nous proposons le développement. Enfin nous détaillerons les évolutions possibles liées à ce dernier au chapitre 4 avant de conclure.

2 Principes du placement

Ce chapitre a pour but de présenter les principes généraux du placement de tâches, sans prétendre être une description exhaustive du domaine, car la problématique du placement de tâches d'applications parallèles est un sujet donnant lieu à de nombreux travaux ainsi qu'à une littérature abondante. En effet, le placement est certainement une des clés permettant une utilisation performante des machines parallèles.

2.1 Introduction

L'ordonnancement est un mécanisme ou algorithme effectuant la gestion de ressources, qui correspond dans le cadre du parallélisme plus particulièrement à l'allocation de tâches aux processeurs ou machines. Si l'on se réfère aux classifications de l'ordonnancement de tâches proposées dans la littérature [Casavant 88, Folliot 93], nous pouvons distinguer les deux groupes d'ordonnancements suivants tel que présenté à la figure 2 :

- *local*
  cette catégorie regroupe les politiques d'ordonnancement dites temporelles qui ont pour but d'affecter – de manière locale à chaque machine – une tâche à un processeur pendant une tranche de temps donnée. Notons que le terme *ordonnancement* est souvent appliqué de manière restrictive à l'ordonnancement local (temporel) dans le cadre des systèmes d'exploitation. Un cas particulier existe pour les machines parallèles dans lequel

l'ordre d'exécution des tâches peut être défini simultanément pour un ensemble de processeurs d'une même machine parallèle [Scherson 96] ;

– *global*
cet ordonnancement correspond aux algorithmes de placement à proprement parler, c.-à-d. où il s'agit d'affecter avec contraintes un ensemble de tâches à un ensemble de processeurs. Cette opération peut être décrite comme un ordonnancement spatial, car il s'agit de déterminer quelle tâche s'exécute sur quelle machine, avec l'hypothèse que les tâches placées démarrent immédiatement. Dans le cas contraire, les tâches sont liées entre-elles par une relation de précédence d'exécution et il est nécessaire d'effectuer une opération de regroupement afin d'obtenir des groupes de tâches pouvant être placées sans contraintes temporelles [Bouvry 94].

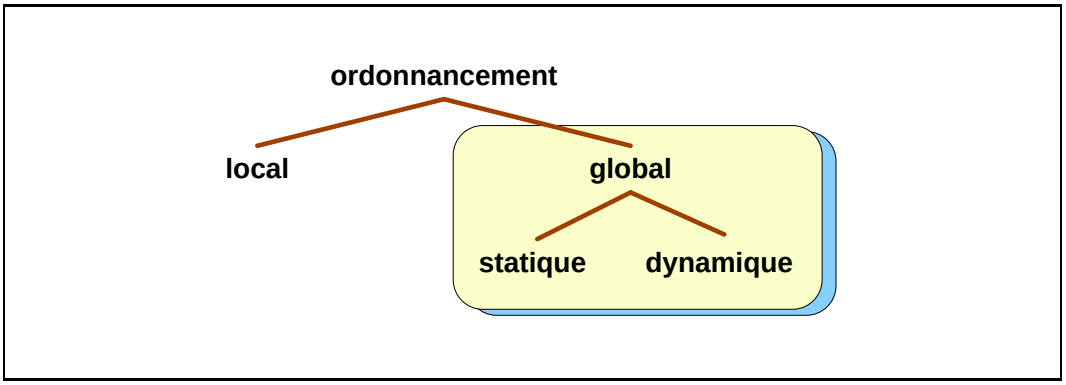


FIG. 2 – *Classes d'algorithmes d'ordonnancement*

L'ordonnancement local des stations de travail est en général fixe et ne peut être facilement influencé. Par conséquent, les outils de placement ne prennent en compte que l'ordonnancement global pour effectuer leur travail, c.-à-d. que les deux ordonnanceurs ne collaborent pas afin d'améliorer le placement.

Les algorithmes de placement sont classés en deux grandes catégories : le placement statique et le placement dynamique. La principale différence entre ces deux placements se situe au niveau du moment de calcul de l'affectation des tâches aux machines. Dans le cas statique, les processeurs sont attribués aux tâches de manière définitive à la compilation. Dans le cas dynamique, cette

affectation est effectuée au moment du démarrage ou durant l'exécution en tenant compte de l'état courant des machines.

La suite du chapitre est consacrée à la présentation de ces deux grandes catégories en mettant plutôt l'accent sur le placement statique, car c'est la technique retenue pour l'outil de placement PLATO de notre environnement.

2.2 Placement statique

Le placement statique a pour but de diminuer le temps d'exécution d'une application en affectant les tâches aux processeurs (ou machines) disponibles. Cette affectation est calculée à la compilation, c.-à-d. avant l'exécution de l'application parallèle, impliquant les contraintes suivantes :

- architecture du réseau de machines cible figée ;
- comportement de l'application connu et statique, c.-à-d. que le nombre de tâches est fixé au lancement pour toute la durée de l'exécution ;
- les interférences externes lors de l'exécution de l'application sont négligées.

Compte tenu de ces contraintes, le placement statique est principalement utilisé dans le cadre d'un petit nombre d'applications ayant un comportement bien connu s'exécutant sur des machines parallèles dédiées au calcul.

Le problème qui consiste à trouver un placement optimal pour les tâches sur une architecture donnée est dans le cas général NP-complet, c.-à-d. qu'il n'existe pas d'algorithme qui résout le problème en un temps borné par une fonction polynômiale de la taille du problème [Garey 79]. Dans de nombreux cas, la recherche de la solution approchée est préférable à l'optimum afin de diminuer les temps de calcul du placement.

La figure 3 présente une classification des caractéristiques des algorithmes de placement statiques. Il existe dans la littérature des synthèses [Muntean 91, Belhamissi 91] présentant de manière plus ou moins détaillée les divers algorithmes existants ainsi que leurs modèles sous-jacents [Norman 93].

Au premier niveau de la classification présentée nous distinguons d'un côté les algorithmes calculant une solution *optimale* et de l'autre côté ceux qui produisent une solution *sous-optimale*, mais néanmoins globalement acceptable.

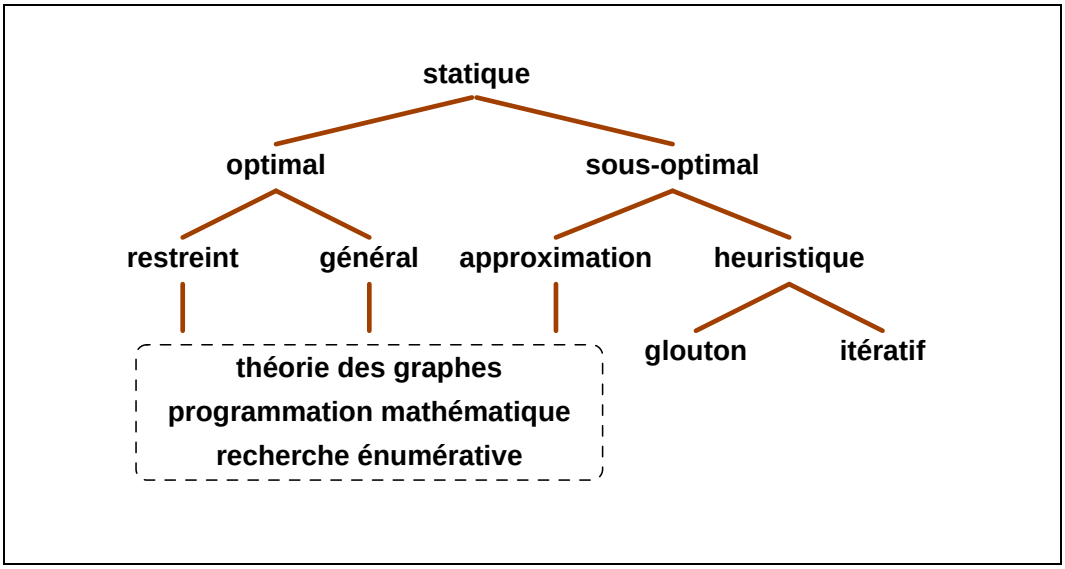


FIG. 3 – *Caractéristiques des algorithmes de placement statique*

2.2.1 Algorithme optimal

Dans cette catégorie, il faut différencier le cas où l'algorithme *restreint* la structure du problème posé afin d'obtenir l'optimum en un temps polynômial et le cas où le problème est posé dans un contexte plus *général*. Les modèles de calcul utilisés pour trouver la solution sont issus des domaines de la théorie des graphes, de la programmation mathématique ou de la recherche énumérative.

Parmi les algorithmes basés sur les graphes, une méthode consiste à déterminer les coupes minimales d'un graphe contenant les tâches, les processeurs et les liens de communication. Cette solution permet de trouver une solution optimisant les communications et les temps d'exécution. L'inconvénient de cette approche est de ne pouvoir représenter des contraintes liées aux applications ou aux architectures.

Une solution hybride entre la théorie des graphes et la recherche exhaustive consiste à calculer la projection (recherche d'un homomorphisme faible) du graphe des tâches sur le graphe des processeurs [Monien 90]. La meilleure solution, c.-à-d. qui minimise les coûts de communication et d'exécution, s'obtient par utilisation de l'algorithme d'optimisation A*, issu du domaine de l'intel-

ligence artificielle [Shen 85] combiné à une fonction de coût. En parallélisant cette méthode [Kafil 97], il est possible d'en augmenter les performances.

Les algorithmes basés sur la programmation mathématique, représentent le calcul du placement sous forme d'un problème d'optimisation avec minimisation d'une fonction de coût sous contraintes. L'avantage de cette méthode est de permettre une modélisation très fine du problème posé, mais l'algorithme est exponentiel en temps de calcul avec la taille du problème.

2.2.2 Algorithme sous-optimal

Les algorithmes sous-optimaux, sont divisés en deux grands groupes. D'un côté, les algorithmes par *approximation* utilisent les mêmes modèles que les optimaux pour calculer une solution, mais s'arrêtent dès qu'une solution est dans une certaine limite de l'optimum. De l'autre côté, certains algorithmes tentent de diminuer la complexité du problème en utilisant des *heuristiques* afin d'augmenter la vitesse de recherche d'une solution sans garantir que cette dernière soit toujours la meilleure.

Par ailleurs, il est possible de subdiviser les heuristiques en deux types, même si cette division n'est pas toujours très nette. Premièrement, les algorithmes dits *gloutons* tentent d'atteindre une solution sans remettre en question les choix déjà effectués. Ils ont l'avantage d'être peu coûteux en temps, mais fournissent souvent de moins bonnes solutions que les autres approches.

Comme exemple de ce type d'algorithmes, citons l'algorithme glouton efficace proposé dans [Pellegrini 95], qui est basé sur des heuristiques de bipartitionnement récursif du graphe des tâches et du graphe des processeurs. Le principe est de subdiviser les graphes en des sous-problèmes plus simples, en minimisant les coûts de communication des tâches (modélisés par une fonction de coût) tout en maintenant l'équilibrage de charge des processeurs à un niveau acceptable.

Le second type est constitué par les algorithmes *itératifs* qui partent d'une bonne solution initiale et tentent de l'améliorer par des itérations successives. Citons comme exemples d'algorithmes itératifs [Bouvry 94] : le recuit simulé, les algorithmes génétiques et la recherche tabou.

Le principe du *recuit simulé* est de minimiser une fonction de coût en calculant à chaque étape une nouvelle configuration, par exemple par échange

de paires de tâches. En principe, la solution courante améliore (diminue) la fonction de coût, modélisant le placement. Cependant, il est possible d'accepter une mauvaise solution courante avec une certaine probabilité, qui décroît avec le degré d'avancement de la recherche. Ainsi, il est possible de sortir des minima locaux de la fonction de coût et d'atteindre une meilleure solution. Cet algorithme donne de bons résultats, mais son coût en temps de calcul et la difficulté d'ajuster correctement les divers paramètres le rendent peu utilisable.

L'*algorithme génétique* part d'une population d'individus initiale correspondant à un ensemble de solutions de placement valides et effectue des manipulations (p.ex. croisements et mutations) pour produire de nouveaux individus. Les meilleurs individus – qui ont le coût le plus faible – sont gardés et remplacent les plus mauvais. Cette méthode fournit de très bons résultats pour des problèmes de grande taille, mais est assez difficile à mettre en œuvre.

La *recherche tabou* part d'une solution initiale et tente d'atteindre une meilleure solution à l'aide d'un certain nombre de relations de voisinage (transformations) et d'une fonction coût. À chaque étape l'algorithme calcule plusieurs solutions voisines de la solution courante et sélectionne parmi ces dernières la meilleure, même s'il s'agit d'une moins bonne solution (coût plus élevé) que la solution courante. Afin d'éviter les répétitions (revenir sur des solutions déjà explorées) une mémoire des solutions parcourues, de taille limitée, est introduite : c'est la liste tabou. Cet algorithme donne de meilleurs résultats que les autres algorithmes itératifs, mais est en général plus difficile à mettre en œuvre.

2.3 Placement dynamique

Le but du placement dynamique est de diminuer le temps de réponse moyen du système en affectant les nouvelles tâches aux machines (ou processeurs) ou en migrant les tâches qui s'exécutent déjà. Le placement est dit dynamique, car la prise de décision tient compte de l'état courant du système, tel que la charge des machines. Ce principe est donc très bien adapté au contexte des systèmes répartis où l'état est en perpétuelle évolution.

Il s'en suit, que le principe de fonctionnement d'un tel système de placement est de répondre à la questions suivante :

Faut-il placer ou migrer, et si oui, quelle tâche et sur quelle machine ?

La réponse à cette question permet de décomposer le placement en trois composantes [Zhou 88] :

1. *politique d'information*
   détermine la quantité et le type des informations sur les tâches et les machines mises à la disposition de l'algorithme, ainsi que leur répartition ;

2. *politique de transfert*
   détermine quand et quelle tâche placer ou migrer ;

3. *politique de localisation*
   détermine vers quelle machine placer ou migrer la tâche sélectionnée.

2.3.1 Politique d'information

Un des problèmes liés à la politique d'information est de déterminer l'état courant du système. Cet état est souvent caractérisé par la charge qui s'obtient par combinaison linéaire d'un ou plusieurs indicateurs. Toute la difficulté réside donc, dans le choix de ces derniers, car ils doivent refléter correctement les différents paramètres décrivant l'état du système et de son évolution probable.

En général, les algorithmes utilisent pour des raisons de simplicité l'indicateur de charge UNIX qui est calculé en fonction de la longueur de la file d'attente des processus actifs [Ferrari 88]. Notons que l'indicateur de charge ainsi obtenu dépend fortement du passé et du présent de la machine. Or, pour le placement c'est la charge future qui est importante, car c'est elle qui définit le temps d'exécution des tâches présentes et en cours de placement. Pour obtenir de meilleurs résultats l'algorithme de placement doit donc anticiper la charge d'une machine après le placement ou la migration sur cette dernière [Folliot 93].

2.3.2 Politique de transfert

Suivant l'objectif du système de placement, nous pouvons distinguer les deux cas suivants [Eager 86] répondant au *quand* de la politique de transfert :

- *équilibrage de charge*
  le système tente à tout moment de maintenir la même charge sur l'en-

semble des machines. Ceci implique qu'à chaque évolution de l'état global, la décision de placement doit être remise en cause pour toutes les machines, pouvant provoquer un nombre important de migration de tâches ;

- *partage de charge*
  le système tente de diminuer la charge d'une machine sur-chargée en l'affectant à une machine sous-chargée. Dans ce cas, une remise en cause du placement des tâches locales d'une machine n'est nécessaire que lors d'un changement de l'étal local de la machine sur-chargée.

La politique de transfert se base en général sur la notion de seuils pour déterminer si une machine est sur/sous-chargée ou que la charge est trop déséquilibrée nécessitant par conséquent un déplacement de la charge vers une autre machine. Transférer de la charge signifie simplement lancer ou migrer des tâches sur une machine sous-chargée et arrêter le cas échéant la même tâche sur la machine sur-chargée.

Cependant, dans certaines circonstances, le coût du placement ou de migration d'une tâche peut être plus élevé que son exécution locale. Il est donc nécessaire de restreindre le choix des tâches à transférer uniquement à celles qui sont susceptibles d'améliorer l'état du système, c.-à-d. à celles qui ont une durée de vie pas trop courte : 95 % des tâches durent moins de 8 secondes CPU [Cabrera 86]. Cet ensemble de tâches dites éligibles, est obtenu après application de certains critères de filtrage [Zhou 92, Folliot 93] à toutes les tâches. Citons comme exemples de filtrages possibles : par l'utilisateur (manuel), par le type, par le nom ou par l'âge des tâches.

Notons, que les mécanismes de placement sont d'autant plus intéressants que la charge des machines est fortement déséquilibrée. En effet, placer une application sur un ensemble de machines toutes sous-chargées ou toutes sur-chargées n'apporte aucun gain de performance à l'application en question. Même dans le cas d'une différence de charge importante, la technique qui consiste à migrer des tâches pendant leur exécution afin d'équilibrer la charge, apporte un gain en performance plus faible que celui intuitivement escompté [Eager 88]. Il n'est donc pas nécessaire d'utiliser la migration pour améliorer les performances, mais elle autorise la fiabilisation de l'exécution des applications (reprise sur arrêt machine). Ceci n'est pas inutile dans le contexte d'un réseau de stations de travail, car une machine y est relancée tous les 2,7 jours

en moyenne [Clark 92]. De plus, la migration permet la prise en compte du caractère personnel des stations de travail en déplaçant les tâches gourmandes en temps de calcul, mettant ainsi la puissance de calcul de la station à la disposition de son utilisateur principal.

2.3.3 Politique de localisation

La dernière étape dans l'algorithme de placement consiste à sélectionner une machine devant accueillir la tâche choisie. Afin de mieux caractériser les politiques, nous présentons dans la figure 3 une classification [Bernard 96] des algorithmes de placement dynamique de tâches qui regroupe à la fois la politique d'information et la politique de localisation.

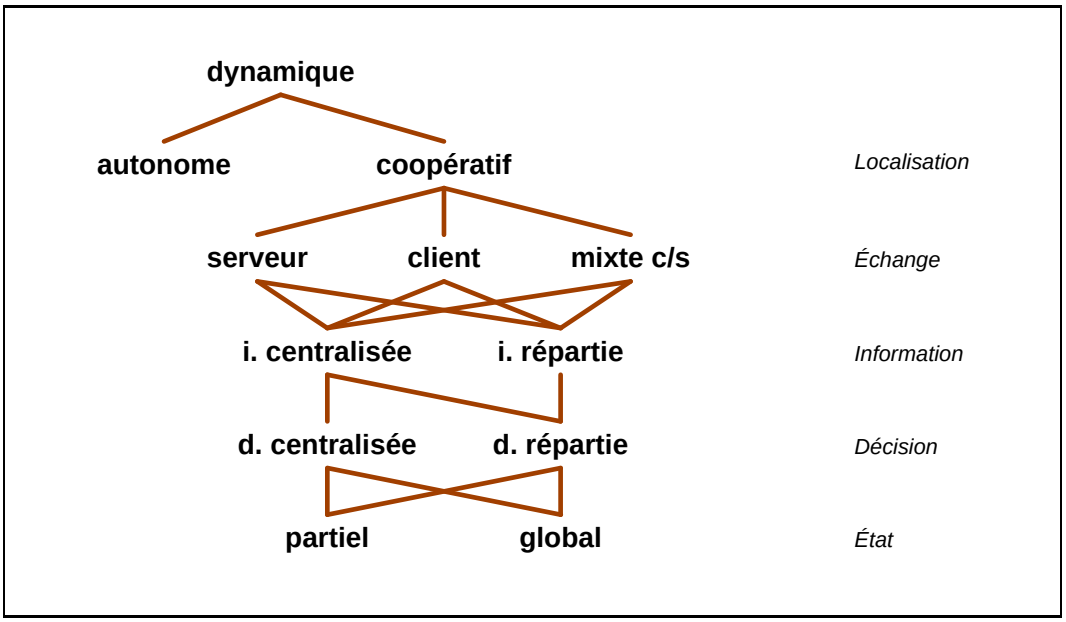


FIG. 4 – *Caractéristiques des politiques d'information et de localisation*

Un algorithme *autonome* détermine le site destinataire sans aucune connaissance de l'état de ce dernier. La localisation *coopérative* est dite *serveur* lorsque c'est la machine sous-chargée qui prend l'initiative du transfert de charge vers son site. Dans le cas *client* c'est la station en sur-charge qui décide de trouver une machine destinataire pour son trop plein de charge. Lorsque les deux

méthodes sont utilisées la localisation est dite *mixte client/serveur*. L'information, comme la décision, peut être *centralisée* ou *répartie*. Finalement, la décision est prise soit en connaissant l'état *global* ou *partiel* du système, c.-à-d. en fonction d'informations portant sur l'ensemble ou sur un sous-ensemble des machines du réseau.

Une extension de cette classification qui émerge tente de prendre en compte les aspects dynamiques des algorithmes de placement. La première caractéristique détermine si l'algorithme supporte la migration (*préemptif*), s'il effectue uniquement un placement (*non-préemptif*) ou mixte s'il effectue un placement initial avec migration ultérieure (*placement & migration*). La seconde caractéristique définit l'adaptabilité de l'algorithme, c.-à-d. si le nombre de tâches de l'application est fixé avant l'exécution (*non-adaptatif*), au démarrage (*semi-adaptatif*) ou variant en cours d'exécution en fonction de l'évolution de l'état global du système (*adaptatif*).

2.3.4 Algorithmes « classiques »

Dans ce paragraphe nous donnons un aperçu succinct des algorithmes les plus connus [Bernard 91], ainsi que les résultats liés à leurs performances.

- ALÉATOIRE : *autonome*
  l'algorithme choisit une machine au hasard et y transfère la tâche à placer ;

- CENTRAL : *coopératif, client, info. centralisée, déc. centralisée, global*
  une machine centralisée reçoit les charges des autres machines et prend la décision sur requête de placement émanant d'une autre machine, en sélectionnant la machine la moins chargée ;

- GLOBAL : *coopératif, client, info. centralisée, déc. répartie, global*
  une machine centralisée reçoit les charges des autres machines et rediffuse le vecteur des charges aux autres machines. Chacune sélectionne alors localement la machine la moins chargée pour y transférer la tâche à placer ;

- OFFRE : *coopératif, client, info. répartie, déc. répartie, global ou partiel*
  chaque machine diffuse périodiquement sa charge aux autres machines. Chacune sélectionne localement la machine la moins chargée pour y transférer la tâche à placer. La sélection peut s'opérer sur l'ensemble des machines

ou sur un sous-ensemble et cela en fonction de la quantité d'informations sauvegardée sur chaque site ;

- DEMANDE : *coopératif, client, info. répartie, déc. répartie, global*
  lorsqu'une machine a une tâche à placer, elle demande leur charge à toutes les autres machines et prend ensuite sa décision en sélectionnant la machine la moins chargée ;

- A_SEUIL : *coopératif, client, info. répartie, déc. répartie, partiel*
  une variante de DEMANDE, où l'algorithme demande leur charge successivement à un certain nombre de machines choisies aléatoirement et sélectionne la première machine ayant une charge inférieure à un certain seuil ;

- MOINDRE : *coopératif, client, info. répartie, déc. répartie, partiel*
  une variante de DEMANDE, où l'algorithme demande leur charge à un sous-ensemble de machines choisies aléatoirement et prend ensuite sa décision en sélectionnant la machine la moins chargée parmi ce sous-ensemble.

La comparaison des algorithmes classiques donne des résultats assez surprenants. Dans [Eager 86] les auteurs montrent que logiquement l'algorithme A_SEUIL est plus efficace que ALÉATOIRE, mais qu'en revanche la complexité supérieure de MOINDRE n'améliore pas sensiblement les performances du placement A_SEUIL.

Par ailleurs, différents auteurs ont montré [Zhou 88, Theimer 89] que l'algorithme CENTRAL affichait de très bonnes performances en terme de temps de réponse comparé par exemple aux algorithmes OFFRE ou GLOBAL. Cependant, CENTRAL pose le problème classique de la centralisation, c.-à-d. qu'il existe un goulot d'étranglement pour l'accès aux informations et un point de défaillance limitant la fiabilité du système de placement.

Notons cependant, que ces comparaisons de performances sont effectuées dans un cadre de placement de tâches simples, c.-à-d. sans communication entre-elles.

2.3.5 Outils de placement

Pour conclure la description des principes du placement dynamique, il est intéressant de présenter quelques outils de placement pouvant être intéressants dans le cadre de notre environnement.

Le mécanisme généralement utilisé pour le placement dynamique prenant en compte les communications entre les tâches d'une application parallèle est celui de la répulsion et attraction de tâches. C'est à dire que les tâches qui communiquent ensemble sont attirées sur une même machine, alors que celles qui utilisent les mêmes ressources (p.ex. la CPU) se répulsent et ont, par conséquent, tendance à s'exécuter sur des machines différentes.

Cette approche est utilisée dans Arcadia [Bernon 96], une plate-forme de placement distribuée. La coopération de divers agents situés sur chaque machine permet d'équilibrer la charge : les tâches qui communiquent ont tendance à se rapprocher lors de migrations, mais sont expulsées lorsque la charge locale à une machine augmente. Afin de favoriser le parallélisme, les auteurs proposent d'augmenter artificiellement la charge des machines par un facteur proportionnel aux communications. Ainsi, les tâches qui veulent communiquer ne sont pas pénalisées par le temps partagé, car elles auront tendance à s'exécuter sur des machines sous-chargées différentes.

La plate-forme d'exécution GatoStar [Folliot 95] effectue le placement complet multi-critères par raffinements successifs avec différents algorithmes de sélection de tâches en fonction de divers critères, tels que la précédence entre tâches, la charge, la localisation des fichiers ou les communications. Le principe de la sélection repose sur la notion d'attraction/répulsion. De plus, un mécanisme de migration de processus est inclus.

Dans le domaine de la programmation parallèle par objets, la problématique du placement (respectivement équilibrage de charge) se pose également, mais pour les entités plus fines que sont les objets. Une approche intéressante est celle de la « dérive de connaissances » telle que proposée dans [Chatonnay 96]. Le principe est très proche de celui exposé ci-dessus : les groupes d'objets sont rapprochés en fonction de la force des liens qui les lie. La force entre deux groupes augmente lorsque les activations (appels de méthodes) entre des objets des deux groupes augmentent, mais diminue lorsque la charge machine augmente et le temps passe (oubli).

Une nouvelle approche de la programmation permet de combiner l'exécution sur réseau de stations de travail et machines parallèles : la programmation hybride [Edjlali 96]. Les applications sont alors programmées en utilisant deux niveaux de granularités : gros grain (par échange de message) pour stations de travail et grain fin (mémoire partagée) pour les SMP. Il existe d'ailleurs un environnement supportant ce style de programmation : Stardust [Cabillic 96].

Citons également deux approches de placement d'applications parallèles plus en lien avec la gestion de ressources. La première [Neuman 94] utilise une hiérarchie de gestionnaires afin de fournir un service d'allocation de ressources (principalement des processeurs) dans les systèmes répartis de grande taille. Le second travail [Maier 97] effectue de la gestion de ressources et de l'équilibrage de charge en utilisant la migration pour des applications PVM.

Notons pour finir, que l'équilibrage de charge n'est pas forcément lié à la migration des tâches entre les machines, mais peut utiliser d'autres mécanismes dans des cas particuliers. Une approche utilisée par les applications basées sur la décomposition de domaines pour répartir les données de calcul repose sur le transfert périodique des données entre tâches [Hanxleden 91], en général avec synchronisation globale de l'application. Dans ce même esprit, l'outil MARS [Hafidi 96] ajoute ou détruit des tâches suivant l'état de charge des machines. Cette approche n'est possible que pour des applications SPMD spécifiques basées sur un modèle maître/esclave. Dans le cas d'applications programmées pour des processus légers (*thread*), l'équilibrage de charge peut être mis en œuvre par des mécanismes d'exécution de procédures à distance et de migration de *threads* [Namyst 95].

2.4 Conclusion

Ce chapitre montre que l'approche du placement statique permet de trouver une solution optimale au problème de placement de tâches d'une application parallèle. Cependant, en pratique seules les méthodes sous-optimales permettent d'obtenir une solution en un temps de calcul raisonnable. De plus, il est nécessaire d'évaluer (p.ex. estimation, simulation ou instrumentation) des coûts d'exécution et de communication [Sarkar 89] afin de permettre la modélisation sous forme de fonction de coût.

Le défaut majeur de l'approche statique reste néanmoins de ne pas prendre en compte l'état du système ainsi que son évolution lors du calcul du placement et durant l'exécution de l'application.

Le calcul du placement par des méthodes dynamiques permet de prendre en compte l'état du système lors du placement initial et de modifier en cours d'exécution la solution obtenue afin de tenir compte de l'évolution de l'état du système. De plus, il est possible de maintenir la charge équilibrée sur l'ensemble des machines. Cependant, les algorithmes classiques sont principalement destinés au placement de tâches sans communications entre-elles dans les systèmes répartis.

3 Environnement

Ce chapitre présente l'environnement de développement et d'exécution que nous proposons. Cette description se concentre principalement sur les aspects liés au placement et à l'outil PLATO.

3.1 Architecture

L'environnement que nous proposons doit aider l'utilisateur à développer des applications parallèles basées sur le modèle de programmation par échange de messages et de les exécuter en utilisant au mieux les ressources machine et réseau disponibles. Rappelons que nous nous plaçons dans un contexte de réseau de stations de travail où les liens et les machines peuvent être hétérogènes. Il est à noter que dans le cas général les machines ne sont pas dédiées au calcul, parallèle ou non.

Pour simplifier nous nommerons *machine parallèle virtuelle* (MV) l'ensemble des machines et liens de communication exécutant une application parallèle liée à notre environnement. Cette MV est, en principe, spécifique à une application et est définie par l'utilisateur en sélectionnant les machines les plus adaptées à son problème parmi celles qui lui sont accessibles. De plus, nous utiliserons le terme *système d'exécution* pour désigner l'ensemble des machines et réseaux accessibles aux applications parallèles gérées par notre environnement.

La figure 5 présente l'architecture de l'environnement proposé, constitué de quatre composantes principales :

1. MeDLey, le générateur pour le langage de spécification de messages ;
2. PlaTo, le module de placement de tâches ;
3. Analyse, traitement des traces d'exécution ;
4. administration de réseaux.

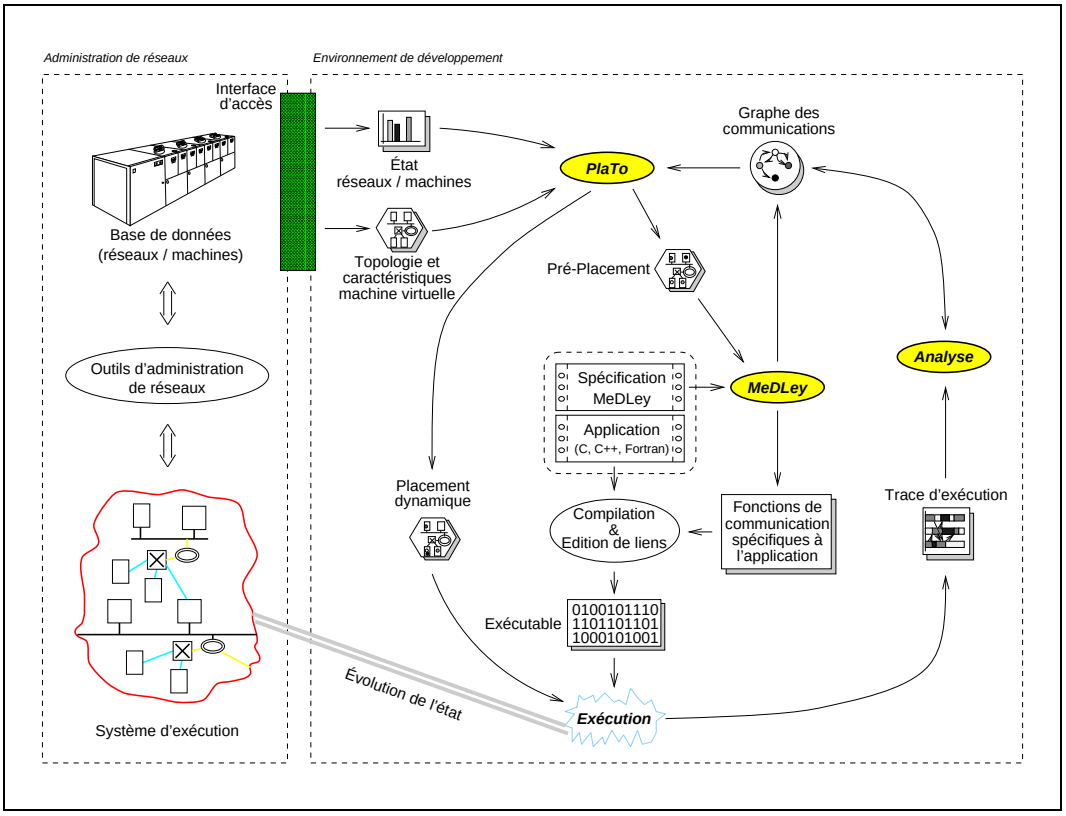


FIG. 5 – *Architecture de l'environnement*

L'utilisateur développe son application parallèle dans un langage cible (C, C++ ou Fortran) et spécifie les messages dans le langage MeDLey. À partir de cette dernière, le générateur MeDLey fournit un ensemble de fonctions

d'échange de données spécifiques à l'application et appelées par ses tâches. De plus, un graphe des communications décrivant la structure des messages qui sont échangés entre les différentes tâches du programme est construit.

Pour exécuter l'application parallèle, l'outil PLATO calcule un placement définissant quelles tâches s'exécutent sur quelles machines. Afin d'obtenir de bonnes performances *a priori*, il détermine ce placement en tenant compte du graphe des communications, de la topologie du réseau et de l'état courant de la MV associée à cette application. Les informations relatives à la MV sont obtenues par l'intermédiaire des outils de l'administration de réseaux. Bien-sûr l'exécution des tâches modifie l'état du système d'exécution (MV).

Afin d'augmenter les informations contenues dans le graphe par une composante quantitative, une génération des traces d'exécution est possible. Ainsi, le module d'analyse est en mesure d'enrichir le graphe des communications, permettant à PLATO de déterminer un meilleur placement pour l'application en question lors d'une exécution ultérieure.

3.2 MeDL*ey*

Ce paragraphe a pour but de présenter brièvement le langage de spécification MeDL*ey*, permettant de comprendre le genre d'informations contenues dans le graphe des communications ajoutées par le générateur. Pour plus d'informations sur la syntaxe et le langage se reporter à [Dillon 96b] ou au manuel de référence [Dillon 97].

3.2.1 Modèle M-SPMD

Les applications visées par notre environnement sont basées sur le modèle d'exécution M-SPMD (*Multiple - Single Program flow Multiple Data flow*), qui est une extension du modèle MPMD (*Multiple Program flow Multiple Data flow*).

Le modèle proposé regroupe les tâches de même fonctionnalité au sein de *familles*. Comme les tâches ont le même code exécutable mais agissent sur des données différentes, elles sont donc du type SPMD.

Le modèle M-SPMD permet de modéliser les applications allant du SPMD (une seule famille) au MPMD (plusieurs familles réduites à un membre unique).

L'application prototype de ce modèle contient donc plusieurs familles, dont certaines sont composées de tâches SPMD. La figure 6 présente un exemple de quatre familles décrivant une application M-SPMD.

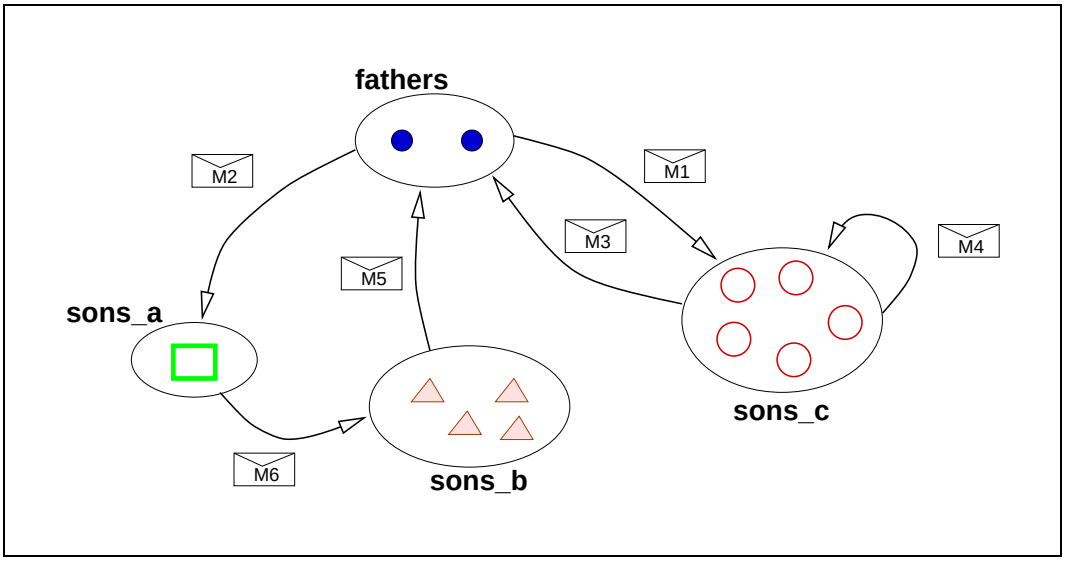


FIG. 6 – *Exemple d'application M-SPMD*

Le nombre de tâches au sein d'une famille n'est pas obligatoirement une constante du programme, mais peut être défini au démarrage en fonction de divers critères, tel que le nombre de machines disponibles. Ceci entraîne également, que le nommage d'une tâche est composé de deux parties : le nom de la famille de la tâche et son rang au sein de celle-ci.

3.2.2 Langage de spécification

Le principe de base du langage `MeDLey` est de spécifier les communication entre les tâches d'une application parallèle sans s'occuper du contrôle existant dans les tâches.

Une tâche MeDLey est décrite dans un *module*, qui se décompose en trois parties principales :

1. *uses*
   contient les déclarations des variables locales utilisées pour le calcul et les communications ;

2. *sends*
   déclare tous les échanges de données à destination d'autres tâches. Ces échanges sont définis sous forme de messages à émettre vers des tâches destinataires. Les données contenues dans la déclaration d'un message sont basées sur les variables définies dans la partie *uses*. Notons que la taille d'un message peut être dynamique ;

3. *receives*
   déclare tous les échanges de données provenant d'autres tâches. De nouveau, le contenu de la définition d'un message se rapporte aux déclarations de la partie *uses*.

De façon informelle, la partie *sends* spécifie comment construire un message à partir des variables déclarées dans la partie *uses*, alors que la partie *receives* spécifie comment projeter le contenu des messages entrants sur les variables locales.

Ajoutons que MeDLey permet d'utiliser différentes sémantiques d'échange pour chaque message : opérations synchrones ou asynchrones, en point-à-point, diffusion ou multidiffusion.

3.2.3 Graphe des communications

À partir de la spécification, le générateur MeDLey construit un graphe des communications contenant deux types d'informations : le nombre de familles de l'application et la structure des messages avec les familles source et destinataire associées.

Notons que la source ou la destination d'un message est indiquée par la famille et non par une tâche particulière. En effet, le nombre de tâches au sein d'une famille n'est défini qu'au démarrage de l'application. Il s'ensuit que la spécification des communications entre les tâches peut être imprécise dans

le graphe, c.-à-d. que certaines tâches échangent théoriquement des messages, mais que la structure SPMD des familles peut faire en sorte que ces échanges n'aient pas lieu. Ce problème se pose aussi bien pour les communication inter-familles qu'intra-familles.

Pour illustrer ce problème, prenons des exemples de spécification d'échange de messages de la figure 6. La tâche i de la famille `fathers` communique avec la tâche j de la famille `sons_c` avec un message du type `M1`. Ceci signifie qu'*a priori* chacune des deux tâches de la famille `fathers` communique avec chaque membre de la seconde famille. Or, l'algorithme peut très bien faire communiquer les tâches de rang pair (respectivement impair) d'une famille avec les tâches paires (respectivement impaires) de l'autre famille. Dans ce cas, le nombre de communications spécifiées en termes de couples émetteur/récepteur est supérieur au nombre d'échanges effectifs.

Le cas intra-familles est encore plus délicat, car c'est la structure des échanges qui n'est absolument pas prise en compte. Par exemple, nous souhaitons que l'échange du message `M4` se fasse sous forme d'anneau à l'intérieur de la famille `sons_c`. La spécification d'un tel échange se fait pourtant en indiquant qu'une tâche de rang i communique avec une tâche de rang j, qui correspond également à un échange total (chaque tâche communique avec toutes les autres). De nouveau, la spécification est imprécise et ne permet pas de connaître quelles tâches communiquent réellement entre-elles.

Le graphe des communications généré ne contient donc que des informations qualitatives, parfois imprécises, car seule la structure du message et le nombre de familles sont définis. L'aspect quantitatif des communications (le nombre et la taille réelle des messages échangés) ne peut être généré directement à partir de la spécification, d'où l'importance d'obtenir ces informations par le biais de la trace d'exécution de l'application.

3.2.4 Relation de précédence

Le graphe d'une application parallèle décrit souvent la relation de précédence entre les tâches [Cosnard 93]. Cette précédence indique la relation d'ordre partiel d'exécution entre tâches, c.-à-d. qu'une tâche ne peut commencer son exécution que si un certain nombre d'autres tâches se sont terminées.

Cependant, l'absence de spécification des structures de contrôle dans MeDLey ne permet pas de disposer de cette information. Pour contourner cette absence dans l'environnement, toutes les tâches sont lancées simultanément au démarrage de l'application. Ainsi, grâce au blocage des réceptions de messages la relation est préservée de manière implicite. Les principaux désavantages d'une telle approche sont les suivants :

- les tâches inactives, lors de l'attente d'un premier message, consomment néanmoins des ressources ;
- le placement ne peut tenir compte de l'ordre séquentiel d'exécution pour placer deux tâches qui se suivent sur la même machine.

3.3 Administration de réseaux

Pour effectuer le placement dynamique des tâches, il est nécessaire d'obtenir un certain nombre d'informations sur l'état du système d'exécution. Pour cela, il est possible d'intégrer cette obtention directement dans l'outil de placement (souvent le cas pour les outils actuels). Dans PLATO, par contre, nous faisons appel à l'administration de réseaux afin de gérer partiellement la récupération des ces informations.

L'administration de réseaux vise à la mise en place des logiciels de contrôle et de supervision des liens de communication entre des machines. Dans le contexte de notre environnement, l'utilisation de l'administration de réseaux va permettre d'avoir accès à un certain nombre d'informations et de services, dont les plus importantes sont :

- l'état courant (p.ex. charge) des machines et des liens de communication ;
- la topologie du réseau, incluant la majeure partie des équipements importants ;
- la description des ressources disponibles sur les différentes machines de calcul ;
- la réservation de ressources et/ou qualités de service.

Le principe de l'administration s'appuie sur des agents présents sur les différents équipements du réseau ayant des ressources à gérer : calculateurs, routeurs, etc. Une plateforme centralisée détermine dynamiquement la topologie du réseau grâce à ces agents. Ces informations sont ensuite stockées dans une base de données locale à la plateforme de gestion. Un des standards de gestion est le protocole SNMP (*Simple Network Management Protocol*) [Stallings 93].

L'exploitation des informations d'administration se fait sous forme de requêtes au travers d'une interface d'accès. Lorsque les données requises ne sont pas présentes dans la base, le gestionnaire se charge de les obtenir auprès des agents respectifs. La configuration des agents autorise l'avertissement du gestionnaire lorsque certains événements surviennent qui sont ensuite rajoutés à la base.

Les agents existants ne gèrent pas toutes les ressources utiles au placement de tâches, comme par exemple les indicateurs de charge d'une machine ou du réseau. Il donc nécessaire dans certains cas, d'étendre les fonctions des agents pour prendre en compte ces ressources spécifiques.

De plus, la gestion de ressources par des agents offre la possibilité de faire de la réservation de certaines ressources. Dans notre cas il est concevable de réserver de la bande passante sur les liens de communication permettant à PLATO d'assurer qu'une certaine qualité de service pourra être respectée. Bien entendu, cette approche ne se limite pas uniquement aux réseaux et à leurs débits.

3.4 Analyse

Comme déjà indiqué, une spécification `MeDLey` ne permet que de définir la structure d'un message pour une communication donnée. Or, PLATO doit connaître le volume de données échangées entre les tâches pour déterminer le meilleur placement tenant compte des liens de communication. C'est justement cette quantité que doit récupérer ce module en analysant les traces d'exécution du programme.

Le mécanisme de base pour obtenir l'information quantitative consiste à comptabiliser les messages échangés par le biais des fonctions générées par `MeDLey`. En principe, cette instrumentation ne perturbe quasiment pas le dé-

roulement des tâches. Cette opération peut s'effectuer de deux manières différentes.

La première méthode consiste à *pré-exécuter* l'application avec un jeu de données réduit, permettant d'obtenir le poids relatif des communications au sein de l'application parallèle en question. Néanmoins, c'est à l'utilisateur de fournir un jeu d'essai représentatif des échanges pour des données réelles.

Dans la seconde méthode, il s'agit d'instrumenter les calculs réels, c.-à-d. en comptabilisant les messages réellement échangés lors des exécutions sur des données réelles.

La difficulté majeure du module d'analyse réside dans la modification du graphe en fonction des résultats des mesures et de la pertinence du jeu de données de pré-exécution par rapport aux jeux de données réels. De plus, le nombre de communications dépend, *a priori*, des données en entrée de l'application et peut donc changer avec chaque nouvelle exécution, ce qui n'autorise pas une décoration immédiate du graphe des communications à partir des mesures.

3.5 PlaTo

Ce paragraphe a pour objectif de définir les caractéristiques du module de placement PlaTo à mettre en œuvre dans l'environnement proposé et cela en lien avec les principes exposés au chapitre 2. Les caractéristiques essentielles, découlent directement des contraintes liées au contexte d'utilisation de l'environnement et du générateur `M`*e*`DL`*ey*, tels que présentés aux paragraphes précédents. En outre, nous exposons également les problèmes induits par les divers choix effectués.

La première contrainte – et certainement la plus importante – est liée à l'utilisation de l'environnement dans un contexte de réseaux de stations de travail non-dédiées. En effet, l'état de charge d'un tel système est rarement stable c.-à-d. qu'il est constamment perturbé par le lancement ou la terminaison de processus ainsi que par l'état de disponibilité des ressources. La conclusion évidente qui s'ensuit est que PlaTo doit être dynamique, donc qu'il place les tâches en tenant compte de l'état courant du système d'exécution.

Par conséquent, la présentation du module est découpée suivant les trois composantes principales du placement dynamique (cf. chapitre 2.3).

Notons également, que les tâches développées dans ce cadre sont plutôt d'une durée d'exécution moyenne ou longue. Par conséquent, le coût de la recherche d'un bon placement peut se permettre d'être plus ou moins long, car le temps de placement reste, *a priori*, négligeable par rapport au temps total d'exécution de l'application.

3.5.1 Politique de transfert

La politique de transfert détermine quand a lieu l'opération de placement et quelle tâche placer. Dans le contexte d'utilisation de l'environnement, les tâches à placer sont celles issues d'une application parallèle développée avec l'environnement, c.-à-d. celles qui possèdent un graphe des communications. De plus, le calcul du placement s'effectue au démarrage de l'ensemble des tâches de l'application parallèle, car toutes les tâches `MeDLey` sont lancées dès le début de l'exécution.

Néanmoins, cette approche a le désavantage de ne pas prendre en compte ni les perturbations survenant lors du lancement/terminaison de processus utilisateur non liés à l'environnement, ni les pannes des stations de travail.

3.5.2 Politique d'information

La politique d'information est également facile à définir. Les informations nécessaires sont d'une part la topologie du réseau comprenant les machines de calcul, les liens de communication et les autres équipements influençant les délais d'acheminement d'un message. Il est donc indispensable de modéliser dans cette topologie la capacité d'un lien de communication, ainsi que les temps de transfert en fonction du volume de données émises [Dillon 96a]. Notons, que la politique d'information est partiellement déportée dans le gestionnaire de l'administration de réseaux.

D'autre part, le placement dynamique repose sur la connaissance ou une évaluation de l'état de la MV (charge) au moment du placement. Un des indicateurs indispensables est la charge de la machine modélisée par la longueur de la file d'attente telle que proposée dans [Ferrari 88]. Pour la charge réseau il faut créer un indicateur qui donne le pourcentage d'utilisation du médium par rapport à la valeur maximale en régime saturé. Pour éliminer les effets de

sur-charge momentanés, un lissage temporel, semblable à celui de la charge machine, doit être appliqué.

La troisième source d'information à utiliser par le module de placement est constituée par le graphe des communications de l'application parallèle.

La politique d'information spécifie également de quelle manière les données relatives au placement sont réparties. Compte tenu des résultats présentés dans [Zhou 88], la centralisation des données est intéressante dans le cas de l'algorithme CENTRAL. Ceci nous a donc conduis à sélectionner CENTRAL pour le noyau de PLATO. De plus, l'approche centralisée de la prise de décision et des données simplifie la conception de l'outil de placement, bien qu'il soit nécessaire d'introduire des mécanismes de reprise en cas de panne du cite central. Cette simplification n'est d'ailleurs pas mise en défaut par les plateformes d'administration dans la mesure où ces dernières sont souvent centralisées, disposant ainsi d'une plus grande visibilité de l'ensemble des applications et du système d'exécution.

3.5.3 Politique de localisation

La politique de localisation a pour but de choisir parmi les machines disponibles, celles qui permettent une bonne exécution de l'application parallèle.

Cette sélection de machines cible pour les différentes tâches de l'application reste néanmoins un problème difficile, dans la mesure où il faut tenir compte de plusieurs critères dont le principal est lié aux communications au sein de l'application. De plus, l'opération de sélection est mise en œuvre lors de deux étapes différentes : d'un côté la pré-exécution et de l'autre l'exécution réelle de l'application parallèle.

Lors de l'étape de *pré-exécution* aucune information quantitative sur les échanges n'est encore disponible ; la sélection d'une machine ne peut donc tenir compte des communications. D'un côté le choix des machines peut s'effectuer simplement en fonction de la charge des machines en considérant une machine par tâche. D'un autre côté, la pré-exécution peut traiter un jeu de données réduit, destiné à diminuer le temps d'exécution en comparaison avec les données réelles. Ainsi, une seule machine peut exécuter l'ensemble des tâches de l'application en un temps raisonnable.

Dans le cas de l'exécution réelle, la sélection des machines peut s'effectuer en fonction des réseaux d'interconnexion ainsi que des débits prévus entre les tâches avec réservation de bande passante entre des machines. De cette manière, la charge courante des machines devient alors uniquement un critère secondaire, si les communications représentent un aspect important de l'application instrumentée.

4 Perspectives

Le chapitre précédent présente l'environnement et les problèmes liés au placement. Afin d'améliorer les performances des applications, diverses évolutions en relation avec le placement peuvent être envisagées qui sont liées à une meilleure connaissance des points suivants :

- *de l'application*
  introduire le pré-placement et les schémas de communication dans `MeDLey`,
- *du système d'exécution*
  ajouter la collaboration avec des outils de gestion par flots et un mécanisme de migration.

4.1 Pré-placement

Une première amélioration introduit la notion de pré-placement dans l'environnement, tel que présenté sur la figure 5. Le but de ce dernier est de prendre en compte des contraintes imposées par l'utilisateur dès la phase de développement de l'application, comme par exemple l'exécution d'une certaine tâche sur une machine donnée ou possédant une ressource donnée.

En imposant des contraintes aux tâches, l'utilisateur restreint de cette manière les choix de placement. Grâce à ce mécanisme, le pré-placement peut sélectionner une machine (ou architecture) particulière et ainsi commander le générateur `MeDLey` pour que ce dernier génère des fonctions de communications optimisées pour la machine et ses interfaces réseau.

L'information sur les interfaces disponibles est fournie par les outils d'administration, qui possèdent par ailleurs une meilleure connaissance de la per-

formance des réseaux connectant plusieurs machines de la MV de l'application. Cette information permet donc de choisir, dès le départ, les meilleures connexions.

4.2 Schémas de communication

MeDLey permet de spécifier les communications s'effectuant entre les diverses familles d'une applications. Au paragraphe 3.2, nous avons indiqué qu'une spécification pouvait être incomplète, car ne prenant pas en compte la structure des échanges. Pour résoudre ce problème pour les communications intra-familles (échanges SPMD) nous proposons d'introduire les notions de *topologie* et principalement de *schéma de communication*.

Dans le cadre du modèle SPMD, le terme topologie désigne généralement la manière dont sont interconnectés les processeurs entre-eux, c.-à-d. l'architecture physique : p.ex. grille, anneau, hypercube, etc. Comme MeDLey a pour but de faciliter la programmation parallèle en faisant abstraction des problèmes d'architecture physique, il n'est pas intéressant de spécifier la topologie physique des tâches.

Par contre, il est plus utile d'introduire la *topologie logique* des tâches d'une famille, qui définit quelles tâches peuvent communiquer avec quelles autres tâches, au sein d'une même famille. Ceci permet de faciliter la programmation, car les fonctions générées par MeDLey pour la famille en question n'offriront que les échanges conformes à la topologie définie et correspondant ainsi mieux à l'algorithme de l'utilisateur.

Par exemple, les tâches d'une famille spécifiée avec une topologie en grille ne pourront communiquer qu'avec les tâches voisines. De plus, cette approche évite des erreurs de communication, (p.ex. envoi d'un message à un destinataire non prévu), car le choix des tâches source et destination est plus restreint que dans le cas sans topologie (ou topologie quelconque).

En raffinant l'idée de topologie logique, une nouvelle notion apparaît : le *schéma de communication*. Ce dernier définit pour un message échangé entre tâches d'une même famille, la structure que peut emprunter ce message. Le générateur MeDLey génère dans ce cas-ci des fonctions spécialisées pour le message n'autorisant que les communications suivant le schéma précisé.

La figure 7 présente un exemple d'une famille spécifiée avec une topologie logique en grille torique (complètement connectée) et deux messages. Le premier message s'échange avec un schéma de communication correspondant à la topologie logique, alors que le second emprunte le schéma de communication en anneau.

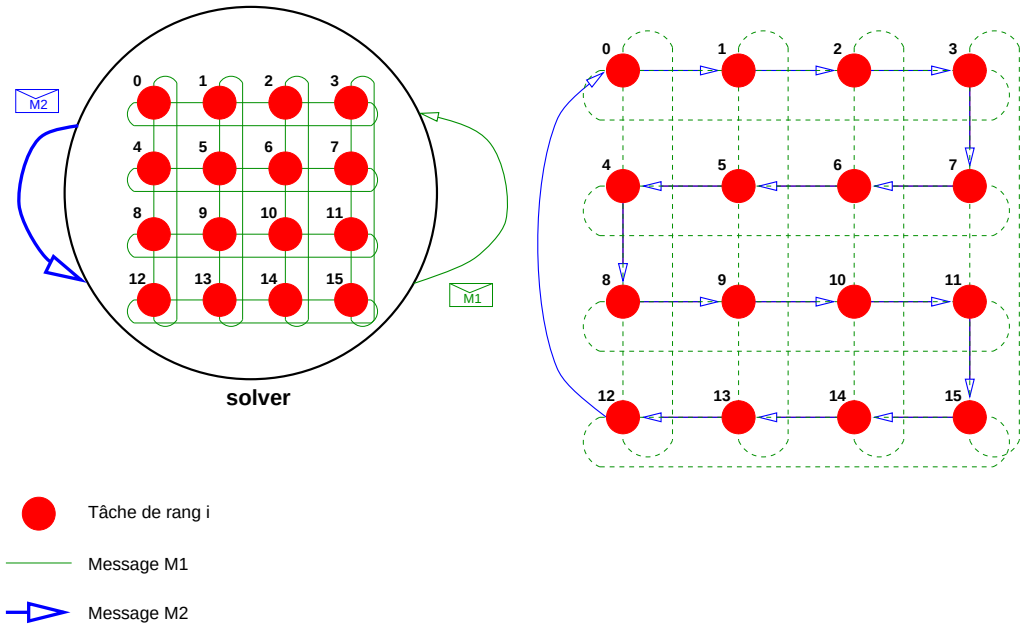
L'ajout de la notion de schéma augmente un peu le niveau d'abstraction de la programmation des tâches SPMD. En effet, l'utilisateur peut spécifier une topologie logique au niveau de la famille (les schémas de communication des messages correspondent à cette topologie) et concevoir son algorithme avec une vision classique en SPMD. Par contre, il peut également spécifier un schéma pour un ou plusieurs messages, lui permettant ainsi d'utiliser des fonctions de communication spécifiques à chaque message adaptées à son algorithme. Dans l'exemple de la figure 7, le message `M2` peut également être échangé en empruntant la topologie logique en grille torique. Dans ce cas, l'information indiquant que le message suit un schéma en anneau disparaît.

Pour le placement, ces informations de topologie (logique ou schéma), présentes dans le graphe des communications, ne sont pas négligeables, car elles offrent une meilleure connaissance des communications pouvant avoir lieu au sein d'une famille. Ainsi, PLATO est en mesure de mieux placer les tâches en fonction de leur volume de communication.

4.3 Outils d'administration

La seconde approche du domaine de l'administration consiste à améliorer la collaboration entre le module de placement PLATO et les outils de gestion de ressources. Ce type d'outils, comme par exemple LSF [LSF 96] ou NQE [NQE 97], permettent de lancer des tâches individuelles sur un ensemble de machines en fonction de critères tels que la charge, la disponibilité de certaines ressources ou simplement de contraintes utilisateur. Le principe de fonctionnement est généralement basé sur des files de soumission plus ou moins prioritaires, autorisant l'outil à lancer, arrêter ou suspendre et même dans certains cas à migrer des programmes en fonction de critères tels que la charge machine ou la durée d'exécution.

Même si les versions récentes proposent des mécanismes de soumissions pour des applications parallèles, le placement des tâches ne prend pas en



(a) Spécification : topologie en grille torique

(b) Schéma de communication : en anneau

FIG. 7 – *Exemple de topologie et de schéma de communication*

compte les dépendances (de communication ou de précédence) pouvant exister entre ces tâches.

L'interaction entre le gestionnaire de programmes et PLATO permet à ce dernier d'avoir accès à une *connaissance partielle* du futur. En effet, le gestionnaire est en mesure de connaître les tâches en attente d'exécution dans ses files, fournissant ainsi au placeur une information sur les tâches qui seront lancées dans un futur plus ou moins proche. Dans le cas où l'exécution de programmes importants sur les machines du réseau passe par un tel outil, PLATO serait en mesure de déterminer un meilleur placement pour les applications parallèles sous son contrôle.

4.4 Migration

Le mécanisme de migration de tâches n'est pas inclus dans l'outil PLATO pour des raisons évidentes de complexité de mise en œuvre. Néanmoins, à terme il est indispensable d'inclure un tel mécanisme dans l'outil de placement. En effet, les réseaux de stations de travail banalisés (non-dédiés) sont soumis à des variations importantes de leur charge, principalement dues aux lancements et terminaisons des processus interactifs.

Comme une telle évolution de l'état de la MV peut rendre le placement initialement calculé totalement inefficace, un mécanisme de migration des tâches des applications parallèles est donc intéressant, afin d'obtenir la meilleure performance possible. Cependant, dans le cas de mécanismes de migration quasi-transparents pour l'utilisateur et sans modification du système d'exploitation, la durée d'une relance d'une tâche est très longue (de l'ordre de la dizaine de secondes [Litzkow 92]). Par conséquent, la migration est inefficace comme technique de partage de charge, sauf pour les situations de très fort déséquilibre de charge.

Par contre, la migration est un mécanisme très utile dans la fiabilisation de l'exécution des applications parallèles, afin de redémarrer une tâche après un problème (p.ex. arrêt) d'une station de travail.

La troisième raison pour laquelle la migration est importante dans un environnement comme celui que nous proposons, est de permettre de prendre en compte le caractère personnel d'une station de travail. En effet, l'utilisateur principal d'une station – exécutant en général des sessions interactives – ne doit pas être pénalisé par l'exécution d'applications parallèles en provenance d'autres utilisateurs. Le meilleur moyen de prévenir ce problème, consiste à migrer les tâche concernées sur d'autres stations.

Pour terminer, notons que la migration de tâches est un mécanisme complexe qui consiste à déplacer un processus UNIX d'une machine vers une autre. Ceci pose de nombreux problèmes, partiellement résolus, tels que l'hétérogénéité, les fichiers ouverts par l'application et les communications en cours. Il s'ensuit que cette technique ne peut être utilisée dans tous les cas : une tâche d'une application parallèle doit répondre à certaines contraintes afin qu'elle puisse être migrée.

5 Conclusion

Le placement de tâches est un sujet qui a fait l'objet de nombreuses études, mais dont la problématique a été relancée par la constitution de machines parallèles virtuelles sur réseau de stations. Dans ce document, nous avons présenté les principes du placement de tâches statique et dynamique. Il en ressort d'un côté, que le placement statique d'une application parallèle permet en théorie d'obtenir un temps d'exécution optimal. Par contre en pratique les coûts et temps de calcul d'un tel placement étant prohibitifs, des méthodes heuristiques obtiennent de résultats avec des coûts moindres. De plus, l'approche statique est plutôt adaptée aux machines parallèles dédiées au parallélisme. D'un autre côté, le placement dynamique permet de placer des tâches en tenant compte de l'état courant des machines. Couplé avec de la migration, le placement dynamique permet de répondre efficacement aux déséquilibres de charge pouvant survenir sur des systèmes répartis.

Nous avons également présenté notre environnement en cours de développement et d'exécution du point de vue du placement des tâches. Cet environnement est destiné à faciliter la programmation des applications parallèles M-SPMD et à obtenir de bonnes performances d'exécution. Ceci passe par la définition d'un langage de spécification qui favorise l'optimisation les communications et de placer au mieux les tâches sur les stations de travail disponibles.

Notre approche du placement consiste à de tenir compte à la fois des critères classiques (p.ex. charge machine), des communications attendues et de l'état des réseaux pour calculer un placement efficace. Pour l'obtention des informations sur l'état actuel des machines et réseaux, nous utilisons des principes et techniques issues de l'administration de réseaux.

Dans l'immédiat, nous comptons terminer le développement de notre environnement avec les choix effectués au départ. À plus ou moins long terme, nous comptons l'étendre avec de nouvelles fonctionnalités – principalement liées à la collaboration avec d'autres outils – indispensables pour garantir de bonnes performances d'exécution des applications parallèles `M`*e*`DL`*ey* et des autres utilisateurs des stations de travail.

Références

[Belhamissi 91] Yasmina Belhamissi et Maurice Jégado. Scheduling in distributed systems : Survey and questions. Rapport de Recherche no. 1478, INRIA, juillet 1991.

[Bernard 91] Guy Bernard, Dominique Stève et Michel Simatic. Placement et migration de processus dans les systèmes répartis faiblement couplés. *Technique et Science Informatique*, 10(5) :375–392, 1991.

[Bernard 96] G. Bernard, J. Chassin de Kergommeaux, B. Folliot et C. Roucairol, éditeurs. *Placement Dynamique et Répartition de Charge : Application aux Systèmes Répartis et Parallèles.* Collection Didactique INRIA. INRIA, décembre 1996.

[Bernon 96] Carole Bernon, Claude Bétourné et Amal Sayah. Évaluation d'une plat-forme pour le placement dynamique de processus communicants. *Calculateurs Parallèles*, 8(1) :49–68, 1996.

[Bouvry 94] Pascal Bouvry. *Placement de tâches sur ordinateurs parallèles à mémoire distribuée.* Thèse de doctorat, Institut National Polytechnique de Grenoble, octobre 1994.

[Cabillic 96] Gilbert Cabillic et Isabelle Puaut. Répartition de charge dans stardust : un environnement pour l'exécution d'applications parallèles en milieu hétérogène. In Bernard et al. [Bernard 96], pages 167–174.

[Cabrera 86] Luis-Felipe Cabrera. The influence of workload on load balancing strategies. *Proceedings of the USENIX Summer Conference*, pages 446–458, Atlanta, USA, 1986.

[Casavant 88] Thomas L. Casavant et Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed systems. *IEEE Transactions on Software Engineering*, 14(2) :141–154, février 1988.

[Chatonnay 96] Pascal Chatonnay, Bénédicte Herrmann, Laurent Philippe, François Bourdon, Pascal Bar et Christian Jacquemot. Placement dynamique dans les systèmes répartis à objets. *Calculateurs Parallèles*, 8(1) :11–30, 1996.

[Clark 92] Henry Clark et Bruce McMillin. DAWGS – a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2) :175–186, février 1992.

[Cosnard 93] Michel Cosnard et Denis Trystram. *Algorithmes et Architectures Parallèles*. InterEditions, 1993.

[Dillon 96a] Eric Dillon, Carlos Gamboa Dos Santos et Jacques Guyard. Impact des réseaux ethernet et ATM sur les performances de PVM et MPI. *Calculateurs Parallèles*, 8(2), 1996.

[Dillon 96b] Eric Dillon, Jacques Guyard et Georges Wantz. Medley : an abstract approach to message passing. *PARA96 : Workshop on Applied Parallel Computing in Industrial Problems and Optimisation*, volume 1184, série *Lecture Notes in Computer Science*, pages 196–203, août 1996.

[Dillon 97] Eric Dillon. Medley : Efficient communications for distributed applications. Manuel d'utilisateur, INRIA, 1997. To appear.

[Eager 86] Derek L. Eager, Edward D. Lazowska et John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5) :662–675, mai 1986.

[Eager 88] Derek L. Eager, Edward D. Lazowska et John Zahorjan. The limited performance benefits of migrating active processes for load sharing. *Performance Evaluation Review*, 16(1) :63–72, mai 1988.

[Edjlali 96] Guy Edjlali, Serge G. Petiton et Nahid Emad. Interleaved parallel hybrid arnoldi method for a parallel machine and

a network of workstations. *Proceedings of ISAS'96 (Information, Systems, Analysis and Synthesis) Conference*, Orlando, FL, USA, juillet 1996.

[Ferrari 88] Domenico Ferrari et Songnian Zhou. An empirical investigation of load indices for load balancing applications. P.-J. Courtois et G. Latouche, éditeurs, *Proceedings PERFORMANCE'87*, pages 515–528. Elsevier Science Publishers, 1988.

[Folliot 93] Bertil Folliot. *Méthodes et Outils pour le Partage de Charge pour la Conception et la Mise en Oeuvre d'Applications dans les Systèmes Répartis Hétérogènes.* Thèse de doctorat de l'université de paris 6, Institut Blaise Pascal, avril 1993.

[Folliot 95] Bertil Folliot, Pierre Sens et Pierre-Guillaume Raverdy. Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti. *Calculateurs Parallèles*, 7(4) :345–366, 1995.

[Garey 79] M. R. Garey et D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[Geist 94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek et Vaidy Sunderam. *PVM: Parallel Virtual Machine.* Scientific and Engeneering Computation Series. The MIT Press, 1994.

[Gropp 94] William Gropp, Ewing Lusk et Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engeneering Computation Series. The MIT Press, 1994.

[Hafidi 96] Z. Hafidi, E.-G. Talbi et J.-M. Geib. Parallélisme adaptatif dans un environnement multi-utilisateurs hétérogène. In Bernard et al. [Bernard 96], pages 237–244.

[Hanxleden 91] Reinhard V. Hanxleden et L. Ridgway Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13(3) :312–324, novembre 1991.

[Kafil 97] Muhammad Kafil et Ishfaq Ahmad. Optimal task assignment in heterogeneous computing systems. *Proceedings of the Sixth Heterogeneous Computing Workshop*, pages 135–146, Geneva, Switzerland, avril 1997.

[Litzkow 92] Michael Litzkow et Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. *Proceedings of the USENIX Winter Conference*, pages 283–290, San Francisco, CA, USA, janvier 1992.

[LSF 96] Platform Computing Corporation, Canada. *LSF 2.2 Programmer's Guide*, 1 édition, février 1996.

[Maier 97] Ursula Maier et Georg Stellner. Distributed resource management for parallel applications in networks of workstations. *High-Performance Computing and Networking*, volume 1225, série *Lecture Notes in Computer Science*, pages 462–463, avril 1997.

[Monien 90] B. Monien et H. Sudborough. Embedding one interconnection network in another. *Computational Graph Theory*, Computing Supplement 7 :257–282, 1990.

[Muntean 91] Traian Muntean et El-Ghazali Talbi. Méthodes de placement statique des processus sur architectures parallèles. *Technique et Science Informatique*, 10(5) :355–373, 1991.

[Namyst 95] Raymond Namyst et Jean-François Mehaut. PM2 : Parallel multithreaded machine. a multithreaded environment on top of PVM. *2nd European PVM Users' Group Meeting*, pages 179–184, Lyon, France, septembre 1995.

[Neuman 94] B. Clifford Neuman et Santosh Rao. The Prospero Resource Manager : A scalable framework for processor allocation in

distributed systems. *Concurrency : Practice and Experience*, 6(4) :339–355, juin 1994.

[Norman 93] Michael G. Norman et Peter Thanish. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 253 :263–302, septembre 1993.

[NQE 97] Cray Research, Inc. *NQE 3.2 User's Guide*, janvier 1997. SG-2148.

[Pellegrini 95] François Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme.* Thèse de doctorat, Université de Bordeaux I, janvier 1995.

[Sarkar 89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman - MIT Press, 1989.

[Scherson 96] Isaac D. Scherson, Raghu Subramanian, Verônica L. M. Reis et Luis Miguel Campos. Scheduling computationally intensive data parallel programs. In Bernard et al. [Bernard 96], pages 107–130.

[Shen 85] Chien-Chung Shen et Wen-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3) :197–203, mars 1985.

[Stallings 93] William Stallings. *SNMP, SNMPv2, and CMIP : The Practical Guide to Network-Management Standards.* Addison-Wesley, 1993.

[Tannenbaum 95] Andrew S. Tannenbaum. *Distributed Operating Systems.* Prentice-Hall, 1995.

[Theimer 89] Marvin M. Theimer et Keith A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11) :1444–1458, novembre 1989.

[Zhou 88] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9) :1327–1341, septembre 1988.

[Zhou 92] Songnian Zhou, Xiaohu Zhen Jingwen Wang et Pierre Delisle. UTOPIA : A load sharing facility for large, heterogeneous distributed computer systems. Rapport no. CSRI-257, Computer Systems Research Institute, University of Toronto, avril 1992.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399