Mémoire de Master 2 IFA
Université Louis-Pasteur de Strasbourg

Sélection de pairs et allocation de tâches dans P2P-MPI

Ghazi Bouabene

Master 2 IFA 2005/2006
laboratoire : Image et Calcul Parallèle Scientifique

Stage encadré par Stéphane Genaud

Remerciements

Je remercie tout d'abord mes frères, Anis et Karim ainsi que Victoria pour leur soutien quotidien.

Un grand merci à Stéphane Genaud pour ses qualités bienveillantes d'encadrant et son soutien dans les candidatures de thèses.

Enfin merci à tous les membres de l'ICPS pour leur acceuil chaleureux, avec une "spéciale dédicace" pour la basket-team.

Table des matières

Table des figures

Chapitre 1

Introduction

L'évolution constante des architectures des ordinateurs ainsi que celle des technologies réseau ont fait émerger le concept de *Grille* de calcul. En effet il devient désormais possible de fédérer les ressources de plusieurs machines de bureau afin d'atteindre des capacités de calcul rivalisant avec celles des supercalculateurs ou encore celles des clusters. Ces grilles sont souvent construites par l'intermédiaire d'un **middleware**(intergiciel) qui a pour rôle d'offrir des services au niveau système aux utilisateurs tels que la découverte des ressources, le transfert de fichiers, le lancement de tâches à distance, etc.

Dans ce cadre on trouve des projets originaux tel P2P-MPI[10], un middleware développé pendant la thèse de M. Choopan Rattanapoka sous la direction de M. Stéphane Genaud. P2P-MPI permet de construire une grille de calcul dont la gestion est totalement distribuée selon un mode pair-à-pair. Les pairs participant à la grille ont tous un statut d'égal à égal et peuvent bénéficier des ressources disponibles ainsi que mettre les leurs à disposition.

Cependant, vu le grand nombre de défis à relever afin d'obtenir un intergiciel de grille fiable, robuste aux erreurs et offrant une API de programmation complète, le problème d'une allocation optimale des tâches sur les machines formant la grille n'a pas encore pu être traité. D'autant plus que ce problème est rendu plus difficile par l'hétérogénéité des ressources. En effet les machines formant une grille P2P-MPI sont géographiquement distribuées et sous l'autorité de différents organismes (utilisateurs privés, universités, entreprises ..). Cette grille est donc formée de machines à architectures différentes avec différents débits d'interconnexion.

Nous allons donc dans le cadre de ce mémoire nous intéresser au problème difficile de l'allocation des tâches d'une application parallèle aux ressources d'une grille hétérogène. Pour cela nous allons dans un premier temps étudier et classifier les différentes approches proposées dans la littérature pour traiter de ce problème. Nous proposerons ensuite une méthode d'allocation que nous pensons adaptée à la grille P2P-MPI, que nous validerons enfin par des expérimentations sur des applications types i.e des applications fréquentes dans les calculs parallèles.

Chapitre 2

État de l'art

L'allocation de tâches sur un ensemble de ressources est un problème largement traité dans la littérature, ceci étant certainement dû à l'aspect *NP-complet* du problème. En effet plusieures heuristiques ont été développées afin d'approcher une allocation optimale dans différents cas problématiques.

Le problème dans le cadre de ce mémoire est d'allouer le plus efficacement possible les tâches (processus) d'une application MPI sur un ensemble de processeurs formant une grille de calcul. Les applications MPI sont du type SPMD (Single Process Multiple Data-stream) où un même code paramétré par le numéro du processus est exécuté. Le numéro d'un processus MPI est appelé son *rang*. Les processus d'une application MPI démarrent en même temps et peuvent communiquer par envoi de message, et ce à tout moment pendant leur exécution. Le but de la recherche d'une allocation est de minimiser le temps d'exécution de l'application. Nous allons voir dans ce chapitre une énumération des principales techniques développées qui s'approchent de notre problème.

Une taxonomie des travaux déjà effectués dans le domaine de l'allocation permettra de déterminer une ligne directrice pour notre travail de recherche.

2.1 Modèles de représentation des applications parallèles

On trouve dans la littérature principalement deux catégories de méthodes d'allocation prenant en entrée deux modèles différents de représentation des applications parallèles. Ces modèles de représentation sont :

Les DAG : Un DAG (Direct Acyclic Graph), aussi appelé TPG (Task Precedence Graph), est un graphe $G = (N, E)$ où N est un ensemble de noeuds représentant des tâches. Une tâche est une suite de calcul i.e une séquence d'instructions ne contenant aucune primitive de communication. Chaque noeud $n_i \in N$ est pondéré par la complexité de calcul $w(n_i)$ de la tâche qu'il représente. E est l'ensemble des arcs représentant des dépendances de données (ou

communications) entre tâches. Une communication entre deux tâches n_i et n_j est notée :

$$e_{i,j} : n_i \rightarrow n_j$$

Chaque arc $e_{i,j}$ est pondéré par le volume de données transférées de n_i vers n_j.

Les communications, suivant le modèle DAG, ne peuvent être effectuées qu'au début et à la fin des tâches. En effet un arc $e_{1,2}$ partant d'un noeud n_1 vers un autre n_2 signifie la terminaison de n_1 avant le commencement de n_2. Les arcs repésentent ainsi une contrainte de précédence (dépendance temporelle) entre les tâches. [23],[14],[20]

Les STG : Un STG (Static Task Graph), aussi appelé TIG (Task Interaction Graph) est un graphe $G = (N, E)$ où N est un ensemble de noeuds représentant des tâches. Chaque noeud n_i est pondéré par la complexité de calcul $w(n_i)$ de la tâche qu'il représente. E est l'ensemble des arcs représentant des communications inter-tâches sans donner d'indication temporelle sur ces communications. Pour deux noeuds n_i et n_j, un arc noté $e_{i,j}$ joignant n_i à n_j représente une ou plusieures communications entre n_i et n_j.
$e_{i,j}$ est pondéré par le volume total des données transférées entre n_i et n_j dans les deux sens.

A la différence du DAG les communications n'induisent pas une dépendance de type fin-début, i.e une communication de n_1 à n_2 n'oblige pas à attendre la terminaison de n_1 pour commencer n_2. Les dépendances temporelles ne sont donc pas spécifiées dans ce modèle de graphe : deux tâches liées par un arc dans un STG peuvent être exécutées simultanément. Le modèle de représentation STG est donc adéquat pour les applications parallèles de type SPMD, où l'on a une tâche unique qui démarre *simultanément* sur plusieurs processeurs. [5], [9], [15].

2.2 Notre problème

Tout d'abord, il est important de clarifier la différence entre *ordonnancement* et *allocation*. L'ordonnacement est la recherche de *où* et de *quand* exécuter une tâche en vue d'améliorer un critère de performance, comme la durée d'exécution totale de l'application par exemple. L'allocation se limite quant à elle trouver le placement des tâches sur les processeurs qui maximise un critère de performance (donc le *où* uniquement). Dans le cadre de ce mémoire nous cherchons à optimiser la parallélisation d'une application MPI. Les tâches d'une telle application peuvent démarrer simultanément et communiquer tout au long de leur exécution. Comme il n'y a pas de dépendances temporelles entre les tâches, notre travail d'optimisation se résume donc à trouver le meilleur placement des tâches sur les processeurs. On a donc affaire à un problème d'allocation. Dans le cadre du problème d'allocation deux cas de figure se présentent :

- **Le nombre de processeurs est *inférieur* au nombre de tâches**
  Dans ce cas on retombe dans le problème de l'ordonnancement car on doit

prendre en compte les facteurs temps et précédence lors de l'allocation des tâches, notamment pour éviter des situations de blocage. Par exemple dans le cas où l'on n'assigne qu'une tâche par processeur, des tâches qui occupent tous les processeurs peuvent rester indéfinément en attente d'une communication de la part d'une tâche non allouée (et qui ne pourra jamais l'être).

- **Le nombre de processeurs est *supérieur* au nombre de tâches** Ce cas est le plus probable dans le contexte de ce mémoire puisqu'on dispose d'une grille de calcul ayant un grand nombre de processeurs.

Le problème de l'allocation est donc un cas particulier (sous-problème) de celui de l'ordonnancement, mais demeure cependant NP-Complet [13]. L'approche choisie dans ce cas pour obtenir des allocations satisfaisantes en un temps polynomial est d'utiliser des méthodes heuristiques. Nous allons, dans la section suivante, classifier et détailler quelques heuristiques d'allocations.

2.3 Approches pour résoudre le problème d'allocation

2.3.1 Ordonnancement de DAG

Beaucoup d'heuristiques ont été développées dans la littérature pour ordonnancer les tâches d'un DAG de manière presque optimale, [4], [21], [7], [12]. Cette grande variété de solutions, ne peut malheureusement pas être directement adoptée dans notre cas, où l'on alloue les processus d'une application MPI. En effet, les tâches d'une application MPI n'ont pas de relation stricte de précédence puisqu'elles commencent toutes en même temps. De ce fait elles sont modélisées dans un STG.

Une solution possible pour pouvoir profiter de ces méthodes serait de transfomer le graphe STG en un graphe DAG. Ceci peut être réalisé en partitionnant chaque tâche (noeud) du STG en ses phases de calcul comprises entre deux primitives de communications. On obtient ainsi des sous-tâches inter-dépendantes à leurs début et fin et donc une représentation de l'application conforme au modèle DAG. Prenons par exemple deux tâches T_1 et T_2 représentées dans le graphe STG de la Fig. 2.1

La tâche T_1 se décompose en trois phases de calcul brut (ne contenant pas de communications) : A_1, A_2 et A_3. T_2 se décompose en B_1 et B_2. L'exécution détaillée des deux tâches est montrée dans la Fig. 2.1

En considérant les phases de calcul brut, on peut transformer l'intéraction modélisée dans le STG entre T_1 et T_2 en un DAG représentant des contraintes de précédence entre A_1, A_2, A_3, B_1 et B_2. La Fig. 2.2 donne ce DAG. Cependant cette solution est loin d'être pratique car :

- **elle donne des graphes de très grande taille** : ce qui augmente grandement la complexité de l'allocation, et donc le temps nécessaire à la calculer.

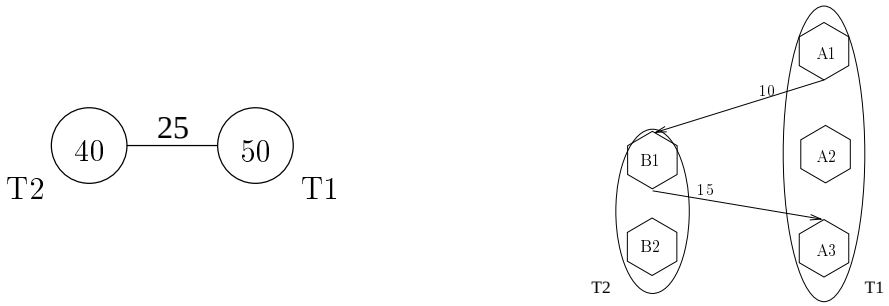


FIG. 2.1 – Graphe STG de T_1 et T_2, Graphe d'interaction entre T_1 et T_2

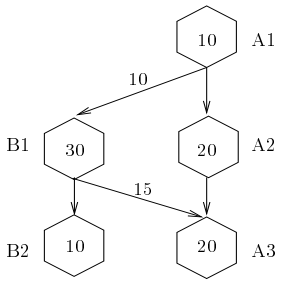


FIG. 2.2 – Graphe DAG résultant

– **elle introduit une forte contrainte de cohérence** : en effet on doit s'assurer que les sous-tâches d'un même processus MPI soient placées sur le même processeur afin de maintenir la cohérence des données et du programme. Ceci inhibe grandement le fonctionnement des heuristiques prévues pour le modèle DAG.

2.3.2 Allocation de STG

Les techniques et heuristiques concernant l'ordonnacement d'un STG sont quant à elles moins nombreuses dans la littérature.

Elles peuvent être classifiées selon deux critères principaux :

– **La nature des ressources cibles :** qui peuvent être soit *homogènes* ; Par exemple dans le cas d'un unique cluster avec des machines et liens identiques ; ou *hétérogènes* dans le cas d'une grappe de cluster avec plusieures architectures de machines et des connexions inter-cluster à débits variables.

– **Le graphe modélisant l'application :** qui peut être soit *régulier* ; c'est à dire pouvant être construit en répétant itérativement un sous-graphe qu'on appelle *motif* ; ou *irrégulier*, avec des connexions arbitraires entre ses noeuds.

La table suivante indique la classification des méthodes d'allocations que l'on

va décrire.

	ressources homogènes	ressources hétérogènes
graphe régulier	OREGAMI[17]	PROPHET[24]
graphe irrégulier	TTIG[22]	Bowen, Ghafoor [6]

2.3.3 Cas des ressources homogènes et graphe d'application régulier

Dans [17] les auteurs définissent un outil d'allocation de tâches nommé OREGAMI, allouant des applications parallèles à passage de messages sur des architectures parallèles.

OREGAMI traite le problème de l'allocation en exploitant des régularités dans la structure de l'application parallèle ainsi que dans les architectures cibles. Les auteurs se basent sur l'hypothèse que beaucoup d'applications parallèles sont caractérisées par des motifs de communication réguliers. L'architecture du système parallèle cible est régulière c'est-à-dire formée de processeurs homogènes connectés par une topologie de réseau régulière (hypercube, mesh, deBruijn). L'allocation dans OREGAMI est faite en trois étapes : la *contraction* du graphe des tâches en un graphe plus petit, dans le cas où le nombre de tâches dépasse celui des processeurs, *l'affectation* des clusters de tâches contractées à des processeurs et le *routage* des messages à travers le réseau d'interconnexion afin de minimiser les contentions.

2.3.4 Cas des ressources homogènes et graphe d'application irrégulier

Certaines approches dans ce cas de figure se basent sur le principe de "clusterisation" qui consiste à regrouper plusieurs noeuds d'un graphe de tâches afin de les affecter à une même ressource. Ce procédé de clusterisation permet d'obtenir des graphes de tâches plus simples (et donc plus réguliers). Par exemple dans [11] les auteurs se placent dans le cadre d'un unique cluster de machines avec des processeurs et interconnexions homogènes. Les applications qu'ils parallélisent sur ce système ont un graphe irrégulier. Les auteurs définissent une extension au modèle de graphe STG, sous le nom de TTIG (Temporal Task Interaction Graph) permettant de préciser non seulement les poids en communication et calcul des tâches, mais en plus un degré de concurrence entre elles. Ils développent une stratégie d'allocation nommée MATE (Mapping Algorithm based on Task dEpendencies) qui tire avantage de ce modèle en évaluant l'utilité du regroupement de deux tâches.

modèle TTIG [22] Les noeuds de ce modèle ont la même signification que dans le modèle STG à savoir qu'ils représentent les tâches pondérées par leur complexité en calcul. Un arc d'un graphe TTIG dirigé d'un noeud T_i vers T_j représente le volume total des communications effectuées de T_i vers T_j. La spécificité du modèle TTIG réside dans le fait qu'il associe un paramètre additionnel

aux arcs, qui correspond au *degré de parallélisme*. Pour deux tâches T_i et T_j adjacentes, i.e liées par un arc $e_{i,j}$ allant de T_i vers T_j, le degré de parallélisme p associé à $e_{i,j}$ est :

$$p = \frac{T_{c_p}}{T_{c_j}}$$

où T_{c_p} correspond au temps pendant lequel T_i et T_j s'exécutent en parallèle. T_{c_j} est le temps d'exécution toal de T_j.

L'appellation *Temporal* TIG dérive de ce nouveau paramètre car il devient possible de modéliser une relation de précédence dans le TTIG. En effet la relation " T_i précède totalement T_j " se modélise dans le TTIG par un degré de parallélisme associé à l'arc allant de T_i à T_j égal à 0.

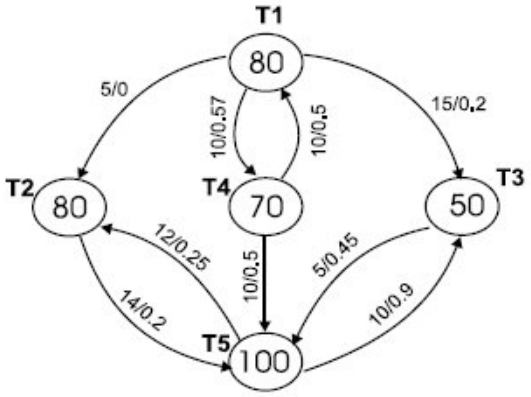


FIG. 2.3 – un graphe TTIG

algorithme MATE [11] L'objectif de cet algorithme est de minimiser le temps d'exécution de l'application en utilisant le modèle TTIG défini précédemment. L'idée principale est de "clusteriser" les tâches fortement dépendantes c'est-à-dire de les associer au même processeur. MATE construit une liste des tâches à exécuter en leur associant une priorité. Cette liste est dynamique i.e recalculée a chaque affectation d'une tâche à un processeur.

L'algorithme commence par définir la liste des noeuds initiaux. Un noeud est dit initial si aucune contrainte de communication ne l'empêche de démarrer au début de l'application. On introduit alors une notion de *niveau*. Les noeuds à une même distance en arcs des noeuds initiaux sont dits du même niveau. Pour chaque noeud t_i on calcule son niveau $N(T_i)$ correspondant à sa distance minimale en nombre d'arcs d'un noeud initial quelconque. Dans le graphe de la Fig.2.3, T_1 est la seule tâche initiale et on a les valeurs $N(T_1) = 0$, $N(T_2) =$

1, $N(T_3) = 1$, $N(T_4) = 1$ et $N(T_5) = 2$. L'algorithme considère ensuite les noeuds selon un ordre croissant des niveaux. L'idée principale est d'évaluer la dépendance des tâches du niveau actuel (tâches en cours d'allocation) avec celles des niveaux précédents (tâches déjà allouées). Si une tâche du niveau actuel est fortement dépendante d'une tâche déjà allouée alors elle est groupée avec celle-ci i.e affectée au même processeur. Cette évaluation de la dépendance se fait grâce au paramètre de degrès de parallèlisme entre tâches indiqué dans le graphe TTIG. L'algorithme évalue le *gain* à associer une tâche T_i en cours d'allocation au même processeur qu'une tâche T_j déjà allouée et adjacente [1] à T_i. Pour cela l'algorithme calcule les valeurs suivantes :

- **temps d'exécution si agrégés :** qu'on note t_a. C'est le temps minimum nécessaire pour exécuter la tâche Ti sur le même processeur que Tj. Ce temps est donné par la formule :

$$t_a = T_{c_i} + T_{c_j} + T_{a_j}$$

  où T_{c_i} et T_{c_j} sont les temps de calcul respectifs de T_i et T_j. T_{a_j} correspond au temps accumulé de T_j dû à l'affectation d'autres tâches adjacentes au même processeur. Ce temps accumulé sert a éviter d'affecter trop de tâches à un même processeur.
- **temps d'exécution si disjoints :** qu'on note t_d. C'est le temps d'exécution minimal des deux tâches lorsque Ti est affectée à un processeur différent de celui de Tj. Ce temps est égal à

$$t_d = T_{com} + T_{c_i} + T_{c_j} - T_{par}$$

  où T_{com} correspond au temps de communication entre T_i et T_j, T_{par} est le temps durant lequel T_i et T_j s'exécutent en parallèle, ce temps obtenu grâce au degré de parallélisme indiqué dans le graphe TTIG.

le gain est alors calculé comme suit :

$$gain = t_d - t_a$$

Une valeur positive du gain indique une forte dépendance de T_i avec T_j. T_i est donc affectée au même processeur que T_j. Si le gain est négatif, T_i est alors affectée au processeur le moins chargé.

2.3.5 Cas de ressources hétérogènes et graphe régulier

Dans [24] les auteurs proposent une plateforme d'allocation de tâches sur *une grappe de clusters* (groupement de plusieurs clusters distants), guidée par des informations sur l'application et les ressources. Un modèle de coût tenant compte du calcul ainsi que des communications y est proposé. Des heuristiques utilisant ce modèle de coût sont aussi proposées.

[1] deux tâches sont dites adjacentes si elle sont reliées par un arc

Le modèle de côut : Le coût des communications intra-cluster dépend de l'interconnexion des processeurs, de la topologie des communications, de la taille des messages échangés ainsi que du nombre de processeurs participants à la communication. Les auteurs expriment des coûts pour des topologies de communication régulièrement trouvées dans les applications parallèles :

- Unidimensionnelle : fréquente dans les problèmes de calcul scientifique basés sur les grilles et matrices. Les processeurs échangent des messages avec leurs deux voisins (nord et sud)
- Anneau : Chaque processeur reçoit de son voisin gauche et envoit au droit.
- Arbre : souvent utilisée dans les applications ayant des opérations globales comme la réduction.

Ce coût est alors exprimé comme suit :

$$T_{comm}[C_i, \tau](mtaille, p) = (c_1 * p) + (p * mtaille)/c_2$$

où C_i est le cluster sur lequel les tâches sont placées, τ est la topologie de communication de l'application, $mtaille$ est la taille de messages (nombre d'octets) transmis en un cycle d'itération de l'exécution, p est le nombre de processeurs sélectionnés communiquants, c_1 est la latence du réseau, c_2 est la bande passante.

Il se peut que toutes les tâches ne puissent pas être placées sur un même cluster. La communication entre différents clusters entraîne une pénalité de routage. Il faut donc considérer en plus, dans la fonction de coût de communication, un coût de routage.

$$T_{routage}[C_i, C_j](mtaille) = r_1 + r_2 * mtaille$$

où r_1 est une pénalité de latence, et r_2 une pénalité par octet routé de C_i vers C_j. Une fonction de coût total de communication est alors définie comme suit :

$$T_{comm_{total}}[C_i, \tau] = T_{comm}[C_i, \tau] + k * T_{routage}[C_i, C_j]$$

où k est le nombre de messages routés par cycle. Cette fonction dépend donc de la topologie de communication τ de l'application, les auteurs donnent les formules pour les trois topologies citées précédemment :

$$T_{comm}[1-D] = max_i T_{comm[C_i, 1-D]}$$

$$T_{comm}[Anneau] = sum_i T_{comm}[C_i, Anneau]$$

$$T_{comm}[Arbre] = T_{comm}[C_{racine}, Arbre] + max_{i \in feuilles} T_{comm}[C_i, Arbre]$$

Pour la topologie $1-D$, tous les processeurs communiquent simultanément. Donc le coût de communication est limité par le cluster le plus lent. Pour le cas de $l'Anneau$, les processeurs reçoivent d'abord de leur voisin gauche puis envoient au droit. Le coût de communication est donc additif. Pour la topologie $Arbre$, les feuilles communiquent de manière simultanée entre elles puis envoient

à leur noeud père. Le coût de communication pour cette topologie est donc défini de manière récursive.

Une fonction de coût de calcul est aussi définie comme suit :

$$T_{cal} = ctaille * coutarchi * nbPDU/p$$

où $nbPDU$ est le nombre de PDU (Primitive Data Unit : plus petit morceau du domaine de données, par exemple 4 octets) manipulés durant la phase de calcul, *ctaille* est la quantité de calcul exécuté sur un PDU, *coutarchi* est un coût d'exécution spécifique à l'architecture du processeur.

Le coût d'exécution total de l'application est alors finalement donné par

$$T_{exe} = nbcycles * (T_{cal} + T_{comm})$$

où nbcycles est le nombre de cycles d'itération de l'exécution.

Les heuristiques : Les auteurs proposent ensuite deux heursistiques, H1 et H2 qui cherchent des allocations minimisant le coût exprimé ci-dessus.

- Heuristique H1 : Cette heuristique se place dans le cadre d'un système avec des capacités de communication identiques entre les ressources. Dans cette hypothèse, il s'agit de baser la recherche sur la puissance de calcul des clusters. Les clusters sont donc classés selon la puissance cumulée de leurs processeurs. On commence par explorer le meilleur cluster. On attribue à chaque tâche un processeur disponible. S'il n'y a plus de processeurs disponibles et s'il reste des tâches à placer, on explore le cluster suivant dans le classement, et ainsi de suite.
- Heuristique H2 :
  Cette heuristique se place dans un cadre plus général, où les capacités de communication du système sont hétérogènes. Un simple classement des clusters selon la puissance n'est alors plus approprié car le coût de routage vers un cluster puissant peut être trop pénalisant par rapport à un routage vers un cluster certes moins puissant mais nécessitant des temps de communication inférieurs. L'heuristique H2 est basée sur la valeur de T_{exe} définie précédemment, qui tient compte à la fois des temps de calcul et des temps de communication. Tout d'abord, on établit un nouveau classement d'exploration des clusters. En procédant toujours dans l'ordre des clusters établi par H1, on cherche, pour chaque cluster pris individuellement, celui permettant d'obtenir une valeur de T_{exe} minimale. On peut remarquer que si un cluster ne dispose que d'un nombre inférieur de ressources par rapport au nombre de tâches à placer, certaines de ces tâches sont, lors de cette phase d'évaluation de clusters uniquement, considérées comme en attente sur certaines ressources derrière des premières tâches déjà placées. Les clusters sont ensuite classés suivant la valeur minimale de T_{exe} obtenue. On explore chaque cluster dans l'ordre du classement en exécutant deux phases :

- Phase 1 : On emploie la même stratégie que dans H1. On sélectionne tous les processeurs disponibles et on attribue une tâche à chaque processeur sélectionné.
- Phase 2 : On essaie de réduire la composante T_{comm} de T_{exe} . Si nous sommes dans le cas d'un placement interclusters (toutes les tâches n'ont pas pu être placées sur le même cluster), on considère parmi les clusters explorés celui qui engendre le plus de temps de communication T_{comm}. On enlève une des tâches placées sur un processeur sélectionné et on la place sur le cluster en cours d'exploration, ceci tant que le temps de communication global diminue et tant qu'il existe des processeurs disponibles sur le cluster en cours d'exploration. A ce stade, nous avons sélectionné des processeurs diminuant le temps de communication mais appartenant à un cluster moins bien classé. Ce sont donc des processeurs peut être moins puissants, engendrant ainsi une hausse du temps de calcul. Il s'agit donc de considérer la configuration, parmi celles obtenues avant et après la minimisation de la composante $Tcomm$, qui permet d'obtenir un temps d'exécution minimal.

2.3.6 Cas de ressources hétérogènes et graphe irrégulier

Dans [6] les auteurs proposent une méthode de clusterisation hiérarchique qui prend en entrée le graphe des tâches ainsi que celui des processeurs. Les tâches fortement communiquantes sont groupées ainsi que les processeurs topologiquement proches afin d'obtenir des graphes plus simples. L'algorithme de clusterisation génère ainsi deux arbres, un pour les processeurs et un pour les tâches, qu'il cherche ensuite à combiner.

Algorithme de clusterisation hiérarchique : L'algorithme de clusterisation présenté dans l'article est utilisé pour le graphe d'application ainsi que pour le graphe du système parallèle. Il est une variation des algorithmes de clusterisation par *agglomération* où l'on part d'un ensemble de tâches que l'on considère comme clusters et qu'on groupe pour former de plus grands clusters. L'entrée de l'algortihme est un graphe $G = (V, F)$ où V est un ensemble de noeuds et F un ensemble d'arcs. Un ensemble E contient les poids associés aux arcs de F. Une notion d'*affinité* est utilisée pour le groupement des noeuds. Plus le poids d'un arc reliant deux noeuds est grand plus leur affinité est grande. L'algorithme fonctionne en effectuant plusieurs passes intermédiaires. Durant une passe les noeuds avec la plus grande affinité sont groupés dans des clusters. La passe termine lorsque tous les noeuds ont été ajoutés à un cluster. Il est à noter qu'un cluster peut ne contenir qu'un seul noeud (noeud isolé i.e ne communiquant pas avec les autres).

La première étape d'une passe intermédiaire est de sélectionner un noeud *pivot*. Ce noeud est celui adjacent à l'arc le plus large (au plus grand poids) du graphe. Puisqu'il y a au minimum deux noeuds remplissant cette condition, on les départage d'abord selon :

- le plus grand nombre d'arcs rejoignant le noeud.

– le plus petit indice de noeud.

La prochaine étape de la passe est alors de sélectionner les voisins non clusterisés du pivot. Ceux-ci sont ordonnés par affinité décroissante. Un paramètre seuil est alors utilisé pour sélectionner les candidats à l'inclusion dans un cluster. Ce paramètre a pour rôle de permettre que l'ensemble des arcs contenus dans un cluster aient approximativement les mêmes poids. Si le poids associé à l'arc liant un voisin du pivot au pivot peut être considéré similaire (selon le paramètre seuil) à ceux liant les membres du cluster au pivot, ce voisin est ajouté au cluster. Par exemple prenons le cas des noeuds P, A, B et C de la figure 2.4 où P est le noeud pivot et A et B on été groupés avec P. C peut aussi être ajouté au cluster car le poids de son lien avec le pivot est assez semblable à ceux des liens compris dans le cluster. Une fois qu'un noeud est ajouté au cluster, il est aussi considéré comme pivot. Donc si C est ajouté au groupe on considère ses voisins non clusterisés pour une inclusion probable au cluster courant. A la fin de la passe les groupes de noeuds à forte affinité (clusters) sont identifiés et un nouveau graphe est construit en remplacant chaque cluster par un noeud le représentant. On applique ensuite une deuxième passe sur ce nouveau graphe. L'algorithme de clusterisation s'arrête lorsque le graphe ne contient plus qu'un seul noeud.

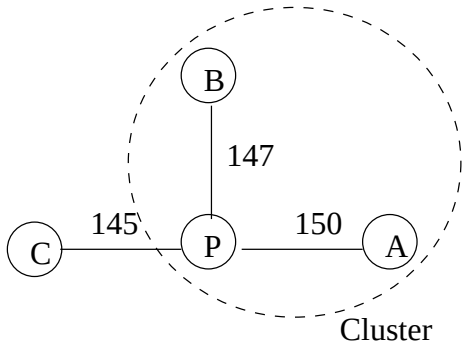


FIG. 2.4 – Exemple de clusterisation

Cette clusterisation est dite hiérarchique car elle permet de construire un arbre retraçant les clusterisation au fil des passes. Dans l'exemple de la figure 2.5, la première passe de l'algorithme va identifier deux clusters, l'un comprenant les noeuds a, b, c et d, et l'autre comprenant e et f. Appelons ces clusters g et h respectivement. La deuxième passe de l'algorithme prend alors en entrée un graphe contenant les deux noeuds g et h représentants les clusters identifiés dans la passe précédente. Ces deux noeuds sont regroupés et l'algorithme de clusterisation termine.

Algorithme d'allocation : L'algorithme d'allocation proposé dans l'article prend en entrée un arbre résultant de la clusterisation du graphe des processeurs

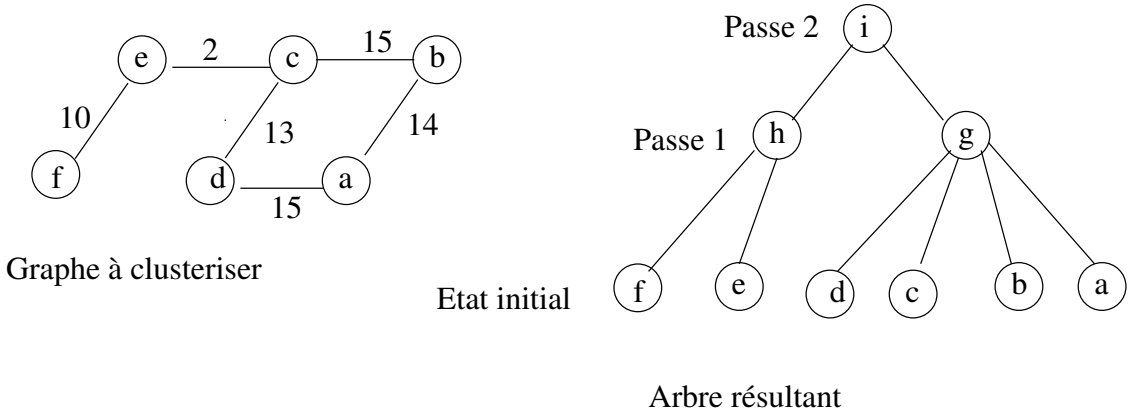


FIG. 2.5 – Arbre de clusterisation

ainsi qu'un arbre résultant de la clusterisation du graphe des processus. Associées à chaque feuille k de l'arbre de clusterisation des processeurs (à chaque processeur donc), on trouve les valeurs L_k et U_k définissant respectivement la charge de travail minimum et maximum souhaitée pour la machine. Ces valeurs sont répercutées en valeurs m_i et M_i, aux noeuds non feuilles(clusters) R_i tels que :

$m_i = \sum L_k$ et $M_i = \sum U_k$ où les k sont les feuilles du sous-arbre enraciné en R_i

On définit aussi une valeur de violation V_i correspondant à la violation de la contrainte de charge de travail minimum :

$$V_i = \frac{m_i - c_i}{m_i}$$

où c_i est la charge de travail courante sur R_i. Avec cette valeur de violation deux ensembles peuvent être définis :

$$S_v = \{V_i | V_i > 0\} \text{ et } S_a = \{V_i | V_i \leq 0\}$$

L'ensemble S_v contient tous les processeurs n'ayant pas encore remplit leur contrainte de charge minimum. Lorsque S_v est ordonné par V_i décroissants, la première machine dans l'ordre est celle qui est la plus loin de sa charge de travail minimum, c'est donc à elle qu'on doit affecter du travail en priorité. Une fois qu'une machine remplit sa contrainte de charge de travail minimum, elle passe dans l'ensemble S_a. Cet ensemble appelé *auxiliaire*, regroupe les machines ayant atteint leur charge de travail minimum et qui sont encore en dessous de la maximum.

L'algorithme récursif d'allocation est appelé avec les paramètres (r, R) où r est un noeud de l'arbre des processus et R un noeud de l'arbre des processeurs.

Pour l'appel initial (r, R) sont les racines des deux arbres. A ce stade l'algorithme vérifie la faisabilité de la solution. Si le nombre de processus n'est pas compris entre la charge de travail minimum et maximum de tous les processeurs alors il n'y a pas de solution. L'algorithme vérifie ensuite si le noeud R de l'arbre des processeurs est une feuille (donc une machine). Dans ce cas tous les fils de r sont assignés au processeur R. Cette deuxième vérification est la condtition d'arrêt de l'algorithme récursif d'allocation.

L'algorithme boucle tant que les fils de r n'ont pas tous été alloués. Il sélectionne d'abord le cluster de processeurs R_m, fils de R avec la plus grande valeur V_i i.e le cluster de processeurs le plus loin de sa charge de travail minimum. Il sélectionne ensuite r_m : le fils de r ayant le plus grand nombre de fils qu'il alloue au cluster séléctionné précédemment en réitérant l'algorithme avec les paramètres (r_m, R_m).

2.3.7 Autres heuristiques intéressantes

D'autres méthodes ne rentrent pas dans la classification décrite ci-dessus, du fait qu'elle aient un but très restreint mais néanmoins intéressant. Dans [19] par exemple les auteurs proposent une méthode dont le but est de réduire le coût en communication d'une application parallèle. La démarche consiste à faire correspondre au mieux le graphe d'interconnexion des processeurs à celui des communications inter-tâches. Afin de pouvoir estimer le coût en communication généré par une allocation possible, les auteurs définissent trois problèmes principaux :

- **Déterminer le besoin en communication de l'application parallèle :** Pour pouvoir déterminer *exactement* ce besoin, il faut exécuter totalement l'application en mesurant les volumes de données échangés entre tâches. Cependant les auteurs font appel à leur méthode d'allocation pendant l'exécution de l'application et ne souhaitent pas gaspiller toute une exécution pour extraire le graphe de communications. Ils se basent sur une assomption de régularité dans le graphe de communication des applications parallèles. Ils surveillent donc le comportement de l'application pour une durée relativement petite par rapport à son temps d'exécution total afin de détecter des motifs réguliers et prédire le besoin total en communication.
- **Déterminer les ressources réseau disponibles sur le système** Cette information est représentée sous la forme d'une table de distance où un coût est associé à chaque communication possible entre les processeurs du système.
- **Définir un critère de sélection des allocations**
  Ce critère est évalué à partir des deux tables de distances entres tâches et processeurs. La méthode consiste à minimiser le coût total en communication de l'application. Ce coût est évalué comme suit :

$$C = \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij}.d_{m_i m_j}$$

où c_{ij} représente le volume d'une communication de la tâche T_i à T_j, et $d_{m_i m_j}$ le coût d'une communication du processeur m_i associé à T_i vers m_j associé à T_j.

L'algorithme proposé dans l'article commence par effectuer une allocation aléatoire qu'il évalue, puis la perturbe en permutant deux placements, puis réévalue le coût en communication. Si le coût est réduit la permutation est validée. Ce processus s'arrête lorsque le coût n'a pas diminué depuis un nombre fixé de permutations. A ce stade l'allocation en cours et son coût sont enregistrés et le processus précédent recommence avec une nouvelle *graine* (allocation aléatoire de départ). L'algorithme s'arrête lorsqu'un nombre souhaité de graines a été évalué.

Chapitre 3

Proposition d'une méthode d'allocation

3.1 P2P-MPI

P2P-MPI peut être vu comme une implémentation de MPI en JAVA. Cependant, les fonctionnalités de P2P-MPI dépassent largement celles d'une simple librairie de communication, ce qui rend l'exploitation des applications plus confortable qu'avec un environnement traditionnel de programmation parallèle. Nous donnons ici une description rapide des fonctionnalités de P2P-MPI, le lecteur intéressé peut se référer à [10] pour plus de détails.

L'objectif premier de P2P-MPI est de fournir un environnement en *grille* pour la programmation d'applications parallèles. P2P-MPI a deux rôles : c'est un **middleware**(intergiciel) et a donc le devoir d'offrir des services au niveau système à l'utilisateur tels que la découverte des ressources demandées, le transfert de fichiers, le lancement de tâches à distance, etc. Le deuxième rôle est celui de l'**API** (Application Programming Interface) de programmation parallèle qu'il fournit au développeur.

3.1.1 API

La plupart des projets de recherche ayant pour but la formation d'une grille de calcul permettent le calcul de travaux constitués de tâches indépendantes uniquement, avec comme modèle de programmation sous-jacent le mode client-serveur (ou RPC). L'avantage de ce modèle est qu'il est adéquat pour les environnements de calcul distribué mais il limite cependant la diversité des constructions parallèles à des applications de type client-serveur. P2P-MPI offre un modèle de programmation plus général basé sur le passage de messages, dont le mode client-serveur peut être vu comme un cas particulier. On trouve dans la distribution P2P-MPI une libraire de communication ressemblant à l'API MPI.

L'implémentation de la spécification MPI est faite en Java en suivant les

recommendations MPJ [8] dans lesquelles les primitives ressemblent fortement à la spécification C/C++/fortran [18].

La motivation première de l'utilisation de Java est la portabilité des codes sur la grille qui est par nature hétérogène.

3.1.2 Middleware

Un utilisateur souhaitant faire partager les ressources de sa machine peut la faire rejoindre une grille P2P-MPI en tapant simplement `mpiboot` qui lance un processus *gatekeeper*. Le gatekeeper joue deux rôles :

- il déclare la machine locale comme étant disponible au reste de la communauté, et décide d'accepter ou de décliner les requêtes des autres "pairs".
- lorsque l'utilisateur émet une requête de job, le gatekeeper est chargé de trouver le nombre requis de machines ainsi que d'organiser le lancement du job.

Le lancement d'un job MPI requière l'assignation d'un identifiant unique pour chaque tâche (ou processus) puis de synchroniser tous les processus à la barrière MPI_Init.

Nous détaillerons le processus de découverte des machines dans la section suivante.

Une fois que suffisamment de ressources ont été sélectionnées, le gatekeeper transmet à chaque hôte sélectionné le programme à exécuter ainsi que les données en entrée ou une URL pour les y chercher. Lorsque tous les hôtes ont acquité le transfert, une table listant les identifiants assignés à chaque processus est diffusée, et l'application peut alors démarrer.

3.2 Mécanismes de découverte des pairs et allocation dans P2P-MPI

Les mécanismes pair à pair de P2P-MPI sont entièrement basés sur JXTA [2]. JXTA est un projet Open Source lancé par Sun Microsystems en avril 2001. JXTA vient du mot anglais « Juxtapose ». Le but premier de JXTA est de pouvoir interconnecter n'importe quel système sur n'importe quel réseau. Le peer-to-peer permet d'interconnecter un ordinateur avec un PDA, un téléphone portable, etc. JXTA permet donc de créer une sorte de réseau applicatif au dessus des réseaux physiques. Les principaux éléments de JXTA sont :

- **Peer :** un noeud (une machine, un processeur, un utilisateur) d'un réseau P2P.
- **Peer group :** un ensemble de peers offrant un service spécifique
- **Rendezvous peer :** fournit à des peers des informations utiles pour découvrir d'autres peers ou des ressources de peer.
- **Router peer :** utilisé pour trouver un chemin de communication pour peers qui sont séparés par un firewall.
- **Pipe :** canal de communication permettant d'envoyer et de recevoir des messages. Un pipe est asynchrone.

- **Endpoint :** une destination logique à laquelle un pipe est lié. Exemple : AdresseIP :Port
- **Advertisements :** un document en XML qui nomme, décrit et publie l'existence de peer, peer group, de pipes ou de services.
- **Service :** fonctionnalités offertes par un peer pouvant être utilisées par d'autres peers.

De point de vue JXTA, une grille P2P-MPI est un Peer group offrant un service nommé P2P-MPI.

Un utilisateur souhaitant partager ses ressources de calcul doit d'abord rejoindre la plateforme P2P-MPI. Pour cela il suffit de lancer un processus appelé MPD. Dès que le MPD est lancé, il s'enregistre auprès du JXTA netPeerGroup (le Peer Group principal de JXTA), et interroge ensuite le Rendezvous peer pour un advertisement du peer Group P2P-MPI. S'il trouve un advertisement, il rejoint le peer Group. Sinon, il crée un nouveau peer group P2P-MPI, publie un advertisement correspondant et rejoint le peer group. Il publie ensuite un pipe advertisement ainsi qu'un advertisement spécial : le *MPD Advertisement* renseignant sur les caractéristiques de la machine (adresse IP, taille mémoire, CPU, système d'exploitation).

3.2.1 Recherche de noeuds pour exécuter une application P2P-MPI

Dans l'implémentation actuelle de P2P-MPI, lors du lancement d'une application parallèle comprenant N tâches depuis une machine exécutant un MPD, le MPD interroge le Rendezvous Peer pour $N-1$ Pipe advertisements(la machine à l'origine de la requête est obligatoirement participante avec les N-1 autres machines, elle a au moins le processus numéro 0). Le Rendezvous Peer lui renvoie la liste des $N-1$ premiers pipe Advertisements qu'il connait. Le MPD affecte alors les tâches aux hôtes selon une méthode *round-robin* suivant l'ordre dans lequel les hôtes sont renseignés dans la réponse du Rendez vous.

Par exemple dans le cas où un utilisateur souhaite lancer une application comptant 6 tâches parallèles T_0 à T_5, son MPD interroge le Rendezvous Peer pour trouver 5 hôtes exécutant un MPD. Si le Rendezvous peer lui renvoie la liste $\{P_3, P_8, P_1, P_5, P_9\}$, le MPD affecte les tâches selon le mode round-robin :

$$T_0 \rightarrow machine_{locale},\ T_1 \rightarrow P_3,\ T_2 \rightarrow P_8,\ T_3 \rightarrow P_1,\ T_4 \rightarrow P_5,\ T_5 \rightarrow P_9$$

3.3 Illustration du problème

Un programme parallèle s'exécutant sur la grille P2P-MPI est formé d'un ensemble de tâches disposant du même code source dont le comportement varie selon un paramètre entier appelé le *rang*. Afin d'avoir une exécution rapide du programme, une première idée intuitive serait d'affecter les tâches du programme aux machines les plus puissantes sur la grille. Cependant cette solution est loin d'être optimale car elle ne prend pas du tout en considération la qualité des

liens réseau ni les volumes de communications entre les tâches. Prenons par exemple le cas des tâches T_0, T_1, T_2 et T_3 du graphe STG de la fig 3.1. Le coût en calcul des tâches est indiqué au centre des cercles. Un arc reliant deux tâches indique des communications dont le volume est indiqué à côté de l'arc. On veut exécuter cette application sur la grille de la fig 3.2 formée des processeurs suivants listés par ordre décroissant de puissance : P_1, P_2, P_3, P_5 et P_4. Si on applique la méthode d'allocation proposée ci-dessus on se retrouve par exemple avec les affectations suivantes : $T_0 \rightarrow P_5$, $T_1 \rightarrow P_1$, $T_2 \rightarrow P_2$ et $T_3 \rightarrow P_3$. Supposons maintenant que les volumes de communication allant vers T_0 soient importants vis-à-vis de la capacité du lien réseau, les latences dûes au réseau peuvent donc inhiber l'accélération dûe à la puissance des machines. Il aurait donc été plus judicieux d'affecter T_0 à P_4 qui se trouve dans le même réseau que les processeurs où s'exécutent le reste des tâches. De même, si l'on envisage, lors de l'allocation, d'optimiser en priorité les communications réseau, on n'a alors aucune garantie de sélectionner des machines puissantes. Le problème dans le cadre de ce mémoire est donc de trouver une approche d'allocation qui tienne à la fois compte des volumes de communications et liens réseau ainsi que de la puissance des machines afin d'apporter une accélération dans la majorité des cas. Nous avons évoqué dans l'état de l'art l'aspect NP-Complet du problème d'allocation. Notre approche d'allocation sera donc une heuristique.

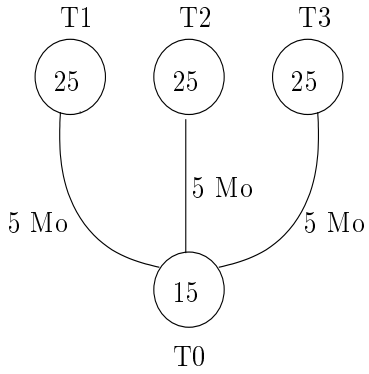


FIG. 3.1 – Exemple d'application maitre esclave

3.4 Conventions de représentation des graphes

Nous avons évoqué précédemment les deux types de graphes utilisés pour la représentation des applications parallèles, à savoir les *DAG* et les *STG*. Nous avons aussi remarqué que les applications MPI tombent dans le modèle SPMD qui est représenté par les graphes STG. Les graphes d'applications que nous allouons et que nous allons donc utiliser dans les exemples sont du modèle STG. Les volumes utilisés pour pondérer les arcs du STG correspondent aux volumes

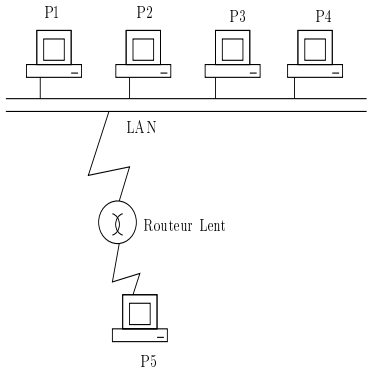


FIG. 3.2 – Grille de 5 processeurs

échangés dans les communications entre tâches. Les poids que nous choisissons pour pondérer les noeuds su graphe STG sont les temps d'exécutions *normalisés* des tâches. Par exemple pour un noeud n_i représentant une tâche T_i dans un graphe STG, le poids w_i pondérant n_i(inscrit dans le cercle représentant le noeud) est le temps pris par T_i pour s'exécuter sur une machine référence. Un exemple de graphe d'application se trouve dans la figure 3.1.

Les graphes représentant les grilles de machines sont quant à eux des graphes complets i.e tous les noeuds sont totalement interconnectés du fait qu'aucun poids d'arc ne soit négligeable. En effet, dans ces graphes les noeuds représentent des machines et les arcs des interconnexions entre elles. Comme les machines formant une grille P2P-MPI sont connectées à Internet (disposent d'une adresse IP, adresse JXTA) chaque machine peut joindre toutes les autres formant la grille selon des routes à différents débits. Les poids affectés aux arcs correspondent aux débits de ces routes. Par exemple pour deux noeuds n_i et n_j représentant respectivement deux machines P_1 et P_2 on a un arc $e_{i,j} : n_i \rightarrow n_j$ pondéré par $d_{i,j}$ le débit de la route allant de P_i à P_j, ainsi qu'un arc $e_{j,i} : n_j \rightarrow n_i$ pondéré par $d_{j,i}$ le débit de a route allant de P_j à P_i. Il est a noter que souvent $d_{i,j} \neq d_{j,i}$ vu que la route n'est pas la même dans les deux sens.
Les poids attribués aux noeuds sont les coefficients de performance des machines par rapport à une machine standard (ayant pour coefficient de performance 1). Pour obtenir ces valeurs nous partons de l'hypothèse que les performances des machines sont linéaires selon la complexité des applications. Il suffit alors d'exécuter sur une machine P_i une tâche prenant un temps t_d sur la machine référence. La performance de P_i est alors $\frac{t_i}{t_d}$.

3.5 Cadre

Le choix d'une allocation accélérant l'exécution de l'application parallèle devra se faire selon des critères stricts que l'on se fixe :

- Afin de respecter la granularité choisie par le programmeur et d'avoir une répartition équitable des charges entre les pairs on se fixe une contrainte de n'exécuter qu'une seule tâche par machine, i.e. on ne regroupe pas plusieurs tâches sur une même machine.
- Un autre critère concerne le temps de recherche de l'allocation proposée. Il est important que ce temps soit raisonnable par rapport à l'ampleur de l'application parallèle.

La grille P2P-MPI peut quant à elle être fortement hétérogène vue la nature pair à pair de l'intergiciel. Cependant, il est important de noter que le public intéressé pour former une grille P2P-MPI, dispose en général d'un ensemble de machines assez homogènes du point de vue architectures ou tout au moins du point de vue interconnexion réseau. La grille P2P-MPI serait donc, dans la majorité des cas, un ensemble hétérogène de sous-ensembles homogènes de machines, autrement dit, un groupement de mini-clusters.

3.6 Etude du comportement de l'application

Afin de pouvoir allouer les tâches d'une application parallèle de manière "intelligente" il est important d'en connaître le comportement exact. Ce *comportement* peut être modélisé par un graphe contenant les volumes des données échangées entre tâches ainsi que les poids en calcul de ces tâches. Ces informations peuvent être obtenues de deux manières :

- en étudiant une exécution entière de l'application,
- en prédisant son comportement : i.e en étudiant le comportement de l'application sur une courte période de son exécution afin de prédire son comportement pour le reste de l'exécution. Cette démarche se base sur l'hypothèse que les applications parallèles présentent souvent un comportement régulier.

Cependant, la prédiction pouvant être incorrecte, nous optons pour la première solution : suivre une exécution entière de l'application sur la grille afin de déterminer son comportement. Notre méthode d'allocation s'adresse donc à des utilisateurs souhaitant exécuter à plusieures reprises la même application sur la grille.

3.6.1 Construction des traces

Les informations nécessaires pour caractériser le comportement d'une application parallèle sont :

- Les volumes de données échangées entre les tâches : taille en octets des messages envoyés entre les tâches. Cette information peut être représentée sous la forme d'une matrice d'adjacence. Cette matrice d'adjacence est une matrice carée symétrique M_t où une valeur $M_t(i,j) = M_t(j,i) = V$ signifie que les tâches T_i et T_j se sont échangées V octets.
- Le poids en calcul des tâches : ces poids correspondent aux temps d'exécution des tâches normalisés par rapport à une machine standard.

Les volumes de communications peuvent être mesurés en modifiant les primitives de communication dans P2P-MPI. En effet les tâches d'une application P2P-MPI communiquent grâce à des fonctions fournies par l'API et dont la syntaxe ressemble fortement aux primitives de communication MPI standard[1] :

- **MPI.Send(java.lang.Object sendBuffer, int offset, int count, Datatype datatype, int dest, int tag) :** fonction d'envoi de *count* objets de type *datatype* vers le processus *dest*
- **MPI.Recv(java.lang.Object sendBuffer, int offset, int count, Datatype datatype, int src, int tag) :** fonction de réception de *count* objets de type *datatype* venant du processus *src*.
- ou encore **MPI.Gather(java.lang.Object sendBuffer, int sendOffset,int sendCount, Datatype sendType,java.lang.Object recvBuffer, int recvOffset,int recvCount,Datatype recvType,int root) :** qui permet de recevoir *recvCount* objets de type *sendType* venant de tous les autres processus.

On constate donc qu'il nous suffit de modifier les primitives de communication P2P-MPI afin qu'elles calculent la taille des données échangées, à partir des paramètres passés dans l'entête. On peut donc ainsi construire une trace que nous mémorisons dans l'objet *Trace*, contenant tous les volumes de données reçus et envoyés par le rang. Par exemple pour **MPI.Send(buf, 0, 1000, MPI.BYTE,3,0)**, le volume de données envoyé vers le rang 3 est égal à : $1000 * taille(MPI.BYTE) = 1000$ octects.

En plus des volumes, nous comptabilisons les temps passés dans certaines sections du code :

- le temps entre MPI.Init(), le début du programme et MPI.Finalize(), sa fin.
- les durées de chaque communication de type envoi (en même temps que le destinataire et le volume transféré).

Chaque pair envoie ensuite ces enregistrements au pair ayant le processus 0, qui s'occupera de construire le graphe total de l'application à partir des traces reçues de tous les rangs.

Le pair du rang 0, peut alors facilement construire la matrice d'adjacence des tâches M_t indiquant les volumes de données échangés entre les processus. Par contre les poids en calcul de ces tâches sont moins évidents à déterminer. En effet non seulement la durée d'exécution doit être normalisée, mais en plus le temps d'exécution du rang, i.e la différence entre son temps de terminaison et son temps de commencement, ne correspond pas à un temps de calcul pur mais prend aussi en compte des temps d'attente de messages comme par exemple dans le cas d'une réception bloquante `MPI.Recv`. Le temps de calcul pur peut être obtenu en additionnant les durées des phases de calcul comprises entre deux évènements de communication.
Cette somme doit ensuite être normalisée par rapport à une machine standard. En effet les temps mesurés dans les traces varient selon les performances des machines où ils ont été pris. Afin d'avoir des valeurs significatives on doit alors

[1] Elles suivent le standard MPJ

fixer une machine référence (standard) et attribuer aux autres machines une note représentant leur facteur de puissance par rapport à la machine standard. Il suffit ensuite de diviser le temps obtenu depuis les traces par ce facteur pour obtenir un temps d'exécution normalisé sur une machine standard que l'on pourra utiliser comme poids de calcul de la tâche. Les notes des machines peuvent être attribuée suite à l'exécution de benchmarks ou être spécifiées par l'utilisateur selon ses préférences.

3.7 Découverte des liens forts entre tâches

3.7.1 Objectif

Afin d'accélérer l'exécution d'une application parallèle sur la grille, il est important de réduire les temps d'attente de ses tâches dûs à des mauvaises communications(échange de gros volumes sur des liens faibles). Pour cela, les tâches échangeant des volumes de données importants doivent impérativement être affectées à des hôtes fortement interconnectés.

On cherchera donc lors de l'allocation à favoriser la localité spatiale des tâches fortement dépendantes i.e attribuer les tâches fortement communiquantes à des machines d'un même réseau ou à débit d'interconnexion fort. Pour ce faire, il est donc important d'identifier les groupes de tâches fortement communiquantes afin d'éviter le plus possible de les séparer. Nous proposons dans la suite un algorithme de découverte de ces groupes.

3.7.2 Algorithme de groupement des tâches

A la suite de la collecte des traces d'exécutions des tâches, on dispose d'une matrice d'adjacence M_t indiquant les volumes des données échangés entre les différentes tâches. Notre algorithme prend cette matrice en paramètre et en extrait des groupes de tâches fortement interconnectées. Une définition formelle de l'algorithme se trouve dans la figure suivante.

```
/* Ayant une matrice d'adjacence carrée M_t(N,N)                    */
Entrées: M_t
Soit Grouped = {};
/* Grouped est l'ensemble contenant les indices des tâches
   groupées                                                         */
Soit G = {} ;
/* G est l'ensemble des groupes de tâches                           */
Soit tolerance = 0.1;
Soit maxConn ;
/* maxConn indique la connectivité du groupe actuel                 */
tant que Taille(Grouped) < N faire
    /* on cherche un pivot de groupement                            */
    Soit ligne_pivot = min({i ∉ Grouped|∃(i,j) ∈ ℕ² tq j ∉
    Grouped et ∀(x,y) ∈ ℕ², M_t[i][j] ≥ M_t[x][y]}) ;
    on ajoute un nouveau Groupe à G ;
    maxConn = M_t[ligne_pivot][j] ;
    /* on appelle ensuite la fonction récursive pour le pivot
       */
    ClusterizeRank(ligne_pivot);
fin

/* Fonction ClusterizeRank(pivot) récursive de groupement           */
Entrées: indice de ligne du pivot p
On ajoute p au groupe actuel G[Taille(G)-1]];
/* on definit l'ensemble V_p des voisins de p                       */
Soit V_p = {v_i ∉ Grouped|M_t[p][v_i] ≠ 0} ;
V_p est ordonné selon des valeurs décroissantes de M_t[p][v_i];
pour tous les v_j ∈ V_p faire
    si maxConn * (1 − tolerance) < M_t[p][v_j] alors
        ClusterizeRank(v_j);
    fin
fin
```

Algorithm 1: Algorithme de Groupement des tâches

Etant donné une matrice d'adjacence M_t définie comme suit : M_t est une matrice symétrique carrée (N, N) où N est le nombre de tâches de l'application parallèle. La valeur $M_t[i][j]$ indique le volume de données échangées entre les tâches T_i et T_j. Cette valeur comprend les communications dans les deux sens, donc $M_t[i][j] = M_t[j][i]$.

L'algorithme commence par chercher un pivot de groupement. Le pivot est l'indice de la première tâche effectuant le plus gros volume de communication ; il correspond donc à l'indice de la première ligne contenant la plus grande valeur dans la matrice. Par exemple si $M_t[i][j] = M_t[j][i] = ValeurMax(M_t)$ et $i < j$ alors le pivot est i. On crée alors un nouveau groupe de tâches auquel on attribue une valeur de connectivité référence : $maxConn$. Cette valeur

va être utilisée dans l'ajout d'autres tâches au groupe. Après avoir identifié le premier pivot *pivot*, et créé le premier groupe $G[0]$, on appelle la fonction récursive *clusterizeRank*(*pivot*). Cette fonction ordonne les voisins du pivot selon un ordre décroissant de leur volume de communication avec le pivot. Elle évalue ensuite les voisins v_i dans l'ordre afin de tester s'ils vérifient la condition suivante :

$$maxConn * (1 - tolerance) < M_t[p][v_i]$$

où *tolerance* est un paramètre fixé au début de l'algorithme, compris entre 0 et 1 et permettant de définir un seuil de similarité pour les volumes de communication. Si v_i vérifie cette condition on le considère comme un pivot et on appelle la fonction récursive *clusterizeRank*(v_i). Lorsque toutes les fonctions récursives terminent, les tâches interconnectées par des volumes similaires et nettement supérieurs au reste des tâches forment le groupe courant G. A ce stade, si le nombre de tâches groupées est inférieur au nombre total de tâches N, on recommence là même manoeuvre à savoir chercher un nouveau pivot et créer un nouveau groupe. Sinon (si toutes les tâches ont été groupées), la découverte des groupes de tâches est alors terminée.

3.7.3 Complexité de l'algorithme

Pour évaluer la complexité de l'algorithme nous allons utiliser une mesure commune dans les algorithmes traîtant des graphes à savoir le nombre total de visites d'arcs. Dans notre structure de données représentant le graphe, une visite d'un arc se traduit par un accès à une valeur de la matrice M_t. Nous allons donc évaluer la complexité de l'algorithme en nombre d'accès à la matrice M_t.

Pour déterminer un pivot de groupement il faut trouver la valeur maximum de la matrice. Pour cela il nous suffit de parcourir la moitié des valeurs de la matrice M_t puisqu'elle est symétrique. Le processus de recherche d'un pivot de groupement se fait donc après $\frac{N^2}{2}$ accès à M_t où N est le nombre de lignes de M_t. L'appel à la fonction ClusterizeRank(*pivot*) nécessite $N * \frac{N-1}{2}$ accès à la matrice M_t pour ordonner les voisins du pivot selon des valeurs d'interconnexion décroissantes. En effet, ayant l'indice de la ligne du pivot, on effectue $N-1$ accès pour trouver la première valeur maximale puis $N-2$ puis $N-3$ ce qui nous donne la somme de la suite arithmétique : $1 + 2 + .. + N - 1 = N * \frac{N-1}{2}$.

Le processus de formation d'un groupe à partir d'une matrice carrée (N, N) a donc une complexité de $N * (N - \frac{1}{2})$ accès.

On peut remarquer que le pire cas pour cet algorithme est celui où à chaque appel de la fonction ClusterizeRank(*pivot*), on ne regroupe que le *pivot* et son voisin maximum. Dans ce cas le processus de formation d'un groupe doit se répéter pour une matrice carrée $(N-2, N-2)$ puis $(N-4, N-4)$, etc...

La complexité de l'algorithme de groupement s'exprime donc comme ceci :

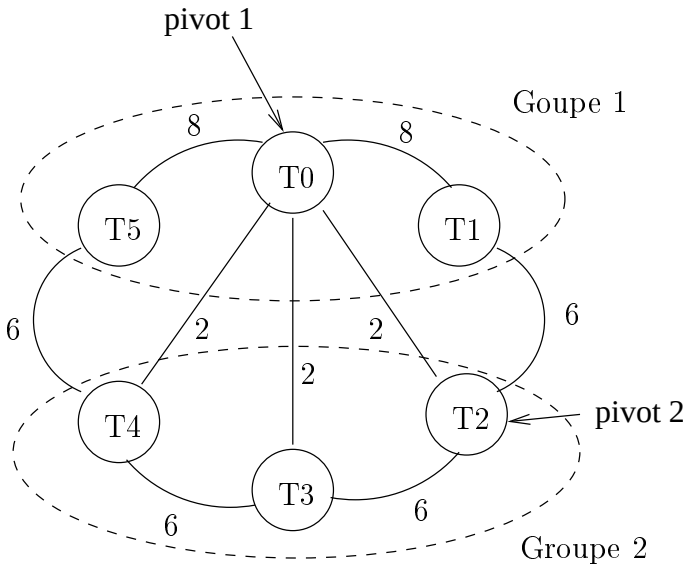
$$\sum_{k=0}^{\frac{N}{2}} (N - 2 * k)^2 - \frac{N - 2 * k}{2} \text{ si N est impair}$$

$$et \sum_{k=0}^{\frac{N}{2}-1} (N - 2 * k)^2 - \frac{N - 2 * k}{2} \text{si N est pair}$$

Cette somme a au plus $\frac{N}{2}$ termes, la complexité de l'algorithme de groupement des tâches est donc en $O(N^3)$.

3.7.4 Exemples de groupements

Nous montrons dans les figures 3.3 et 3.4 deux exemples de groupements montrant le fonctionnement de l'algorithme aussi bien sur des graphes d'application réguliers qu'irréguliers.



Tolérance = 0.1

Matrice d'adjacence :

$$\begin{bmatrix} 0 & 8 & 2 & 2 & 2 & 8 \\ 8 & 0 & 6 & 0 & 0 & 0 \\ 2 & 6 & 0 & 6 & 0 & 0 \\ 2 & 0 & 6 & 0 & 6 & 0 \\ 2 & 0 & 0 & 6 & 0 & 6 \\ 8 & 0 & 0 & 0 & 6 & 0 \end{bmatrix}$$

FIG. 3.3 – Exemple avec une application en Anneau

Déroulons le cas de la figure 3.3. Pour la première itération, nous recherchons un pivot dans la moitié supérieure de la matrice d'adjacence. La valeur maximum est le 8 se trouvant à la ligne 0. T_0 est donc choisi comme premier pivot de groupement. On crée alors un nouveau groupe g_0 qu'on ajoute à G. On initialise la valeur de connectivité de g_0 (mémorisée dans $maxConn$) à 8. Nous appelons ensuite la fonction Clusterizerank(0). A ce stade on ajoute T_0 à g_0 et on marque T_0 comme groupée en l'ajoutant au vecteur Grouped. On ordonne ensuite les voisins de T_0 selon un ordre décroissant des volumes de données échangés ; ce qui nous donne l'ordre suivant : T_1, T_5, T_2, T_3 et T_4. T_1 satisfait la condition de tolérance et peut donc être ajouté au groupe ; on appelle donc ClusterizeRank(1). A ce stade on ajoute T_1 à g_0 et on le marque comme groupé. On ordonne ensuite ses voisins non groupés ce qui donne : T_2, T_3, T_4 et T_5. En

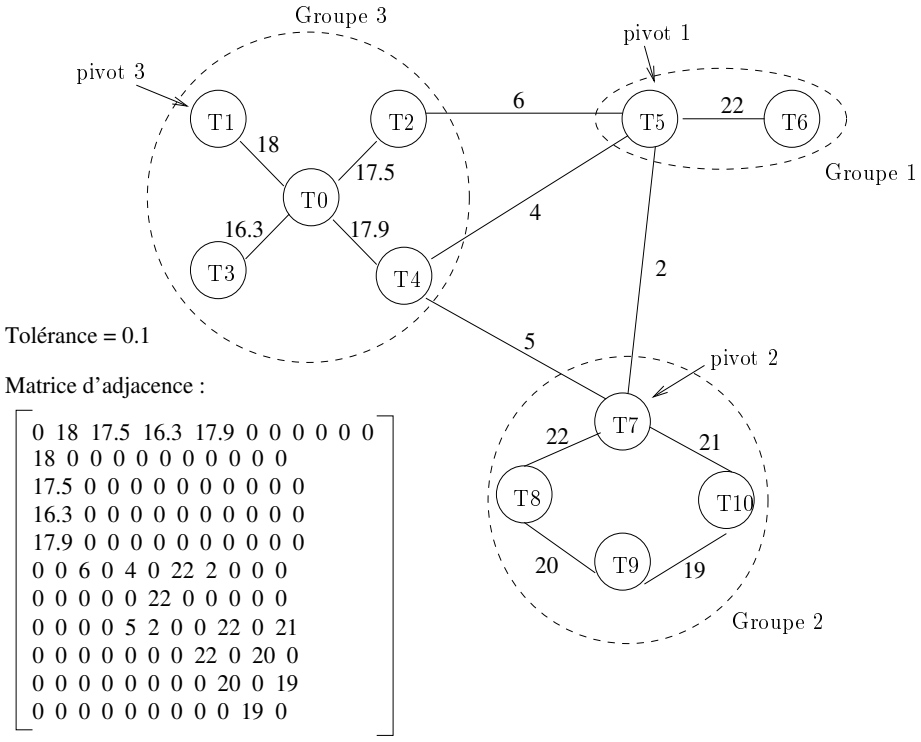


FIG. 3.4 – Exemple avec une application irrégulière

examinant la connexion de T_1 avec T_2 on constate qu'il ne satisfait pas la condition de tolérance ($6 < 7.2$). On ne cherche alors plus à intégrer les voisins de T_1 à g_0. On revient alors dans le contexte de ClusterizeRank(0). T_5 est le deuxième voisin le plus communiquant avec T_0 et il satisfait la condition de tolérance. On appelle donc ClusterizeRank(5). Pareil que précédemment, T_5 est ajouté à g_0 et l'ordre de ses voisins non groupés est : T_4, T_3 et T_2. Comme T_4 ne satisfait pas la tolérance, ClusterizeRank(5) termine ce qui termine ClusterizeRank(0) et du coup finalise le groupe g_0. On a donc à ce stade le groupe g_0 contenant T_0, T_1 et T_2. Les étapes suivantes sont :

- on recherche un nouveau pivot parmi T_2, T_3 et T_4. T_2 est choisi comme pivot (car la ligne 2 contient la valeur maximale 6 et a l'indice de ligne le plus petit).
- On crée un nouveau groupe g_1 avec $maxConn = 6$, puis on appelle ClusterizeRank(2).
- T_2 est ajouté à g_1. Ses voisins non groupés ordonnés : T_3 et T_4.
- T_3 satisfait la tolérance. On appelle ClusterizeRank(3).
- T_3 est ajouté à g_1. Ses voisins non groupés ordonnés : T_4.

- T_4 satisfait la tolérance. On appelle ClusterizeRank(4).
- T_4 est ajouté à g_1. Tous ses voisins sont groupés. ClusterizeRank(4) termine.
- ClusterizeRank(3) termine puis ClusterizeRank(2) aussi, finalisant le groupe g_1 qui contient alors T_2, T_3 et T_4.
- On revient alors dans le contexte de la boucle *Tantque* où l'on constate que toutes les tâches ont été groupées et l'algorithme de groupement des tâches s'arrête.

3.8 Découverte des clusters de machines

3.8.1 Objectif

Nous avons proposé dans la section précédente un algorithme de détection de groupes de tâches fortement dépendantes afin de les affecter à des machines à forte proximité. Cependant, vue la nature décentralisée de P2P-MPI, on ne dispose à priori d'aucune information sur la localité des machines. Il nous faut donc un moyen permettant d'identifier les groupes de machines fortement interconnectées i.e les mini-clusters de machines.

3.8.2 Représentation des interconnexions des hôtes

Pour pouvoir affecter de gros volumes de communications sur des liens à fort débit, il est nécessaire de connaître le graphe d'interconnexion des hôtes. Le graphe d'interconnexion est un graphe orienté complet dont les noeuds représentent les hôtes, et les arcs les routes entre les hôtes.

Ce graphe est représenté sous forme d'une matrice d'adjacence carrée M_p où une valeur $M_p[i][j]$ indique le débit de la route allant de l'hôte h_i vers h_j. Il est à noter que M_p n'est pas symétrique, i.e $M_p[i][j] \neq M_p[j][i]$ car la route allant de l'hôte $h_i \rightarrow h_j$ n'est pas forcémment la même que $h_j \rightarrow h_i$.

Les noeuds du graphe sont déterminés par un mécanisme de découverte exhaustif. Le point faible majeur de la méthode de découverte de P2P-MPI est qu'elle a une vision très limitée des ressources disponibles sur la grille. En effet le MPD ne demande de découvrir que le nombre requis de machines pour exécuter les tâches parallèles. Il délègue ainsi la sélection des machines qui exécuteront l'application au Rendezvous peer qui renvoie très souvent la même liste.
Afin de pouvoir bénéficier des meilleures machines il nous faut découvrir le plus de machines possibles. On interroge donc le Rendezvous peer pour tous les MPD advertisements qu'il connaît. Comme P2P-MPI a une architecture décentralisée, on ne peut pas déterminer si l'on a découvert toutes les machines afin d'arrêter le processus de découverte. On termine donc ce processus à l'écoulement d'un "timeout" qu'on se fixe (moins d'une dizaine de secondes).

La détermination des poids des arcs formant la matrice M_p peut quant à elle être faite de deux manières :

- **Par un mécanisme de découverte automatique :** un mécanisme analyse systématiquement les connexions qu'il est possible d'établir entre toute paire d'hôtes et évalue leurs débits. Des travaux sur le sujet ont montré la difficulté de cette tâche [16], qui dépasse largement le cadre de ce travail.
- **Spécifiée par l'utilisateur :** ce qui lui permet de définir ses préférences et donc d'influer sur l'algorithme d'allocation.

Nous optons pour la deuxième manière : la matrice M_p est spécifiée par l'utilisateur. Dans ce cas, chaque MPD doit disposer d'un fichier indiquant la liste des hôtes connnus ainsi que leur matrice d'adjacence. Après le processus de découverte du maximum de pairs possibles, la matrice M_p est simplifiée pour ne contenir que les hôtes connus ayant été découvert. Ceci permet de simplifier les données en entrée de l'algorithme de clusterisation que nous allons voir dans la suite et donc d'en réduire la complexité.

3.8.3 Algorithme de groupement des processeurs

L'algorithme de recherche de clusters prend en entrée la matrice d'adjacence des hôtes M_p qui a été simplifiée après le processus de découverte pour ne contenir que les hôtes connus et disponibles sur la grille. Une définition formelle de l'algorithme se trouve dans la figure suivante.

```
/* Ayant une matrice d'adjacence carrée M_p(N,N) où N est le
   nombre d'hôtes connus découverts                          */
Entrées: M_p
Soit Clustered = {};
/* l'ensemble contenant les indices des processeurs
   clusterisés                                               */
Soit C = {} ;
/* l'ensemble des clusters de machines                       */
Soit tolerance = 0.1;
Soit maxConn ;
/* maxConn indique la connectivité du cluster actuel         */
tant que Taille(Clustered) < N faire
    /* on cherche un pivot de groupement                     */
    Soit ligne_pivot = min({i ∉ Clustered|∃(i,j) ∈ ℕ² tq j ∉
    Clustered et ∀(x,y) ∈ ℕ², M_p[i][j] ≥ M_p[x][y]}) ;
    on ajoute un nouveau Cluster à C ;
    maxConn = M_p[ligne_pivot][j] ;
    /* on appelle ensuite la fonction récursive pour le pivot
       */
    ClusterizeHost(ligne_pivot);
fin

/* Fonction ClusterizeHost(pivot) récursive de groupement des
   machines                                                  */
Entrées: indice de ligne du pivot p
On ajoute p au cluster actuel C[Taille(C)-1];
/* on definit l'ensemble V_p des voisins de p                */
Soit V_p = {v_i|M_p[p][v_i] ≠ 0} ;
/* On cherche maintenant la liste des voisins symetriques V_sp
   de p                                                      */
Soit V_sp = {};
pour tous les v_i de V_p tq v_i ∉ Clustered faire
    si M_p[p][v_i] * (1 − tolerance) < M_p[v_i][p] < M_p[p][v_i] * (1 + tolerance)
    alors
        /* on ajoute v_i aux voisins symétriques             */
        V_sp.add(v_i);
    fin
fin
V_sp est ensuite ordonné selon des valeurs décroissantes de M_p[p][v_i];
pour tous les v_j ∈ V_sp faire
    si maxConn * (1 − tolerance) < M_p[p][v_j] alors
        ClusterizeHost(v_j);
    fin
fin
```

Algorithm 2: Algorithme de Clusterisation des machines

Etant donné une matrice d'adjacence de machines M_p définie comme suit : M_p est une matrice carrée (N, N) où N est le nombre d'hôtes connus et découverts (présents). La valeur $M_p(i, j)$ indique le débit de la route entre les hôtes h_i et h_j.

L'algorithme commence par chercher un pivot de groupement. Le pivot est l'indice de la première machine disposant du lien à plus haut débit ; il correspond donc à l'indice de la première ligne contenant la plus grande valeur dans la matrice ; par exemple si $M_p[i][j] = ValeurMax(M_p)$ et $i < j$ alors le pivot est i.

On crée alors un nouveau cluster de machines auquel on attribue une valeur de connectivité référence : $maxConn$. Cette valeur va être utilisée dans l'ajout d'autres hôtes au cluster. Après avoir identifié le premier pivot *pivot*, et créé le premier groupe $C[0]$, on appelle la fonction récursive $clusterizeHost(pivot)$.

Cette fonction séléctionne d'abord les voisins symétriques du pivot. Un voisin v_i est dit *symétrique* si le débit de la route du pivot vers v_i est équivalent, selon un niveau de tolérance, à celui de la route de v_i vers le pivot. On parcourt donc la liste des voisins non clusterisés du pivot en testant s'ils vérifient la condition suivante :

$$M_p[p][v_i] * (1 - tolerance) < M_p[v_i][p] < M_p[p][v_i] * (1 + tolerance)$$

On ordonne ensuite les voisins symétriques du pivot selon un ordre décroissant du débit de leur lien avec le pivot. On évalue ensuite les voisins symétriques v_i dans l'ordre afin de tester s'ils vérifient la condition suivante :

$$maxConn * (1 - tolerance) < M_p[p][v_i]$$

où *tolerance* est un paramètre fixé au début de l'algorithme, compris entre 0 et 1 et permettant de définir un seuil de similarité pour les débits des routes. Si v_i vérifie cette condition on le considère comme un pivot et on appelle la fonction récursive $clusterizeHost(v_i)$. Lorsque toutes les fonctions récursives terminent, les machines interconnectées par des débits similaires et nettement supérieurs au restes des hôtes forment le cluster courant $C[Taille(C) - 1]$. A ce stade, si le nombre des hôtes groupés est inférieur au nombre total d'hôtes N, on recommence là même manoeuvre à savoir chercher un nouveau pivot et créer un nouveau cluster. Sinon (si le nombre d'hôtes 'clusterisés' est égal à N), la découverte des clusters de machines est alors terminée.
Une fois la détection des clusters de machines terminée il reste à déterminer les débits d'interconnexion entre clusters. Cette information de proximité des clusters s'avèrera très utile lors du processus d'allocation, notamment dans le cas où l'on ne dispose que de clusters à petits nombres de machines. Pour cela on emploie la méthode suivante :

```
/* Ayant un vecteur V_c contenant la liste des clusters formés */
Entrées: V_c
Sorties: ClusterAdjacence
Soit ClusterAdjacence = une matrice carrée (n, n) où n est le nombre de clusters ;
/* la matrice d'adjacence des clusters                          */
pour tous les c_i ∈ V_c faire
    pour tous les c_j ∈ V_c et c_j ≠ c_i faire
        Soit t={};
        /* le vecteur contenant tous les débits allant de c_i
           vers c_j.                                             */
        pour tous les host_l ∈ c_i faire
            pour tous les host_k ∈ c_j faire
                t.add(M_p[host_l][host_k]);
            fin
        fin
        On ordonne ensuite t selon un ordre croissant;
        ClusterAdjacence[c_i][c_j] = t[Taille(t)/2];
        /* la valeur au milieu du tableau t correspond à la
           valeur médiane                                        */
    fin
fin
```

Algorithm 3: Algorithme de calcul de l'adjacnce des clusters de machines

Pour déterminer une valeur de débit reliant un cluster c_i à c_j nous choisissons la valeur médiane de tous les liens allant des hôtes de c_i vers ceux de c_j. La valeur médiane est calculée en ordonnant toutes les valeurs de débits de c_i vers cj dans un tableau (l'ordre peut être croissant ou décroissant). La valeur médiane se trouve alors au mileu du tableau ordonné.

3.8.4 Complexité de l'algortihme de groupement des processeurs

De même que pour l'algorithme de groupement des tâches, la complexité de l'algorithme de clusterisation sera évaluée en fonction du nombre d'accès aux valeurs de la matrice M_p.

Pour déterminer le pivot de clusterisation il faut trouver la valeur maximum de la matrice. Pour cela on doit parcourir toutes les valeurs de la matrice M_p car la matrice M_p contrairement à M_t n'est pas symétrique. Ce processus a donc une complexité de N^2 accè où N est le nombre de lignes de la matrice.

L'appel à la fonction ClusterizeHost(*pivot*) nécessite $2 * (N - 1)$ accès pour trouver les voisins symétriques du pivot. Ensuite si n est le nombre des voisins symétriques, il faut $\frac{n*(n+1)}{2}$ accès pour les ordonner. Le pire cas pour la fonction ClusterizeHost est atteint lorsque tous les voisins du pivot sont symétriques i.e

$n = N - 1$. Dans ce cas la fonction ClusterizeHost(pivot) a une complexité de $\frac{N^2-N}{2} + 2 * (N-1) = \frac{N^2+3N-2}{2}$ accès.

Le processus de recherche d'un cluster de machines dans une matrice carrée comprenant N lignes nécessite alors $\frac{N^2+3N-2}{2} + N^2 = \frac{3N^2+3N-2}{2}$ accès. Le pire cas pour l'algorithme de clusterisation est le même que pour celui du groupement des tâches : lorsque à chaque appel de ClusterizeHosts(), seulement deux machines sont groupées. Dans ce cas le processus de formation d'un cluster doit se répéter pour une matrice d'adjacence carrée comprenant $N - 2$ lignes, puis $N - 4$, etc..

La complexité de l'algorithme de groupement s'exprime donc comme ceci :

$$\sum_{k=0}^{\frac{N}{2}} \frac{3 * (N - 2 * k)^2 + 3 * (N - 2 * k) - 2}{2} \text{ si N est impair}$$

$$et \sum_{k=0}^{\frac{N}{2}-1} \frac{3 * (N - 2 * k)^2 + 3 * (N - 2 * k) - 2}{2} \text{ si N est pair}$$

Cette somme a au plus $\frac{N}{2}$ termes, la complexité de l'algorithme de clusterisation des hôtes est donc aussi en $O(N^3)$.

3.8.5 Limites

L'efficacité de la détection des clusters de machines sera montrée dans la section de validation de la méthode d'allocation. Nous montrons dans la figure 3.5 un exemple où la clusterisation peut donner de mauvais résultats. Ce cas peut

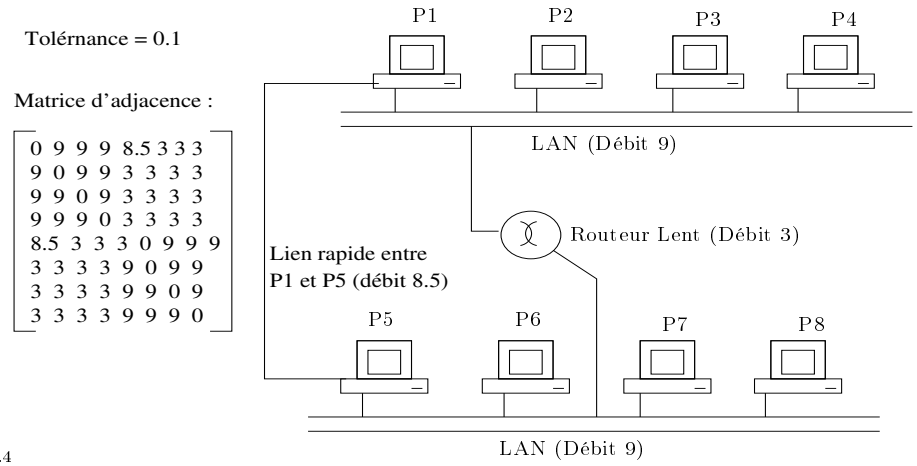


FIG. 3.5 – Exemple d'un cas défavorable de clusterisation

être dû à un "bruit" (fausse valeur) dans la matrice d'adjacence des machines ou encore un cas particulier d'interconnexion que l'on considère comme rare. Dans ce cas lors de la sélection de P_1 comme premier pivot de clusterisation, P_5 est un voisin vers qui le lien satisfait la condition de tolérance et il sera don considéré comme pivot à son tour. Il s'en suit que toutes les machines seront considérées comme membres d'un unique cluster bien que la majorité d'entre elles aient de mauvaises interconnexions. Une solution à ce problème serait d'effectuer des prétraitements sur la matrice d'adjacence des machines afin d'en enlever les valeurs statistiquement insignificatives et qui peuvent induire en erreur l'algorithme de clusterisation.

3.9 Algorithme d'allocation

Nous avons dans les deux sections précédentes proposé deux algortihmes dont le but est de découvrir les dépendances fortes entre tâches ainsi que les liens forts entre machines. Ces informations sont les entrées de notre algorithme d'allocation.

Nous avons vu précédemment que lors d'une allocation des tâches d'une application parallèle, deux facteurs influent sur les performances de l'application :

- **L'affectation des tâches aux machines :** des tâches nécessitant beaucoup de calcul affectées à des machines à faibles performances induisent une lenteur d'exécution en plus du ralentissement des tâches ayant une dépendance de données avec elles.
- **L'affectation des communications aux liens réseaux :** des communications volumineuses s'effectuant sur des liens à faibles débits induisent des temps d'attente pouvant bloquer plusieures tâches.

Nous avons aussi remarqué que l'optimisation de la qualité de l'un de ces deux facteurs en priorité pouvait détériorer gravement la qualité de l'autre et donc amener à de mauvaises performances de l'application. Nous essayons dans notre algorithme de concilier au maximum ces deux facteurs afin de garantir de bonnes performances dans une majorité des cas.

L'idée que nous proposons est d'allouer, dans l'ordre, les groupes de tâches les plus demandeurs en calcul aux clusters découverts les plus puissants sur la grille. On pourrait alors penser que cela revient au même que chercher à optimiser en priorité le premier facteur cité ci-dessus en risquant de détériorer la qualité du deuxième. Or ce n'est pas le cas. En effet en exécutant un groupe de tâches fortement dépendantes sur les machines d'un cluster, on est assuré que les communications volumineuses se feront sur des machines interconnectés par des liens homogènes et généralement à fort débit.

La sélection du cluster disposant des meilleures performances de calcul ne diminue donc pas la qualité des communications. Cette stratégie *max-max* (allocation du groupe de tâche de poids maximum au cluster de processeurs le plus puissant) peut encore être utilisée sans risque de détérioration des performances lors de l'allocation *intra-cluster* i.e l'allocation des tâches d'un groupe aux machines d'un cluster. En effet, comme les liens d'interconnexion des machines

sont homogènes, on peut se permettre d'affecter, dans l'ordre, les tâches les plus nécessiteuses en calcul aux machines les plus puissantes sans risquer de détériorer la qualité des communications. Une description formelle de l'algorithme se trouve dans la figure suivante :

```
/* Ayant la liste des Groupe de tâches G, et la liste des
   cluster de machines C                                    */
Entrées: G, C
/* on remplit un vecteur Mapping indiquant pour chaque tache
   le processeur auquel elle est affectée i.e indice dans le
   vecteur = indice tâche et valeur = indice de la machine
   */
Mapping ← [ ] ;
MappedGroups ← [ ] ;
/* MappedGroups est liste des groupes affectés à des clusters
   */
groupMapp ← [ ] ;
/* dans groupMapp on mémorise l'allocation des groupes aux
   tâches l'indice est le numéro du groupe la valeur est le
   numéro du cluster                                        */
/* on affecte la tâche 0 à la machine locale                */
Mapping[0] = id_machineLocale;
/* On doit donc ensuite affecter le groupe contenant la tâche
   0 au cluster contenant la machine locale                 */
Soit g_0 le groupe de tâches contenant la tâche 0;
Soit c_local le cluster contenant la machine locale;
/* On appelle la fonction d'allocation d'un groupe à un
   cluster                                                  */
MappGroupToCluster(g_0, c_local );
/* puis on ajoute le groupe 0 dans la liste de groupes
   affectés                                                 */
MappedGroups.add(g_0);
tant que Taille(MappedGroups) < Taille(G) faire
    Soit g_max ∈ G le groupe le plus nécessiteux en calcul;
    si ∃c_l tq NbTaches(g_max) < NbMachines(c_l) et c_l contient d'autres
    groupes de taches alors
        MappGroupToCluster(g_max, c_l);
    sinon
        Soit c_max ∈ C le cluster le plus puissant;
        MappGroupToCluster(g_max, c_max);
    fin
    MappedGroups.add(g_max);
fin
```

Algorithm 4: Algorithme d'allocation des groupes de tâches au clusters de machines

Nous remarquons que l'algorithme alloue d'office la tâche ayant pour identifiant 0 à la machine locale. Ceci est fait pour respecter le fonctionnement normal de P2P-MPI. En effet lorsqu'un utilisateur lance un Job sur une grille P2P-MPI, le processus ayant pour rang 0 s'exécute toujours sur sa machine. Ceci nous impose donc une forte contrainte dont on réduit l'impact en allouant le groupe de tâches contenant T_0 au cluster de machines contenant la machine locale.

La suite de l'algorithme est alors simple. Tant que tous les groupes de tâches n'ont pas été alloués à des clusters, on cherche d'abord le groupe de tâches le plus demandeur en calcul. On cherche ensuite un cluster contenant d'autres groupes et disposant d'assez de machines pour contenir le groupe maximal. A défaut on alloue le groupe maximal au plus puissant cluster disponible.

Nous allons détailler dans la suite la fonction MappGroupToCluster.

Nous avons fixé précédemment des critères stricts pour notre méthode d'allocation. L'un de ces critères impose qu'une machine n'exécute au maximum qu'une seule tâche. Lors de l'allocation des tâches d'un groupe de tâches à un cluster de machines, deux cas de figures peuvent apparaître :

- Le nombre de tâches du groupe est inférieur au nombre de machines du cluster : dans ce cas le critère cité ci-dessus est respecté. Il suffit d'adopter une stratégie d'allocation des tâches aux machines, de type max-max, et de mettre à jour ensuite les informations concernant le cluster.
- Le nombre de tâches est supérieur au nombre de machines : dans ce cas nous ne pouvons pas affecter toutes les tâches du groupe au cluster choisi sans enfreindre le critère d'unicité de tâche par machine. Si n_t est le nombre de tâches du groupe et n_m le nombre de machines du cluster, on doit alors affecter n_m tâches du groupe au cluster courant puis les $n_t - n_m$ tâches restantes au cluster le plus proche.

Dans le deuxième cas de figure (le nombre de tâches du groupe est supérieur au nombre de machines du cluster) le choix des tâches qui s'exécuteront sur le cluster doit se faire de manière judicieuse. En effet, le fait que deux tâches fassent partie du même groupe, ne signifie pas forcémment qu'elles communiquent ensemble. Prenons par exemple le cas du groupe G dans la figure 3.6 que l'on veut allouer sur le cluster C.

Si l'on choisit d'affecter les tâches T_0, T_2 et T_4 aux machines du cluster C, on effectue toutes les communications du groupe sur des liens externes au clusters et on dégrade donc les performances de l'application. Il faut donc une stratégie de division du groupe de tâches qui tienne compte des communications des tâches. C'est le rôle de l'algorithme de division des groupes de tâches décrit dans la figure suivante.

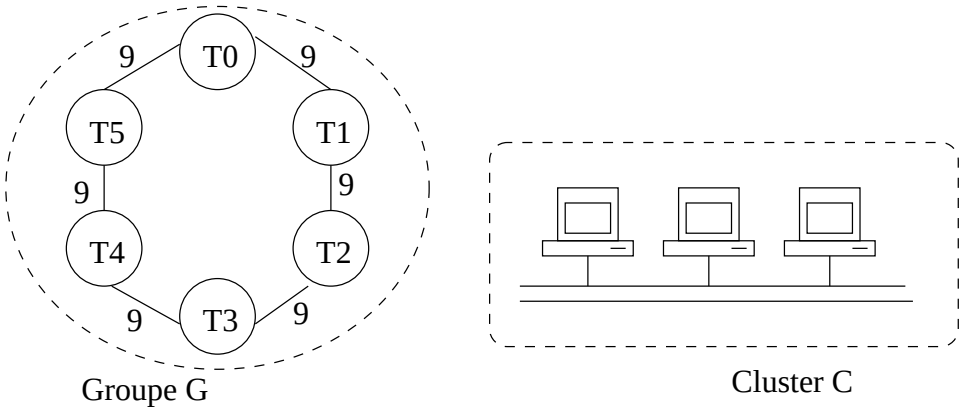


FIG. 3.6 – Exemple de division de groupe

```
/* Ayant un groupe de tâches : G et la taille du sous groupe
   souhaitée : t                                              */
Entrées: G : Groupe, t : Entier
Sorties: Agglomerated : Vecteur
/* contient les indices des tâches agglomérées                */
Soit M_{t_g} la matrice d'adjacence des tâches M_t réduite au groupe G;
/* cette matrice ne contient que les adjacences entres les
   taches de G                                                */
/* On commence par rechercher un pivot d'agglomeration        */
Soit ligne_pivot = min({i ∉ Agglomerated|∃(i,j) ∈ ℕ² tq j ∉
Agglomerated et ∀(x,y) ∈ ℕ², M_{t_g}[i][j] ≥ M_{t_g}[x][y]}) ;
/* et on ajoute le pivot à l'agglomération                    */
Agglomerated.add(ligne_pivot);
tant que Taille(Agglomerated)< t faire
    /* on cherche la tâche la plus communiquante avec les
       tâches agglomérées                                     */
    Soit t_c tq ∑_{i∈Agglomerated} M_t[t_c][i] est maximale;
    On ajoute t_c à Agglomerated;
fin
```

Algorithm 5: Algorithme de division d'un groupe de tâches

Dans le cas de la figure 3.6, l'algorithme considère T_0 comme pivot d'agglomération car elle échange un volume maximal de données(9) tout en ayant le plus petit indice. T_1 est le premier t_c choisi car il communique un volume de 9 avec T_0. T_2 est le deuxième t_c choisi car il communique de 9 avec T_1. On obtient donc un sous groupe de tâches fortement interconnectée (T_0, T_1 et T_2) afin de les affecter aux machines du cluster C.

On peut remarquer dans la figure 3.7 que cet algorithme d'agglomération respecte la régularité des graphes d'applications en les découpants. En effet on

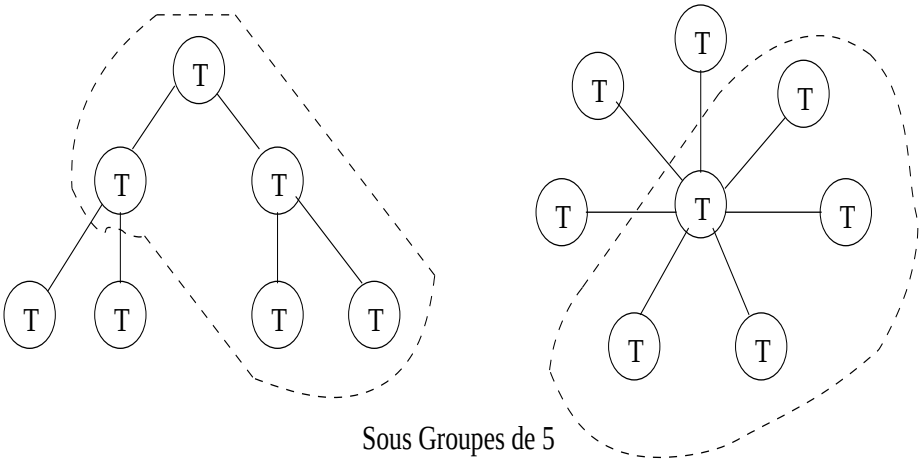


FIG. 3.7 – régularité de l'agglomération

remarque que les sous-graphes des graphes réguliers sont souvent du même type.

La fonction mappGroupToCluster peut s'écrire comme ceci :

```
/* Fonction mappGroupToCluster                                    */
Entrées: g : entier, c : entier
/* g est l'identifiant du groupe de tâches ; c est
   l'identifiant du cluster                                       */
/* on teste si le groupe peut être entièrement contenu dans
   le cluster                                                     */
si NbTaches(g) < NbMachines(c) alors
    groupMapp[g]=c;
    /* on agit par effet de bord sur le vecteur groupMapp        */
sinon
    Soit sousGroupe ← Division(g, NbMachines(c));
    groupMapp[sousGroupe] ← c;
    Soit c' le cluster disponible le plus proche de c;
    Soit resteGroupe ← g privé de sousGroupe ;
    mappGroupToCluster(resteGroupe, c');
fin
```

Algorithm 6: Algorithme d'allocation d'un groupe de tâches à un cluster

Il est important de noter qu'on n'affecte pas les tâches directement aux machines. On se contente dans cette fonction de répertorier l'affectation des groupes aux tâches. Le but est d'éviter de "gaspiller" les meilleures machines. Prenons par exemple le cas d'un cluster c de 6 machines et deux groupes de tâches g_1 et g_2 de 3 tâches chacun. Si lors de l'appel mappGroupToCluster(g_1, c) on alloue les tâches directement selon le stratégie max-max, g_1 s'exécuterait

sur les 3 meilleures machines de c et ne laisserait pour g_2 que les trois pires. Pour éviter cela on se contente de référencer quels groupes sont alloués à quels clusters. Une fois que l'allocation des groupes aux clusters est terminée on applique la stratégie max-max en considérant chaque cluster indépendamment : on recherche itérativement sa machine la plus puissante à laquelle on affecte la tâche la plus nécessiteuse en calcul parmi celles de tous les groupes qui sont affectés au cluster courant.

3.10 Complexité de l'algorithme d'allocation des groupes de tâches aux clusters

Soit N_c est le nombre de clusters de machines découverts par l'algorithme de clusterisation et N_g est le nombre de groupes de tâches identifiés par l'algorithme de groupement des tâches. La procédure de sélection du groupe de tâches le plus lourd en calcul prend N_g puis $N_g - 1$ puis $N_g - 2$ etc.. accès à la liste des groupes. Ce qui donne en total $\frac{N_g*(N_g+1)}{2}$ accès à cette liste. De même, la sélection du cluster le pluissant nécessite $\frac{N_c*(N_c+1)}{2}$ accès à la liste des clusters pour la totalité de l'algorithme. Le pire cas lors de l'allocation de N tâches parallèles sur une grille comprenant M machines se produit lorsque $N_g = N$ et $N_c = M$ i.e chaque tâche est considérée comme un groupe et chaque machine comme un cluster. La complexité dans ce cas est alors en $O(max(N, M)^2)$. Comme nous avons imposé une contrainte d'unicité de tâche par machine, on a donc la relation suivante $M \geq N$. on a donc une complexité en $O(M^2)$ de l'algorithme d'allocation des groupes de tâches aux clusters de machines.

3.11 Conclusion

Dans l'ensemble des travaux présentés dans le chapitre 2, le travail qui s'apparente le plus au nôtre est [6] qui est décrit dans la section 2.3.6. L'inconvénient majeur de cette méthode d'allocation est qu'elle ne respecte pas le critère d'unicité de tâche par machine que l'on s'est fixé pour l'allocation de tâches sur la grille P2P-MPI. En effet lors du plaquage de l'arbre de clusterisation du graphe des tâches sur l'arbre de clusterisation des processeurs, un noeud non feuille de l'arbre des tâches (représentant un groupe de tâches donc) peut être alloué à un noeud feuille de l'arbre des processeurs (représentant une machine). L'algorithme d'allocation récursif n'est donc pas adapté pour notre cas.

De plus, dans le cadre de notre objectif pour l'allocation des tâches, nous ne voyons pas l'utilité d'une clusterisation hiérarchique des graphes. En effet, le résultat d'une première clusterisation (une passe de l'algorithme [6]) est suffisant pour identifier les groupes de tâches et de machines à forte affinité. Ces groupes peuvent être suffisamment volumineux pour qu'on ait besoin dans notre application de les découper (lors de la recherche d'un sous-groupe de tâche par exemple). La réitération de l'algorithme de clusterisation sur ces groupes afin de former des groupes plus volumineux nous est donc inutile.

Finalement en éliminant la récursivité hiérarchique de la clusterisation ainsi que de l'allocation nous obtenons une méthode d'allocation à moindre complexité.

Chapitre 4

Exemples de Validation

4.1 Notre grille de tests

Les tests ont été effectués sur une grille composée de 21 machines réparties entre le campus de l'esplanade, et celui d'illkirch. Nous détaillons dans le tableau suivant la localité de ces machines ainsi que les notes de puissance qui leur sont attribuées.

Ressource	Lieu	nb Machines	Note puissance	id
marathon	Esplanade	1	0.5	A
dinadan	Illkirch (ICPS)	1	0.7	B
merlin	Illkirch (ICPS)	1	0.9	C
lattice	Illkirch (ICPS)	1	0.8	D
tintagel	Illkirch (ICPS)	1	0.5	E
icps-server	Illkirch (ICPS)	1	0.8	F
lancelot	Illkirch (ICPS)	1	1	G
mordred	Illkirch (ICPS)	1	0.9	H
gauvain	Illkirch (ICPS)	1	1	I
sekhmet	Illkirch (ICPS)	1	0.7	J
HPC (CECPV)	Illkirch	11	1	K1-K11

Nous disposons donc de deux clusters de machines : HPC et les machines du laboratoire ICPS, ainsi que d'une machine distante : marathon. Nous reflétons ces localités dans la matrice d'adjacence suivante :

A	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B	1	0	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
C	1	2	0	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
D	1	2	2	0	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
E	1	2	2	2	0	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
F	1	2	2	2	2	0	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
G	1	2	2	2	2	2	0	2	2	2	1	1	1	1	1	1	1	1	1	1	1
H	1	2	2	2	2	2	2	0	2	2	1	1	1	1	1	1	1	1	1	1	1
I	1	2	2	2	2	2	2	2	0	2	1	1	1	1	1	1	1	1	1	1	1
J	1	2	2	2	2	2	2	2	2	0	1	1	1	1	1	1	1	1	1	1	1
K1	1	1	1	1	1	1	1	1	1	1	0	2	2	2	2	2	2	2	2	2	2
K2	1	1	1	1	1	1	1	1	1	1	2	0	2	2	2	2	2	2	2	2	2
K1	1	1	1	1	1	1	1	1	1	1	2	2	0	2	2	2	2	2	2	2	2
K4	1	1	1	1	1	1	1	1	1	1	2	2	2	0	2	2	2	2	2	2	2
K5	1	1	1	1	1	1	1	1	1	1	2	2	2	2	0	2	2	2	2	2	2
K2	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	0	2	2	2	2	2
K7	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	0	2	2	2	2
K8	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	0	2	2	2
K9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	0	2	2
K10	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	0	2
K11	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	0

où l'on modélise grossièrement le réseau en distinguant 3 groupes :

- Les 11 machines HPC
- Les machines du laboratoire ICPS
- La machine *marathon* comme étant équidistante des machines HPC et ICPS.

Ces trois groupes seront identifiés comme clusters de machines par notre algortihme de clusterisation.

4.2 Applications types

Dans les tests de validation de notre méthode d'allocation nous avons choisi d'allouer des applications à graphes régulier qui sont souvent utilisées dans les applications parallèles de type MPI, à savoir :

- **Applications à barrières régulières :** Ce type d'applications est utilisé dans les calculs parallèles nécessitant une synchronisation fréquente des tâches comme par exemple les simulations numériques où on a plusieures générations ou étapes de temps qui commencent après une synchronisation de tous les processus. Les applications de ce type peuvent avoir un graphe de communication quelconque.
- **Applications de type maître esclaves :** Ce type d'applications parallèle est très fréquent car il permet de favoriser les machines les plus efficaces en leur allouant plus de tâches. En effet, dans les applications maître esclaves, un processus central joue le rôle de maître en allouant des tâches à d'autres processus dits esclaves. Dès qu'un esclave termine

la tâche qui lui a été assignée, il renvoie le résultat au maître qui lui assigne aussitôt une nouvelle tâche. Ainsi les esclaves les plus performants exécutent le plus de tâches. Les applications de type maître esclave ont souvent une topologie de graphe de communication en étoile dont le centre est le processus maître.

– **Application de type divide and conquer :** Ce type d'applications permet de résoudre des problèmes difficiles. En effet, les processus d'une telle application exécutent souvent les mêmes calculs sur des espaces de données différents afin de parcourir plus vite l'espace total des données. Ce type d'applications a souvent un graphe de communication en arbre.

4.3 Résultats

Dans nos expériences nous comparons les temps d'exécution de nos trois applications types avec l'allocation faite par la version originale de P2P-MPI et celle faite par notre méthode d'allocation.

4.3.1 Application à barrières de synchronisation

Dans ce test nous lançons une application à barrières de synchronisation comprenant 13 tâches, depuis la machine *dinadan*.

Allocation standard : Notons que si chaque exécution peut donner des allocations différentes, dans la pratique on retrouve très souvent les mêmes allocations d'une exécutuion à l'autre en raison d'un mécanisme de cache de *JXTA*. Le processus de découverte et d'allocation round-robin nous donne les allocations suivantes :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.118.171 : hpc-n31
[Master of Rank 2] : IP = 130.79.118.172 : hpc-n32
[Master of Rank 3] : IP = 130.79.118.173 : hpc-n33
[Master of Rank 4] : IP = 130.79.118.174 : hpc-n34
[Master of Rank 5] : IP = 130.79.118.175 : hpc-n35
[Master of Rank 6] : IP = 130.79.118.176 : hpc-n36
[Master of Rank 7] : IP = 130.79.118.177 : hpc-n37
[Master of Rank 8] : IP = 130.79.118.178 : hpc-n38
[Master of Rank 9] : IP = 130.79.118.179 : hpc-n39
[Master of Rank 10] : IP = 130.79.118.180 : hpc-n40
[Master of Rank 11] : IP = 130.79.118.181 : hpc-n41
[Master of Rank 12] : IP = 130.79.186.117 : marathon

on peut remarquer dans cette allocation que le processus de rang 0 est affecté à la machine dinadan depuis laquelle l'application est lancée. On remarque aussi le choix de marathon qui est peu judicieux car elle peut ralentir les synchronisations globales. Cependant vu le grand nombre de machines performantes de

0	70100704	70100704	70100704	70100704	70100704	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..
70100704	0	0	0	0	0	etc..

FIG. 4.1 – Matrice d'adjacence des taches

HPC choisies, cette allocation donne un bon temps d'exécution : 700.716 s

La figure 4.1 donne la matrice d'adjacence des tâches construite après réception des traces.

Avec notre méthode d'allocation : Notre algorithme de clusterisation des tâches détecte dans ce cas un seul groupe contenant toutes les tâches de l'application. L'allocation proposée est donc :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.192.153 : mordred
[Master of Rank 2] : IP = 130.79.192.149 : tintagel
[Master of Rank 3] : IP = 130.79.192.165 : icps-server
[Master of Rank 4] : IP = 130.79.118.174 : hpc-n34
[Master of Rank 5] : IP = 130.79.118.179 : hpc-n39
[Master of Rank 6] : IP = 130.79.118.175 : hpc-n35
[Master of Rank 7] : IP = 130.79.118.178 : hpc-n38
[Master of Rank 8] : IP = 130.79.118.171 : hpc-n31
[Master of Rank 9] : IP = 130.79.118.176 : hpc-n36
[Master of Rank 10] : IP = 130.79.118.173 : hpc-n33
[Master of Rank 11] : IP = 130.79.118.172 : hpc-n32
[Master of Rank 12] : IP = 130.79.118.177 : hpc-n31

Cette affectation est visualisée dans la figure 4.2 (en couleurs)

On remarque dans ce choix que certaines tâches faisant partie du groupe de la tâche 0 sont affectées à des machines du cluster de la machine *dinadan* lançant l'application. On remarque aussi que toutes les machines de ce cluster ne sont pas présentes dans l'allocation. Ceci est dû au mécanisme de découverte de *jxta* qui n'a pas réussi à trouver toutes les machines du laboratoire ICPS.

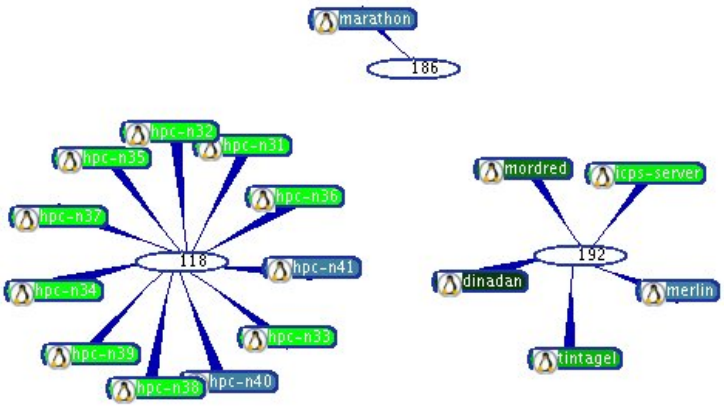


FIG. 4.2 – Allocation de l'application à barrières

Cependant cette allocation donne de meilleures performances que la précédente en réalisant un temps d'exécution de 623.109 s donc 77 secondes de moins. Ceci est certainement dû au fait qu'on ait ignoré, lors de l'allocation, la machine distante *marathon* qui ralentissait la synchronisation. La recherche de l'allocation n'a pris quant à elle que *243 ms* sur la machine *dinadan*.

4.3.2 Application de type maître-esclave

Dans ce test nous lançons une application de type maître-esclave comprenant 6 tâches, depuis la machine *dinadan*. Dans cette application le maître transmet aux esclaves un gros volume de données sur lequel ils doivent effectuer des calculs et lui renvoyer le résultat.

Allocation standard : Le processus de découverte et d'allocation round-robin nous donne les allocations suivantes :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.118.171 : hpc-n31
[Master of Rank 2] : IP = 130.79.118.172 : hpc-n32
[Master of Rank 3] : IP = 130.79.118.173 : hpc-n33
[Master of Rank 4] : IP = 130.79.118.174 : hpc-n34
[Master of Rank 5] : IP = 130.79.192.151 : lattice

Nous remarquons dans cette allocation que 4 des processus esclaves se trouvent affectés sur un autre cluster que celui du maître. Ceci suppose donc des délais dans les transmissions des gros volumes de données entre le processus maître s'exécutant sur *dinadan* et ses 4 esclaves s'exécutant sur des noeuds de *HPC*. Cette allocation donne un temps d'exécution de 47.083 s

La matrice d'adjacence de cette application est la suivante :

0	8000163	31400631	15000303	22100445	24000483
8000163	0	0	0	0	0
31400631	0	0	0	0	0
15000303	0	0	0	0	0
22100445	0	0	0	0	0
24000483	0	0	0	0	0

FIG. 4.3 – Matrice d'adjacence des taches de l'application maître-esclave

Avec notre méthode d'allocation : Notre algorithme de clusterisation des tâches détecte dans ce cas 4 groupes de tâches $G_1 = \{T_0, T_5, T_3\}$, $G_2 = \{T_1\}$, $G_3 = \{T_2\}$ et $G_4 = \{T_4\}$.

L'allocation proposée est :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.192.149 : tintagel
[Master of Rank 2] : IP = 130.79.192.151 : lattice
[Master of Rank 3] : IP = 130.79.192.165 : icps-server
[Master of Rank 4] : IP = 130.79.192.153 : mordred
[Master of Rank 5] : IP = 130.79.192.160 : merlin

On remarque dans cette allocation l'intérêt de la découverte des clusters de machines. En effet dans ce cas le cluster formé des machines du laboratoire ICPS peuvent contenir totalement l'application parallèle. Notre méthode d'allocation favorise donc la localité spatiale des tâches en les allouant toutes à des machines de l'ICPS. Ceci donne un temps d'exécution de 31.654 s soit 15.429 secondes de moins qu'avec une allocation standard. Le temps de recherche de l'allocation n'a pris quant à lui que 71 ms sur la machine *dinadan*.

4.3.3 Application de type divide and conquer

Dans ce test nous lançons une application de type divide and conquer comprenant 15 tâches, depuis la machine *dinadan*. Le graphe de communication des tâches de l'application est structuré en arbre comme le montre la Fig 4.4.

Allocation standard : Le processus de découverte et d'allocation round-robin nous donne les allocations suivantes :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.118.171 : hpc-31
[Master of Rank 2] : IP = 130.79.118.172 : hpc-32
[Master of Rank 3] : IP = 130.79.118.173 : hpc-33
[Master of Rank 4] : IP = 130.79.118.175 : hpc-35
[Master of Rank 5] : IP = 130.79.118.176 : hpc-36
[Master of Rank 6] : IP = 130.79.118.180 : hpc-40

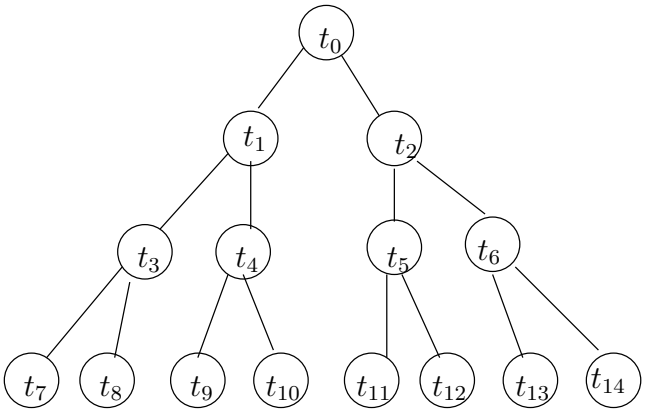


FIG. 4.4 – application divide and conquer

[Master of Rank 7] : IP = 130.79.118.181 : hpc-41
[Master of Rank 8] : IP = 130.79.186.117 : marathon
[Master of Rank 9] : IP = 130.79.192.160 : merlin
[Master of Rank 10] : IP = 130.79.192.145 : dinadan
[Master of Rank 11] : IP = 130.79.118.171 : hpc-31
[Master of Rank 12] : IP = 130.79.118.172 : hpc-32
[Master of Rank 13] : IP = 130.79.118.173 : hpc-33
[Master of Rank 14] : IP = 130.79.118.175 : hpc-35

On remarque dans cette allocation que la régularité du graphe n'a pas été respectée lors de l'allocation des tâches aux machines, ce qui implique qu'on effectue beaucoup de communications volumineuses sur des liens inter-clusters. Cette allocation donne un temps d'exécution de 21.032 s

Avec notre méthode d'allocation : Notre algorithme de clusterisation des tâches reconnait l'arbre entier des tâches comme étant un seul groupe. Cependant ne disposant pas d'assez de machines sur aucun des 3 clusters découverts pour affecter la totalité du groupe à un seul cluster, on doit diviser le groupe. En respectant la régularité de l'arbre lors de la division i.e en regroupant les tâches en sous-arbres on minimise les communications sur des liens inter-clusters. Notre algorithme donne l'allocation suivante :
[Master of Rank 0] : IP = 130.79.192.145 : dinadan
[Master of Rank 1] : IP = 130.79.192.160 : merlin
[Master of Rank 2] : IP = 130.79.118.175 : hpc-35
[Master of Rank 3] : IP = 130.79.192.165 : icps-server
[Master of Rank 4] : IP = 130.79.192.153 : mordred
[Master of Rank 5] : IP = 130.79.118.171 : hpc-31
[Master of Rank 6] : IP = 130.79.118.172 : hpc-32
[Master of Rank 7] : IP = 130.79.192.149 : tintagel

[Master of Rank 8] : IP = 130.79.192.151 : lattice
[Master of Rank 9] : IP = 130.79.118.178 : hpc-38
[Master of Rank 10] : IP = 130.79.118.179 : hpc-39
[Master of Rank 11] : IP = 130.79.118.173 : hpc-33
[Master of Rank 12] : IP = 130.79.118.176 : hpc-36
[Master of Rank 13] : IP = 130.79.118.177 : hpc-37
[Master of Rank 14] : IP = 130.79.118.174 : hpc-34

Cette allocation a été inhibée par le mécanisme de découverte de $JXTA$ qui n'a pu trouver toutes les machines du cluster de l'ICPS. Cette allocation donne un temps d'exécution de 16.408 s soit une amélioration de 4.624 s. La recherche de l'allocation a pris quant à elle 209 ms sur *dinadan*.

Chapitre 5

Conclusion

Notre but dans le cadre de ce mémoire était de proposer ainsi que d'implémenter une méthode d'allocation des tâches d'une application parallèle sur une grille de calcul P2P-MPI. Cette grille étant de nature fortement hétérogène, la sélection des ressources adéquates permettant d'obtenir des temps d'exécutions minimaux est d'autant plus difficile.

Nous avons proposé une méthode d'allocation à base de clusterisation des graphes de communications des applications ainsi que celle du graphe d'interconnexion des ressources formant la grille. La clusterisation du graphe d'application permet d'identifier les groupes de tâches fortement communiquantes. La clusterisation du graphe d'interconnexion des ressources permet quant à elle d'identifier des groupes de ressources à forte localité spatiale (forts débits d'interconnexion), comparables à des mini-clusters. L'idée est ensuite d'allouer, dans l'ordre, les groupes de tâches les plus demandeurs en calcul aux clusters découverts les plus puissants sur la grille. Ceci nous permet de sélectionner les machines les plus performantes d'un cluster en ayant comme garantie que les communications volumineuses entre les tâches s'effetueront sur des liens à forts débits. Nous avons implémenté cette méthode d'allocation dans une distribution de P2P-MPI permettant de collecter des traces d'exécution afin de construire le graphe de l'application parallèle. Nous avons ensuite testé cette méthode sur des applications parallèles types (fréquentes), que nous avons lancées sur une grille formée des machines disponibles au laboratoire. Les résultats montrent une amélioration des performances par rapport à la méthode d'allocation standard de P2P-MPI. La poursuite de ce travail se fera par de plus amples tests sur des configurations plus diverses que celle utilisée dans le cadre de ce mémoire. En effet notre grille de test (ICPS, HPC) ainsi que l'outil Grid5000 formé de noeuds fortement homogènes et interconnectés par des réseaux très performants, n'offrent pas un bon environnement d'expérimentation pour une grille hétérogène. Cependant avec des outils tels que Grid eXplorer[1] premettant d'imposer des latences au niveau des routeurs on encore l'outil *Netem*[3] permettant d'imposer des latences au niveau des hôtes, de meilleurs tests peuvent être conduits.

Bibliographie

[1] Grid explorer : http ://www.lri.fr/ fci/gdx/.

[2] Jxta project : http ://jxta.org.

[3] Netem : http ://linux-net.osdl.org/index.php/netem.

[4] Olivier Beaumont, Vincent Boudet, and Yves Robert. A realistic modern and efficient heuristic for scheduling with heterogeneous processors. *Technical Report 2001-37 ENS-Lyon - INRIA*, September, 2001.

[5] S.H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transactions on Computers*, 37 :48–57, 1988.

[6] N.S. Nikolaou Bowen and A. C.N. Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *Computers, IEEE Transactions on*, 41 :257–273, 1992.

[7] F.W. Burton, G.P. McKeown, and V.J. Rayward-Smith. On process assignment in parallel computing. *Information Processing Letters*, 29 :31,34, 1988.

[8] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. Mpj : Mpi-like message passing for java. *Concurrency : Practice and Experience*, 12(11), September 2000.

[9] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, June :50–56, 1982.

[10] Stéphane Genaud and Choopan Rattanapoka. A peer-to-peer framework for robust execution of message passing parallel programs. In B. Di Martino et al., editor, *EuroPVM/MPI 2005*, volume 3666 of *LNCS*, pages 276–284. Springer-Verlag, September 2005.

[11] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Performance prediction using an application-oriented mapping tool. *pdp*, 00 :184, 2004.

[12] J.-J Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput*, 18(2) :244,257, April 1989.

[13] E.G Coffman Jr. Computer and job shop scheduling theory. 1976.

[14] Sang Cheol Kim and Sunggu Lee. Push-pull : Guided search dag scheduling for heterogeneous clusters. *icpp*, 00 :603–610, 2005.

[15] S.-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Comput.*, 36(4) :433–442, 1987.

[16] Arnaud Legrand, Frédéric Mazoit, and Martin Quinson. An application-level network mapper. Technical Report 2002-09, Ecole Normale Superieure de Lyon, February 2002.

[17] Virginia M. Lo, Sanjay Rajpoadhye, Samik Gupta, David Keldsen, Moataz A.Mohamed, Bill Nitzberg, Jan Arne Telle, , and Xiaoxiong Zhong. Oregami : Tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming*, 20(3), 1991.

[18] MPI Forum. MPI : A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[19] Juan Manuel Ordu na, Federico Silla, and José Duato. On the development of a communication-aware task mapping technique. *Journal of Systems Architecture*, 2004.

[20] Choung Shik Park and Sang Bang Choi. Multiprocessor scheduling algorithm utilizing linear clustering of directed acyclic graphs. *icpads*, 00 :392, 1997.

[21] H. El Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9 :138,153, 1990.

[22] C. Roig, A. Ripoll, M.A Senar, F. Guirado, and E. Luque. A new model for static mapping of parallel applications with task and data parallelism. In *Proceedings of the International and Distributed Processing Synmposium (IPDPS'02)*, 2002.

[23] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. *ipdps*, 02 :111b, 2004.

[24] J. B. Weissman and A. S. Grimshaw. A framework for partitioning parallel computations in heterogeneous environments. *Concurrency : Practice and Experience*, 7(5) :455–478, 1995.