

UNIVERSITÉ DE FRANCHE-COMTÉ

CENTRE DE TÉLÉ-ENSEIGNEMENT UNIVERSITAIRE

**Master Informatique Avancé et Applications (I2A) option Recherche**

**Mémoire de fin d’Etudes**

|  |
| --- |
|  |

*Encadrant de Recherche*:

Laurent Philippe

*Etudiant :*

Parfait Pascal Wangun

**COMPARAISON D’HEURISTIQUES D’ORDONNANCEMENT DES TÂCHES SUR DES MACHINES PARALELLES**

I Ière Partie : Introduction générale 3

I.1 Contexte et justification 3

I.2 Problématique 4

I.3 C’est quoi l’ordonnancement des tâches? 4

I.4 Les algorithmes d’Ordonnancement de Types Liste 5

I.5 Méthodologie 6

II IIème Partie : Présentation et Description de quelques algorithmes d’ordonnancement 8

II.1 Généralités sur la problématique de l’ordonnancement des tâches sur des machines parallèles 8

II.2 ILHA (Iso-Level Heterogeneous Allocation) 9

*II.2.1* Minimum partial completion time static priority algorithm(PCT) 10

*II.2.2* Best imaginary level (BIL) 11

*II.2.3* Heterogeneous ealiest finish time (HEFT) and critical path on a processor(CPOP) algorithms 11

*II.2.4* Generalized dynamic level (GDL) algorithm 13

*II.2.5* The iso-level Heterogeneous Allocation(ILHA) algorithm 15

*II.2.6* Load balancing algorithm 15

II.3 [13]Scheduling Parallel Program Tasks onto Arbitrary target Machines 17

II.4 [7]Multi Strict Bound constraint algorithm 20

II.5 [1] : Dynamic priority scheduling heuristic for heterogeneous computing systems 21

II.6 [3] Improving Scheduling of Tasks in a Heterogeneous Environment 24

*II.6.1* Description de fonctionnement des algorithmes HEFT et CPOP 24

*II.6.2* Analyse des performances et résultats 25

*II.6.3* Comparaison entre l’algorithme HEFT et CPOP 26

II.7 Description de fonctionnement de l’algorithme PEFT 26

*II.7.1* Table de coût optimiste(OCT) 27

*II.7.2* Phase de priorisation des tâches 27

*II.7.3* Phase de sélection du processeur 27

*II.7.4* Description détaillée de l’algorithme PEFT 27

*II.7.5* Analyse des performances et résultats 28

II.8 Description de fonctionnement de l’algorithme PETS 28

*II.8.1* Phase de tri par niveau 29

*II.8.2* Phase de priorisation des tâches 29

*II.8.3* Phase de sélection du processeur 29

*II.8.4* Analyse des performances 30

II.9 Description de l’algorithme DLS 30

*II.9.1* Communication inter processeur 31

*II.9.2* Les niveaux dynamiques des processeurs hétérogènes 31

*II.9.3* L’ordonnancement HLEFT 32

*II.9.4* L’ordonnancement à niveau dynamique(DLS) 32

*II.9.5* Algorithme DLS 32

III IIIème Partie : Comparaison des heuristiques 33

III.1 Critères de de comparaison 33

III.2 Génération des graphes de taches 33

IV Conclusion 35

I. Référence 36

1. Ière Partie : Introduction générale
   1. Contexte et justification

Le développement fulgurant des réseaux informatiques a favorisé l’émergence des environnements de calcul distribués permettant l’exécution rapide des programmes informatiques. Ces environnements de calculs peuvent être répartis en deux catégories :

* Les environnements de calcul distribués homogènes ;
* et les environnements de calcul hétérogènes.

Contrairement aux environnements de calculs homogènes où les ressources de calcul sont toutes identiques, dans un environnement de calcul hétérogène les ressources de calcul présentent un certain nombre de différences. Ces différences sont entre autres, la puissance de calcul des processeurs, la capacité de la mémoire, la bande passante de transfert des informations, le nombre de processeurs etc... La problématique de l’ordonnancement des applications dans les systèmes de calcul hétérogène a jusqu’ici fait l’objet de nombreuses études dans la littérature, l’objectif étant de trouver une solution optimale à cette question d’ordonnancement. La recherche d’une solution optimale au problème d’ordonnancement des tâches sur des machines parallèles, a suscité le développement de nombreuses heuristiques dans le but de proposer à chaque fois une solution améliorée. En effet, il s’agit d’un problème NP-complet pour laquelle il n’existe pas à priori une solution efficace.

* 1. Problématique

Considérant la multitude des heuristiques développées dans la littérature au sujet de l’ordonnancement des taches sur des machines parallèles, quelles sont les meilleures? En d’autres termes, quelle est l’heuristique qui offre les meilleures performances dans le processus d’exécution des tâches sur des machines parallèles ?

* 1. Méthodologie

Les heuristiques présentées dans la littérature sont généralement mis en relief par rapport à celles déjà existantes. L’objectif des auteurs étant de mesurer les performances des algorithmes proposés par rapport à celles qui existent. En effet, le développement de nouveaux algorithmes vise à proposer une solution améliorée par rapport à celles qui existent déjà. Il s’agit par exemple du cas de l’article [2] où l’auteur Hamid Arabnejad et al. procède à la selection des algoithmes avec lequel il voudrait faire la comparaison. Dans ce cas d’espèce, il s’agit de HEFT,HCP,HPS PETS et Lookahead. Après une brève description de ceux-ci, l’auteur propose un algorithme et procède ensuite à des études expérimentales pour effectuer la comparaison des performances. La comparaison peut être faite en générant de manière aléatoire des graphes ou alors en s’appuyant sur des cas réels de certaines applications. Dans de l’article [23], Haluk Topcuoglu et al. S’appuie les graphes de trois applications réels pour effectuer la comparaison. Il s’agit des applications de transformation de fourrier, le code dynamique moléculaire, et l’algorithme de l’élimination de Gauss. Les tests sont fait par le moyen de simulateur qui permet de tracer sur des courbes les performances des différents algorithmes.

Le travail que nous nous proposons de réaliser vise à faire une comparaison des heuristiques d’ordonnancement. Il est important d’entrée de jeu de rappeler que les systèmes de calcul parallèles permettent définir des architectures permettant de traiter des informations de manière simultanée, ainsi que de définir les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser un plus grand nombre d’opérations en un temps le plus court possible. La question de l'’ordonnancement consiste donc à distribuer une application sur une machine parallèle de manière à obtenir de meilleures performances ou de traiter les problèmes plus larges. Il s’agit d’un problème NP-difficile pour lequel il n’existe pas de solution optimale. De nombreuses heuristiques ont dont été développées proposant les unes les autres des solutions approximatives et partielles.

Il est question au cours de cette étude de définir ce qu’est l’ordonnancement, de procéder à l’état de l’art en identifiant et en présentant quelques heuristiques existantes dans la littérature, ensuite nous ferons un bilan comparatif de ces heuristiques. Nous proposerons un protocole expérimental qui nous servira de base de comparaison et enfin nous procéderons à la simulation des de l’exécution des heuristique avant de procéder à l’analyse des données obtenues. Nous achèverons ce travail par discussion sur les résultats obtenus de l’analyse, ce qui nous permettra de tirer une conclusion.

Ce travaille vise donc à faire une étude exhaustive des algorithmes afin de les comparer leur performance sur différents types de DAG.

1. IIème Partie : Présentation et Description de quelques algorithmes d’ordonnancement
   1. Qu’est-ce l’ordonnancement des tâches?

L'ordonnancement est un processus consistant à affecter des tâches à des ressources de calcul pour traitement. L’environnement de traitement des tâches pouvant être un homogène ou hétérogène. L’efficacité de l’exécution des applications parallèles dépend des méthodes utilisées pour ordonnancer les tâches. L’objectif étant d’attribuer des tâches aux processeurs, ordonner leur exécution de manière à satisfaire la relation de précédence des tâches et améliorer les performances. Ces performances sont définies par un ensemble de critères parmi lesquels le temps d’exécution, le makespan etc... La meilleure façon d’aborder le traitement/l’exécution d’une application de grande taille consiste à la diviser/partitionner en un ensemble de tâches en identifiant les relations de dépendance entre ces tâches . Ces tâches sont modélisées par un graphe de dépendance orienté sans circuit. Le graphe permet ainsi d’établir les relations d’interdépendance entre les tâches (quelle tâche doit-elle être exécutée avant quelle autre ?) et inclut par ailleurs les caractéristiques d’une application telles que : le temps d’exécution des tâches, la taille des données qui sont transférées entre les tâches, et l’interdépendance des tâches.

Le problème d’ordonnancement des tâches d’un graphe orienté sans circuit est NP-difficile aussi bien pour le cas des systèmes homogène que celui du cas des systèmes hétérogènes.

Etant donné l’importance de la question d’ordonnancement, elle a longuement été étudiée, et les travaux sur le sujet ont permis de classer les algorithmes d’ordonnancement plusieurs catégories : on peut par exemple citer les algorithmes d’ordonnancement de liste, les algorithmes de clustering, les algorithmes basés sur la duplication, et les algorithmes basé sur les méthodes de recherche aléatoires. Notre étude reposera essentiellement sur les algorithmes d’ordonnancement de type liste.

* 1. Les algorithmes d’Ordonnancement de Type Liste

Les algorithmes d’ordonnancement avec lesquels nous travaillons dans le cadre de cette étude sont ceux de type liste. Il s’agit des algorithmes qui fonctionnent en deux phases : la phase de priorisation des tâches et la phase d’ordonnancement. En effets, Ils déterminent pour un ordre de tâches, qui peut être donné par une liste, un ordonnancement correspondant. Dans un ordonnancement dit de liste, des ordres de priorités sont associés à chaque tâche et placés ensuite dans une liste ordonnée de façon décroissante des ordres de priorité. La tâche ayant une plus forte priorité sera ordonnancée avant celle de faible priorité. Dans le cas où deux tâches posséderaient la même priorité, une méthode spécifique peut être utilisée pour les départager ou alors le choix peut tout simplement se faire de manière aléatoire.

L’ordonnancement de type liste se déroule en deux phases :

* Phase de calcul de priorité des taches
* La phase de sélection du processeur

La différence qui existe sur les nombreux heuristiques dont l’ordonnancent est basé sur les listes réside sur les méthodes de calcul des priorités et celle de sélection du processeur. Les critères majeurs pour attribuer les priorités au taches sont en général le top level ou (*TL*) et le bottom level ou (*BL*).

Le top level (t-level) d’une tâche est défini comme étant la longueur du plus long chemin (coût d’exécution + coût de communication) entre cette tâche et celle d’entrée. Le TL permet d’évaluer le paramètre EST (earliest start time) d’une tâche.

Le bottom level (b-level) est la longueur du chemin le plus long (coût d’exécution + coût de communication) partant de cette tache vers la tâche de sortie. Le BL est borné par le chemin critique du graphe. Lorsque deux taches sont ordonnancées sur le même processeur, le poids de l’extrémité reliant les deux tâches est égal à zéro, ce qui implique que le TL d’une tache change et donc peut être dynamique. Lorsqu’un algorithme utilise un TL qui change pendant le processus d’ordonnancement, cet algorithme est qualifié d’algorithme dynamique. Dans le ca s contraire si l’algorithme utilise un TL déterminé avant le démarrage du processus d’ordonnancement, cet algorithme est qualifié d’algorithme statique.

Les heuristiques qui feront l’objet de cet étude sont basés sur l’ordonnancement de type liste et cible principalement les environnements de calcul hétérogènes.

* 1. Généralités sur la problématique de l’ordonnancement des tâches sur des machines parallèles

Un système d’ordonnancement est un ensemble composé d’une application et un environnement de calcul cible. Le principe de l’ordonnancement consiste à partitionner une application en tâches de façon à pourvoir la modéliser par un graphe orienté sans cycle (DAG), où est un ensemble de tâches et un ensemble de extrémités entre les tâches. Chaque extrémité représente les contraintes de précédence entre les tâches (la tâche doit s’achever avant la tâche ). Une tâche d’entrée (entry task) est une tâche ne possédant aucun parent/prédécesseur), une tâche de sortie (exit task) est une tâche ne possédant aucun successeur/tâche fille. L’environnement de calcul est un ensemble Q de *q* processeurs hétérogènes interconnectés. La communication inter processeur se fait sans contention.

La problématique d’ordonnancement des tâches consiste donc à affecter les tâches d’une application donnée sur des processeurs. Des critères permettent d’évaluer la performance parmi lesquels : le (Schedule length ratio), le temps d’exécution de l’agorithme.

Le problème de l’ordonnancement se résume donc à la reduction du cout d’exécution d’une application en réduisant autant que possible le*.* De nombreuses variables et paramètres sont pris en compte dans le calcul du.

|  |  |
| --- | --- |
| Nome de variale | descriptif |
| V | Un ensemble de tâches |
|  | Un ensemble de tâche ordonnacées |
|  | Ensemble de tâches non encore ordonnancées |
|  | Ensemble de relations entre les tâches |
|  | Cardinalité de lensemble |
|  | Nombre total des tâches |
|  | Nombre total des processeurs |
|  | Ième task |
|  | J ième proesseur |
|  | Une relation orienté partant de vers |
|  | temps d’exécution de la tâche i sur le processeur j |
|  | Temps de communication entre les processeurs i et j |
|  | Le temps pris pour transférer les données de vers |
|  | Mtrice des couts d’exécution de dimension |
|  | Coût d’exécution des taches |
|  | Le coût estimatif de la tache sur tout le processus |
|  |  |
|  | Ensemble des predecesseurs immédiats à la tache |
|  | Ensemble de successeurs immédiats à la tache |
|  | Le temps de début au plus tôt d’une tâche |
|  | Le temps de fin au plus tôt d’une tâche |
|  | ... |
|  | ... |
|  | Temps de disponibilié des données |
|  | Temps de disponibilité du processeur |
|  | Cout de l’ordonnancement |
|  |  |
|  |  |

Le but de cette section est de présenter quelques algorithmes  d’ordonnancement que nous avons identifiés dans la littérature: HEFT et CPOP, PEFT, PETS, DLS afin d’ évaluer et de comparer leur performances. Les métriques (paramètres) de comparaison étant les suivants : SLR (makespan), le facteur d’accélération (speedup), le nombre d’occurrence d’une meilleure qualité d’ordonnancement,  le temps d’exécution de l’algorithme.

* 1. Bilan comparatif des algorithmes d’ordonnancement des tâches

Dans ce chapitre, nous allons parcourir quelques algorithmes identifiés dans la littérature et procéder à une analyse de manière à pouvoir dégager un bilan comparatif des protocoles expérimentaux.

* + 1. ILHA (Iso-Level Heterogeneous Allocation)

L’article [4] d’Olivier Beaumont et al. introduit un heuristique d’ordonnancement dans un environnement de calcul hétérogène appelé ILHA(Iso-Level Heterogeneous Allocation). Le principe consiste à proposer à chaque niveau le meilleur équilibrage de charge (load balancing) en considérant un groupe de plusieurs tâches en lieu et place seule. L’idée ici est de procéder à l’affectation sur chaque processeur d’un nombre de tâches proportionnel à sa puissance de calcul. Une comparaison de l’heuristique ILHA est faite avec 5 autres :

* Minimum partial completion, time static priority;
* Best imaging level;
* heterogeneous earliest finish time;
* Critical path on processor;
* Generelised dynamic level.

La comparaison s’est faite en utilisant 5 bancs d’essai (testbeds) (LAPLACE, LU, STENCYL ,FORKJOIN,DOOLITTLE, LDMT)

* + 1. Minimum partial completion time static priority algorithm(PCT)

Cet algorithme se déroule en deux phases : celle de priorisation des taches et de l’affectation d’une tache sur un processeur. La priorité est calculée en faisant une estimation du temps nécessaire pour terminer le programme. Considérons le processeur sur lequel la tache est affectée : le numéro de priorité est définie par la formule suivante :

*Priority(Ni)=ei,j + (ci,k + priority(Nk)) où ci,k=Di,k x τ* est le temps nécessaire pour acheminer les données du nœud *Ni* vers le nœud *Nk*avec une bande passante *τ. D*ans la seconde phase les taches sont affectées selon l’ordre de priorité.

Soit  le processeur associé à la tâche, on définit *pct(,**)* comme étant le temps partiel d’exécution de la tache sur le processeur,. et l’instant auquel toutes les données nécessaires pour l’execution de sont disponibles. Si est le processeur sur lequel la tache est exécutée, on déduit l’équation suivante :

est l’instant auquel le processeur est disponible pour démarrer l’exécution d’une nouvelle tâche.

**L’algorithme PCT est défini ainsi qu’il suit :**

Compute the priority for each task

ReadyTask

While ReadyTask is not empty

Choose n in ReadyTask with the highest priority

Compute *pct(n,p)* for all *p* in *P*

Assign n to the processor which minimize pct(n,p)

Update A[p] and ReadyTask

End while

* + 1. Best imaginary level (BIL)

Cet algorithme base le calcul de priorité sur la notion de (Best Imaginary Level). Le est la longueur (durée) du chemin critique partant du nœud. L’expression imaginary découle du fait qu’il n’est toujours pas d’ordonnancer un nœud au meilleur des temps.

Une fois le BIL calculé pour chaque nœud, la seconde phase consiste à calculer la priorité. Il suffit alors d’ajuster le BIL du nœud de manière à calculer le Best Imaginary

Pour chaque nœud, si on considère étant le nombre de processeur, il existedifférentes valeurs du. Si à une étape il existe nœuds en cours de sélection, on defini le nœud comme le plus petite valeur du BIM.

Une fois le nœud sélectionné la prochaine étape consiste à trouver le processeur sur lequel la tâche sera affectée. Dans le cas où le nombre de de nœud à l’état prêt est supérieur au nombre de processeurs, on de une vision revisée du BIM comme il suit :

BIM\*(Ni,Pj) = BIM(Ni ,Pj) + ei,jxmax(. Le processeur choisi est celui ayant la valeur la plus grande du BIM revisé. Dans le cas où plus d’un processeur a la meme valeur du BIM revisé, le processeur sélectionné est celui dont la somme des valeurs du BIM revisé des autres nœuds sur le processus maximum.

L’algrothme est le suivant :

Compute BIL(n,p) for all n and p

ReadyTask

While ReadyTask is not empty

Compute BIM for every task in RedayTask

Choose the node n with the highest priority

Compute BIM\*(n,p) for all p

Assign n to the processor p that maximizes BIM\*(n,p)

Update ReadyTask

End while

* + 1. Heterogeneous ealiest finish time (HEFT) and critical path on a processor(CPOP) algorithms

Topcuoglu, Harir and Wu present dans l’article [1] deux heuristiques. Il s’agit des algorithme dit de liste qui se deroulent en deux phases. Priorisation et selection du processeur. Avant de se plonger dans une étude de ces deux algorithmes, il est important de s’attarder sur certaines définitions. On défini EST(earliest start time), EFT(earliest finish time) du nœud Ni sur le processeur Pj de la façon suivante : et

identifie le processeur auquel est affectée. EST retourne le temps auquel toutes les données necessaire par la tache sont disponibles sur le processeur ainsi que le temps auquel est disponible. Les taches sont classé suivant le rang montant (upward) et descendant (downward). Le rang montant d’une tâche ni est défini par la formule mathématique suivante :



est ltemps moyen d’exécution de la tache sur les processerus. **d**esigne la longeur du chemin critique partant de la tache vers le nœud de sortie.

De la meme façon nous definissons de manière recursive le rang descendant de la tache de la manière suivante :

Le est la distance la plus longue partant du noeud d’entrée jusqu’au noeud

HEFT

Pour calculer la priorité d’une tache , l’agorithme HEFT fait usage de la valeur rang montant (upward rank value). Les taches pretes sont classées en considérant l’ordre décroissant de la valeur du . dans le cas où deux nœuds possèdent la meme priorité, le choix se fait de manière aléatoire. HEFT utilise la valeur du EFT pour sélectionnner le processeur devant exécuter la tache selectionnée. Lorsque lorsque toutes les taches du graphes ont été affectées et exécutées, le temps d’ordonnancement est égale est egale à la valeur du EFT du nœud de sortie.

**L’algorithme:**

Compute for all nodes

ReadyTask

While ReadyTask is not empty

Select the task *n* with highest priority

Assign the task *n* to the processor p that minimizes the *EFT* value of *n*

Update *EST* values and *ReadyTasks*

End While

**CPOP**

L’algorithme CPOP utilise la valeur de pour le calcul de la priorité. La selection de la tache selon l’ordre de priorité les plus hautes, c’est dire les tache appartenant au chemin critique. Une tache appartient au chemin critique si la valeur de est égale à la valeur de . *Ns* étant le nœud de démarrage.le processeur de chemin critique(CPP) est le processeur qui offre le cout minimum du chemin critique. Lorsqu’une tache appartient au chemin critique, elle est affectée au CPP dans le cas contraire elle est affectée au processeur qui offre le plus petit EFT.

**L’algorithme**

Compute for all nodes

ReadyTask

While ReadyTask is not empty

Select the task *n* with highest priority

If n is on the critical processor

Assign n to CPP

Else

Assign the task n to the processor

Update *EST* values and *ReadyTasks*

End While

* + 1. Generalized dynamic level (GDL) algorithm

Le principe de cet algorithme est basé sur le calcul du chemin critique dans un système hétérogène. Contrairement à CPOP, GDL défini le temps d’exécution du nœud *Ni* comme étant le temps d’exécution median du noeud sur tous les processeurs, il est noté.

On note etant la plus grande des somme des temps d’exécution sur tout chemin orienté partant de. En prenant en compte la différence des processeurs, une nouvelle variable est introduite : . La notion de niveau dynamique est introduite =Cette formule ne considère comment les descendants de sont associé à . Pour chaque on note le descendant auquel

Passe le plus de données et la quantité de données passées,

Donne des indication sur le temps auquel peut etre complètement exécuté sur tout processeur. Si est exécuté sur alors , = . Ceci est considéré comme étant la borne inférieure du temps nécessaire pour terminer l’exécution de sur tout processeur autre que *Pj,* on deduit donc

Dans le cas où deus deux nœuds seraient affectés sur le meme processeurs, on defini le resource scarcity cost end définissant de prime abord le professeur preféré pour chaque nœud. Il s’agit du processeur maximisant son niveau dynamique. On deduit le cout lié à l’affectation de la tache non sur le processeur preférré. Ce cout est où est l’index du processeur préféré de la tache d.

EN prenant en compte le descendant et le resource scarcity, on defini le niveau dynamique généralisé (GDL) de la façon suivante :

L’algorithme :

Compute and for all nodes

Compute for all nodes

ReadyTask

While ReadyTask is not empty

Compute fo every node in *ReadyTask ande every processor*

Select the pair that maximizes

Update *ReadyTask*

End While

* + 1. The iso-level Heterogeneous Allocation(ILHA) algorithm

L’idée principale dégagée de cet algorithme est de proposer un équilibrage de charge entre les taches dont le calcul peut se faire en parallèle. Dans un premier temps on defini une méthode permettant d’équilibrer un ensemble de taches sur processeurs hétérogènes, ensuite on etudie comment le graphe de tache peut être divisé en niveaux qui sont par la suite affectés aux processeurs par le moyen de l’algorithme d’équilibrage de charge.

* + 1. Load balancing algorithm

Considéron groupes de calcul, chacun necessitant la meme charge de travail, comment affecter ces groupes de calcul à processeurs de manière à ce que la charge de travail de chaque processeur soit équilibrée ? si est le cycle de temps du processeur alors, la charge de doit etre inversement proportionnel à . Chaque devra recevoir une charge Cette stratégie conduit à un équilibrage de charge parfait lorsque est un multiple de

L’algorithme d’équilibrage de charge est le suivant :

for

find

L’algorithme ILHA propose deux version .

Première version : iso-level splitting

Dans cette version le graphe de tache est divisé en plusieurs niveaux constitués de graphes de taches indépendantes. Il s’agit des taches qui seront disponible au meme moment. Le niveau 0 est composé des taches d’entrée. Le niveau (i+1) est composé des taches qui sont à l’état prêt et dont le niveau i est exécuté.

Dans la première version de l’algorithme ILHA, le graphe on parcourt le graphe le graphe de tache pour le diviser en niveaux de taches indépendantes. Cela fait, il faut déterminer quelle tâche sera affectée à quelle processeur avec comme objectif minimiser le cout de communication. Pour chaque tache d’un niveau, on considère ses prédécesseurs. Si les prédécesseurs sont affectés sur un même processeur, on affectera toutes les taches de ce niveau sur le même processeur. Sinon, les taches seront affectées sur le processeur le plus rapide non encore saturé, d’où l’algorithme suivant :

*ReadyTask*

*While ReadyTask*  is not empty

Compute the optimal distribution with || *ReadyTask* || tasks

For each task *t* are on *p* ans *p* is free

Assign *t* to *p*

For each task *t* of *ReadyTask* not yet assigned

Assign *t* to the first free processor

Update *ReadyTask*

End While

Version revisée de l’agorithme ILHA

Dans l’algorithme précédente, toutes les taches à l’etat pret dans une étape ont été exécutées. Dans certain cas il serait preferable de prendre en compte le niveau bas ainsi d que la première tache sur le chemin critique. Ainsi, un tri des taches sera effectué sur la base du bottom level. On introduit le paramètre  *B*  comme etant le nombre maximum des taches pretes à considérer à chaque étape. Les B taches ayant les plus grands bottom level son affecté en utilisant le principe de l’algorithme d’équilibrage des charges. La liste des taches prete est mises à jour et à nouveau triée.

Compute the bottom level of each task

*ReadyTask*  sorted by decreasing

Value their bottom level

*While ReadyTask*  is not empty

Take the *B* first tasks of the *Readytask*

Compute the optimal distribution with *B* tasks

For each task *t* are on *ReadyTask*

If all predecessors of t are on p and p is free

Assign *t* to *p*

For each task *t* of *ReadyTask* not yet assigned

Assign *t* to the first free processor

Update th list of *ReadyTask* by inserting the new ready tasks in the sorted list

End While

* + 1. [13]Scheduling Parallel Program Tasks onto Arbitrary target Machines

Cet article introduit un nouvel heuristique appelé MH. Il s’agit d’un algorithme permettant l’allocation et l’affectation des taches dans les processeurs. La particularité de l’allocation réside ici sur la prise en compte des contraintes liées à la communication. Cet algorithme prend en considération les contraintes liées à la communication, les collisions et l’équilibre être le calcul et la communication.

Un mapper est un algorithme prenant en entrée (1) une description du programme ainsi que les interactions sous formes de graphe de taches (2) la description d’une machine cible sous forme de graphe non orienté. Comme sortie, cet algorithme un diagramme de gant qui consiste en une liste de tous les éléments de calcul. Pour chaque élément de calcul, une liste de toutes les taches affectées classé suivant la durée d’exécution. Dans un premier temps une première version, de l’algorithme est présentée sans prise en compte des contraintes liées à la collision des données. Une seconde version de l’algorithme serait par la suite présentée avec prise en compte des collisions.

Conformément aux recommandation de Adam *et al*., le calcul de priorité est défini en exploitant la notion de niveau déterminé pour chaque nœud. Après l’apport du delai de communication, le niveau de nœud n’est plus static et pourrait changer.

**Description de l’algorithme.**

1. Le niveau de chaque nœud sur le graphe de tâche est calculé et exploité comme le numéro d’ordre de priorité. Dans le cas où deux nœuds auraient le même numéro de priorité, le nœud possédant le plus grand nombre de successeurs dans le cas contraire la sélection sera faite de manière aléatoire. Pour chaque nœud n’ayant pas de successeur, établir une liste d’évènement ayant comme pour valeur « task is ready »
2. Tant que la liste d’évènement n’est pas vide, obtainir un evenement, dans le cas ou l’evenement indique « task is ready » une element de calcul est sélectionnné pour exécuter la tache. Le processeur sélectionné doit etre celui offrant le meilent temps d’exécution. Une fois la tache exécutée, elle passe à l’etat « task T is done ».Dans le cas ou l’evennemlent indique « task T is done », l’etat du successeur de cette tache est modifié.
3. L’etape 2 est repetée jusquà ce que tous les nœuds du graphe de tache sont affecté à un processeur.

Algorithme

Load the program task graph

Load the target machine

Compute the level of each graph

Initialize the event\_list(E)

While E is not empty do

Get event(e) from E

Process\_event(e)

End while

MH et prise en compte des conflits

Dans un système où plusieurs utilisateurs partagent un canal commun, il existe très souvent des conflits. Dans le paragraphe précèdent nous avons supposé que le canal de communication justifiait d’une capacité suffisante pour transmettre toute les informations sans un retard significatif. Cette section les modifications nécessaires sont introduites pour prendre en compte les confilts et lesretard liés au canal de communication. Les modifications qui vonst suivre visent à : introduire les delais dus aux conflits dans le paramètre de communication , rechercher les routes proposant des delais de communication faibles, et enfin produire un ordonnancement realiste. Pour chaque processeur, on maintient une table de routage contenant les informations suivantes : les informations de conflit, et des informations sur chacun des autres processeurs (le n nombre de sot *H* , la ligne de sortie preférée *L*, le delai de communication dû au conflit). Entre deux processeurs on prend en consideration le plus court chemin pour définir le nombre de sot et la ligne prefére de sortie.au cas où on aurait plus d’un plus court chemin, l’un sera choisi de façon aléatoire. *H(pi,pj)* designe le nombre de sots entre lmes processeur *pi,pj. .*  il en est de meme pour *l(pi,pj )* et *D(pi,pj).*

La table de routage qui est defini pour chaque processeur permet d’obtenir le meilleur chemin pour acheminer un message ainsi que pour calculer le delai dû à la cimmunication. Le delai de communication est influencé par : le temps de retard lors de la tranmission des données de *m1* vers *m2,* ainsi que le delai lié au conflit causé par les multiples messages transmis sur le chemin entre *n1 et n2.*  Le paramètre .

La table de routage defini pour chaque processeur est mise à jour pendant l’ordonnancement de manière à ce que le choix de la route pour envoyer des messages et la sélection d’un processeur pour exécuter une tache sont basés sur des informations concernant le traffic actuel.

Les questions qui nécessitent une réponse sont les suivantes : quand est-ce qu’une table de routage est-elle mise à jour ? Quelle table est-elle mise à jour ? Comment la mise à jour se produit-elle ? la table de routage est mise à jour lorsque l’un ou l’autre des evenements suivants se produit : un tache comment l’envoie de message à une autre tâche, ou un message arrive à sa destination.

Deux routines sont modifiés pour prendre en compte la gestion des conflits. La routine process\_event ainsi que la routine schedule task. Le reste de routine reste inchangé.

* + 1. [7]Multi Strict Bound constraint algorithm

Contrairement à HEFT, MSBC une méthode alternative de calcul de priorité des taches. Cet heuristique introduit une nouveau système de priorisation des taches dont les paramètres de cacule sont les suivants :

est defini comme la machine favorie dont le cout d’execution minimise la valeur . ce qui signifie que toutes les données necessaires par la tache du graphe sont disponible sur la machine .

defini la longeur du chemin critique partant de la tache jusqu’à la tache de sortie. La valeur *sbct* pour la tache d’entrée du graphe *G’*  est égale à

La valeur *start time (st)* de la tache *nj* d’un graphe G pour une fonction *ifav* est defini de manière recursive par :

Pour la tache d’entrée, *nentry*  ,

L’algorithme MSBC

L’algorithme fait use d’un système de priorisation differente de celle proposée par HEFT. Une precedence elevée est attribuée aux taches aux tache possedant une haute priorité. Une fois toutes les taches priorisées, chaque tache est ensuite affectée à sa machine favorie pour exécution. Pour déterminer la machine favorie, il suffit de calculer la valeur du EST (earliest start timpe). Entre deux taches déjà ordonnancées Il existe la possibilité d’insertion de tache pendant la période d’inactivité de la machine entre deux taches dejà ordonnancées.

[2]List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table.l MSBC offer une complexité égale à *O(*v2x *q)*

Algorithme

Calculate and for every task on graphe *G*

Calculate and for every task

Initially, the ready task pool includes only the entry task graphe *G*

**Repeat**

The unscheduled task with the largest value is selected from the ready task pool

Allocate te task to the machine with the minimum earliest times and then set it to be

Add new ready task in graph *G*  to the ready task pool

**Untill** all task are scheduled

Cet algorithme a été compare avec deux autres algorithme de la littérature(DLS, HEFT) il decoule des expérimentations faites que ses performances sont bien meilleures que ces duex algorithme lors qu’il est pris en compte la possibilité d’insertion de tache à des instants d’inactivité du processeur. L’expérience montre les performances de MSBC s’améliore considérablement lorsque la valeur du CCR augmente.

* + 1. [1] : Dynamic priority scheduling heuristic for heterogeneous computing systems

L’heureistique présenté dans cet article fait partie de la catégorie des heuristiques d’ordonnancment dite de liste. Les heuristique d’ordonnancement de liste règleent le problème d’ordonnancement dans les systèmes distribués en les divisant en deux étapes : la pahse de selection des taches et celle de selection du processeur. La tache à ordonnancer est celle ayant la plus forte priorité . une liste des taches à l’etat prêt est donc maintenue. Après l’ordonnancement d’une tache provenant du ready list, la prochaine tache à ordonnancer est celle dont les predecesseurs ont été ordonnacés. Elles sont donc insérés dans le ready list et marquée comme prêt.

Le calul des priorité est basé sur l’evaluation des valeur BL(bottom level) et TL(top level). Le cacul du BL et TL depend de la methode d’evaluation du doute d’execution des taches. Ondistingue quatre methodes :

* + 1. The average of the excution cost (moyenne des couts d’execution des taches)
    2. The median of the excution cost(la médiane des couts d’exécution des taches)
    3. The maximum of the execution cost(le maximum des couts d’execution)
    4. The minimum of the execution cost (le minimum des couts) d’exeution.

L’agorithme DPS peut se vervir de l’une ou l’autre des methodes suscités pour evaluer le BL et le TL des taches, ce qui pourra avoir un effet sur le calcul de la priotité des taches et par conséquent de la durée d’ordonnancement. Le calcul du BL et TL peut s’effectuer de plusieurs façon prenant en compte les changement suceptible d’arriver sur le DAG au cours de l’ordonnancement. Les valeurs du BL et TL sont évaluées en utilisant deux règles :

Règle 1/: le TL d’une tache *i*  est recursivementdefini de la manière suyivante

* + 1. For entry nodes (ie, tasks for which
    2. If is undetermined () then

Else

* + 1. If

Else if then

Else

Règle 2:BL of task is recursively defined as

* + 1. For all exit nodes (that is tasks for which {}),
    2. If is undetermined then

Else

* + 1. If or is undermined then =

Else if  *k=l*  then

Else

Au depart lorsqu’aucune tache n’est affectée à aucun des processeurs, l’attribut (le maximum des couts d’exécution des taches ) est utilisé pour calculer le TL. Par ailleurs, après chaque étape de l’ordonnancement, le BL et TL sont calculés en considérant le cout d’execution des taches de toutes les taches ordonnancées sur leur processeur correspondants ainsi que le couts de toutes les taches non en core ordonnancées defini par .

**L’algorithme DPS**

Compute TL and BL of each task according to rule 1 and rule 2 respectively and determine priority of each task;

Build a task list based on priority

Ready\_list initialize\_ready\_list (task\_list)

**While**(ready\_list not empty) do begin

Select tast with the highest priority value from the ready\_list

Select the processor of task accordint to rule 3 for earliest finish time

Sxhedule task onto processor

Delete task from the ready\_list

Ready\_list update\_ready\_list (task\_list)

Recompute the TL of ans unscheduled tas by using 1:

Recompute the priority of all tasks in the ready\_list

**End While**

* + 1. [3] Improving Scheduling of Tasks in a Heterogeneous Environment
       1. Description de fonctionnement des algorithmes HEFT et CPOP
       2. HEFT

L’algorithme d’ordonnancement HEFT est décrit dans l’article [1]. Cet article traite essentiellement de la question de l’ordonnancement statique des tâches dans un environnement hétérogène. Cet algorithme se classe parmi les algorithmes d’ordonnancement dits de liste. Il s’agit d’unalgorithme d’ordonnancement adapté à un nombre limité de processeurs. Cet algorithme se déroule en deux phases : la phase de priorisation des tâches et la phase de sélection du processeur. En ce qui concerne la phase de priorisation des tâches, elle consiste à définir un ordre de priorité des tâches en effectuant un classement de celles-ci. L’algorithme HEFT s’appuie sur les attributs du graphe pour définir l’ordre de priorité des tâches. L’ordre de classement des priorités des tâches se définit suivant un classement des rangs descendant ou ascendant. Une liste des tâches est générée en opérant un tri suivant l’ordre décroissant des rangs montant/ascendant.Le rang montant d’une tâche *ni* est défini par la formule mathématique suivante :

=`

*Ranku* désigne le rang de la tâche ni.  et désigne en d’autre termes la taille du chemin critique allant de la tâche ni à la tâche de sortie (exit task).

Il s’agit d’une fonction récursive où *succ(ni)* désigne l’ensemble des successeurs immédiats de la tâche *ni.*  définit le coût moyen de communication entre les extrémité (i,j), et  est coût moyen de calcul de la tâche ni. . De façon similaire le rang descendant de la tâche *ni*est calculé en traversant de façon descendante en commençant par la tâche d’entrée (entry task ) :

=

La phase de sélection du processeur quant à elle consiste à optimiser le *makespan* en adoptant une stratégie d’insertion. Cette stratégie consiste à insérer une tâche en considérant le temps d’inactivité au plus tôt entre deux tâches préalablement ordonnancée. la recherche d’un temps d’inactivité approprié pour une tâche *ni*commence à partir d’un temps égale au ready\_time(temps à l’état prêt) de *ni*sur le processeur *pj* c’est-à-dire l’instantoù l’ensemble des données transmises par les successeurs immédiats à la tâche *ni* parviennent au processeur *pj.* La recherche continue jusqu’à ce que le premier temps d’inactivité capable de contenir le coût de calcul de la tâche *ni* soit trouvé*. La complexité en temps de*  L’algorithme HEFT est égale à *O(exq)* (q étant le nombre de processeur et e le nombre d’extrémités). Pour un graphe dense, lorsque le nombre d’extrémités est proportionnel à *O(v2),* la complexité en temps est de l’ordre de O(v2xp).

* + - 1. CPOP (Critical Path on Processor)

Aussi bien que l’algorithme HEFT, l’algorithme CPOP se déroule en deux phases: la phase de priorisation et la phase de sélection. Cependant, cet algorithme utilise non seulement des attributs différents pour définir l’ordre de priorité des tâches mais aussi une stratégie différente pour déterminer le processeur optimal pour chaque tâche sélectionnée.

#### Phase de priorisation des tâches :

Dans cette phase, les valeurs des rangs montants et descendants des tâches se calculent en utilisant les coûts moyens de calcul et de communication. CPOP fait usage du chemin critique d’un graphe d’application donné. La taille de ce chemin |CP| est égale à la somme des coûts de calcul sur le chemin critique du graphe. La somme des coûts de calcule sur le chemin critique détermine la borne inférieur de la taille d’ordonnancement générée par l’algorithme d’ordonnancement des tâches. La priorité des tâches se calcule avec le résultat issu de la somme des rangs montant et descendants. La priorité de la tâche d’entrée est égale à la taille de du chemin critique. Initialement la tâche d’entrée est sélectionnée et marquée comme tâche de chemin critique. Un successeur immédiat ayant une priorité supérieure est sélectionné et marqué comme tâche de chemin critique. Ce processus se répète jusqu’à l’atteinte du nœud de sortie. Toutes les tâches à l’état prêt sont maintenues dans une file d’attente.

#### Phase de sélection du processeur :

Le processeur du chemin critique (pcp) minimise la valeur cumulative des coûts de calcul des tâches appartenant au chemin critique. Toute tâche sélectionnée et appartenant au chemin critique est affectée au processeur du chemin critique. Dans le cas contraire, elle est affectée au processeur qui minimise le temps d’exécution au plus tôt. Dans les deux cas une stratégie d’ordonnancement basée sur l’insertion des tâches est adoptée. La complexité en temps de l’algorithme CPOP est *O(exp).*

* + - 1. Analyse des performances et résultats

Les résultats obtenus dans cet article découlent de l’analyse des performances de l’algorithme HEFT proposé et les algorithmes existants (LMT, MH, DLS). Les critères de comparaisons sont les suivants : les métriques de comparaison étant, le ratio de la durée d’ordonnancement, le facteur d’accélération, l’efficacité, le nombre d’occurrences de meilleure qualité d’ordonnancement, le temps d’exécution de l’algorithme. Pour procéder à la comparaison des différents algorithmes, un générateur aléatoire de graphe a été développé, permettant ainsi de générer des DAG avec des caractéristiques variables en fonction d’un certain nombre de paramètres (le nombre de tâches, dans le graphe, le degré entrant, le degré sortant, le ratio de communication de calcul….). Pour chacun des critères de comparaison un graphe de comparaison a été réalisé. Le graphe ayant en abscisse le nombre de nœud et en ordonné le critère de comparaison.

Les résultats expérimentaux faisant suite aux de tests nous révèlent les informations suivantes :

* Performance basé sur le SLR : le classement des algorithmes est le suivant :

Performance basée sur le facteur d’accélération : La valeur moyenne du SLR (Schedule Length ratio) de l’algorithme HEFT sur tous les graphes générés a été de l’ordre de 7% meilleure que celle de l’algorithme CPOP, 8% meilleure que l’algorithme DLS, 16% meilleure que l’algorithme MH et 52 % meilleure que l’algorithme LMT.

* + - 1. Comparaison entre l’algorithme HEFT et CPOP

Sur la base des expérimentations réalisées, il a été remarqué que les performances l’algorithme HEFT surpassent suffisamment celles de l’algorithme CPOP en terme de SLR (Schedule length ratioin) ou makespan. L’algorithme HEFT est plus rapide que L’algorithme CPOP tandis que l’algorithme DLS est le moins rapide. HEFT est de l’ordre de de 8 %.plus rapide que l’algorithme CPOP.

* + 1. Description de fonctionnement de l’algorithme PEFT

l’article [2] présente un nouvel algorithme traitant de la question de l’ordonnancement statique des tâches dans un environnement hétérogène. Cet algorithme se classe parmi les algorithmes d’ordonnancement dits de liste. Il est question dans cet article de présenter un nouvel algorithme d’ordonnancement de liste pour un nombre de processeurs hétérogènes limité et entièrement connectés appelé PEFT (predicted earliest finish time). Cet algorithme dont la complexité en temps est de l’ordre *de O(v2xp)* surpasse l’algorithme HEFT en terme de makespan et d’efficacité. Il s’agit en effet du premier algorithme à surclasser l’algorithme HEFT toute fois en maintenant la même complexité. Cet algorithme introduit le concept dénommé lookahead assez nouveau et qui n’augmente pas la complexité de l’algorithme. Comme tout algorithme de liste, l’algorithme PEFT a deux phases : la phase de priorisation des tâches et celle de sélection du processeur. Cet algorithme découle de l’observation faite selon laquelle l’on ne saurait parvenir à une meilleure heuristique d’ordonnancement si la stratégie de sélection des processeurs est uniquement basée sur le temps d’exécution en cours. Il est cependant important d’aller au-delà de la tâche courante et de prendre en compte ce qui ce passe en dehors de la tâche courante (lookahead). L’algorithme Lookahead offre la possibilité d’aller au-delà de la tâche en cours et de prévoir l’impact de l’affection de toutes les tâches filles de la tâche en cours. Il s’agit d’un algorithme offrant le makespan le plus faible/bas. L’innovation de cet algorithme réside sur sa capacité à prévoir en calculant la table de coût optimiste (OCT optimistic cost table) et en maintenant une complexité de temps quadratique.

* + - 1. Table de coût optimiste(OCT)

Les phases de priorisation des tâches et de sélection du processeur sont basées sur le calcul d’une table de coût (OCT). L’OCT est en fait une matrice sur laquelle les lignes indiquent le numéro des tâches et les colonnes le numéro des processeurs. Chaque élément OCT (ti,pk) indique le maximum du chemin le plus court allant des ti tâches filles jusqu’au nœud/ tâche de sortie en considérant que le processeur pk est sélectionnée pour la tâche ti. La valeur OCT de la tâche ti sur le processeur pk est définie récursivement par la fonction suivante :

****

 Représente le coût moyen de communication et  représente le temps d’exécution de la tâche tj sur le processeur pw.  représente le temps optimiste maximum de traitement de la tâche ti.

* + - 1. Phase de priorisation des tâches

Pour définir la priorité des tâches, l’on calcule la valeur moyenne OCT de chaque tâche selon la formule proposée par l’équation suivante :



* + - 1. Phase de sélection du processeur

Pour sélectionner le processeur pour une tâche, le temps optimiste  qui est en fait la somme entre l’*EFT (earliest finish time)* et le temps de calcul du chemin le plus long jusqu’au nœud de sortie sont évalués. Le but est de garantir que la tâche en avant s’achèvera plus tôt. .

* + - 1. Description détaillée de l’algorithme PEFT

1. Calcul de la table OCT ainsi que du rang 
2. Création d’une liste vide et place la tâche d’entrée à la tête de liste
3. Ordonnancer ensuite les tâches ayant  les plus élevés.
4. Après avoir sélectionné la tâcher à ordonnancer, on calcule le temps optimiste  de la tâche sur tous les processeurs
5. Le processeur  qui réalise le minimum  est sélectionné pour exécuter la tâche 
6. Mettre à jour la liste
7. Les étapes 3,4,5,6 sont répétées jusqu’à ce que la liste devienne vide.
   * + 1. Analyse des performances et résultats

Les performances de cet algorithme ont été évaluées sur la base d’un certain nombre de métrique :

Le SLR (schedule length ratio), l’efficacité, la robustesse (slack). Comparé à l’algorithme HEFT, PEFT réalise un meilleur ordonnancement dans 72% d’exécution, un ordonnancement équivalent dans 3% et un très mauvais ordonnancement dans 25% d’exécution.

PEFT surpasse de ce fait l’algorithme HEFT dont le temps d’exécution était la meilleure avant la proposition de l’heuristique PEFT. Il s’agit en effet du premier algorithme à offrir une meilleur performance que l’algorithme HEFT.

* 1. Description de fonctionnement de l’algorithme PETS

L’algorithme d’ordonnancement PETS tel que décrit dans l’article [3] traite essentiellement de la question de l’ordonnancement statique des tâches dans un environnement hétérogène. Cet algorithme se classe parmi les algorithmes d’ordonnancement dits de liste. Il est question dans cet article de présenter un nouvel algorithme d’ordonnancement de liste pour un nombre de processeurs hétérogènes entièrement connectés appelé PETS (predicted earliest finish time) . Cet algorithme offre de meilleures performances en termes de ratio du coût d’ordonnancement(SLR), facteur d’accélération, efficacité et fréquence de meilleur résultat. )}. Le travail de recherche dans cet article a consisté à comparer l’algorithme proposé PETS et les études précédemment réalisées.  Algorithme LMT, HEFT, et l’algorithme CPOP. Les critères de comparaisons sont les suivants, durés d’ordonnancement des tâches, le temps de début au plus tôt d’une tâche (ESF) sur un processeur donné, le temps de fin au plus tard d’une tâche sur un processeur donné.

L’algorithme PETS se déroule en trois phases :

* La phase de tri par niveau
* La phase de priorisation des tâches
* La phase de sélection du processeur
  + 1. Phase de tri par niveau

Cette phase consiste à effectuer un tri en traversant de manière descendante le graphe orienté sans cycle (DAG) dans le but de regrouper les tâches dépendantes les unes des autres. Les tâches appartenant au même niveau sont exécutées en parallèle. La tâche d’entrée se trouve au niveau 0 et la tâche de sotie se trouve au dernier niveau. Le niveau i consiste à l’ensemble des tâches vk telle que pour chaque extrémité (vj,vk) , la tâche vj se situe dans un niveau inférieur que i, et il existe au moins une extrémité (vj,vk) telle que vj se trouve au niveau i-1.

* + 1. Phase de priorisation des tâches

Durant cette phase une priorité est calculée et affectée à chaque tâche. Trois attributs permettent d’assigner les priorités aux tâches : le coût moyen de calcul (ACC average computation cos), le coût de transfert des données (DTC data transfert cost), et le rang de la tâche précédente(RPT rank of predecessor task). L’ACC d’une tâche est le coût moyen de calcul sur m processeur et se calcule suivant l’équation suivante :

****

Le DTC d’une tâche mesure la quantité de communication pour transférer les données d’une tâche vi vers tous les successeurs immédiats. Il se calcule à chaque niveau suivant la formule suivante :

**** n désigne le nombre de nœud correspondant au niveau suivant.

Le RPT d’une tâche vi désigne le rang le plus élevé de tous ses prédécesseurs immédiats.

**** *v1, v2,…… vn* désignes les prédécesseurs immédiats de la tâche *vi.*

Le calcul du rang de chaque tâche est basé sur sur l’ACC, DTC, RPT correspondant à cette tâche.

****

La priorité des tâches est affectée a chaque niveau l, sur la base de la valeur de son rang. A chaque niveau, la tâche ayant le rang le plus élevé recoit la plus grande priorité, ainsi de suite.

* + 1. Phase de sélection du processeur

Durant cette phase, la tâche possédant le temps de fin au plus tôt (EFT) est sélectionné et affectée au processeur.  Il existe une stratégie d’insertion considérant la possibilité d’inserer une tâche dans un temps d’inactivité au plus tôt entre deux tâches déjà ordonnancée sur le processeur.

* + 1. Analyse des performances

L’analyse des performances se fait sur la base d’un certain nombre de facteurs :

* Ratio de taille d’ordonnancement (SLR/makespan)
* Facteur d’accéllération(speedup)
* Efficacité(Efficiency)
* Nombre d’apparition de meilleure qualité d’ordonnancement
* Temps d’exécution de l’algorithme

Trois tests ont été menés pour évaluer les performances des différents algorithmes.

Le 1er test a consisté à évaluer la qualité d’ordonnancement générée par chacun des algorithmes. Ce test s’est effectué sur la base d’un ensemble de graphe généré de façon aléatoire. Les tâches ont été ordonnancées sur un système hétérogène regroupant 15 processeurs. Les résultats issus de cette expérimentation montrent bien que l’algorithme PETS en terme de SLR/makespan surclasse l’algorithme HEFT de l’ordre de 8%, l’algorithme CPOP de de l’ordre de 17% et l’algorithme LMT de l’ordre de 40%. Pour ce qui concerne le temps d’exécution, l’algorithme PETS s’est révélé plus rapide que les algorithmes HEFT de l’ordre de 23%, CPOP de l’ordre de 39% et LMT de 48%.

Le deuxième test s’est effectué sur des applications réelles (transformation de fourrier rapide, décomposition LU). Le classement des algorithmes basé sur la fréquence d’apparition des meilleurs résultats était le suivant : {PETS, HEFT, CPOP, LMT}.

* 1. Description de l’algorithme DLS

L’algorithme d’ordonnancement DLS tel que décrit dans l’article [4] traite essentiellement de la question de l’ordonnancement des tâches dans un environnement de traitement parallèle en prenant en compte les surcoûts liés aux communications inter processeurs. Le travail effectué dans cet article a consisté à développer une heuristique d’ordonnancement non préemptif faisant correspondre un graphe orienté sans cycle à une architecture susceptible de contenir processeurs hétérogènes interconnectés de façon irrégulière. Cette technique est appelé ordonnancement à niveau dynamique (dynamique level scheduling). Le point d’entrée de l’algorithme d’ordonnancement est un graphe de précédence sans circuit (APEG : acyclic precedence expansion graph). G= {N.A} où N={Ni: i=1,……,n} est un ensemble de nœuds/tâches représentant le programme de calcul. A représente un ensemble d’arcs orientés {Ai,j} ayant un label Di,j . Di,j définit la quantité de donné qui circule entre Ni  et Nj. L’architecture cible est un ensemble de processeurs hétérogènes P= {Pk: k=1,….,n} disposants d’une ressource matérielle spécifique dédiée à la communication de manière à ce que la communication soit imbriquée au calcul/traitement par les processeurs. L’objective de l’ordonnancement est bien évidement de minimiser la taille d’ordonnancement (SLR) ou le makespan et par ailleurs maximiser le facteur d’accélération (speedup). Une revue rapide de la littérature révèle un ensemble de solution proposant des approches de solutions (l’approche de regroupement linéaire et l’approche d’internationalisation). Cet article propose une nouvelle approche de solution. Avant de proposer la solution, il a été question dans l’article d’introduire la notion d’ordonnancement dynamique à niveau (DLS) pour des processeurs homogènes, présenter la méthode d’organisation de l’algorithme, comparer les performances de l’algorithme à celle de deux autres algorithmes prenant en compte la communication inter processeur et enfin étendre l’algorithme aux systèmes hétérogènes.

* + 1. Communication inter processeur

L’ordonnancement en présence de la communication inter processeur comporte deux aspects fondamentaux : l’affectation des processeurs aux tâches/nœuds de calcul, et l’allocation des ressources pour le transfert des données de communication inter processeurs. Ces deux problèmes sont connus comme étant le « problème de mapping ». En prenant en compte ces deux aspects, la stratégie d’ordonnancement évite les surcharges des ressources de communication en ajustant convenablement le nœud–processeur de mapping. Les communications étant ordonnancées sur le processeur, les communications sont quant à elles ordonnancées sur les ressources de communication inter processeur. Dans ce cas, une ressource dédiée est utilisée dans le transfert de la donnée pour la durée de communication. Avec la garantie de disponibilité d’une ressource le temps de communication peut être calculé de façon déterministe en utilisant la localisation des processeurs source et destination, la quantité de donnée à transférer et les caractéristiques de l’architecture de communication.

* + 1. Les niveaux dynamiques des processeurs hétérogènes

Aussi bien que les algorithmes classique dites de liste, l’algorithme DLS utilise les même approche d’ordonnancement qui consistent en : la priorisation des tâches, le stockage des tâches dans une liste et le tri dans l’ordre décroissant de celles-ci. Les nœuds dont tous les prédécesseurs immédiats sont exécutés sont dits disponibles pour l’affectation. Une horloge globale s’occupe de la régulation du processus d’ordonnancement. Lorsqu’un processeur est disponible l’algorithme lui affecte la première tâche disponible. Après l’affectation le nœud est supprimé de la liste. Lors que la liste des tâches/nœuds prêts est vide ou l’ensemble des processeurs disponible est épuisé, l’horloge global est incrémenté jusqu’à l’achèvement de l’exécution des tâches assignés aux processeurs.

Les approches d’ordonnancements identifiés sont les suivantes :

* L’ordonnancement HLFET
* L’ordonnancement à niveau dynamique
  + 1. L’ordonnancement HLEFT

HELFT (high level first with estimated time), est un algorithme d’ordonnancement populaire dont la technique consiste à: affecter un niveau ou priorité à chaque nœud/tâche,

* + 1. L’ordonnancement à niveau dynamique(DLS)
    2. Algorithme DLS

1. IIIème Partie : Comparaison des heuristiques

Proposition d’un protocole expérimental

**Objectif**

La comparaison des heuristiques que nous nous proposons de réalisé est basée un protocole expérimental. Ce protocole nous permet de définir la méthodologie de travail, d’identifier les critères et paramètres d’évaluation ainsi que les valeurs qui serviront à l’évaluation.

**Méthodologie**

Notre approche méthodologique s’appuie sur la simulation des algorithmes. Il est effectivement question d’implémenter un simulateur capable de générer les DAG, exécuter les algorithmes et à l’exécution et générer des données de sortie. Les données de sorties seeront ensuite analysées selon des critères bien définie

Plus précisément il s’agit de la construction d’un simulateur qui nous permettra d’une part de générer les graphes de tâches (DAG), d’autre part d’exécuter algorithmes, et enfin d’analyser les données générées.

**Génération des DAG**

La génération des DAG se fait de façon aléatoire. Toutefois leur structure dépends d’un certain nombre de critère bien identifiés :

* Nombre de nœud : n
* CCR (Computer to communication ration)
* Nombre de processeurs
* Le coefficient de variation des processeurs qui nous permettra de définir le dégré d’homogenéité ou d’hétérogénéités des processeur
* Le nombre de niveaux du DAG

Ces paramètres nous permettrons de définir des DAG de différentes topologies. Il est important de faire l’analyse sur differentes topologies des DAG pour avoir un résultat pertinent. Le coefficient de variation des processeurs permet de définir le niveau d’homogenéité des processeur. Lors que cette valeur est petite, cela signifie que les processeurs sont prest homogènes et lors que cette valeur est grande, celà traduit la forte hétérogénéité entre les Processeurs. Le CCR est un paramètre qui lorsqu’on le fait varier permet de définir soit un DAG pour lequel la communication entre tâche est intensive ou alors un DAG représentant des calculs intensifs au niveau des processeurs. Lorsque le CCR présente une valeur faible ce là signifie que l’application présente des calculs intenses. Dans le cas contraire, si cette valeur est élevée, cela signifie que la communication entre les tâches de l’application est intense.

Ces paramètres permettent donc de génrer les DAG et en faisant varier leur valeur, nous obtenons des DAG ayant des topologies. Ces paramètres permettent par ailleurs de définir le dégré d’homogenéité d’un DAG.

Pour générer les DAGs nous utilisont un programme developpé avec le langage python. La fonction de génération prend en entrée les paramètres suivants : length, depth, filename, sdComp, sdComm, CCR, nbproc et génère en sorties un fichier GML représentant le DAG

* :param length: Length of the graph (number of nodes).
* :type length: int
* :param depth: Depth of the graph (number of levels)
* :type depth: int
* :param filename: Output filename
* :type filename: str
* :param sdComp: Standard Deviation of computations costs
* :type sdComp: float
* :param sdComm: Standard Deviation of communications costs
* :type sdComm: float
* :param CCR: Communications to Computations Ratio
* :type CCR: float
* :param nbproc: Number of processors used when generating the graph
* :type nbproc: int
* :return: Generated and converted graph

**Choix des valeurs**

La littérature présente le PEFT comme étant l’heuristique qui offre la meilleure performance. Nous nous proposons de maintenir les mêmes valeurs de génération des graphes qui ont été prien compte dans l’article PEFT. Ces valeurs sont :

* length= [10; 20; 30; 40; 50; 60; 70; 80; 90; 100; 200 300*;* 400*;* 500],
* CCR =[ 0:1; 0:5; 0:8; 1; 2; 5; 10]
* sdComp = [0:1; 0:2; 0:5; 1; 2]
* nbproc = [2; 4; 8; 16; 32]
* sdComm = [0.1,0.25,0.5,0.75,1]
* level = [1,2,3,4,5,length]

en modifiant les paramètres avec les valeurs identifiés ci-dessus permet de de définir des DAG ayant des topologies variées. La génération des DAG de ayant des topologies différentes permet de réaliser une analyse pertinente afin d’obtenir des résultats non biaisés qui prennent en compte les différentes configuration des DAGs .

Ces valeurs représentent une évolution linéaire du nombre des nœuds et permettent d’observer si les résultats obtenus en fonction du nombre des nœuds et du nombre de processeurs restent constant ou alors varient. Dans le cas d’une évolution constante nous auront des arguments sur les garanties de la qualité des résultats.

# Choix des heuristiques

Nous travaillerons principalement avec les 5 heuristiques suivants : HEFT,PEFT,PETS,DLS,CPOP.

En effet, HEFT est présenté dans la littérature comme un algorithme offrant de très bonnes performances. PEFT est l’algorithme dont les performances supplantent celles HEFT. En faisant le choix de ses deux algorithmes principalement, il est question de vérifier cette affirmation. Chaque auteur essayant de présenter son algorithme comme étant le meilleur, ce travail s’inscrire dans la dynamique visant à offrir les bases d’évaluation pour les algorithmes identifiés et d’en tirer les conclusions.

**Métriques de mesure des performances**

De nombreux paramètres permettent l’évaluation de la performance d’un Heuristique. Le paramètre principal est le makespan. Ce paramètre permet de mesurer le coût d’exécution de la tâche de sortie. Tant il est vrai que ce paramètre est très important dans la mesure des performances des algorithmes, il demeure vrai qu’il n’est pas suffisant pour analyser les performances de plusieurs heuristiques. Etant donné que nous travaillons avec des DAG de différentes caractéristiques et topologies, il est important de faire appel à de nouveau paramètre ou métrique d’analyse. Il est donc question d’évaluer la moyenne des performances obtenues par chaque heuristique pour chaque topologie

Le critère de base sur lesquels nous allons nous appuyer dans notre étude sont les suivants :

* SLR
* makespan
* temps d’exécution

**Analyse des performances**

L’analyse des performances se fait sous forme de diagramme. Pour chacune des caractéristiques du graphe (nombre de nœuds, valeur du CCR). Une analyse de performance sera faite entre chaque algorithme.

L’algorithme le plus performant sera celui dont le nombre d’occurrence du meilleur résultat sera le plus élevée .

**Choix des technologies**

Il sera question de développer un parseur qui servira de convertir les fichier .gml ( représentant le DAG générer en python) sur un format consommable par notre simulateur developpé Java.

Nous allons par la suite implémenter les algorithmes. Il s’agit de construire un simulateur qui prendra en entrée les DAG et fournira en sortir l’ordonnancement des tâches sur les processeurs. Les données d’ordonnancement des tâches seront ensuite formater pour être analysées en python sous forme de courbe.

**Modélisation des graphes sur java**

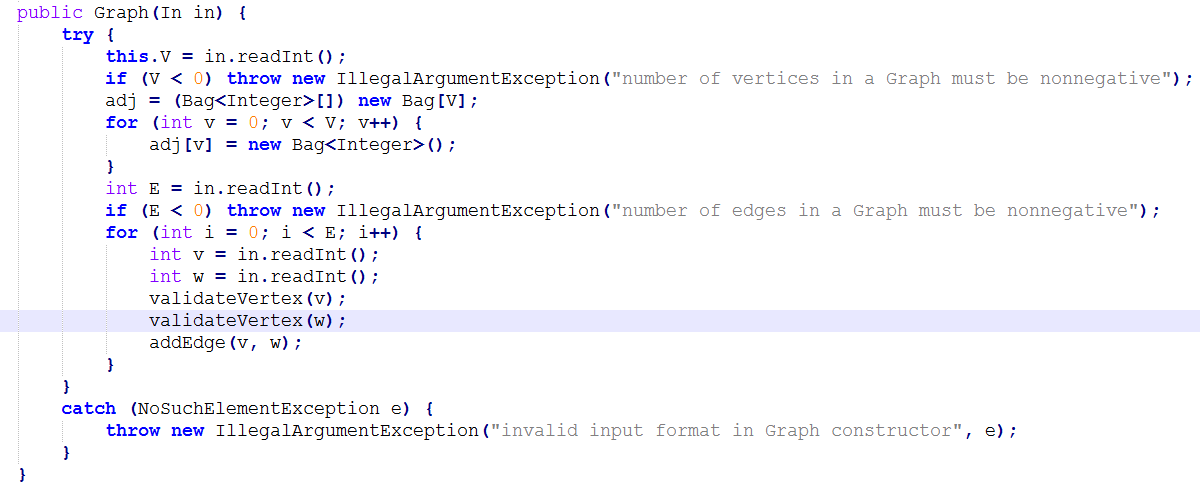
Le langage Java n’offre aucune structure de donnée permettant de manipuler directement les graphes. Il faut donc définir une structure de donnée susceptible de représenter les graphes sous forme de matrice d’adjacence. Nos nous sommes servis des ressources prévenants du lu lien suivant : <https://algs4.cs.princeton.edu/41graph/>.

De manière native, java n’implémente aucune structure de donnée permettant de manipuler les graphes. Il sera question pour nous de définir une structure de données nous permettant de définir un graphe comme une matrix d’adjacence. De ce fait nous allons définir les structures de données suivantes :

* Bag

D’après [[1]](#footnote-1) “*A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items.*[*Stats.java*](https://algs4.cs.princeton.edu/13stacks/Stats.java.html)*is a bag client that reads a sequence of real numbers from standard input and prints out their mean and standard deviation*”.

La structure de données Bag permet est implémenté selon la notion des listes chaînées. C’est donc cette structure qui nous permettra de définir notre graphe comme état une matrice d’adjacence des tâches.

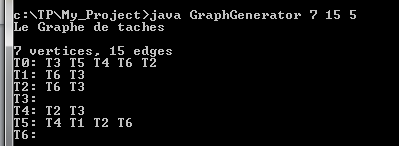


Implémentation et exécution des algorithmes PEFT et HEFT

Nous avons définir un programme que nous appelons GraphGenerator. Ce programme prend en entrée les informations suivantes :

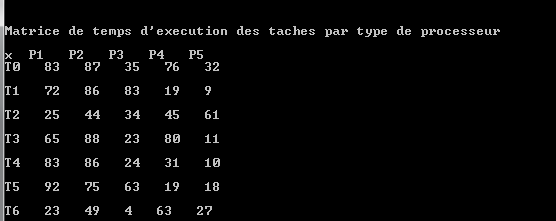
* V : le nombre tâches
* E : les nombre de relations dans le graphe
* P : le nombre de processeur

Exemple d’exécution du programme :



Cette image illustre ungraphe de tâche générée. Pour chaque tâche Ti, nous avons la liste des tâches qui lui sont adjacentes.

Ce programme génère ensuite les temps d’exécution des tâches sur chaque processeur comme l’illustre l’image suivante :

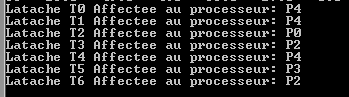


Nous enfin la matrice illustrant les couts de communication inter-processeur :

Ces données servent de base pour l’exécution des algorithmes que nous envisageons analyser les performances :

Exécution de PEFT

L’exécution de l’algorithme PEFT nous permis d’obtenir pour les données précédentes les résultats suivants :



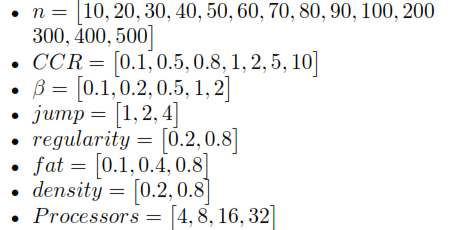
Exécution De HEFT (en cours de finalisation…….)

**Analyse des résultats**

L’analyse des résultats consistera à comparer le temps d’exécution global (SLR) et des algorithmes PEFT et HEFT afin de tirer.

* 2. Génération des graphes de taches

Les critères de comparaison servent aussi à définir le type et la structure des graphes. Ainsi, les types de graphes générés résultent de la combinaison des paramètres de comparaison identifiés. Pour obtenir un type de graphe, on fait varier un certain nombre de paramètres. A titre d’exemple, dans l’article **List scheduling Algorithm for heterogeneous systems by optimistic table (PEFT),** à la section 5.2**,** la définition des paramètres suivant ainsi que leur combinaison a permis de générer 70560 type de graphe, et pour chaque type de graphe, 10 graphe sont générés de manière aléatoire en faisant varier les paramètres tels que poids de chaque nœud , ce qui conduit le nombre de graphe de l’expérimentation à 705600:



Dans l’article [2], une évaluation des performances des algorithmes ci-dessus-cité a été réalisée et les résultats suivants ont été obtenus.

L’analyse des performances s’est fait sur la base d’un certain nombre de facteurs :

* Ratio de taille d’ordonnancement (SLR/makespan)
* Facteur d’accéllération(speedup)
* Efficacité(Efficiency)
* Nombre d’apparition de meilleure qualité d’ordonnancement
* Temps d’exécution de l’algorithme

Trois tests ont été menés pour évaluer les performances des différents algorithmes.

Le 1er test a consisté à évaluer la qualité d’ordonnancement générée par chacun des algorithmes. Ce test s’est effectué sur la base d’un ensemble de graphe généré de façon aléatoire. Les tâches ont été ordonnancées sur un système hétérogène regroupant 15 processeurs. Les résultats issus de cette expérimentation montrent bien que l’algorithme PETS en terme de SLR/makespan surclasse l’algorithme HEFT de l’ordre de 8%, l’algorithme CPOP de de l’ordre de 17% et l’algorithme LMT de l’ordre de 40%. Pour ce qui concerne le temps d’exécution, l’algorithme PETS s’est révélé plus rapide que les algorithmes HEFT de l’ordre de 23%, CPOP de l’ordre de 39% et LMT de 48%.

Le deuxième test s’est effectué sur des applications réelles (transformation de fourrier rapide, décomposition LU). Le classement des algorithmes basé sur la fréquence d’apparition des meilleurs résultats était le suivant : {PEFT, HEFT, CPOP, LMT}.

D’après la littérature il découle que PEFT suggère l’algorithme d’ordonnancement de tâches le plus performant, suivi de HEFT. Dans les paragraphes qui suivent, nous allons tenter de confirmer ou non cette affirmation. Dans notre approche, nous allons implémenter les algorithmes PEFT et HEFT et procéder ensuite à la comparaison sur la base des résultats obtenus.

Nous allons implémenter ces algorithmes en utilisant langage Java, ensuite nous exploiterons les résultats obtenus pour faire des comparaisons en exploitants en utilisant R.

1. Conclusion

Pendant cette étude nous avons lu avec beaucoup d’intérêt les articles relatifs aux algorithmes d’ordonnancement des tâches dans les systèmes parallèles. Notre «étude s’est principalement focalisée sur les articles décrivant les algorithmes DLS, PETS, HEFT, et PEFT. Chacun de ces articles présente de façon claire les performances des algorithmes étudiés. Le calcul de performance a été évalué en utilisant aussi bien les graphes de tâches générés de façon aléatoire que celles découlant des applications réels. Les critères de performances étaient les suivant :

* Le temps moyen d’exécution de l’algorithme ;
* Le facteur d’accélération ;
* le nombre d’occurrence d’une meilleure qualité d’ordonnancement ;
* le temps d’exécution de l’algorithme.

Les résultats démontrent à suffisance que sur la base des critères de performance cités plus haut, PEFT est l’algorithme qui réalise une meilleure performance. D’où le classement suivant :

{PEFT, PETS, HEFT, DLS} suivant l’ordre ascendante des meilleures performances.

# Référence

[1] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Eﬀective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed* *Systems*, 13(3), pp. 260-274, 2002.

[2]H. Arabnejad and J. Barbosa. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2014.

[3]E. Ilavarasan and P. Thambidurai. Low complexity performance eﬀective task schedu-ling algorithm for heterogeneous computing environments. *Journal of Computer Science*, 3(2) :94–103, 2007.

[4]G.C. Sih and E.A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architecture. *IEEE Transactions on Parallel and* *Distributed Systems*, 4(2), pp. 175-187, 1993.

1. https://algs4.cs.princeton.edu/13stacks/ [↑](#footnote-ref-1)