Equilibrage de charge distribué sur StarpU

Pélagie Alves




UNIVERSITÉ DE BORDEAUX

Mémoire de stage de Master 2 CISD

Équilibrage de Charge Distribué sur StarPU

Pélagie ALVES

Tuteurs : Olivier AUMAGE et Laércio LIMA PILLA

Référente : Marie-Christine COUNILH

01/03/2022 - 19/08/2022

Remerciements

Je tiens à remercier tout particulièrement Olivier et Laércio mes tuteurs de stage pour leur gentillesse, leur disponibilité, leur bienveillance et leur soutien.

Je remercie également Marie-Christine, mon enseignante référente, pour son aide et sa bienveillance tout au long de ce stage.

J'ai également reçu beaucoup de soutien de la part de toute l'équipe, plus particulièrement : Nathalie, Denis, Amina, Emmanuelle.

« Bonsoir » à Raymond, et merci pour ses très bon conseils et pour les opportunités qu'il m'a apporté.

Mention spéciale pour Alice pour ces bons moments passés ensemble et sa bienveillance.

Et pour finir, merci à mes collègues de l'open-space pour leur accueil chaleureux.

Table des matières

Présentation de l'Institut de Recherches

J'ai fait mon stage de fin d'étude dans le domaine de la recherche, chez Inria Bordeaux un institut national de recherche en sciences et technologies du numérique. Le stage a duré du 1er mars au 19 août. Inria est composé de 3900 chercheurs divisés en plus de 200 équipes de recherche, situées sur 9 centres différents. Le bâtiment accueille des équipes de recherche qui travaillent sur un thème bien spécifique. Le bâtiment est découpé en zones qui sont attribuées aux équipes avec à l'intérieur des bureaux, un open-space et une salle de pause.

Mon stage s'est déroulé dans l'équipe STORM, *Static Optimizations and Runtime Methods*, qui élabore et développe des solutions pour répondre aux besoins actuels et futurs dans le domaine du Calcul Haute Performance (HPC, *High Performance Computing*). Ils ont notamment créé un support d'exécution nommé StarPU [1] sur lequel j'ai travaillé durant ce stage.

L'équipe de permanents est principalement composée d'enseignants chercheurs ou de chercheurs, il y a aussi des ingénieurs. Ils travaillent aussi avec le CNRS, le LaBRI, l'Université de Bordeaux et Bordeaux INP. Les membres permanents sont : Olivier AUMAGE, Marie-Christine COUNILH, Nathalie FURMENTO, Amina GUERMOUCHE, Laércio LIMA PILLA, Raymond NAMYST, Mihail POPOV, Emmanuelle SAILLARD, Samuel THIBAULT, Pierre-André WACRENIER et Denis BARTHOU le responsable d'équipe. Scott BADEN et Jean-Marie COUTEYEN sont des membres rattachés. L'autre partie de l'équipe est composée de doctorants, d'ingénieurs et de stagiaires.

Toutes les semaines, je faisais un point avec mes tuteurs pour discuter de l'avancement du projet. Toutefois, si j'avais des questions, je peux passer les voir dans leur bureau où discuter avec eux sur discord. Les gens de l'équipe étaient ouverts à la discussion.

Introduction

Une des applications du HPC (*High Performance Computing*) est la simulation numérique. En modélisant les phénomènes physiques, on peut faire des prédictions, comme les prévisions météorologiques ou encore de l'aide à la conception dans l'industrie et la recherche, permettant ainsi de réduire les coûts de production. Ces simulations sont de plus en plus précises et nécessitent une puissance de calcul toujours supérieure. Les architectures des super-calculateurs quant-à elles évoluent constamment à la recherche de plus de puissance de calcul. Mais pour tirer pleinement parti des ressources mises à disposition, il faut d'abord paralléliser l'exécution des applications HPC et ensuite, veiller à l'équilibrage de charge de travail au fil de l'exécution. Pour ce faire, il existe des outils et des modèles de programmations spécifiques.

C'est notamment le cas du support d'exécution ou *runtime system* qui se charge de paralléliser l'exécution d'une application. Cette solution a pour principal avantage de rajouter une couche d'abstraction entre l'application et la machine, permettant la portabilité de l'application sur d'autres types d'architecture. StarPU est un support d'exécution à base de tâches développé par STORM. L'application transmet des données et des tâches à StarPU qui se charge de construire le graphe de dépendances des tâches appelé DAG (*Direct Acyclic Graph*). Ensuite StarPU répartit l'exécution des tâches sur les processeurs disponibles. Si des processeurs partagent la même zone mémoire, on dit qu'ils sont sur le même nœud.

Il existe différentes versions de StarPU: une version intra-nœud, qui distribue et équilibre la charge de travail sur les différents processeurs au sein d'un seul nœud et une version inter-nœud qui le fait sur plusieurs nœuds en même temps. Lorsque l'on distribue les tâches sur différents nœuds, on peut utiliser le modèle master-workers où le maître se charge de la distribution des tâches et de l'équilibrage, mais cette solution n'est pas scalable en fonction du nombre de nœuds.

Une autre solution consiste à avoir une instance de StarPU par nœud, c'est le modèle fully distributed. Dans ce cas là, chaque nœud possède une copie du graphe de tâches. Les données sont distribuées aux nœuds qui deviennent leur propriétaire. L'exécution d'une tâche est réservée au nœud qui possède la ou les données en sortie c'est-à-dire les données sur lesquelles la tâche écrit. On a donc un lien entre les données et les tâches et en déplaçant une donnée ayant un accès en écriture sur une tâche, on déplace également cette tâche.

L'autre particularité de ce modèle est la vision locale de la situation sur chaque nœud, en effet un nœud ne connaît que sa propre charge de travail. Ce détail est crucial car sans informations sur la charge globale de la machine, l'équilibrage de charge devient beaucoup plus complexe à réaliser.

Dans la version actuelle du modèle fully distributed, c'est à l'utilisateur de faire la distribution des données sur les différents nœuds et de préciser quels transferts de données faire pour rééquilibrer la charge de travail. A l'avenir on souhaiterait que l'utilisateur n'ait plus à intervenir pour l'équilibrage de charge mais que celui ci se fasse dans StarPU via le déplacement des données.

Ce stage est une première approche pour défricher le problème et mieux le comprendre. Le but du stage est de réaliser un simulateur python simplifiant le fonctionnement de StarPU et de tester des algorithmes de rééquilibrage de charge via des mouvements de données. Afin de faciliter la mise en oeuvre, je ne travaillerai pas dans StarPU mais via un simulateur python.

Le rapport est structuré comme suit. Dans la première partie, je présenterai le contexte autour de ce sujet en expliquant l'intérêt des supports d'exécutions, décrivant les différents types de support d'exécution à base de tâches et particulièrement le fonctionnement de StarPU. Je décrirai ensuite les différents types d'algorithmes d'ordonnancement existants et j'énoncerai la problématique. Dans la seconde partie, j'expliquerai pourquoi mettre en oeuvre l'équilibrage de charge distribué en fonction des données et des tâches et un problème complexe et je proposerai un modèle simplifié pour réaliser le simulateur. Dans la troisième partie, je présenterai la structure de mon simulateur et comment j'ai réalisé une implantation à partir du modèle décrit dans la partie précédente. Je décrirai les problèmes que j'ai rencontré en voulant adapter des algorithmes d'ordonnancement de la littérature à notre contexte. Je clôturerai cette partie en présentant des points techniques concernant le simulateur notamment l'adaptation en version distribuée. Dans la quatrième partie, je présenterai les différentes applications que j'ai réalisées et montrerait le fonctionnement de mon simulateur sur un exemple concret en comparant les résultats obtenus avec les différents algorithmes. Pour finir, je conclurai et présenterai les perspectives d'évolution pour mon travail.

Chapitre 1

Contexte

Cette section est une mise en contexte autour de mon sujet, elle montre comment le calcul intensif répond à la demande de puissance de calcul, sur le plan architectural et logiciel notamment avec les supports d'exécution à base de tâches. Elle permet de présenter les différents types de support d'exécution à base de tâches avec dépendance de données et plus particulièrement StarPU. En fonction de l'architecture de la machine et des outils utilisés, le contexte pour faire l'équilibrage de charge change. C'est pourquoi il existe différents type d'algorithmes d'ordonnancement qui fonctionnent dans une situation bien spécifique. Ces éléments me permettront de dessiner les enjeux autour de mon sujet et de poser une problématique.

1.1 Calcul Intensif

De nos jours, la simulation numérique est devenue incontournable car elle est utilisée dans notre quotidien et dans de nombreux domaines comme la recherche ou l'industrie. Avec les avancées technologiques, il est dorénavant possible d'obtenir des représentations très réalistes en se basant sur des phénomènes physiques et sur des données récupérées avec des capteurs. La simulation numérique a trouvé sa place dans la phase de production industrielle où elle permet de compléter ou même remplacer certaines phases d'expérimentations, ayant pour but de valider la conception et ainsi limiter les crash-tests coûteux. La simulation est aussi utilisée pour représenter des éléments non reproductible à échelle humaine, c'est par exemple le cas de modélisations de l'espace basées sur des données récupérées par les satellites qui permettent de faire des découvertes et d'expérimenter dans des environnements et des échelles d'espace et de temps qui sont inaccessibles.

Ce type de simulations numériques est appelé simulation réaliste et s'appuie sur des techniques telles que la discrétisation pour créer un modèle numérique discontinu à partir du modèle réel continu. La précision de la simulation dépend donc de la discrétisation que l'on a faite du réel, on appelle ce paramètre le grain. Plus le grain est fin, plus on a un résultat précis et réaliste mais plus le calcul est coûteux. C'est pourquoi, le besoin en puissance de calcul augmente constamment.

Pour pallier ce besoin, il faut augmenter la puissance de calcul sur les machines. Pour cela, une des solutions q longtemps été d'augmenter la fréquence des processeurs et donc la vitesse de calcul. Ce n'est maintenant plus possible à cause de la déperdition de chaleur générée par les processeurs à trop haute fréquence. La solution qui s'impose maintenant est d'utiliser plusieurs processeurs pour faire les calculs en parallèle. Cette technique permet d'augmenter la puissance de calcul globale de la machine.

Si tous les processeurs de la machine partagent la même zone mémoire, on considère que l'on est dans le cadre du parallélisme classique : les données et le travail sont connus par la machine qui partage la charge de travail entre les différents processeurs. Mais si on a plusieurs machines répliquées, appelées nœuds, la mémoire n'est plus partagée et les informations ne sont pas forcément connues par tous les nœuds. On appelle cette architecture distribuée. Il peut aussi y avoir des architectures hétérogènes qui mélange des processeurs (CPU) et des cartes graphiques (GPU).

Les machines capables d'une grande puissance de calcul sont appelées super-calculateurs et sont organisées selon une architecture spécifique. Les machines les plus puissantes selon le top 500[8], reprennent le concept de l'architecture distribuée, c'est notamment le cas de Frontier[10] le meilleur super-calculateur du monde actuellement.

Pour exploiter de telles machines, il faut être capable de répartir les calculs sur les différents processeurs c'est-à-dire de paralléliser l'exécution d'une application. L'objectif étant de maximiser l'utilisation des ressources mises à disposition pour l'exécution de l'application. Cette tâche est complexe et nécessite l'utilisation d'outils de programmation conçus pour le parallélisme. Pour cela, il existe des langages de programmation parallèle comme OpenMP[13], ou encore des bibliothèques comme MPI[11] qui permettent de faire de la programmation sur les architectures distribuées. Il faut donc adapter les applications séquentielles avec ces outils pour qu'elles fonctionnent sur les super-calculateurs.

La science des super-calculateurs et de leur architecture couplée aux outils pour programmer sur les machines de calcul est appelée calcul intensif ou *HPC* en anglais. Certains outils interviennent en amont de l'exécution d'une application comme les compilateurs, d'autres pendant l'exécution, sous la forme d'une bibliothèque, comme les supports d'exécution.

Le rôle du support d'exécution est de paralléliser l'exécution d'une application pendant l'exécution de celle-ci. Il ajoute une couche d'abstraction entre la machine et l'application, comme le montre la Figure 1.1, ce qui permet une plus grande portabilité de l'application puisque grâce au support d'exécution, elle s'adaptera à différents types d'architectures. Par exemple, les supports d'exécution à base de tâches parallélisent l'exécution des différentes tâches de l'application sur les ressources, ici une tâche est un noyau de calcul avec des entrées et des sorties. L'application doit donc être écrite à base de tâches. Pour préciser les dépendances entre les tâches, on utilise des synchronisations. Ce procédé n'est pas idéal car il rend l'écriture de l'application plus complexe et difficile à déboguer. De plus, les synchronisations créent du temps d'inactivité pour les processeurs, puisqu'il faut attendre que tout le monde ait fini pour passer le point de synchronisation. Pour optimiser la parallélisation, on préfère éviter ce genre de contraintes et se concentrer sur des solutions asynchrones. C'est pourquoi il existe des supports d'exécution qui déterminent les dépendances entre tâches via les informations d'accès aux données en lecture et/ou en écriture. Je présenterai les supports d'exécution à base de tâche avec des dépendances dans la sous-section suivante.

1.2 Supports d'Exécution à Base de Tâches avec Dépendance de Données

Avec les supports d'exécution à base de tâches avec dépendances, les dépendances sont créées en fonction des accès en lecture et en écriture sur les données. Elles sont ensuite, représentées sous la forme d'un graphe de tâches appelé DAG (*Direct Acyclic Graph*) comme montré dans la Figure 1.2. L'exécution d'une tâche se fait uniquement si ses dépendances ont été satisfaites. Cette méthode permet l'exécution en concurrence des tâches tout en supprimant les points de synchronisation qui

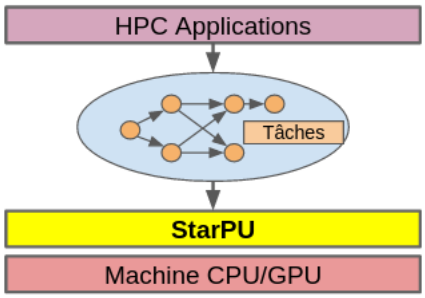


FIG. 1.1 – *Rôle de* STARPU *dans l'exécution d'une application*

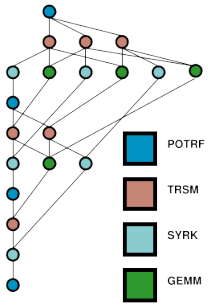


FIG. 1.2 – *Graphe de dépendances des tâches (DAG)*

induisent des temps d'inactivité. Il existe deux moyens d'exprimer les dépendances entre les tâches : la méthode explicite et la méthode implicite.

Le principal modèle de programmation à base de tâches déclarées explicitement est le PTG[6] (*Parametrized Task Graph*). Dans ce cas, les dépendances sont exprimées dans le code de l'application manuellement et doivent prendre en compte les flux de données en lecture et écriture. Le programme est compilé et génère un graphe de tâches, ensuite les tâches sont réparties et exécutées. Le principal inconvénient avec cette méthode est lié au fait d'avoir à ré-exprimer toutes les tâches de l'application avec les dépendances de données en lecture et en écriture. Ce problème n'est pas trivial et engendre une grande quantité de travail, puisqu'il faut exprimer quelles données seront les entrées et les sorties de toutes les tâches. Cette précision permet de réduire l'empreinte mémoire car le résultat de l'exécution d'une tâche sera directement transmis à la prochaine tâche qui en aura besoin. Cette solution est efficace sur des applications itératives, comme les opérations d'algèbre linéaire dense, car on réutilise les même tâches plusieurs fois sur les mêmes données. En revanche, sur des applications dynamiques avec des comportements irréguliers, il faut exprimer chaque particularité et cela rend cette méthode trop coûteuse. PaRSEC[4], un support d'exécution développé par l'université du Tennessee utilise le modèle PTG, pour exprimer les dépendances le langage particulier appelé JDF est utilisé. Leur solution est scalable car la représentation du graphe dépend du nombre de tâches différentes dans l'application et non pas de la taille du problème. Mais la complexité qu'elle nécessite pour sa mise en oeuvre, fait qu'elle est relativement peu utilisée. Les utilisateurs préfèrent des méthodes plus abordables et productives comme le modèle STF.

Le modèle STF (*Sequential Task Flow*) permet de garder la structure séquentielle du code en se basant sur une déclaration des dépendances implicite. Des séquences de tâches sont soumises

au support d'exécution pendant le fil de l'exécution de l'application, grâce à une fonction non bloquante. Les dépendances sont calculées automatiquement à partir de l'ordre de soumission et des entrées/sorties des tâches, elles sont ajoutées au graphe de tâches DAG. Ensuite lorsque la tâche a satisfait toutes ses dépendances, elle est considérée comme prête, le support d'exécution peut alors l'ordonnancer pour qu'elle soit exécutée. L'exécution des tâches se fait de manière asynchrone et parallèle. Ce modèle est plus simple d'utilisation car il faut uniquement s'assurer de la soumission des tâches. Il nécessite moins de travail d'adaptation de code car la structure séquentielle de l'algorithme est préservée, il fonctionne donc sur tous types d'applications. Il offre un meilleure rendement à l'utilisateur que le PTG et est donc plus utilisé.

Les contraintes et la scalabilité du modèle utilisé dépendent de la configuration architecturale sur la machine. En utilisant un runtime system sur un seul nœud, on fait de l'intra-nœud, la mémoire est partagée au sein du nœud, toutes les informations sur le nœud sont à disposition de tous les processeurs. Il existe un cas particulier où le nœud est hétérogène : composé de CPU et GPU et où la mémoire n'est pas directement partagée. Pour simplifier l'explication je considérerai des nœuds homogènes de CPU. En intra-nœud, le nœud possède le DAG et les tâches sont distribuées aux processeurs sur le nœud qui les exécutent en respectant les dépendances décrites, il s'agît d'un contexte de parallélisation. L'algorithme utilisé pour l'équilibrage de charge connaît la situation globale sur le nœud et la disponibilité de chaque processeur. Cette vision omnisciente de la situation est appelée ordonnancement centralisé.

Parfois les applications à traiter sont trop grandes pour tenir en mémoire ou s'exécuter sur un seul nœud. C'est pourquoi les super-calculateurs sont composés de plusieurs nœuds de calculs utilisables simultanément. Cette situation est appelée distribuée ou inter-nœud. Pour répartir et équilibrer la charge de travail en distribué, il existe différents modèles comme le modèle master-workers, où le nœud maître connaît les informations relatives à tous les nœuds et se charge de la bonne répartition des tâches sur les différents nœuds. Il a une vision globale de la situation et prend ses décisions en fonction de ses connaissances. Ce modèle s'utilise avec de l'ordonnancement centralisé. Mais si le nombre de nœuds devient trop élevé, le maître sera saturé par les requêtes de tous les nœuds workers pour récupérer du travail, et n'arrivera plus à gérer la situation de manière optimale. Cette solution n'est donc pas scalable en fonction du nombre de nœuds.

Un modèle distribué scalable en fonction du nombre de nœuds, a donc été proposé, il s'appelle le modèle complètement distribué ou fully distributed [1]. Ici, chaque nœud a une vision locale de la situation, c'est-à-dire qu'il ne connaît que ses informations personnelles. En cas de rééquilibrage de charge, il faut communiquer avec les nœuds voisins pour réussir à obtenir des informations sur leur charge. Cette contrainte peut conduire à des problème de scalabilité en fonction du nombre de communications et de l'algorithme de charge choisi. Ce contexte particulier pour faire de l'équilibrage de charge est nommé ordonnancement distribué. De plus, chaque nœud doit garder en mémoire une copie du DAG qui permet de prendre en charge l'exécution des tâches tout en respectant les dépendances. Mais le DAG peut prendre une grande place en mémoire et poser un problème de scalabilité en fonction de la taille du DAG s'il est répliqué sur toute la machine.

1.3 Support d'Exécution StarPU

Maintenant que j'ai présenté les concepts autour de StarPU, je vais exposer son fonctionnement détaillé. StarPU est un support d'exécution à base de tâches avec dépendance de données, il utilise le modèle STF. Il fournit une couche d'abstraction permettant d'exécuter des applications

sur des nœuds hétérogènes. Il fonctionne en parallèle (intra-nœud) et en distribué (inter-nœud). En distribué, StarPU propose deux sous-modèles, le master-workers et le fully distributed.

StarPU définit la structure *data handles* qui fournit une abstraction des données de l'application et permet de ne pas tenir compte au niveau applicatif des différents espaces mémoires possibles sur les nœuds hétérogènes. Par soucis de simplicité, on utilisera données à la place de data handles dans la suite du document.

StarPU définit la structure *codelet* qui est un ensemble de noyaux de calculs pour exécuter une opération spécifique. Les noyaux de calculs varient en fonction de l'architecture ou des optimisations que l'on souhaite utiliser. Par exemple, pour faire un produit matriciel sur des CPU Intel, on ne va pas utiliser le même noyau que sur une carte graphique Nvidia. En plus des noyaux de calcul, une codelet définit également la liste des accès en lecture et écriture requis pour la réalisation de l'opération.

En combinant la notion de codelet et de données, on obtient une tâche StarPU. La tâche a son codelet et une liste de données sur lesquelles on appliquera un des noyaux de calcul du codelet en respectant les accès lecture et écriture.

Voici le déroulé pour exécuter une application à l'aide de StarPU sur un nœud. Les données de l'application sont enregistrées sur le nœud. Ensuite, l'application soumet des tâches à StarPU au fil de son exécution, celles-ci sont ajoutées au DAG, leurs dépendances sont calculées implicitement en fonction de l'ordre de soumission et des accès en lecture et en écriture *read (in) write (out)* sur les tâches. Elles sont exécutées en concurrence sur le nœud, dès qu'elles ont satisfait leurs dépendances. StarPU se charge également de l'équilibrage. La décision d'ordonnancement est relativement simple à prendre dans un contexte centralisé. En revanche, le temps de prise de décision pour le rééquilibrage et de redistribution des tâches doit être inférieur au temps d'exécution global des tâches restantes puisque l'exécution et le rééquilibrage se font en parallèle. Ce contexte d'ordonnancement dynamique implique d'utiliser des algorithmes d'assez faible complexité.

Dans certains cas, il est préférable d'être dans un contexte distribué pour exécuter une application, notamment pour bénéficier d'une mémoire totale plus grande. Pour cela, StarPU a deux solutions : le modèle master-workers et le modèle fully distributed. Avec le modèle master-workers, le maître connaît le DAG et distribue les données et les tâches aux nœuds travailleurs, il est aussi chargé de l'équilibrage de charge. L'ordonnancement est fait de manière centralisée mais cette solution n'est pas scalable en fonction du nombre de nœuds, c'est pourquoi une version fully distributed a été implémentée.

Avec la version fully distributed, chaque nœud a une vision locale de l'application. Une instance de StarPU intra-nœud, par nœud gère la distribution des tâches et l'équilibrage de charge au sein d'un nœud. Mais l'utilisateur doit préciser la distribution des données sur les nœuds dans l'application, les données sont distribuées et les nœuds deviennent propriétaires des données qu'ils possèdent. Sur la Figure 1.3, le nœud 0 possède les données A et D, le nœud 1 B et E et le nœud 2 C et F, la distribution des données a été réalisée selon la politique 2D-bloc-cyclique[12] illustrée avec la Figure 1.4. Les tâches sont ensuite distribuées en fonction de leurs sorties (accès écriture), un nœud propriétaire d'une donnée sera responsable d'exécuter toutes les tâches qui écriront sur cette donnée. Sur la figure, la tâche 1 écrit sur la donnée A, c'est donc au nœud 0 propriétaire de A de l'exécuter. Si une tâche a des sorties sur plusieurs données qui sont sur différents nœuds, l'exécution de la tâche revient à un des nœuds aléatoirement. La distribution des tâches dépend donc de la distribution des données. Pour l'exécution des tâches, le nœud se réfère au DAG. Le contexte d'ordonnancement distribué complique la situation pour mettre en place l'équilibrage de

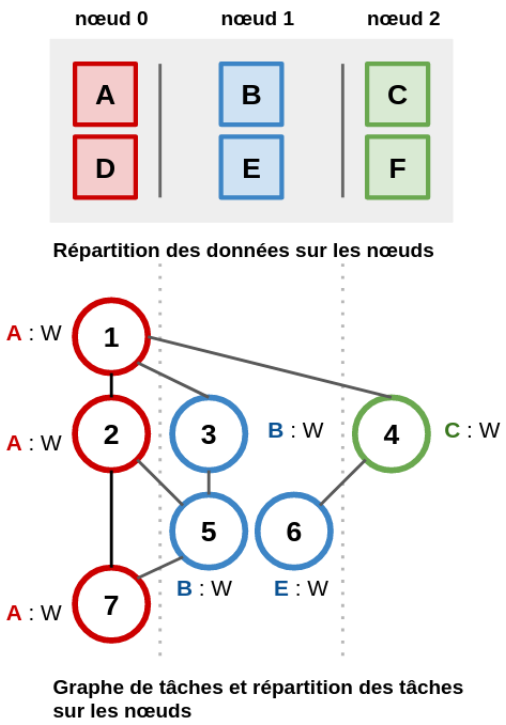


FIG. 1.3 – *Schéma représentant le fonctionnement du modèle fully distributed de* STARPU

charge, surtout en considérant la dépendance qu'il y a entre les données et les tâches. Pour l'instant, l'ordonnancement n'est pas fait par STARPU et doit être précisé par l'utilisateur dans l'application. Il doit ajouter des appels à une fonction de STARPU *data_migrate* dans son code pour déplacer des données sur un autre nœud. Cet appel a donc pour effet indirect de redistribuer les tâches associées aux données déplacées en écriture au nouveau nœud propriétaire. Prédire les déséquilibres de charge lors de l'exécution d'une application est un problème complexe tout comme trouver une politique de déplacement de données permettant de rééquilibrer la charge au fil de l'exécution. Cette solution temporaire rajoute du travail et peut devenir très compliquée en fonction du profil de l'application que l'on souhaite exécuter. De plus, le DAG qui est présent sur chaque nœud pose un problème de scalabilité en mémoire en fonction de sa taille. Pour supprimer ce problème une

FIG. 1.4 – *Schéma montant la répartition 2D-bloc-cyclique des données d'une matrice, chaque couleur représente un processeur*

optimisation consistant à élaguer le DAG a été réalisée dans StarPU. En effet, chaque nœud a besoin de connaître les tâches qu'il aura à exécuter et les dépendances qui attraient à ces tâches, les autres informations ne sont pas nécessaires et prennent énormément de place. La Figure 1.5 montre l'élagage du graphe pour le nœud 2. En élaguant le DAG de chaque nœud en fonction de ses besoins, on obtient une connaissance partielle du DAG et on limite le problème de scalabilité en fonction de la taille du DAG. Mais cette optimisation pose problème pour le déplacement des données et des tâches lors du rééquilibrage de charge. Si on déplace des données sur un nœud qui ne connaît pas les tâches et les dépendances liées à ces nouvelles données, il doit être capable de reconstruire le DAG pour réaliser correctement l'exécution. Ce problème n'a pas encore été résolu. Toutes ces problématiques expliquent l'absence d'équilibrage de charge distribué automatique dans StarPU pour le modèle complètement distribué.

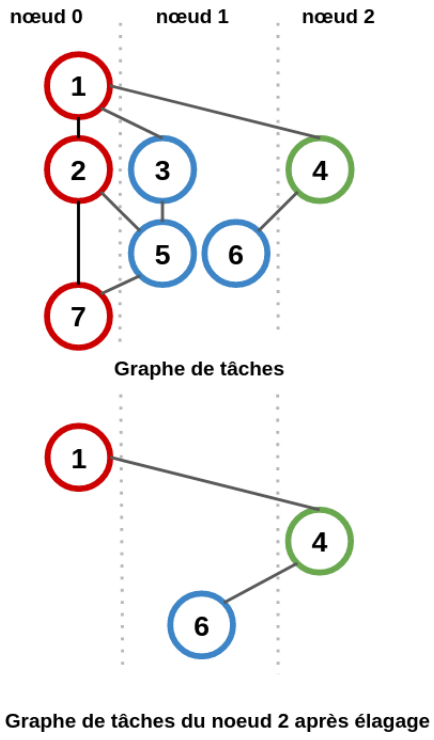


Fig. 1.5 – *Schéma représentant l'élagage du DAG pour le nœud 2*

1.4 Algorithmes d'Équilibrage de Charge

Pour trouver une solution pour l'équilibrage de charge distribué sur StarPU, il faut comprendre et connaître les différents types d'algorithmes d'ordonnancement existant et comprendre le contexte dans lequel ils fonctionnent. Durant mon stage, j'ai vu trois type d'algorithmes, les algorithmes centralisés, les algorithmes distribués et les algorithmes hiérarchiques.

1.4.1 Algorithmes Centralisés

Les algorithmes centralisés fonctionne dans un contexte d'ordonnancement centralisé. List Scheduling est un algorithme centralisé simple, le but est d'ordonnancer une liste de tâches de charges différentes sur un groupe de processus. La liste est parcourue dans l'ordre et les tâches sont attribuées au processus le moins chargé. La Figure 1.6 montre la distribution des tâches avec cet algorithme, la tâche rouge est donnée à P1 le processus le moins chargé, la tâche rose est aussi distribuée à P1 et pour finir la tâche bleue est attribuée à P2 qui est devenu le processus le moins chargé. La complexité de cet algorithme s'exprime en fonction de la taille de la liste n et du nombre de processus p, en assumant $p < n$ est $O(n \log p)$ [2].

L'algorithme LPT *Largest Processing Time* est une version améliorée du list scheduling, au lieu de distribuer les tâches dans l'ordre de la liste, la tâche la plus volumineuse est ordonnancée en premier. La Figure 3.2.3 donne l'ordonnancement avec le LPT dans le même contexte que pour list scheduling, la tâche rose est placé sur P1 en premier ensuite les tâches bleue et rouge sur P2. Le temps d'exécution total des tâches est inférieur avec le LPT car il donne la meilleur répartition des tâches possible. Sa complexité est de $O(n \log n)$[2], elle est légèrement supérieure au list scheduling puisqu'il faut parcourir la liste pour connaître la charge des tâches. Les algorithmes centralisés ont une faible complexité mais nécessitent une connaissance de la charge globale pour prendre les décisions.

FIG. 1.6 – *Ordonnancement avec le List Scheduling*

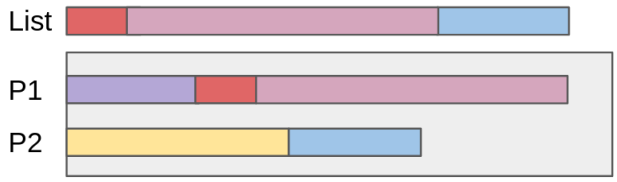
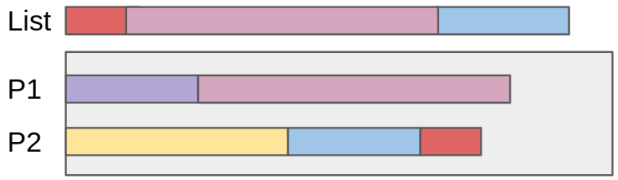


FIG. 1.7 – *Ordonnancement avec le LPT*



1.4.2 Algorithmes Distribués

Les algorithmes distribués fonctionnent dans un contexte d'ordonnancement distribué, où chaque nœud a une vision locale de la situation. L'équilibrage de charge est plus difficile à mettre en place car il doit se faire en autonomie et implique des communications entre les différents nœuds. Il faut tâtonner pour s'approcher d'une situation équilibrée. En ayant une connaissance partielle des informations, l'équilibrage risque d'être de mauvaise qualité. L'algorithme Selfish[3] associe à une tâche, un agent qui doit choisir un nœud. La charge sur un nœud est représentée par le nombre

d'agents qui l'ont choisi. Les agents qui sont sur un nœud surchargé doivent aller sur des nœuds sous-chargés, ces déplacements se font sans prendre connaissance de ce que font les autres agents. Selfish nécessite plusieurs itérations pour atteindre une situation approximativement équilibrée. Cela implique d'évaluer la situation et de déplacer les agents jusqu'à ce que le critère d'équilibrage recherché soit atteint. Cette solution fonctionne pour les applications avec des tâches qui ont un coût d'exécution similaire, car en évaluant la charge du nœud en fonction de son nombre d'agents, on suppose que toutes les tâches ont le même coût. Obtenir une solution approximativement équilibrée coûte $O(\log \log n)$ et pour une situation parfaitement équilibrée, il faut $O(\log \log n + p^4)$, avec n le nombre de tâches et p le nombre de nœuds. Le rééquilibrage arrête l'exécution des tâches donc l'algorithme doit avoir une complexité très faible.

PackStealLB[7] est un autre algorithme d'équilibrage de charge distribué qui reprend le fonctionnement général de Selfish. Cet algorithme a été réalisé pour le support d'exécution Charm++[9] qui fonctionne à base de tâches et avec des communications asynchrones comme StarPU. Les nœuds communiquent leur charge ce qui permet d'obtenir une moyenne de la charge de calcul sur la machine. En fonction de cette moyenne, les nœuds sont regroupés en deux catégories surchargé et sous-chargé. Les nœuds surchargés réalisent des paquets de tâches de même taille, cela équivaut à résoudre le problème du sac à dos. La réalisation de paquets permet de réduire les communications en regroupant des petites tâches ensemble, les paquets seront ensuite volés par les nœuds sous-chargés. Pour trouver des victimes, ils discutent avec leurs voisins, à la recherche de paquets à leur voler. Un paquet peut être choisi une seule fois, il faut donc s'assurer que deux nœuds sous-chargés ne prennent pas le même paquet. Si la récupération des paquets met trop de temps à converger, des combinaisons aléatoire de nœuds sont réalisées, mais il y a des chances de réaliser des paires de nœuds sous-chargés ou de nœud surchargés. Pour obtenir un équilibre, il faut itérer plusieurs échanges de paquets. La fin de la phase d'équilibrage est définie par un critère d'arrêt. Une fois que les paires ont été réalisées et que le critère d'arrêt est satisfait, les tâches sont déplacées. Pour éviter de considérer des nœuds proches de la moyenne en charge de travail comme surchargés ou sous-chargés, il est possible d'ajouter un seuil à partir du quel un nœud est considéré comme déséquilibré. Les nœuds non déséquilibrés sont considérés comme neutres et ne participent pas aux échanges de tâches. Cette solution converge rapidement car elle a été conçu pour faire l'équilibrage lorsque l'exécution des tâches est stoppée. Elle a donc un faible sur-coût *overhead*. Je n'ai pas étudié en détail la complexité de cet algorithme qui dépend de facteurs variés car je n'ai pas implémenté cet algorithme dans le cadre de mon stage. Cet algorithme produit un équilibrage de bonne qualité sur des applications de types variés. Mais il ne fonctionne pas dans le contexte de StarPU distribué car il ne permet pas un ordonnancement dynamique en parallèle de l'exécution des tâches.

1.4.3 Algorithmes Hiérarchiques

Il existe une dernière catégorie d'algorithmes d'ordonnancement, basés sur une arborescence hiérarchique des nœuds. L'arbre est composé de différents niveaux, à chaque niveau le nœud parent partage son travail avec ses nœuds fils comme le montre la Figure 1.8. Il y a 8 nœuds, le nœud 0 est la racine de l'arbre, il se divise en deux branches une avec lui même et une avec le nœud 4 qui est un fils de 0. Le nœud 4 contient les sous nœuds 6,5,7. Du coté gauche de l'arbre, le nœud 0 se sépare du nœud 2 dans lequel est inclus le nœud 3. Ainsi de suite. A la racine de l'arbre, la vision de la situation est globale. En s'enfonçant dans l'arbre, on distribue le travail à des sous groupes de nœuds qui reproduisent ce schéma, à chaque niveau inférieur la vision de la

situation est plus locale qu'avant. Sur chaque niveau le rééquilibrage se fait de manière centralisé par le nœud parent, cette particularité offre la possibilité d'utiliser des algorithmes centralisés pour l'ordonnancement. Les algorithmes d'ordonnancement utilisés à chaque niveau peuvent être différents pour tirer au mieux parti de l'architecture de la machine et de la structure de l'arbre. Cette solution est plus scalable que l'approche centralisée et permet d'avoir un équilibrage de charge très efficient par rapport à certains algorithmes d'équilibrage distribués. Periodical Hierarchical LB [14] est un algorithme implémenté sur Charm++. L'ordonnancement se fait uniquement lorsqu'un critère, permettant d'évaluer si un rééquilibrage de charge sera profitable, le stipule. L'arborescence est construite en fonction de l'architecture de la machine et de l'application exécutée, à l'aide de Charm++. Lors d'un rééquilibrage, il faut remonter les informations des branches au nœud racine pour qu'il connaissent la charge, les tâches sur les nœuds. Pour éviter de remonter toutes les informations, comme les temps de communications, les temps d'inactivité sur les processeurs, il existe une variable M qui limite la quantité maximale de données tolérée sur le parent. Pour respecter cette contrainte il est possible par de faire des groupes de tâches en fonction de leurs charges ou alors de supprimer certaines données qui sont considérées comme moins importantes. Finalement les informations remontent et le nœud racine peut commencer l'équilibrage de charge qui se propagera dans tous les niveaux. L'algorithmes RefineLB est utilisé pour les niveaux supérieurs car il permet de minimiser le nombre de tâches à déplacer entre les nœuds. Il cherche parmi les nœuds surchargé qu'elles tâches redistribuer à des nœuds sous-chargés pour équilibrer au mieux la charge. Pour la dernière couche de l'arbre qui est décomposée en groupe de processeurs, un groupe correspondant à un nœud on utilise l'algorithme centralisé LPT. Cet algorithme fonctionne très bien sur des applications itératives car il arrive à prédire correctement quand le rééquilibrage doit être fait à partir des données récupérées sur les nœuds. L'ordonnancement périodique a l'avantage de ne pas rajouter un sur-coût à l'exécution car il s'exécute de temps en temps. Cette solution est scalable à grande échelle et s'affranchit des problèmes que l'on rencontre avec les algorithmes d'ordonnancement distribués. En revanche, elle n'est pas applicable à StarPU qui est un STF et doit faire de l'ordonnancement dynamique. La complexité de cette solution n'a pas été étudiée en détail car ce contexte ne s'applique pas à StarPU et qu'elle dépend des algorithmes utilisés et de l'arborescence de l'arbre.

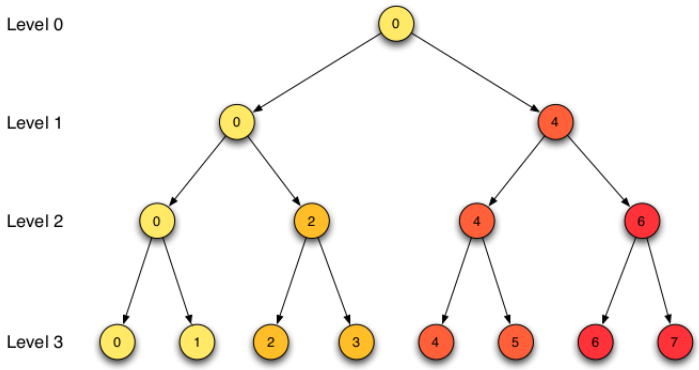


Fig. 1.8 – *Organisation hiérarchique sur 8 nœuds*

1.5 Problématique

Actuellement, l'équilibrage de charge distribué sur StarPU est pris en charge sur le modèle master-workers centralisé qui n'est pas scalable. Sur le modèle fully distributed qui est scalable, la distribution des données et l'équilibrage de charge doivent être fournis pas l'utilisateur. Sur le long terme, nous souhaiterions que StarPU prenne en charge ces aspects sur la version fully distributed. Pour cela, il faut trouver des algorithmes d'équilibrage de charge applicables dans un contexte d'ordonnancement dynamique et distribué. Ce modèle a la particularité d'attribuer les tâches sur les nœuds en fonction de la distribution des données, cela implique qu'il faille déplacer les données lors de l'équilibrage de charge. Or les algorithmes d'ordonnancement existants ne considèrent pas le lien entre les données et les tâches. L'objectif de ce stage est de comprendre comment utiliser les données pour faire de l'équilibrage de charge en distribué sur StarPU. Cette approche est nouvelle et complexe car nous cherchons une solution scalable, qui considère le lien tâches/données. En plus de ces contraintes, il faudrait que la solution fonctionne avec la version optimisée qui réalise l'élagage du graphe. A cela s'ajoute d'autres détails qui complexifient le problème comme la gestion du cache que je présenterai en Section 2.1.

Pour commencer à répondre à cette problématique, mon rôle est de proposer un environnement d'équilibrage de charge distribué en fonction des données pour StarPU. Pour cela nous proposons de réaliser un simulateur permettant de mieux comprendre le problème, en le modélisant et en explorant et adaptant des algorithmes d'équilibrage de charge.

Chapitre 2

Proposition

La solution pour aborder et comprendre le problème d'équilibrage de charge sur StarPU sur le modèle fully distributed est de réaliser un simulateur qui prend en charge l'équilibrage de charge distribué. Lors de l'ordonnancement, les données sont déplacées, ce déplacement induit une redistribution des tâches sur les nœuds. Cette approche est nouvelle est nécessite d'adapter des algorithmes d'équilibrage existants, premièrement car il n'existe pas d'algorithmes prenant en compte le lien entre les données et les tâches pour faire de l'ordonnancement dans la littérature et deuxièmement parce qu'il faut adapter les algorithmes existants au contexte dynamique et distribué de StarPU. Résoudre un problème d'équilibrage de charge minimisant le temps d'exécution total d'une application est un problème de la classe NP-difficile. Mais à cela s'ajoutent des contraintes qui rendent vite le problème difficile à traiter telle que les dépendances entre les tâches, l'optionnel élagage du graphe, le coût de déplacement des données ou les éventuels multiples accès en écriture sur une tâche. Toutes ces contraintes peuvent être prise en compte pendant l'ordonnancement mais cela aura pour conséquence d'augmenter fortement la complexité. Or nous voulons un algorithme avec une faible complexité car l'ordonnancement se fait pendant l'exécution. Pour éviter de traiter toutes les difficultés d'un coup, je vais implémenter un simulateur qui modélise le problème d'une manière simplifiée. Une fois que le simulateur sera opérationnel, on pourra envisager de traiter le problème dans son ensemble en ajoutant du grain à la simulation. Sur le long terme, le but est d'évaluer l'impact de ces options dans la qualité de la prise de décision et leur coût pour élaborer un algorithme d'équilibrage de charge efficace. Dans ce chapitre je vais présenter comment le rééquilibrage de charge devrait fonctionner dans l'idéal, ensuite j'expliquerai pourquoi nous abordons le sujet en tâtonnant sans avoir en vue une manière précise de régler le problème et enfin je proposerai un modèle simplifié permettant d'y voir plus clair.

2.1 Cas Idéal

Sur le long terme, l'objectif est d'avoir un équilibrage de charge distribué sur StarPU complètement distribué qui minimise le temps d'exécution de l'application en proposant un équilibrage de charge le plus précis possible et de l'autre coté, il faut que l'algorithme soit suffisamment simple pour qu'il s'exécute plus rapidement que les tâches car l'ordonnancement doit être fait au fil de l'exécution. Ce problème est compliqué car de nombreux paramètres sont à prendre en compte. D'abord, il faut considérer le type d'application que l'on souhaite ordonnancer. Un algorithme d'équilibrage de charge peut fonctionner pour un certain type d'application mais pas sur un autre.

De plus, si un algorithme est efficace pour ordonnancer tous types d'application il a forcément une forte complexité. Pour cette raison, il serait idéal de pouvoir choisir l'algorithme et les paramètres à utiliser pour l'ordonnancement dans le but de créer une situation adéquate à l'exécution de l'application.

Les dépendances entre les tâches sont également à prendre en compte car elles rajoutent de la complexité pour évaluer la charge sur un nœud. Prenons une tâche associée à un nœud, si elle ne satisfait pas ses dépendances, la véritable charge de travail du nœud ne comprend pas l'exécution de cette tâche car elle n'est pas exécutable. Pour éviter ce problème, StarPU distribue aux nœuds les tâches prêtes pour l'exécution. Lorsqu'il y a des dépendances, l'ordre d'exécution des tâches peut réduire ou augmenter le temps d'exécution de l'application. Par exemple, en exécutant en dernier une tâche au sommet de la chaîne de dépendances, certains nœuds vont se retrouver sans travail, à attendre la fin de l'exécution de la chaîne de dépendance. Le temps d'exécution de l'application sera plus long et les ressources n'auront pas été optimisées.

Les coûts des communications jouent aussi un rôle, un algorithme d'ordonnancement peut être très coûteux en communications ou une mauvaise répartition des données peut nécessiter un grand nombre de communications et rajouter de la latence. Les données à modifier lors de l'exécution de la tâche sont déjà connues car l'attribution des tâches se fait en fonction de ce facteur. En revanche, les données utilisées en entrées ne sont pas forcément sur le même nœud, il faut alors les demander au nœud propriétaire. Il peut être avantageux en fonction de l'application de mettre sur le même nœud, les données d'entrées et de sorties d'une tâche pour éviter un grand nombre de communications. Pour réduire les communications, il existe un cache qui stocke les données qui ont déjà été demandées aux autres nœuds. Si les données d'entrées n'ont pas été modifiées, alors le nœud n'a pas besoin de les redemander. Le fonctionnement du cache est illustré sur la Figure 2.2 qui se base sur la répartition de données et le DAG Figure2.1. Pour réduire encore les communications, une solution pourrait être de trouver la bonne combinaison d'exécution des tâches pour éviter d'avoir à répliquer le cache trop souvent.

Lors du rééquilibrage de charge, les données sont déplacées sur des nœuds, ce qui a pour conséquence de changer l'exécution des tâches. En partant de ce principe, une donnée peut être associée avec une charge de travail. Pour optimiser la charge sur les nœuds, il faut déplacer les bonnes données sur les bons nœuds. Ces migrations de données ont un coût. Mais dans certaines situation, déplacer une donnée peut amener à une diminution des communications. Par exemple en déplaçant une donnée et sa tâche sur un nœud qui possèdent toutes les données en entrées, la tâche peut être exécutée directement. Dans le cas particulier où une tâche écrit sur plusieurs données à la fois, il est possible de redistribuer la charge sans déplacer de données. Ce cas crée un sous problème d'ordonnancement consistant à associer les tâches à la donnée la moins chargée pour répartir au mieux la charge.

En plus de ces contraintes, il faut aussi considérer que chaque nœud possède une instance de StarPU intra-nœud qui fait déjà l'équilibrage de charge en son sein. Il faut que la solution proposée soit compatible avec cette version.

Pour finir, la solution idéale d'ordonnancement distribué fonctionnerait sur la version optimisée qui rajoute l'élagage. Il faudrait être capable de reconstruire le graphe de dépendance en fonction des migrations de données et ce problème est très complexe. Une solution alternative consisterait à créer des groupes de nœuds qui garderait un arbre commun et élaguerait la partie de l'arbre concernant les autres groupes de nœuds. Cela impliquerait que le rééquilibrage de charge se fasse uniquement au sein d'un groupe. Un nouveau problème se présente alors : comment faire les groupes

pour qu'ils soient équilibrés ?

Pour trouver le juste milieu entre un algorithme d'équilibrage de charge précis et une complexité convenable, il faut créer un environnement d'équilibrage de charge pour StarPU simplifié. Ensuite il faut rajouter des éléments plus complexe à la simulation via des options pour évaluer l'impact et l'intérêt de tous ces paramètres dans l'ordonnancement. Dans la partie suivante je présenterai le modèle que j'ai réalisé.

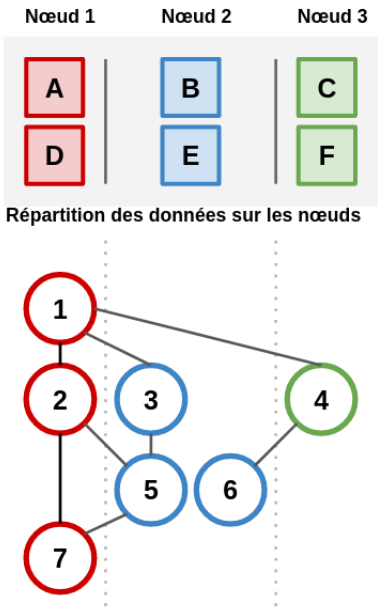


Fig. 2.1 – *Graphe de tâches et répartition des tâches sur les nœuds*

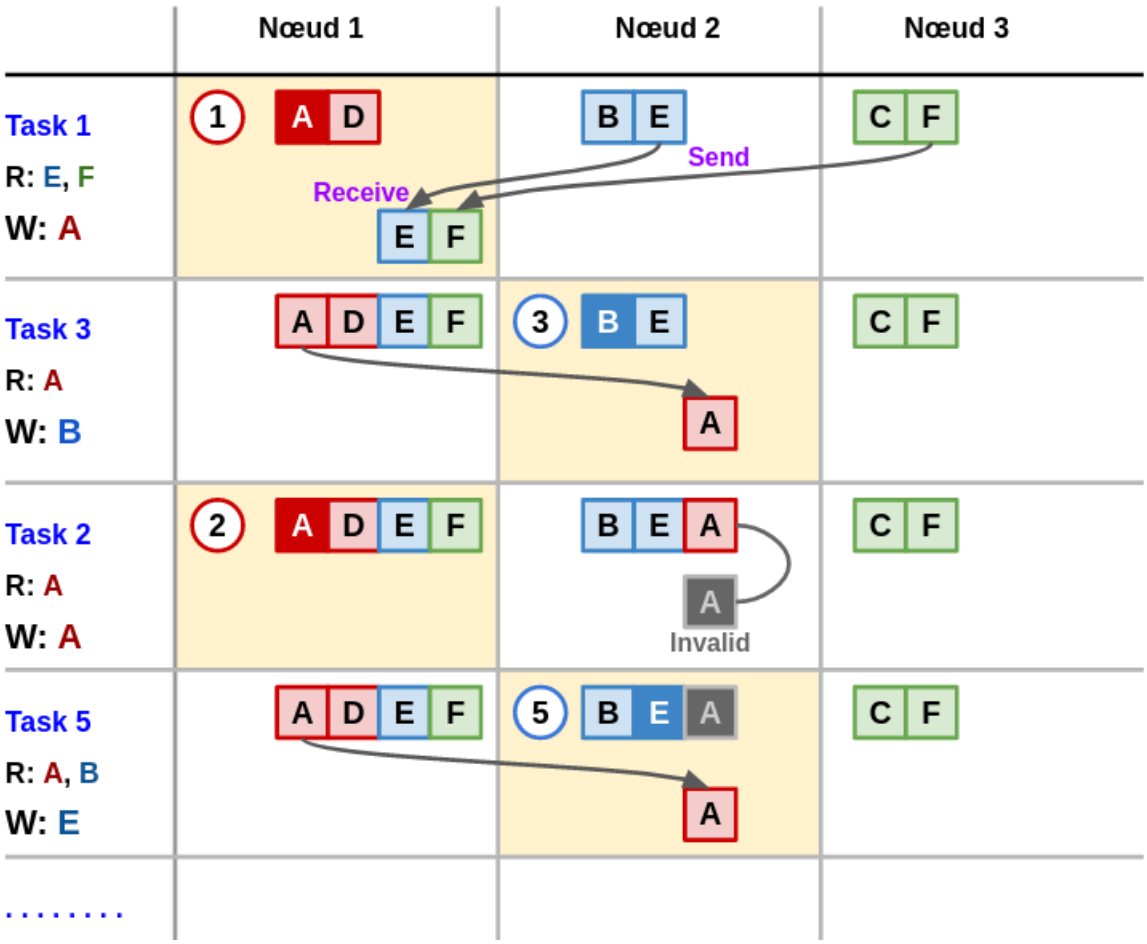


FIG. 2.2 – *Schéma de l'attribution et l'exécution des tâches sur les différents nœuds en prenant en compte la gestion de cache et les transferts de données. La tâche 1 doit être exécutée sur le nœud 1 car elle écrit sur A qui appartient au nœud 1, pour cela le nœud 1 demande les informations manquantes en lecture au nœuds 2 et 3. La tâche 3 est exécutée par le nœud 2 qui demande la donnée A au nœud 1, ensuite le nœud 1 modifie A en exécutant la tâche 2 et l'information connue par le nœud 2 devient obsolète. Il doit la redemander pour exécuter la tâche 5.*

2.2 Modélisation

J'ai d'abord réalisé un modèle distribué avec de l'ordonnancement centralisé. Faire l'ordonnancement de manière centralisé permet de vérifier le modèle avant de passer à un ordonnancement distribué où les informations sont partielles. Dans le simulateur, les tâches ne sont pas vraiment exécutées, la charge de travail est simulée avec des unités arbitraires. Les données et tâches sont représentées sur les nœuds et déplacées lors du rééquilibrage de charge. Le but étant d'avoir approximativement la même charge sur tous les nœuds. Ce modèle est une première approche qui sera améliorée en prenant en compte des cas plus compliqués.

Description du simulateur

- *Ordonnancement centralisé* : On choisit initialement de faire un ordonnancement centralisé car cela permet de travailler avec une seule instance du simulateur. La modélisations des échanges entre plusieurs instances du simulateur en distribué est remise à plus tard.

- *Distribution des données sur les nœuds selon une fonction* : La distribution des données se fait comme dans la version fully distributed de StarPU, l'utilisateur choisi avec un paramètre au lancement de l'exécution la distribution qu'il souhaite.

- *Attribution des tâches en fonction de leur sortie* : Le simulateur se concentre sur le lien qu'il y a entre les données et les tâches, en attribuant les tâches sur les nœuds exclusivement en fonction de leur unique sortie. Pour l'instant on suppose qu'une tâche écris sur une seule donnée en sortie.

- *Évaluation de la charge sur les nœuds et la machine* : La charge sur les nœuds est calculée avec une fonction de coût associée à une tâche qui évalue le temps d'exécution de la tâche en fonction de l'opération et de la taille des données sur lesquelles elle est réalisée. Les unités sont arbitraires.

- *Séquence de tâches* : Les tâches sont soumises sous forme de séquences ayant satisfaites leurs dépendances, l'ordonnancement est fait sur cette séquence. Pour simuler une tâche exécutée, on l'efface de la machine.

Simplifications Initiales

- *nœuds homogènes* : On considère uniquement des nœuds homogènes car sinon on devrait prendre en compte les différents noyaux possibles pour l'exécution d'une même tâche et la charge de travail serait plus complexe à évaluer.

- *Ignorer l'ordonnancement intra-nœud* : L'ordonnancement au sein d'un nœud est géré par StarPU intra-nœud. Cette partie est fonctionnelle et intervient une fois que l'équilibrage de charge distribué sur la machine a été réalisé. Cet aspect n'a pas besoin d'être pris en compte pour le moment, mais à terme il faudrait le prendre en compte.

– *Ignorer les dépendance de données* : Il existe deux cas : soit des séquences de tâches prête à l'exécution sont soumises, soit les dépendances ne sont pas prises en compte et on traite toutes les tâches de l'application en une fois. Dans tous les cas l'ordonnanceur n'a pas à ce soucier de la dépendance des tâches. Dans le cas numéro 2 on a une estimation de la durée que pourrait prendre l'exécution des tâches dans le meilleur cas, cette représentation est appelée ABE (Area Bound Estimation).

– *Ignorer les communications* : Pour la première version l'ordonnancement se fait de manière centralisé pour éviter d'avoir à modéliser les communications. On considère que toutes les informations sont connues par les nœuds et qu'ils n'ont pas besoin de les demander. Avec une instance du simulateur par nœud les communications seront obligatoires pour échanger des informations.

– *Ignorer le coût lié au déplacement des données* : Les déplacements des données sont aussi des communications, comme elles ne sont pas prises en compte, le coût de déplacement des données non plus. Dans la version avec des communications on évalue le coût de déplacement en fonction de la taille de la communication.

– *Ignorer les possibles multiples sorties sur une tâche* : Une tâche peut avoir plusieurs accès sur des données en sortie, pour éviter de prendre en compte des cas particulier dans l'attribution des tâches et lors du déplacement des tâches, on considère que chaque tâche à une seule sortie.

– *Ignorer les surcoûts liés aux données en entrées d'une tâche* : Avec l'ordonnancement centralisé les données sont connues par l'ensemble des nœuds qui n'ont pas besoin des les demander si elles ne sont pas sur leur nœud. Vu que les communications ne sont pas modélisées, on ne peut pas évaluer le sur-coût liée à l'exécution d'une tâche qui soit demander des données. En plus, les dépendances ne sont pas modélisées donc une donnée pourrait être modifiée plusieurs fois, cela n'aura pas d'impact sur l'exécution des tâches : une fois que la donnée est connue par le nœud c'est pour la vie même si elle est modifiée entre temps. Pour toutes ces raisons on ne considère pas les sur-coûts liées aux données en entrées.

– *Pas de cache* : Les tâches n'ont pas besoin de récupérer des données en lecture pour s'exécuter, en partant de ce principe le cache n'a plus d'intérêt. Il n'est pas modélisé pour l'instant.

– *Ignorer la réplication du cache* : La réplication du cache pourra être modélisée une fois que les dépendances seront prises en compte. Comme expliqué dans le point Ignorer les sur-coûts liés aux données en entrées d'une tâche.

Chapitre 3

Éléments d'Implémentation

Le simulateur est implémenté en python, il suit la modélisation décrite dans la partie précédente. Actuellement, la version ordonnancement distribué est en cours de développement mais la version ordonnancement centralisée est terminée. Pour pouvoir tester le simulateur, il a fallu créer des applications à ordonnancer et adapter des algorithmes de la littérature à notre contexte.

3.1 Représentation du Modèle

Le simulateur est décomposé en classes qui représentent les principaux éléments présents dans la simulation. L'utilisateur lance l'exécution de l'application sur le simulateur avec les options choisies en ligne de commande. Dans l'application le runtime est instancié en fonction de ces options, les données et codelets sont instanciés sur le runtime, suivi par les tâches qui sont créées à partir d'un codelet et de données. Ensuite les tâches sont distribuées sur les nœuds puis, les tâches assignées aux nœuds en fonctions des données. Ensuite on peut évaluer la charge de travail sur la machine et demander un rééquilibrage de charge. La Figure 3.1 illustre ce mécanisme avec une application simple ayant une donnée un codelet et une tache et un nœud. Les classes sont décrites de manière plus détaillée ci-dessous.

- **Machine** : La classe machine représente la machine et permet de définir le nombre de nœuds dans le simulateur.
- **Data handle** : Les données représentent les données utilisées pour l'exécution de l'application, ils ont un nom, une taille et un propriétaire. Pour les déplacer, il suffit de changer leur propriétaire.
- **Codelet** : Les codelets représentent les noyaux de calculs, ils ont un nom, une fonction de coût qui permet d'évaluer la quantité de travail que le codelet représente et une liste des accès lecture et écriture nécessaire à l'utilisation du noyaux de calcul. Comme on n'exécute pas réellement les tâches dans la simulation, le codelet n'a pas de fonction avec les instructions à réaliser.
- **Tâche** : Les tâches sont un codelet appliqué sur des données. Dans la simulation, elles ont un numéro unique qui permet de les différencier, un codelet et une liste de données. L'évaluation de la charge de travail est faite en fonction des tâches. Leur coût d'exécution est calculé en utilisant la fonction de coût dans leur codelet appliquée sur leurs données.

– **Algo** : La classe algo est la classe mère des différents algorithmes utilisés pour l'équilibrage de charge. Elle permet d'obtenir la charge que représente une donnée et de classer les nœuds en fonction de leur charge de travail. Dans les classes filles sont décrites les méthodes pour chaque algorithme.
– **Runtime** : Le runtime permet d'articuler tous les éléments entre eux, il crée et sert d'intermédiaire entre l'utilisateur et tous les objets de la simulation. Il est initialisé à partir des paramètres donnés en ligne de commande. Le runtime connaît la situation dans son ensemble.

```
1  import source.runtime as rt
2  from source.task import *
3  import source.__init__
4
5  runtime = rt.Runtime(0, 1)
6  dh = runtime.create_data_handle('data 01', 0, 42)
7  cl = runtime.create_codelet('codelet 01', [cod.RW]
8  task = runtime.submit_task(cl, [dh])
9
10 runtime.put_data_on_node_2Dcyclic()
11
12 runtime.assign_task_to_node(task)
13
14 runtime.workload_on_machine()
15
16 runtime.rebalancing()
```

FIG. 3.1 – *Exemple simple d'application*

3.2 Algorithmes

Cette section présente les algorithmes implémentés sur le simulateur. Ils s'inspirent de la littérature mais ont été adaptés au contexte. Le simulateur est dans un contexte d'ordonnancement centralisé car la version distribué n'est pas encore finalisée. Les algorithmes qui ont été implémentés durant mon stage sont eux aussi centralisés.

3.2.1 Éléments pour mettre en place l'ordonnancement à base de tâches

L'ordonnancement se fait en déplaçant des données et non pas des tâches. Il faut alors trouver un moyen de déplacer ensuite les tâches liées à cette donnée. Chaque donnée est sur un nœud avec les tâches qui ont un lien avec elle en sortie. Pour les trouver, on parcourt les tâches sur son nœud et on vérifie si elles écrivent sur cette donnée, si oui on ajoute la tâche dans une liste, la fonction *Runtime::find_tasks_associated_to_1data* s'occupe de cela. Ensuite les tâches sont déplacées avec la fonction *Runtime::redistribute_task* du runtime. Pour récupérer la charge de travail associée à une donnée, on utilise la fonction *Runtime::find_tasks_associated_to_1data*, pour obtenir la liste

des tâches associées à une donnée. Ensuite, il suffit de sommer la charge de chaque tâche pour connaître la charge d'une donnée, c'est le rôle de la fonction *Algo::find_data_load*.

La charge sur un nœud est calculée en fonction du coût total des tâches présentes sur ce nœud dans la fonction *Runtime::work_load_on_node*. En répétant cette opération sur tous les nœuds on obtient la charge moyenne sur la machine (*Runtime::average_load*). En fonction de cette moyenne, les nœuds sont classés en différentes catégories comme dans l'algorithme PackstealLB. Il y a trois catégories de nœuds : surchargé (over), sous-chargé (under) et neutre. La catégorie neutre est optionnelle, elle permet de ne pas rééquilibrer des nœuds qui n'ont pas exactement la même charge que la moyenne mais en sont quand même proche. Pour cela, j'ai mis en place un seuil de tolérance qui considère les nœuds au-dessus de ce seuil comme over et les nœuds en dessous comme under. Le seuil est ajouté à la moyenne pour créer un intervalle. Il peut être donné par l'utilisateur ou alors il représente un pourcentage de la charge totale sur le nœud, généralement 10%. Le fonctionnement du seuil est illustré par la Figure 3.2.

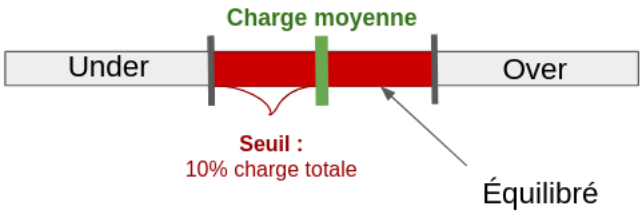


FIG. 3.2 – *Différentes catégories de nœuds en fonction du seuil*

3.2.2 Adaptation du List Scheduling

Dans le List Scheduling, les tâches dans une liste sont distribuées aux différents processus. Dans notre contexte, nous n'avons pas de liste de données à ordonnancer. Nous connaissons seulement une liste de nœuds surchargés et une liste de nœuds sous-chargés. Pour adapter cet algorithme, je considère que la liste de tâches est notre liste de nœuds à rééquilibrer et j'enlève les premières données de chaque nœud à rééquilibrer tant que le nœud est toujours surchargé. L'algorithme 1 est l'adaptation de list scheduling à StarPU. On cherche les nœuds à rééquilibrer, on parcourt la liste de nœud surchargés. Tant qu'un nœud est surchargé, on enlève la première donnée sur le nœud nommée data1, on déplace les tâches associées à cette donnée et on met à jour les charges des nœuds.

Algorithme 1 : List Scheduling

Data: *Runtime*, *threshold*
Result: *Runtime*
$Over$, $Under \leftarrow$ **find_over_under_load_nodes**$(Runtime, threshold)$
$min = \emptyset$
for all $over \in Over$ **do**
    while *over* is overloaded **do**
        $min \leftarrow$ **min**$(Under)$
        Runtime $\leftarrow$ Move the first data in *over* named $data1_i$ on the node *min*
        Runtime $\leftarrow$ Redistribute tasks associated with $data1_i$
        $Over$[over] $\leftarrow Over$[over]-**find_data_load**$(Runtime, data1_i)$
        $Under$[min] $\leftarrow Under$[min]+**find_data_load**$(Runtime, data1_i)$
return *Runtime*

Complexité : Soit d le nombre de données dans l'application, t le nombre de tâches et n le nombre de nœuds, avec $n \leq d \leq t$. Chaque nœud a en moyenne $\frac{d}{n}$ données et $\frac{t}{d}$ tâches qui écrivent sur ces données. Je suppose que les informations telles que la charge sur chaque nœud ou quelles tâches sont associées à quelles données, sont stockées dans des structures de données pour éviter d'avoir à faire le calcul. Le coût de ces informations est $O(n+t)$. La boucle coûte $O(n)$, le while coûte $O(\frac{t}{d})$, trouver le nœud le moins chargé coûte $O(log(n))$, déplacer des données et des tâches coûte $O(1 + \frac{t}{d})$ et mettre à jour les charges des nœuds coûte $O(2)$.
Le calcul coûte alors $O(d \times (3 + \frac{t}{d} + log(n)))$ ce qui donne $O(t + d \times log(n))$. Cette complexité est raisonnable et convient pour notre problème.

3.2.3 Adaptation du LPT

L'algorithme LPT2 est la version adaptée du LPT à StarPU. Le LPT est la version optimisée du list scheduling, au lieu de prendre les tâches dans l'ordre de la liste, on cherche la tâche la plus volumineuse pour la placer en premier. Nous rencontrons le même problème qu'avec list scheduling car nous n'avons pas de liste de tâches ou de données à ordonnancer, mais nous ne pouvons utiliser la même technique car il faut connaître la liste des données à déplacer avant de procéder à l'ordonnancement. Il faut trouver un moyen d'obtenir une liste de données à déplacer. Sur chaque nœud over, il faut retirer des données et les ajouter dans la liste puis faire l'ordonnancement. Il existe plusieurs moyens de construire la liste. Trois techniques ont été explorées :

1. **Déplacer les données les moins chargées : Light**
   On choisi la donnée la moins chargée, elle est ajoutée à la liste des données à déplacer et on enlève sa charge à la charge du nœud. La situation est réévaluée pour voir s'il faut encore déplacer des données. Cette technique a un intérêt si les nœuds sont assez peu déséquilibrés et ont des charges assez proches. Ou pour alléger un nœud qui aurait une donnée qui représente une énorme partie de sont travail, en déplaçant les petites tâches on peut trouver l'équilibre alors que la grosse aurait surchargé tout les autres nœuds. En revanche, si toutes les tâches ont la même taille cette approche n'est pas intéressante.
2. **Minimiser le nombre de données à déplacer : Min**
   Pour cette algorithme, je me suis inspirée de bruteforce : je cherche la plus petite combinaison de données qui équilibrera le nœud. Cette résolution peut vite prendre du temps, avec un nœud

très déséquilibré, la solution risque d'être longue à calculer. Elle a l'avantage de déplacer le minimum de données possible ce qui peut être un avantage en fonction des coûts liés aux déplacements.

3. **Déplacer les données seulement si cela améliore l'équilibrage de charge sur le nœud : Worth_it**
   Cette solution ressemble le plus au list scheduling, on parcourt les données dans l'ordre, on regarde si en déplaçant une donnée sur le nœud le moins chargé la situation globale est meilleure. Si oui, on enlève définitivement la donnée du nœud et on le place dans la liste, le nœud qui a récupéré la donnée de manière fictive garde sa nouvelle charge pour essayer d'estimer la situation. Cette technique fonctionne lorsque aucun déplacement de données ne pourraient améliorer la charge de travail sur les nœuds.

Une fois que la liste a été récupérée, j'applique le LPT. On trie les nœuds en fonction de leur charge, on crée la liste de données à déplacer à partir des nœuds surchargés. La liste est créée avec une des options présentées ci-dessus. La fonction *find_data_to_move_for_balance* représente les différentes possibilités. La donnée le plus volumineuse est placée sur le nœud le moins chargé, on fait les déplacements et on met les charges à jour. On répète jusqu'à ce que la liste soit vide.

Algorithme 2 LPT StarPU

Data: *Runtime*, *threshold*
Result: *Runtime*
$data\_to\_remap = \emptyset$, $data\_to\_move = \emptyset$
$Over$, $Under \leftarrow$ **find_over_under_load_nodes**$(Runtime, threshold)$
for all $over \in Over$ **do**
    $data\_to\_move \leftarrow$ **find_data_to_move_for_balance**$(Runtime, over)$
    Add $data\_to\_move$ to $data\_to\_remap$
while $data\_to\_remap \neq \emptyset$ **do**
    $moving\_data \leftarrow$ **max**$(data\_to\_remap)$
    $min \leftarrow$ **min**$(Under)$
    $Runtime \leftarrow$ Move $moving\_data$ on the node min
    $Runtime \leftarrow$ Redistribute tasks associated with $moving\_data$
    $Under$[min] $\leftarrow Under$[min]-**find_data_load**$(Runtime, moving\_data)$
    Delete $moving\_data$ from $data\_to\_remap$
return *Runtime*

Complexité : En reprenant les mêmes notations et conditions que dans la complexité au dessus. L'ordonnancement de la liste coûte : Le while coûte *O(d)*, le maximum coûte *O(log(d))*, le minimum coûte *O(log(n))*, les déplacements des données et des tâches coûtent $O(1+\frac{t}{d})$, mettre à jour la charge du nœud under coûte *O(1)*. Cela fait $O(d \times (log(n)) + log(d) + \frac{t}{d} + 2)$, en simplifiant on obtient $O(t + d \times log(n) + d \times log(d))$.
Il faut en plus ajouter la complexité pour obtenir la liste tâche : la complexité de *Light* est : $O(d \times log(d))$, celle de *Min* est $O(n \times (\frac{d}{n})!)$ et celle de *Worth_It* est $O(d \times log(n))$.

3.3 Options techniques

3.3.1 Simulateur distribué

Nous avions initialement envisagé d'implémenter le modèle distribué avec des coroutines de la bibliothèque asyncio[5] de python. Finalement, les tests ont montré qu'adapter le code avec les coroutines n'était pas suffisamment abstrait et impliquait de modifier tout le code qui avait déjà été réalisé. Cette solution étant trop contraignante, nous avons donc décidé d'utiliser mpi4py [11] une bibliothèque python qui permet de faire du MPI et donc de gérer les communications entre les différentes instances de runtime qui symbolisent les nœuds.

Le simulateur est en cours d'adaptation pour se tourner vers le modèle avec de l'ordonnancement distribué. Pour l'instant, on peut lancer différents processus MPI qui instancient leur runtime avec mpi4py, les runtimes représentent les différentes instances de StarPU, comme dans le modèle fully distributed. Les processus MPI peuvent communiquer entre eux, mais chaque processus via le runtime a toujours une vision globale de la situation. Il reste à ajouter des fonctions pour donner une vision locale à chaque processus, les données et tâches doivent être distribuées sur un seul processus, les autres nœuds ne doivent pas être connus. Pour exécuter le simulateur en distribué, il faut le préciser avec une option en ligne de commande.

3.3.2 Paramètres en ligne de commande

Le simulateur a deux types de paramètres : ceux qui concernent l'application et ceux qui concernent le simulateur comme l'algorithme d'ordonnancement sélectionné ou le nombre de nœuds dans la machine. Tous les paramètres ont une valeur par défaut.

Paramètres du simulateur

- *Version* : Permet de sélectionner si le simulateur est en centralisé ou distribué.
- *Session size* : Permet de sélectionner le nombre de nœuds sur le simulateur.
- *Seuil* : Permet de sélectionner le seuil, soit en le précisant en dur soit en lui attribuant un pourcentage de la charge sur la machine.
- *Algorithme* : Permet de sélectionner l'algorithme d'ordonnancement.
- *liste* : Cette option est particulière au LPT, elle permet de choisir la manière dont la liste va être réalisée.
- *Affichage* : Permet d'afficher les résultats lors de l'exécution.
- *Debug* : Permet d'afficher encore plus de détails pour débugger.

Les paramètres relatifs à l'application seront détaillés dans la partie suivante qui présente les différentes applications réalisées et montre les différents rééquilibrage de charge possibles en fonction de l'algorithme sélectionné.

Chapitre 4

Validation de la Simulation et Résultats

Dans cette partie, je vais présenter les différentes applications que j'ai créées pour tester le simulateur. Ensuite, je présenterai un exemple d'application à rééquilibrer et je montrerai l'impact des différents algorithmes sur l'équilibrage d'une application. Pour valider le simulateur, je n'ai pas eu le temps d'intégrer des tests unitaires, mais j'ai vérifié le bon déroulé de l'exécution de mes algorithmes sur les applications.

4.1 Les différentes applications

Différentes applications

- *Stencil* : À chaque itération, on calcule chaque cellule du maillage. La tâche associée à ce calcul dépend en entrée des cellules voisines. Les tâches pour les cellules aux bords ne sont pas exactement les mêmes.
- *Stencil_diagonal* : On reprend le principe de l'application précédente mais en rajoutant des tâches supplémentaires sur la diagonale pour créer un déséquilibre de charge.
- *Unbalanced* : Sur cette application, on peut changer la taille des données pour créer un déséquilibre de charge, chaque donnée a trois tâches, sauf le premier qui en a une et le deuxième deux. Il y a trois type de tâches : une qui n'a aucun accès lecture, une qui utilise la donnée précédente en lecture et une qui utilise les deux données précédentes en lecture, la charge des tâches est calculée en fonction de la taille des données.

Paramètres des applications

- *Taille données* : Permet de sélectionner la taille des données.
- *Facteur augmentation de la taille* : Permet d'augmenter la taille des données au fur et à mesure pour créer une situation déséquilibrée.
- *Distribution des données* : Permet de distribuer les données sur les nœuds. On peut faire une distribution 2D-bloc-cyclique, aléatoire, ou tout mettre sur un nœud.
- *Nombre données* : Permet de choisir le nombre de données dans l'application.
- *Taille du maillage* : Permet de choisir la taille du maillage dans les applications de type stencil.

– *Fonction de coût* : Permet de sélectionner le fonction de coût pour évaluer la charge des tâches. La fonction de coût peut prendre en compte la taille des données ou bien le nombre.

4.2 Rééquilibrage sur un exemple d'application

J'ai schématisé l'exécution d'une application avec les algorithmes d'équilibrage de charge que j'ai adapté pour illustrer ce que fait concrètement le simulateur et montrer leur fonctionnement. L'application qui va être rééquilibrée est *unbalanced*, la Figure 4.1 représente la situation à rééquilibrer. Elle s'exécute sur 3 nœuds, possède 8 données qui ont toutes une taille différente : chaque donnée est plus grosse de 8ua (unités arbitraire) par rapport à son prédécesseur (en violet). La distribution des données est faite aléatoirement et est très déséquilibrée. La charge de travail en fonction des données, représentée en orange, dépend de la taille de la donnée et du nombre de tâches qui lui sont associées.

La charge moyenne sur la machine est de 1208ua, le seuil est de 120ua donc si un nœud est entre 1088ua et 1328ua, il est équilibré. Dans notre exemple, le **nœud 1 est surchargé** il a 2998ua de charge et les **nœuds 0 et 2 sont sous-chargés** avec 166ua et 462ua de charge respective.

On va procéder à l'ordonnancement avec les algorithmes list scheduling et LPT et toutes ses variantes pour créer des listes.

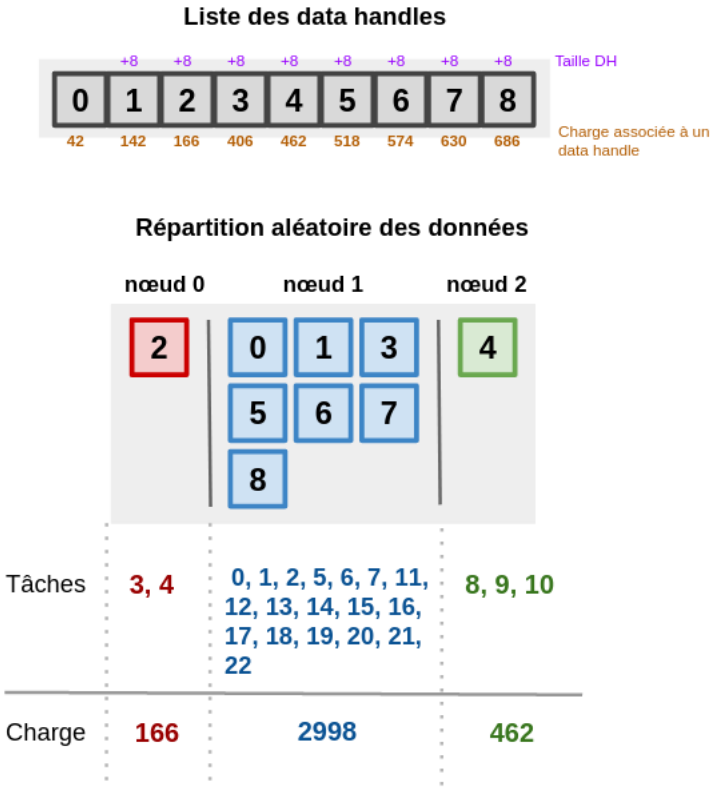


FIG. 4.1 – *Situation à équilibrer*

4.2.1 Ordonnancement avec List Scheduling

Je vais présenter l'exécution pour arriver au résultat montré dans la Figure 4.2. List scheduling parcourt les données dans l'ordre sur le nœud et les place sur le nœud le moins chargé. Au début de l'exécution le nœud 0 est le moins chargé, il récupère tous les données jusqu'à ce que le nœud 2 devienne le moins chargé. Les charges cumulées des données 0, 1 et 3 font 590, avec cette nouvelle charge le nœud 0 passe à 756ua de charge. La donnée 5 qui vaut 518ua est donné au nœud 2 ainsi sa charge devient 980ua et le nœud 1 passe à 1890ua de charge. Il est toujours au dessus du seuil de 1328ua pour être considéré comme équilibré, la donnée 6 est donnée au nœud 0 qui passe à 1330ua et le nœud 1 passe à 1316ua en dessous du seuil. L'algorithme s'arrête lorsque la situation est équilibrée. Le nœud 0 est maintenant considéré comme surchargé de 2 par rapport au seuil de tolérance et le nœud 2 est sous-chargé de 108ua.

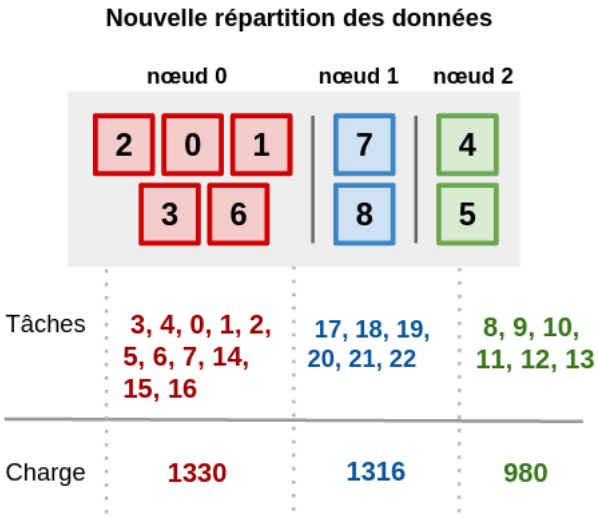


FIG. 4.2 – *Rééquilibrage de charge de l'application avec List Scheduling*

4.2.2 Ordonnancement avec LPT et tri avec les fonctions worth_it et light

Pour fonctionner l'algorithme LPT nécessite une liste de données. Les fonctions pour obtenir les différentes listes ont été présentées en Section 3.2.3. La liste créée avec *light* prend la plus petite donnée du nœud jusqu'à ce qu'il soit équilibré. La liste créée avec *worth_it* parcourt les données sur le nœud dans l'ordre et prend les éléments tant que cela améliore la situation globale. Or les données sont rangées de manière croissante sur le nœud, ces deux méthodes donnent alors la même liste. La Figure 4.3 montre le résultat obtenu avec le LPT et le tri de liste *worth_it* et *light*. La donnée 6 est ordonnancée en premier car c'est elle le plus lourde, elle va sur le nœud 0, la donnée 5 va sur le nœud 2 qui est devenu le nœud le moins chargé. La donnée 3 est mise sur le nœud 0 et le reste part sur le nœud 2. On arrive a une charge de 1146ua sur le nœud 0, de 1316ua sur le nœud 1 et de 1164ua sur le nœud 2. La situation est mieux équilibrée qu'avec list scheduling, tous les nœuds sont équilibrés.

FIG. 4.3 – *Rééquilibrage de charge avec LPT, la liste est obtenue avec l'algorithme worth_it ou ight*

4.2.3 Ordonnancement avec LPT et tri avec la fonction min

La liste faite avec la fonction *min* trouve la combinaison permettant d'équilibrer le nœud en déplaçant le moins de données possible. En prenant les données les plus chargées sur le nœud 1, on a trois données à ordonnancer. La donnée 7 est placée sur le nœud 0 qui est le moins chargé, ensuite la 6 sur le nœud 2 et la 5 sur le nœud 0 encore une fois. Le nœud 2 est considéré comme sous-chargé de 52ua par rapport au seuil, mais il n'y a pas de nœud surchargé donc la situation est satisfaisante. La charge de la machine est mieux équilibrée qu'avec le liste scheduling, mais moins bien qu'avec *worth_it* ou *light*. Ce résultat est expliqué par la charge élevée des données à redistribuer. Avoir des données moins chargées dans la liste permet d'avoir un équilibrage plus fin.

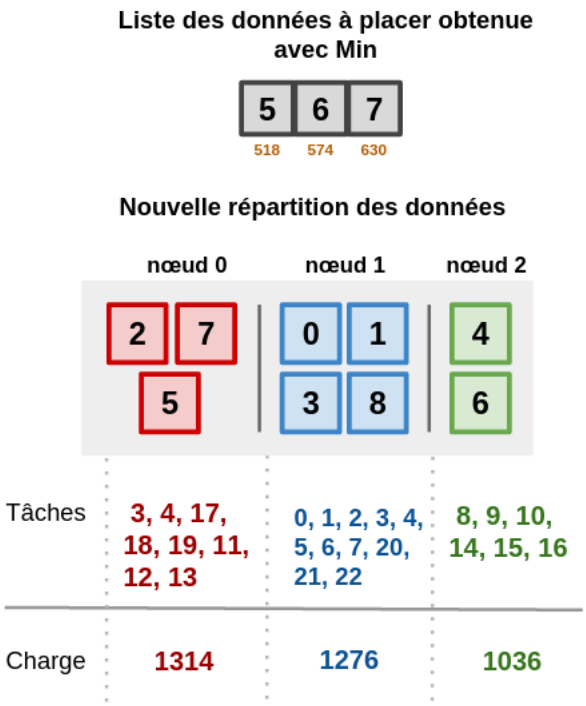


FIG. 4.4 – *Rééquilibrage de charge avec LPT, la liste est obtenue avec l'algorithme Min*

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Les support d'exécution sont des outils permettant la bonne exécution parallèle des applications, ils sont chargés de répartir le travail sur la machine. Il permettent aussi une portabilité des applications sur différents types d'architecture. StarPU est un support d'exécution à base de tâches développé par STORM, il fonctionne en intra-nœud et en inter-nœud. La version inter-nœud propose deux modèles : le modèle master-workers qui n'est pas scalable en fonction du nombre de nœuds et le fully distributed qui est scalable mais ne prend pas en charge la distribution des données sur les nœuds et l'équilibrage de charge. Actuellement, l'utilisateur doit préciser ces aspects dans le code de l'application. L'objectif à long terme serait de pouvoir faire le rééquilibrage de charge en fully distributed. Ce problème est complexe car il faut trouver une solution qui fonctionne dans le cadre d'un ordonnancement dynamique et distribué et considère le lien particulier qu'il y a entre les tâches et leur donnée en sortie. Les algorithmes dans la littérature ne prennent pas en charge ce lien entre les données et les tâches. À cela s'ajoute d'autres éléments qui complexifient le problème comme la version optimisée qui réalise un élagage du graphe ou la gestion du cache. Pour y voir plus clair et dégrossir le problème, j'ai proposé un simulateur qui fournit un simulateur qui fournit un environnement d'équilibrage de charge distribué sur StarPU en prenant en compte le lien entre les données et les tâches. J'ai conçu et implémenté un modèle préliminaire simplifié. Il réalise l'équilibrage de charge dans un contexte centralisé et est encours d'évolution vers un contexte distribué et affiné au fil du stage. La version centralisée du simulateur a été réalisée et fonctionne avec un ordonnancement qui prend en compte le lien entre les données et les tâches. Pour cela j'ai attribué aux données une charge en fonction des tâches qui leurs sont associées. J'ai ensuite adapté des algorithmes d'ordonnancement de la littérature au contexte de StarPU pour qu'ils prennent en compte ce lien entre les tâches et les données. J'ai également conçu des applications paramétrables pour exécuter et tester mon simulateur. Les résultats obtenus montrent qu'il est possible de faire de l'ordonnancement en prenant en compte à la fois les tâches et les données. La version distribuée n'est pas encore complète et nécessite encore du travail. Pour l'instant chaque processus MPI crée une instance du simulateur en centralisé, le travail n'est pas encore réparti entre les différents processus et chaque nœud n'a pas une vision locale de la situation.

5.2 Perspectives

Sur le moyen terme, il faudrait mettre en place des fonctions pour distribuer les tâches et les données en fonction de l'ID des processus MPI pour que chaque nœud ait une vision locale. Comme mettre en place l'ordonnancement distribué tout bénéficiant de la scalabilité qu'offre l'élagage du graphe était compliqué, nous avons pensé créer des groupes de processus MPI qui représenteraient des groupes de nœuds. Dans chaque groupe il y aurait un DAG élagué en fonction des nœuds du groupe, ainsi nous pourrions faire l'équilibrage de charge au sein des groupes sans problème. Les données et tâches ne pourraient donc pas sortir du groupe. Cette solution permet de conserver les bénéfices de l'élagage en évitant que les nœuds aient à connaître l'intégralité du DAG. Elle nécessite de mettre en place un moyen de faire des groupes déjà équilibrés sinon le rééquilibrage ne servira pas à grand chose. Il faut aussi implémenter des algorithmes d'équilibrage de charge distribués en les adaptant au contexte. Pour générer des applications, nous avions pensé récupérer des traces d'exécution d'applications avec StarPU et recréer l'application avec ces traces, nous pourrions ainsi tester nos algorithmes sur des vraies applications StarPU. Sur le long terme, le but serait d'affiner la simulation au maximum pour voir quels paramètres sont importants à prendre en compte pour avoir un équilibrage de charge précis et efficace. Ensuite il faudrait implémenter l'équilibrage de charge dans StarPU fully distributed. Avant d'avoir l'équilibrage de charge dans StarPU, nous pourrions utiliser la simulation pour le faire soit en parallèle de l'exécution ou soit avant l'exécution en overhead. Dans ce cas, le simulateur devra être adapté pour envoyer à StarPU les données à déplacer et leur destination.

Bibliographie

[1] Emmanuel Agullo et al. "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model". In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: 10.1109/TPDS.2017.2766064. URL: https://hal.inria.fr/hal-01618526.

[2] Anne Benoit et al. "Update on the Asymptotic Optimality of LPT". en. In: *Euro-Par 2021: Parallel Processing*. Ed. by Leonel Sousa, Nuno Roma, and Pedro Tomás. Vol. 12820. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 55–69. ISBN: 978-3-030-85664-9 978-3-030-85665-6. DOI: 10.1007/978-3-030-85665-6_4. URL: https://link.springer.com/10.1007/978-3-030-85665-6_4 (visited on 08/09/2022).

[3] Petra Berenbrink et al. "Distributed Selfish Load Balancing". en. In: *SIAM Journal on Computing* (Nov. 2007). Publisher: Society for Industrial and Applied Mathematics. DOI: 10.1137/060660345. URL: https://epubs.siam.org/doi/10.1137/060660345 (visited on 08/09/2022).

[4] George Bosilca et al. "Scalable dense linear algebra on heterogeneous hardware". In: *Advances in Parallel Computing* 24 (Jan. 2013), pp. 65–103. DOI: 10.3233/978-1-61499-324-7-65.

[5] *Coroutines and Tasks — Python 3.10.6 documentation*. URL: https://docs.python.org/3/library/asyncio-task.html (visited on 08/15/2022).

[6] M. Cosnard and M. Loi. "Automatic task graph generation techniques". In: *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. Vol. 2. Jan. 1995, 113–122 vol.2. DOI: 10.1109/HICSS.1995.375471.

[7] Vinicius Freitas et al. "PackStealLB: A scalable distributed load balancer based on work stealing and workload discretization". In: *Journal of Parallel and Distributed Computing* 150 (Apr. 2021). Publisher: Elsevier, pp. 34–45. DOI: 10.1016/j.jpdc.2020.12.005. URL: https://hal.archives-ouvertes.fr/hal-02405735 (visited on 08/09/2022).

[8] *Home - — TOP500*. URL: https://www.top500.org/ (visited on 08/11/2022).

[9] Emmanuel Jeannot et al. "Communication and topology-aware load balancing in Charm++ with TreeMatch". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. ISSN: 2168-9253. Sept. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702666.

[10] *June 2022 — TOP500*. URL: https://www.top500.org/lists/top500/2022/06/ (visited on 08/10/2022).

[11] *MPI - HPC Wiki*. URL: https://hpc-wiki.info/hpc/MPI (visited on 08/11/2022).

[12] Lucas Leandro Nesi, Lucas Mello Schnorr, and Arnaud Legrand. "Communication-Aware Load Balancing of the LU Factorization over Heterogeneous Clusters". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. ISSN: 2690-5965. Dec. 2020, pp. 54–63. DOI: 10.1109/ICPADS51040.2020.00017.

[13] tim.lewis. *Specifications*. en-GB. URL: https://www.openmp.org/specifications/ (visited on 08/11/2022).

[14] Gengbin Zheng et al. "Periodic hierarchical load balancing for large supercomputers". en. In: *The International Journal of High Performance Computing Applications* 25.4 (Nov. 2011). Publisher: SAGE Publications Ltd STM, pp. 371–385. ISSN: 1094-3420. DOI: 10.1177/1094342010394383. URL: https://doi.org/10.1177/1094342010394383 (visited on 08/10/2022).

Annexes

Description des algorithmes

Pélagie ALVES

May 2022

1 Définitions

1.1 Les variables

1.1.1 Les différents éléments/objets dans la simulation

node : A part of the machine which has its own storage and compute space.
data_handle : A object to represente data with an id, an node owner and a size.
codelet : Is a list of Read Write acces and a cost function.
task : A task is a list of data handle apply on a codelet.
runtime : An object which know everything on the machine, the number of nodes, data handles and tasks. As well as the distribution of these objects on the different nodes and their load.

1.1.2 Variables pour désigner les éléments

N_i : Represente the node N_i with $i \in (0, size)$
size : Number of nodes on the runtime
$data_d$: The data handle indice d, $d \in (0, nb\_data)$
$data\_N_i$: The data handles on N_i
$data1_i$: Is the first data handle on the node i.
data_to_remap : A dictionary of data handle to move for balancing. The key is a data handle and the value is the load of the data handle.
data_to_move : Is a dictionary used to build data_to_remap.
nb_data_i : $Number of data handle on the node i$
moving_data : $Is the current data handle to move.$

1.1.3 Variables pour évaluer la charge

*load_*N_i *: Load on* N_i.
Over : Dictionary of nodes overloaded on the runtime. The key is N_i *for the node* i *and the value is the current charge on the node.*
Under : Dictionary of nodes underloaded on the runtime. The key is N_i *for the node* i *and the value is the current charge on the node.*
min : Is the least loaded node among the underloaded nodes, $min \in Under$.
average_load : Is the average load on the machine.
threshold : The value to add and substract to the average_load to obtain the bound_load.

bound_load : Is the average load added to the threshold.
$Tasks_d$: List of each task which write on the data handle d, $d \in (0, nb\_data)$

1.1.4 Variables pour la complexité

P *: Nombre de processeurs/noeuds,* ***p*** *désigne un processeur.*
D *: Nombre de data handle,* ***d*** *désigne un data handle.* $\mathbf{D_p}$ *représente le nombre de ta handle sur P.*
T *: Nombre de tâches* $\mathbf{T_p}$ *représente le nombre de tâches sur P et* $\mathbf{T_d}$ *le nombre de tâches associées au data handle d.*
$\mathbf{cod_t}$ *: Nombre de données associées à la tâche* ***t****.*
over *: Nombre de noeuds surchargés,* ***o*** *désigne une noeud surchargé.*
$\mathbf{data_o}$ *: Nombre de data handle à déplacer pour rééquilibrer o.*

1.2 Les fonctions

find_over_under_load_nodes() *: Returns 2 disctionary. One contains the overloaded nodes and the other the underloaded nodes.*
find_data_load() *: Returns the charge of a data handle d. This load is computed by taking all the tasks related to d and adding their cost. The cost is calculated with the cost function provided in the task codelet.*
find_tasks_associated_to_1data(*$data_d$****)*** *: Returns the list of tasks which write on the $data_d$.* ***find_min_data_to_move_for_balance()*** *: Returns a dictionary with minimum number of data to move onto an overloaded node to balance it.*
find_lightest_data_to_move_for_balance() *: Returns a dictionary with the lightest data handles to move to balance the overloaded node.*
find_worth_it_data_to_move_for_balance() *: Returns a dictionary of data handles to move to balance the overloaded node if the move improve the global situation.*

1.3 Les Algorithmes

List scheduling
LPT

2 Fonctions utiles

Cette fonction permet de calculer la charge associée à une data handle, ***d****. Pour cela, on cherche la liste de tâches associées à* ***d*** *et on somme le coût de ces tâches. La complexité de cette fonction est la suivante :*
Soit $\mathbf{T_d}$ *le nombre de tâches associées à* ***d****. On trouve* $\mathbf{T_d}$ *avec la fonction* ***find_tasks_associated_to_1data*** *qui a une complexité en fonction de* $\mathbf{T_p}$ *le nombre de tâche sur le noeud de* ***d*** *et de* $\mathbf{cod_t}$ *qui la longueur de la liste parcourut pour vérifier les accès lecture/écriture.*

Complexité : $\mathbf{T_p} \times \mathbf{cod_t} + \mathbf{T_d} \times \mathbf{cod_t} = \mathbf{cod_t} \times (\mathbf{T_p} + \mathbf{T_d})$

Si on prend la complexité dans le pire cas, on peut majorer $\mathbf{T_d}$ *par* $\mathbf{T_p}$ *car* $\mathbf{T_d} \in \mathbf{T_p}$. *On peut ensuite majorer* $\mathbf{T_p}$ *par* $\boldsymbol{T}$, *le nombre de tâches totales. On obtient alors (sans compter les coefficients) :* $\mathbf{cod_t} \times \mathbf{T}$

Algorithme 1 : find_data_load

Data: *Runtime*, $data_d$
Result: *cost*
$Tasks_d = \emptyset$, *cost*=0
$Tasks_d \leftarrow$ **find_tasks_associated_to_1data**(data$_d$)
for all task $\in$Tasks$_d$ **do**
    cost $\leftarrow$ cost+**task_cost**(task)
return cost

3 List Scheduling

List scheduling est un algorithme centralisé de référence. On parcourt une liste d'éléments et on les distribue au noeud le moins chargé. L'algorithme présente une modification : on parcourt les noeuds surchargés au lieu d'une liste de data handle à placer. Dès que le noeud n'est plus surchargé, on passe au noeud suivant.
Pour calculer la complexité de cette algorithme, on doit prendre en compte la complexité des focntions suivantes : ***find_over_under_load_nodes()*** *et* ***find_data_load()***.

Algorithme 2 : List Scheduling

Data: *Runtime*, *threshold*
Result: *Runtime*
Over, *Under* $\leftarrow$ **find_over_under_load_nodes**(*Runtime*, *threshold*)
min $=\emptyset$
for all *over* $\in$ *Over* **do**
    while *over* is overloaded **do**
        min $\leftarrow$ **min**(*Under*)
        Runtime $\leftarrow$ Move the first data in *over* named $data1_i$ on the node *min*
        Runtime $\leftarrow$ Redistribute tasks associated with $data1_i$
        Over[over] $\leftarrow$ *Over*[over]-**find_data_load**(*Runtime*, $data1_i$)
        Under[min] $\leftarrow$ *Under*[min]+**find_data_load**(*Runtime*, $data1_i$)
return *Runtime*

4 Largest Processing Time

LPT est un algorithme d'ordonnancement centralisé de référence. Pour une liste de tâche à ordonnancer, on sélectionne la tâche la plus coûteuse et on la distribue au noeud le moins chargé. L'algorithme implémenté ici n'est pas la version exacte du LPT car on doit créer notre liste en enlevant du travail aux noeuds surchargés. Pour ce faire on peut utiliser différentes méthodes. Ces méthodes sont détaillées ci-dessous.

Algorithme 3 LPT

Data: *Runtime*, *threshold*
Result: *Runtime*
$data\_to\_remap = \emptyset$, $data\_to\_move = \emptyset$
Over, *Under* ← **find_over_under_load_nodes**(*Runtime*, *threshold*)
for all $over \in Over$ **do**
    $data\_to\_move$ ← **find_data_to_move_for_balance**(*Runtime*, *over*)
    Add $data\_to\_move$ to $data\_to\_remap$
while $data\_to\_remap \neq \emptyset$ **do**
    $moving\_data$ ← **max**($data\_to\_remap$)
    min ← **min**(*Under*)
    Runtime ← Move $moving\_data$ on the node *min*
    Runtime ← Redistribute tasks associated with $moving\_data$
    Under[min] ← *Under*[min]-**find_data_load**(*Runtime*, $moving\_data$)
    Delete $moving\_data$ from $data\_to\_remap$
return *Runtime*

4.1 Fonctions de tri de la liste

Algorithme 4 find_min_data_to_move_for_balance

Data: *Runtime*, N_i, $load\_N_i$, *threshold*
Result: $data\_to\_move$
$data\_to\_move = \emptyset$,
$load\_to\_move = load\_N_i$,
$bound\_load = load\_N_i + threshold$
for *iter* in (0, Nb_data_i)
    for all *combi* combinations of *iter* size among $data\_N_i$
        $data\_load \leftarrow 0$
        for *data* in *combi*
            $data\_load \leftarrow data\_load +$ **find_data_load**(*Runtime*, *data*)
        if $bound\_load >$ load_N$_i - data_load$
            if $load\_to\_move > data\_load$
                $load\_to\_move \leftarrow data\_load$
                Clear $data\_to\_move$
                Add elements of *combi* to $data\_to\_move$
    if $data\_to\_move \neq \emptyset$
        return $data\_to\_move$

Algorithme 5 find_lightest_data_to_move_for_balance

Data: *Runtime*, N_i, $load\_N_i$, *threshold*
Result: *data_to_move*
$data\_to\_move = \emptyset$,
$data\_on\_N_i = \emptyset$, `/* a dictionary */`
$bound\_load = load\_N_i + threshold$
for d in $data\_N_i$
    $data\_on\_Ni$[d] ← **find_data_load**(*Runtime*, d)
while $load\_N_i > bound\_load$ **do**
    $moving\_data$ ← **min**($data\_on\_N_i$)
    Add $moving\_data$ and its value to $data\_to\_move$
    Delete $moving\_data$ from $data\_on\_N_i$
if $data\_to\_move \neq \emptyset$
    return $data\_to\_move$

Algorithme 6 find_worth_it_data_to_move_for_balance

Data: *Runtime*, N_i,*Under*, $load\_N_i$, *threshold*
Result: *data_to_move*
$data\_to\_move = \emptyset$
$bound\_load = load\_N_i + threshold$
$min = \emptyset$
for d in $data\_N_i$
    $load_d$ ← **find_data_load**($Runtime, d$)
    min← **min**($Under$)
    if is_better_balanced(*Runtime*, $load\_N_i$, $load\_N_i - load_d$, $Under[min]$, $Under[min] + load_d$)
        load_N$_i$ ←load_N$_i$ − $load_d$
        $Under$[min] ← $Under[min] + load_d$
    if bound_load ≥ load_N$_i$
        break
return *data_to_move*