Négociation multi-agents pour la réallocation dynamique de tâches

Quentin Baert

Centre de Recherche en Informatique Signal et Automatique de Lille

Université de Lille - Faculté des Sciences et Technologies

Thèse en vue de l'obtention du titre de
Docteur en Informatique et Applications de l'Université de Lille

Négociation multi-agents pour la réallocation dynamique de tâches

et application au patron de conception MapReduce

Quentin Baert

Soutenue le 13 septembre 2019

Jury :

Rapporteurs :	
Amal El Fallah Seghrouchni	Professeure à l'Université Paris-Sorbonne
Laurent Vercouter	Professeur à l'INSA de Rouen
Examinateurs :	
Salima Hassas (*présidente*)	Professeure à l'Université Claude Bernard Lyon 1
Nicolas Maudet	Professeur à l'Université Paris-Sorbonne
Directeurs :	
Anne-Cécile Caron	Maître de Conférences à l'Université de Lille
Maxime Morge	Maître de Conférences à l'Université de Lille
Jean-Christophe Routier	Professeur à l'Université de Lille

Remerciements

Ce document propose une approche collaborative pour la résolution d'un problème complexe. Aussi complexes qu'elles aient été à écrire, les pages suivantes sont également le fruit de multiples collaborations professionnelles et personnelles. Qu'ils aient participé de près ou de loin aux travaux présentés par la suite, beaucoup ont contribué à faire de cette thèse une belle expérience de recherche.

Mes premiers remerciements vont naturellement à Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier pour leur encadrement durant ces trois ans. Je tiens d'abord à les remercier pour leur patience, pour leur disponibilité et pour avoir répondu à chacune de mes questions. Je les remercie également pour leur sympathie, leur confiance et leurs conseils. Si, par l'utilisation de la première personne, je semble m'attribuer les contributions de cette thèse, il n'en est rien. Derrière ces « je » se cachent des « nous », et je n'oublie pas l'importance cruciale de nos échanges lors des réunions hebdomadaires.

Je souhaite ensuite remercier Amal El Fallah Seghrouchni et Laurent Vercouter d'avoir accepté de rapporter cette thèse. Je remercie également Salima Hassas et Nicolas Maudet d'examiner mes travaux et de participer au jury.

Merci à l'ensemble de l'équipe SMAC de m'avoir offert l'environnement de travail et les ressources nécessaires à faire aboutir cette thèse. Pour nos amicales conversations, je remercie Jean-Christophe Routier, Anne-Cécile Caron, Maxime Morge, Antoine Nongaillard, Philippe Mathieu, Patricia Everaere et Jean-Paul Delahaye. Pour son accueil à la Royal Holloway University of London, un grand merci à Kostas Stathis.

Un doctorant est aussi un étudiant et même si j'ai conscience qu'il est temps que je devienne un agent autonome, la relation mentor/élève a toujours été importante pour moi. Ainsi, à l'Université de Lille, et plus particulièrement au FIL, j'ai pu rencontrer des mentors que je n'oublierai pas. Je remercie chaleureusement Jean-Christophe Routier, Philippe Mathieu, Samuel Hym et Guillaume Dubuisson-Duplessis qui m'ont appris à raisonner, à coder, et bien plus que ça.

Le doctorat représente la fin de mon voyage d'étudiant, et j'y ai côtoyé de fabuleux camarades sans qui l'aventure n'aurait pas été la même. Pour commencer, je remercie Thomas qui, en plus d'avoir été un binôme irremplaçable, est aujourd'hui un ami indispensable. Ensuite, un énorme merci à Elliot, Camille, Anne-Sophie, Alexandre, Antonin, Gaëtan, Émilie, Pierre, Benjamin et Romain. Les avoir toutes et tous rencontrés est la plus belle réussite de mes études supérieures.

J'ai beaucoup de mal à trouver les mots pour remercier l'ensemble de mes proches. Je leur dois tout et ce n'est pas suffisant de l'écrire. Cependant, si mon nom est sur la première page de ce document, le leur doit se trouver sur la deuxième. Merci à mes parents Nathalie, Roland, Stéphane ; à mes frangins et frangines Antoine, Mélanie, Baptiste, Joseph ; à mes grands-parents Jeanine, René, Nadine, Pierre, Bernard, Maryse, Jeannine. Un merci tout particulier à Jérôme et Pauline car ils me montrent que l'on devient qui l'on veut par le travail, la curiosité et l'écoute. Merci également à ma seconde famille, à celles et ceux qui rassurent, qui questionnent, qui

supportent, qui encouragent : Robin, Florence, Thomas, Anne-Sophie, Elliot et Camille.

Enfin, et peut être surtout, je remercie Mylène. Son courage, sa détermination et sa joie me portent tous les jours. Quel plaisir d'apprendre et de mûrir à ses côtés.

Table des matières

Introduction

Scénario introductif

Une fois par semaine, les chirurgiens du bloc opératoire de cardiologie se réunissent pour mettre en place le planning des blocs de la semaine à venir. Pour cela, ils disposent de la liste des opérations prévues pour un jour donné et ils doivent décider de la répartition des opérations dans les différentes salles opératoires. Chaque chirurgien dispose de sa propre salle, est capable de réaliser toutes les opérations de la liste et donne le temps qu'il estime nécessaire pour qu'il réalise chaque opération. L'objectif est alors de répartir les opérations de sorte que toutes les équipes chirurgicales puissent quitter le bloc opératoire au plus tôt.

John, Horace et Maria réfléchissent au planning de lundi. Cinq opérations sont prévues et les temps estimés par les trois chirurgiens sont présentés dans le tableau 1.

	Opération n° 1	Opération n° 2	Opération n° 3	Opération n° 4	Opération n° 5
John	100	140	350	200	80
Horace	80	130	210	200	110
Maria	120	130	320	120	130

Tableau 1 : Temps estimé (en minutes) par les chirurgiens pour chaque opération prévue lundi.

Après mûre réflexion, les trois chirurgiens décident de se répartir les opérations de la manière suivante :

- John réalisera les opérations n° 2 et n° 5 en 220 minutes ;
- Horace réalisera l'opération n° 3 en 210 minutes ;
- Maria réalisera les opérations n° 1 et n° 4 en 240 minutes.

Ainsi, toutes les opérations seront réalisées en 240 minutes, ce qui représente le temps minimal selon les estimations.

Plusieurs difficultés se posent ici. La première consiste à trouver la répartition optimale des opérations. Ceci représente un défi non trivial qui peut également prendre du temps. La seconde difficulté réside dans les mesures correctives qu'il pourrait être nécessaire d'appliquer. Par exemple, à cause d'une faute humaine ou matérielle, il se peut qu'une opération prenne du retard et qu'il faille revoir tout le planning des opérations. Enfin, la troisième difficulté est d'estimer précisément le temps nécessaire à un chirurgien pour réaliser une opération. Une telle mesure est compliquée (voire impossible), ce qui relativise tout plan préalable et souligne la nécessité d'ajuster la répartition des opérations.

Ce scénario introductif présente un cas d'allocation de tâches (ici les opérations) à plusieurs agents (ici John, Horace et Maria). Les difficultés citées ci-dessus se posent dès lors que des

agents collaboratifs doivent exécuter un ensemble de tâches le plus rapidement possible. Pour réduire l'impact de ces difficultés, je propose une méthode de réallocation distribuée et dynamique de tâches. Par distribuée, j'entends qu'il n'existe pas d'agent qui possède l'ensemble des informations sur l'état global du système et qui décide de l'allocation des tâches pour l'ensemble des autres agents. La réallocation est plutôt le résultat d'interactions entre agents, qui déterminent individuellement et selon leurs connaissances si un ajustement de l'allocation des tâches est nécessaire. Par dynamique, je désigne le fait que certaines tâches sont réallouées pendant que d'autres sont exécutées par les agents. Ici par exemple, si John estime mal le temps dont il a besoin pour l'une des opérations et qu'il y consacre plus de temps que prévu, il faut pouvoir réallouer dynamiquement les opérations qu'il reste à réaliser aux chirurgiens. Cela permettrait de remettre en cause l'allocation des opérations pour prendre en compte le retard de John, toujours dans l'objectif de faire terminer l'équipe chirurgicale au plus tôt. Les informations utilisées par le mécanisme présenté dans cette thèse ne sont pas fixes, ni issues d'une connaissance préalable du problème ou des tâches à exécuter. Ce sont celles perçues par les agents au moment de la réallocation.

Contexte scientifique

Les travaux exposés dans ce document se situent dans la continuité de ceux de l'équipe SMAC [1], membre du laboratoire CRIStAL [2]. En effet, cette thèse se positionne dans le cadre de la résolution multi-agents de problèmes distribués et prend place à l'intersection de trois contextes scientifiques : les problèmes d'allocation de tâches, la négociation multi-agents et le traitement de données massives à l'aide du patron de conception MapReduce. L'apport de la négociation multi-agents pour l'allocation de ressources ou de tâches à des agents non-coopératifs est connu. Cependant, mes travaux portent sur la réallocation dynamique de tâches pour des agents collaboratifs qui négocient dans l'objectif de corriger une allocation qui offre une mauvaise répartition de la charge de travail. Cette approche est centrée individu et repose sur un modèle de comportement des agents qui, en prenant des décisions locales, permettent une meilleure répartition des charges et donc un temps d'exécution plus court de l'ensemble des tâches.

Contributions

Dans ce document, je défends la thèse selon laquelle, dans un cadre collaboratif et dans l'objectif de faire décroître le temps d'exécution d'un ensemble de tâches sujettes à perturbations, il est pertinent et efficace de réallouer ces tâches dynamiquement et de façon décentralisée grâce à des décisions locales. Les travaux présentés ici peuvent être résumés en trois contributions principales.

1. Systèmes Multi-Agents et Comportements
2. Centre de Recherche en Informatique, Signal et Automatique de Lille

Contribution n° 1 : un processus de réallocation de tâches concurrent à leur exécution

Les problèmes d'allocation de tâches peuvent avoir différents objectifs dans de nombreux domaines applicatifs. Chaque processus d'allocation doit prendre en compte toutes les particularités des tâches et des agents auxquels elles sont allouées. Cela implique généralement de concevoir des solutions dédiées qui, spécifiques à une application donnée, supposent de manipuler des informations exactes, notamment le coût des tâches pour les agents. Dans le cadre d'une application pratique, j'estime que connaître le coût exact d'une tâche pour un agent est une hypothèse forte qu'il est difficile de vérifier a priori.

Les processus d'allocation de tâches sont souvent utilisés préalablement à l'exécution des tâches afin de produire une allocation qui maximise (ou minimise) un objectif qui reflète la qualité de l'allocation. Si la qualité de l'allocation est jugée suffisante, celle-ci est utilisée lors de l'exécution des tâches sans être remise en question. Ne pas remettre en cause l'allocation alors que les tâches sont exécutées suppose qu'aucun événement ne peut dégrader la qualité de cette dernière. Là encore, je considère que cela est une hypothèse forte et difficile à vérifier en pratique.

Dans cette thèse, je considère le problème qui consiste à allouer les tâches aux agents de manière à terminer l'exécution de toutes les tâches au plus tôt. Je me restreins à des profils de tâches simples mais je suppose n'avoir qu'une estimation du coût de ces tâches pour les agents. Ainsi, à tout moment de l'exécution des tâches, je considère que l'allocation peut être remise en question afin d'être améliorée. Pour cela, je propose un mécanisme de réallocation qui a lieu durant l'exécution. Cette réallocation dynamique permet d'améliorer l'allocation courante et donc de corriger les approximations liées à l'estimation du coût des tâches ou à un problème qui survient pendant l'exécution.

Comme énoncé dans ces deux citations de [Pinedo 2008, Chapitre 20], la réallocation de tâches est un problème qui mérite d'être exploré et qui peut donner lieu à d'intéressantes recherches théoriques et expérimentales.

> A rescheduling problem may have multiple objectives : the objective of the original problem [...] and the minimization of the difference between the new schedule (after rescheduling) and the old schedule (before rescheduling). It may be necessary to have formal definitions or functions that measure the "difference" or the "similarity" between two schedules for the same job set or between two schedules for two slightly different job sets, e.g., one job set having all the jobs of a second set plus one additional job [...].

> [...] the concepts of robustness and rescheduling may lead to interesting theoretical research. However, they may lead to even more interesting empirical and experimental research. New measures for robustness have to be developed. The definition of these measures may depend on the machine environment.

Cette thèse présente donc un cadre formel qui décrit un processus de réallocation de tâches pendant l'exécution.

Contribution n° 2 : la négociation multi-agents pour la réallocation dynamique de tâches

L'utilisation de la négociation multi-agents pour l'allocation de tâches est commune. En revanche, la négociation est généralement découplée de l'exécution des tâches et elle s'opère souvent entre des agents compétitifs qui souhaitent maximiser leur intérêt individuel.

Dans le cadre de cette thèse, les agents sont collaboratifs : leur objectif est que toutes les tâches soient réalisées au plus tôt. Une allocation de qualité est donc une allocation qui répartit équitablement la charge de travail parmi les agents. Le protocole d'interaction qu'utilisent les agents leur permet d'effectuer des délégations de tâche socialement rationnelles. De ce fait, les agents acceptent d'augmenter leur charge de travail tant que cela est bénéfique à l'ensemble du système. De plus, ce rééquilibrage des charges se déroule au travers de multiples négociations concurrentes pendant l'exécution des tâches. Cela a pour conséquence que l'ensemble des tâches à répartir est continuellement modifié. De plus, à chaque instant, l'allocation courante est confrontée à la réalité de l'exécution. Les aléas de l'exécution peuvent donc venir perturber l'équilibre des charges précédemment construit.

Dans cette thèse, je propose une architecture, des comportements et des stratégies qui permettent aux agents d'exécuter les tâches tout en s'impliquant dans plusieurs négociations concurrentes. Le processus de négociation est décentralisé et chaque agent possède ses propres croyances sur l'état du système. Malgré l'écart possible entre les croyances des agents et la réalité du système, la proposition de cette thèse garantit une amélioration monotone de l'allocation de tâches.

Par la suite, les agents sont distribués sur différents nœuds de calculs. Afin de pouvoir utiliser la proposition de cette thèse pour de véritables applications distribuées, le protocole de négociation utilisé par les agents est asynchrone. Il prend également en considération la perte et le retard de messages.

Contribution n° 3 : un système multi-agents pour la correction des biais du patron de conception MapReduce

Le patron de conception MapReduce permet de distribuer sur plusieurs machines le traitement de larges volumes de données. Amplement utilisé et étudié, MapReduce est très efficace mais peut souffrir de biais lorsque les différentes étapes du flux de données ne sont pas spécifiquement ajustées aux données. Un des biais de la phase de *reduce* implique une mauvaise répartition de la charge de travail sur les différents nœuds de calcul. Un job MapReduce étant considéré comme terminé lorsque l'ensemble des tâches ont été exécutées, un mauvais équilibre des charges implique un temps d'exécution plus élevé.

Le mécanisme de réallocation présenté dans cette thèse permet la correction de certains des biais de MapReduce. Mon ambition est de proposer une solution générique qui corrige une allocation de tâches biaisée sans connaissances préalables des données à traiter. Pour cela, j'ai développé un prototype distribué qui implémente le patron de conception MapReduce. En associant un agent à chaque nœud de calcul et en implémentant les comportements, stratégies et protocoles évoqués ci-dessus, le prototype permet d'exécuter un job MapReduce et de corriger les biais de la phase de *reduce* pendant son déroulement.

Le prototype est fonctionnel et son code est accessible [Baert *et al.* 2019c]. Il permet de traiter des jeux de données réelles de manière distribuée grâce au patron de conception MapReduce. Durant la phase de *reduce*, des agents *reducers* réallouent les tâches afin de diminuer le temps nécessaire à l'exécution de toutes les tâches. Ce prototype confirme l'applicabilité de l'approche défendue dans cette thèse et m'a permis d'en valider expérimentalement les propositions.

Proposition

Grâce à l'étude du problème de réallocation multi-agents de tâches situées, appelé MASTA (pour *Multi-Agent Situated Task Allocation*) dans ce manuscrit, je propose d'étudier comment des agents collaboratifs peuvent mettre en place un mécanisme de réallocation de tâches distribué, sans partage de connaissances, et concurrent à l'exécution de ces mêmes tâches. J'identifie également l'impact de la réallocation des tâches sur la qualité de l'allocation et je propose des stratégies pour accroître cet impact. Pour la mise en œuvre de ces propositions, je décris un protocole d'interaction robuste qui autorise de multiples négociations concurrentes, ainsi qu'une architecture composite d'agent qui facilite la transition entre la spécification du comportement des agents et leur implémentation. Ces travaux m'ont permis de développer un prototype qui implémente le patron de conception MapReduce grâce à la distribution d'un système multi-agents. Ce prototype offre un contexte applicatif pour valider l'approche défendue sur des données réelles.

Organisation du document

Le document est organisé en trois parties. Chacune des parties contient trois chapitres. La figure 1 propose un plan de lecture de ce document et exprime les dépendances entre les chapitres.

Partie 1 : État de l'art

Cette partie présente l'état de l'art lié aux problématiques abordées dans cette thèse.

Chapitre 1 : Allocation de tâches et équilibrage de charges dans un système distribué. Ce chapitre présente les différents problèmes d'allocation de tâches et il pose les définitions de base du problème sur lequel se concentre la thèse. Les différentes typologies de systèmes distribués, de tâches et d'objectifs y sont explorées. J'y décris également plusieurs travaux qui étudient et/ou utilisent l'allocation de tâches dans un système distribué.

Chapitre 2 : Négociation multi-agents pour l'allocation de tâches. Ce chapitre parcourt les différents ingrédients nécessaires à la négociation multi-agents. Il présente notamment le *Contract Net Protocol*, protocole d'interaction sur lequel se base celui développé dans cette thèse. Il explore également comment, à partir des évaluations individuelles des agents, il est possible d'évaluer collectivement une allocation de tâches.

Chapitre 3 : Patron de conception MapReduce. Ce chapitre décrit le patron de conception MapReduce ainsi que ses biais. J'y présente les travaux qui proposent des solutions pour contrer les biais de MapReduce et j'y positionne mon approche par rapport à ces derniers.

Partie 2 : Modélisation multi-agents

Cette partie présente le cadre formel qui décrit la proposition défendue dans cette thèse. Ses trois chapitres constituent le cœur de la proposition.

Chapitre 4 : Réallocation multi-agents de tâches situées. Ce chapitre formalise précisément le problème d'allocation de tâches situées dans un système multi-agents (MASTA). J'y présente les délégations socialement rationnelles, les opérations que réalisent les agents pour améliorer l'allocation de tâches courante, ainsi que leur impact sur la qualité de l'allocation. J'explique également pourquoi une réallocation des tâches concurrente à leur exécution permet de corriger une allocation dont les coûts des tâches auraient été mal estimés.

Chapitre 5 : Protocole d'interaction. Ce chapitre définit en détail le protocole d'interaction utilisé par les agents pour procéder aux délégations de tâches socialement rationnelles. Ce protocole est basé sur le *Contract Net Protocol*, lui même décrit dans le chapitre 2. Le fait que ce mécanisme de délégation soit décentralisé implique que les agents prennent des décisions en considérant leur connaissance locale du système. Malgré cela, le protocole d'interaction garantit que toutes les délégations de tâches qui aboutissent sont socialement rationnelles. Ce chapitre décrit également comment un agent peut être engagé dans plusieurs négociations simultanées.

Chapitre 6 : Stratégies. Comme les négociations se déroulent en même temps que les exécutions, un agent doit pouvoir choisir quelle tâche déléguer et quelle tâche exécuter. Ce chapitre présente les différentes stratégies utilisées par les agents pour qu'ils exécutent et négocient efficacement les tâches. En plus de considérer le coût des tâches, la stratégie de sélection de tâches localisées considère la localité des ressources nécessaires à l'exécution d'une tâche pour déterminer la tâche à exécuter ou à déléguer. Si le profil des tâches le permet, il est également possible de les découper pour affiner la répartition de la charge de travail parmi les agents. Ce chapitre propose une stratégie de découpe de tâche.

Partie 3 : Mise en œuvre

Cette partie présente la transition entre le cadre formel exposé dans la partie précédente et l'application pratique.

Chapitre 7 : Architecture composite d'agent. Ce chapitre décrit la structure interne des agents. Je propose une architecture composite d'agent qui permet de simplifier la spécification et la conception des agents. Un agent est ainsi composé de trois agents composants : un *manager*, un *broker* et un *worker*. Chaque agent composant dispose de son propre rôle et de son propre comportement. Le déroulement concurrent des comportements des agents composants définit le comportement global de l'agent composite.

Chapitre 8 : MAS4Data. Ce chapitre expose comment la phase de *reduce* d'une exécution MapReduce (décrite dans le chapitre 3) peut être exprimée et résolue comme un problème MASTA. J'y présente MAS4Data, le prototype distribué implémenté dans le cadre de cette thèse, que je compare à Hadoop, une implémentation référence de MapReduce. Dans MAS4Data, des agents *reducers* réallouent les tâches pendant la phase de *reduce*. De ce fait, ils améliorent le temps d'exécution de l'ensemble des tâches, c'est-à-dire le *makespan* du job MapReduce.

Chapitre 9 : Validations expérimentales. Ce chapitre présente des expériences menées sur

de véritables jeux de données grâce à MAS4Data. Les résultats de ces expériences montrent que la réallocation dynamique de tâches permet de corriger une allocation de tâches mal équilibrée mais aussi une mauvaise estimation du coût des tâches ou encore un événement imprévu durant l'exécution (par exemple la chute de performance d'un nœud).

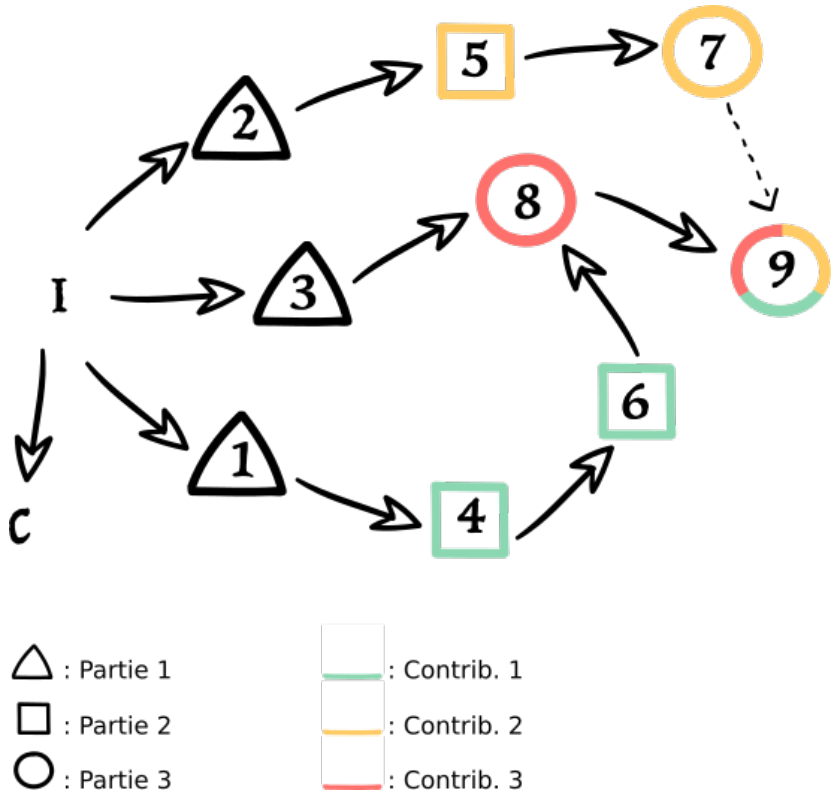


FIGURE 1 : Plan de lecture de cette thèse sous la forme d'un graphe de dépendances. Chaque chapitre est représenté par son numéro. L'introduction est représentée par la lettre « I » et la conclusion par la lettre « C ». Les chapitres 1, 2 et 3 définissent l'état de l'art. Les chapitres 4, 5 et 6 constituent le cœur de la proposition. Les chapitres 7, 8 et 9 décrivent comment elle a été implémentée et expérimentée. La contribution n° 1 est décrite dans les chapitres 4 et 6. La contribution n° 2 est décrite dans les chapitres 5 et 7. La contribution n° 3 est décrite dans le chapitre 8. Le chapitre 9 valide les trois contributions. La compréhension de l'architecture composite d'agent présentée dans le chapitre 7 n'est pas nécessaire pour la lecture du chapitre 9 mais le prototype utilisé lors des validations expérimentales utilise cette architecture.

Notations

Notation	Description
S	Système distribué
$\mathcal{N}$	Ensemble des nœuds de calcul
$\mathcal{A}$	Ensemble des agents
$\mathcal{T}$	Ensemble des tâches
$\mathcal{E}$	Ensemble des relations entre agents
$\mathcal{R}$	Ensemble des ressources
l	Fonction de distribution des agents sur les nœuds de calcul
d	Fonction de distribution des ressources sur les nœuds de calcul
$\mathcal{R}_\tau$	Resources nécessaires à l'exécution de la tâche τ
$\mathcal{R}_i$	Ressources locales à l'agent i
$d_i(\tau)$	Nombre de ressources locales à l'agent i et nécessaires à l'exécution de la tâche τ
ρ_k^τ	k-ième *chunk* (ressource) de la tâche τ
P	Allocation de tâches
P_i	Tâches allouées à l'agent i dans l'allocation P
c	Fonction de coût
$c_i(\tau)$	Coût de la tâche τ pour l'agent i
$w_i(P)$	Charge de travail de l'agent i pour l'allocation P
$t_i(\tau)$	Délai de réalisation de la tâche τ pour l'agent i
$C(P)$	*Flowtime* de l'allocation P
$C_{max}(P)$	*Makespan* de l'allocation P

Notation	Description
$MASTA$	Instance d'un problème MASTA
δ	Accord / délégation de tâche
γ	Consommation de tâche
$\Gamma_i(P)$	Ensemble des délégations socialement rationnelles pour l'agent i dans l'allocation P
$\mathcal{B}_i(P)$	Croyances de l'agent i sur les charges de ses pairs pour l'allocation P
$\Gamma_i^{\mathcal{B}}(P)$	Ensemble des délégations potentiellement socialement rationnelles pour l'agent i dans l'allocation P
$\mathcal{D}_j$	Ensemble des délégations de tâche dans lesquelles l'agent j est engagé comme enchérisseur
$v_j(\mathcal{D})$	Surcharge de travail virtuelle de l'agent j
$\overset{>}{\tau}$	Tâche à exécuter selon la stratégie de sélection de tâche
$\overset{<}{\tau}$	Tâche à déléguer selon la stratégie de sélection de tâche
$o_i(\tau)$	Ratio de localité de la tâche τ pour l'agent i
$\hat{o}(\tau)$	Ratio de localité maximum de la tâche τ
μ_k^τ	k-ième sous-tâche de la tâche τ
Δ_i^k	Delta de charge entre celle de l'agent i et celle du k-ième agent le moins chargé selon les croyances de i

Première partie

État de l'art

Chapitre 1

Allocation de tâches et équilibrage de charges dans un système distribué

Sommaire

1.1 Introduction

L'allocation de tâches dans un système distribué consiste à répartir un ensemble de tâches parmi plusieurs agents qui exécutent ensuite ces tâches. Dans ce chapitre, le terme **agent** désigne toute entité propre au domaine d'application capable d'effectuer une tâche (par exemple un nœud de calcul, un drone, une machine dans une chaine de production, etc.). Bien qu'il soit possible d'allouer aléatoirement les tâches aux agents, il est nécessaire de le faire de manière plus intelligente afin d'atteindre un **objectif**. Le plus souvent, on souhaite :

- soit minimiser le temps d'éxécution d'une tâche (c'est-à-dire le temps entre le moment où une tâche apparaît dans le système et le moment où son résultat est produit) ;
- soit minimiser le temps d'exécution de l'ensemble des tâches ;
- soit maximiser le rendement (c'est-à-dire le nombre de tâches effectuées par pas de temps) ;
- soit maximiser la fiabilité du système (c'est-à-dire le nombre de tâches exécutées avec succès).

Afin d'être exécutée, une tâche nécessite généralement l'utilisation de **ressources**. Ces ressources peuvent être de différentes natures : données, capacités de calculs, mémoire, etc. Elles peuvent aussi être réparties inégalement parmi les agents, ce qui justifie le fait qu'une tâche puisse coûter plus pour un agent que pour un autre. Le mécanisme d'allocation doit prendre en compte la

répartition des ressources lors de l'affectation des tâches aux agents. Une ressource est **partageable** si elle est utilisable par un autre agent que celui qui la possède initialement, elle est consommable si elle disparait une fois utilisée. Si un agent a directement accès à une ressource, celle-ci est **locale** pour lui. Une ressource non locale et non partageable est inacessible. Une ressource non locale mais partageable est **distante**.

Le mécanisme d'allocation doit également prendre les caractéristiques des tâches en compte. Ainsi, une tâche est dite **divisible** si elle peut être divisée en plusieurs sous-tâches sans en altérer le résultat, **préemptive** si elle peut être commencée par un agent puis suspendue pour être reprise plus tard (possiblement par un autre agent), **indépendante** si son résultat ne dépend pas de l'exécution d'une autre tâche. Dans certains problèmes, une tâche possède également une date butoir avant laquelle elle doit être exécutée. Si une tâche est allouée à plusieurs agents, on dit que ceux-ci forment une coalition.

Les agents sont **coopératifs** s'ils partagent un objectif commun et considèrent l'intérêt du système plutôt que leur intérêt individuel. Ils sont égoïstes dans le cas contraire.

La **topologie** du réseau d'agents représente les liens qui existent entre les agents. Il est possible qu'un agent ne puisse pas communiquer avec tous les autres, et donc que toutes les ressources ne soient pas accessibles pour tous les agents. La topologie peut également établir une distance entre deux agents et donc entre un agent et une ressource qui lui est distante.

Afin de produire une allocation qui réponde à un objectif, il faut souvent être capable de mesurer le coût d'une tâche pour un agent. Dans le cadre d'applications pratiques, une telle mesure peut être compliquée et il est possible que les capacités des agents ne soient pas constantes au cours de l'exécution. Les questions suivantes se posent alors : une tâche est-elle allouée une fois pour toute ou peut-elle encore circuler parmi les agents ? Quel est le coût d'une potentielle réallocation des tâches ? La réallocation d'une tâche doit-elle être globale, c'est-à-dire relancer l'ensemble du processus d'allocation ou peut-elle être locale et se faire grâce à des **ajustements locaux** ?

Plusieurs formes de problème d'allocation existent. Par exemple, [Kuhn 2010] décrit un problème d'affectation dans lequel il existe autant de tâches que d'agents et où chaque agent doit traiter une seule tâche. Cette thèse s'intéresse plus particulièrement au problème $R_m||C_{max}$ tel que défini dans [Pinedo 2008]. R_m désigne un environnement de m agents hétérogènes et qui traitent les tâches en parallèle. C_{max} désigne l'objectif de minimiser le temps pour réaliser l'ensemble des tâches, aussi appelé le ***makespan***. Dans ce cadre, n tâches sont à allouer et on a souvent $n >> m$. Un agent peut donc se voir allouer plusieurs tâches. Ce problème est reconnu comme étant NP-difficile [Horowitz & Sahni 1976].

Par la suite, la section 1.2 définit les notions nécessaires à la compréhension du problème d'allocation abordé dans cette thèse. La section 1.3 présente des exemples issus de l'état de l'art. Enfin la section 1.4 décrit plus en détail la proposition de cette thèse et la positionne par rapport aux travaux précédemment décrits.

1.2 Problème d'allocation de tâches

Je présente ici une formalisation des notions nécessaires pour décrire un problème d'allocation de tâches **indépendantes**, **non divisibles** et **non préemptives**. Je considère des ressources

partageables et **non consommables**. Chaque tâche nécessite l'utilisation de ses propres ressources et celles-ci sont distribuées parmi les agents du système. Ainsi, si le coût de collecte d'une ressource est non nul, le coût d'une tâche varie pour un agent en fonction du nombre de ressources propres à la tâche qui lui sont locales. Les lecteurs intéressés pourront trouver plus de détails sur les différents types de problèmes d'allocation de tâches dans [Pinedo 2008, Jiang 2016].

Définition 1.1 (Système distribué).
Soient $\mathcal{N}$ un ensemble de nœuds, $\mathcal{A}$ un ensemble d'agents et $\mathcal{R}$ un ensemble de ressources. Un système distribué $S = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{R}, l, d)$ est un graphe simple non orienté dont les sommets sont les nœuds de $\mathcal{N}$ et dont les arêtes sont définies par $\mathcal{E}$, une relation binaire et symétrique entre nœuds.

l est la fonction de distribution des agents de $\mathcal{A}$. Elle associe à chaque agent un nœud du système.

$$l : \mathcal{A} \mapsto \mathcal{N} \tag{1.1}$$

d est la fonction de distribution des ressources de $\mathcal{R}$. Elle associe à chaque ressource un nœud du système :

$$d : \mathcal{R} \mapsto \mathcal{N} \tag{1.2}$$

*Un système distribué est **complet** si tous les nœuds sont en relation directe, c'est-à-dire $\forall \nu_i, \nu_j \in \mathcal{N}, (\nu_i, \nu_j) \in \mathcal{E}$. $\mathcal{E}^*$ est la clôture réflexive et transitive de $\mathcal{E}$.*

Dans la plupart des travaux sur l'allocation de tâches, il y a exactement un agent sur chaque nœud et les ensembles $\mathcal{N}$ et $\mathcal{A}$ sont donc confondus. Dans ce cas, et pour la suite de ce chapitre, on représente le système distribué $S = (\mathcal{A}, \mathcal{A}, \mathcal{E}, \mathcal{R}, \textit{identité}, d)$ par le quadruplet $S = (\mathcal{A}, \mathcal{E}, \mathcal{R}, d)$.

Par la suite, les ressources associées à l'agent $i \in \mathcal{A}$ par la fonction de distribution des ressources d sont notées $\mathcal{R}_i = d^{-1}(i)$. La structure du système distribué peut avoir un impact important sur l'allocation des tâches, par exemple rendre la résolution de l'ensemble des tâches impossible si un agent possède une ressource essentielle à l'exécution d'une tâche mais n'est pas en relation par $\mathcal{E}^*$ avec l'agent chargé de l'exécuter.

De manière générale, exécuter une tâche consiste à utiliser des ressources afin de produire un résultat.

Définition 1.2 (Tâche).
Soit $S = (\mathcal{A}, \mathcal{E}, \mathcal{R}, d)$ un système distribué et Res l'espace des résultats. Une tâche τ est une fonction qui associe un résultat à un ensemble de ressources :

$$\tau : 2^{\mathcal{R}} \mapsto Res \tag{1.3}$$

$\mathcal{R}_\tau \subseteq \mathcal{R}$ représente l'ensemble des ressources nécessaires à l'exécution de la tâche τ. L'ensemble Res est propre à chaque problème. Par exemple, il pourrait être : un ensemble de fichiers, l'ensemble des réels $\mathbb{R}$, l'ensemble des valeurs booléennes $\mathbb{B}$, etc.

Un problème d'allocation de tâches consiste à répartir des tâches dans un système distribué de manière à atteindre un objectif.

Définition 1.3 (Problème d'allocation de tâches).
Un problème d'allocation de tâches est défini par un couple $(S, \mathcal{T})$ *avec* $S = (\mathcal{A}, \mathcal{E}, \mathcal{R}, d)$ *un système distribué et* $\mathcal{T}$ *un ensemble de tâches à allouer.*

Supposons maintenant défini un système distribué $S = (\mathcal{A}, \mathcal{E}, \mathcal{R}, d)$ tel que $|\mathcal{A}| = m$ et un problème d'allocation de tâches $(S, \mathcal{T})$ tel que $|\mathcal{T}| = n$. Une solution de $(S, \mathcal{T})$ est une allocation de tâches, qui répartit les n tâches parmi les m agents. La validité d'une allocation dépend des caractéristiques du problème que l'on essaie de résoudre, de la forme des tâches à allouer et de la structure du système distribué. Dans le cadre de cette thèse, une allocation P est valide si :

- toutes les tâches sont allouées (cf. équation 1.4) ;
- chacune des tâches n'est allouée qu'à un seul agent (cf. équation 1.5),
- toutes les tâches sont réalisables. C'est-à-dire que pour chaque tâche, l'agent chargé de la réaliser possède toutes les ressources nécessaires ou est en mesure de les obtenir via ses pairs (cf. équation 1.6).

Définition 1.4 (Allocation de tâches).
Une allocation P *est une association entre les agents et un ensemble de tâches. À chaque agent* $i \in \mathcal{A}$ *est associé un ensemble de tâches* $P_i \subseteq \mathcal{T}$.

Formellement P *est valide si*

$$\bigcup_{i \in \mathcal{A}} P_i = \mathcal{T} \tag{1.4}$$

$$\forall i \in \mathcal{A}, \forall j \in \mathcal{A} \setminus \{i\}, P_i \cap P_j = \emptyset \tag{1.5}$$

$$\forall i \in \mathcal{A}, \forall \tau \in P_i, \exists \mathcal{A}' \subseteq \mathcal{A} \text{ tel que } \forall j \in \mathcal{A}', (i,j) \in \mathcal{E}^* \wedge (\mathcal{R}_\tau \subseteq \bigcup_{j \in \mathcal{A}'} \mathcal{R}_j) \tag{1.6}$$

Certains objectifs nécessitent de calculer le coût que représente une allocation pour un agent ou pour l'ensemble du système. Une fonction de coût est une structure de préférence cardinale quantitative qui donne le coût d'une tâche pour un agent (cf. [Chevaleyre *et al.* 2006]).

Définition 1.5 (Fonction de coût).
Une fonction de coût associe un coût à une tâche :

$$c : \mathcal{T} \mapsto Val \tag{1.7}$$

Val *est un ensemble de valeurs numériques strictement positives (par exemple* $\mathbb{N}_*^+$, $\mathbb{R}_*^+$, $]0,1]$, *etc.).* $c_i(\tau)$ *est le coût de la tâche* $\tau \in \mathcal{T}$ *pour l'agent* $i \in \mathcal{A}$.

Considèrons $i, j \in \mathcal{A}$ deux agents homogènes en termes de capacité de calcul et $\tau \in \mathcal{T}$ tels que i possède plus de ressources nécessaires à l'exécution de τ que j. On a $|\mathcal{R}_i \cap \mathcal{R}_\tau| > |\mathcal{R}_j \cap \mathcal{R}_\tau|$. Si la collecte de ressources distantes représente un surcoût non négligeable, alors τ est moins coûteuse pour i que pour j, c'est-à-dire $c_i(\tau) \leq c_j(\tau)$. Cette propriété sera formalisée au chapitre 4.

Pour connaître le coût de l'ensemble des tâches allouées à un agent, il est nécessaire d'agréger le coût de chacune d'elles. Dans notre cas, le coût d'une tâche pour un agent ne dépend pas des

autres tâches qui lui sont allouées. Un ensemble de tâches représente donc une charge de travail définie comme suit.

Définition 1.6 (Charge de travail).
La charge de travail (workload) de l'agent i pour l'allocation de tâches P est donnée par la fonction additive :

$$w_i(P) = \sum_{\tau \in P_i} c_i(\tau) \tag{1.8}$$

On suppose maintenant que $c_i(\tau)$ représente le temps d'exécution de la tâche τ pour l'agent i. $w_i(P)$ représente alors le temps nécessaire à l'agent i pour exécuter l'ensemble des tâches qui lui sont allouées dans P. Le délai de réalisation d'une tâche est le temps écoulé entre son allocation et son exécution.

Définition 1.7 (Délai de réalisation).
Soit P une allocation de tâches et $\tau_{i_k} \in P_i$ la $k^{ième}$ tâche exécutée par l'agent $i \in \mathcal{A}$. Le délai de réalisation de la tâche τ_{i_k} est défini comme :

$$t_i(\tau_{i_k}) = \sum_{l=1}^{k} c_i(\tau_{i_l}) = t_i(\tau_{i_{k-1}}) + c_i(\tau_{i_k}) \tag{1.9}$$

De cette manière, calculer le délai de réalisation de la dernière tâche exécutée par l'agent i revient à calculer le temps qu'il met pour traiter l'ensemble des tâches qui lui ont été allouées. Ainsi, si $l = |P_i|$, alors $t_i(\tau_{i_l}) = w_i(P)$.

Comme énoncé dans la section précédente, il existe de nombreux objectifs liés aux problèmes d'allocation de tâches. Cependant, les deux objectifs les plus courants sont :

- maximiser le nombre de tâches exécutées à tout moment, ce qui revient à minimiser le délai de réalisation de chacune des tâches. Cet objectif convient quand les tâches possèdent des dates butoirs. Il convient également aux situations où il est coûteux d'utiliser le système et où l'on souhaite traiter un maximum de tâches à moindre coût. Un autre cas de figure est celui où plusieurs utilisateurs sollicitent le système distribué, et que chaque utilisateur souhaite que ses tâches soient traitées au plus vite. On appelle cet objectif le *flowtime*.
- minimiser le temps pour traiter l'ensemble des tâches. Cela revient à minimiser le délai de réalisation de la dernière tâche ou, autrement dit, à minimiser la charge de travail maximale du système. On appelle cet objectif le *makespan*.

Définition 1.8 (*Flowtime*).
Soit P une allocation de tâches. Le flowtime de P mesure le temps de réponse moyen des tâches du système (c'est-à-dire le temps moyen entre l'entrée d'une tâche dans le système et l'obtention de son résultat) :

$$C(P) = \frac{1}{n} \sum_{i \in \mathcal{A}} \sum_{\tau \in P_i} t_i(\tau) \tag{1.10}$$

Dans le cadre de tâches indépendantes, le *flowtime* de P est calculé en considérant l'ordre d'exécution des tâches qui minimise leurs temps de réponse, c'est-à-dire par ordre de coût croissant. Trouver l'allocation qui minimise le *flowtime* revient à résoudre un problème d'appariement bipartite avec d'un côté les n tâches et de l'autre les $n \times m$ positions dans lesquelles les tâches peuvent être exécutées par les agents. L'allocation qui minimise le *flowtime* peut être trouvée en temps polynomial [Horn 1973, Bruno *et al.* 1974].

Définition 1.9 (*Makespan*).
Soit P une allocation de tâches. Le makespan de P mesure le temps de réponse maximum de l'ensemble des tâches du système ou, de manière équivalente, la charge de travail maximum du système :

$$C_{max}(P) = \max_{i \in \mathcal{A}}(\max_{\tau \in P_i}(t_i(\tau))) = \max_{i \in \mathcal{A}}(w_i(P)) \tag{1.11}$$

Dans [Graham 1969], l'auteur considère le problème $P_m||C_{max}$, dans lequel le coût d'une tâche est unique, quelque soit l'agent auquel elle est allouée (c'est-à-dire $\forall i, j \in \mathcal{A}, c_i(\tau) = c_j(\tau)$). Il y prouve l'existence de bornes concernant le gain de *makespan* qui peut être amené par la modification d'un des paramètres du problème c'est-à-dire l'ordre d'allocation des tâches, le coût des tâches, les contraintes de précédences et le nombre d'agents. Par exemple, il étudie le cas dans lequel les tâches sont indépendantes et où, plutôt que d'être allouées, elles sont accessibles par tous les agents, triées par ordre de coût croissant. Ainsi, une fois qu'un agent termine d'exécuter sa tâche, il va piocher la tâche la plus coûteuse restante et en fait sa tâche courante. Si l'on note P l'allocation obtenue par cet algorithme, et P^{mks} l'allocation qui minimise le *makespan*, le ratio entre le *makespan* obtenu ($C_{max}(P)$) et le *makespan* de l'allocation optimale ($C_{max}(P^{mks})$) possède la borne supérieure suivante :

$$\frac{C_{max}(P)}{C_{max}(P^{mks})} \leq \frac{4}{3} - \frac{1}{3m}$$

Exemple 1.1 (Allocations optimales).
Soit $S = (\mathcal{A}, \mathcal{E}, \mathcal{R}, d)$ un système distribué complet tel que $\mathcal{A} = \{1, 2, 3\}$. Soit $(S, \mathcal{T})$ un problème d'allocation de tâches tel que $\mathcal{T} = \{\tau_1, \ldots, \tau_5\}$. Pour cet exemple, il n'est pas nécessaire de connaître les ressources ni leur répartition parmi les agents. Le tableau 1.1 présente le coût de chacune des tâches pour les trois agents. Les allocations P^{flw} et P^{mks} sont respectivement l'allocation de flowtime minimum et l'allocation de makespan minimum :

- $C(P^{flw}) = \frac{(8+22)+(8+29)+12}{5} = 15,8$ *avec,*
  - $P_1^{flw} = \{\tau_5, \tau_2\}$,
  - $P_2^{flw} = \{\tau_1, \tau_3\}$.
  - $P_3^{flw} = \{\tau_4\}$.
- $C_{max}(P^{mks}) = 24$ *avec,*
  - $P_1^{mks} = \{\tau_2, \tau_5\}$, $w_1(P^{mks}) = 22$
  - $P_2^{mks} = \{\tau_3\}$, $w_2(P^{mks}) = 21$
  - $P_3^{mks} = \{\tau_1, \tau_4\}$, $w_2(P^{mks}) = 24$

	τ_1	τ_2	τ_3	τ_4	τ_5
$c_1(\tau_k)$	10	14	35	20	8
$c_2(\tau_k)$	8	13	21	20	11
$c_3(\tau_k)$	12	13	32	12	13

TABLEAU 1.1 : Fonction de coût associée à l'exemple 1.1.

La figure 1.1 illustre les charges de travail de chacun des trois agents pour l'allocation P^{mks} issue de l'exemple 1.1. On y constate que l'allocation qui offre le *makespan* optimal correspond également à un équilibrage des charges optimal parmi les agents.

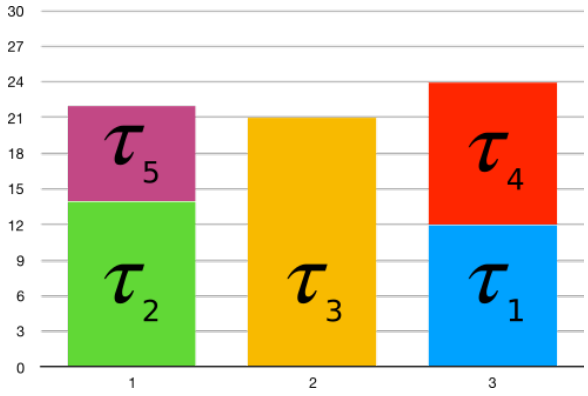


FIGURE 1.1 : Charges de travail induites par l'allocation P^{mks} de l'exemple 1.1. On trouve les trois agents en abscisse. La charge de travail d'un agent, qui est calculée en additionnant les coûts des tâches allouées à cet agent, est représentée en ordonnée.

Je viens de présenter les définitions et notions nécessaires à la compréhension du problème d'allocation de tâches abordé dans cette thèse. La prochaine section propose d'explorer plusieurs travaux de la littérature et de présenter brièvement divers contextes d'allocation de tâches dans un système distribué.

1.3 Processus d'allocation de tâches

Chaque domaine d'application nécessite la définition de ses propres propriétés (structures des tâches et ressources, agents coopératifs ou en compétition) et de ses propres contraintes (topologie du réseau, qualité et validité d'une solution, nécessité de réallouer les tâches). Cependant, les mécanismes d'allocation de tâches se classent principalement selon deux dimensions :

- statique vs. dynamique ;
- centralisé vs. décentralisé.

Les processus d'allocation de tâches se distinguent par la nature des informations qu'ils utilisent. Alors que les processus d'allocation **dynamiques** s'adaptent à l'état courant du système, les processus d'allocation statiques n'en tiennent pas compte et considèrent son comportement moyen. Ainsi, les processus statiques ont l'avantage d'avoir des paramètres issus des exécutions précédentes qui facilitent leur analyse mathématique. Cependant, ils peuvent se révéler moins performants lorsqu'ils rencontrent des tâches (ou profils de tâches) pour la première fois. En

effet, il est compliqué de connaître le coût d'une tâche pour un agent avant que celui-ci ne l'ait exécutée au moins une fois. De plus, les systèmes statiques ne s'adaptent pas aux fluctuations de performances qui sont susceptibles d'apparaître en cours d'exécution. À l'inverse, les processus dynamiques sont plus adaptatifs, mais plus complexes à concevoir. Ils posent le problème de la collecte d'informations sur le système en cours d'exécution et du surcoût que cela peut représenter. Par exemple, dans [Zhang *et al.* 1997], les auteurs illustrent ces différences en comparant deux stratégies d'allocations statiques avec deux stratégies d'allocation dynamiques.

Les processus d'allocation de tâches se distinguent également par la manière dont est construite l'allocation de tâches. Un processus d'allocation centralisé délègue la construction de l'allocation à un seul agent. Il est donc nécessaire que cet agent possède l'ensemble des informations disponibles sur le système. Cela permet de créer une allocation optimale mais peut aussi représenter un coût important en mémoire ou un goulot d'étranglement pour l'ensemble du système. Pour ces raisons, de tels processus sont souvent limités par la taille des instances qu'ils sont en mesure de gérer. De plus, en cas de défaillance de cet agent, la production d'une allocation devient impossible. À l'inverse, un processus **décentralisé** produit une allocation par la somme d'interactions locales des agents. Cela permet d'avoir un système réactif, adaptatif aux variations du système et qui passe facilement à l'échelle. Cependant, les allocations produites sont le plus souvent sous-optimales (ou localement optimales) et le dialogue entre agents peut représenter un coût supplémentaire qui pourrait s'avérer important.

Les sous-sections qui suivent répartissent des exemples de processus d'allocation de tâches issus de l'état de l'art selon ces caractéristiques.

1.3.1 Processus statiques centralisés

Les processus statiques centralisés sont caractérisés par la mise en œuvre d'un algorithme qui utilise des données déjà disponibles sur les tâches et les agents pour produire l'allocation de tâches. Ces données sont supposées exactes et ne sont pas remises en causes en cours d'exécution. Certains des travaux suivants prennent en compte de nouvelles tâches à mesure qu'elles sont soumises au système. Pour autant, elles ne sont pas catégorisées comme dynamiques car les informations utilisées pour allouer ces nouvelles tâches restent fixées et issues d'une connaissance préalable du système.

Une première approche consiste à exprimer le problème d'allocation de tâches dans un système distribué comme un problème de programmation linéaire (ou optimisation linéaire). L'exercice consiste alors à traduire le problème et son objectif en différentes fonctions linéaires et à relâcher certaines contraintes afin de rendre les algorithmes d'une complexité acceptable. [Lenstra *et al.* 1990] présentent un algorithme polynomial de programmation linéaire pour résoudre le problème $R_m||C_{max}$. Cet algorithme délivre une solution qui garantit un *makespan* d'au pire deux fois l'optimum. Les auteurs prouvent aussi qu'aucun algorithme polynomial ne peut garantir un facteur inférieur à $\frac{3}{2}$, à moins que P = NP. Ils étudient les bornes du problème dans des situations spécifiques. Par exemple, il peut être résolu en temps polynomial dans le cas où $\forall i \in \mathcal{A}, \forall \tau \in \mathcal{T}, c_i(\tau) \in \{1, 2\}$. Par la suite, [Martello *et al.* 1997] proposent des algorithmes exacts et approximatifs qui surpassent l'algorithme de [Lenstra *et al.* 1990].

Une seconde approche consiste à utiliser la théorie des jeux pour trouver une solution au problème d'allocation. Par exemple, dans [Grosu *et al.* 2002] et [Penmatsa & Chronopoulos 2011], le

problème d'allouer les tâches de sorte à minimiser le *flowtime* est formalisé comme un jeu collaboratif et l'utilisation de la solution de négociation de Nash (en anglais, *Nash bargaining solution*) permet d'obtenir un optimum de Pareto.

D'autres travaux abordent le problème d'allocation de tâches grâce aux processus stochastiques. Dans [Abdallah & Lesser 2005], l'exécution complète d'une tâche donne lieu à une récompense. Les médiateurs sont des agents qui décident quelles tâches exécuter, à quels agents allouer quelles tâches, et quand abandonner l'exécution d'une tâche en cours pour commencer une nouvelle tâche. Les auteurs se servent des processus décisionnels de Markov pour apprendre le meilleur comportement des médiateurs afin de maximiser les récompenses obtenues. Par exemple, cette solution utilise des informations statiques telles que la probabilité qu'une nouvelle tâche arrive dans le système, le type des tâches, les agents requis pour exécuter un type de tâche donné ou encore les récompenses qu'offrent les tâches.

D'autres approches consistent à construire diverses heuristiques. [Ibarra & Kim 1977] présentent des heuristiques simples pour résoudre $R_m||C_{max}$. Ces heuristiques ont une complexité d'au plus $O(n^2)$ mais offrent des résultats pouvant être jusqu'à cinq fois pire que l'optimum. Ils étudient également le cas où $m = 2$ et présentent un algorithme de complexité $n\ log\ n$ ($n = |\mathcal{T}|$) pour ce cas particulier. Enfin, les auteurs confirment l'efficacité de l'algorithme LPT pour $P_m||C_{max}$. Cet algorithme consiste à allouer les tâches par ordre de coût décroissant et à attribuer la tâche courante à l'agent le moins chargé. L'heuristique ECT (*Early Completion Time*) présentée dans cet article est ensuite réutilisée par [Hariri & Potts 1991] dans leur heuristique à deux phases. La première de ces phases applique une méthode de programmation linéaire pour construire une allocation partielle. La seconde phase alloue les tâches restantes à l'aide de l'heuristique ECT.

[Yin *et al.* 2007] proposent d'aborder le problème d'allocation de tâches dans un système distribué par une métaheuristique qui mêle un algorithme génétique à une heuristique *hill-climbing*. Le principe est de générer une population de solutions potentielles au problème et de les améliorer par itérations successives jusqu'à l'obtention d'une condition d'arrêt, par exemple la qualité des solutions ou nombre d'itérations. À chaque itération, les solutions sont évaluées. Celles qui ne correspondent pas à une solution valide ou qui obtiennent un score trop faible sont éliminées, et les solutions restantes (les meilleures donc) sont croisées et mutées pour en générer de nouvelles. La taille de la population restant stable, les solutions qui s'y trouvent s'améliorent à chaque itération. Toujours en mêlant algorithme génétique et heuristique de recherche locale, [Kang *et al.* 2010] proposent d'utiliser une métaheuristique inspirée du mécanisme d'accouplement des abeilles pour optimiser la fiabilité d'une allocation de tâches.

Plus récemment, [Jiang & Li 2011] identifient deux approches d'allocation et d'équilibrage des charges. Ces deux approches nécessitent des ressources partageables ainsi qu'une connaissance exacte de la répartition des ressources. La distinction se trouve dans le critère de choix pour allouer une tâche à un agent. La première approche considère uniquement les ressources qui sont propres aux agents, la seconde considère également leurs accointances et les ressources accessibles via la coopération entre agents. Les auteurs considèrent l'équilibrage des charges uniquement par rapport au nombre de tâches allouées aux agents, c'est-à-dire $\forall i \in \mathcal{A}, \forall \tau \in \mathcal{T}, c_i(\tau) = 1$. Dans [Jiang & Huang 2012], les auteurs proposent une méthode d'équilibrage des charges qui mixe les deux adages « les riches deviennent plus riches » et « les gagnants ne remportent pas tout ». Leur méthode propose donc de gérer l'allocation des nouvelles tâches en décidant s'il vaut mieux les donner aux agents qui ont accès à plus de ressources et qui sont susceptibles de les

traiter plus rapidement ou les donner aux agents les moins chargés pour favoriser l'équilibrage des tâches. Cette méthode se base uniquement sur l'expérience des agents et ne demande pas de calculs coûteux. De plus, elle ne nécessite pas d'avoir des informations exactes sur l'état et la disponibilité des ressources à tout instant (ce qui est difficile et/ou coûteux en pratique). Cependant, cette méthode demande une mise en cache des ressources, c'est-à-dire la duplication des ressources afin de réduire leur temps d'accès. Les auteurs supposent également connaître exactement le temps d'exécution d'une tâche ainsi que le coût de communication entre deux agents.

1.3.2 Processus statiques décentralisés

Les agents qui agissent dans les processus statiques décentralisés produisent l'allocation grâce à leurs interactions mais utilisent des informations préalablement disponibles sur le système. Ces informations ne sont pas remises en questions lors de l'exécution.

Dans le cadre d'équilibrage des charges multi-utilisateurs, toutes les tâches qui arrivent dans le système n'appartiennent pas au même utilisateur. [Grosu & Chronopoulos 2005] et [Penmatsa & Chronopoulos 2011] souhaitent que les tâches d'approximativement la même taille aient un temps de réponse approximativement égal pour tous les utilisateurs. Pour cela, ils formalisent le problème comme un jeu non coopératif et présentent un algorithme distribué pour construire une solution qui soit un équilibre de Nash : les agents proposent une répartition des tâches sur les différents nœuds de calculs et ajustent itérativement l'allocation jusqu'à obtenir un équilibre de Nash. Les auteurs utilisent des connaissances *a priori* et des informations statistiques sur le système.

1.3.3 Processus dynamiques centralisés

Dans les processus dynamiques centralisés, un agent, souvent appelé coordinateur ou superviseur, observe le système au cours de l'exécution et produit l'allocation des tâches pour l'ensemble des autres agents. Un tel mécanisme est mis en œuvre grâce à des messages qu'envoient régulièrement les agents du système au superviseur. Ces messages contiennent des données prélevées par les agents sur leur situation, par exemple leur charge de travail ou la disponibilité de leurs ressources. Ils permettent de mettre à jour les informations du superviseur sur le système pour la suite de l'allocation ou pour une potentielle réallocation des tâches.

[Su *et al.* 2018] proposent une approche prédictive et réactive qui cherche à minimiser le *makespan*. La proposition des auteurs consiste dans un premier temps à allouer les tâches grâce à une heuristique puis à déterminer une séquence d'exécution des tâches en prenant en compte des mesures de flexibilité afin de prévoir de probables événements indésirables. Ils sont donc capables de produire des allocations révisables en cas d'événements imprévus afin d'éviter des réallocations trop coûteuses.

1.3.4 Processus dynamiques décentralisés

Les processus dynamiques décentralisés produisent des allocations qui résultent d'interactions entre les agents. L'aspect dynamique de ces systèmes est souvent lié au fait que les agents

possèdent une base de connaissances qu'ils mettent à jour en cours d'exécution grâce à une observation de l'environnement et/ou grâce aux résultats de leurs interactions avec leurs pairs.

[Shehory & Kraus 1998] présentent des algorithmes d'allocation de tâches par formation de coalitions : des agents coopératifs acceptent d'exécuter des tâches en groupe afin de profiter à l'ensemble du système. Leur objectif est de maximiser le bénéfice, c'est-à-dire le rapport entre le coût et le nombre de tâches réalisées. Ces algorithmes sont utilisés dans le cas particulier où un agent est moins efficace pour réaliser une tâche seul plutôt qu'en groupe. De plus, ces algorithmes supposent qu'il est acceptable de ne pas réaliser certaines tâches pour favoriser la rentabilité du système. Dans [Kraus *et al.* 2003], le fait qu'il est impossible de traiter une tâche par un agent seul motive la formation de coalitions. Les agents sont égoïstes et cherchent à maximiser leur profit personnel mais ils sont forcés de collaborer pour exécuter les tâches.

[Walsh & Wellman 1998] étudient le cas d'allocation de tâches dans le cadre particulier de ressources limitées. En effet, dans le cas de l'élaboration d'une chaine logistique, les ressources ne font pas partie du système en tant que tel mais sont produites par des agents. Les auteurs proposent ici un protocole de négociation entre agents afin de fixer les prix d'achats et de reventes des ressources. Ces négociations permettent de produire une chaine logistique valide et qui satisfait les prix de réserve de chacun des agents du système. [An *et al.* 2010a] proposent également un mécanisme d'allocation basé sur des lois de marché où les agents égoïstes négocient pour acheter ou vendre des ressources et cherchent à maximiser leur profit.

[Wagner *et al.* 1999] cherchent à optimiser le nettoyage d'un espace par une flotte de robots. Les robots doivent être capables de s'adapter à un environnement inconnu. Pour cela, ils découvrent l'espace à mesure qu'ils le parcourent et ils le modélisent sous forme de graphe où les nœuds représentent une surface à laver. Une mémoire partagée permet une modélisation collective de l'environnement. Les auteurs proposent ensuite d'utiliser la **stigmergie** (c'est-à-dire la communication par l'environnement) pour optimiser le parcours de l'espace à nettoyer. Ainsi, assimilés à des fourmis qui déposent des phéromones pour leurs pairs, les robots laissent une trace de leur passage sur une surface. Trois algorithmes qui utilisent ces traces sont présentés dans l'article. [Turner *et al.* 2018a] étudient également l'allocation de tâches dans une flotte de robots. Leur but est de maximiser le nombre de tâches effectuées par l'ensemble de la flotte avant que les robots tombent à court de carburant. La proposition des auteurs consiste à donner une fonction de prédiction aux robots afin qu'ils choisissent quelles tâches exécuter en cours d'exécution. Le choix des tâches à exécuter se fait via une heuristique. Ainsi, la fonction de prédiction permet aux robots de choisir quelle heuristique de sélection de tâches utiliser. Les auteurs utilisent l'apprentissage automatique supervisé (machines à vecteurs supports et réseaux de neurones) pour établir la fonction de prédiction à l'aide de données prélevées lors d'exécutions précédentes. Cependant, les robots emploient cette fonction en cours d'exécution pour choisir dynamiquement et de manière décentralisée quelle heuristique de sélection de tâche utiliser. Dans [Turner *et al.* 2018b], les auteurs étudient le temps de convergence d'un tel mécanisme et proposent de le réduire en introduisant une méthode de résolution de conflit basée sur la topologie du système distribué. Cette méthode nécessite d'avoir des données sur les tâches et requiert une phase d'apprentissage.

1.4 Synthèse

Ce chapitre présente et définit les notions nécessaires afin d'aborder le problème d'allocation de tâches considéré dans cette thèse. Il présente également des travaux de la littérature afin de donner des exemples de différents problèmes d'allocation et de décrire les types de solutions mises en place pour les résoudre. Le tableau 1.2 consigne les références citées ainsi que leurs caractéristiques. En plus de leurs objectifs, ce tableau indique pour chaque référence si elle concerne un processus centralisé ou décentralisé, statique ou dynamique, si ce processus fait appel à la coopération entre agents, s'il existe une notion de localité des ressources et si la proposition permet la réallocation des tâches. Sans rentrer dans les détails des objectifs de beaucoup de ces références, certains sont centrés sur les agents/les utilisateurs et demandent à ce qu'un maximum de tâches soient exécutées. Ils sont assimilables au *flowtime*. D'autres sont centrés sur le système et demandent un équilibre des charges entre les agents. Ils sont assimilables au *makespan*.

La résolution de problèmes dans un système distribué, et notamment de $R_m||C_{max}$, a amplement été étudié dans la littérature. Plusieurs approches existent et elles se distinguent par deux aspects : quelles informations sont utilisées pour produire l'allocation ? Et comment est-elle produite ? Les informations utilisées sont statiques si elles ne sont pas remises en cause au cours de l'allocation ou de l'exécution. Les informations statiques proviennent généralement de l'étude d'exécutions précédentes ou de certitudes sur les tâches et les agents. À l'inverse, des informations dynamiques proviennent directement des agents et sont prélevées lors de l'allocation ou l'exécution. Une allocation est produite de manière centralisée si une seule entité (un agent superviseur) possède l'ensemble des informations sur le système et décide seule de l'allocation des tâches. Une allocation est produite de manière décentralisée si elle est le résultat d'interactions (échanges, enchères, etc.) entre les agents.

Dans cette thèse, les tâches sont indépendantes, non divisibles [1] et non préemptives, les ressources sont partageables et elles sont en nombre suffisant dans le système pour que toutes les tâches puissent être exécutées. Ces caractéristiques découlent du problème applicatif sous jacent (cf. chapitre 3). Contrairement à certains travaux comme [Attiya & Hamam 2006, Yin *et al.* 2007], je suppose que le système est fiable : tous les agents sont en mesure d'effectuer toutes les tâches et aucun des agents ne peut échouer.

Il faut donc allouer n tâches à m agents hétérogènes. L'hétérogénéité des agents découle de deux faits :

1. les agents sont susceptibles d'avoir des capacités de calculs différentes ;
2. les ressources sont inégalement distribuées et la collecte des ressources est coûteuse. Ainsi même si deux agents sont homogènes en termes de capacité de calcul, si l'un possède plus de ressources nécessaires à l'exécution d'une tâche que l'autre, alors la tâche sera moins coûteuse pour le premier que le second.

En tenant compte de ces observations, notre objectif est de distribuer les tâches aux agents de sorte à minimiser le *makespan* (c'est-à-dire le temps d'exécution maximum du système). Ce problème est noté $R_m||C_{max}$ et est connu pour être NP-difficile.

Le cadre applicatif de cette thèse est le patron de conception MapReduce (cf. chapitre 3). Dans

1. La section 6.3 discute du cas particulier des tâches divisibles.

ce cadre, les tâches sont souvent très nombreuses et il est souvent trop coûteux de réunir toutes leurs informations à un endroit unique du système. De plus, les tâches sont produites à partir de données non pré-traitées. On ne peut donc présumer d'aucune connaissance préalable sur celles-ci, et il est compliqué de prédire leur coût exact pour les agents avant leur entrée dans le système. Ces imprécisions peuvent mener à des déséquilibres de charges imprévus et imprévisibles. Enfin, le trop grand nombre de tâches ainsi que leur profil non défini avant l'exécution nécessitent une stratégie d'allocation fixée (cf. section 3.2 pour plus de détails sur le partitionnement des tâches dans le cadre de MapReduce). Une allocation nécessite donc d'être corrigée une fois qu'elle a été produite.

Pour toutes ces raisons, l'approche défendue dans cette thèse est dynamique et décentralisée. Elle ne demande aucune connaissance préalable, ni aucun paramétrage spécifique aux tâches. Les agents sont collaboratifs et constituent un système distribué complet (cf. définition 1.1). Ils corrigent l'allocation tout au long de l'exécution des tâches pour obtenir un meilleur équilibre des charges de travail et améliorent ainsi le *makespan* de l'allocation. Ces corrections se font grâce à des ajustements locaux, résultats de négociations entre agents. Une telle réallocation dynamique et continue permet également de s'adapter aux variations de performance des agents et d'aligner leurs charges de travail sur leurs capacités de calculs en temps réel. Ainsi, je propose de corriger une allocation de tâches biaisée alors que les tâches sont en cours d'exécution afin d'améliorer le temps d'exécution de l'ensemble des tâches.

	Décentralisé	Dynamique		Coopération	Localité	Réallocation	Objectifs
[Lenstra *et al.* 1990]	-	-		-	-	-	*makespan*
[Hariri & Potts 1991]	-	-		-	-	-	*makespan*
[Martello *et al.* 1997]	-	-		-	-	-	*makespan*
[Grosu *et al.* 2002]	-	-		✓	-	-	∼*makespan*
[Abdallah & Lesser 2005]	-	-		-	-	-	∼*flowtime*
[Yin *et al.* 2007]	-	-		-	✓	-	∼*flowtime*
[Kang *et al.* 2010]	-	-		-	✓	-	∼*flowtime*
[Penmatsa & Chronopoulos 2011]	-	-		-	-	-	*flowtime*
[Jiang & Li 2011]	-	-		✓	✓	-	∼*makespan*
[Jiang & Huang 2012]	-	-		✓	✓	-	*flowtime*
[Grosu & Chronopoulos 2005]	✓	-		-	-	-	*flowtime*
[Su *et al.* 2018]	-	✓		-	✓	✓	*makespan*
[Shehory & Kraus 1998]	✓	✓		✓	-	-	∼*flowtime*
[Wagner *et al.* 1999]	✓	✓		✓	-	-	couverture d'un graphe
[Walsh & Wellman 1998]	✓	✓		-	✓	-	trouver une allocation
[Kraus *et al.* 2003]	✓	✓		-	-	-	∼*flowtime*
[An *et al.* 2010a]	✓	✓		-	✓	-	∼*flowtime*
[Turner *et al.* 2018a]	✓	✓ / -		✓	-	-	nombre de tâches exécutées
MAS4Data	✓	✓		✓	✓	✓	*makespan*

Tableau 1.2 : Ensemble des caractéristiques des références citées dans le chapitre 1. MAS4Data note la proposition de cette thèse. ∼*flowtime* désigne un objectif similaire au *flowtime*. ∼*makespan* désigne un objectif similaire au *makespan*. Les caractéristiques de ces références sont avérées (✓) ou non avérées (-).

Chapitre 2

Négociation multi-agents pour l'allocation de tâche

Sommaire

2.1 Introduction

La négociation multi-agents intervient dans les situations où plusieurs agents sont en compétition pour satisfaire leurs buts tout en **collaborant** [Schelling 1980]. Ainsi, la situation initiale est un conflit d'intérêt mêlé à un besoin de coopération. Le but est de réaliser un échange ou de conclure une affaire, c'est-à-dire de trouver un compromis attractif pour les participants. Chacun d'eux souhaite maximiser un gain via un échange mutuellement bénéfique. Ce gain peut être **individuel** dans le cas où il ne bénéficie qu'à un seul agent ou **social** s'il bénéficie à l'ensemble des agents.

La majorité des modèles informatiques de négociation n'explore qu'un nombre restreint de dimensions et d'étapes envisagées par les sciences humaines et sociales [Guttman *et al.* 1998, Lopes *et al.* 2008]. Les modèles informatiques se limitent à la recherche d'un accord, c'est-à-dire au « processus grâce auquel plusieurs parties prenantes aboutissent à une décision collective en verbalisant leurs demandes et convergent vers un accord par succession de concessions ou recherche de nouvelles alternatives » [Pruitt 1981]. Ainsi la pré-négociation (détection des conflits, définition de l'agenda et de l'objet, etc.) et l'après-négociation (évaluation, sanction, etc.) sont souvent écartées.

L'émergence de la négociation dans la littérature en Informatique, en particulier en Intelligence Artificielle, a deux objectifs majeurs : d'une part, la résolution des conflits entre humains [Aydogan *et al.* 2014] ; d'autre part, la conception de systèmes distribués. Dans le premier cas, un agent connaît les préférences d'un participant, il négocie pour l'intérêt de cette personne. De tels systèmes d'aide à la négociation visent à réduire l'effort cognitif des parties prenantes dans l'exploration des possibles. Dans le second cas, un agent représente une entité (souvent matérielle) du système distribué et la négociation permet **la résolution distribuée d'un problème** [Kraus 1997]. Cette forme d'interaction permet de coordonner des entités faiblement couplées ayant un but commun dans la recherche d'une solution [Durfee *et al.* 1989]. Pour certains objectifs, par exemple l'optimisation du *makespan* d'une allocation de tâches (cf. chapitre 1), les agents qui négocient pour la résolution distribuée de ce problème sont coopératifs : ils ont un but commun qui prime sur l'intérêt individuel. À l'inverse, dans les systèmes de négociations automatiques, les agents favorisent leur intérêt individuel et sont dits « égoïstes ».

La négociation est un processus coûteux en temps et en énergie. L'un des enjeux d'un processus de négociation est son efficacité. Celle-ci peut se mesurer grâce au temps écoulé avant l'élaboration d'un accord ou grâce au nombre de messages nécessaire avant de parvenir à l'accord.

L'objectif de ce chapitre est de définir les notions de la négociation multi-agents utilisées dans cette thèse en proposant une grille d'analyse inspirée de [Morge 2018]. Je commence par y présenter les ingrédients essentiels à la négociation dans la section 2.2. Ensuite, la section 2.3 expose comment le résultat de la négociation peut être évalué d'un point de vue collectif. Dans la section 2.4, les différents critères qui régissent un processus de négociation sont présentés. Ensuite, la section 2.5 discute plus en détail des enjeux de la négociation concurrente dans un système multi-agents, c'est-à-dire comment gérer plusieurs négociations simultanées. Enfin, la section 2.6 précise le type de négociation et les métriques utilisées dans cette thèse.

2.2 Ingrédients de la négociation

Afin d'analyser, de concevoir et d'implémenter un système de négociation, il est nécessaire d'en identifier les ingrédients essentiels, à savoir :

1. l'objet de la négociation, c'est-à-dire l'ensemble de ses résultats potentiels ;
2. les parties prenantes, c'est-à-dire les agents avec leurs propres objectifs et préférences potentiellement conflictuels ;
3. le protocole d'interaction, c'est-à-dire les règles qui régissent les échanges entre les agents ;
4. les stratégies individuelles, c'est-à-dire le comportement des agents.

2.2.1 Objet de négociation

L'objet de la négociation est le sujet sur lequel porte les échanges. On peut distinguer :

- la négociation d'un appariement entre agents pour associer les agents les uns avec les autres [Gale & Shapley 2013] ;
- la prise de décision collective où les agents essayent de sélectionner conjointement un candidat, un plan d'actions ou un ensemble de buts mutuellement acceptables [Delecroix *et al.* 2016] ;

- **la négociation de tâches** ou de ressources qui vise à établir une (ré-)allocation au sein du groupe d'agents dont les coûts/bénéfices sont évalués individuellement [Endriss *et al.* 2006, Nongaillard & Mathieu 2011].

L'espace des résultats possibles, que l'on appelle également accords, est une variable ou un ensemble de variables discrètes ou continues. L'ensemble non-vide des accords possibles représente toutes les alternatives possibles à la situation courante. Toutes les valeurs possibles de l'espace des accords ne sont pas forcément des solutions accessibles via la négociation. On dit que la négociation est multi-attributs si elle porte sur plus d'une dimension, elle est mono-attribut sinon. L'objet de négociation peut s'avérer complexe et nécessiter un langage logique pour représenter les accords potentiels [Parsons *et al.* 1998]. Dans le cadre de cette thèse, les négociations servent à la réallocation de tâches. Un accord entre deux agents porte sur la tâche transférée du lot de tâches du premier vers le lot de tâches du second.

2.2.2 Parties prenantes

Les parties prenantes (ou *stakeholders* en anglais) peuvent être deux ou plus. On parle respectivement de négociation bilatérale ou de négociation multi-parties. On peut également considérer une négociation un-à-plusieurs constituée de multiples négociations bilatérales où un agent négocie un-à-un avec un ensemble de ses pairs (cf. section 2.2.3).

De par la nature compétitive de la négociation, les agents évaluent de manière individuelle les accords potentiels. Ces évaluations peuvent être contradictoires selon les préférences des agents. [Greco *et al.* 2016] proposent un état de l'art sur la modélisation des préférences des agents.

Préférences ordinales

Pour représenter les préférences d'un agent, une première approche consiste à employer une relation qui permet de comparer les alternatives les unes aux autres ainsi que le désaccord. Une telle relation de préférence est binaire et transitive [Arrow 1958]. Ainsi, un agent rationnel cherche à sélectionner une alternative optimale si la relation de préférence est totale, ou non-dominée si la relation de préférence est partielle.

Dans un graphe de préférences, les sommets sont des alternatives et les arcs matérialisent les préférences entre elles. Quand l'objet de la négociation est multi-attributs, la représentation de ce graphe devient coûteuse. Par exemple, 10 attributs et 10 valeurs pour chaque attribut nécessitent 10 milliards (10^{10}) de nœuds. Toutefois, l'hypothèse de l'indépendance des préférences sur les critères est souvent rejetée au profit de celle moins forte du *Ceteris Paribus* ou « toutes choses étant égales par ailleurs ». Sous cette hypothèse, les préférences ne sont peut être pas indépendantes mais les dépendances potentielles sont circonscrites. Les préférences se réfèrent donc à un nombre restreint d'alternatives qui ne sont spécifiées que dans une mesure limitée. La structure de l'ensemble des alternatives dépend des besoins et des buts des agents [Hansson 1996]. Ainsi les CP-nets constituent un modèle graphique pour la représentation compacte des préférences [Boutilier *et al.* 2011]. Plus spécifiquement, les graphes d'utilité [Robu *et al.* 2005] et les langages de préférences d'enchères [Boutilier & Hoos 2001] sont des représentations compactes pour la négociation de ressources.

Préférences cardinales

L'approche utilisée dans cette thèse consiste à capturer les préférences des agents à l'aide d'une fonction qui associe une mesure à un accord. On dit que cette fonction est une fonction d'utilité si elle correspond à un profit. À l'inverse, c'est une **fonction de coût** si elle représente une perte. Typiquement, une fonction d'utilité est utilisée dans le cadre d'allocations de ressources alors qu'une fonction de coût est utilisée dans le cadre d'allocations de tâches. Une telle fonction peut également être accompagnée d'une valeur de réserve, c'est-à-dire le seuil à partir duquel un agent est prêt à négocier.

Exemple 2.1 (Allocations de tâches (1)).
Soit $\mathcal{A} = \{1, 2\}$ *un ensemble de deux agents et* $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$ *trois tâches indivisibles et non partageables*[a]*. On donne* c_i *la fonction de coût de l'agent* i *telle que :*

$$\begin{array}{lll} c_1(\tau_1) = 9 & c_1(\tau_2) = 6 & c_1(\tau_3) = 4 \\ c_2(\tau_1) = 8 & c_2(\tau_2) = 4 & c_2(\tau_3) = 6 \end{array}$$

a. cf. chapitre 1 pour la description des différentes typologies de tâche.

Pour garantir que la négociation se termine dans un temps raisonnable, une possibilité est, comme dans [Fatima *et al.* 2001], d'ajouter une date butoir à partir de laquelle le résultat de la négociation est considéré comme un désaccord. Afin de modéliser cette notion d'urgence chez les agents, certains travaux introduisent un taux de réduction (resp. d'accroissement) qui fait baisser (resp. augmenter) l'utilité (resp. le coût) d'un accord au cours du temps.

Quand l'objet de la négociation est multi-attributs, les fonctions d'utilité/de coût le sont aussi. [Keeney & Raiffa 1993] introduisent la fonction d'utilité additive, c'est-à-dire une somme pondérée d'utilités pour chaque critère. Cela suppose que les préférences sur les attributs sont indépendantes. Dans le cas contraire, une fonction k-additive capture la synergie entre les attributs en évaluant l'utilité de chaque alternative comme la somme des utilités de chaque attribut plus une valeur qui dépend des rapports de préférence de l'agent entre plusieurs attributs [Grabisch 1997]. D'une manière plus générale, l'approche hypercube pondérée de [Ito *et al.* 2008] permet de représenter n'importe quelle fonction d'utilité multi-attributs définie sur des domaines discrets.

Exemple 2.2 (Allocations de tâches (2)).
Dans l'exemple 2.1, le coût d'une tâche pour un agent ne dépend pas des autres tâches qui lui sont allouées. Soit P_i *l'ensemble de tâches alloué à l'agent* i *dans l'allocation* P*, on note* $w_i(P)$ *la fonction additive qui calcule le coût de* P_i*. Le tableau 2.1 présente le coût de chaque allocation pour les agents* 1 *et* 2*.*

En résumé, les fonctions d'utilité ou de coût constituent un modèle cardinal et quantitatif des préférences, alors que la relation de préférence est un modèle ordinal et qualitatif. Les préférences cardinales induisent les préférences ordinales. Dans la suite de ce chapitre comme dans le reste de cette thèse, restreignons nous à l'utilisation d'une fonction de coût.

	$w_1(P)$	$w_2(P)$
$P_1 = \emptyset$, $P_2 = \{\tau_1, \tau_2, \tau_3\}$	0	18
$P_1 = \{\tau_1\}$, $P_2 = \{\tau_2, \tau_3\}$	9	10
$P_1 = \{\tau_2\}$, $P_2 = \{\tau_1, \tau_3\}$	6	14
$P_1 = \{\tau_3\}$, $P_2 = \{\tau_1, \tau_2\}$	4	12
$P_1 = \{\tau_1, \tau_2\}$, $P_2 = \{\tau_3\}$	15	6
$P_1 = \{\tau_1, \tau_3\}$, $P_2 = \{\tau_2\}$	13	4
$P_1 = \{\tau_2, \tau_3\}$, $P_2 = \{\tau_1\}$	10	8
$P_1 = \{\tau_1, \tau_2, \tau_3\}$, $P_2 = \emptyset$	19	0

TABLEAU 2.1 : Coût induit par chaque allocation possible pour les deux agents de l'exemple 2.2.

2.2.3 Protocole d'interaction

Un protocole d'interaction est un ensemble de règles partagées par les agents auquel les interactions doivent se conformer. Pour la négociation, ce sont les « règles de la confrontation ». Plusieurs protocoles existent [Verrons 2004], notamment le protocole d'offres alternées [Stahl 1972, Rubinstein 1982] ou le protocole de concession monotone [Rosenschein & Zlotkin 1994]. Cette section se concentre sur le *Contract Net Protocol* (CNP).

Introduit dans [Smith 1980], le CNP est le premier protocole d'interaction proposé pour l'allocation de tâches. Selon ce protocole, les agents cherchent à établir des contrats au travers d'un type de négociation spécifique : les **enchères**. Le CNP est un protocole multi-parties où chaque agent peut jouer les deux rôles suivants : l'**initiateur** qui propose la délégation de tâches ou l'**enchérisseur** (*bidder* en anglais) qui répond à une enchère pour s'approprier des tâches. Ainsi, l'initiateur commence par soumettre un **appel à proposition** (`cfp` pour *call for proposal*) aux enchérisseurs. Chaque enchérisseur évalue ce `cfp` afin de faire une **proposition**. L'initiateur agrège alors l'ensemble des propositions reçues, alloue les tâches aux enchérisseurs qui ont généré les meilleures propositions et rejette les autres propositions. Après cela, les enchérisseurs sélectionnés informent l'initiateur de la réussite ou de l'échec du traitement de la tâche. Il se peut que les enchérisseurs fassent une succession de propositions où chaque nouvelle proposition est générée en fonction des propositions des autres enchérisseurs. On dit alors que le CNP est à plusieurs tours.

Exemple 2.3 (*Contract Net Protocol*).
La figure 2.1 présente une instance du CNP entre un initiateur et trois participants. Les messages 1 à 3 sont les appels à participation de l'initiateur. Après avoir considéré le `cfp`*, le participant 1 génère la proposition p_1 (message 4), le participant 2 ne souhaite pas répondre à l'appel (message 5) et le participant 3 génère la proposition p_3 (message 6). Une fois qu'il a reçu une réponse de l'ensemble des participants, l'initiateur rejette la proposition du participant 1 (message 7) et accepte celle du participant 3 (message 8). Enfin, le participant 3 confirme qu'il accepte le contrat proposé par l'initiateur (message 9).*

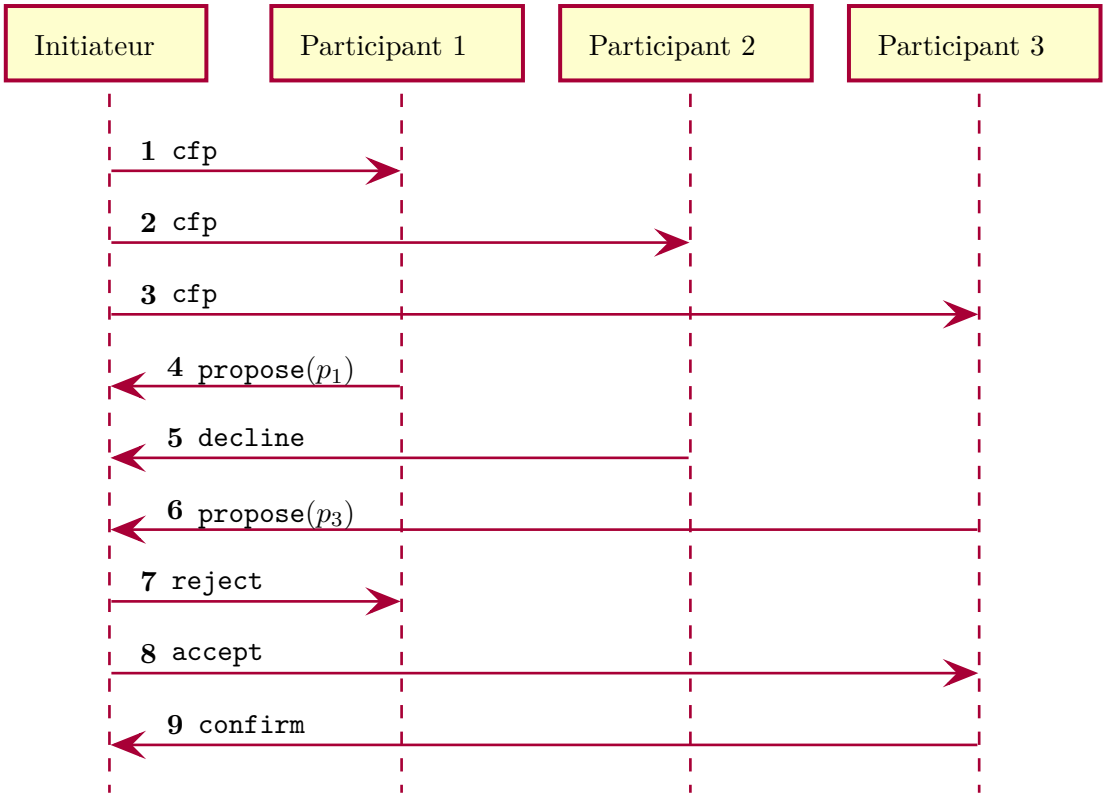


FIGURE 2.1 : Déroulement du *Contract Net Protocol* issu de l'exemple 2.3.

2.2.4 Stratégie

La stratégie d'un agent interface ses préférences et le protocole afin de générer l'appel d'offre, de produire les offres et d'évaluer les offres reçues. La stratégie définit le comportement de l'agent à chaque point de choix en déterminant les messages émis en conformité au protocole. Une stratégie est spécifique à l'environnement de négociation (objet, préférences, protocole et connaissances *a priori* sur l'interlocuteur).

Pour étudier les stratégies de négociation, on distingue trois approches : la théorie des jeux, l'approche argumentative et l'approche heuristique. La théorie des jeux propose des modèles théoriques abstraits où les propriétés sur les accords et les processus sont prouvées formellement. L'approche argumentative permet aux agents d'échanger des offres et des arguments. Ces deux approches ne sont pas étudiées dans ce document.

À l'inverse de la théorie des jeux, l'approche heuristique propose des modèles qui s'appuient sur des hypothèses réalistes et qui sont évaluées empiriquement. Une stratégie heuristique adaptée au CNP se décompose en :

1. une stratégie de formulation d'appels à propositions qui permet de générer les messages `cfp` en tant qu'initiateur ;
2. une stratégie d'offre qui permet à un enchérisseur de générer une proposition (ou de décliner l'appel d'offres) en fonction de ses préférences, de l'historique des négociations, éventuel-

lement d'une valeur de réserve, du temps et d'un modèle de leurs pairs [1] dont les agents disposent ou qu'ils construisent ;
3. une règle d'acceptabilité qui permet à l'initiateur de décider si une offre est admissible ou non ;
4. une stratégie d'apprentissage ou une stratégie d'adaptation qui s'appuie sur un modèle d'opposant et qui influe sur les trois composants précédents.

Dans le cadre de l'approche heuristique, les fonctions de décisions introduites par [Faratin *et al.* 1998] sont fondatrices. Alors que les offres reçues sont évaluées par des fonctions de coût multi-attributs, les offres générées le sont à partir d'une combinaison linéaire de fonctions simples, appelées tactiques. Une tactique calcule la valeur d'un attribut (le prix, la quantité, la qualité, etc.) d'une offre en fonction d'un critère particulier. Trois familles de tactiques sont suggérées selon les critères qu'elles considèrent :

- les tactiques temporelles qui déterminent quand un agent doit concéder ;
- les tactiques imitatives qui déterminent le comportement de l'agent en fonction de celui de son opposant ;
- les tactiques qui contraignent les offres en fonction des ressources disponibles.

Selon la fonction de concession d'une tactique temporelle, un agent débute la négociation en proposant une valeur maximale (ou minimale si l'utilité décroît avec la valeur de l'attribut) puis concède plus ou moins rapidement. On distingue deux tactiques temporelles : la tactique *Boulware* qui consiste à rester sur ses positions jusqu'à la date butoir pour y proposer sa valeur de réserve et la tactique *Conceder* qui consiste à concéder rapidement.

Selon la fonction de concession réciproque, l'agent reproduit proportionnellement le comportement de son opposant. Cette fonction peut être raffinée pour être pseudo-aléatoire ou lissée en prenant en compte le comportement de l'opposant dans une fenêtre temporelle.

Une fonction de concession qui prend en compte les ressources est similaire à une fonction de concession temporelle. En effet, le temps peut être considéré comme une ressource dont la quantité diminue de manière continue tout au long de la négociation. L'idée est ici de rendre les agents progressivement plus conciliants à mesure que la quantité de ressources disponibles diminue. Le nombre d'agents qui négocient peut également être considéré comme une ressource : plus il y a d'agents capables de négocier, moins la pression est forte. À l'inverse, s'il ne reste qu'un agent avec qui négocier, il est important de concéder pour donner lieu à un accord.

2.3 Évaluation collective des accords

Si chaque agent évalue un accord grâce à sa fonction de coût, il est également possible d'évaluer un accord selon un point de vue collectif, c'est-à-dire de lui attribuer une valeur sociale. Il existe plusieurs méthodes d'évaluation collective [Chevaleyre *et al.* 2006]. Sont décrits ici le critère de Pareto ainsi que deux autres métriques du bien-être social.

1. Souvent appelé modèle d'opposant (*opponent model*) dans la littérature [Jonker *et al.* 2017].

Optimalité de Pareto

Le critère de Pareto compare les alternatives du point de vue d'un groupe d'agents en conflit. Une alternative x domine au sens de Pareto une alternative y pour un groupe d'agents si x est au moins aussi bien que y pour tous les agents et qu'au moins un agent préfère strictement x à y. Une alternative est considérée comme optimale au sens de Pareto si elle n'est dominée par aucune autre alternative. En d'autres termes, une alternative n'est pas optimale s'il en existe une meilleure pour un agent sans qu'elle ne soit pire pour aucun des autres. Le critère de Pareto suppose un minimum d'hypothèses sur les préférences individuelles mais débouche sur un classement partiel des alternatives. En effet, même si nous sommes en mesure de comparer deux alternatives pour un agent, la satisfaction ne peut pas être comparée d'un agent à un autre [Arrow 1951].

Bien-être social

Contrairement à l'approche de Pareto, la théorie du choix social réhabilite l'analyse des comparaisons interpersonnelles [Sen 1970]. Les fonctions de choix social agrègent les utilités des agents. Plusieurs types d'agrégation existent et offrent une évaluation différente d'un même accord. Dans le cadre d'une allocation de tâches, moins le coût que représente le lot de tâches alloué à un agent est élevé, plus cet agent est individuellement satisfait. Pour rappel, on note $w_i(P)$ la charge de travail que représente l'allocation de tâches P pour l'agent i.

Améliorer le bien-être utilitaire (noté sw_u pour *utilitarian social welfare*) revient à minimiser la charge sociale utilitaire, c'est-à-dire la somme des coûts individuels des agents du système. Cette mesure représente l'insatisfaction de la société dans son ensemble.

Définition 2.1 (Contribution utilitaire).
Soit P une allocation de tâches parmi les agents de $\mathcal{A}$. La contribution utilitaire que représente P pour $\mathcal{A}$ est :

$$sw_u(P) = \sum_{i \in \mathcal{A}} w_i(P) \tag{2.1}$$

Pour faire le lien avec les notions présentées dans le chapitre 1, le *flowtime* d'une allocation de tâches (cf. définition 1.8) est une mesure la contribution utilitaire que représente cette allocation pour les agents du système. Une allocation qui minimise la contribution utilitaire des agents est optimale au sens de Pareto [Delecroix 2015, Chapitre 4].

Améliorer le **bien-être égalitaire** (noté sw_e pour *egalitarian social welfare*) revient à minimiser la charge individuelle maximale. Améliorer le bien-être égalitaire consiste à réduire les inégalités dans la population d'agents, c'est-à-dire réduire la contribution individuelle de l'agent le plus chargé.

Définition 2.2 (Contribution égalitaire).
Soit P une allocation de tâches parmi les agents de $\mathcal{A}$. La contribution égalitaire que repré-

sente P pour $\mathcal{A}$ est :

$$sw_e(P) = \max_{i \in \mathcal{A}} \left(w_i(P)\right) \tag{2.2}$$

Le *makespan* d'une allocation de tâches mesure la contribution égalitaire des agents vis-à-vis de celle-ci (cf. définition 1.9).

Exemple 2.4 (Allocations de tâches (3)).
Suite à l'exemple 2.2, soit P l'allocation telle que $P_1 = \{\tau_3\}$ et $P_2 = \{\tau_1, \tau_2\}$. Comme présenté dans le tableau 2.2, P est optimale en terme de bien-être utilitaire ($sw_u(P) = 16$). Si l'on considère le bien-être égalitaire, il faut lui préférer l'allocation P' qui alloue les tâches telles que $P'_1 = \{\tau_1\}$ et $P'_2 = \{\tau_2, \tau_3\}$ ($sw_e(P') = 10$). On remarque que P et P' sont toutes deux optimales au sens de Pareto.

	$w_1(P)$	$w_2(P)$		$sw_u(P)$	$sw_e(P)$
$P_1 = \emptyset$, $P_2 = \{\tau_1, \tau_2, \tau_3\}$	0	18		18	18
$P_1 = \{\tau_1\}$, $P_2 = \{\tau_2, \tau_3\}$	9	10		19	**10**
$P_1 = \{\tau_2\}$, $P_2 = \{\tau_1, \tau_3\}$	6	14		20	14
$P_1 = \{\tau_3\}$, $P_2 = \{\tau_1, \tau_2\}$	4	12		**16**	12
$P_1 = \{\tau_1, \tau_2\}$, $P_2 = \{\tau_3\}$	15	6		21	15
$P_1 = \{\tau_1, \tau_3\}$, $P_2 = \{\tau_2\}$	13	4		17	13
$P_1 = \{\tau_2, \tau_3\}$, $P_2 = \{\tau_1\}$	10	8		18	10
$P_1 = \{\tau_1, \tau_2, \tau_3\}$, $P_2 = \emptyset$	19	0		19	19

TABLEAU 2.2 : Mesure des contributions sociales induites par chacune des allocations considérées dans l'exemple 2.4.

2.4 Processus de négociation

Un processus de négociation représente les interactions entre agents autonomes qui, grâce à un protocole d'interactions, communiquent selon leur stratégie individuelle. On peut évaluer un processus de négociation selon les critères suivants.

Rationalité. Les agents doivent être incités à participer à la négociation. Dans le cas d'agents égoïstes, un agent est individuellement rationnel s'il accepte un accord qui améliore son bien-être personnel. Dans le cas d'agents collaboratifs, un agent est socialement rationnel s'il accepte un accord qui bénéficie au système (par exemple en augmentant le bien-être égalitaire) mais qui n'améliore pas forcément son bien-être individuel.

Monotonie. Un processus est monotone si chacune de ses itérations améliore le bien-être des agents. Dans le cas d'agents égoïstes, chaque accord améliore le bien-être individuel des participants. Dans le cas d'agents collaboratifs, chaque accord améliore le bien-être social. Dans cette thèse, les agents améliorent le bien-être égalitaire local (c'est-à-dire celui des participants) à chaque négociation réussie. Le cumul de ces améliorations locales permet d'améliorer le bien-être égalitaire de l'ensemble du système. Dans le cas d'un processus de négociation pour l'allocation

de tâches, la monotonie du processus induit également sa terminaison car il existe un nombre fini d'allocations possibles.

Distribution. Un processus est distribué s'il ne nécessite pas un point central qui collecte les offres (on parle alors de médiateur) voire qui calcule l'accord (on parle alors d'arbitre). La distribution est souhaitable pour éviter les éventuelles pannes du point central et les goulots d'étranglement.

Concurrence. Un processus de négociation est concurrent si plusieurs négociations peuvent avoir lieu simultanément. On distingue les cas où bien que plusieurs négociations aient lieu, chaque agent ne peut participer qu'à une seule négociation à la fois des cas où un même agent peut être investi dans plus d'une négociation à la fois.

Garantie de succès. Un processus de négociation est garanti de succès si un accord est atteint, quelles que soient les préférences des agents.

Complexité communicationnelle. La complexité communicationnelle d'un protocole de négociation correspond au nombre et à la taille des messages nécessaires pour atteindre un accord. Lorsque les connaissances des agents sont suffisantes, il est préférable de s'adresser directement à un interlocuteur susceptible d'offrir un accord plutôt que d'utiliser l'appel à la cantonnade (ou *broadcasting*), qui consiste à envoyer un message à tous les agents du système.

Complexité algorithmique. Il faut privilégier les stratégies individuelles dont la complexité reste praticable.

2.5 Négociation multi-agents concurrente

Certains processus de négociations autorisent les négociations concurrentes, c'est-à-dire le fait que plusieurs négociations aient lieu simultanément. Proposer une stratégie d'offre pour qu'un agent coordonne plusieurs négociations simultanées consiste à résoudre le problème de « l'enchérisseur enthousiaste » [Schillo *et al.* 2002] : si un agent s'engage dans trop de négociations en parallèle, il risque de ne pas pouvoir honorer ses engagements. En revanche, si un agent ne s'engage que dans une seule négociation, alors il ne peut pas faire d'offre dans les autres négociations en cours. Si son (unique) offre est rejetée, l'agent est passé à côté des potentiels accords que représentent les autres négociations concurrentes. Pour pallier ce problème, on peut envisager d'ajouter une phase de pré-engagement pour retarder le moment de l'engagement définitif lié à une offre [Fischer *et al.* 1995, Knabe *et al.* 2002] ou on peut estimer, comme le fait une compagnie aérienne qui pratique des sur-réservations, le risque de rupture d'engagement en terme de pénalité [An *et al.* 2010b, Habes *et al.* 2014].

Les auteurs de [Najjar *et al.* 2017] utilisent la négociation multi-agents un-à-plusieurs pour la gestion de l'élasticité du SaaS (*Software as a Service*). On appelle négociation un-à-plusieurs un processus constitué de plusieurs négociations bilatérales qui possèdent toutes un même agent participant (par exemple l'initiateur d'un `cfp` dans le CNP). En se basant sur une estimation de l'acceptabilité des services et sur l'apprentissage du modèle des pairs, la stratégie proposée ajuste le processus de négociation du fournisseur de logiciel afin de garantir un taux d'acceptabilité désiré tout en respectant des contraintes budgétaires.

[Alrayes *et al.* 2018] présente un processus de négociations concurrentes pour l'allocation de

ressources. Ce processus prend en compte la dynamicité du marché : de nouveaux agents entrent et sortent du système régulièrement. L'article fournit la spécification de la stratégie CONAN (*COncurrent Negotiating AgeNts*) comme part d'un modèle d'agent adaptatif impliqué dans de multiples négociations bilatérales et simultanées.

Récemment, [Niu *et al.* 2018] montrent la difficulté de mener de multiples négociations interdépendantes. Les défis sont de modéliser de telles négociations et d'offrir aux agents une stratégie qui tienne compte de la dépendance et de la co-occurence des négociations. L'objectif consiste donc à proposer un mécanisme générique de coordination dynamique de plusieurs fils de négociations asynchrones.

Dans le contexte de l'allocation multi-agents de tâches, l'inter-dépendance entre les fils de négociations provient en général des contraintes sur les ressources et sur les tâches. D'une part, les ressources peuvent être partagées, c'est-à-dire que la réalisation d'une tâche requiert des ressources qui ne seront plus disponibles pour la réalisation d'une autre tâche. D'autre part, les tâches peuvent être dépendantes, c'est-à-dire que la réalisation d'une tâche peut être conditionnée par la réalisation d'une autre tâche qui doit la précéder. Dans cette thèse, les ressources ne sont pas partagées et les tâches sont indépendantes, ce qui permet à la co-occurence de négociations multiples d'améliorer la réactivité du système multi-agents.

2.6 Synthèse

Les processus de négociation se distinguent d'abord par leurs ingrédients de négociation, c'est-à-dire l'objet de la négociation, les préférences des agents, le protocole de négociation et les connaissances *a priori* des agents vis-à-vis de leurs opposants. Les stratégies de négociations sont adaptées à un environnement particulier. Outre la rationalité individuelle et la distribution qui sont requises pour un processus de négociation, une stratégie de négociation se caractérise par :

- son efficacité qui détermine si l'accord atteint est optimal au sens de Pareto ou s'il maximise un bien-être social (utilitaire ou égalitaire) ;
- sa complexité algorithmique et communicationnelle, c'est-à-dire la simplicité du calcul de la stratégie et le nombre d'échanges nécessaires à un accord.

Plusieurs approches peuvent être adoptées pour la conception des stratégies d'agents. En théorie des jeux, l'environnement de négociation est restreint et les propriétés peuvent être prouvées formellement. Selon l'approche heuristique adoptée par la suite, l'environnement de négociation est réaliste et les propriétés sont évaluées empiriquement.

Dans cette thèse, les objets négociés sont des tâches que des agents collaboratifs souhaitent réallouer. Chaque agent peut essayer de négocier l'une de ses tâches alors qu'il est en train d'en exécuter une autre. Les agents tentent ainsi de réduire la charge de travail de l'agent le plus chargé, ce qui revient à améliorer le bien-être égalitaire que représente l'allocation de tâches courante. Une représentation quantitative et cardinale des préférences est utilisée : chaque tâche représente un coût différent pour chaque agent. Le coût d'une tâche pour un agent dépend du volume intrinsèque de la tâche mais aussi de la localité des ressources nécessaires à son exécution. Cette représentation numérique de la satisfaction mène à des opérations d'agrégation qui peuvent être mal interprétées. Une charge de travail élevée peut signifier qu'un agent possède beaucoup de

tâches à réaliser ou que certaines des tâches réclament l'utilisation de ressources dont ne dispose pas l'agent. De plus, même si plusieurs tâches ont le même coût pour un agent, il se peut que ce dernier possède plus de ressources que ses pairs pour l'une d'entre elles. Connaître le volume intrinsèque d'une tâche n'est pas suffisant pour déterminer par quel agent il est préférable qu'elle soit exécutée. J'explore ce problème et propose une stratégie qui tient compte de la localité des ressources dans le chapitre 6.

Le protocole de négociation utilisé par les agents est le *Contract Net Protocol*, présenté dans la section 2.2.3. Ce protocole « à un seul tour » inclut un appel à proposition (`cfp`) par l'initiateur, une proposition par enchérisseur, l'acceptation d'une proposition et le refus de toutes les autres par l'initiateur. Aucune contre-proposition n'est générée par l'initiateur. Comme les agents sont collaboratifs et qu'ils partagent le même objectif (cf. chapitre 4), ces derniers calquent leur modèle d'opposant sur eux-mêmes. Pour générer une offre susceptible d'être acceptée, un agent se met à la place de son opposant et se demande s'il accepterait l'offre dans sa situation. J'ai ajouté au CNP un ensemble de messages accusés de réception (cf. chapitre 5). Ces ajouts ont permis de rendre le processus de négociation robuste à la perte de messages et ainsi de produire des résultats empiriques dans des environnements véritablement distribués (cf. chapitre 9).

Afin de s'assurer de la dynamicité du processus, les négociations se déroulent de manières concurrente, alors que les agents exécutent les tâches. En tant qu'enchérisseur, un agent peut être impliqué dans plus d'une négociation à la fois. Pour répondre au problème de l'enchérisseur enthousiaste décrit dans ce chapitre, je propose un mécanisme de charge de travail virtuelle (cf. chapitre 5). Cela permet d'assurer qu'un enchérisseur est en mesure de remplir l'ensemble des engagements qu'il tient simultanément. De plus, chaque accord améliore le bien-être égalitaire des agents impliqués et le processus converge vers une solution stable (cf. chapitre 4).

Chapitre 3

Patron de conception MapReduce

Sommaire

3.1 Introduction

La science des données vise à traiter de grands volumes de données pour y extraire de nouvelles connaissances. Comme le potentiel technologique et la demande sociale ont augmenté, de nouvelles méthodes, de nouveaux modèles, systèmes et algorithmes sont développés. L'analyse de ces données, en raison de leur volume et de leur vitesse d'acquisition, demande de nouvelles formes de traitements. À cette intention, le patron de conception MapReduce [Dean & Ghemawat 2008] est parallélisable et utilisable, par exemple pour mettre en œuvre l'algorithme PageRank, c'est-à-dire le calcul d'index inversé, pour identifier les articles les plus populaires sur Wikipedia ou pour réaliser le partitionnement en k-moyennes. Le *framework* le plus populaire pour MapReduce est Hadoop [White 2015] mais d'autres implémentations existent comme le *framework* Spark [Zaharia *et al.* 2016]. Beaucoup de bases de données distribuées de la famille NoSQL utilisent également MapReduce comme par exemple Riak construite par Amazon Dynamo [DeCandia *et al.* 2007]. Avec ces approches, l'extraction des données ainsi que leur traitement sont distribués et exécutés sans échantillonnage (c'est-à-dire sans prétraitement des données).

Le patron de conception MapReduce représente la problématique applicative et justifie les contraintes imposées au mécanisme d'allocation (et de réallocation) de tâches décrit dans cette thèse. La section 3.2 présente le concept de MapReduce et explique comment il a été distribué. Puis, la section 3.3 discute de l'existence de biais lors des exécutions distribuées de MapReduce ainsi que des propositions de la littérature pour les corriger. La section 3.4 donne quelques détails sur l'implémentation d'un tel patron de conception et énonce certaines de ses alternatives. Enfin, la section 3.5 inscrit cette thèse dans son contexte applicatif.

3.2 Distribution du patron de conception MapReduce

La distribution du patron de conception MapReduce permet de traiter de grands volumes de données [Dean & Ghemawat 2008]. Dans ce modèle, deux fonctions empruntées à la programmation fonctionnelle sont utilisées. La fonction *map* filtre les données en entrée du système pour en extraire les informations utiles. La fonction *reduce* agrège et traite les données préalablement filtrées afin de créer les résultats. Ainsi, les jobs MapReduce sont divisés en deux ensembles de tâches, les tâches *map* et les tâches *reduce*, qui sont distribuées sur plusieurs nœuds de calcul (par exemple une grappe de PCs). Ces deux fonctions, fournies par l'utilisateur pour le traitement spécifique de ses données, possèdent les signatures suivantes [1] :

$$map : (K_1, V_1) \rightarrow [(K_2, V_2)]$$
$$reduce : (K_2, [V_2]) \rightarrow [(K_3, V_3)]$$

Parmi les nœuds de calculs, ceux qui exécutent la fonction de *map* sont appelés les *mappers* et ceux qui exécutent la fonction de *reduce* les *reducers*. La figure 3.1 illustre le flux de données et de contrôle dans un système distribué qui applique MapReduce :

1. le superviseur affecte les différents fragments des données d'entrée aux *mappers* ;
2. les *mappers* appliquent la fonction *map* sur les données d'entrée et créent les couples intermédiaires clé-valeur de type (K_2, V_2) ;
3. chaque *mapper* agrège les valeurs associées à la même clé, créant ainsi des couples intermédiaires $(K_2, [V_2])$. En fonction des caractéristiques du système de fichier, l'ensemble des valeurs associées à une même clé peuvent être regroupées dans un ou plusieurs fichiers, que l'on appelle des ***chunks***. Une fonction de partitionnement est ensuite appliquée sur les clés pour répartir les *chunks* au *reducers* de sorte que tous les *chunks* d'une même clé soient envoyés au même *reducer*. La fonction de partition peut être personnalisée pour spécifier quelle clé doit être gérée par quel *reducer* ;
4. une fois qu'un *mapper* a traité son entrée, il en informe le superviseur ;
5. lorsque tous les mappers ont traité leur entrée, le superviseur informe les *reducers* pour lancer la seconde phase ;
6. les *reducers* agrègent les couples intermédiaires des différents *mappers* et construisent les entrées définitives de la fonction de *reduce* ;
7. les *reducers* exécutent la fonction de *reduce* sur chaque groupe de valeurs associées à chaque clé (c'est-à-dire sur chaque couple $(K_2, [V_2])$) ;
8. les couples finaux clé-valeurs (K_3, V_3) sont écrits dans un fichier du système de fichiers distribué ;
9. enfin, les *reducers* informent le superviseur de la localisation du résultat final. Lorsque tous les reducers ont terminé de traiter toutes les tâches de *reduce*, le travail est terminé.

Sans prétraitement des données, il est impossible de savoir quel *mapper* va produire quelle clé. Cependant, il faut s'assurer que toutes les occurences d'une même clé soient envoyées au même

1. (X, Y) dénote un couple de valeurs dont la première est de type X et la seconde de type Y. $[X]$ dénote une liste d'éléments de type X.

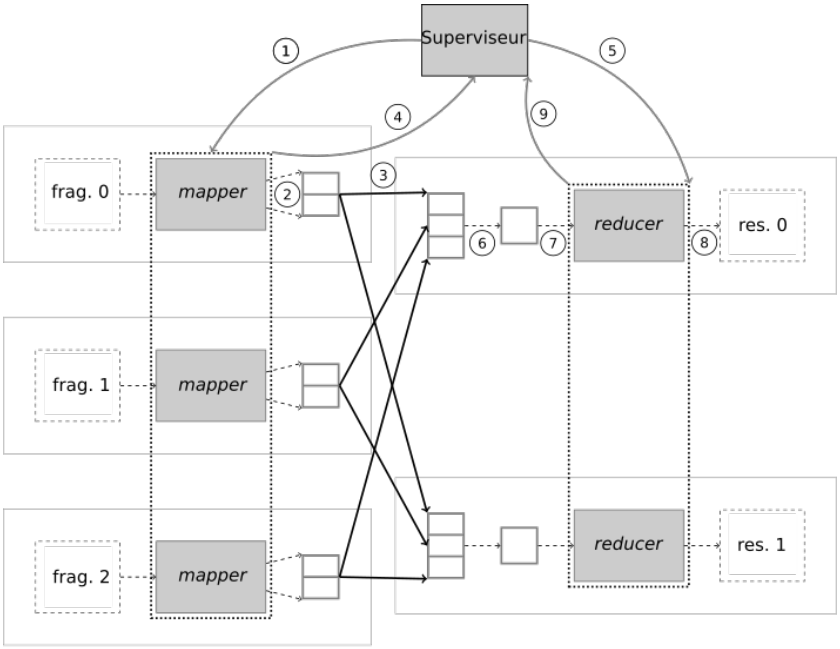


FIGURE 3.1 : Flux de données et de contrôle MapReduce.

reducer afin de garantir la cohérence du résultat final. C'est pourquoi, qu'il soit réalisé par une fonction par défaut ou par une fonction spécifique apportée par l'utilisateur, le partitionnement des tâches de *reduce* aux *reducers* est fixé *a priori*. Par exemple, pour déterminer quel *reducer* traitera une clé `cle`, la fonction de partition par défaut de Hadoop effectue un modulo sur le nombre de *reducers* :

```
cle.hashCode() % nombre_reducers
```

Exemple 3.1 (Job MapReduce : température moyenne).
Considérons le job MapReduce qui consiste à calculer la température moyenne de plusieurs villes. Pour ce faire, nous devons analyser des données qui contiennent des relevés de température parmi d'autres informations. Les données d'entrée sont présentées dans le tableau 3.1. Comme nous pouvons le voir, un relevé est composé d'une ville, d'une date et du relevé de température en lui-même.

Ces données sont analysées par un système de calculs constitué de deux nœuds n_1 et n_2. Ces deux nœuds vont successivement être utilisés comme des mappers (m_1 et m_2) et comme des reducers (r_1 et r_2).

D'abord, les données d'entrée sont divisées en fragments et confiées aux mappers. m_1 gère le `fragment 0` et m_2 gère le `fragment 1`. Afin de calculer la température moyenne par ville, les mappers exécutent la fonction de map suivante :

$$map : (Ville, ^oC, Date) \rightarrow [(Ville, ^oC)]$$

Ce faisant,

- m_1 *crée les couples* $(Lille, 27)$, $(Paris, 29)$ *et* $(Lyon, 30)$ *;*
- m_2 *crée les couples* $(Lille, 14)$, $(Paris, 12)$ *et* $(Lille, 2)$.

Ensuite, les mappers produisent des chunks pour chaque clé qu'ils ont rencontrées :

- m_1 *produit les chunks* $\rho_1^{Lille} = (Lille, [27])$, $\rho_1^{Paris} = (Paris, [29])$ *et* $\rho_2^{Lyon} = (Lyon, [30])$
- m_2 *produit les chunks* $\rho_2^{Lille} = (Lille, [14, 2])$ *et* $\rho_2^{Paris} = (Paris, [12])$.

Afin de déterminer à quel reducer envoyer les chunks, les mappers utilisent une fonction de partition. Ici, r_1 *doit gérer les tâches qui concernent les villes de Lille et Lyon, et* r_2 *doit gérer la tâche qui concerne Paris. En collectant l'ensemble des chunks associés à une clé, les reducers peuvent construire les entrées cohérentes pour la fonction de reduce afin d'avoir le résultat associé à cette clé. Par exemple,* r_1 *collecte le chunk non local* ρ_2^{Lille} *et crée l'entrée* $(Lille, [27, 14, 2])$ *qui contient toutes les informations nécessaires au calcul de la température moyenne à Lille.*

Pour finir, les reducers exécutent la fonction de reduce suivante afin d'obtenir le résultat du job :

$$reduce : (Ville, [^oC]) \rightarrow (Ville, \overline{^oC})$$

Pour la ville de Lille, r_1 *calcule le résultat final* $14,3^oC$.

	Ville	^oC	Date du relevé
`fragment 0`	Lille	27	Juin
	Paris	29	Juin
	Lyon	30	Août
`fragment 1`	Lille	14	Septembre
	Paris	12	Octobre
	Lille	2	Janvier

Tableau 3.1 : Données associées à l'exemple 3.1.

Une étape optionnelle de **combinaison** peut être ajoutée afin d'alléger les données envoyées aux *reducers*. Cette étape intervient entre les étapes 5 et 6 du flux de données décrit précédemment. Elle nécessite que l'utilisateur renseigne une fonction de combinaison qui prétraite les résultats des *mappers* avant qu'ils soient envoyés aux *reducers*. L'étape de combinaison étant optionnelle, elle doit s'intégrer de manière transparente au flux des données de MapReduce. Elle produit donc des données de type $(K_2, [V_2])$. Autrement dit, le rôle de la fonction de combinaison est généralement de réduire la taille de la liste des valeurs associées à une clé en les agrégeant. Dans le cas d'une fonction de *reduce* associative et commutative (par exemple la somme, la multiplication ou le maximum), la fonction de combinaison est identique à la fonction de *reduce*.

Exemple 3.2 (Fonction de combinaison).
Considérons ici un job MapReduce dont la fonction de reduce consiste à calculer la valeur maximale associée à une clé. Après application de la fonction de map, un reducer reçoit deux chunks qui viennent des mappers m_1 *et* m_2 *et qui concernent la clé* `cle` *:*

- $\rho_1^{cle} = (\texttt{cle}, [0, 20, 10])$

- $\rho_2^{\texttt{cle}} = (\texttt{cle}, [25, 15])$

Sans fonction de combinaison, le reducer crée l'entrée $(\texttt{cle}, [0, 20, 10, 25, 15])$*, lui applique la fonction de reduce et écrit le résultat* $(\texttt{cle}, 25)$*.*

Avec la fonction de combinaison, les chunks peuvent être traités au préalable afin de produire les chunks suivants :

- $\rho_1'^{\texttt{cle}} = (\texttt{cle}, [20])$
- $\rho_2'^{\texttt{cle}} = (\texttt{cle}, [25])$

Dans ce cas, le reducer applique sa fonction sur l'entrée $(\texttt{cle}, [20, 25])$ *et trouve le même résultat.*

Pour les fonctions de *reduce* qui ne sont ni associatives ni commutatives, il est parfois nécessaire d'adapter les fonctions de *map* et de *reduce* pour pouvoir bénéficier de la fonction de combinaison.

Exemple 3.3 (Modification d'un job MapReduce pour y intégrer une fonction de combinaison).
Dans l'exemple 3.1, la fonction de reduce consiste à faire la moyenne d'une liste d'entiers. Cette fonction n'est pas associative et peut donc difficilement profiter d'une fonction de combinaison en l'état. Cependant, si l'on modifie la fonction de map telle que :

$$map : (Ville, {}^oC, Date) \rightarrow [(Ville, ({}^oC, 1))]$$

Il est alors possible d'ajouter la fonction de combinaison :

$$combine : (Ville, [({}^oC, 1)]) \rightarrow (Ville, [(\sum {}^oC, \sum 1)])$$

La fonction de reduce produit, pour chaque ville, la somme des températures (une température par mapper grâce à la fonction de combinaison) et la divise par la somme du nombre d'enregistrements rencontrés par chaque mapper.

3.3 Biais

Les données et les flux d'entrées peuvent faire l'objet de **biais**, de pics d'activités périodiques (quotidiens, hebdomadaires ou mensuels) ou de pics d'activités déclenchés par un événement particulier. Ces distorsions peuvent être particulièrement difficiles à gérer. Dans les *frameworks* existants, une affectation efficace des tâches (c'est-à-dire la répartition des clés) demande une connaissance *a priori* de la distribution des données. Cependant, même avec un réglage précis, des biais et/ou un environnement d'exécution hétérogène peuvent remettre en cause les choix d'implémentation.

3.3.1 Description des biais

[Kwon *et al.* 2011] identifient cinq biais communs dans les applications de MapReduce. Trois de ces biais concernent la phase de *map* et deux concernent la phase de *reduce*. Ces cinq biais

ont des causes différentes mais résultent du fait que la charge de travail est mal répartie parmi les *mappers* ou les *reducers*, ce qui pénalise l'ensemble du processus. Ces biais sont issus de la distribution du modèle MapReduce mais peuvent aussi être amplifiés par l'hétérogénéité des nœuds de calculs. Après avoir évoqué les trois biais de la phase de *map*, passons aux deux biais de la phase de *reduce*.

Biais de la phase de *map*

Le premier biais de la phase de *map* provient d'enregistrements coûteux et s'explique comme suit. Lors de la phase de *map*, chaque *mapper* reçoit un fragment des données à traiter. Ces fragments contiennent plusieurs enregistrements qui servent d'entrées à la fonction de *map*. Les fragments sont créés de sorte à représenter le même volume de données et, comme le temps de traitement ne diffère généralement pas d'un enregistrement à l'autre. Ils représentent le plus souvent la même quantité de travail pour chaque *mapper*. Cependant, il existe des fonctions de *map* pour lesquelles le traitement de certains enregistrements peut réclamer plus de ressources (CPU ou mémoire) que d'autres. Ces enregistrements coûteux apparaissent donc lorsque le temps d'exécution de l'algorithme de *map* dépend de la valeur de l'enregistrement.

Le second biais de la phase de *map* est appelé le biais d'hétérogénéité. MapReduce est un opérateur unaire, c'est-à-dire qu'il s'applique normalement sur un seul ensemble de données. Cependant, il peut être utilisé pour émuler un opérateur n-aire en concaténant plusieurs ensembles de données en entrée. Chacun de ces jeux de données peut nécessiter différents traitements et mener à une distribution irrégulière des temps d'exécution des processus *map*.

Le troisième biais de la phase de *map* est appelé le biais des *map* non-homomorphes. Il provient de fonctions de *map* qui pour le même volume d'entrée ne fournissent pas forcément un résultat de même volume, ou ne le fournissent pas dans le même temps. Par exemple, opérer une jointure entre deux ensembles de données lors de la phase de *map* (*map-side join*) peut constituer une opération de *map* non homomorphe.

Ces deux derniers biais proviennent de la relative liberté laissée à l'utilisateur. En effet, ce dernier peut exécuter n'importe quel job dès lors qu'il se conforme aux spécificités des fonctions de *map* et de *reduce* décrites plus haut. Cette liberté peut, par exemple, être utilisée pour détourner l'usage prévu de la phase de *map* en créant des résultats qui dépendent de plusieurs entrées.

Biais de la phase de *reduce*

Le **biais de partitionnement** est le premier biais de la phase *reduce*. Il découle du mécanisme de répartition des clés aux *reducers*. Pour rappel, ce mécanisme est statique car l'utilisateur ne connaît pas à l'avance les clés produites par les *mappers*, ni quels *mappers* produisent quelles clés, ni la fréquence à laquelle chaque clé est générée. Bien que la fonction de hachage par défaut soit généralement efficace pour distribuer les clés de manière équitable, il se peut qu'un sous-ensemble des *reducers* se voit attribuer un nombre beaucoup plus élevé de clés et/ou des clés qui représentent une quantité de travail plus importante. Un job MapReduce est considéré comme terminé une fois que toutes les tâches de *reduce* ont été exécutées. Le temps d'exécution de la phase de *reduce* est égal au temps d'exécution du *reducer* le plus chargé ou le plus lent. Si un *reducer* se trouve bien plus chargé que les autres, alors l'ensemble du processus est pénalisé.

C'est pourquoi, il est important de corriger ce biais et de répartir équitablement la charge de travail parmi les *reducers*.

Exemple 3.4 (Biais de partitionnement).
L'ensemble de données SYNOP essentielles OMM [Météo France 2019] contient plus de 3 millions d'enregistrements météo (numéro de station, heure de l'enregistrement, température, pluviométrie, etc.) provenant de 62 stations et prélevés durant les 20 dernières années. Considérons le job qui consiste à compter le nombre de relevés par demi-degré de température :

1. *la fonction de map lit un enregistrement et retourne un couple (`temp`, 1) où `temp` correspond au relevé de température effectué par la station, arrondi au demi-degré le plus proche ;*
2. *la fonction de reduce effectue la somme `s` du nombre de relevés associés à une clé `temp` et produit le résultat (`temp`, `s`).*

Si l'on considère un système à 20 reducers, pour déterminer à quel reducer est allouée la clé `temp`, la fonction de partition par défaut effectue l'opération suivante[a] *:*

$$|\texttt{temp}.hashCode() \ \% \ 20|$$

Les clés `temp` sont des nombres à virgule, ils sont donc représenter comme des `Double`[b]*. Il s'avère qu'appliquer la fonction de partition ci-dessus sur des nombres flottants arrondis au demi-entier le plus proche répartit les clés de manière inégale*[c]*. En effet, seulement un reducer sur quatre se voit attribuer des clés. Sur les 20 reducers, seuls 5 d'entre eux seront utilisés lors de la phase de reduce (cf. figure 3.2).*

a. La valeur absolue permet d'obtenir un numéro de *reducer* positif pour les températures négatives.
b. Typer les clés `temp` en `Float` ne change pas les propriétés de cet exemple.
c. Opération réalisée avec Java 1.8.0.

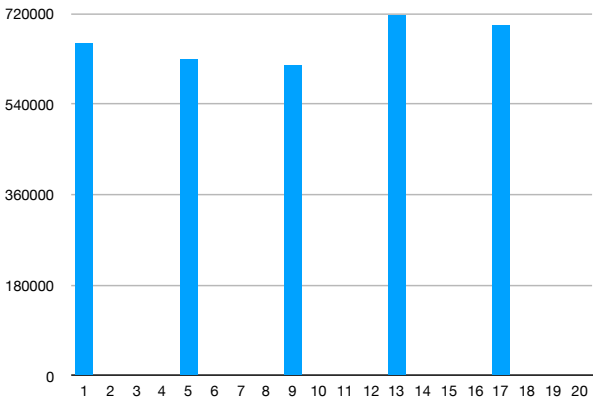


FIGURE 3.2 : Répartition de la charge de travail qui résulte de la situation expliquée dans l'exemple 3.4. On retrouve les 20 *reducers* en abscisse et leurs charges de travail en ordonnée. La charge de travail est calculée en fixant le coût d'une tâche dont la clé est `temp` au nombre de fois où un couple (`temp`, 1) a été créé par les *mappers*.

L'exemple 3.4 illustre le biais de partitionnement. L'annexe C offre un script Scala qui permet

de vérifier les propriétés décrites dans cet exemple. Il aurait été facile d'éviter cette occurence du biais de partitionnement en modifiant la fonction de partition (ou en typant les clés `temp` en chaines de caractères). Seulement, le constat d'un mauvais équilibrage de la charge de travail est postérieur à l'exécution ou nécessite des connaissances *a priori* sur les données.

Le second biais de la phase de *reduce* est le **biais des clés coûteuses**. Les tâches de *reduce* consistent à traiter un ensemble de valeurs associées à une clé. Ces valeurs ont été produites précédemment par les *mappers* et leur nombre détermine souvent la charge de travail que représente le traitement de la clé associée pour un *reducer*. Ainsi, même si les tâches sont équitablement réparties en nombre parmi les *reducers*, certaines sont susceptibles de représenter un surcoût non négligeable et donc de surcharger les *reducers* à qui elles ont été allouées. La création de clés coûteuses est souvent causée par des données distribuées selon une loi de Zipf [Li 1992, Lin 2009] à l'entrée du système ou à la sortie des *mappers*.

> **Exemple 3.5 (Biais des clés coûteuses).**
> *Considérons le job qui consiste à compter les occurences des mots dans un texte écrit en français. Comme pour l'exemple précédent, la fonction de map produit un couple* $(\texttt{mot}, 1)$ *pour chaque* `mot` *qu'il rencontre. Chaque* `mot` *donne lieu à une tâche de reduce. Le rôle des reducers est alors de faire la somme des occurences des mots rencontrés par les mappers. Selon le site* `http://eduscol.education.fr` *[Ministère de l'éducation nationale et de la jeunesse 2019], le mot « le » est le plus fréquent de la langue française. Il apparait même* 20 % *plus souvent que « de », le second mot le plus fréquent. En extrapolant ces faits, on peut déduire que la tâche qui consiste à compter les occurences du mot « le » sera au moins* 20 % *plus volumineuse que les autres tâches. Cette tâche va représenter un coût non négligeable et provoquer une surcharge de travail pour le reducer auquel elle est allouée.*

Ces biais sont connus de la littérature et plusieurs propositions existent pour les aborder. Cependant, la grande majorité des travaux s'intéressent aux biais liés à la phase de *reduce*. La section suivante présente ces travaux.

3.3.2 Traitement des biais de la phase de *reduce*

Afin d'éviter les biais décrits ci-dessus, [Kwon *et al.* 2011] proposent des bonnes pratiques telles que :

1. l'exploitation d'une connaissance préalable du domaine ou des données pour l'élaboration de la fonction de partitionnement ;
2. l'expérimentation de plusieurs méthodes de partitionnement sur des échantillons des données ;
3. l'utilisation d'algorithmes dont le temps d'exécution dépend uniquement des données d'entrées et pas de la manière dont elles sont réparties.

Ces bonnes pratiques requièrent de connaître les données, de procéder à des échantillonnages ou de redéfinir les fonctions de *map* et de *reduce*.

Pour traiter le biais de partitionnement, il faut trouver une alternative à la fonction de partition statique. [Gufler *et al.* 2011] proposent que les *mappers* produisent plus de *chunks* qu'il n'y a de *reducers*. De cette manière, un nœud contrôleur peut équilibrer la charge des *reducers* à l'aide de

cette granularité plus fine. Cette approche est centralisée et nécessite que l'utilisateur configure le nombre de *chunks* supplémentaire à produire par *mapper*, qu'il estime le coût d'un ensemble de *chunks* et qu'il choisisse l'heuristique de répartition des *chunks* parmi les *reducers*.

D'autres travaux utilisent des connaissances préalables sur les nœuds et les données pour configurer la fonction de partition. SAMR [Chen *et al.* 2010] est un algorithme d'ordonnancement qui utilise l'historique des exécutions précédentes pour identifier les performances des nœuds. Cet ordonnanceur centralisé assigne donc les tâches en fonction des performances précédentes des nœuds. Cela permet de contourner le biais de partitionnement en allouant peu de tâches aux *reducers* qui ont été précédemment peu efficaces. [Slagter *et al.* 2013] proposent plutôt une méthode d'échantillonnage qui consiste à affiner les paramètres du système et de l'allocation en traitant des sous-ensembles des données. Cette méthode permet d'améliorer l'équilibre des charges et la consommation de mémoire pour un jeux de données précis.

Une autre approche est de ne pas chercher à corriger la fonction de partition mais d'ajouter du dynamisme dans la phase de reduce. [Kwon *et al.* 2013] proposent la solution SkewTune pour contrer le biais de partitionnement. Quand un nœud est libre après avoir exécuté l'ensemble de ses tâches, l'algorithme centralisé SkewTune identifie le *reducer* le plus lent et réalloue certaines de ses tâches au nœud libre.

Il est plus difficile de réduire l'impact du biais des clés coûteuses puisqu'une clé doit être traitée par un seul *reducer*. [Gufler *et al.* 2011] évoquent également ce biais mais proposent simplement d'alerter l'utilisateur. FP-Hadoop [Liroz-Gistau *et al.* 2016] traite les deux biais de la phase de *reduce* avec un processus de division de tâche centralisé. Ils introduisent une nouvelle phase de *reduce* intermédiaire qui parallélise le traitement d'une même clé. Cette nouvelle phase consiste à faire appliquer aux *reducers* une fonction similaire à la fonction de combinaison. Les résultats intermédiaires sont ensuite envoyés à un unique *reducer* qui produira le résultat final de la clé. Les contraintes liées à cette phase de *reduce* intermédiaire sont les même que celles liées à la phase de combinaison. De ce fait, il existe des fonctions de *reduce* qui peuvent difficilement bénéficier de FP-Hadoop. De plus, cette nouvelle phase est gérée par le superviseur qui dispose d'une structure de données centralisée dont la paramétrisation nécessite des connaissances préalables sur les données.

3.4 Implémentations

La distribution du patron de conception MapReduce se trouve aujourd'hui dans de nombreux *frameworks*. Hadoop [White 2015] en est l'implémentation de référence. L'efficacité d'une exécution Hadoop dépend de plusieurs paramètres comme le nombre de *mappers*, la taille des *chunks* envoyés aux *reducers*, etc. Comme pour la fonction de partition, Hadoop propose des valeurs par défaut pour l'ensemble de ses paramètres mais leur ajustement à un job spécifique permet d'améliorer la performance du *framework*. À ce titre, [Verma *et al.* 2011, Lama & Zhou 2012] prédisent les paramètres à utiliser pour l'exécution d'un job en déduisant son profil depuis les historiques d'exécutions précédentes. Comme l'illustre le biais de partitionnement, la pertinence de la fonction de partition a aussi un impact sur l'efficacité de l'exécution du job. Hadoop propose aux utilisateurs de personnaliser cette fonction de partition afin qu'elle distribue les *chunks* aux *reducers* plus efficacement que la fonction par défaut ne pourrait le faire. Encore une fois,

cette modification est spécifique aux données et au job pour laquelle elle est réalisée. Malgré ces possibilités d'adapter une exécution aux données, soulignons qu'elles se limitent à préparer l'exécution d'un job. Toutefois, lorsqu'une exécution est lancée, il se peut qu'un biais persiste ou qu'un nœud subisse une fluctuation de ses performances. Pour de telles situations, Hadoop propose de relancer un nœud défectueux. Cependant, aucun mécanisme dynamique capable de corriger l'équilibre des charges et d'améliorer le temps d'exécution n'est proposé.

Hadoop Distributed File System (HDFS) est le système de fichiers distribué d'Hadoop. Il permet d'abstraire les disques des nœuds de calculs utilisés par Hadoop pour créer une structure de fichiers unifiée. HDFS autorise la duplication des données utilisées dans le flux MapReduce (données d'entrées et *chunks*). Cela permet d'offrir plus de possibilités pour instancier les *mappers* et les *reducers* sur les nœuds où se trouvent les données. En effet, éviter d'effectuer l'opération coûteuse de transférer des données entre différents nœuds du système permet d'améliorer les performances du processus global. Cette réplication est également utile pour la tolérance aux pannes du système. Si un nœud subit une chute de performance ou un arrêt complet, Hadoop peut relancer (complètement) les tâches exécutées sur un autre nœud qui possède également les données nécessaires localement.

Apache Spark [Zaharia *et al.* 2016] est un moteur d'analyse unifié pour le traitement de données massives. Ce *framework* est réputé pour sa rapidité et peut s'exécuter comme une surcouche à Hadoop. Dans [Zaharia *et al.* 2012], les auteurs présentent les structures de données utilisées par Spark. Ces structures sont appellées des *Resilient Distributed Datasets* (RDD). Spark implémente le patron de conception MapReduce de la manière suivante : la phase de *map* consiste à traiter chaque élément de l'ensemble de données d'entrée pour créer un nouveau RDD ; la phase de *reduce* agrège ces RDD afin de produire les résultats finaux. Les RDD sont des structures de données qui ont la particularités d'être tolérantes aux pannes car elles sont distribuées. Ils améliorent également l'efficacité du processus en permettant à l'utilisateur d'explicitement demander à ce que des résultats intermédiaires persistent en mémoire. Spark fournit également un ensemble riche d'opérateurs afin de manipuler ces résultats intermédiaires. Même si Spark est plus souple qu'Hadoop et permet lui aussi de personnaliser la fonction de partition, il peut tout de même souffrir des biais décrits dans la section précédente.

3.5 Synthèse

Ce chapitre présente le patron de programmation MapReduce. La distribution de celui-ci est amplement utilisée pour traiter de larges volumes de données. Cependant, des biais découlent de cette distribution et peuvent avoir un impact sur les performances du processus. Ces biais sont bien connus de la littérature et une majorité des travaux se concentrent sur les biais de la phase de *reduce*. Ces derniers constituent également le cadre applicatif de cette thèse.

Je propose de modéliser la phase de *reduce* comme un problème d'allocation de tâches dans un système multi-agents (cf. chapitre 4). Dans un tel système, l'objectif des agents *reducers* est d'exécuter l'ensemble des tâches de *reduce* le plus rapidement possible, c'est-à-dire de minimiser le *makespan* (cf. définition 1.9). Après la phase de *map*, les tâches de *reduce* sont allouées aux agents *reducers*. Ces derniers commencent alors à les exécuter. Se faisant, ils négocient avec leurs pairs pour procéder à des délégations de tâches (cf. chapitre 5). Ces délégations de

tâches permettent de réallouer les tâches de sorte à améliorer le *makespan* et à traiter le biais de partitionnement. Afin de traiter le biais des clés coûteuses, je considère le même procédé de découpe de tâches que FP-Hadoop. Cependant, ce mécanisme de découpe est intégré au processus décentralisé et aucun paramètre supplémentaire n'est nécessaire pour le mettre en place (cf. chapitre 6).

L'optimisation de la phase de *map* n'est pas étudiée dans cette thèse. Cependant, les travaux proposés dans [Vernica *et al.* 2012, Wolf *et al.* 2012] sont complémentaires à l'approche de cette thèse et pourraient être mis en place par un système multi-agents. Prendre en compte les biais de la phase de *map* représente néanmoins une perspective.

À ma connaissance, l'unique système multi-agents qui implémente le patron de conception MapReduce s'appuie sur des agents mobiles pour s'assurer la réplication du code et des données afin de garantir la tolérance aux pannes [Essa *et al.* 2014]. Ces objectifs ne sont pas les nôtres. De plus, ce travail ne met en œuvre aucune technique d'auto-organisation [Serugendo *et al.* 2003] pour que le système multi-agents s'adapte aux données ou à l'environnement informatique.

Le tableau 3.2 compare la proposition de cette thèse à celles citées dans ce chapitre. Cette proposition est dynamique et décentralisée. Elle ne demande ni des connaissances préalables sur les données, ni des données historiques et ne requiert aucun paramètre qui dépende des données. Ce processus dynamique et décentralisé est adaptatif car il permet d'ajuster l'allocation de tâches à des *reducers* hétérogènes sans supervision. De plus, l'aspect itératif des négociations autorise les agents à adapter l'allocation à une hétérogénéité non prévue, et ce sans mécanisme supplémentaire. Plusieurs travaux proposent une approche qui nécessite une connaissance préalable des données et une paramétrisation spécifique à un job. Elles demandent donc des connaissances spécifiques sur les données, l'implémentation du job, voire celle de la plateforme qui applique le patron de conception MapReduce. L'approche proposée dans cette thèse consiste plutôt à fixer ces paramètres en ne considérant aucune connaissance préalable sur les données ou les jobs. En effet, mon ambition est de concevoir une solution capable de corriger les biais de la phase de *reduce* sans considérer les données et les fonctions définies par l'utilisateur. Malgré l'absence de paramètre spécifique aux données, la réallocation dynamique des tâches de *reduce* améliore l'équilibre des charges de travail en continu. Cela permet d'améliorer le *makespan* tout en exécutant les tâches, quelques soient les données d'entrée.

Dans le cadre de cette thèse, j'ai également construit le prototype distribué d'un système multi-agents qui implémente le patron de conception MapReduce (cf. chapitre 8). Ce prototype a permis de vérifier la faisabilité de ma proposition ainsi que d'étudier l'impact du mécanisme de réallocation dynamique des tâches en le comparant à une exécution classique de MapReduce. Le système multi-agents n'utilise pas HDFS, la gestion des *chunks* est décrite dans le chapitre 8.

	Biais		Caractéristiques des propositions pour la résolution des biais			
	Partit.	Clés coût.	Décentralisé	Dynamique	Pas de conn. préalables	Faible param.
Hadoop [White 2015]	-	-	NP	NP	NP	NP
[Gufler *et al.* 2011]	✓	-	-	-	-	-
SAMR [Chen *et al.* 2010]	✓	-	-	-	-	✓
SkewTune [Kwon *et al.* 2013]	✓	-	-	✓	✓	✓
FP-Hadoop [Liroz-Gistau *et al.* 2016]	✓	✓	-	✓	✓	-
MAS4Data	✓	✓	✓	✓	✓	✓

TABLEAU 3.2 : Caractéristiques des propositions citées dans la section 3.3.2 pour contrer les biais de la phase de *reduce*. MAS4Data désigne la proposition de cette thèse. Les caractéristiques de ces propositions sont avérées (✓), non avérées (-) ou non pertinentes (NP).

Deuxième partie

Modélisation multi-agents

Chapitre 4

Réallocation multi-agents de tâches situées

Sommaire

4.1 Introduction

Ce chapitre présente le modèle formel qui permet d'exprimer un processus de réallocation de tâches concurrent à leur exécution. J'ai décrit dans le chapitre 1 les notions nécessaires pour définir un problème d'allocation de n tâches indépendantes, non divisibles et non préemptives parmi m agents collaboratifs et dont l'objectif est de minimiser le *makespan*, c'est-à-dire le temps d'exécution de l'ensemble des tâches. J'y suppose que toutes les ressources nécessaires à l'exécution des tâches sont disponibles dans le système. Un agent peut donc, selon le nœud sur lequel il se trouve, avoir un accès direct à l'ensemble, une partie, ou aucune des ressources nécessaires à l'exécution d'une tâche donnée. S'il ne possède pas l'ensemble de ces ressources, il doit récupérer les ressources manquantes auprès de ses pairs. L'opération qui consiste à récupérer des ressources avant d'exécuter une tâche est coûteuse, ce qui constitue la première raison pour laquelle les tâches n'ont pas le même coût pour tous les agents du système. La seconde raison est simplement que les nœuds de calculs sur lesquels sont déployés les agents ne sont pas forcément homogènes en termes de capacité de calcul. Même avec un accès complet aux ressources nécessaires à l'exécution d'une tâche, il se peut qu'un agent soit plus lent qu'un autre pour en produire le résultat.

Dans le chapitre 3, j'ai décrit le cadre applicatif de cette thèse. La distribution du patron de conception MapReduce est très utilisée pour le traitement de données massives. Seulement, celle-ci donne lieu à des biais qui résultent en une répartition non équitable de la charge de travail parmi les nœuds de calculs. Je cherche à résoudre les biais de la phase de *reduce* et ainsi à améliorer son temps d'exécution. De ce fait, je souhaite résoudre le problème d'allocation décrit ci-dessus : allouer les tâches de *reduce* à des agents *reducers* de sorte à obtenir le meilleur *makespan* possible. Améliorer le *makespan* aide à mieux répartir la charge de travail parmi les agents et donc à produire les résultats de la phase de *reduce* plus rapidement.

Dans le cadre d'une application MapReduce, plusieurs particularités existent :

- les données d'entrée ne sont généralement pas prétraitées. Il est donc compliqué d'évaluer le coût exact d'une tâche pour un agent ;
- les ressources nécessaires à l'exécution des tâches de *reduce* sont des *chunks*, des blocs de données préalablement générés par les *mappers*. D'une manière générale, ces *chunks* sont écrits localement sur les disques des *mappers*. Toutefois, si le système distribué dispose d'un système de fichiers distribué, les *chunks* peuvent être répliqués sur plusieurs machines du système. Quoiqu'il en soit, les *chunks* d'une même tâche se trouvent probablement sur différents nœuds du système, et il se peut aussi que certains d'entre eux soient localisés sur des machines où aucun *reducer* ne se trouve ;
- différents *mappers* peuvent créer des *chunks* pour une même clé. Pour s'assurer de la cohérence des résultats de la phase de *reduce*, il est indispensable qu'un seul *reducer* reçoive tous ces *chunks*. La répartition des clés aux *reducers* doit donc se faire à l'aide d'une fonction de partition statique qui doit être connue avant le début de la phase de *map*. Cela peut mener à un mauvais équilibre des charges. Adapter la fonction de partition aux données nécessite de les connaître au préalable et n'est, de toute façon, une stratégie valable que pour ces données. L'alternative que j'ai choisie est de réallouer les tâches pendant la phase de *reduce*.

Formulées ici dans le contexte du cadre applicatif de cette thèse, ces contraintes apparaissent dès lors que l'on souhaite effectuer une allocation de tâches où (a) toutes les ressources nécessaires à l'exécution des tâches ne sont pas directement accessibles pour tous les agents et (b) où il est compliqué d'évaluer le coût d'une tâche pour un agent.

Par souci de présenter des définitions générales et parce que ces points ne sont généralement pas des hypothèses dans l'état de l'art, le chapitre 1 n'en fait pas mention. La section 4.2 présente le formalisme du problème de réallocation de tâches situées dans un système multi-agent, ou MASTA pour *Multi-agent Situated Task Allocation*. Ce problème considère les contraintes énoncées ci-dessus et correspond donc au cadre applicatif. Ensuite, je définis dans la section 4.3 les délégations de tâches socialement rationnelles qui constituent les opérations utilisées par les agents pour améliorer dynamiquement le *makespan*, pendant qu'ils exécutent les tâches. Dans la section 4.4, j'illustre comment les opérations de délégations et de consommations de tâches sont concurrentes et modifient l'instance MASTA courante. Dans la section 4.5, je discute de l'impact de la fonction de coût sur l'efficacité du processus de co-occurence des délégations et des consommations de tâches. Enfin, je propose une synthèse et explore les perspectives de cette thèse vis-à-vis de ce modèle dans la section 4.6.

4.2 Réallocations de tâches situées

Un problème MASTA est un problème de **réallocation** de tâches résolu par un système d'agents coopératifs qui cherchent à réduire le *makespan* de l'allocation courante.

Définition 4.1 (Instance MASTA).
Une instance MASTA de taille (k, m, n) *telle que* $k \geq 1$, $m \geq 2$, $n \geq 1$ *est un quadruplet* $MASTA = (S, \mathcal{T}, P, c)$ *où :*

- $S = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{R}, l, d)$ *est un système distribué complet*[a] *tel que* $|\mathcal{N}| = k$ *et* $|\mathcal{A}| = m$ *;*
- $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ *est un ensemble de* n *tâches à exécuter ;*
- P *est une allocation valide*[b] *des tâches de* $\mathcal{T}$ *aux agents de* $\mathcal{A}$ *;*
- c *est la fonction qui estime le coût strictement positif d'une tâche si elle est exécutée sur un nœud donné :*

$$c : \mathcal{T} \times \mathcal{N} \mapsto \mathbb{R}_+^* \tag{4.1}$$

a. cf. définition 1.1
b. cf. définition 1.4

Par souci de lisibilité, on note $d_i(\tau)$ le nombre de ressources nécessaires à l'exécution de la tâche $\tau \in \mathcal{T}$ qui ont été distribuées sur le nœud de l'agent $i \in \mathcal{A}$, c'est-à-dire $d_i(\tau) = |\mathcal{R}_\tau \cap \mathcal{R}_i|$. On note également $c_i(\tau)$ l'estimation du coût de la tâche τ si elle est exécutée sur le nœud de l'agent i, c'est-à-dire $c_i(\tau) = c(\tau, l_i)$.

Exemple 4.1 (Instance MASTA).
Soit $MASTA = (S, \mathcal{T}, P, c)$ *une instance MASTA de taille* $(2, 2, 7)$. *On a donc* $\mathcal{T} = \{\tau_1, \ldots, \tau_7\}$, *sept tâches à allouer dans un système distribué* $S = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{R}, l, d)$ *constitué de deux nœuds de calculs aux performances homogènes (*$\mathcal{N} = \{\nu_1, \nu_2\}$*), deux agents (*$\mathcal{A} = \{1, 2\}$*) capables de communiquer (*$\mathcal{E} = \{(1, 2)\}$*) et chacun situé sur un nœud de calculs (*$l_1 = \nu_1$, $l_2 = \nu_2$*). On considère l'allocation de tâches courante* P *et la fonction de distribution des ressources* d *quelconques. La fonction de coût* c *est définie par extension dans le tableau 4.1.*

Pour cette instance, on identifie P^{mks} *l'allocation qui minimise le makespan. Dans cette allocation :*

- *le lot de tâches de l'agent* 1 *contient les tâches* τ_1, τ_3, τ_5 *et* τ_6 *(*$P_1^{mks} = \{\tau_1, \tau_3, \tau_5, \tau_6\}$*), ce qui lui attribue une charge de travail de 33 (*$w_1(P^{mks}) = 33$*) ;*
- *le lot de tâches de l'agent* 2 *contient les tâches* τ_2, τ_4 *et* τ_7 *(*$P_2^{mks} = \{\tau_2, \tau_4, \tau_7\}$*), ce qui lui attribue une charge de travail de 35 (*$w_2(P^{mks}) = 35$*).*

Le makespan optimal est donc de 35 ($C_{max}(P^{mks}) = 35$*).*

On dit que des agents améliorent une instance MASTA lorsqu'ils réalisent une action qui modifie une instance $MASTA = (S, \mathcal{T}, P, c)$ en une autre instance $MASTA' = (S, \mathcal{T}, P', c)$ tel que l'allocation de la nouvelle instance ait un meilleur *makespan* que l'instance précédente ($C_{max}(P') < C_{max}(P)$). Résoudre un problème MASTA revient alors à améliorer une instance donnée jusqu'à ce que son allocation courante soit de *makespan* optimal.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$c_1(\tau_k)$	1	8	15	30	9	8	14
$c_2(\tau_k)$	2	4	12	24	6	13	7

TABLEAU 4.1 : Fonction de coût de l'instance MASTA de l'exemple 4.1.

Par la suite, considérons qu'un système distribué $S = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{R}, l, d)$ et qu'une instance $MASTA = (S, \mathcal{T}, P, c)$ sont définis.

4.3 Délégations socialement rationnelles

Dans le cadre de la résolution d'un problème MASTA, on appelle **accord** un échange local de tâches qui implique plusieurs agents.

Définition 4.2 (Accord).
Soit P l'allocation courante et b_{i_j} un lot de tâches issu de l'ensemble des tâches allouées à l'agent $i_j \in \mathcal{A}$, c'est-à-dire $b_{i_j} \subseteq P_{i_j}$. Un accord δ est une liste ordonnée de k ($2 \leq k \leq m$) lots de tâches $[b_{i_1}, \ldots, b_{i_k}]$ telle que $\bigcup_{j \in [1,k]} b_{i_j} \neq \emptyset$. Les agents $Con_\delta = \bigcup_{j \in [1,k]} i_j$ sont les contractants de l'accord δ.

L'allocation $P' = \delta(P)$ est l'allocation qui résulte de l'accord δ telle que :

$$\forall j \in [2,k], P'_{i_j} = (P_{i_j} \cup b_{i_{j-1}}) \setminus b_{i_j} \quad (4.2)$$

$$P'_{i_1} = (P_{i_1} \cup b_{i_k}) \setminus b_{i_1} \quad (4.3)$$

$$\forall i \in \mathcal{A} \setminus Con_\delta, P'_i = P_i \quad (4.4)$$

Un accord consiste donc à ce que chaque participant i_j donne un sous-ensemble de ses tâches b_{i_j} au participant i_{j+1} (cf. équation 4.2) à l'exception du dernier participant i_k qui donne son lot b_{i_k} au premier participant i_1 (cf. équation 4.3). Les agents qui ne sont pas impliqués dans un accord ne sont pas impactés (cf. équation 4.4). Définis ainsi, les accords permettent d'exprimer tous types d'échanges de tâches entre agents du système. Par exemple, l'accord suivant consiste en un échange mutuel de tâches uniques (ou *swap*) :

Exemple 4.2 (Échange de tâches entre deux agents).
Soient $i, j \in \mathcal{A}$ et P l'allocation courante. On peut définir δ_e l'échange de la tâche $\tau \in P_i$ avec la tâche $\tau' \in P_j$ en fixant $Con_{\delta_e} = \{i, j\}$, $b_i = \{\tau\}$ et $b_j = \{\tau'\}$. En notant $P' = \delta(P)$, on a $P'_i = (P_i \setminus \{\tau\}) \cup \{\tau'\}$ et $P'_j = (P_j \setminus \{\tau'\}) \cup \{\tau\}$.

Dans cette thèse, je me restreins à un type d'accord que l'on appelle les **délégations de tâches**.

Définition 4.3 (Délégation de tâche).
Soient $i, j \in \mathcal{A}$ et P l'allocation courante. On appelle délégation de la tâche τ l'accord δ

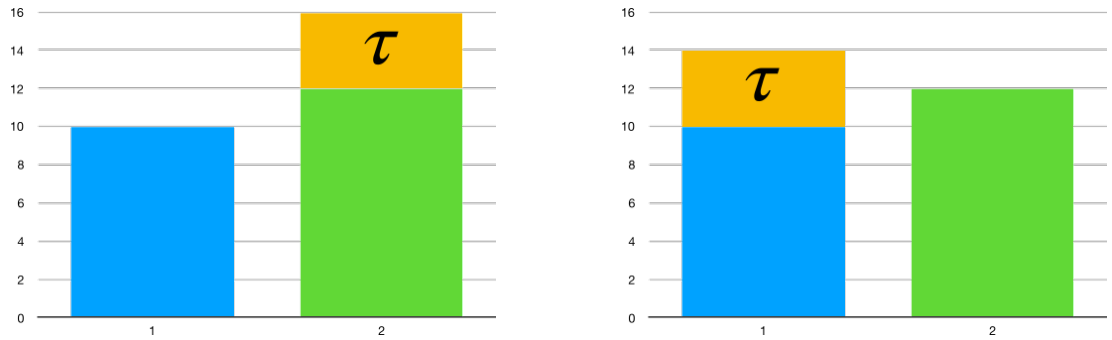
tel que $b_i = \{\tau\}$ *et* $b_j = \emptyset$. *Dans l'allocation* $P' = \delta(P)$, *on a donc* $P'_i = P_i \setminus \{\tau\}$, *et* $P'_j = P_j \cup \{\tau\}$.

Après une délégation δ de la tâche τ de l'agent i à l'agent j, on remarque que $w_i(\delta(P)) = w_i(P) - c_i(\tau)$ et que $w_j(\delta(P)) = w_j(P) + c_j(\tau)$. Afin qu'un tel accord soit bénéfique pour l'ensemble du système, l'agent j accepte une délégation de tâche de la part de i si elle fait décroître le *makespan* local entre les deux agents. On dit alors que cette délégation de tâche est **socialement rationnelle** pour les deux contractants.

Définition 4.4 (Délégation de tâche socialement rationnelle).
Soient P *l'allocation courante et* δ *la délégation de la tâche* τ *de l'agent* i *à l'agent* j. *On dit que* δ *est socialement rationnelle si et seulement si :*

$$w_j(\delta(P)) < w_i(P) \tag{4.5}$$

Une délégation est utile à l'ensemble du système dès lors qu'elle améliore le *makespan* des deux contractants et, par conséquent, l'équilibre de leurs charges de travail. La figure 4.1 illustre une délégation de tâche socialement rationnelle. On y observe que le *makespan* local des deux contractants est plus faible après la délégation et que leurs charges de travail sont mieux équilibrées. On remarque également qu'il se peut que, à l'issue d'une délégation, l'agent qui reçoit la tâche soit plus chargé que celui qui la lui donne. En accumulant ces opérations locales, les agents diminuent progressivement le *makespan* de l'allocation de tâches et équilibrent la charge de travail globale du système ce qui permet de traiter l'ensemble des tâches plus rapidement.



(a) Répartition de la charge de travail avant la délégation socialement rationnelle de la tâche τ.

(b) Répartition de la charge de travail après la délégation socialement rationnelle de la tâche τ.

FIGURE 4.1 : Délégation de tâche socialement rationnelle. En notant P l'allocation de tâches qui mène à la situation présentée en figure 4.1a, on observe que $w_1(P) = 10$, et que $w_2(P) = 16$. En sachant que $c_i(\tau) = c_j(\tau) = 4$, la délégation δ de la tâche τ est socialement rationnelle car $w_1(\delta(P)) < w_2(P)$ $(14 < 16)$.

Propriété 4.1.
Une délégation de tâche socialement rationnelle δ *n'accroît pas le makespan de l'allocation courante* P, *c'est-à-dire*

$$C_{max}(\delta(P)) \leq C_{max}(P) \tag{4.6}$$

Preuve 4.1. *Soient* P *l'allocation courante et* δ *une délégation de tâche de l'agent* i *à l'agent* j. *Comme* δ *est socialement rationnelle, nous avons :*

- $w_i(P) > w_j(P)$ *et* $w_j(\delta(P)) < w_i(P)$ *par définition ;*
- $w_i(\delta(P)) < w_i(P)$ *car* $\forall \tau \in \mathcal{T}, c_i(\tau) > 0$.

Deux cas sont alors possibles.

1. *Soit l'agent i est l'agent le plus chargé du système et aucun agent n'est autant chargé que lui. On a donc $C_{max}(P) = w_i(P)$. Quelque soit l'agent le plus chargé dans l'allocation $\delta(P)$, nous avons :*
$$C_{max}(\delta(P)) < C_{max}(P)$$
2. *Soit il existe au moins un agent qui est autant ou plus chargé que i. Dans ce cas,*
$$C_{max}(\delta(P)) = C_{max}(P)$$ □

Parmi les tâches allouées à l'agent i, nous sommes en mesure d'identifier les tâches qui peuvent mener à une délégation socialement rationnelle.

Définition 4.5 (Ensemble des délégations socialement rationnelles).
L'ensemble des délégations de tâche socialement rationnelles de l'agent i est :

$$\Gamma_i(P) = \{\tau \in P_i \mid \exists j \in \mathcal{A} \text{ tel que } w_j(P) + c_j(\tau) < w_i(P)\} \tag{4.7}$$

On dit qu'une allocation de tâches est **stable** si aucun des agents du système n'a de délégation socialement rationnelle à considérer.

Définition 4.6 (Allocation stable).
Soit P l'allocation de tâches courante. P est stable si et seulement si $\forall i \in \mathcal{A}, \Gamma_i(P) = \emptyset$.

Une fois que les agents atteignent une allocation stable, ils ne peuvent plus améliorer l'instance MASTA courante en utilisant des délégations socialement rationnelles.

Propriété 4.2.
Pour toute allocation non stable, il existe un chemin fini de délégations socialement rationnelles qui mène à une allocation stable.

Preuve 4.2. *Soient P l'allocation courante et $\overrightarrow{W_P} = \langle w_{i_1}(P), \ldots, w_{i_m}(P)\rangle$ le vecteur des charges de travail des m agents dans l'ordre décroissant. On note $\overrightarrow{W_P}(r) = w_{i_r}(P)$ la r-ième charge de travail la plus élevée.*

Par définition, nous savons que P est instable si et seulement si il existe une délégation de tâche socialement rationnelle pour améliorer le makespan local de deux agents. Considérons la transition de l'allocation P à l'allocation P' grâce à la délégation de tâche socialement rationnelle δ. Comme δ est socialement rationnelle, nous savons qu'un agent a délégué une tâche à un autre agent tel que leur makespan local a diminué. Formellement,

$$\exists i, j \in \mathcal{A} \text{ tels que } max(w_i(P'), w_j(P')) < max(w_i(P), w_j(P)) \wedge \forall k \in \mathcal{A} \setminus \{i, j\}, w_k(P) = w_k(P')$$

Une telle transition implique que $\overrightarrow{W_{P'}} < \overrightarrow{W_P}$ selon l'ordre lexicographique, c'est-à-dire

$$\exists r \in [1, m] \text{ tel que } \forall r' < r, \overrightarrow{W_{P'}}(r') = \overrightarrow{W_P}(r') \wedge \overrightarrow{W_{P'}}(r) < \overrightarrow{W_P}(r)$$

Ainsi, chaque transition d'une allocation non stable vers une autre allocation grâce à une délégation de tâche socialement rationnelle fait décroître le vecteur $\overrightarrow{W}$ dans l'ordre lexicographique jusqu'à atteindre une allocation stable. De plus, comme il existe un nombre fini d'allocations et que $\overrightarrow{W}$ décroît strictement à chaque étape, il ne peut y avoir qu'un nombre fini de telles transitions. □

Exemple 4.3 (Allocation stable).
Considérons l'instance MASTA issue de l'exemple 4.1 avec l'allocation courante P suivante :

- $P_1 = \{\tau_2, \tau_4, \tau_6\}$,
- $P_2 = \{\tau_1, \tau_3, \tau_5, \tau_7\}$.

Dans cette situation, les charges de travail des agents sont $w_1(P) = 46$ et $w_2(P) = 27$. On a donc $C_{max}(P) = 46$ (cf. figure 4.2a). On observe que les délégations des tâches τ_2 et τ_6 sont socialement rationnelles pour l'agent 1 ($\Gamma_1(P) = \{\tau_2, \tau_6\}$). Il est donc possible d'améliorer P afin de faire décroitre son makespan. En revanche, l'agent 2 est moins chargé que l'agent 1 et ne peut donc pas déléguer de tâche ($\Gamma_2(P) = \emptyset$).

Si l'agent 1 initie la délégation δ_1 de la tâche τ_6, l'allocation courante devient $P' = \delta_1(P)$ avec :

- $P'_1 = \{\tau_2, \tau_4\}$,
- $P'_2 = \{\tau_1, \tau_3, \tau_5, \tau_6, \tau_7\}$.

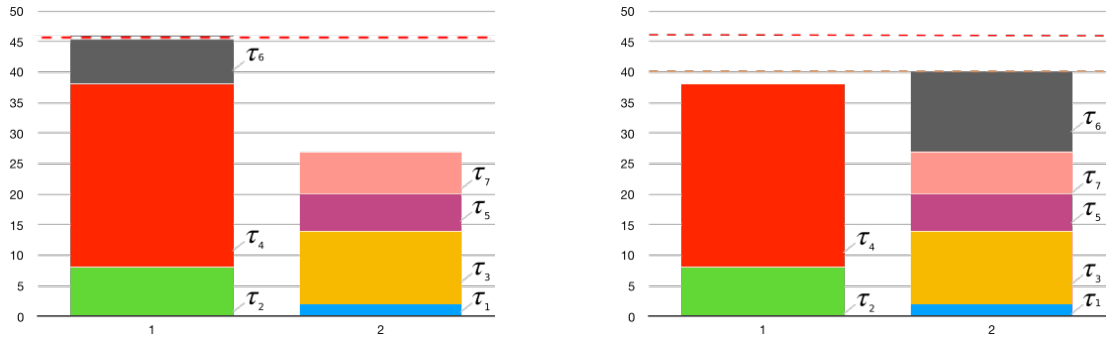
L'allocation de tâches P' (cf. figure 4.2b) offre une répartition des charges plus équilibrée ($w_1(P') = 38$ et $w_2(P') = 40$) et un meilleur makespan ($C_{max}(P') = 40$) mais n'est pas stable. En effet, on observe que $\Gamma_1(P') = \emptyset$ mais $\Gamma_2(P') = \{\tau_1\}$.

La délégation δ_2 de la tâche τ_1 de l'agent 2 à l'agent 1 crée une nouvelle allocation $P'' = \delta_2(P')$ où $w_1(P'') = 39$ et $w_2(P'') = 38$ (cf. figure 4.2c). Comme $\Gamma_1(P'') = \Gamma_2(P'') = \emptyset$, l'allocation P'' est stable. L'allocation P'' n'est pas optimale ($C_{max}(P'') > C_{max}(P^{mks})$) mais elle ne peut plus être améliorée par une délégation de tâche socialement rationnelle.

Les délégations de tâche δ_1 et δ_2 permettent d'améliorer de manière successive le makespan de l'allocation de tâche courante ($C_{max}(P) > C_{max}(P') > C_{max}(P'')$) jusqu'à ce qu'une allocation stable soit atteinte.

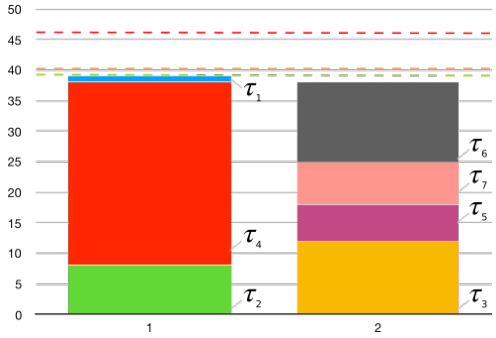
4.4 Réallocation des tâches pendant leur exécution

Les agents peuvent simultanément exécuter les tâches et entreprendre des délégations de tâches. En d'autres mots, pendant que les agents délèguent les tâches pour améliorer l'allocation courante, ils réalisent (ou consomment) également les tâches. Les opérations de délégation et de consommation modifient toutes deux l'instance MASTA courante. D'une part, comme nous l'avons vu dans la section précédente, une délégation modifie l'allocation courante. D'autre part, une fois qu'une tâche est consommée, elle est supprimée de l'ensemble des tâches $\mathcal{T}$ et de l'allocation courante. Cette section définit la consommation d'une tâche et présente ensuite le processus utilisé par les agents pour gérer simultanément les délégations et les consommations de tâches.



(a) Charges de travail des agents 1 et 2 pour l'allocation P.

(b) Charges de travail des agents 1 et 2 pour l'allocation P'.



(c) Charges de travail des agents 1 et 2 pour l'allocation P''.

FIGURE 4.2 : Évolutions des charges de travail des agents 1 et 2 après les délégations δ_1 et δ_2 de l'exemple 4.3.

4.4.1 Consommations de tâches

Les consommations de tâches sont des perturbations qui modifient l'instance MASTA courante en supprimant une tâche de l'allocation courante et de l'ensemble des tâches à consommer.

Définition 4.7 (Consommation de tâche).
Soit $MASTA = (S, \mathcal{T}, P, c)$ l'instance MASTA courante. La consommation de la tâche τ par l'agent i, notée γ, crée une nouvelle allocation courante $P' = \gamma(P)$ et donc une nouvelle instance $MASTA' = (S, \mathcal{T}', P', c)$ tel que :

$$\mathcal{T}' = \mathcal{T} \setminus \{\tau\} \tag{4.8}$$
$$P'_i = P_i \setminus \{\tau\} \tag{4.9}$$
$$\forall j \in \mathcal{A} \setminus \{i\}, P'_j = P_j \tag{4.10}$$

Lorsqu'un agent consomme une tâche, elle ne fait plus partie de son lot de tâches (cf. équation 4.9), ni de l'ensemble des tâches de l'instance (cf. équation 4.8). Les lots de tâches des autres agents ne sont pas modifiés (cf. équation 4.10). Évidemment, une consommation de tâche produit une allocation avec un *makespan* plus faible.

Propriété 4.3.
Une consommation de tâche γ n'accroît pas le makespan d'une allocation de tâches P, c'est-à-dire

$$C_{max}(\gamma(P)) \leq C_{max}(P) \tag{4.11}$$

Preuve 4.3. *Soit P une allocation de tâches et γ une consommation de tâche appliquée à P. Si l'agent qui consomme est l'agent le plus chargé du système, on a $C_{max}(\gamma(P)) \leq C_{max}(P)$, sinon $C_{max}(\gamma(P)) = C_{max}(P)$.* □

La succession de consommations de tâches supprime progressivement les tâches de l'allocation initiale P^{init} jusqu'à atteindre l'allocation finale vide P^{finale}, c'est-à-dire l'allocation dans laquelle il n'existe plus de tâche à consommer ($\forall i \in \mathcal{A}, P_i^{finale} = \emptyset$). On rappelle que l'on considère que le coût d'une tâche pour un agent est l'estimation du temps nécessaire à cet agent pour consommer la tâche. En débutant avec P^{init} et sans appliquer de délégation de tâche, le temps théorique nécessaire pour consommer l'ensemble des tâches est égal au *makespan* de P^{init} ($C_{max}(P^{init})$). Cependant, si P^{init} n'est pas stable, les délégations de tâches socialement rationnelles aident à équilibrer la charge de travail des agents et améliorent ainsi le temps d'exécution de l'ensemble des tâches.

4.4.2 Co-occurence des consommations et des délégations

Pour résoudre un problème MASTA, les agents utilisent les délégations socialement rationnelles. De plus, les consommations de tâches ont lieu simultanément et suppriment progressivement des tâches de l'ensemble $\mathcal{T}$. Alors que les délégations sont des opérations provoquées par les agents, les consommations sont subies. En effet, quelque soit l'instance MASTA et jusqu'à parvenir à l'allocation P^{finale}, les agents consomment les tâches en continu. De cette façon les agents passent d'instance en instance en améliorant systématiquement le *makespan* de l'allocation courante. Les agents peuvent atteindre une allocation stable, c'est-à-dire une allocation où aucune délégation de tâche socialement rationnelle n'est possible. Cependant lorsqu'une consommation de tâche se produit, l'allocation courante se trouve perturbée et il est possible qu'elle ne soit plus stable. Dans ce cas, celle-ci peut de nouveau être améliorée. Ces délégations et consommations s'enchaînent jusqu'à ce que toutes les tâches aient été consommées. Les délégations socialement rationnelles et les consommations améliorent le *makespan* de l'allocation courante de manière concurrente. Ces deux opérations sont complémentaires car une consommation peut offrir de nouvelles opportunités d'effectuer des délégations socialement rationnelles.

La figure 4.3 illustre comment les agents passent d'une instance MASTA à l'autre à chaque délégation/consommation de tâche. Afin d'exprimer les délégations de tâches socialement rationnelles et les consommations de tâches, on note $P^{n,k}$ l'allocation de n tâches ($|\mathcal{T}| = n$), ayant bénéficié de k délégations de tâche socialement rationnelles. En partant de l'allocation initiale $P^{n,0}$, les agents utilisent k délégations de tâches socialement rationnelles pour l'améliorer (de $P^{n,0}$ à $P^{n,k}$) jusqu'à ce qu'une tâche soit consommée (de $P^{n,k}$ à $P^{n-1,0}$). Une consommation de tâche peut se produire quand les agents ont atteint une allocation stable (par exemple P^{n-1}_{stable}) ou pas (par exemple $P^{n,k}$).

La co-occurence de ces deux phénomènes offre deux caractéristiques intéressantes.

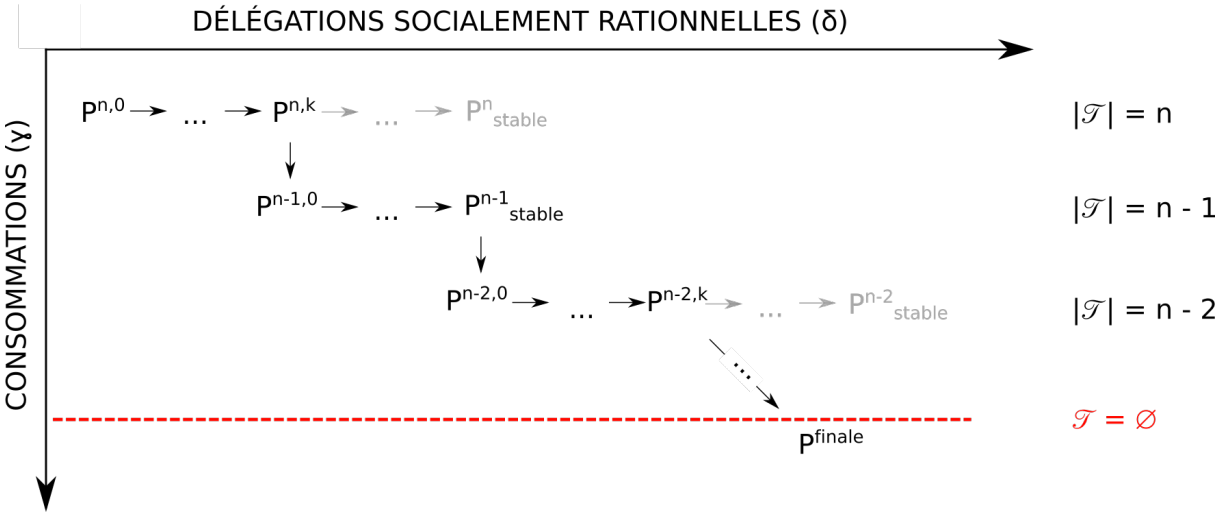


FIGURE 4.3 : Co-occurence des délégations et des consommations de tâches durant la résolution d'un problème MASTA. Les transitions horizontales correspondent à des délégations de tâches socialement rationnelles. Les transitions verticales correspondent à des consommations de tâches. Les états grisés montrent des opérations qui auraient pu avoir lieu sans consommation de tâche. La ligne en pointillés représente la dernière consommation de tâche. P^{finale} est la seule allocation finale possible dans laquelle les lots de tâches de tous les agents sont vides car toutes les tâches ont été exécutées.

1. Le système distribué est intrinsèquement adaptatif à une variation de performances des nœuds de calculs. En effet, si un nœud ralentit lors de l'exécution, sa charge de travail diminue moins rapidement que celles de ses pairs car il consomme les tâches qui lui sont allouées moins rapidement. De ce fait, il arrive un temps où une de ses tâches peut être déléguée à l'un de ses pairs pour rééquilibrer la charge de travail.
2. Le système est robuste à l'utilisation d'une fonction de coût mal calibrée. Dès lors qu'une fonction de coût préserve le caractère socialement rationnel d'une délégation, celle-ci peut ne pas indiquer précisément le coût d'une tâche pour un agent sans que l'allocation des tâches soit totalement faussée. En effet, les ajustements continus des agents permettent de corriger l'allocation tout au long de l'exécution et d'atténuer l'impact que pourrait avoir une mauvaise estimation du coût des tâches pour les agents.

4.5 Estimation du coût d'une tâche

La fonction de coût est capitale pour le mécanisme de délégation de tâche décrit dans ce chapitre. Elle permet de calculer la charge de travail d'un agent mais également de déterminer si une délégation de tâche est socialement rationnelle. Si l'on s'en tient au modèle théorique, la fonction de coût c d'une instance $MASTA = (S, \mathcal{T}, P, c)$ est exacte. En revanche, lorsque, comme dans cette thèse, le formalisme sert une application pratique, il est plus réaliste de considérer que c donne une estimation du coût des tâches de $\mathcal{T}$ pour chaque agent de $\mathcal{A}$. Je considère qu'il n'est pas nécessaire que c soit précise pour que la réallocation de tâche par délégations socialement rationnelles soit efficace. En effet, quelque soit la qualité du *makespan* de l'allocation courante, des tâches seront consommées ce qui perturbera cette allocation. Même si les agents parviennent à obtenir une allocation optimale à l'aide des délégations de tâches socialement rationnelles,

celle-ci sera perturbée dès lors qu'un agent aura consommé une tâche. Ces perturbations offrent aux agents la possibilité de réévaluer régulièrement la qualité de l'allocation de tâches et d'ajuster l'équilibre des charges de travail. Ces ajustements continus permettent de compenser une fonction de coût peu précise. La dynamicité du processus due à la co-occurence des délégations et consommations de tâches est plus importante que la précision de la fonction de coût.

Supposons qu'il existe une fonction de coût $\overset{\diamond}{c}$ qui donne le coût exact des tâches pour chaque agent du système. Si P est l'allocation de tâches courante, la charge de travail exacte de l'agent i, calculée avec $\overset{\diamond}{c}$, est notée $\overset{\diamond}{w}_i(P)$. Il peut exister une différence entre $\overset{\diamond}{w}_i(P)$, la charge de travail exacte de l'agent i, et $w_i(P)$, sa charge de travail estimée. Dans ce cas, i est susceptible d'engendrer des délégations de tâches qui ne sont pas véritablement socialement rationnelles. Deux situations sont alors possibles :

1. si la charge de travail de i est sur-estimée ($w_i(P) > \overset{\diamond}{w}_i(P)$), alors ce dernier risque de priver momentanément le système de délégations socialement rationnelles. Cependant, celui-ci consomme également ses tâches plus rapidement qu'estimé et, après avoir effectué des consommations, il sera de nouveau candidat pour accepter des délégations et rééquilibrer la charge de travail des agents ;
2. si la charge de travail de i est sous-estimée ($w_i(P) < \overset{\diamond}{w}_i(P)$), alors ce dernier risque de recevoir des tâches dont la délégation n'est pas véritablement socialement rationnelle. À terme, il refuse les délégations de tâches car il s'estime aussi (ou plus) chargé que ses pairs. Comme sa charge est sous-estimée, celui-ci consomme ses tâches plus lentement qu'estimé et pourra donc déléguer certaines de ses tâches une fois que ses pairs auront effectué des consommations.

Si le nœud de l'agent i ralentit, cela ne sera pas répercuté sur la fonction de coût mais la discordance entre la charge de travail réelle et estimée de i pourra être corrigée en déléguant certaines de ses tâches à ses pairs après que ceux-ci aient effectué des consommations.

Ainsi, la co-occurence des délégations et des consommations permet de régulièrement corriger l'allocation de tâches courante. Dans le contexte de cette thèse, la localité des ressources associées à une tâches influe sur son coût. Afin de préserver un ordre de grandeur raisonnable entre le coût exact d'une tâche et son coût estimé, il est donc nécessaire que la fonction de coût possède les propriétés suivantes.

Définition 4.8 (Fonction de coût valide).
Soit $MASTA = (S, \mathcal{T}, P, c)$ une instance MASTA. La fonction de coût c est valide si :

$$\forall \tau \in \mathcal{T}, \forall i \in \mathcal{A}, c_i(\tau) > 0 \tag{4.12}$$

$$\forall i, j \in \mathcal{A}, d_i(\tau) > d_j(\tau) \Rightarrow c_i(\tau) \leq c_j(\tau) \tag{4.13}$$

Une fonction de coût est valide si le coût d'une tâche est strictement positif (cf. équation 4.12) et si, en considérant les nœuds de performances homogènes, le coût d'une tâche est moindre sur un nœud que sur un autre si le premier possède plus de ressources nécessaires à la production du résultat de la tâche que le second (cf. équation 4.13). Par la suite, supposons que la fonction de coût associée à une instance MASTA est valide.

Par la suite, le formalisme MASTA sera utilisé pour représenter la phase de *reduce* du patron de conception MapReduce et les propriétés démontrées dans ce chapitre serviront à améliorer le

temps d'exécution de cette phase tout en préservant les propriétés essentielles à sa distribution sur plusieurs nœuds de calcul. Dans ce cadre, le coût des tâches est difficile à calculer de manière exacte car la localité des ressources et le volume des tâches ne sont pas connus avant leur allocation (cf. chapitre 3). Ce modèle a également vocation à s'adapter à des configurations matérielles et des jeux de données qui ne sont pas connus à l'avance, ce qui justifie également la difficulté d'avoir une fonction de coût exacte. Enfin, des incidents, comme la baisse de puissance de calcul d'un nœud, peuvent apparaître pendant l'exécution et faire varier les coûts réels des tâches de manière imprévisible.

4.6 Conclusion

4.6.1 Synthèse

Le formalisme MASTA permet de représenter un problème de réallocation de tâches dans lequel les ressources et les agents sont situés sur différents nœuds de calculs. Une instance MASTA est constituée d'un système distribué, d'un ensemble de tâches à exécuter, d'une allocation de ces tâches, et d'une fonction de coût qui donne une estimation du coût de l'exécution d'une tâche sur un nœud du système. Le coût d'une tâche varie d'un nœud à l'autre car les ressources nécessaires à son exécution ne sont pas toutes disponibles sur tous les nœuds du système et car l'utilisation des ressources non locales présente un surcoût.

Dans le but de traiter l'ensemble des tâches le plus rapidement possible, les agents améliorent le *makespan* de l'allocation courante grâce aux délégations de tâches socialement rationnelles et consomment les tâches de manière concurrente. Une délégation de tâche socialement rationnelle est le don d'une tâche d'un agent à un autre qui permet de rééquilibrer les charges de travail des deux agents et fait donc diminuer leur *makespan* local. Une consommation de tâches se produit lorsqu'un agent a terminé d'exécuter une tâche. Celle-ci est alors supprimée de l'allocation courante et de l'ensemble des tâches. En appliquant itérativement ces opérations, les agents diminuent le temps nécessaire à l'exécution de l'ensemble de tâches alors qu'ils sont en train d'exécuter ces même tâches.

Du point de vue du cadre applicatif, ce formalisme permet de modéliser la réallocation des tâches de *reduce* lors d'une exécution de MapReduce. Cette réallocation permet de corriger le biais de partitionnement et ainsi d'optimiser l'utilisation des nœuds de calculs. En effet, en appliquant des délégations de tâches socialement rationnelles, les agents *reducers* sont en mesure d'équilibrer leurs charges de travail et de diminuer le temps d'exécution de la phase de *reduce* pendant son déroulement. De plus, cette méthode ne nécessite pas de connaitre le profil des tâches *a priori*. Formellement, le composant nécessaire à la résolution d'un problème MASTA est une fonction de coût qui, comme on l'a vu, n'est pas nécessairement exacte. Cependant, on constatera dans les chapitres 5 et 6 que (a) considérer la distribution effective des agents, (b) concevoir le protocole d'interaction des agents et (c) proposer des stratégies de sélection de tâches, ajoutent d'autres difficultés.

4.6.2 Perspectives

Comme nous l'avons vu dans l'exemple 4.3, une allocation stable n'est pas forcément optimale. Pour continuer d'améliorer une allocation stable ou atteindre des allocations non atteignables avec des délégations socialement rationnelles, il faudrait que les agents considèrent d'autres formes d'accords. Une possibilité pourrait être d'effectuer des échanges de tâches (ou *swap*) comme définis dans l'exemple 4.2. Avec ce nouveau type d'accord, les agents 1 et 2 de l'exemple 4.3 pourraient directement passer de l'allocation P à l'allocation de *makespan* minimal P^{mks} en échangeant les tâches τ_6 et τ_1.

Une seconde perspective est d'étendre le cadre formel présenté dans ce chapitre. Comme pour MapReduce, les tâches sans date butoir, ni contraintes de précédence (c'est-à-dire qu'elles peuvent être exécutées dans n'importe quel ordre). De même, les ressources nécessaires à l'exécution de l'ensemble des tâches sont disponibles, c'est-à-dire qu'aucune ressource n'est rare. Le modèle demande à être étendu pour prendre en charge d'autres typologies de tâches et des contraintes différentes sur les ressources.

Chapitre 5

Protocole d'interaction

Sommaire

5.1 Introduction

Dans le chapitre précédent, j'ai décrit le problème MASTA qui correspond à la réallocation dynamique de tâches alors que les agents les exécutent dans le but de traiter l'ensemble des tâches au plus tôt. J'ai également décrit ce que sont les délégations de tâches socialement rationnelles qui constituent les opérations grâce auxquelles les agents réallouent les tâches. Deux questions se posent maintenant :

1. comment les agents sélectionnent-ils les tâches qu'ils vont exécuter et celles qu'ils vont proposer à leurs pairs pour une délégation ?
2. lorsqu'un agent a sélectionné une tâche à déléguer, comment la propose-t-il à ses pairs et quel mécanisme concret permet d'aboutir à une délégation de tâche socialement rationnelle ?

Alors que la première question est abordée dans le chapitre suivant, ce chapitre permet d'apporter une réponse à la seconde.

Le protocole d'interaction est un composant essentiel du processus de négociation qui permet aux agents d'équilibrer la charge de travail et de faire décroitre le *makespan* de l'allocation courante. En pratique, les délégations de tâches sont effectuées grâce à des négociations un-à-plusieurs

entre un **initiateur** (c'est-à-dire l'agent qui délègue la tâche) et plusieurs **enchérisseurs** (c'est-à-dire les agents susceptibles de recevoir la tâche). Une fois que l'initiateur a déterminé quelle tâche déléguer, le processus de délégation est divisé en trois étapes. La première consiste à initier la négociation en indiquant aux autres agents quelle tâche est proposée à la délégation. Le seconde étape a lieu quand un agent reçoit la proposition de délégation par l'initiateur. Il doit alors confirmer individuellement que la délégation serait socialement rationnelle si toutefois elle devait avoir lieu. La dernière étape apparaît quand, à partir des réponses des enchérisseurs, l'initiateur choisit auquel d'entre eux il délègue effectivement la tâche. Ces étapes successives font partie d'un processus de négociation basé sur le *Contract Net Protocol* (CNP) [Smith 1980] présenté dans le chapitre 2. Ce processus permet la simultanéité de plusieurs négociations mais un agent est, à un temps donné, soit initiateur, soit enchérisseur. En d'autres termes, tant qu'un agent est impliqué dans des délégations de tâches en tant qu'enchérisseur, il ne peut pas en initier une (et inversement).

Dans ce chapitre, une instance $MASTA = (S, \mathcal{T}, P, c)$ est implicitement définie. Toutefois, pour simplifier le discours, je parlerai uniquement d'un ensemble de tâches $\mathcal{T}$, d'un ensemble d'agents $\mathcal{A}$, d'une fonction de coût c valide (cf. propriété 4.8) et d'une allocation courante P, en supposant les autres éléments du système distribué S quelconques. La section 5.2, complète le formalisme défini au chapitre précédent avec la notion de croyances : dû à l'absence de connaissances communes, les agents possèdent une perception biaisée de l'allocation de tâches ainsi que de la charge de travail de leurs pairs. Ensuite, dans la section 5.3, le processus d'enchères qui donne lieu aux délégations de tâches est décomposé et décrit en détail. La section 5.4 présente le principe des multi-enchères, à savoir l'idée selon laquelle un enchérisseur peut être impliqué dans plus d'une négociation à la fois. Dans la section 5.5, j'explique comment les agents passent en état de pause lorsqu'ils ne peuvent plus initier de délégations de tâches susceptibles d'améliorer le *makespan* de l'allocation courante. Enfin, la section 5.6 présente la conclusion et quelques perspectives.

5.2 Délégations de tâches décentralisées

Afin de ne pas dépendre d'une entité qui centralise l'ensemble des connaissances sur le système et d'éviter les goulots d'étranglement (en termes de calculs et d'échanges de messages), le système est décentralisé et les agents agissent de façon autonome. Ils prennent leur décisions localement selon leur connaissance du système. En effet, si pour une allocation de tâches courante P donnée, un agent i est en mesure de calculer exactement sa propre charge de travail $w_i(P)$, il ne possède en revanche que des **croyances** à propos des charges de travail de ses pairs. Ainsi, il est possible qu'au moment de prendre une décision, la connaissance qu'un agent possède sur la charge de travail d'un autre agent ne soit plus à jour (suite à des délégations ou des consommations de tâche). Cette connaissance incertaine est le prix à payer pour la décentralisation du processus de réallocation.

Définition 5.1 (Base de croyances).
Soit P l'allocation courante. La base de croyances de l'agent i est

$$\mathcal{B}_i(P) = \{w_1^i(P), \ldots, w_{i-1}^i(P), w_{i+1}^i(P), \ldots, w_m^i(P)\} \tag{5.1}$$

> *où $w_j^i(P)$ est la croyance de l'agent i à propos de la charge de travail de l'agent j pour l'allocation P.*

En se référant à ses croyances, il est donc possible qu'un agent ait une vue erronée de l'allocation courante. La base de croyances d'un agent est mise à jour grâce aux messages échangés lors des délégations de tâches (cf. section 5.3).

On rappelle qu'une délégation de tâche est le don d'une tâche d'un agent à un autre (cf. définition 4.3). Cette opération est socialement rationnelle si elle fait décroître le *makespan* local des deux agents (cf. définition 4.4). En prenant maintenant en compte le fait que le mécanisme de délégation de tâches doit être initié par les agents, ces derniers doivent être en mesure d'identifier les tâches susceptibles d'être déléguées de manière socialement rationnelle. Lorsqu'un agent souhaite déléguer une tâche, il doit baser cette décision sur ses croyances individuelles.

> **Définition 5.2 (Ensemble des délégations de tâches potentiellement rationnelles).**
> *Soient i un agent et P l'allocation de tâches courante. L'ensemble de tâches qui pourraient donner lieu à une délégation de tâche socialement rationnelle avec i pour initiateur est :*
>
> $$\Gamma_i^{\mathcal{B}}(P) = \{\tau \in P_i \mid \exists w_j^i(P) \in \mathcal{B}_i(P) \text{ telle que } w_j^i(P) + c_j(\tau) < w_i(P)\} \tag{5.2}$$

Sans connaissance exacte sur les charges de travail de ses pairs, un agent utilise donc sa base de croyances afin de choisir une tâche à déléguer. La base de connaissances n'étant pas exacte, il existe un risque pour qu'un agent initie une délégation qui n'est pas socialement rationnelle. Cependant, le protocole détaillé dans la section 5.3 empêche que de telles délégations aboutissent, ce qui préserve les propriétés de monotonie et de terminaison prouvées dans le chapitre 4 (cf. propriétés 4.1 et 4.2).

5.3 *Contract Net Protocol* pour la délégation de tâche

Cette section décrit en détail le processus d'interaction et son usage pour la délégation de tâche socialement rationnelle. Dans un premier temps, je présente comment le CNP permet aux agents de mener à bien des délégations de tâches tout en mettant régulièrement à jour leurs bases de connaissances individuelles. Lorsque l'on distribue effectivement un tel mécanisme, il faut prendre en compte que les communications entre agents sont **asynchrones** et qu'il n'y a aucune garantie que tous les messages soient effectivement transmis. Dans la seconde partie de cette section, je décris comment le protocole d'interaction a été enrichi pour prendre en compte cette difficulté.

5.3.1 Mécanisme de délégation de tâche

Considérons l'allocation courante P. Comme je l'ai brièvement décrit précédemment, le mécanisme de délégation d'une tâche τ à l'aide du CNP se déroule en trois étapes.

Étape 1 : l'initiateur déclenche un appel à proposition

Considérons l'allocation courante P. L'agent i possède au moins une tâche qui pourrait aboutir à une délégation socialement rationnelle (c'est-à-dire $\Gamma_i^{\mathcal{B}}(P) \neq \emptyset$) et il n'est pas engagé comme enchérisseur dans une autre négociation. Il peut donc déclencher une délégation de tâche socialement rationnelle. Après avoir sélectionné la tâche à déléguer (que l'on note $\tau \in \Gamma_i^{\mathcal{B}}(P)$ dans la suite de cette section), i devient initiateur et émet un appel à proposition via un message `cfp`. Ce message contient trois informations (cf. message 1 de la figure 5.1) :

- un identifiant unique qui sera utilisé dans l'ensemble des messages liés à cette enchère spécifique. Il offre aux agents la possibilité d'avoir une mémoire des enchères auxquelles ils ont participé et des messages qu'ils ont reçus et générés pour une enchère donnée ;
- une description de la tâche à déléguer τ, en particulier la description des ressources nécessaires à son exécution. Ces informations suffisent pour calculer le coût de la tâche pour l'ensemble des agents du système ;
- la charge de travail actuelle de l'initiateur qui permet aux enchérisseurs de calculer si la délégation de τ est socialement rationnelle mais aussi de mettre à jour leur base de croyances.

Lorsqu'il émet un `cfp`, un initiateur s'adresse à l'ensemble de ses pairs, et pas seulement à ceux dont il croit qu'ils sont moins chargés que lui. En effet, s'il ne s'adressait qu'à ceux-ci, il existerait un risque pour que la base de croyances de l'initiateur ne soit pas à jour et qu'il prive le système d'une délégation de tâche socialement rationnelle en ne s'adressant qu'à un sous-ensemble des agents.

Étape 2 : chaque enchérisseur évalue l'appel d'offre

Lorsqu'il reçoit un `cfp`, chaque agent j met à jour sa base de croyances avec la charge courante de l'initiateur. Deux situations sont alors possibles.

L'agent j est initiateur d'une autre enchère. À l'instant, il n'est pas en mesure de faire une proposition. En effet, (a) il ne sait pas encore si la délégation de tâche qu'il a initiée aboutira et (b) il ne peut pas risquer d'augmenter sa charge de travail sous peine d'avoir transmis de fausses informations lors de son propre `cfp`, et donc de perdre la garantie que sa propre délégation de tâche restera socialement rationnelle. Afin de pouvoir envisager de faire une proposition au `cfp` de l'agent i, l'agent j stocke le message et le reconsidérera une fois qu'il ne sera plus initiateur.

L'agent j n'est pas initiateur. S'il n'est pas lui-même à l'origine d'une enchère en cours, l'agent j devient enchérisseur. Un enchérisseur connaît sa propre charge de travail ainsi que celle de l'initiateur, il est en mesure de déterminer si la délégation de tâche est socialement rationnelle en s'assurant que $w_j(P) + c_j(\tau) < w_i(P)$. Si la délégation de tâche est socialement rationnelle, l'enchérisseur indique à l'initiateur qu'il est susceptible d'accepter la délégation via un message `propose` (cf. message 2 de la figure 5.1), sinon il la décline via un message `decline` (cf. message 6 de la figure 5.1). Quelque soit la réponse de l'enchérisseur, il y joint sa charge de travail courante, ce qui permet à l'initiateur de mettre à jour sa base de croyances.

Étape 3 : l'initiateur sélectionne la meilleure offre

La dernière partie du processus concerne l'initiateur qui doit choisir l'agent auquel il va effectivement déléguer la tâche parmi les enchérisseurs qui lui ont répondu. L'initiateur commence par mettre à jour sa base de croyances avec chacune des réponses qu'il a reçues. Ensuite, il choisit l'enchérisseur qui prendra la tâche en charge via un message `accept` (cf. message 3 de la figure 5.1) et rejette les autres enchérisseurs qui ont formulé une proposition via un message `reject` (cf. message 5 de la figure 5.1). Le choix de l'enchérisseur qui prend finalement en charge la tâche déléguée dépend d'une stratégie de choix de l'initiateur. Dans la suite de cette thèse, considérons que l'initiateur choisit l'enchérisseur le moins chargé, ce qui permet de faire décroître un maximum le *makespan* local.

Enfin, une fois que l'enchérisseur choisi par l'initiateur a ajouté la tâche à son lot de tâches, ce dernier confirme à l'initiateur que le transfert de la tâche a bien eu lieu via un message `confirm` (cf. message 4 de la figure 5.1).

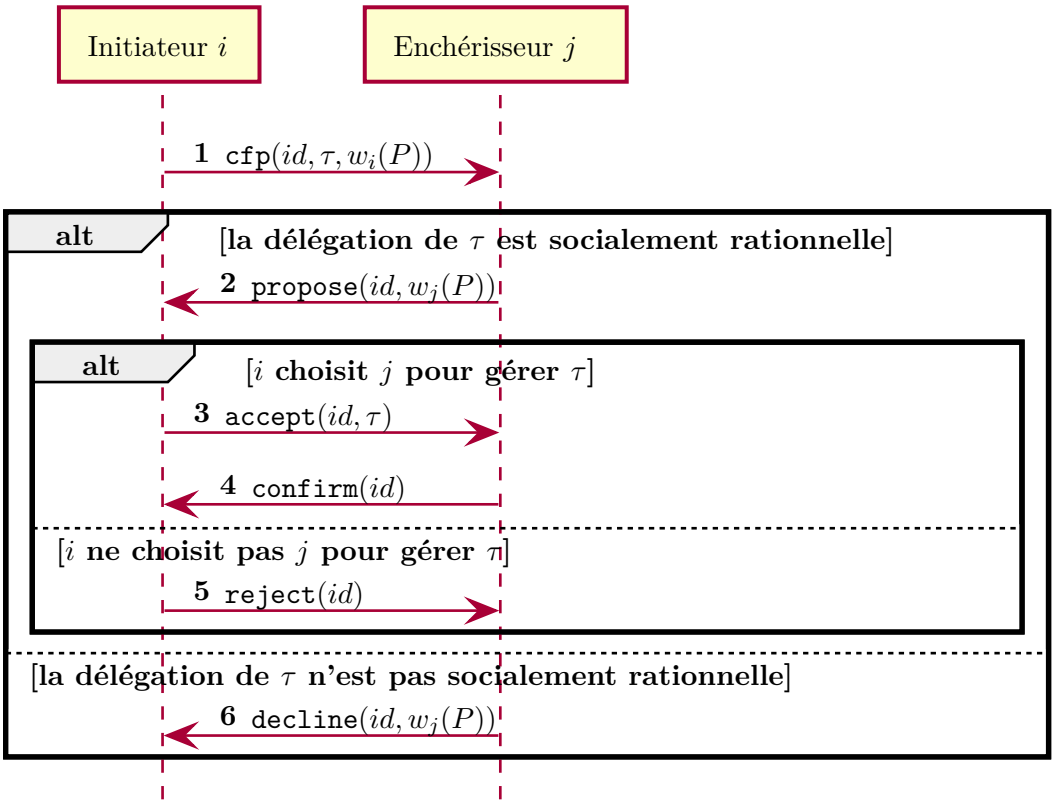


FIGURE 5.1 : Échanges de messages entre un agent initiateur i et un agent enchérisseur j pour la délégation de la tâche τ. Les boîtes « alt » présentent les alternatives possibles. Par exemple, pour celle de premier niveau, les messages au-dessus de la ligne en pointillés concernent le cas où la délégation de τ est socialement rationnelle ; les messages en-dessous de la ligne en pointillés concernent le cas où elle ne l'est pas.

Exemple 5.1 (Délégation de tâche socialement rationnelle à l'aide du CNP). *Soient* $\mathcal{A} = \{1, 2, 3, 4\}$ *quatre agents et* P *l'allocation de tâches courante telle que* $w_1(P) =$

10, $w_2(P) = 8$, $w_3(P) = 3$ et $w_4(P) = 5$. On remarque que l'agent 1 est le plus chargé, on a donc $C_{max}(P) = w_1(P) = 10$ (cf. figure 5.2a). Parmi les tâches allouées à l'agent 1 se trouve la tâche τ pour laquelle $c_1(\tau) = c_2(\tau) = c_3(\tau) = c_4(\tau) = 3$ [a]. Selon ses croyances, l'agent 1 pense être en mesure de déléguer τ, il produit donc un $\texttt{cfp}(id, \tau, 10)$ et l'envoie aux agents 2, 3 et 4 (cf. figure 5.2b). Après avoir mis à jour leurs bases de croyances, les agents doivent déterminer s'ils peuvent prendre en charge la tâche τ. Pour cela, ils vérifient que la délégation de τ est socialement rationnelle (cf. figure 5.2c) et génèrent donc les réponses suivantes :

- *l'agent 2 n'est pas en mesure de prendre τ en charge car $w_2(P) + c_2(\tau) > w_1(P)$ $(11 > 10)$. Il répond donc avec un message $\texttt{decline}(id, 8)$;*
- *l'agent 3 est en mesure de prendre τ en charge car $w_3(P) + c_3(\tau) < w_1(P)$ $(6 < 10)$. Il répond donc avec un message $\texttt{propose}(id, 3)$;*
- *l'agent 4 est en mesure de prendre τ en charge car $w_4(P) + c_4(\tau) < w_1(P)$ $(8 < 10)$. Il répond donc avec un message $\texttt{propose}(id, 5)$.*

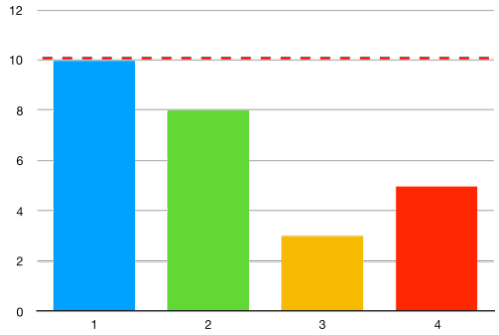
À chaque réponse reçue, l'agent 1 met à jour sa base de croyances avec les charges de travail précises de ses interlocuteurs. Lorsqu'il a reçu une réponse de l'ensemble de ses pairs, il doit décider lequel d'entre eux prendra en charge la tâche τ (cf. figure 5.2d). Le critère de sélection selon lequel τ doit être donnée à l'agent le moins chargé permet de maximiser le gain de makespan local engendré par la délégation de tâche. L'agent 2 ayant décliné l'appel à proposition, l'agent 1 sélectionne donc $\underset{j \in \{3,4\}}{\operatorname{argmin}}(w_j(P)) = 3$ l'agent qui va recevoir τ. Ainsi, l'agent 1 accepte la proposition de l'agent 3 avec un message $\texttt{accept}(id)$ et rejette celle de l'agent 4 avec un message $\texttt{reject}(id)$. En nommant P' l'allocation qui résulte de cette délégation, on a $C_{max}(P') = w_2(P') = 8$ (cf. figure 5.2e). La délégation de τ permet donc de faire décroître le makespan de l'allocation courante ($C_{max}(P') < C_{max}(P)$) et de mieux équilibrer les charges des agents.

a. Le but de cet exemple étant d'illustrer le protocole d'interaction, cette hypothèse simplifie son déroulement.

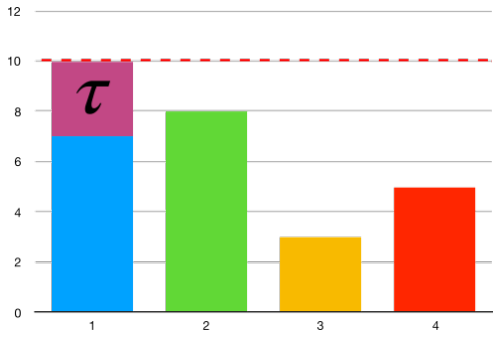
Plusieurs de ces enchères peuvent avoir lieu simultanément, chacune d'entre elles avec son propre initiateur. En itérant ces interactions, les agents réallouent dynamiquement les tâches afin d'améliorer le *makespan* de l'allocation courante.

5.3.2 Distribution du mécanisme d'enchères

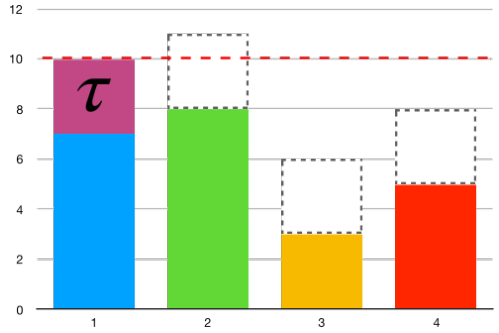
La distribution effective du mécanisme d'enchères décrit dans la section précédente demande de prendre en compte le risque de retard et de perte de message. En effet, comme les agents sont distribués et qu'ils communiquent, il existe un risque que les messages n'arrivent pas à destination. De plus, dans le cas où un message arrive effectivement à destination, il est impossible de garantir un délai entre son émission et sa réception. Les phénomènes de retard ou de perte de messages réclament la mise en place d'interruptions temporelles (ou *timeouts*), à partir desquelles les agents considèrent qu'une opération est terminée ou doit être réitérée. À quelques moments clés du protocole d'interaction, un agent met en place un décompte qui, une fois à son terme, lui rappelle qu'il doit réenvoyer un message pour réinterpeler son interlocuteur ou qu'il doit passer à l'étape suivante.



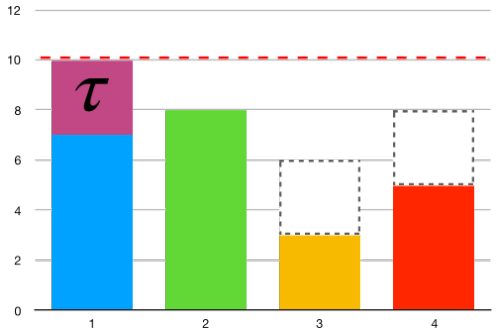
(a) Charges de travail des agents 1, 2, 3 et 4 pour l'allocation P.



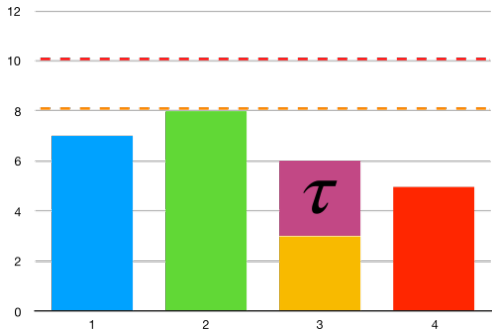
(b) L'agent 1 propose la tâche τ à ses pairs.



(c) Les agents 2, 3 et 4 vérifient si la délégation de τ est socialement rationnelle. Elle ne l'est pas pour l'agent 2.



(d) L'agent 1 décide lequel des agents 3 et 4 va prendre en charge τ.



(e) L'agent 1 délègue τ à l'agent 3. Cela génère une nouvelle allocation P' avec un meilleur *makespan* et des charges de travail plus équilibrées.

FIGURE 5.2 : Illustration des différentes étapes du protocole d'interaction présenté dans l'exemple 5.1.

Par la suite, les trois situations dans lesquelles un initiateur ou un enchérisseur a recours aux *timeouts* sont décrites. La figure 5.3 illustre les différentes situations dans lesquelles un *timeout* peut apparaître. Tous les numéros de messages cités jusqu'à la fin de cette section font référence à cette figure.

Quand l'initiateur attend les propositions des enchérisseurs. Pour que les délégations de tâches se fassent régulièrement, un initiateur n'attend pas forcément une réponse de l'ensemble de ses pairs pour considérer qu'une enchère est terminée. Après un délai (message 8), il conclut l'enchère avec les réponses reçues jusqu'ici. S'il a mis fin à une enchère à cause d'un *timeout*, un enchérisseur peut recevoir après coup une proposition pour une délégation de tâche déjà considérée comme terminée (message 15). Pour déterminer si les propositions qu'il reçoit concernent l'enchère en cours, il suffit à l'initiateur de vérifier que l'identifiant lié à la proposition correspond à l'identifiant de l'appel à propositions courant. Si la proposition reçue concerne une enchère déjà expirée, l'initiateur rejette systématiquement l'enchérisseur via un message `reject` (message 16). En effet, si la délégation est terminée, (a) soit l'initiateur n'a reçu aucune réponse dans le temps imparti et la délégation n'est plus pertinente car la charge de travail communiquée par l'initiateur n'est plus à jour ; (b) soit il a déjà reçu la confirmation que l'enchérisseur choisi a bien pris en charge la tâche (message 14). Quoiqu'il en soit l'initiateur doit donc rejeter toute autre proposition.

Quand l'initiateur attend une confirmation. Une autre situation de *timeout* apparaît lorsqu'un initiateur attend la confirmation de l'enchérisseur sélectionné pour gérer la tâche proposée à la délégation (message 11). Après un certain temps, et jusqu'à ce qu'il reçoive une confirmation, l'initiateur réitère son envoi du message `accept` (message 12). Si l'enchérisseur choisi par l'initiateur reçoit plusieurs messages `accept` (messages 9 et 12), cela peut vouloir dire que ses confirmations précédentes ont été perdues, il répond donc systématiquement par un message `confirm` (messages 13 et 14).

Quand l'enchérisseur attend une réponse de l'initiateur. Après avoir envoyé une proposition, un enchérisseur attend que celle-ci soit acceptée ou rejetée par l'initiateur. Tant qu'il ne reçoit pas de réponse de l'initiateur, l'enchérisseur réitère régulièrement sa proposition (messages 6 puis 7). Si l'initiateur reçoit une proposition qu'il a déjà reçue pour l'appel à proposition en cours (message 7), il l'ignore.

5.4 Multi-enchères

Comme les agents prennent la décision d'initier une délégation de tâche de manière autonome, il est possible que plusieurs délégations issues d'initiateurs différents se produisent en même temps. Les rôles d'enchérisseur et d'initiateur sont mutuellement exclusifs. Comme expliqué dans la section 5.3.1, si un initiateur reçoit un `cfp`, il ne fait pas de proposition. Cependant, une fois qu'un agent est enchérisseur, il est possible qu'il s'engage simultanément dans plusieurs enchères. Cette section décrit comment un enchérisseur gère son implication dans plusieurs enchères simultanées sans risquer d'être un enchérisseur enthousiaste (cf. chapitre 2), c'est-à-dire sans faire des propositions qui l'engagent dans des délégations de tâche qui risqueraient de ne pas être socialement rationnelles.

On note $\mathcal{D} \subset \mathcal{T}$ l'ensemble des tâches en cours de délégation et $\mathcal{D}_j$ l'ensemble des tâches pour

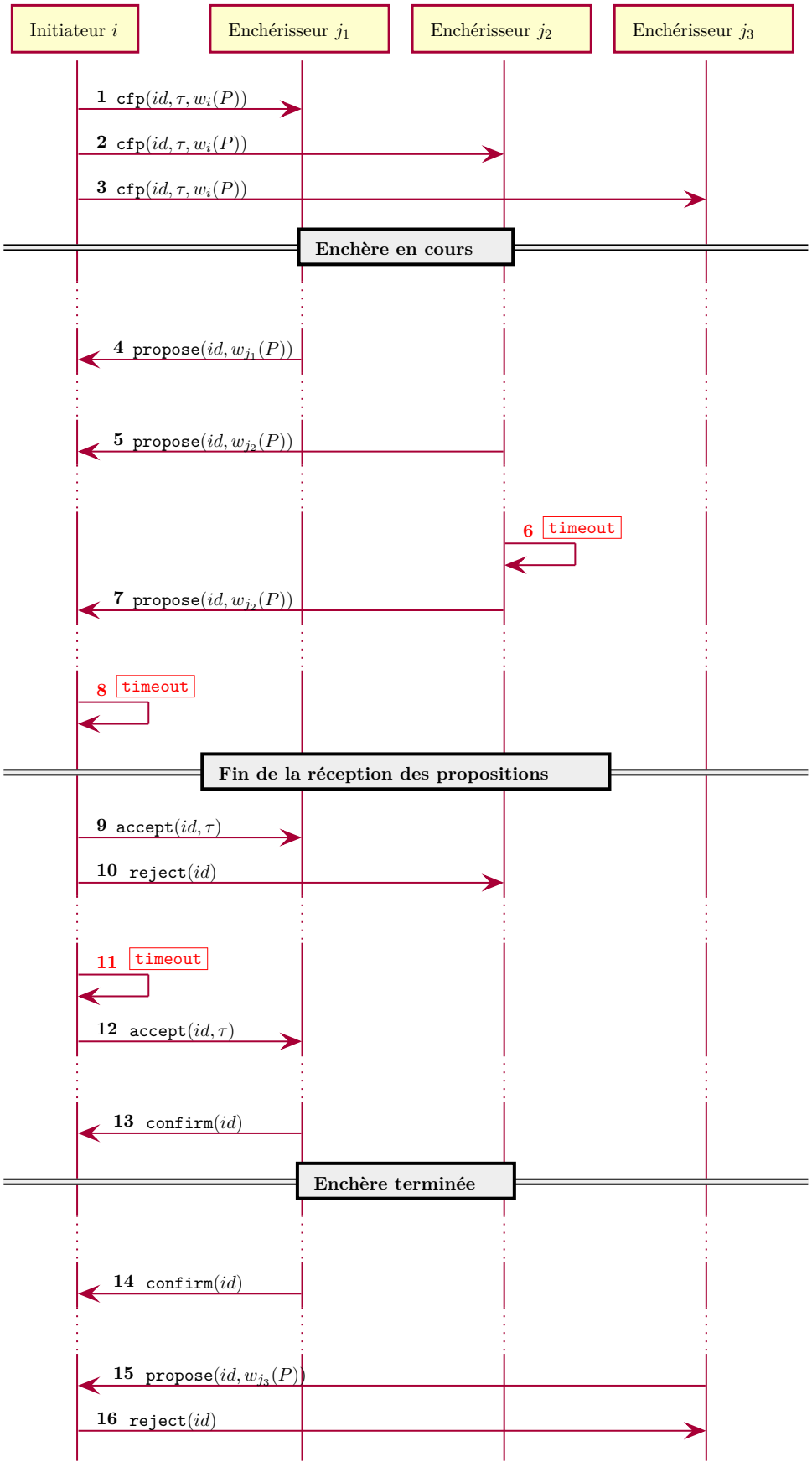


FIGURE 5.3 : Illustration des interruptions temporelles (*timeouts*) susceptibles d'être générées au cours d'une enchère ainsi que de leurs conséquences en terme de réémission de messages.

lesquelles l'agent j a fait une proposition et dont il attend la réponse des initiateurs. Les tâches de $\mathcal{D}_j$ sont donc les tâches que l'agent j est susceptible d'obtenir suite à une délégation socialement rationnelle. Pour gérer des enchères multiples et s'assurer de ne pas faire de propositions qui meneraient à une délégation de tâche qui n'est pas socialement rationnelle, l'enchérisseur maintient une **surcharge de travail virtuelle**.

Définition 5.3 (Surcharge de travail virtuelle).
Soient P l'allocation de tâches courante et $\mathcal{D}_j$ l'ensemble des tâches pour lesquelles l'agent j a fait une proposition et dont il attend la réponse des initiateurs. La surcharge de travail virtuelle de j est :

$$v_j(\mathcal{D}) = \sum_{\mathcal{D}_j} c_j(\tau) \tag{5.3}$$

On appelle charge virtuelle la somme de la charge de travail et de la surcharge virtuelle d'un enchérisseur j $(w_j(P) + v_j(\mathcal{D}))$. La charge virtuelle correspond à la charge dont bénéficierait l'agent s'il remportait l'ensemble des enchères dans lesquelles il s'est engagé. Maintenir sa surcharge virtuelle à jour permet à l'enchérisseur de faire des propositions réalistes et de ne pas être trop optimiste lorsqu'il s'engage dans une délégation de tâche. En effet, en générant des propositions avec sa charge virtuelle, un enchérisseur peut s'engager dans plusieurs délégations de tâches sans risquer d'être un enchérisseur enthousiaste, c'est-à-dire sans que cela risque de donner lieu à un transfert de tâche qui ne serait pas socialement rationnel.

Considérons l'allocation courante P. Lorsqu'un enchérisseur j reçoit un `cfp` pour une tâche τ de la part de l'initiateur i, trois alternatives sont possibles :

- si $w_j(P) + c_j(\tau) \geq w_i(P)$, l'enchérisseur décline l'enchère car elle ne peut pas mener à une délégation socialement rationnelle ;
- si $w_j(P)+v_j(\mathcal{D})+c_j(\tau) < w_i(P)$, l'enchérisseur génère une proposition en indiquant $w_j(P)$. Cette opération est sûre : si j remporte l'enchère, la délégation de τ sera socialement rationnelle quels que soient les dénouements des autres enchères dans lesquelles j s'est engagé ;
- si $w_j(P) + c_j(P) < w_i(P) \leq w_j(P) + v_j(\mathcal{D}) + c_j(\tau)$, j pourrait générer une proposition pour τ en fonction des dénouements des autres enchères dans lesquelles il est engagé. j stocke le `cfp` et envisagera d'y répondre plus tard.

En considérant la fin de δ, la délégation de la tâche τ dans laquelle l'agent j est engagé comme enchérisseur, on pose $P' = \delta(P)$ et $\mathcal{D}' = \mathcal{D} \setminus \{\tau\}$. Deux situations sont possibles.

- Soit j est choisi par l'initiateur et :

$$\begin{aligned} P'_j &= P_j \cup \{\tau\} \\ w_j(P') &= w_j(P) + c_j(\tau) \\ v_j(\mathcal{D}') &= v_j(\mathcal{D}) - c_j(\tau) \end{aligned}$$

- Soit j est rejeté par l'initiateur et :

$$\begin{aligned} P'_j &= P_j \\ w_j(P') &= w_j(P) \\ v_j(\mathcal{D}') &= v_j(\mathcal{D}) - c_j(\tau) \end{aligned}$$

Chaque fois qu'une de ses enchères se termine, un enchérisseur passe en revue l'ensemble des `cfp` qu'il a stockés du plus ancien au plus récent et envisage à nouveau de faire une proposition (ou de décliner) en utilisant sa nouvelle charge de travail virtuelle. Lorsqu'un enchérisseur j a stocké un $\texttt{cfp}_1$ d'un initiateur i, il se peut que l'enchère liée à $\texttt{cfp}_1$ se termine avant que j fasse une proposition ou décline. Dans le cas où i initie une nouvelle enchère, j reçoit un second $\texttt{cfp}_2$ de la part de i. Il n'est plus nécessaire pour j de conserver $\texttt{cfp}_1$, il peut l'oublier et le remplacer $\texttt{cfp}_2$. Pour résumer, un enchérisseur ne conserve que le `cfp` le plus récent de chaque initiateur.

Exemple 5.2 (Multi-enchères).
Soient P l'allocation courante et $\mathcal{A} = \{1, 2, 3, 4, 5, 6\}$ l'ensemble des agents. Dans P, les agents possèdent les charges de travail suivantes :

$$w_1(P) = 30 \quad w_2(P) = 45 \quad w_3(P) = 39 \quad w_4(P) = 28 \quad w_5(P) = 34 \quad w_6(P) = 40$$

Considérons uniquement ici l'évolution de la charge virtuelle de l'agent 1 et son impact sur les réponses de l'agent. Pour cela, les agents $i \in \{2, 3, 4, 5, 6\}$ vont successivement soumettre une tâche τ_i à l'agent 1. Les coûts des tâches τ_i pour l'agent 1 sont les suivants :

$$c_1(\tau_2) = 6 \quad c_1(\tau_3) = 2 \quad c_1(\tau_4) = 4 \quad c_1(\tau_5) = 3 \quad c_1(\tau_6) = 3$$

Les figures 5.4 et 5.5 illustrent les interactions de l'agent 1 avec chacun de ses pairs ainsi que l'évolution de ses charges réelle et virtuelle. Lorsqu'il reçoit les appels à propositions des agents 2 et 3 (messages 1 et 3), l'agent 1 est en mesure de faire une proposition (messages 2 et 4, figures 5.5b et 5.5c). En revanche, la délégation proposée par l'agent 4 (message 5) n'est pas socialement rationnelle, elle est donc rejetée (message 6 et figure 5.5d). Lorsqu'il reçoit les appels à proposition des agents 5 et 6 (messages 7 et 8), l'agent 1 se trouve dans la situation où il doit stocker ces `cfp` pour les reconsidérer plus tard. En d'autres termes, sa charge virtuelle ne lui permet pas de faire une proposition mais sa charge réelle oui (cf. figures 5.5e et 5.5f). L'agent 1 sera éventuellement en mesure de faire une proposition pour ces `cfp` en fonction du dénouement des délégations dans lesquelles il est engagé. Après la confirmation de la délégation de la tâche τ_2 (message 10 et figure 5.5g), l'agent 1 réévalue les `cfp` des agents 5 et 6. Avec ses nouvelles charges de travail (réelle et virtuelle), il décline celui de l'agent 5 (message 11 et figure 5.5h) mais ne peut toujours pas se décider pour celui de l'agent 6 (cf. figure 5.5i). Une fois que sa proposition est rejetée par l'agent 3 (message 12 et figure 5.5j), l'agent 1 génère une proposition pour l'agent 6 (message 12 et figure 5.5k).

5.5 État de pause

Comme nous l'avons vu dans le chapitre 4, il n'est plus possible d'améliorer le *makespan* d'une allocation stable grâce aux délégations de tâches socialement rationnelles. N'ayant qu'une connaissance partielle de l'allocation courante et des charges de travail qu'elle induit, les agents doivent être capables d'identifier les situations dans lesquelles il n'est plus possible d'effectuer une délégation socialement rationnelle pour éviter de déclencher des enchères inutiles. Lorsqu'un agent

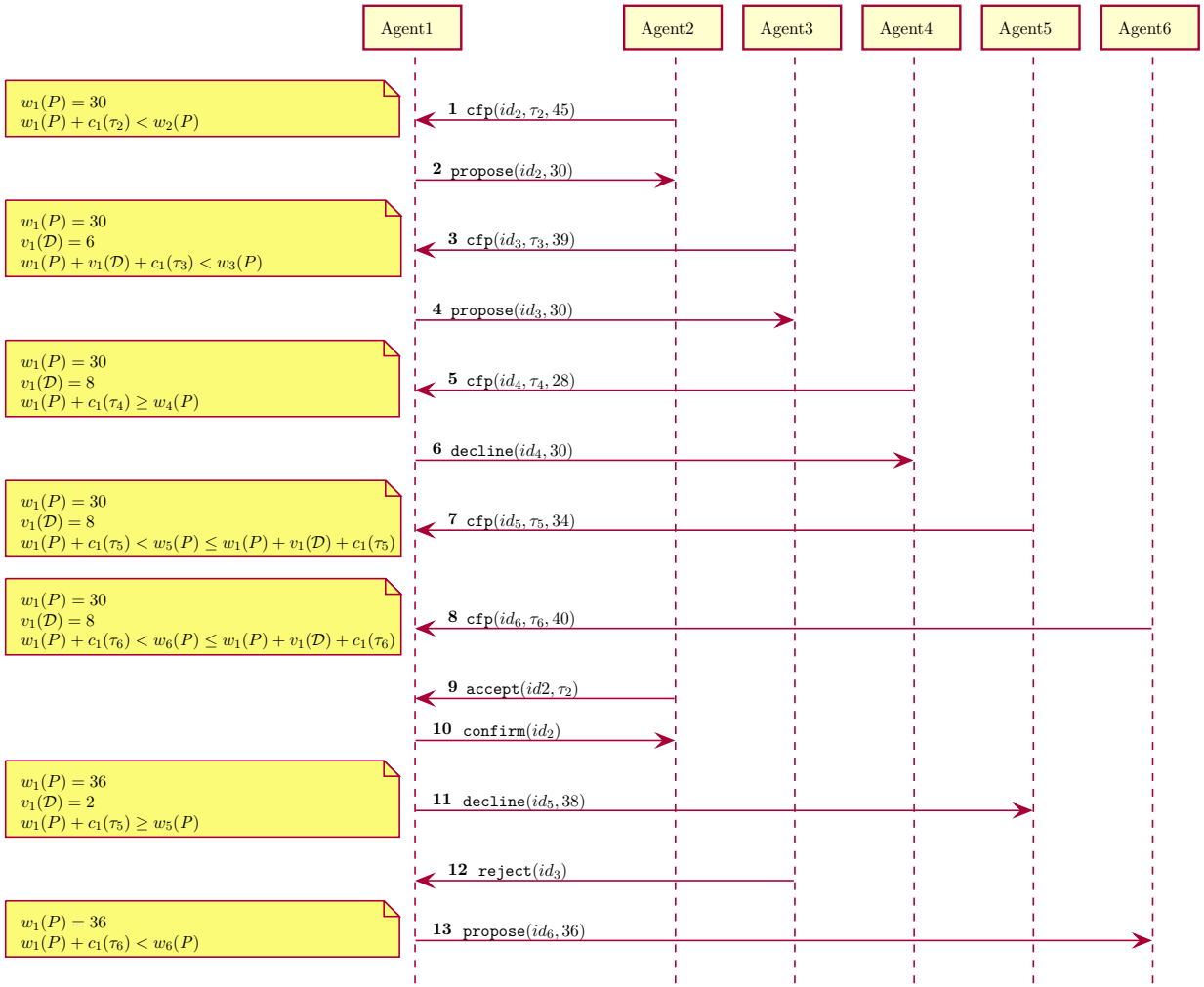
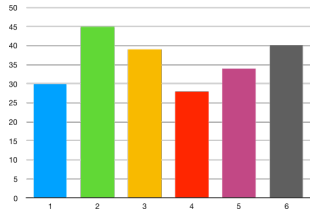
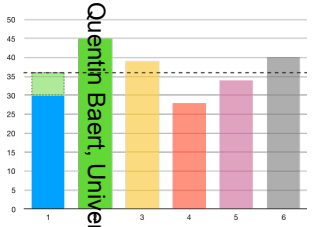


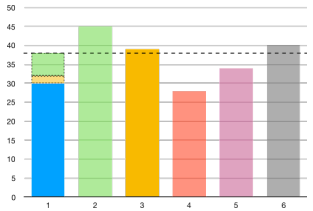
FIGURE 5.4 : Interactions décrites dans l'exemple 5.2.



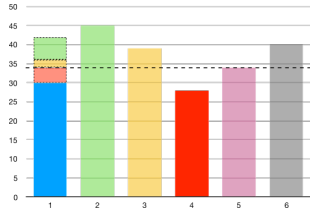
(a) Situation initiale.



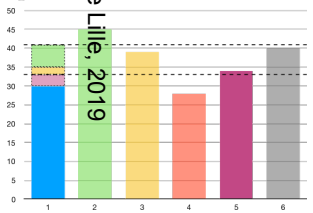
(b) L'agent 1 reçoit le `cfp` de l'agent 2 et fait une proposition.



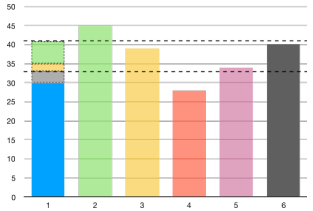
(c) L'agent 1 reçoit le `cfp` de l'agent 3 et fait une proposition.



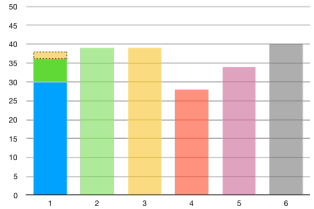
(d) L'agent 1 reçoit le `cfp` de l'agent 4 et décline.



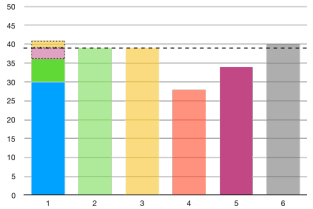
(e) L'agent 1 reçoit le `cfp` de l'agent 5 et le stocke.



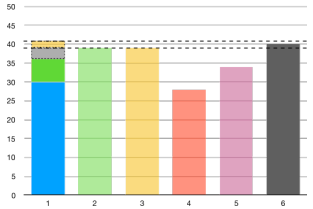
(f) L'agent 1 reçoit le `cfp` de l'agent 6 et le stocke.



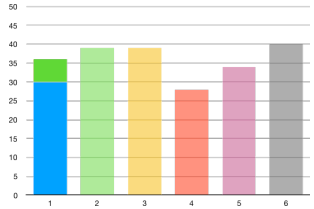
(g) La proposition de l'agent 1 est acceptée par l'agent 2.



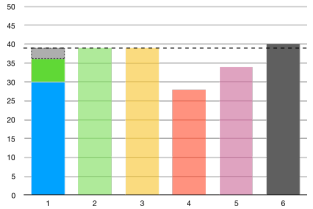
(h) L'agent 1 reconsidère le `cfp` de l'agent 5 et le décline.



(i) L'agent 1 reconsidère le `cfp` de l'agent 6. Il ne peut pas prendre de décision.



(j) La proposition de l'agent 1 est rejetée par l'agent 3.



(k) L'agent 1 reconsidère le `cfp` de l'agent 6 et fait une proposition.

FIGURE 5.5 : Évolution des charges réelle et vituelle de l'agent 1 de l'exemple 5.2. Les tâches en transparence et entourées de pointillés sont celles qui constituent la surcharge de travail virtuelle de l'agent 1. La couleur d'une tâche est celle de l'agent qui a initié sa délégation. Les lignes en pointillé représentent les valeurs utilisées par l'agent 1 pour se comparer avec l'initiateur du `cfp` courant.

se trouve dans cette situation, il passe en **état de pause**, c'est-à-dire un état dans lequel il répond aux `cfp` de ses pairs mais n'en émet plus lui-même.

La première situation dans laquelle un agent i passe en état de pause apparaît quand, selon sa base de croyances sur l'allocation P, aucune de ses tâches n'est délégable, c'est-à-dire $\Gamma_i^{\mathcal{B}}(P) = \emptyset$. L'agent considère alors qu'il est inutile d'émettre un `cfp` car aucun de ses pairs ne serait en mesure d'accepter une délégation de tâche.

Une seconde situation de passage en pause apparaît lorsqu'un agent ayant émis un `cfp` reçoit un message `decline` de l'ensemble de ses pairs. Il sait alors que la tâche qu'il estime la plus susceptible d'être déléguée n'est en fait pas socialement délégable. Il faut préciser qu'un agent décide de quelle tâche joindre à un `cfp` grâce à sa stratégie de sélection de tâche. Cette stratégie est une heuristique qui retourne à l'agent la tâche à déléguer en fonction de ses croyances (cf. chapitre 6). Ainsi, lorsque l'ensemble des pairs d'un initiateur décline son appel à proposition, ce dernier passe en pause et cesse d'émettre des `cfp`.

La sortie d'état de pause d'un agent est uniquement conditionnée par sa base de croyances. Même si un agent i est en pause, ce dernier continue de recevoir des `cfp` et d'y répondre. Sa base de croyances est donc mise à jour régulièrement. De plus, i exécute ses tâches en continu, ce qui fait baisser continuellement sa charge de travail. Dès lors que sa charge de travail ou qu'une mise à jour de sa base de croyances fait qu'une de ses tâches devient potentiellement délégable, c'est-à-dire $\Gamma_i^{\mathcal{B}}(P) \neq \emptyset$, l'agent i sort de l'état de pause et peut à nouveau initier une enchère.

L'état de pause permet de réguler les enchères durant l'exécution des tâches. Les agents initient les délégations de tâches nécessaires au rééquilibrage de leurs charges de travail et peuvent ainsi faire décroître le *makespan* de l'allocation courante. Si aucune des tâches d'un agent n'est délégable selon ses croyances, il n'a plus à initier de `cfp` mais continue de répondre à ceux de ses pairs. Si les croyances des agents sont exactes, dire qu'une allocation est stable équivaut à dire que tous les agents sont en pause.

5.6 Conclusion

5.6.1 Synthèse

Le processus d'interaction décrit dans ce chapitre permet aux agents d'effectuer des délégations de tâche socialement rationnelles pour améliorer le *makespan* de l'allocation de tâches courante. Basé sur le *Contract Net Protocol*, ce mécanisme en trois étapes implique deux types d'agents : un initiateur et des enchérisseurs. L'initiateur propose une délégation de tâche à l'ensemble de ses pairs via un message `cfp`. Un agent qui reçoit un `cfp` détermine si le délégation de la tâche proposée par l'initiateur est socialement rationnelle. Si c'est le cas, il fait une proposition à l'initiateur et devient l'un des enchérisseurs. Sinon, il décline l'appel à proposition et n'est pas engagé dans la délégation de tâche.

Les messages envoyés par les agents dans le cadre des délégations de tâches contiennent peu d'informations. Lorsqu'un initiateur émet un `cfp` il y joint un identifiant unique qui permet aux agent de reconnaître les messages liés à cette délégation, sa charge de travail actuelle ainsi qu'une description de la tâche à déléguer. Ces informations sont suffisantes pour calculer le coût

d'une tâche pour l'ensemble des agents du système.

Le mécanisme d'enchères est décentralisé, robuste à la perte de messages et les agents n'ont pas de connaissances communes : leurs décisions sont locales. De ce fait, chaque agent entretient sa propre base de croyances sur les charges de travail de ses pairs. Cette base est régulièrement mise à jour grâce aux enchères. En effet, lorsque les agents communiquent pour mener à bien des délégations de tâche, ils indiquent leur charge de travail actuelle. Chaque agent utilise ses croyances pour décider d'initier ou non une enchère. La stratégie de sélection de tâche qu'utilise un agent pour décider quelle tâche proposer à la délégation fait l'objet du chapitre 6.

Un enchérisseur peut être engagé dans plusieurs délégations simultanément. Afin d'éviter d'être un enchérisseur enthousiaste, c'est-à-dire de génèrer des propositions qui ne tiendraient pas compte de ses multiples engagements, il entretient une surcharge de travail virtuelle qu'il utilise pour décider s'il est en mesure de faire une proposition lorsqu'un nouveau `cfp` lui arrive.

Enfin, les agents passent en état de pause lorsque, selon leurs croyances ou les réponses de leurs pairs, ils ne sont plus en mesure d'initier une délégation de tâche qui soit socialement rationnelle. Dans cet état, un agent n'initie plus de `cfp` mais continue de répondre à ceux de ses pairs. Les mises à jour régulières de la base de croyances des agents et les consommations de tâches continues leur permettent de sortir de pause lorsque de nouvelles délégations de tâches deviennent possibles.

5.6.2 Perspective

La décentralisation effective du protocole d'interaction décrit dans ce chapitre réclame la mise en place d'interruptions temporelles aux points clés du protocole afin de prendre en compte les pertes ou retards de messages. Cependant, le protocole n'est pas adapté à la panne d'un nœud de calcul, c'est-à-dire à la perte de contact avec un agent pendant une délégation de tâche. Par exemple, lorsqu'un initiateur attend la confirmation de la part de l'enchérisseur choisi pour gérer la tâche qu'il délègue, il réitère le message `accept` tant qu'il ne reçoit pas de `confirm`. Si le nœud sur lequel se trouve l'enchérisseur ne répond plus, l'initiateur ne considère jamais sa délégation en cours comme terminée. Ainsi, il continue à exécuter ses tâches, ce qui permet à l'ensemble du processus de terminer normalement, mais n'initie plus de `cfp` et n'est enchérisseur dans aucune autre délégation. Une perspective est donc de compléter le protocole et le mécanisme de *timeouts* afin qu'en plus d'être tolérant aux pertes/retards de messages, le mécanisme de délégations de tâches soit également robuste à la perte d'un nœud pendant le processus de délégation. Soulignons également que la perte d'un nœud entraine d'autres problématiques importantes comme la duplication des ressources uniquement sur ce nœud ou la réallocation des tâches allouées à l'agent de ce nœud.

Chapitre 6

Stratégies

Sommaire

6.1 Introduction

Comme nous l'avons vu dans le chapitre 4, une allocation de tâches (indépendantes, non préemptives et sans dates butoirs) dans un système distribué peut être représentée par une instance MASTA. Le processus dynamique qui consiste à ce que les agents réallouent les tâches pendant leur exécution a pour objectif d'améliorer le *makespan* de l'allocation courante et ainsi réduire le temps d'exécution global de l'ensemble des tâches. Dans le cadre du formalisme MASTA, cela revient à appliquer deux types d'opérations sur l'instance MASTA courante : les délégations de tâche socialement rationnelles et les consommations de tâches. Une délégation socialement rationnelle est un don de tâche d'un agent à un autre de sorte à faire décroître leur *makespan* local. Consommer une tâche revient à supprimer une tâche de l'ensemble des tâches à exécuter. On considère qu'une tâche est consommée lorsqu'un agent a terminé son exécution et a produit son résultat. Pour choisir la prochaine tâche à consommer/déléguer, les agents ont recours à une **stratégie de sélection de tâche**. Une telle stratégie dépend des charges de travail des agents et des croyances qu'ils possèdent sur les charges de travail de leurs pairs.

Dans le cas où certaines tâches sont bien plus coûteuses que les autres, il est possible qu'une allocation de tâches soit stable mais déséquilibrée. Dans ce cas, et si la nature des tâches le

permet, il peut être nécessaire pour un agent de découper une de ses tâches en sous-tâches pour ensuite les déléguer et créer de nouvelles opportunités d'affiner l'équilibre des charges de travail et d'améliorer le *makespan* de l'allocation courante. Les agents sont alors munis d'une **stratégie de découpe de tâche** qui détermine quelle tâche découper en combien de sous-tâches.

Les sections 6.2 et 6.3 présentent respectivement les stratégies de sélection de tâches et de découpes de tâches. Enfin, la section 6.4 synthétise le chapitre et offre quelques perspectives.

6.2 Stratégie de sélection de tâche

L'objectif des agents est d'exécuter l'ensemble des tâches $\mathcal{T}$ au plus tôt malgré leur connaissance partielle de l'allocation courante et les perturbations qui peuvent intervenir en cours d'exécution (par exemple le ralentissement d'un nœud de calcul). Ils y parviennent en réitérant régulièrement des délégations de tâches socialement rationnelles pendant qu'ils consomment les tâches. La co-occurence de ces deux opérations permet de faire décroître le *makespan* de l'allocation courante pendant le traitement des tâches. Pour choisir la tâche à exécuter et la tâche à déléguer, les agents adoptent une stratégie de sélection de tâche qui considère l'allocation de tâche courante et leurs croyances. Dans la suite de ce chapitre, considérons l'allocation courante P et les choix de l'agent $i \in \mathcal{A}$ par rapport à son lot de tâches P_i.

6.2.1 Exécution et délégation de tâches

Les agents sont conçus pour être **capables d'exécuter et de déléguer des tâches simultanément**. L'architecture logicielle qui permet aux agents de procéder à ces deux opérations de manière concurrente est décrite dans le chapitre 7.

Exécution de tâche. L'objectif étant d'exécuter l'ensemble des tâches le plus rapidement possible, les tâches sont consommées tout au long du processus. Ainsi, dès qu'un agent possède des tâches dans son lot de tâches (c'est-à-dire $P_i \neq \emptyset$), il lance l'exécution de l'une d'entre elles. Quand l'exécution d'une tâche est terminée et si son lot de tâches est vide, l'agent devient inactif même s'il continue à répondre aux appels à proposition de ses pairs. Dans le cas contraire, il commence immédiatement à exécuter une nouvelle tâche. Tant que son lot de tâches n'est pas vide, un agent doit décider quelle tâche exécuter. On note cette tâche $\overset{>}{\tau} \in P_i$.

Délégation de tâche. Les délégations servent à minimiser le *makespan* global de l'allocation de tâches courante. Afin de déterminer quelle tâche est délégable, l'agent se réfère à sa base de croyances. Le but de l'agent est alors de choisir une tâche à déléguer qui fera diminuer sa charge de travail le plus possible tout en faisant décroître son *makespan* local avec l'un de ses pairs (et éventuellement le *makespan* global de l'allocation). On note $\overset{<}{\tau} \in \Gamma_i^{\mathcal{B}}(P)$ la tâche sélectionnée par l'agent i pour la déléguer. On souligne que cette tâche est sélectionnée parmi l'ensemble des tâches de i dont la délégation est socialement rationnelle selon ses croyances (cf. définition 5.2).

La stratégie de sélection de tâche d'un agent définit la prochaine tâche à exécuter/déléguer.

Définition 6.1 (Stratégie de sélection de tâche).
Soit P l'allocation de tâches courante. La stratégie de sélection de tâche de l'agent $i \in \mathcal{A}$

est un couple de fonctions $(perform, delegate)$ *où :*

- $perform : 2^{\mathcal{T}} \mapsto \mathcal{T} \cup \{\bot\}$*, sélectionne* $\overset{>}{\tau} \in P_i$ *la prochaine tâche à exécuter ou retourne* $\bot$ *si et seulement si* $P_i = \emptyset$ *;*
- $delegate : 2^{\mathcal{T}} \mapsto \mathcal{T} \cup \{\bot\}$*, sélectionne* $\overset{<}{\tau} \in \Gamma_i^{\mathcal{B}}(P)$ *ou retourne* $\bot$ *s'il n'existe pas de telle tâche.*

Alors que l'on pourrait munir les agents d'une stratégie d'exécution de tâche ($perform$) et une stratégie de délégation de tâche ($delegate$), je définis une stratégie comme un couple $(perform, delegate)$ car il est essentiel qu'il existe une cohérence entre les deux fonctions. Pour que le comportement d'un agent soit cohérent et, par exemple, qu'il évite d'exécuter une tâche qu'il est en train d'essayer de déléguer, il faut que les fonctions $perform$ et $delegate$ soient complémentaires. Un agent ne sélectionne jamais simultanément une tâche à déléguer et une tâche à exécuter : ces choix sont faits en séquence. Lorsqu'un agent sélectionne une tâche à exécuter, celle-ci est supprimée de son lot de tâches et ne peut donc plus être sélectionnée pour une délégation. Lorsqu'un agent sélectionne une tâche à déléguer, celle-ci est soumise aux enchères en utilisant le protocole présenté dans le chapitre 5 mais reste dans le lot de tâches tant qu'elle n'est pas effectivement déléguée. Si, pendant l'enchère, l'agent sélectionne cette même tâche pour l'exécuter, alors la priorité est donnée à l'exécution. En effet, l'organisation interne de l'agent (cf. chapitre 7) permet de mettre un terme à l'enchère et de ne pas déléguer la tâche.

Dans la suite de cette section, je présente trois types de stratégies de sélection de tâche. Les deux premières considèrent uniquement le coût des tâches ; la troisième tient également compte de la localité des ressources nécessaires à l'exécution des tâches.

6.2.2 Stratégies de sélection de tâche par coûts ordonnés

Les **stratégies de sélection de tâche par coûts ordonnés** reposent uniquement sur les coûts des tâches pour l'agent qui sélectionne, sans considérer ses croyances sur les charges de ses pairs. Ces stratégies présentent l'avantage d'être peu coûteuses pour la recherche de tâche. En effet, elles demandent simplement à l'agent de maintenir ses tâches triées par ordre de coût croissant. Lorsqu'il doit sélectionner une tâche, l'agent choisit la première ou la dernière de ses tâches ordonnées, ce qui permet de définir naturellement deux stratégies. La première, notée CB-DS (pour *Consume Big - Delegate Small*), consiste à consommer les tâches les plus coûteuses et déléguer les moins coûteuses d'abord. La seconde, notée CS-DB (pour *Consume Small - Delegate Big*), consiste à consommer les tâches les moins coûteuses et déléguer les plus coûteuses d'abord. Ces deux stratégies demandent de faire un pari sur l'état courant de l'allocation de tâches et implémentent deux raisonnements antagonistes.

Stratégie de sélection de tâche CB-DS

La stratégie de sélection de tâche CB-DS adopte le principe suivant : comme le coût des tâches tient compte du nombre de ressources nécessaires à leur exécution, les tâches les moins coûteuses nécessitent généralement le moins de ressources. De ce fait, elles sont susceptibles d'être peu coûteuses pour tous les agents du système et représentent potentiellement une délégation socialement rationnelle.

La stratégie de sélection de tâche par coûts ordonnés CB-DS de l'agent i est définie comme :

$$\overset{\geq}{\tau} = \underset{\tau \in P_i}{\operatorname{argmax}}(c_i(\tau)) \tag{6.1}$$

$$\overset{\leq}{\tau} = \underset{\tau \in \Gamma_i^{\mathcal{B}}(P)}{\operatorname{argmin}}(c_i(\tau)) \tag{6.2}$$

Stratégie de sélection de tâche CS-DB

La stratégie de sélection de tâche CS-DB adopte le principe suivant : si une tâche est coûteuse pour un agent et qu'il est possible de la déléguer, alors il est probable qu'elle soit moins coûteuse pour un autre agent du système. Un agent commence donc par consommer les tâches les moins coûteuses qui lui ont été allouées, en essayant simultanément de déléguer ses tâches les plus coûteuses pour avoir un maximum d'impact sur le *makespan* de l'allocation courante.

La stratégie de sélection de tâche par coûts ordonnés CS-DB de l'agent i est définie comme :

$$\overset{\geq}{\tau} = \underset{\tau \in P_i}{\operatorname{argmin}}(c_i(\tau)) \tag{6.3}$$

$$\overset{\leq}{\tau} = \underset{\tau \in \Gamma_i^{\mathcal{B}}(P)}{\operatorname{argmax}}(c_i(\tau)) \tag{6.4}$$

Pour ces deux stratégies, la fonction *delegate* renvoie $\perp$ si et seulement si $\Gamma_i^{\mathcal{B}}(P) = \emptyset$. Dans ce cas, l'agent peut passer en état de pause.

6.2.3 Stratégies de sélection de tâche k-éligible

Les **stratégies de sélection de tâche k-éligible** sont plus complexes que les stratégies précédentes car elles s'appuient sur les croyances des agents. L'idée sous-jacente à ces stratégies est de déléguer la tâche la plus coûteuse possible tout en s'assurant que la délégation ait des chances raisonnables d'aboutir. Pour ce faire, ces stratégies demandent un paramètre `k` qui représente le nombre minimum de pairs susceptibles d'accepter la tâche à déléguer. Ainsi une stratégie k-éligible sélectionne la tâche la plus coûteuse qui constitue une délégation socialement rationnelle pour au moins `k` pairs selon la base de croyance de l'agent. Là encore, deux variantes découlent de ce principe : la première où les agents consomment les tâches les moins coûteuses d'abord (notée k-éligible CS pour *Consume Small*), la seconde où les agents consomment les tâches les plus coûteuses d'abord (notée k-éligible CB pour *Consume Big*). Pour appliquer ces stratégies, un agent doit être en mesure d'identifier les tâches de son lot qui sont k-éligibles.

Définition 6.2 (Ensemble des tâches k-éligibles).
Soit $\Gamma_i^{\mathcal{B}}(P)$ l'ensemble des tâches de l'agent $i \in \mathcal{A}$ qui peuvent aboutir à une délégation socialement rationnelle d'après ses croyances. Étant donné un paramètre $\mathtt{k}$, l'ensemble des tâches k-éligibles de i est :

$$\Gamma_i^{\mathcal{B},\mathtt{k}}(P) = \{\tau \in P_i \mid \exists \mathcal{A}' \subseteq \mathcal{A} \text{ tel que } \forall j \in \mathcal{A}', w_j^i(P) + c_j(\tau) < w_i(P) \text{ et } |\mathcal{A}'| \geq \mathtt{k}\} \tag{6.5}$$

Stratégie de sélection de tâche k-éligible CS

La stratégie de sélection de tâche k-éligible CS de l'agent i est définie comme :

$$\overset{>}{\tau} = \underset{\tau \in P_i}{\operatorname{argmin}}(c_i(\tau)) \tag{6.6}$$

$$\overset{<}{\tau} = \underset{\tau \in \Gamma_i^{\mathcal{B},\mathtt{k}}(P)}{\operatorname{argmax}}(c_i(\tau)) \tag{6.7}$$

Stratégie de sélection de tâche k-éligible CB

La stratégie de sélection de tâche k-éligible CB de l'agent i est définie comme :

$$\overset{>}{\tau} = \underset{\tau \in P_i}{\operatorname{argmax}}(c_i(\tau)) \tag{6.8}$$

$$\overset{<}{\tau} = \underset{\tau \in \Gamma_i^{\mathcal{B},\mathtt{k}}(P)}{\operatorname{argmax}}(c_i(\tau)) \tag{6.9}$$

Pour ces deux stratégies, la fonction *delegate* renvoie $\perp$ dans le cas où il n'existe aucune tâche k-éligible pour un paramètre `k` donné. Dans cette situation, une variante de ces stratégies est de relancer une recherche avec le paramètre $\mathtt{k}-1$. Si *delegate* renvoie $\perp$ pour $\mathtt{k}=1$, alors aucune tâche n'est délégable et l'agent peut passer en état de pause.

Exemple 6.1 (Stratégie de sélection de tâche k-éligible).
Soit $MASTA = (S, \mathcal{T}, P, c)$ une instance MASTA telle que $\mathcal{A} = \{1,2,3\}$, $\mathcal{T} = \{\tau_1, \ldots, \tau_5\}$ et la fonction de coût c est définie dans le tableau 6.1. On considère l'allocation P telle que :

- $P_1 = \{\tau_2, \tau_3\}$, $w_1(P) = 49$;
- $P_2 = \{\tau_5\}$, $w_2(P) = 11$;
- $P_3 = \{\tau_1, \tau_4\}$, $w_3(P) = 24$.

Si les agents utilisent une stratégie de sélection de tâche k-éligible, et en supposant leurs croyances exactes, on a :

- *pour* $\mathtt{k} = 2$,
  - $\Gamma_1^{\mathcal{B},\mathtt{k}}(P) = \{\tau_2, \tau_3\}$ *et* $\overset{<}{\tau} = \tau_3$;
  - $\Gamma_2^{\mathcal{B},\mathtt{k}}(P) = \emptyset$ *et* $delegate(P_2) = \perp$;
  - $\Gamma_3^{\mathcal{B},\mathtt{k}}(P) = \emptyset$ *et* $delegate(P_3) = \perp$.
- *pour* $\mathtt{k} = 1$,
  - $\Gamma_1^{\mathcal{B},\mathtt{k}}(P) = \{\tau_2, \tau_3\}$ *et* $\overset{<}{\tau} = \tau_3$;
  - $\Gamma_2^{\mathcal{B},\mathtt{k}}(P) = \emptyset$ *et* $delegate(P_2) = \perp$;
  - $\Gamma_3^{\mathcal{B},\mathtt{k}}(P) = \{\tau_1\}$ *et* $\overset{<}{\tau} = \tau_1$.

Les stratégies k-éligibles présentent l'avantage de sélectionner les tâches à déléguer plus finement que les stratégies par coûts ordonnés. Cependant, elles nécessitent de parcourir l'ensemble du lot de tâches pour chaque délégation, voir pour chaque valeur de `k`. Dans l'hypothèse de lots de

	τ_1	τ_2	τ_3	τ_4	τ_5
$c_1(\tau_k)$	10	14	35	20	8
$c_2(\tau_k)$	8	13	21	20	11
$c_3(\tau_k)$	12	13	32	12	13

Tableau 6.1 : Fonction de coût associée à l'exemple 6.1.

tâches conséquents [1], l'utilisation des stratégies k-éligibles pourrait s'avérer trop coûteuse pour être pratiquable.

6.2.4 Stratégie de sélection de tâche localisée

À cause de la localité des ressources qui lui sont associées, il est possible qu'une tâche coûte moins pour certains agents que pour d'autres. En effet, lorsqu'un agent exécute une tâche, il doit récupérer l'ensemble de ses ressources. En particulier, la récupération des ressources distantes à l'agent (c'est-à-dire les ressources qui ne se trouvent pas sur son nœud de calcul) est un processus coûteux. Alors qu'une fonction de coût peut ne pas faire la différence entre une tâche coûteuse et locale et une tâche moins coûteuse et distante (cf. exemple 6.2), la **stratégie de sélection de tâche localisée** permet, contrairement aux stratégies précédentes, de privilégier l'exécution de tâches locales et la délégation de tâches distantes.

Exemple 6.2 (Fonction de coût et localité des tâches).
Considérons de nouveau l'instance $MASTA = (S, \mathcal{T}, P, c)$ issue de l'exemple 4.1. Pour rappel, celle-ci est constituée d'un système distribué $S = (\mathcal{N}, \mathcal{A}, \mathcal{E}, \mathcal{R}, l, d)$ qui contient deux nœuds de calculs aux performances homogènes ($\mathcal{N} = \{\nu_1, \nu_2\}$), deux agents ($\mathcal{A} = \{1, 2\}$) capables de communiquer ($\mathcal{E} = \{(1, 2)\}$), chacun situé sur un nœud de calculs ($l_1 = \nu_1$, $l_2 = \nu_2$) et un ensemble de sept tâches $\mathcal{T} = \{\tau_1, \ldots, \tau_7\}$. La fonction de coût c est également rappelée dans le tableau 6.2. On ajoute la définition de la fonction de répartition des ressources d dans le tableau 6.3 avec $d_i(\tau)$ le nombre de ressources nécessaires à l'exécution de la tâche τ qui sont locales pour l'agent i.

Dans l'exemple 4.1, la fonction de coût est définie par extension. Maintenant que la fonction de répartition des ressources est connue, il est possible d'exprimer le coût d'une tâche pour un agent en fonction du nombre de ressources qui lui sont locales. En effet, la fonction de coût de cette instance MASTA est définie telle que le coût d'une tâche pour un agent vaut le nombre de ressources locales plus deux fois le nombre de ressources distantes. Formellement,

$$c_i(\tau) = d_i(\tau) + 2\Big(|\mathcal{R}_\tau| - d_i(\tau)\Big) = 2|\mathcal{R}_\tau| - d_i(\tau)$$

Si l'on considère l'allocation courante P dans laquelle les tâches τ_2 et τ_6 sont allouées à l'agent 1, on observe qu'elles sont de même coût ($c_1(\tau_2) = c_1(\tau_6) = 8$). Cependant, τ_6 est plus locale pour l'agent 1 que τ_2. Comme son but est de faire décroître le makespan de P, l'agent 1 doit essayer de déléguer τ_2 avant τ_6 même si les deux tâches sont de même coût.

1. Je présente des jeux de données pour lesquels les lots de tâches des agents peuvent contenir jusque 10^5 tâches dans le chapitre 9.

En effet, τ_2 pourrait être plus locale pour d'autres agents du système et donc coûter moins pour eux (cf. exemple 6.3).

À l'inverse, la délégation de la tâche τ_6 ne doit être envisagée qu'après la délégation de τ_2. Comme τ_6 est presque entièrement locale pour l'agent 1, elle ne peut pas coûter moins pour un autre agent du système et sa délégation est moins pertinente pour améliorer le makespan de P.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$c_1(\tau_k)$	1	8	15	30	9	8	14
$c_2(\tau_k)$	2	4	12	24	6	13	7

TABLEAU 6.2 : Fonction de coût de l'instance MASTA de l'exemple 6.2.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$d_1(\tau_k)$	1	0	3	6	1	6	0
$d_2(\tau_k)$	0	4	6	12	4	1	7

TABLEAU 6.3 : Fonction de répartition des ressources de l'instance MASTA de l'exemple 6.2.

Pour intégrer la notion de localité dans leurs choix de tâches, les agents calculent le **ratio de localité** d'une tâche.

Définition 6.3 (Ratio de localité).
Le ratio de localité d'une tâche $\tau \in \mathcal{T}$ pour l'agent $i \in \mathcal{A}$ est le rapport entre le nombre de ressources de τ qui sont locales pour i et le nombre total des ressources de τ :

$$o_i(\tau) = \frac{d_i(\tau)}{|\mathcal{R}_\tau|} \tag{6.10}$$

Le ratio de localité maximum de τ est

$$\hat{o}(\tau) = \max_{i \in \mathcal{A}}(o_i(\tau)) \tag{6.11}$$

Exemple 6.3 (Ratio de localité).
Dans l'exemple 6.2, il est dit que la tâche τ_6 est plus locale pour l'agent 1 que la tâche τ_2. Ceci est confirmé par les ratios de localité des tâches τ_2 et τ_6 pour l'agent 1 :

- $o_1(\tau_2) = \frac{0}{4} = 0$;
- $o_1(\tau_6) = \frac{6}{7} \approx 0,86$.

Comme $o_1(\tau_6) > o_1(\tau_2)$, τ_6 est plus locale pour l'agent 1 que τ_2.

En utilisant le ratio de localité et le coût des tâches, la stratégie de sélection de tâche localisée est construite sur un lot de tâches complexe que l'on appelle le **lot de tâches localisées**. Ce lot de tâches est nécessaire pour mettre en place la mécanique d'exécution/délégation de la stratégie de sélection de tâche localisée. Cette stratégie et ce lot de tâches sont conçus de sorte qu'un agent exécute d'abord les tâches qui ne peuvent pas coûter moins pour d'autres et délègue d'abord les tâches qui sont coûteuses pour lui mais qui peuvent l'être moins pour d'autres.

Le lot de tâches localisées de l'agent i (cf. figure 6.1) est divisé en trois sous-ensembles (que l'on appellera également des lots) et les tâches allouées à l'agent appartiennent à l'un de ces trois sous-ensembles en fonction de leur ratio de localité pour i.

- **Le lot de localité maximum (noté LM)** contient les tâches τ pour lesquelles (a) l'agent i possède au moins une ressource et (b) aucun autre agent ne possède plus de ressources pour τ que l'agent i, c'est-à-dire $o_i(\tau) \neq 0$ et $o_i(\tau) = \hat{o}(\tau)$. Dans ce lot, les tâches sont triées par ordre de coût décroissant (cf. lot de gauche de la figure 6.1).
- **Le lot de localité partielle (noté LP)** contient les tâches τ qui sont partiellement locales pour l'agent i, c'est-à-dire $0 < o_i(\tau) < \hat{o}(\tau)$. Dans ce lot, les tâches sont d'abord triées par ordre de ratio de localité décroissant puis par ordre de coût décroissant (cf. lot du milieu de la figure 6.1). Cet ordre leximax est justifié par l'ordre dans lequel les agents exécutent et délèguent les tâches, comme illustré dans l'exemple 6.4.
- **Le lot distant (noté LD)** contient les tâches τ qui sont distantes pour l'agent i, c'est-à-dire $o_i(\tau) = 0$. Les tâches y sont triées par ordre de coût croissant (cf. lot de droite de la figure 6.1). Les tâches de ce lot ne peuvent pas être plus coûteuses pour un autre agent du système que pour l'agent i.

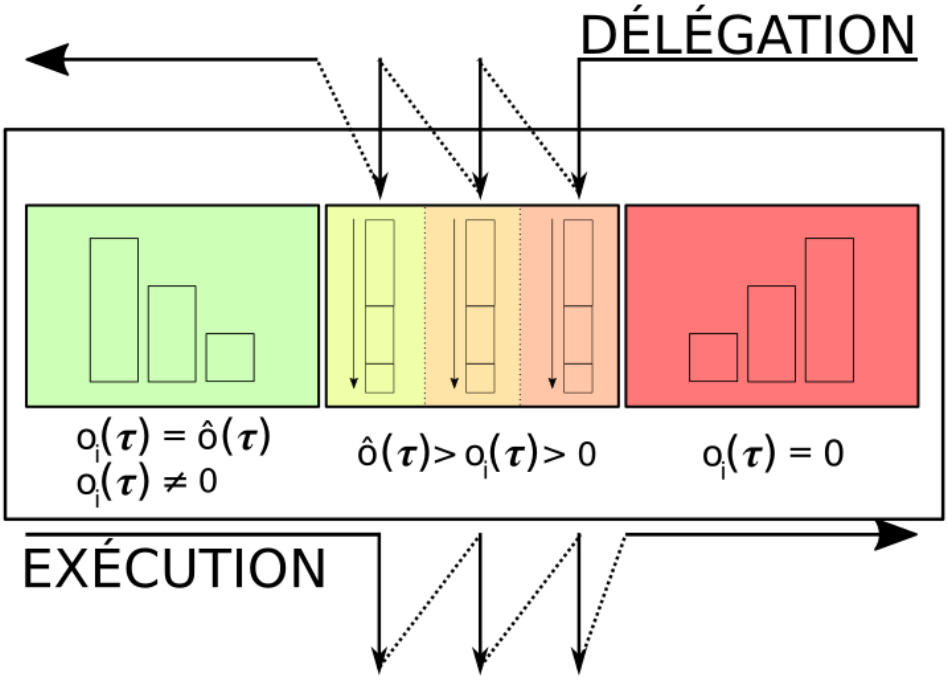


FIGURE 6.1 : Description du lot de tâches localisées d'un agent i. De gauche à droite, on trouve : le lot de localité maximum, le lot de localité partielle et le lot distant. Les rectangles à l'intérieur des lots représentent des tâches. La taille d'une tâche représente son coût, plus la tâche est grande, plus elle est coûteuse pour i. Les couleurs de fond des lots représentent la localité des tâches qu'ils contiennent : vert pour les tâches qui ont un ratio de localité maximum pour i, rouge pour les tâches distantes. Dans le lot de localité partielle, les tâches sont regroupées par ratio de localité décroissant d'abord (ici trois groupes pour les besoins de l'illustration), puis les tâches de même localité sont triées par ordre de coût décroissant. Les flèches indiquent le sens dans lequel l'agent i considère les tâches pour les opérations de délégation et d'exécution.

Durant l'exécution des tâches, les agents prélèvent les tâches depuis leur lot de tâches localisées par l'une ou l'autre de ses extrémités (cf. figure 6.1). Si l'agent cherche une tâche à exécuter, il explore son lot de tâches en partant du lot de tâches de localité maximum, et exécute la tâche la plus locale et la plus coûteuse d'abord. Si l'agent cherche une tâche à déléguer, il explore son lot de tâches en partant du lot distant, c'est-à-dire les tâches moins locales et les plus coûteuses, et

il choisit la première tâche qui peut mener à une délégation socialement rationnelle en fonction de ses croyances $\mathcal{B}_i(P)$. Pour résumer :

- Un agent i sélectionne une tâche à exécuter $\overset{>}{\tau} \in LM$ telle que :

$$\forall \tau \in LM \setminus \{\overset{>}{\tau}\}, c_i(\overset{>}{\tau}) \geq c_i(\tau) \tag{6.12}$$

  Si $LM = \emptyset$, l'agent i sélectionne $\overset{>}{\tau} \in LP$ telle que :

$$\forall \tau \in LP \setminus \{\overset{>}{\tau}\}, o_i(\overset{>}{\tau}) > o_i(\tau) \vee \Big(o_i(\overset{>}{\tau}) = o_i(\tau) \wedge c_i(\overset{>}{\tau}) \geq c_i(\tau)\Big) \tag{6.13}$$

  Finalement, si $LP = \emptyset$, l'agent i sélectionne $\overset{>}{\tau} \in LD$ telle que :

$$\forall \tau \in LD \setminus \{\overset{>}{\tau}\}, c_i(\overset{>}{\tau}) \leq c_i(\tau) \tag{6.14}$$

  Si $LM = LP = LD = \emptyset$, l'agent i n'a aucune tâche à exécuter. Le résultat de $perform$ est $\perp$.

- Un agent i sélectionne une tâche à déléguer $\overset{<}{\tau} \in LD$ telle que :

$$\forall \tau \in \Big(LD \cap \Gamma_i^{\mathcal{B}}(P)\Big) \setminus \{\overset{<}{\tau}\}, c_i(\overset{<}{\tau}) \geq c_i(\tau) \tag{6.15}$$

  Si $LD \cap \Gamma_i^{\mathcal{B}}(P) = \emptyset$, l'agent i sélectionne $\overset{<}{\tau} \in LP$ telle que :

$$\forall \tau \in \Big(LP \cap \Gamma_i^{\mathcal{B}}(P)\Big) \setminus \{\overset{<}{\tau}\}, o_i(\overset{<}{\tau}) < o_i(\tau) \vee \Big(o_i(\overset{<}{\tau}) = o_i(\tau) \wedge c_i(\overset{<}{\tau}) \geq c_i(\tau)\Big) \tag{6.16}$$

  Enfin, si $LD \cap \Gamma_i^{\mathcal{B}}(P) = \emptyset$, l'agent i sélectionne $\overset{<}{\tau} \in LM$ telle que :

$$\forall \tau \in \Big(LM \cap \Gamma_i^{\mathcal{B}}(P)\Big) \setminus \{\overset{<}{\tau}\}, c_i(\overset{<}{\tau}) \leq c_i(\tau) \tag{6.17}$$

  Si $LM \cap \Gamma_i^{\mathcal{B}}(P) = LP \cap \Gamma_i^{\mathcal{B}}(P) = LD \cap \Gamma_i^{\mathcal{B}}(P) = \emptyset$, le résultat de *delegate* est $\perp$.

Remarquons que l'ordre leximax du lot de localité partielle favorise le principe selon lequel un agent doit exécuter les tâches locales et coûteuses et déléguer les tâches distantes et coûteuses. Quand un agent parcourt son lot de localité partielle, il commence par les tâches avec un ratio de localité élevée (c'est-à-dire celles à exécuter) ou par les tâches avec un faible ratio de localité (c'est-à-dire celles à déléguer). Lorsque l'agent rencontre plusieurs tâches avec le même ratio de localité, il les parcourt toujours dans le même ordre : de la plus coûteuse à la moins coûteuse (cf.figure 6.1). Si l'agent utilise le lot de localité partielle pour trouver une tâche à exécuter, il exécute celle qui est la plus coûteuse et la plus locale. En revanche, si l'agent y cherche une tâche à déléguer, il délègue celle qui est la plus coûteuse mais la moins locale.

Exemple 6.4 (Ordre d'exécution et de délégation avec la stratégie de sélection de tâche localisée).
Considérons de nouveau l'instance MASTA décrite dans l'exemple 6.2, qui pour mémoire, possède la fonction de distribution des ressources présentée dans le tableau 6.4. On peut calculer le ratio de localité de chacune des tâches pour les deux agents à partir de d (cf. tableau 6.5).

Pour cet exemple, considérons l'allocation courante P telle que l'agent 1 possède toutes les tâches (c'est-à-dire $P_1 = \mathcal{T}$). Le lot de tâches localisées de l'agent 1 est illustré en figure 6.2. En appliquant la stratégie de sélection de tâches localisées, l'agent 1 :

- *envisage d'exécuter les tâches dans l'ordre suivant :* τ_6, τ_1, $\underline{\tau_4, \tau_3}$, τ_5, τ_2, τ_7 ;
- *envisage de déléguer les tâches dans l'ordre suivant :* τ_7, τ_2, τ_5, $\underline{\tau_4, \tau_3}$, τ_1, τ_6.

D'une part, il est préférable pour l'agent 1 d'exécuter τ_6*. Puisqu'il s'agit de l'agent pour lequel la tâche est la plus locale, elle coûte moins pour l'agent 1 que pour l'agent 2. D'autre part, il est préférable pour le système que l'agent 1 délègue* τ_7 *car elle lui est distante. Ainsi, il est certain que* τ_7 *n'est pas plus coûteuse pour un autre agent. Remarquons que les tâches dans le lot de localité partielle sont triées d'abord par ratio de localité, puis par coût. Que l'agent 1 souhaite déléguer ou exécuter une tâche, il considère les tâches qui ont le même ratio de localité dans le même ordre. Ici,* τ_4 *puis* τ_3*.*

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$d_1(\tau_k)$	1	0	3	6	1	6	0
$d_2(\tau_k)$	0	4	6	12	4	1	7

TABLEAU 6.4 : Fonction de répartition des ressources de l'instance MASTA de l'exemple 6.4.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
$o_1(\tau_k)$	1	0	0,33	0,33	0,2	0,86	0
$o_2(\tau_k)$	0	1	0,66	0,66	0,8	0,14	1

TABLEAU 6.5 : Ratio de localité de chaque tâches pour les agents de l'exemple 6.4.

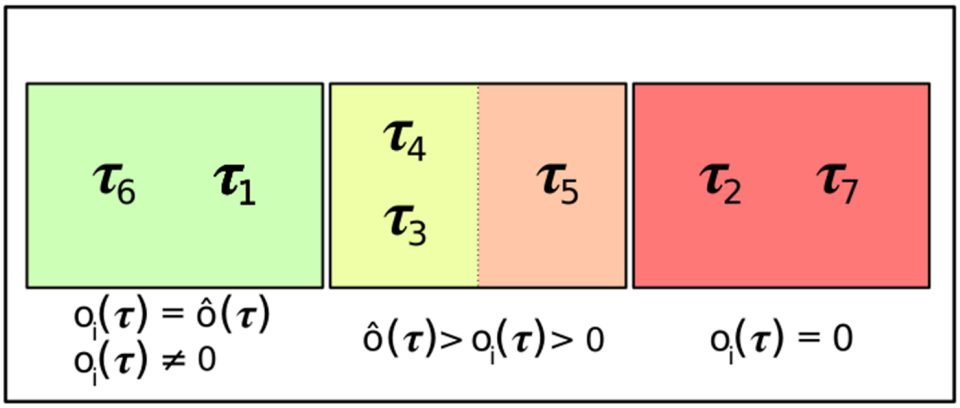


FIGURE 6.2 : Lot de tâches localisées de l'agent 1 dans l'exemple 6.4.

La stratégie de sélection de tâches localisées permet d'affiner le choix de la prochaine tâche à exécuter/déléguer. Comme nous l'avons vu dans l'exemple 6.2 une fonction de coût peut ne pas être suffisante pour distinguer le bénéfice potentiel de la consommation/délégation d'une tâche. Grâce à la stratégie présentée dans cette section et à son lot de tâche associé, les agents négocient de manière plus fine en distinguant le coût estimé d'une tâche de sa localité.

6.3 Stratégie de découpe de tâches

Une allocation de tâches stable (c'est-à-dire une allocation où aucune délégation de tâche socialement rationnelle n'est possible) ne correspond pas nécessairement à une charge de travail équilibrée. Dans le cas où certaines tâches sont significativement plus coûteuses que d'autres, l'agent i qui possède une de ces tâches coûteuses peut être surchargé sans avoir la possibi-

lité d'initier une délégation socialement rationnelle ($\Gamma_i^{\mathcal{B}}(P) = \emptyset$). Si les tâches sont divisibles [2], l'agent i peut décider de découper l'une de ses tâches afin de déléguer les sous-tâches résultantes. Développer une heuristique de découpe de tâche efficace dans le cadre général de cette thèse est un problème compliqué car :

- le coût des sous-tâches générées diffère selon les agents du système ;
- considérer l'ensemble des découpes possibles pour l'ensemble des pairs représente une combinatoire importante qui risque de pénaliser l'agent qui effectue la découpe.

Pour ces raisons, j'envisage dans cette section la découpe de tâche pour un sous-ensemble des problèmes MASTA. Je suppose ici que la localité des ressources n'influence pas le coût d'une tâche. Cela implique que le coût d'une tâche est le même pour tous les agents du système :

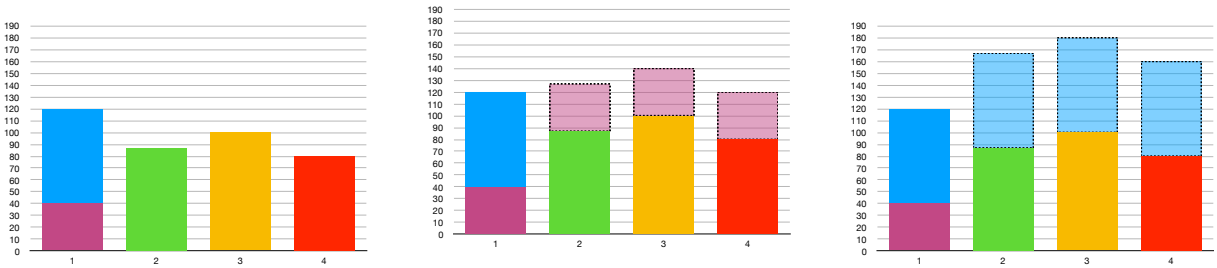
$$\forall \tau \in \mathcal{T}, \forall i, j \in \mathcal{A}, c_i(\tau) = c_j(\tau).$$

On note donc c_τ le coût unique de la tâche τ.

6.3.1 Amorce et principe

Les agents envisagent la découpe de tâche lorsque l'allocation courante est stable mais non équilibrée.

> **Exemple 6.5 (Allocation stable non équilibrée).**
> *Considérons $\mathcal{A} = \{1, 2, 3, 4\}$ un ensemble de quatre agents et P l'allocation courante telle que $w_1(P) = 120$, $w_2(P) = 87$, $w_3(P) = 100$, $w_4(P) = 80$. On remarque que l'agent 1 est l'agent le plus chargé et qu'il possède deux tâches (cf. figure 6.3a) mais qu'aucune d'entre elles n'est délégable (cf. figure 6.3b et 6.3c).*



(a) L'agent 1 est l'agent le plus chargé et possède deux tâches.

(b) La délégation de la première tâche de l'agent 1 n'est pas socialement rationnelle.

(c) La délégation de la seconde tâche de l'agent 1 n'est pas socialement rationnelle.

FIGURE 6.3 : Allocation P issue de l'exemple 6.5.

Pour que la découpe d'une tâche en plusieurs sous-tâches soit possible, il est nécessaire (a) que la notion de sous-tâche existe, on dit alors que les tâches sont **divisibles** (b) de disposer d'une fonction d'aggrégation $\bigoplus$ qui permet de retrouver sans surcoût le résultat de la tâche initiale à partir des résultats des sous-tâches et (c) de disposer d'une fonction de coût c qui est en mesure de donner le coût des sous-tâches. On appelle le résultat d'une sous-tâche un **résultat intermédiaire**.

2. Dans le chapitre 8, nous verrons quelles sont les conditions pour pouvoir découper une tâche de *reduce*.

Définition 6.4 (Découpe de tâche).
Soit $\tau \in \mathcal{T}$ une tâche de résultat $res_\tau \in Res$. La découpe de τ crée n sous-tâches $\mu_1^\tau, \ldots, \mu_n^\tau$ telles que :

$$\sum_{k\in[1;n]} c_{\mu_k^\tau} = c_\tau \tag{6.18}$$

$$\bigcup_{k\in[1;n]} \mathcal{R}_{\mu_k^\tau} = \mathcal{R}_\tau \tag{6.19}$$

$$\bigcap_{k\in[1;n]} \mathcal{R}_{\mu_k^\tau} = \emptyset \tag{6.20}$$

$$\bigoplus_{k\in[1;n]} res_{\mu_k^\tau} = res_\tau \tag{6.21}$$

Ainsi, une tâche est découpée de sorte que :

- la somme du coût des sous-tâches soit égale au coût de la tâche initiale (cf. équation 6.18) ;
- chaque ressource nécessaire à l'exécution de la tâche initiale soit utilisée pour l'exécution d'une et une seule sous-tâche (cf. équations 6.19 et 6.20). On remarque que si un agent crée n sous-tâches à partir de la tâche τ, alors la valeur maximale de n est $|\mathcal{R}_\tau|$;
- le résultat de la tâche initiale est calculé en agrégeant l'ensemble des résultats intermédiaires (cf. équation 6.21).

Les découpes de tâches permettent donc aux agents qui possèdent des tâches trop coûteuses pour être déléguées de faire décroître leur charge de travail. Comme le mécanisme de découpe demande des interactions supplémentaires pour réallouer les sous-tâches, une découpe n'est réalisée que sous certaines conditions.

Définition 6.5 (Condition de découpe de tâche).
Pour que l'agent $i \in \mathcal{A}$ puisse découper sa tâche la plus coûteuse $\tau \in P_i$, il faut que :

$$\exists j \in \mathcal{A}, w_j^i(P) < w_i(P) \tag{6.22}$$

$$\Gamma_i^{\mathcal{B}}(P) = \emptyset \tag{6.23}$$

$$|\mathcal{R}_\tau| > 2 \tag{6.24}$$

Ainsi, pour qu'un agent i puisse envisager de découper sa tâche la plus coûteuse, il faut que :

1. l'agent i ne se croit pas l'agent le moins chargé (cf. équation 6.22) ;
2. l'agent i soit en état de pause, c'est-à-dire que, selon ses croyances, il ne peut pas déléguer de tâche (cf. équation 6.23) ;
3. la tâche la plus coûteuse de l'agent i soit composée d'au moins deux ressources (cf. équation 6.24). Une découpe de celle-ci peut ainsi donner au moins deux sous-tâches, chacune liée à une ressource.

Lors d'une découpe de tâche, l'objectif de l'agent i est de diminuer sa charge de travail. Si m' agents sont moins chargés que i, il découpe sa tâche la plus coûteuse en $k+1$ sous-tâches avec $1 \leq k \leq m'$. L'objectif pour l'agent i est ensuite de déléguer k de ces sous-tâches pour équilibrer sa charge avec celles de ses pairs. La répartition des k sous-tâches est envisagée avec les m'

pairs moins chargés que i pour qu'à l'issue des futures négociations, aucun d'eux ne soit plus chargé que i avant ces négociations. Autrement dit, pour que les sous-tâches soient déléguées de manière socialement rationnelle. k des sous-tâches sont également créées de sorte à contenir le même nombre de ressources et donc à avoir approximativement le même coût, facilitant ainsi leur future délégation. Ces sous-tâches peuvent également être découpées.

6.3.2 Heuristique de découpe de tâche

L'heuristique de découpe de tâches repose sur la différence entre la charge de travail de l'initiateur et les charges de ses pairs.

Définition 6.6 (Delta de charge).
On note $\overrightarrow{\mathcal{B}_i(P)} = \langle w^i_{j_1}(P), \ldots, w^i_{j_{m-1}}(P)\rangle$ *les croyances de l'agent* i *sur les charges de ses* $m-1$ *pairs triés par ordre croissant. Pour tout agent* $j_k \in \mathcal{A}\setminus\{i\}$ *(c'est-à-dire l'agent avec la* k*-ième charge la moins élevée selon* i*), le delta de charge est défini comme :*

$$\Delta^k_i = w_i(P) - w^i_{j_k}(P) \tag{6.25}$$

D'après les conditions de découpe d'une tâche τ, l'agent i est en état de pause et il existe m' agents moins chargés que lui. L'agent i ne peut initier aucune délégation de tâche. En particulier, sa tâche la plus coûteuse τ ne peut pas être déléguée ($\forall k \in [1; m'], c_\tau > \Delta^k_i$). Ainsi la découpe de la tâche τ a pour objectif de déléguer k sous-tâches de même coût. Cette délégation doit réduire au plus possible la charge de l'agent i, ce qui équivaut à répartir au mieux la charge de travail.

L'agent i calcule k, le nombre de sous-tâches à déléguer à partir de τ, de la manière suivante :

$$k = \underset{k\in[1;m']}{\operatorname{argmin}}(w_i(P) - \frac{k\Delta^k_i}{k+1}) \tag{6.26}$$

Ce qui amène à la création de $k+1$ sous-tâches $\mu^\tau_1, \ldots, \mu^\tau_{k+1}$ telles que :

- $c_{\mu^\tau_1} = \ldots = c_{\mu^\tau_k} = \frac{\Delta^k_i}{k+1}$;
- $c_{\mu^\tau_{k+1}} = c_\tau - \frac{k\Delta^k_i}{k+1}$.

L'exemple suivant illustre comment l'indice k est choisi et l'impact qu'a ce choix sur les charges des agents après délégations.

Exemple 6.6 (Découpe de tâche).
Dans la situation décrite par l'exemple 6.5, l'agent 1 possède deux tâches τ *et* τ' *telles que* $c_\tau = 80$ *et* $c_{\tau'} = 40$*. Il ne peut déléguer aucune de ses tâches, il est donc en état de pause. De plus on constate qu'il existe* $m' = 3$ *agents moins chargés que l'agent 1 dans le système (cf. figure 6.4). On suppose que les croyances de l'agent 1 sont exactes.*

Si l'on considère la découpe de la tâche τ *par l'agent 1, on observe :*

- $\overrightarrow{\mathcal{B}_1(P)} = \langle 80, 87, 100\rangle$;
- $\Delta^1_1 = w_1(P) - w^1_4(P) = 120 - 80 = 40$;
- $\Delta^2_1 = w_1(P) - w^1_2(P) = 120 - 87 = 33$;

- $\Delta_1^3 = w_1(P) - w_3^1(P) = 120 - 100 = 20$.

Le nombre de sous-tâches et le nombre de potentiels enchérisseurs pris en compte influencent les charges de travail qui résultent de la découpe de τ. Dans cet exemple, ce n'est pas en fixant $k = m'$ que la charge de travail de l'agent 1 peut décroître le plus.

Si l'agent 1 partage τ avec un seul autre agent ($k = 1$), il crée les sous-tâches afin d'équilibrer sa charge de travail avec celle de l'agent le moins chargé. Ainsi, la meilleure découpe pour équilibrer la charge de l'agent 1 et celle de l'agent 4 au terme d'une enchère consiste à répartir équitablement Δ_1^1, la surcharge que représente τ, entre les deux agents. Par conséquent, les sous-tâches μ_1^τ et μ_2^τ sont créées à partir de τ telles que :

- $c_{\mu_1^\tau} = \frac{\Delta_1^1}{2} = 20$;
- $c_{\mu_2^\tau} = c_\tau - c_{\mu_1^\tau} = c_\tau - \frac{\Delta_1^1}{2} = 60$.

De cette manière, la nouvelle allocation P' qui résulte de la délégation de la tâche μ_1^τ répartit les charges telles que $w_1(P') = w_4(P') = 100$ (cf. figure 6.4a).

En procédant au même raisonnement, les situations pour $k = 2$ et $k = 3$ donnent de nouvelles charges de travail différentes :

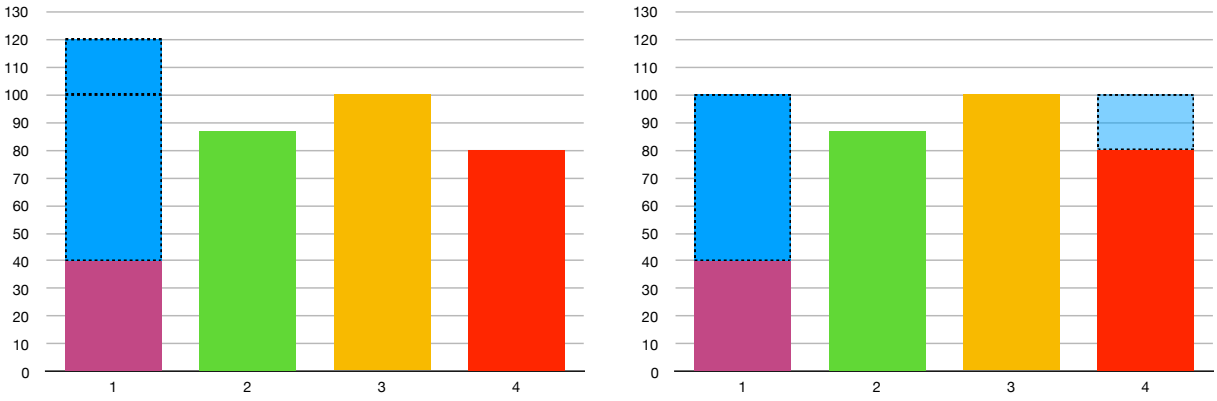
- *pour $k = 2$, $w_1(P') = w_2(P') = 98$ (cf. figure 6.4b) ;*
- *pour $k = 3$, $w_1(P') = w_3(P') = 105$ (cf. figure 6.4c).*

Plus généralement, après k enchères, l'agent 1 délègue k sous-tâches de coût $\frac{\Delta_1^k}{k+1}$ et sa nouvelle charge de travail vaut $w_1(P') = w_1(P) - \frac{k\Delta_1^k}{k+1}$. Comme on peut le constater, il existe une valeur de k (ici $k = 2$) qui minimise $w_1(P')$:

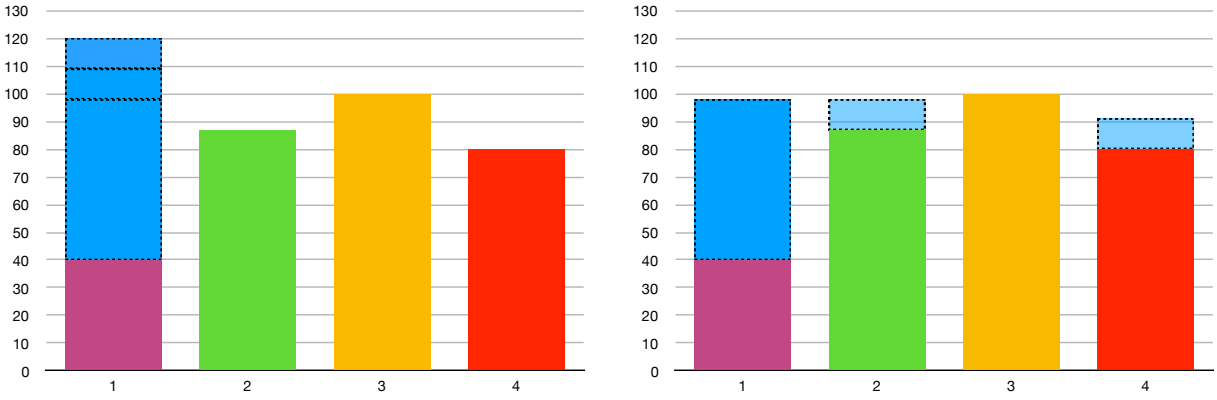
$$k = \underset{k \in [1;3]}{\operatorname{argmin}}(w_1(P) - \frac{k\Delta_1^k}{k+1})$$

Comme l'illustre l'exemple 6.6, un agent découpe une tâche en anticipant lesquels de ses pairs sont susceptibles de prendre en charge les sous-tâches après une délégation socialement rationnelle. Découper une tâche spécifiquement pour un sous-ensemble d'agents semble délicat en tenant compte des croyances éventuellement inexactes de l'initiateur. Cependant, les sous-tâches sont négociées de la même manière que les tâches non découpées, c'est-à-dire que les `cfp` sont envoyés à tous les pairs de l'initiateur et pas seulement aux agents dont la charge a été utilisée pour le calcul de la découpe. La découpe de tâche offre donc de nouvelles opportunités pour améliorer le *makespan* de l'allocation courante et l'opération de délégation socialement rationnelle reste une garantie qu'aucune tâche ou sous-tâche ne sera déléguée sans que cela soit globalement bénéfique au système.

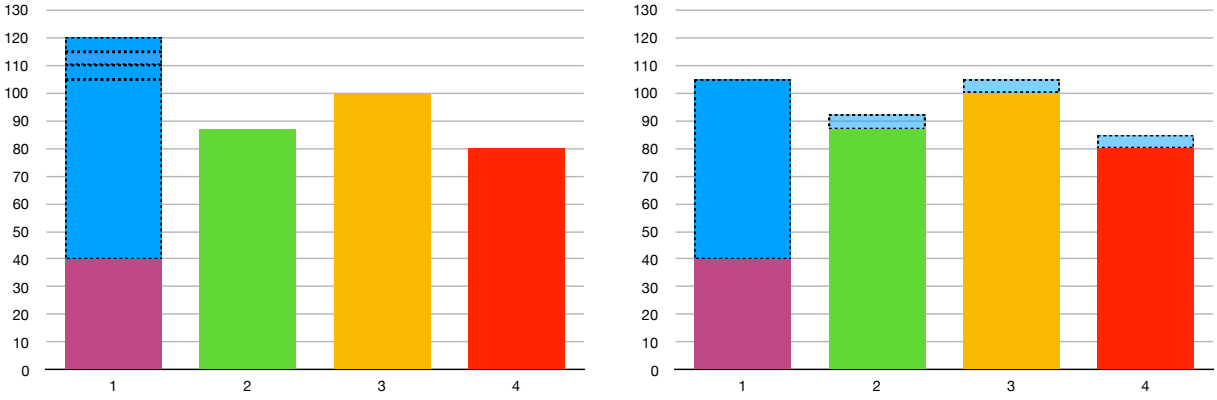
Cette heuristique ne tient pas compte des contraintes de découpe 6.19 et 6.20. Elle demande donc à être adaptée pour être utilisée dans une application pratique où la nature des tâches et des ressources est connue. Cela sera présenté dans le chapitre 8 pour la découpe d'une tâche de *reduce* lors de l'exécution d'un job MapReduce.



(a) Découpe de la tâche τ pour $k = 1$.



(b) Découpe de la tâche τ pour $k = 2$.



(c) Découpe de la tâche τ pour $k = 3$.

FIGURE 6.4 : Illustration de l'impact de la valeur de k dans l'heuristique de découpe de tâche effectuée dans l'exemple 6.6.

6.4 Conclusion

6.4.1 Synthèse

Dans le cadre du formalisme MASTA décrit dans le chapitre 4, les agents font évoluer l'allocation courante en utilisant deux opérations : les délégations de tâches socialement rationnelles et les consommations (ou exécutions) de tâches. Ces deux opérations améliorent le *makespan* de l'allocation courante (cf. propriétés 4.1 et 4.3). Afin d'optimiser l'ordre dans lequel ils délèguent/consomment leurs tâches, les agents ont recours à une stratégie de sélection de tâches. Ce chapitre décrit plusieurs types de stratégies de sélection de tâches.

Les premières stratégies présentées dans ce chapitre sont les stratégies de sélection par coût ordonnés. Elles reposent sur le tri des tâches par le seul ordre de coût croissant. De ce tri découle naturellement deux variantes : déléguer les tâches les plus coûteuses et exécuter les tâches les moins coûteuse d'abord, ou faire l'inverse. Ces stratégies sont les moins sophistiquées mais présentent l'avantage d'être faciles à mettre en place et à utiliser tout au long de l'exécution des tâches. Les stratégies de sélection de tâches k-éligibles affinent le choix de la tâche à déléguer en privilégiant celles qui, selon les croyances de l'initiateur, seraient acceptées par au moins `k` de ses pairs comme une délégation socialement rationnelle. Enfin, la stratégie de sélection de tâches localisées prend en compte le fait que choisir la tâche à déléguer/exécuter uniquement par rapport à son coût peut manquer de finesse (cf. exemple 6.2), et ajoute la localité des ressources comme critère de sélection. Ainsi, les agents exécutent en priorité les tâches locales et coûteuses et délèguent les tâches distantes et coûteuses. Dans le cadre applicatif de ces travaux, des agents *reducer* utilisent ces stratégies de sélection de tâche pour réallouer dynamiquement les tâches de *reduce* pendant l'exécution d'un job MapReduce. Ainsi, j'évalue l'efficacité de ces stratégies dans le chapitre 9.

Ce chapitre présente également une stratégie de découpe de tâche. Une allocation de tâche stable n'est pas nécessairement parfaitement équilibrée. Si les tâches sont divisibles, un agent surchargé mais dont aucune des tâches n'est délégable peut envisager de découper sa tâche la plus coûteuse en sous-tâches délégables afin d'améliorer encore le *makespan* de l'allocation courante. J'ai présenté ici les hypothèses nécessaires à la découpe de tâches ainsi qu'une heuristique de découpe. Dans le chapitre 8, je détaille le processus effectif de découpe des tâches de *reduce* dans le cadre de l'exécution d'un job MapReduce. Cette découpe de tâches permet de contrer le biais des clés coûteuses (cf. chapitre 3).

6.4.2 Perspectives

La mise en place de stratégies pour les différentes prises de décisions locales des agents est un point clé pour l'amélioration du processus de résolution d'un problème MASTA. Trouver de nouvelles stratégies de sélection de tâches (ou raffiner celles présentées dans ce chapitre) selon les contraintes du système reste donc une perspective.

Dans ce chapitre, nous avons étudié des stratégies utilisées par les initiateurs des délégations de tâches. Une seconde perspective est d'envisager des stratégies pour les enchérisseurs, c'est-à-dire pour les agents qui sont susceptibles de recevoir une tâche via une délégation socialement rationnelle. En effet, on peut se questionner sur l'intérêt d'une délégation si elle n'apporte qu'un

gain minime de *makespan*. On pourrait donc munir les agents d'une stratégie d'acceptabilité pour leur permettre de ne pas forcément faire une proposition dès l'instant qu'un `cfp` offre une délégation de tâche socialement rationnelle. Par exemple, un enchérisseur pourrait répondre à un `cfp` par une proposition uniquement si le gain de *makespan* local est supérieur à 5 %.

Enfin, la stratégie de découpe de tâche repose pour l'instant sur des hypothèses qui ne tiennent pas compte de l'ensemble des caractéristiques d'une instance MASTA. Une troisième perspective est donc d'étendre l'heuristique de découpe de tâche pour y intégrer le fait que les tâches peuvent avoir un coût différent pour tous les agents du système ou encore pour prendre en compte la localité des ressources.

Troisième partie

Mise en œuvre

Chapitre 7

Architecture composite d'agent

Sommaire

7.1 Introduction

La résolution d'un problème MASTA nécessite la mise en place distribuée d'agents autonomes. Alors que ces derniers consomment les tâches en continu, ils délèguent également certaines d'entre elles pour faire décroître le *makespan* de l'allocation courante.

Inspiré du modèle d'acteurs de [Hewitt 1977], je considère qu'un agent :

1. possède une adresse unique pour communiquer avec ses pairs ;
2. est réactif aux messages présents dans sa boite de réception ;
3. possède un état courant ;
4. peut créer d'autres agents locaux.

Pour répondre au second point, j'utilise le modèle de transmission asynchrone de messages pour la programmation concurrente [Clinger 1981]. En d'autres termes, je suppose que : (a) le délai de transmission des messages est arbitraire mais non négligeable, (b) l'ordre d'émission/réception des messages est identique par pair émetteur-récepteur et (c) les messages peuvent être perdus. Dans un tel système, les messages sont livrés 0 ou 1 fois. C'est la raison pour laquelle un mécanisme d'interruption temporelle est intégré au protocole d'interaction présenté dans le chapitre 5.

Un agent possède donc une file d'attente qui contient l'ensemble des messages qu'il a reçus et qu'il doit traiter. Pour déterminer sa prochaine action, un agent retire le premier message de sa file d'attente et réagit en fonction de son état courant. Les actions déclenchées par ce message peuvent produire un changement d'état courant. Ainsi, en fonction de son état, un agent peut réagir différemment au même message. Enfin, le quatrième point permet de concevoir une structure composite d'agent, c'est-à-dire qu'un agent est composé de plusieurs agents composants.

Ce chapitre explique comment une architecture composite d'agent permet d'isoler les différentes activités d'un agent d'une instance MASTA (appelé agent MASTA) pour ainsi faciliter sa conception, l'intelligibilité de son comportement et sa mise en œuvre. La section 7.2 commence par présenter les éléments constitutifs d'un agent et notamment comment le comportement des agents est spécifié et implémenté. Ensuite, la section 7.3 présente l'organisation interne d'un agent, c'est-à-dire qu'elle décrit quels sont les agents composants, quels sont leurs rôles et comment ils interagissent. Puis, la section 7.4 décrit les différents états d'un agent et de ses agents composants. Enfin, la section 7.5 synthétise le chapitre.

7.2 Comportement

Pour améliorer une instance MASTA, les agents communiquent avec leurs pairs pendant qu'ils consomment les tâches. Comme nous l'avons vu dans le chapitre 5, les agents communiquent uniquement par envoi de messages et quasi exclusivement de manière asynchrone. Les agents sont réactifs, c'est-à-dire qu'ils ne sont déclenchés qu'en réaction à la réception d'un message. Pour ce faire, un agent possède plusieurs attributs :

- son **adresse**, c'est-à-dire un identifiant unique qui permet de distinguer l'agent, de lui envoyer des messages et de l'identifier comme émetteur d'un message ;
- la **liste de ses accointances**, c'est-à-dire la liste des pairs avec lesquels l'agent est capable de communiquer directement. Comme indiqué dans le chapitre 1, on suppose que le système distribué est complet et donc que tous les agents peuvent directement communiquer les uns avec les autres ;
- sa **file de messages** qui stocke les messages que l'agent reçoit et qu'il n'a pas encore traités. La file de messages d'un agent est une structure de type « premier entré, premier sorti », l'agent traite donc les messages dans leur ordre de réception ;
- son **état courant** qui sélectionne le comportement à appliquer ;
- son **comportement** qui indique comment l'agent réagit lorsqu'il reçoit un message donné dans un état donné.

Dans un premier temps, je décris comment le comportement d'un agent est spécifié par un automate fini déterministe. Puis, je montre comment ces spécifications permettent une implémentation directe grâce à la programmation réactive.

7.2.1 Spécification

Le comportement d'un agent est spécifié à l'aide d'automates finis déterministes (cf. figure 7.1). Le comportement d'un agent dépend de son état courant et du premier message de sa file. En fonction de ces deux paramètres, l'agent exerce une action et/ou change d'état. Même si tous

les agents utilisent le même comportement, chaque agent est indépendant, possède son propre état courant, ses propres croyances et sa propre file de messages. L'exécution indépendante, concurrente et asynchrone de ce comportement permet aux agents d'améliorer dynamiquement une allocation de tâches de manière décentralisée.

L'exécution de tâche mise à part, un agent peut entreprendre deux autres types d'actions : l'envoi de messages (que l'on note grâce à l'opérateur `!`) ou la modification d'une structure de données interne (lot de tâches, base de croyances, etc.). Un agent peut envoyer un message à l'un de ses pairs mais également à lui-même. En s'envoyant un message, un agent est en mesure d'auto-activer son propre comportement.

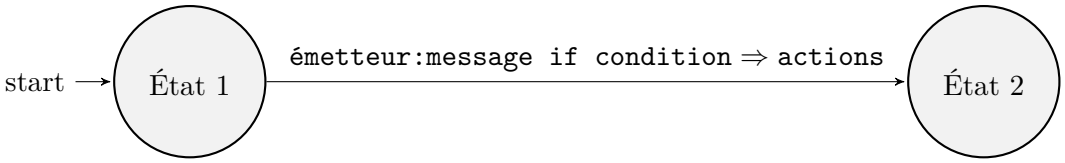


FIGURE 7.1 : Automate fini déterministe spécifiant le comportement d'un agent qui, lorsqu'il reçoit un `message` de `émetteur` dans l'`état 1` et qu'une `condition` est remplie, effectue des `actions` et passe dans l'`état 2`.

Exemple 7.1 (Spécification d'un comportement avec une machine à états finis). *L'automate de la figure 7.2 spécifie le comportement de l'agent Bruce lors de ses interactions avec un agent* `téméraire`*. L'agent Bruce possède une variable interne* `patience` *qui détermine son niveau de patience. L'état initial de l'agent Bruce est l'état* Banner*. Tant qu'il lui reste de la* `patience`*, l'agent Bruce accepte de recevoir le message* `pichenette` *de la part de l'agent* `téméraire` *mais perd progressivement* `patience`*. En revanche, lorsque sa* `patience` *est nulle et qu'il reçoit une* `pichenette`*, l'agent Bruce passe dans l'état* Hulk *et envoie un message* `avertissement` *à l'agent* `téméraire`*. Dans l'état* Hulk*, l'agent Bruce répond systématiquement à une* `pichenette` *par un* `HulkSmash`*. Après avoir délivré un* `HulkSmash`*, l'agent Bruce s'envoie un message* `calme` *grâce auquel, lorsqu'il le traite, il réinitialise sa variable* `patience` *et repasse dans l'état* Banner*.*

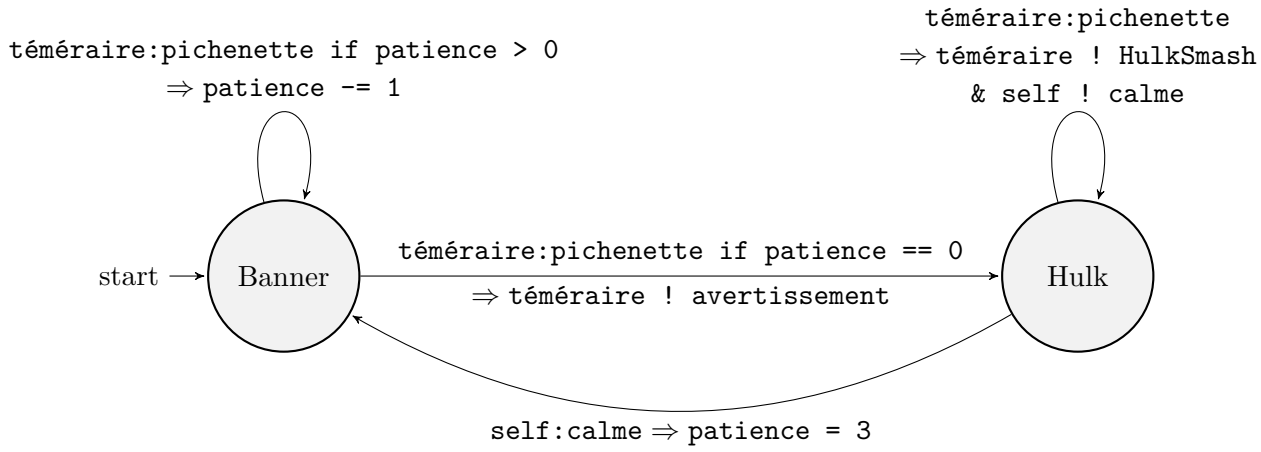


FIGURE 7.2 : Spécification du comportement de l'agent *Bruce* issu de l'exemple 7.1.

Les automates présentés en annexe A permettent une spécification complète et formelle du comportement d'un agent MASTA. Ce sont ces automates qui ont aidé à l'implémentation des comportements effectifs des agents MASTA. On peut, à leur lecture, se faire une idée de la complexité de ces comportements. La section suivante discute de la relation directe entre la spécification d'un comportement par un automate fini déterministe et l'implémentation de ce comportement grâce à la programmation réactive.

7.2.2 Implémentation

Comme nous le verrons dans le chapitre suivant, le prototype distribué MAS4Data utilise des agents MASTA pour la réallocation de tâches pendant leur exécution. MAS4Data est implémenté grâce à la boîte à outils Akka [Akka 2019] qui permet la construction d'applications réactives, concurrentes et distribuées.

Pour illustrer la relation directe entre la spécification d'un comportement par un automate fini détermiste et son implémentation en Akka, l'extrait de code 7.3 présente l'implémentation du comportement de l'agent *Bruce* de l'exemple 7.1. On peut y constater que l'automate qui spécifie le comportement de l'agent (cf. figure 7.2) se traduit naturellement et directement en Akka. Chaque état est traduit par une méthode dans laquelle sont décrites les actions qu'entreprend l'agent pour chaque message qu'il est susceptible de recevoir. Notons l'utilisation de l'opérateur `!` pour l'envoi de message (lignes 26 et 37), de la méthode `become` pour le changement d'état (ligne 28), et de la valeur `sender` qui contient l'expéditeur du message en cours de traitement (lignes 26 et 37).

En Akka, une sous-classe de `Actor` correspond à la définition d'un agent et de son comportement. Les acteurs Akka sont instanciés au sein d'un `ActorSystem`. Cette structure correspond à un système multi-agents dans lequel les agents partagent une configuration materielle commune. Il est également possible de déployer plusieurs `ActorSystem` distants (par exemple un par nœud de calcul) et de les lier de sorte que les agents des différents environnements puissent communiquer entre eux.

Comme le montrent les automates de l'annexe A, le comportement d'un agent MASTA est complexe et nécessite plusieurs automates pour être complétement spécifié. Pour aider à la compréhension de ces automates, les sections suivantes présentent en détail l'organisation interne d'un agent composite, les différentes interactions des agents composants ainsi que la sémantique de leurs différents états.

7.3 Organisation interne d'un agent

Du point de vue du système et de ses pairs, un agent est une entité atomique. Cependant, du point de vue génie logiciel, je propose une architecture composite d'agent pour une séparation des préoccupations. Afin de réduire la complexité liée à la conception d'un agent capable d'effectuer à la fois des délégations et des consommations de tâches, j'adopte une approche modulaire. Dans les chapitres précédents était appelé agent ce qui est en fait **un agent composite**, c'est-à-dire une entité qui regroupe les trois agents composants (cf. figure 7.4) décrits dans la suite de cette section. Ainsi, dans le formalisme MASTA, les agents $i \in \mathcal{A}$ sont des agents composites, tous

```
// Definition des messages
object Pichenette
object Avertissement
object HulkSmash
object Calme

// Definition de l'agent Bruce
class Bruce extends Actor {

  // Niveau de patience de l'agent
  var patience: Int = 3

  // Etat initial de l'agent
  def receive: Receive = this.banner

  // Etat Banner
  def banner: Receive = {

    case Pichenette if this.patience > 0 =>
      // Action declenchee a la reception d'un message Pichenette
      // et dans le cas ou la variable patience est non nulle
      this.patience -= 1

    case Pichenette if patience == 0 =>
      // Actions declenchees a la reception d'un message Pichenette
      // et dans le cas ou la variable patience est nulle
      sender ! Avertissement
      // Changement d'etat Banner -> Hulk
      context become this.hulk

  }

  // Etat Hulk
  def hulk: Receive = {

    case Pichenette =>
      // Action declenchee a la reception d'un message Pichenette
      sender ! HulkSmash
      self ! Calme

    case Calme if sender == self =>
      // Action declenchee a la reception d'un message Calme que l'agent s'est
      // envoye a lui meme
      this.patience = 3
      context become this.banner

  }

}
```

FIGURE 7.3 : Extrait de code Akka traduisant le comportement illustré par la figure 7.2

composés de leurs propres agents composants.

Similairement à l'architecture V3A de [Morge *et al.* 2008], cette architecture composite permet une séparation claire des préoccupations. Un agent composant possède son propre comportement et son propre rôle dans les actions globales de l'agent composite. On peut assimiler ces agents aux facettes de l'architecture V3A. Cependant, contrairement aux facettes, les agents composants ne partagent pas de données et communiquent de façon asynchrone. De plus, les agents composants n'ont pas à régler de conflits de part la définition de leurs comportements individuels.

Une telle approche permet de spécifier de manière intelligible les différents rôles et comportements d'un même agent. Elle facilite également la co-occurence des délégations et des consommations de tâches. Cette section expose une architecture au sein de laquelle les actions de plusieurs agents composants, chacun avec son propre rôle et son propre comportement, produisent le comportement global de l'agent. La section 7.3.1 présente le *manager*, le *broker* et le *worker*, les trois agents composants d'un agent d'une instance MASTA. Ensuite, la section 7.3.2 décrit comment ces agents composants communiquent et interagissent pour faire émerger le comportement global d'un agent.

7.3.1 Agents composants

Un agent composite est composé de trois agents composants, tous dotés de leur propre file de messages et qui agissent simultanément :

1. le ***worker*** est l'agent composant chargé d'exécuter les tâches localement (cf. chapitre 4) ;
2. le ***broker*** est l'agent composant chargé d'initier des enchères en tant qu'initiateur ou de répondre à celles initiées par les pairs en tant qu'enchérisseur (cf. chapitre 5) ;
3. le ***manager*** est l'agent composant chargé de la gestion du lot de tâches. C'est lui qui implémente la stratégie de sélection de tâches et qui distribue les tâches à exécuter au *worker* et les tâches à déléguer au *broker*. Il s'occupe également de la mise à jour de la base de croyances (cf. chapitre 6).

Aucun des agents composants ne communique directement avec les autres agents du système. L'agent composite sert d'interface pour ses agents composants. Par exemple, lorsque le *broker* souhaite initier un `cfp`, il produit le message, l'envoie à son agent qui transfert le message à l'ensemble de ses pairs. Ce sont ensuite les autres agents du système qui reçoivent le `cfp` et le transmettent à leur propre *broker*. Le *broker* et le *worker* sont deux agents indépendants, ils ne communiquent pas. En revanche, le *manager* orchestre la distribution des tâches au *broker* et au *worker*. La figure 7.4 illustre la structure composite d'un agent ainsi que les différentes voies de communication qui existent entre les différentes entités d'un même agent.

7.3.2 Protocoles d'interaction des agents composants

Les agents composants sont créés localement par l'agent composite. Un agent et ses trois agents composants partagent donc tous la même localité. À l'inverse des communications entre agents distants, les communications entre les trois agents composants d'un même agent sont similaires à des appels de méthodes et ne sont donc pas sujettes à la perte ou au retard de messages. De ce fait, il n'est pas nécessaire d'ajouter un mécanisme de *timeout* comme celui présenté dans le chapitre 5.

Afin qu'un agent exhibe un comportement cohérent, les trois agents composants doivent se coordonner. Cette section décrit le protocole d'interaction qui permet au *manager*, au *broker* et au *worker* de communiquer et de se synchroniser pour consommer des tâches en continu tout en participant (en tant qu'initiateur ou enchérisseur) aux délégations socialement rationnelles qui profitent à l'ensemble du système.

Cette section consiste en une succession de descriptions de comportements. Pour chacun de ces

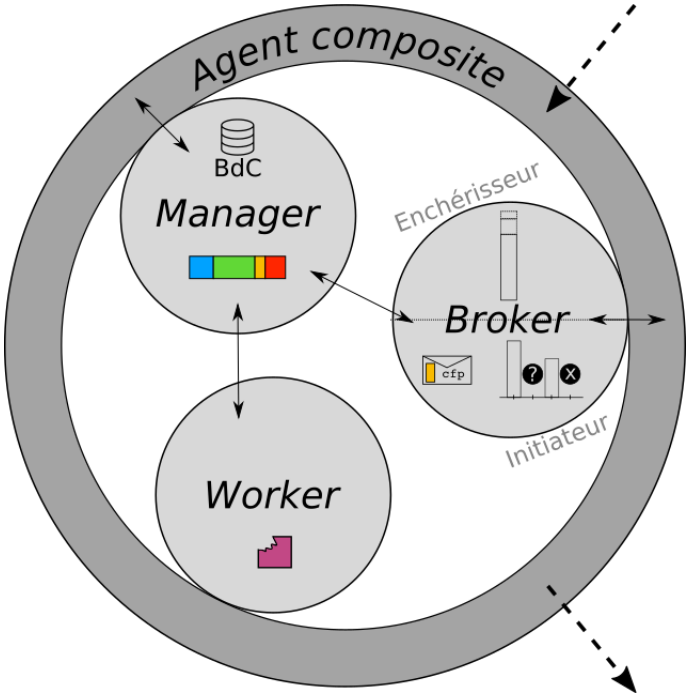


FIGURE 7.4 : Structure interne d'un agent. Les quadrilatères de couleur représentent les tâches (gérées par le *manager*, consommées par le *worker*, négociées par le *broker*). Le *manager* administre la base de croyances (BdC). Le *broker* peut tenir deux rôles mutuellement exclusifs : enchérisseur ou initiateur. Les flèches pleines représentent des communications entre les agents composants. Les flèches en pointillés représentent des communications entre l'agent et ses pairs.

comportements j'indique, entre parenthèses et à la suite de sa description, le nom du message qui le déclenche.

Consommer des tâches

Comme illustré dans la figure 7.5, les consommations de tâches impliquent le *manager* et le *worker*. L'objectif des agents étant d'exécuter l'ensemble des tâches au plus tôt, le *manager* priorise ses interactions avec le *worker* : dès que le *worker* produit le résultat d'une tâche, le *manager* en est informé (`done`) et founit une nouvelle tâche à consommer (`perform`).

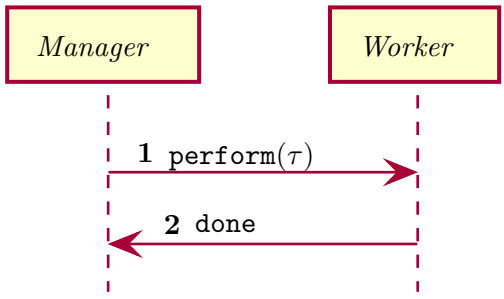


FIGURE 7.5 : Interaction entre le *manager* et le *broker* pour l'exécution de tâches.

Initier des délégations de tâches

Les délégations de tâches impliquent le *manager* et le *broker*. Le *manager* ne fournit une tâche à déléguer au *broker* qu'à partir du moment où le *worker* est occupé.

Comme illustré dans la figure 7.6, c'est le *manager* qui demande au *broker* d'initier une enchère (`submit`). Si le *broker* n'est pas engagé comme enchérisseur, il initie un `cfp`. Sinon, il rejette l'initiative du *manager* (`abort`).

Plusieurs scénarios sont possibles pour conclure une enchère. Quand toutes les réponses ont été reçues ou que le *timeout* survient, le *broker* évalue les réponses. Si aucune proposition n'a été reçue, le *broker* en informe le *manager* (`deny`) et met fin à l'enchère (cf. chapitre 5). Sinon, le *broker* sélectionne la meilleure proposition et rejette les autres. Il informe alors le *manager* qu'il est prêt à déléguer la tâche (`ready`). Le *manager* répond au *broker* en lui indiquant si la tâche est toujours disponible (`approve`) ou si elle a été donnée au *worker* durant l'enchère (`cancel`). Si la tâche n'est plus disponible car elle a été donnée au *worker* entre-temps, la proposition qui a été retenue est finalement rejetée (`reject`). Sinon, la tâche est envoyée à l'enchérisseur qui a remporté l'enchère (`accept`) et une confirmation est attendue. Il est possible que la tâche choisie par le *manager* pour être déléguée donne lieu à un `cfp` décliné par tous les pairs de l'agent, ce qui exprime une imprécision de la base de croyances au moment du choix de la tâche à déléguer. Dans ce cas, le *broker* en informe le *manager* (`declinedByAll`). Néanmoins, le *manager* peut de nouveau chercher une tâche à déléguer car la base de croyances a été mise à jour à chaque réponse d'un pair au *broker* (`informPeerWorkload`). Si aucune tâche ne peut être déléguée, le *manager* passe en état de pause.

Comme illustré dans la figure 7.7, chaque réponse à un `cfp`, que ce soit une proposition (`propose`) ou une déclinaison (`decline`), est prise en compte pour mettre à jour la base de croyances de l'agent (`informPeerWorkload`).

Participer aux enchères

La participation aux enchères demande l'implication des trois agents composants. Comme présenté dans la figure 7.8, lorsque le *broker* reçoit un `cfp` et qu'il n'est pas initiateur, il demande la charge de travail courante de l'agent au *manager* (`queryWorkload`) afin de déterminer s'il peut faire une proposition. Ensuite, le *manager* demande au *worker* une estimation du coût que représente la fin de l'exécution de la tâche courante (`queryRemainingWork`) pour donner au *broker* l'estimation la plus précise possible de la charge de travail de l'agent (`informWorkload`).

Comme le montre la figure 7.9, une fois que le *broker* a reçu l'information sur la charge courante, il peut utiliser sa surcharge virtuelle et déterminer si la délégation est socialement rationnelle. En fonction de la situation, le *broker* fait une proposition (`propose`), stocke le `cfp` pour le considérer plus tard (cf. chapitre 5) ou le décline (`decline`). Lorsque la phase d'appel à propositions est terminée, soit (a) le *broker* remporte l'enchère, transfert la tâche au *manager* (`request`) et confirme le succès du transfert à l'initiateur (`confirm`), soit (b) le *broker* ne remporte pas l'enchère. Lorsque le *broker* n'est plus impliqué dans aucune enchère, il est libre et informe le *manager* (`notBusy`).

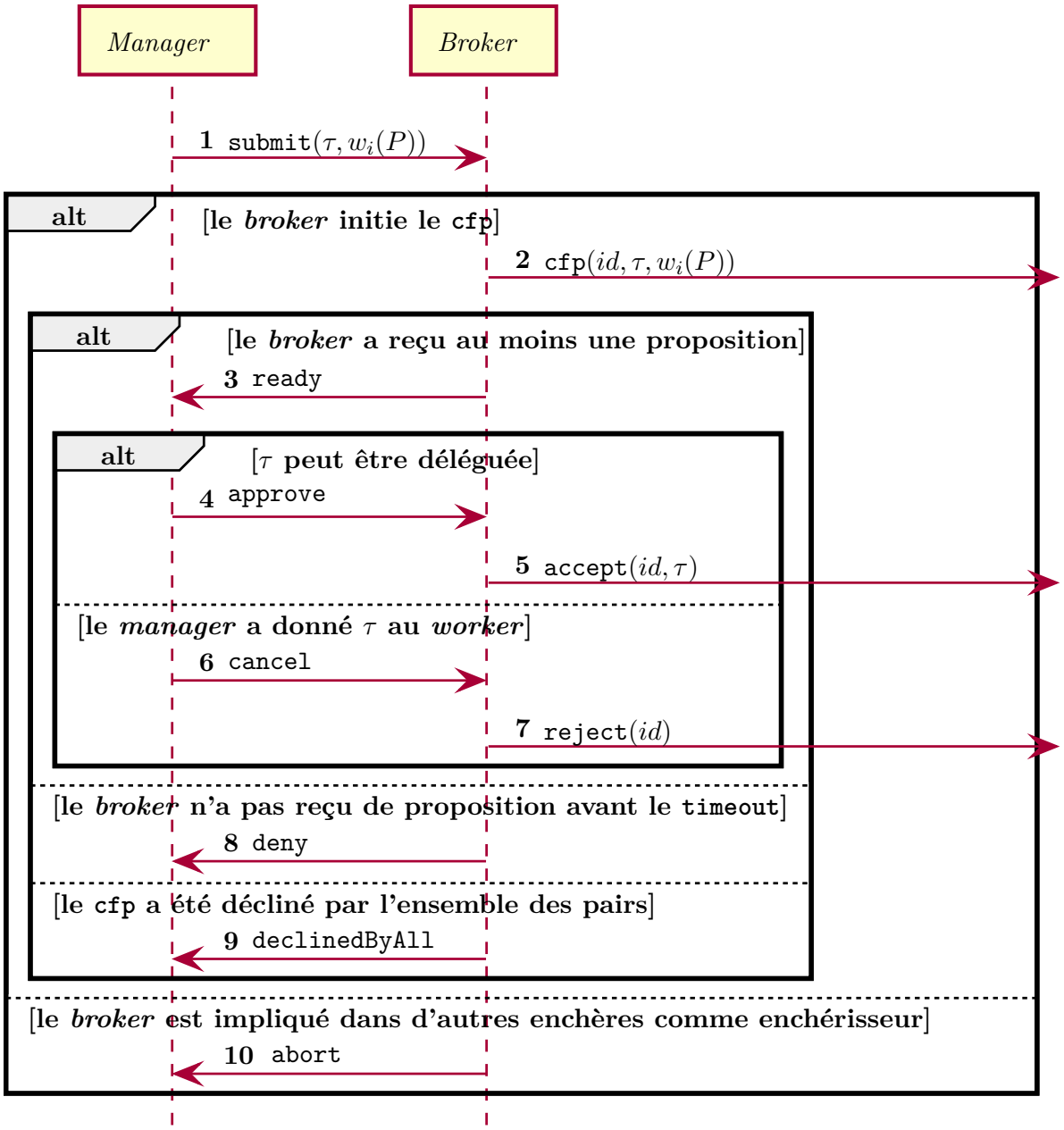


FIGURE 7.6 : Protocole d'interaction pour l'initiation d'enchère.

7.4 États d'un agent et de ses agents composants

Selon son état, un agent réagit différemment aux messages qu'il reçoit. Cette section décrit les différents états dans lesquels peut se trouver un agent composite (cf. section 7.4.1) et ses agents composants (cf. section 7.4.2).

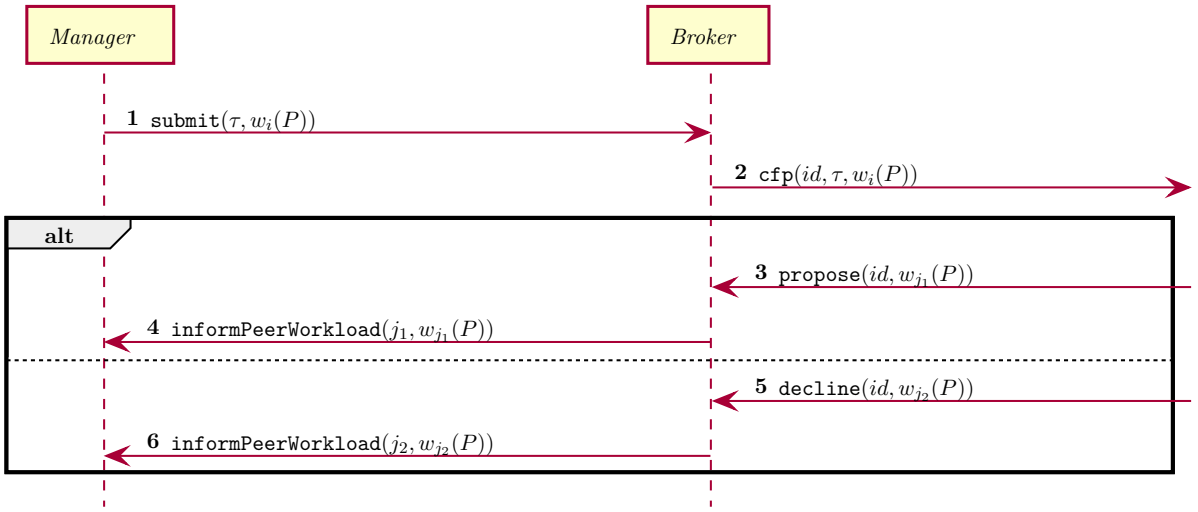


FIGURE 7.7 : Protocole d'interaction pour la mise à jour de la base de croyances.

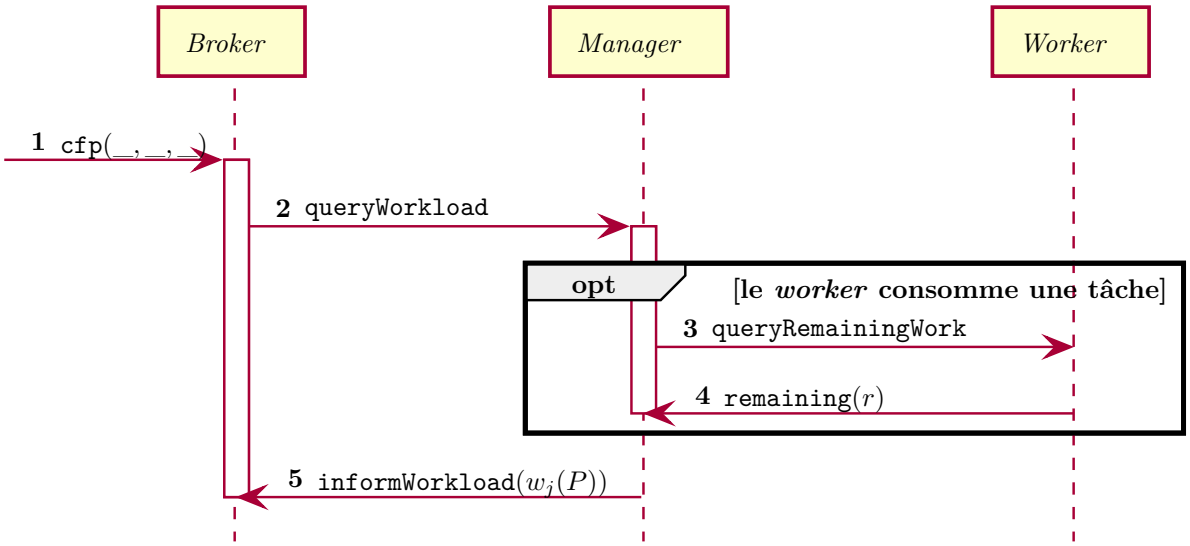


FIGURE 7.8 : Protocole d'interaction pour le calcul de la charge de travail courante. Les rectangles sur les lignes de vies des agents représentent des envois de messages synchrones, c'est-à-dire que l'agent attend la réponse du message qu'il vient d'envoyer.

7.4.1 États d'un agent composite

L'état d'un agent composite reflète les actions que ce dernier est capable d'effectuer, c'est-à-dire l'activité de ses agents composants. Un agent composite peut être dans trois états distincts : **actif** ($i_{\blacktriangleright}$), **en pause** ($i_{\blacksquare\blacksquare}$), ou **inactif** ($i_{\blacksquare}$).

Définition 7.1 (États d'un agent composite).

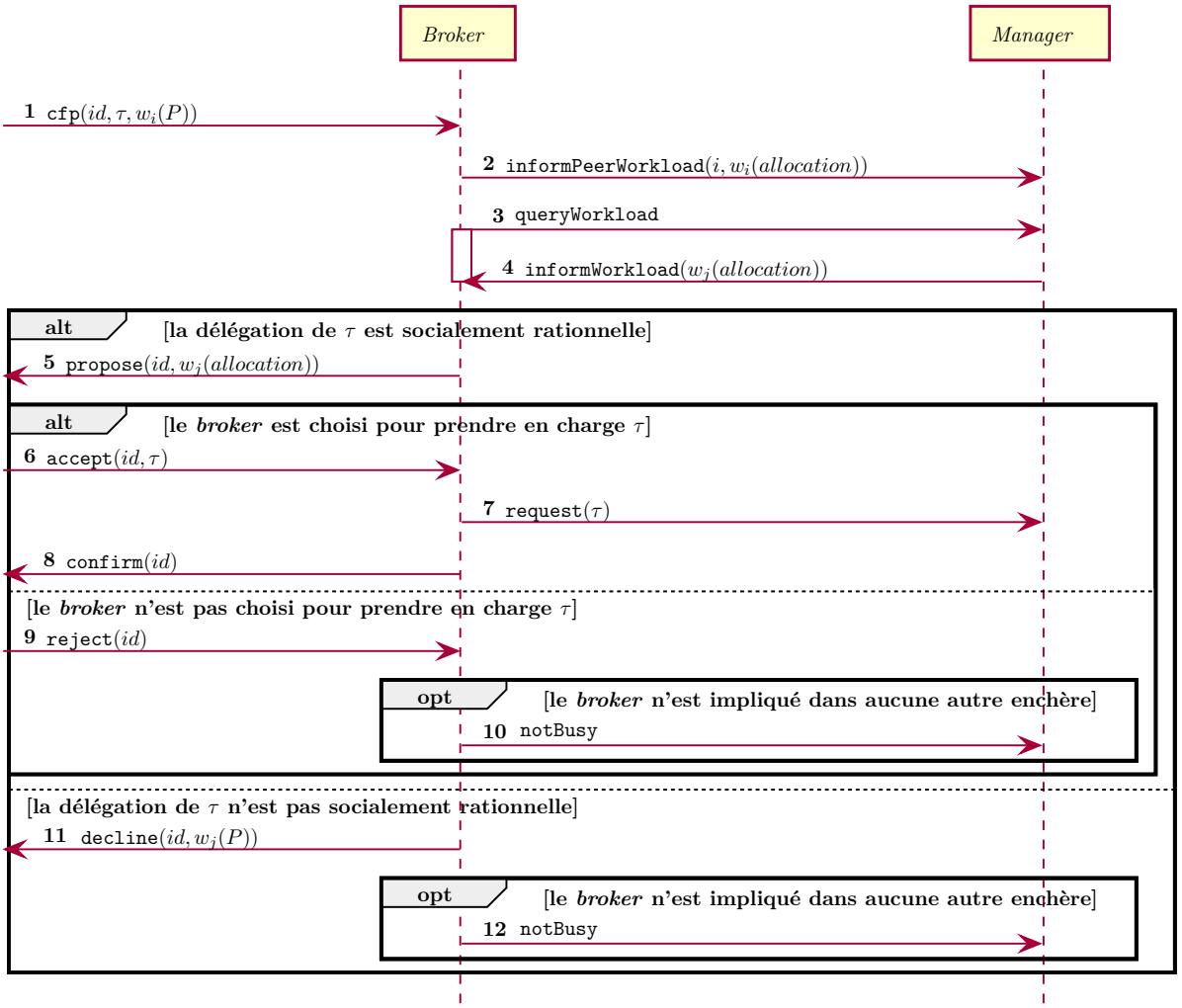


FIGURE 7.9 : Protocole d'interaction pour la participation aux enchères. Les rectangles sur les lignes de vies des agents représentent des envois de messages synchrones, c'est-à-dire que l'agent attend la réponse du message qu'il vient d'envoyer.

Soit P l'allocation courante. L'état de l'agent $i \in \mathcal{A}$ est :

$$i_{\blacktriangleright} \Leftrightarrow P_i \neq \emptyset \wedge \Gamma_i^{\mathcal{B}}(P) \neq \emptyset \tag{7.1}$$

$$i_{\blacksquare\!\blacksquare} \Leftrightarrow P_i \neq \emptyset \wedge \Gamma_i^{\mathcal{B}}(P) = \emptyset \tag{7.2}$$

$$i_{\blacksquare} \Leftrightarrow P_i = \emptyset \tag{7.3}$$

Soit i un agent composite. L'état de l'agent i est déterminé par son lot de tâches et ses croyances. Comme son *manager* gère le lot de tâches et a accès aux croyances, c'est également lui qui détermine quel est l'état de i.

L'agent i est actif s'il lui reste au moins une tâche à exécuter et qu'il est également en mesure d'initier une délégation socialement rationnelle (cf. équation 7.1). Lorsque i est actif, son *worker* consomme les tâches et son *broker* peut être initiateur ou enchérisseur. Si i ne peut plus initier de délégation socialement rationnelle d'après ses croyances, alors il est en état de pause (cf. équation 7.2). Lorsque i est en pause, son *worker* consomme les tâches, son *broker* peut uniquement

être enchérisseur. Enfin, si le lot de tâches de i est vide, alors il est inactif (cf. équation 7.3). Dans cet état, son *worker* est passif car il n'y a plus de tâche à consommer. Cependant son *broker* continue à enchérir lorsqu'il reçoit un `cfp`.

Les états des agents sont également un indicateur de la qualité de l'allocation courante. Tant qu'au moins un agent est actif, l'allocation courante peut être améliorée grâce aux délégations socialement rationnelles. Lorsque tous les agents sont en pause, cela traduit une allocation stable (cf. chapitre 4). Enfin, si tous les agents du système sont inactifs, cela signifie que toutes les tâches ont été consommées ($\mathcal{T} = \emptyset$).

7.4.2 États des agents composants

Au même titre qu'un agent composite, chaque agent composant possède différents états qui traduisent sa situation courante et déterminent son comportement. Par souci de clarté, n'est donnée dans cette section qu'une brève description de chaque état pour chaque agent composant. Cependant, les automates de l'annexe A présentent en détail les conditions de changement d'état et les réponses données à chaque message susceptible d'être reçu dans un état donné.

Manager

Comme précisé dans la section précédente, les états du *manager* reflètent précisément les états de l'agent : actif (*active*), en pause (*pause*) ou inactif (*idle*).

Lorsqu'il est actif, le *manager* donne des tâches à consommer au *worker* et à déléguer au *broker*. La priorité est toujours donnée au *worker*. Le *manager* passe en état de pause lorsqu'il n'a plus de tâche à déléguer. Dans ce cas, il continue à fournir des tâches à consommer au *worker*. Enfin le *manager* est inactif lorsque son lot de tâches est vide.

Quelque soit son état, le *manager* (a) met continuellement la base de croyances à jour grâce aux messages `informPeerWorkload` que lui envoie le *broker* et (b) ajoute systématiquement une tâche obtenue par le *broker* suite à une enchère (`request`). Dans le premier cas, cela permet au *manager* en pause d'éventuellement redevenir actif car une tâche devient délégable avec les nouvelles croyances. Dans le second cas, cela permet, d'une part, au *manager* d'avoir une nouvelle tâche à fournir au *worker* ou à considérer pour une délégation socialement rationnelle, et, d'autre part, de changer d'état en fonction du profil de la tâche (actif si délégable, en pause sinon).

Broker

Le *broker* possède deux rôles mutuellement exclusifs : initiateur ou enchérisseur. Sans directive de son *manager*, le *broker* est en attente (*broker*). Lorsqu'il est en attente, le premier message que le *broker* reçoit détermine son prochain état.

- S'il reçoit un `cfp` d'abord, le *broker* devient enchérisseur (*bidder*). Dans ce cas il refuse toute demande du *manager* (`abort`) et reste enchérisseur tant qu'il reçoit des `cfp` pour lesquels il est en mesure de faire des propositions.

- S'il reçoit une demande du *manager* pour soumettre une tâche (`submit`), le *broker* devient inititateur (*initiator*). Dans cet état, l'agent mène l'enchère (cf. chapitre 5) et stocke les `cfp` qu'il reçoit. Quand son enchère est terminée, il évalue les `cfp` stockés, fait une proposition pour ceux qui représentent une délégation socialement rationnelle et décline les autres. Si le *broker* génère une proposition, il passe directement dans l'état enchérisseur. Sinon, il est de nouveau en attente et en informe le *manager* (`notBusy`).

Tant qu'un *broker* reçoit des `cfp` et qu'il est en mesure de générer des propositions, alors il existe au moins un agent plus chargé dans le système. Dans ce cas, le *broker* reste enchérisseur et participe à décharger au maximum les agents plus chargés que lui.

Les automates de l'annexe A comportent d'autres états qui correspondent aux phases du *broker* en tant qu'enchérisseur ou initiateur. Par exemple, l'état contracteur (*contractor*) correspond à la phase dans laquelle un initiateur attend la confirmation de l'enchérisseur choisi pour bénéficier de la tâche qu'il délègue.

Worker

Le comportement du *worker* est le plus simple : il est occupé (*busy*) tant qu'il consomme une tâche ; il est libre (*free*) sinon. Dès que le *worker* est libre, il informe le *manager* (`done`) qui lui fournit au plus vite une nouvelle tâche à consommer.

7.5 Conclusion

La structure composite d'agent présentée dans ce chapitre est motivée par le fait qu'un même agent possède plusieurs responsabilités : la gestion de son lot de tâches, l'initiation et la participation aux enchères, et l'exécution des tâches. La gestion simultanée de ces actions par un agent unique aurait été d'une complexité élevée. Le fait d'attribuer chacune des trois responsabilités d'un agent à ses trois agents composants permet de clairement isoler, identifier et spécifier les rôles et les comportements de l'agent en fonction de son lot de tâches, de sa charge de travail et de ses croyances. Ainsi, un agent est composé d'un *manager* qui gère le lot de tâches, d'un *broker* qui s'implique dans les enchères comme initiateur ou enchérisseur et d'un *worker* qui exécute les tâches. Chacun des trois agents composants possède sa propre file de messages et l'agent composite redirige les messages extérieurs liées aux enchères directement à son *broker*.

Les ingrédients qui constituent un agent sont : son adresse, sa liste d'accointances, son état courant, son comportement et sa file de messages. En particulier, le comportement d'un agent détermine quelle est sa prochaine action en fonction de son état courant et du premier message de sa file. Chaque agent composant possède ses propres états qui indiquent quelle action réaliser à la réception d'un message. Par exemple, une fois que le *broker* est dans l'état initiateur, il ne répond pas aux `cfp` qu'il reçoit mais les stocke pour les considérer une fois que son enchère est terminée. Autre exemple, si le *manager* est en pause, ce dernier cesse de demander au *broker* d'initier des enchères car aucune de ses tâches n'est délégable selon ses croyances.

La difficulté dans la conception des comportements réside dans le fait que les agents ne partagent pas un état global. Ils n'ont qu'une connaissance partielle de l'état du système et de l'allocation. L'état d'un agent composite et de ses agents composants dépend de la représentation qu'ils

ont de l'allocation courante et traduit les actions que l'agent peut réaliser selon sa charge de travail et ses croyances. J'ai spécifié l'ensemble des comportements de chaque agent (composite ou composant) à l'aide d'automates finis déterministes. Comme le montre la section 7.2.2, cette spécification se transpose naturellement lors de l'implémentation des agents. Afin de quantifier la complexité des comportements des agents composants, le tableau 7.1 donne, pour chacun d'entre eux, le nombre de lignes de code nécessaires pour les implémenter ainsi que le nombre de transitions entrantes et sortantes pour chacun de leurs états.

Agent composant	Lignes de code	États	Transitions entrantes	Transitions sortantes
Worker	511	*free*	3	2
		busy	2	2
Manager	1352	*active*	32	43
		pause	22	17
		idle	16	10
Broker	1349	*broker*	16	11
		initiator	6	8
		awarder	4	6
		contractor	5	6
		bidder	8	7
		enquirer	6	6

Tableau 7.1 : Métriques des comportements des agents composants tels qu'implémentés dans MAS4Data.

Chapitre 8

MAS4Data

Sommaire

8.1 Introduction

Dans ce chapitre je présente MAS4Data (Multi-agent system for analyzing large data sets), le prototype distribué qui implémente le patron de conception MapReduce à l'aide d'un système multi-agents. Je montre que la phase de *reduce* d'un job MapReduce (cf. chapitre 3) peut être vue comme un problème MASTA. Comme décrit dans le chapitre 3, le déploiement du patron de conception MapReduce et la forme des données à traiter peuvent engendrer des biais, qui entrainent une mauvaise répartition de tâches de *reduce* parmi les *reducers*. Cette mauvaise répartition de la charge de travail impacte le temps d'exécution du job et se traduit par une utilisation sous-optimale des nœuds de calcul. En définissant des agents *reducers* capables de réaliser des délégations socialement rationnelles, il est possible d'améliorer la répartition de la charge de travail des différents nœuds de calcul au cours de la phase de *reduce*.

Les deux biais de la phase de *reduce* sont le biais de partitionnement et le biais des clés coûteuses. Alors que le premier peut être résolu par une réallocation des tâches de *reduce*, le second est souvent plus complexe à corriger. Cependant, dans le cas où le job MapReduce possède des propriétés précises, il est possible pour les agents d'utiliser un processus de découpe de tâche basé sur l'heuristique présentée dans le chapitre 6 pour découper les tâches très coûteuses et donc difficilement négociables.

Ce chapitre présente comment la résolution d'un problème MASTA permet d'améliorer les performances d'un job MapReduce distribué. Dans un premier temps, la section 8.2 décrit comment la phase de *reduce* d'un job MapReduce peut être décrite par une instance MASTA. Ensuite, la section 8.3 présente le mécanisme de découpe de tâche de *reduce* basé sur l'heuristique de découpe présentée en section 6.3. Les différences entre l'implémentation de MapReduce par un système multi-agents et l'implémentation de référence Hadoop sont abordées dans la section 8.4. Enfin la section 8.5 conclut ce chapitre et en présente quelques perspectives.

8.2 Liens entre un job MapReduce et une instance MASTA

Représenter la phase de *reduce* d'un job MapReduce comme une instance MASTA permet de bénéficier des propriétés de convergence et de monotonie démontrées dans le chapitre 4. En attribuant un agent à chaque nœud *reducer* et en faisant négocier ces agents, il est possible de diminuer l'impact des biais de la phase de *reduce* et réduire le *makespan* de l'ensemble du job.

Dans un premier temps, cette section illustre comment la phase de *reduce* d'un job MapReduce peut être décrite par une instance MASTA. Ensuite, elle présente comment l'amélioration d'une instance MASTA permet de corriger les biais susceptibles d'apparaître lors d'une phase de *reduce*.

8.2.1 Description de la phase de *reduce* par une instance MASTA

Comme expliqué dans la section 3.2, la distribution d'un job MapReduce sur plusieurs nœuds de calcul (ou une grappe de PCs) réclame (a) de désigner des nœuds *mappers* qui appliquent la fonction de *map* pour filtrer les données et les regrouper par clé et (b) de désigner des nœuds *reducers* qui vont appliquer la fonction de *reduce* sur les données agrégées d'une même clé pour produire le résultat lié à cette clé. Les deux phases étant consécutives, les rôles de *mapper* et de *reducer* ne sont pas mutuellement exclusifs. Autrement dit, un nœud peut être successivement *mapper* et *reducer*.

Lors de la phase de *map*, un sous-ensemble des données est attribué à chaque *mapper*. Lorsqu'un *mapper* applique la fonction de *map* sur ses données, il produit des couples (`clé`, `valeur`). Les couples qui se réfèrent à la même clé sont regroupés en *chunks*. Les *chunks* sont des fichiers de taille limitée, c'est le nombre de valeurs que produit un *mapper* pour une clé qui détermine le nombre de *chunks* associés à cette clé. Ainsi, à la fin de la phase de *map*, chaque *mapper* a produit un ou plusieurs *chunks* pour chaque clé qu'il a rencontrée. Plusieurs *mappers* sont susceptibles d'avoir produit des *chunks* pour la même clé. Pour s'assurer de la cohérence du résultat, il est essentiel que tous les *chunks* d'une même clé soient traités par le même *reducer*. Pour satisfaire cette contrainte, une fonction de partition détermine quelles clés sont affiliées à quel *reducer*. Pour rappel, ce mécanisme est statique et déterminé avant la phase de *map*. En effet, l'utilisateur ne connaît pas à l'avance les clés produites par les *mappers*, ni quels *mappers* produisent quelles clés, ni la fréquence à laquelle chaque clé sera générée. Cette répartition statique des clés est la cause du biais de partitionnement (cf. section 3.3.1).

Un *reducer* connaît donc l'ensemble des *chunks* associés aux clés qu'il doit traiter. L'exécution d'une tâche de *reduce*, c'est-à-dire la production du résultat lié à une clé, consiste à traiter

l'ensemble des valeurs qui lui sont associées [1]. Les *chunks* représentent donc les ressources qu'un *reducer* doit rassembler pour l'exécution d'une tâche de *reduce*.

Une fois qu'il a reconstitué l'ensemble des tâches de *reduce* qu'il doit exécuter, un *reducer* les consomment une à une. En lisant l'ensemble des *chunks* d'une clé, le *reducer* possède toutes les données sur lesquelles appliquer la fonction de *reduce*. Une fois que le résultat d'une tâche est produit, il est écrit sur le disque local du *reducer* et la tâche est considérée comme terminée. Le job MapReduce est achevé une fois que toutes les tâches de *reduce* ont été consommées. De ce fait, le *makespan* de l'allocation des tâches de *reduce* a un impact direct sur le temps nécessaire pour achever le job : améliorer l'allocation des tâches permet de faire décroître le temps d'exécution de l'ensemble du job.

Le tableau 8.1 décrit les correspondances entre les différents éléments d'une instance MASTA et les composants essentiels à la distribution d'une exécution MapReduce. Comme on peut le constater, la fonction de coût est un élément essentiel à la résolution d'une instance MASTA mais ne possède pas d'équivalent explicite pour MapReduce. Une méthode simple pour déterminer le coût d'une tâche peut consister à utiliser les informations disponibles et qui reflètent son volume, par exemple le nombre de *chunks* ou le nombre de valeurs qui lui sont associées.

MASTA		MapReduce
S	$\mathcal{N}$	Grappe de PCs / nœuds de calcul
	$\mathcal{A}$	*Reducers*
	$\mathcal{E}$	Possibilités de communication / d'échange de données entre les nœuds de calcul
	$\mathcal{R}$	*Chunks* produits par les *mappers*
	l	Déploiement d'un *reducer* par nœud de calcul
	d	Chaque *chunk* se trouve sur le nœud du *mapper* qui l'a produit
$\mathcal{T}$		Tâches de *reduce*
P		Allocation des tâches de *reduce* par la fonction de partitionnement
c		NP

Tableau 8.1 : Équivalence entre les termes d'une instance MASTA et les composants d'une exécution MapReduce distribuée. Une exécution MapReduce distribuée ne demande pas d'estimer le coût d'une tâche : la fonction de coût c est un élément non pertinent (NP) dans le cadre applicatif habituel de MapReduce.

8.2.2 Correction des biais de la phase de *reduce*

Considérer la phase de *reduce* comme un problème MASTA permet de réallouer dynamiquement les tâches de *reduce* alors que celles-ci sont exécutées. Ainsi, les agents *reducers* utilisent le protocole décrit dans le chapitre 5 pour effectuer des délégations de tâche socialement rationnelles. Ces délégations, concurrentes aux exécutions des tâches, permettent de corriger le biais de partitionnement en équilibrant la charge de travail parmi les nœuds *reducers* et par conséquent de réduire le temps d'exécution de l'ensemble du job MapReduce.

Même dans le cas d'une allocation stable, le risque qu'un des nœuds de calcul ralentisse en cours d'exécution existe. Comme présenté dans la section 4.4, le fait que les délégations de tâches soient

1. cf. exemple 3.1 pour un exemple de construction d'une tâche de *reduce*.

réalisées pendant l'exécution des tâches permet d'affiner l'allocation de tâches tout au long de la phase de *reduce*. En cas de ralentissement, les agents corrigent dynamiquement l'allocation de tâches grâce aux ajustements continus permis par les délégations socialement rationnelles.

Le second biais de la phase de *reduce* est le biais des clés coûteuses. Ce biais apparaît notamment lorsque le traitement d'un sous-ensemble de clés demande excessivement plus de temps de calcul que les autres clés. Cette disparité provoque une différence de charge de travail qui ne peut être compensée par une meilleure allocation des tâches (cf. chapitre 3). Le biais des clés coûteuses ne peut donc pas être résolu ni par une fonction de partition classique ni par un processus de négociation uniquement. Cependant ce biais peut être corrigé grâce à un mécanisme de découpe de tâches basée sur l'heuristique présentée dans le chapitre 6. Une tâche de *reduce* peut être découpée en répartissant ses chunks de sorte à créer plusieurs sous-tâches. La section 8.3 discute des contraintes spécifiquement liées à la découpe des tâches *reduce*.

Comme cette section l'expose, les mécanismes et stratégies de résolution d'une instance MASTA présentés dans l'ensemble de ce manuscrit permettent de corriger les biais de la phase de *reduce*. En déléguant et découpant les tâches de *reduce*, les agents *reducers* sont en mesure de diminuer le *makespan* de l'allocation courante et ainsi de faire décroître le temps d'exécution du job MapReduce. Cette amélioration du temps d'exécution sera mesurée dans le chapitre 9.

8.3 Découpe des tâches de *reduce*

Cette section, décrit le processus effectif de découpe de tâche de *reduce*. Ce processus est basé sur l'heuristique présenté dans la section 6.3 et il est inspiré du travail de [Liroz-Gistau *et al.* 2016] dont la contribution a été décrite dans le chapitre 3.

Par la suite, je présente les conditions que doit remplir un job MapReduce pour que les tâches de *reduce* puissent être découpées. J'explique également comment conserver la cohérence des résultats malgré le fait que les sous-tâches extraites d'une même tâche *reduce* puissent être consommées par différents *reducers*. L'annexe B, montre comment l'heuristique de découpe de tâche a été modifiée pour être appliquée spécifiquement aux tâches de *reduce*. Cette annexe contient également l'algorithme de découpe de tâche *reduce*.

8.3.1 Conditions pour la découpe de tâches

Pour que la découpe de tâches soit possible et efficace il est indispensable que la fonction de *reduce* puisse être décomposée en deux fonctions complémentaires : la **fonction de *reduce* intermédiaire** (IR) et la **fonction de *reduce* finale**. En d'autres termes, il faut que l'on puisse exprimer le job MapReduce la façon suivante :

$$\begin{aligned} map &: (K_1, V_1) \rightarrow [(K_2, V_2)] \\ IR &: (K_2, [V_2]) \rightarrow (K_2, [V_2]) \\ FR &: (K_2, [V_2]) \rightarrow (K_3, V_3) \end{aligned}$$

Pour une tâche donnée, la fonction IR permet de produire un résultat en ne traitant qu'un sous-ensemble des *chunks* de la tâche. On appelle ce résultat, un **résultat intermédiaire**. La fonction FR permet de produire le résultat final de la tâche à partir de tous ses résultats intermédiaires.

Dans [Gray *et al.* 1997], les auteurs identifient trois familles de fonctions d'agrégation. Les fonctions distributives sont celles où les fonctions de *reduce* intermédiaire et finale sont une seule et même fonction, par exemple `sum`, `min`, `max`, `count`. Les fonctions algébriques peuvent être décomposées en utilisant un nombre borné de fonctions distributives, par exemple `average` est décomposée en utilisant un *reduce* intermédiaire qui calcule une somme (`sum`) et un nombre de valeurs (`count`), la division de la somme totale par le nombre total de valeurs se faisant dans le *reduce* final. Enfin, les fonctions holistiques ne sont ni distributives, ni algébriques, par exemple le calcul d'une médiane. Il est difficile de trouver une « bonne » décomposition d'une fonction holistique, en particulier de trouver une fonction de *reduce* intermédiaire efficace qui ne nécessite pas de mémoriser trop d'informations.

Dans le flux de données MapReduce, le rôle de la fonction de *reduce* intermédiaire est similaire à celui de la fonction optionnelle de combinaison, c'est-à-dire réduire le nombre de valeurs associées à une clé en les traitant partiellement (cf. chapitre 3). Ainsi, un job MapReduce pour lequel une fonction de combinaison est définie peut être décomposée comme ci-dessus en donnant la fonction de combinaison comme fonction de *reduce* intermédiaire.

8.3.2 Agrégation des résultats intermédiaires

La découpe de tâche se base sur la définition de deux fonctions de *reduce* : une fonction de *reduce* intermédiaire (IR) et une fonction de *reduce* finale (FR). Lorsqu'une tâche de *reduce* est exécutée sans être découpée, les *reducers* appliquent directement la fonction FR pour en produire le résultat. Lorsqu'une tâche de *reduce* est découpée, ses sous-tâches sont marquées comme tâches de *reduce* intermédiaires, et les *reducers* doivent leur appliquer la fonction IR. Ces sous-tâches sont considérées comme n'importe quelles autres tâches et sont donc candidates à la négociation ainsi qu'à la découpe. Les résultats intermédiaires d'une même tâche initiale doivent cependant être regroupés pour permettre d'établir le résultat final de la tâche. C'est le *reducer* qui initie la première découpe d'une tâche qui a la responsabilité de produire son résultat final. On appelle ce *reducer* le ***reducer* final**. En plus de la clé concernée et des *chunks* à traiter, chaque sous-tâche porte donc également l'adresse du *reducer* final.

Quelque soit le nombre de découpes intermédiaires, tous les résultats intermédiaires d'une tâche sont amenés à un même *reducer* final. Cela permet de préserver la cohérence du résultat final. Cette particularité demande un traitement spécifique des résultats intermédiaires. Si un *reducer* exécute une tâche de *reduce* intermédiaire et qu'il est lui-même le *reducer* final de cette tâche, il en conserve le résultat jusqu'à ce qu'il ait reçu tous les autres résultats intermédiaires pour produire le résultat définitif de la tâche. En revanche, si un *reducer* exécute une tâche de *reduce* intermédiaire et qu'il n'est pas le *reducer* final de cette tâche, il envoie ce résultat au *reducer* final.

> **Exemple 8.1.**
> *Considérons la découpe de la tâche τ par le reducer i. Ce reducer décide de découper τ en $\{\tau_1, \tau_2, \tau_3\}$, il traite lui même τ_1 et délègue les tâches τ_2 et τ_3 aux agent j_2 et j_3 respectivement. Le reducer j_2 décide à son tour de découper τ_2 en τ_{21} et τ_{22} afin de déléguer τ_{22} à un quatrième reducer. Les résultats de l'application de IR sur toutes les tâches $\{\tau_1, \tau_{21}, \tau_{22}, \tau_3\}$ sont envoyés à i, le reducer qui a initié la division de la tâche τ. Ainsi, i peut appliquer*

> *la fonction FR sur l'ensemble des résultats intermédiaires et produire le résultat final de la tâche τ.*

8.4 Similitudes et différences avec Hadoop

L'implémentation distribuée de référence du patron de conception MapReduce est Hadoop [Hadoop 2019]. Pour mettre en œuvre la proposition de cette thèse, j'ai implémenté MAS4Data, un prototype qui déploie un système multi-agents sur plusieurs nœuds de calculs et autorise la réallocation des tâches *reduce* pendant leur exécution. Cette réallocation offre à MAS4Data la possibilité de contrer les biais de la phase de *reduce*, ce qui est impossible pour Hadoop. Cependant, MAS4Data présente quelques différences avec la philosophie de Hadoop. Cette section a pour but de les identifier.

Exécution des tâches de *reduce*

Hadoop applique le patron de conception MapReduce tel qu'il est décrit dans le chapitre 3. De ce fait, lorsque qu'une tâche de *reduce* est affectée à un *reducer* par la fonction de partition, c'est généralement ce *reducer* qui exécute cette tâche et en produit le résultat.

À l'inverse, dans MAS4Data, les tâches de *reduce* sont constamment réaffectées suite aux négociations initiées par les agents. Dans l'impossibilité de prédire quel *mapper* produit quelle clé sans prétraiter les données, les tâches sont également assignées de manière statique par une fonction de partition prédéfinie. Cette fonction de partition détermine l'allocation initiale des tâches de *reduce*. Toutefois, celle-ci est continuellement remise en question par les *reducers* qui cherchent à améliorer son *makespan* grâce aux délégations socialement rationnelles.

Système de fichiers distribué

Hadoop utilise un système de fichiers distribué. Ainsi, lorsque les *mappers* produisent les *chunks*, ces derniers se trouvent dupliqués sur plusieurs nœuds du système. Malgré la réplication des *chunks*, un *reducer* peut ne pas avoir un accès direct à l'ensemble des *chunks* liés à une clé. Dans ce cas, pour qu'il puisse produire le résultat, les *chunks* manquants doivent être transférés.

Pour faciliter l'interprétation des résultats expérimentaux (cf. chapitre 9), MAS4Data n'utilise pas de système de fichiers distribué. Les *chunks* sont donc écrits sur le système de fichiers local des *mappers* qui les produisent. Comme les *reducers* utilisent les délégations de tâche socialement rationnelles, les tâches de *reduce* circulent tout au long de la phase de *reduce*. Afin de ne pas déplacer inutilement les *chunks*, les *chunks* liés à une tâche ne sont transférés qu'au moment de son exécution par un *reducer*. Ainsi, après la phase de *map*, les *mappers* envoient une description des *chunks* aux *reducers*. Pour chaque *chunk* lié à une clé, le *reducer* connaît le nombre de valeurs qu'il contient ainsi que le nœud sur lequel il se trouve. Les phases de *map* et de *reduce* étant consécutives, il est possible de placer un *mapper* et un *reducer* sur le même nœud sans pénaliser les performances des deux phases. Si un *reducer* se trouve sur un nœud sur lequel un *mapper* a produit des *chunks*, ces *chunks* sont locaux pour lui. Lors de l'exécution d'une tâche, le *reducer* fait des requêtes sur les nœuds où se trouvent les *chunks* qui ne lui sont pas locaux afin de les

transférer et de pouvoir appliquer la fonction de *reduce*.

Tolérance aux pannes

J'appelle panne l'arrêt complet d'un nœud de calcul. D'après la classification présentée dans [Potiron *et al.* 2009], une panne est donc une faute opérationnelle, externe, matérielle, naturelle, non-malveillante, accidentelle et permanente. Un système tolérant aux pannes permet l'arrêt complet d'un nœud en cours d'exécution sans que cela entraîne la perte des tâches ou des résultats. Autrement dit, un système tolérant aux pannes dispose d'un mécanisme qui permet de récupérer les résultats produits ou de réallouer les tâches exécutées par un nœud en panne.

Hadoop est un système centralisé. Chaque nœud de calcul envoie régulièrement des messages de status (*heartbeat messages*) à un superviseur qui centralise la localité de chaque *chunks*, l'état de chaque nœud et l'état d'avancement des tâches en cours. En cas de panne d'un nœud ou d'un ralentissement important, ce superviseur est donc capable de tuer le *reducer* défectueux et de relancer le traitement de ses tâches sur un autre nœud.

MAS4Data est totalement décentralisé. Un unique superviseur est bien en charge d'initier le système distribué et d'y déployer les agents mais ce dernier ne centralise aucune information sur le système. Comme décrit dans le chapitre 4, chaque agent possède sa propre perception de ses pairs et de l'allocation de tâches courante. Ainsi, n'ayant pas à consulter une entité qui concentre toutes les informations sur l'état courant du système, les agents sont capables de réagir de manière réactive et indépendante pour corriger l'allocation de tâches courante. Leurs décisions sont donc locales. Pour l'instant, cela permet seulement aux agents d'être réactifs à une chute de performances et l'implémentation d'une tolérance aux pannes est aujourd'hui une perspective.

8.5 Conclusion

8.5.1 Synthèse

Ce chapitre décrit comment la phase de *reduce* d'un job MapReduce se traduit dans le cadre formel de l'amélioration d'une instance MASTA. Le déploiement d'un agent par nœud *reducer* autorise la réallocation dynamique des tâches pendant la phase de *reduce*. Ainsi, en appliquant les mécanismes et stratégies décrits dans la partie précédente de cette thèse, les agents *reducers* sont à la fois capables d'améliorer le *makespan* de la phase de *reduce* et capables d'adapter l'allocation de tâches aux fluctuations de performances. Au même titre qu'elle est bénéfique à la résolution d'un problème MASTA, la co-occurence des délégations socialement rationnelles et des consommations de tâches *reduce* permet de corriger le biais de partitionnement de la phase de *reduce*.

Le biais des clés coûteuses peut nécessiter la découpe des tâches de *reduce*. Grâce à la structure particulière d'une tâche de *reduce*, il est possible de créer des sous-tâches à partir des *chunks* de la tâche initiale. Cependant, pour que les *reducers* soient capables de conserver la cohérence des résultats malgré la découpe des tâches, il est nécessaire que la fonction de *reduce* puisse être décomposée en une fonction de *reduce* intermédiaire et une fonction de *reduce* finale. Lorsqu'une

tâche est découpée, différents *reducers* traitent les sous-tâches en appliquant la fonction IR et envoient les résultats intermédiaires au *reducer* final. Une fois qu'il a récolté l'ensemble des résultats intermédiaires, ce dernier applique la fonction FR pour produire le résultat définitif de la tâche.

Afin de pouvoir appliquer les concepts décrits dans cette thèse sur des données réelles, j'ai implémenté un prototype distribué appelé MAS4Data. Contrairement à Hadoop, celui-ci est un système décentralisé qui autorise la réallocation des tâches de *reduce* pendant leur exécution grâce aux délégations de tâche sociallement rationnelles. Le chapitre suivant présente les expérimentations et les conclusions quant à l'efficacité et l'applicabilité d'un tel processus.

8.5.2 Perspectives

Une première perspective de ce chapitre est d'explorer la possibilité d'intégrer ce processus de réallocation à l'écosystème Hadoop. Ainsi, il serait possible de comparer une exécution qui utilise le processus de réallocation avec une exécution classique. En terme de modèle, cela demande uniquement d'ajouter la prise en compte d'un système de fichiers distribué, c'est-à-dire de considérer qu'une ressource est répliquée sur plusieurs nœuds. Cependant, cela représente sûrement un travail conséquent d'ingénieurie pour gérer le transfert de tâches, l'accès aux *chunks* et les échanges entre *reducers*.

Une seconde perspective est de considérer l'impact que pourrait avoir l'ajout d'un *reducer* pendant la phase de *reduce*. Avec une exécution classique de MapReduce, un tel *reducer* est inutile car aucune tâche ne lui a été attribuée par la fonction de partition. À l'inverse, le processus peut facilement bénéficier de l'ajout d'un agent en cours d'exécution. En effet, grâce à la réallocation continue des tâches de *reduce*, ce nouveau *reducer* peut rapidement se voir attribuer des tâches et participer à améliorer le *makespan* de l'allocation courante. Ainsi, si le système distribué dispose de nœuds inactifs pendant la phase de *reduce*, il suffit à l'agent qui lui est associé d'annoncer sa présence à ses pairs avant de se voir participer aux enchères et attribuer des tâches. Le problème ici est plutôt de décider quand et pourquoi ajouter un agent au système.

Enfin, une troisième perspective est d'inclure la tolérance aux pannes dans le prototype. Comme énoncé dans la section 8.4, le mécanisme de réallocation dynamique des tâches de *reduce* permet de corriger la baisse de performances d'un nœud de calcul mais pas sa disparition du système distribué en cours d'exécution. En distribuant l'information de quel agent dispose de quelles tâches, il est possible de redistribuer les tâches perdues lors de la panne d'un nœud de calcul.

Chapitre 9

Validations expérimentales

Sommaire

9.1 Introduction

MAS4Data est un prototype qui permet l'exécution distribuée d'un job MapReduce grâce au déploiement d'un système multi-agents (cf. chapitre 8). Dans un tel système, des agents *reducers*, dont le comportement et l'architecture sont décrits dans le chapitre 7, réallouent les tâches de *reduce* pour améliorer l'équilibre des charges sur les différents nœuds de calcul et faire décroître le temps d'exécution du job.

Dans ce chapitre, je corrobore les contributions mises en avant dans cette thèse au travers d'expérimentations réalisées avec MAS4Data sur divers jeux de données. Au travers de ces expériences, je cherche à montrer que :

- la réallocation des tâches permet une meilleure répartition des charges de travail ;
- les agents ajustent l'allocation de tâches tout au long de l'exécution afin de faire diminuer la charge de travail la plus élevée du système ;
- lors d'une exécution MapReduce, la stratégie de découpe de tâche permet de contrer le biais des clés coûteuses ;
- lors des négociations, il est préférable que les agents considèrent la localité des *chunks*, et donc qu'ils utilisent la stratégie de sélection de tâches localisées.

Pour juger de la qualité d'une allocation de tâches, trois nouvelles métriques sont introduites : la **contribution**, l'**équité de contribution** et l'**équité de temps**. La contribution d'un agent représente la quantité de travail qu'il a réalisé. Elle est donc égale à la somme des coûts des tâches que l'agent a exécutées. L'équité de contribution d'une allocation se mesure en divisant la contribution la moins élevée par la contribution la plus élevée. Plus l'équité de contribution d'une allocation est proche de 1, plus la contribution des agents est similaire. Dans le cas d'agents homogènes en terme de capacités de calcul, une équité de contribution proche de 1 signifie donc que la charge de travail a correctement été répartie. De manière similaire, l'équité de temps se mesure en divisant le temps d'exécution de l'agent qui termine le premier par celui de l'agent qui termine le dernier. Une équité de temps proche de 1 signifie que les agents ont terminé leur travail presque simultanément. Dans le cas d'agents hétérogènes, on s'attend à ce que les agents les plus performants aient une contribution plus élevée. Dans ce cas, l'équité de temps est une mesure plus pertinente que l'équité de contribution. En effet, une équité de temps élevée indique que l'allocation des tâches a pris en compte les différences de capacités de calcul des agents. Ces trois métriques sont mesurées *a posteriori*, c'est-à-dire une fois l'exécution des tâches terminée. La majorité des figures de ce chapitre représentent donc des observations *ex post*.

Dans la plupart des expériences, le coût d'une tâche correspond au volume des données à traiter, c'est-à-dire au nombre de valeurs qui lui sont associées. Appelons cette fonction la **fonction de coût par valeurs**. Avec cette fonction, le coût d'une tâche est le même pour tous les agents du système ($\forall i, j \in \mathcal{A}, \forall \tau \in \mathcal{T}, c_i(\tau) = c_j(\tau)$).

La section 9.2 commence par présenter les jeux de données utilisés. La section 9.3 montre l'impact du processus de réallocation sur la répartition de la charge de travail. La section 9.4 met en lumière la dynamique de ce processus. La section 9.5 confirme que les réallocations permettent de s'adapter à l'hétérogénéité des agents, même quand celle-ci n'est pas traduite par la fonction de coût utilisée. Ensuite, la section 9.6 illustre l'apport de la découpe de tâche. La section 9.7 compare quelques stratégies de sélection de tâche présentées dans le chapitre 6. Enfin, la section 9.8 conclut ce chapitre et offre quelques perspectives.

9.2 Description des jeux de données

Quatre jeux de données ont été utilisés lors des expérimentations décrites ci-dessous. Trois de ces jeux de données sont réels et proviennent de Météo France ou de Yahoo !. Le dernier jeux de données a été généré.

Le premier jeu de données utilisé contient plus de 3 millions d'enregistrements météorologiques (identifiant de station, horodatage, température, pluviométrie, ...) provenant de 62 stations au cours des 20 dernières années [Météo France 2019]. Deux jobs sont considérés sur ces données :

1. le premier `EnrPrTemp` (pour « enregistrement par température ») compte le nombre d'enregistrements par demi-degré de température. Ce job crée 209 clés et il est réalisé par 10 *mappers* et 20 *reducers* ;
2. le second `PluPrStat` (pour « pluviométrie par station ») compte les précipitations accumulées par station. Ce job crée 62 clés, il est effectué par 10 *mappers* et 10 *reducers*.

Le second jeu de données est noté Yahoo ! Auction. Il provient de la plate-forme d'enchères exploitée par Yahoo ! pour la vente d'espace publicitaire. Sur cette plateforme, les annonceurs

enchérissent pour apparaître à côté des résultats de requêtes de recherche particulières. Par exemple, un fournisseur de voyages peut demander le droit d'apparaître à côté des résultats de la requête de recherche « Voyage Las Vegas ». L'offre d'un annonceur est le prix qu'il est prêt à payer chaque fois qu'un utilisateur clique sur ses annonces. L'ensemble des données analysées correspond à la période du 15 juin 2002 au 14 juin 2003. Il contient 77×10^6 enchères (jour, identifiant de l'annonceur, liste de mots-clés, etc.), soit l'équivalent de 8 Go de données [Yahoo! 2019]. Puisqu'une enchère est liée à une liste de mots-clés, nous allons considérer le job `CountByKeyword` qui compte le nombre d'enchères pour chaque mot clé. Ce job crée 5209 clés et il est réalisé par 10 *mappers* et 10 *reducers*.

Le troisième jeu de données est noté Yahoo ! Music et représente un sous-ensemble des préférences de la communauté musicale de Yahoo ! pour diverses chansons. Les données contiennent plus de 717 millions d'évaluations de 136 000 chansons par 1,8 millions d'utilisateurs des services de musique Yahoo !. Les données ont été recueillies entre 2002 et 2006 pour un volume total de 10 Go [Yahoo! 2019]. Le job considéré est `MusicEval` qui consiste à compter le nombre d'évaluations à n étoiles ($n \in [1;5]$). Cet ensemble de données contient donc 5 clés (une par nombre d'étoile) et donne 5 tâches de *reduce*. Le job `MusicEval` est réalisé par 10 *mappers* et 12 *reducers*.

Enfin, le dernier jeu de données a été généré pour tester la stratégie de sélection de tâches localisées. Ce jeu de données contient 82283 clés pour un volume de 8 Go. Le job exécuté sur ces données consiste à appliquer l'opération classique du calcul de la moyenne des valeurs associées à une clé. Ce job est réalisée par 8 *mappers* et 16 *reducers*.

9.3 Réallocation dynamique des tâches

Les expériences suivantes cherchent à déterminer s'il est bénéfique de réallouer les tâches au cours du processus d'exécution.

9.3.1 Impact de la réallocation sur l'équité de contribution

Dans cette section, les capacités de calcul des *reducers* sont supposées homogènes. Chaque *reducer* est déployé sur un nœud de calcul Intel (R) Core (TM) i5 3,30 GHz avec 4 cœurs et 8 Go de RAM. Les agents traitent le jeu de données Météo France et ils utilisent la stratégie de sélection de tâches k-éligible avec $\texttt{k} = 2$.

Afin de vérifier l'efficacité du processus de réallocation pour les jobs `EnrPrTemp` et `PluPrStat`, le tableau 9.1 compare les équités de contribution suite à l'exécution de :

- l'allocation de tâches donnée par la fonction de partition par défaut, que nous appellerons **allocation par défaut**. Pour rappel, si l'on considère m *reducers*, cette fonction alloue une tâche τ au reducer i si

$$\tau\texttt{.hashCode() \% m = i;}$$

- l'allocation qui, à partir de l'allocation par défaut, a été corrigée par le processus de réallocation à l'aide des délégations socialement rationnelles présenté dans cette thèse ;

- l'allocation donnée par l'heuristique LPT (*Longest Processing Time*) qui donne les m tâches les plus coûteuses aux m *reducers* puis, assigne la prochaine tâche la plus coûteuse au *reducer* le moins chargé [Graham 1969] (cf. chapitre 1). Il est important de noter que cet algorithme n'est pas applicable après la phase de *map* car nous n'avons alors aucune connaissance sur la forme des tâches de *reduce*. LPT donne ici une indication d'une allocation équilibrée. Cependant, cette allocation ne peut être calculée qu'*a posteriori*, quand le job est terminé et que toutes les tâches ont été identifiées. Elle ne peut donc pas servir d'allocation initiale aux tâches.

	Allocation par défaut	**Allocation MAS4Data**	**Allocation LPT**
`EnrPrTemp`	0	0.82	0.98
`PluPrStat`	0.23	0.87	0.97

TABLEAU 9.1 : Pour les jobs `EnrPrTemp` et `PluPrStat`, équités de contribution des allocations données (a) par la fonction de partition par défaut, (b) par le processus de réallocation et (c) par l'algorithme LPT.

Constatons que le processus de réallocation permet d'améliorer significativement l'équité de l'allocation par défaut. Les allocations LPT détiennent les meilleures équités car l'algorithme dispose des informations sur toutes les tâches et les répartit de manière centralisée. À l'inverse, les allocations issues de MAS4Data sont les résultats de plusieurs ajustements locaux réalisés à l'aide de délégations socialement rationnelles.

Le gain d'équité qu'offre le processus de réallocation permet d'améliorer les temps d'exécution des phases de *reduce* (c'est-à-dire le *makespan* de l'allocation des tâches de *reduce*). En effet, j'ai mesuré un taux d'accélération de $3,67$ (resp. $1,24$) entre le temps d'exécution de l'allocation corrigée par les agents MAS4Data et celui de l'allocation par défaut pour le job `EnrPrTemp` (resp. `PluPrStat`). Pour ces deux jobs, on peut donc affirmer qu'il est bénéfique que les agents *reducers* réallouent les tâches lors de la phase de *reduce*. Le fait que la réallocation des tâches améliore le temps d'exécution de la phase de *reduce* permet également de s'assurer que le coût de la négociation est inférieur aux bénéfices de la réallocation des tâches.

Les figures 9.1 et 9.2 montrent les contributions des *reducers* avec l'allocation de tâches par défaut et l'allocation de tâche corrigée pour chacun des deux jobs. À gauche, la fonction de partition par défaut donne une allocation de tâche inéquitable dans laquelle certains *reducers* sont bien plus chargés que d'autres [1]. À droite, nous pouvons observer que la négociation équilibre la charge de travail entre les *reducers* car les tâches ont été dynamiquement réallouées au cours de la phase de *reduce*. Pour rappel, ces observations sont faites *ex post*, c'est-à-dire une fois le job terminé. Elles correspondent donc au bilan du travail réalisé par chaque agent au cours du job.

Dans le cas de la figure 9.2 on constate que même si l'allocation par défaut est plutôt équilibrée, le processus de réallocation autorise tout de même une amélioration.

9.3.2 Bénéfices de la réallocation sur le temps d'exécution

Les expériences suivantes ont été réalisées sur le jeu de données Yahoo! Auction avec le job `CountByKeyword`. Pour mémoire, nous considérons ici 10 agents *reducers* qui utilisent la stratégie

1. L'allocation de gauche sur la figure 9.1 correspond à la situation déjà décrite dans l'exemple 3.4.

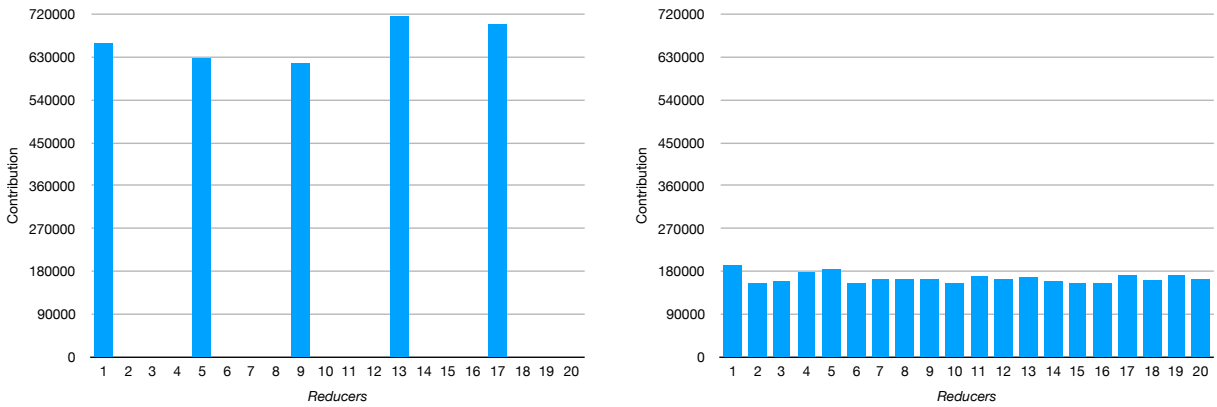


FIGURE 9.1 : Contributions des *reducers* pour l'allocation par défaut (à gauche) et pour l'allocation corrigée par MAS4Data (à droite) lors du job `EnrPrTemp`.

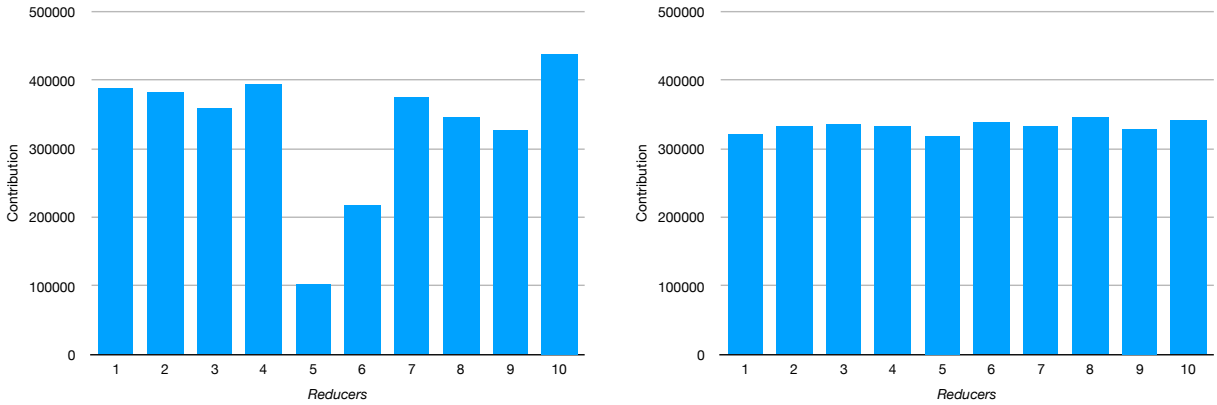


FIGURE 9.2 : Contributions des *reducers* pour l'allocation par défaut (à gauche) et pour l'allocation corrigée par MAS4Data (à droite) lors du job `PluPrStat`.

de sélection de tâche k-éligible ($\mathtt{k} = 2$).

Pour confirmer qu'équilibrer les charges de travail entre les *reducers* diminue le temps d'exécution de la phase de *reduce*, je compare ces temps pour l'allocation de tâches par défaut et pour l'allocation corrigée par MAS4Data. J'étudie ici les temps d'exécutions du job en fonction du nombre de nœuds de calcul utilisés. Le nombre de *reducers* ne varie pas, ils sont répartis selon le nombre de nœuds disponibles. Chaque nœud possède la configuration matérielle suivante : Intel (R) Core (TM) i5 3,30 GHz avec 4 cœurs et 8 Go de RAM. Pour chaque nombre de nœuds utilisés, 3 exécutions ont été effectuées. Comme l'écart-type dû au non déterminisme du processus de réallocation est faible, je compare la moyenne des temps d'exécution.

La figure 9.3 présente le temps d'exécution des différentes phases du job. La phase de *map* est la même pour l'exécution classique et pour l'exécution MAS4Data. Sont ensuite représentées les temps d'exécution de la phase de *reduce* lorsque l'allocation de tâches est celle par défaut, et lorsque que l'allocation est continuellement remise en cause par les agents MASTA. La figure 9.4 met en évidence les équités de temps associées aux exécutions correspondantes.

Nous observons que le temps d'exécution de la phase de *map* diminue avec le nombre de nœuds puisqu'elle bénéficie du parallélisme. Quelle que soit l'approche retenue, le temps d'exécution de la phase de *reduce* diminue globalement. Cette baisse n'est pas parfaitement proportionnelle au nombre de machines puisque la phase de *reduce* est pénalisée par la non-localité des *chunks* : plus il y a de nœuds, plus un *reducer* est susceptible de devoir récupérer des *chunks* distants. De plus, l'approche classique est pénalisée par le biais de partitionnement. Comme elle ne s'adapte pas au partitionnement déséquilibré, cette approche ne comble pas l'écart entre l'effort fait par le *reducer* le plus chargé et ceux qui le sont moins. Dans l'approche classique, l'équité de temps reste proche de 0,5 (cf. figure 9.4) : le *reducer* le moins chargé travaille environ 50 % de moins que le plus chargée. À l'inverse, pour l'approche adaptative, l'équité reste élevée et le temps d'exécution est systématiquement plus faible. Ce gain de temps s'explique par la réallocation continue des tâches qui permet une meilleure exploitation des ressources disponibles.

Ces deux expériences témoignent de l'intérêt du processus de réallocation. En effet, ce dernier permet de corriger une allocation mal équilibrée, et par conséquent de réduire le temps d'exécution de l'ensemble des tâches à traiter.

9.4 Dynamique du processus de réallocation

Les expériences suivantes cherchent à mettre en évidence la dynamique du processus de réallocation en analysant l'évolution des charges de travail maximale et minimale tout au long d'une exécution.

Le jeu de données traité est Yahoo ! Music grâce au job `MusicEval`. Les 12 agents *reducers* sont homogènes. Chacun d'eux se trouve sur un nœud Intel (R) Core (TM) i5 3,30 GHz PC avec 4 cœurs et 8 Go de RAM. Les agents utilisent la stratégie de sélection de tâche k-éligible ($\mathtt{k} = 2$) et la découpe de tâche. La fonction de coût utilisée est la fonction de coût par valeurs : le coût d'une tâche est égal au nombre de valeurs que le *reducer* doit traiter pour produire son résultat.

La figure 9.5 présente l'évolution des charges de travail maximale et minimale du système pendant la phase de *reduce*. Ainsi, la courbe rouge représente la charge maximale, la courbe verte

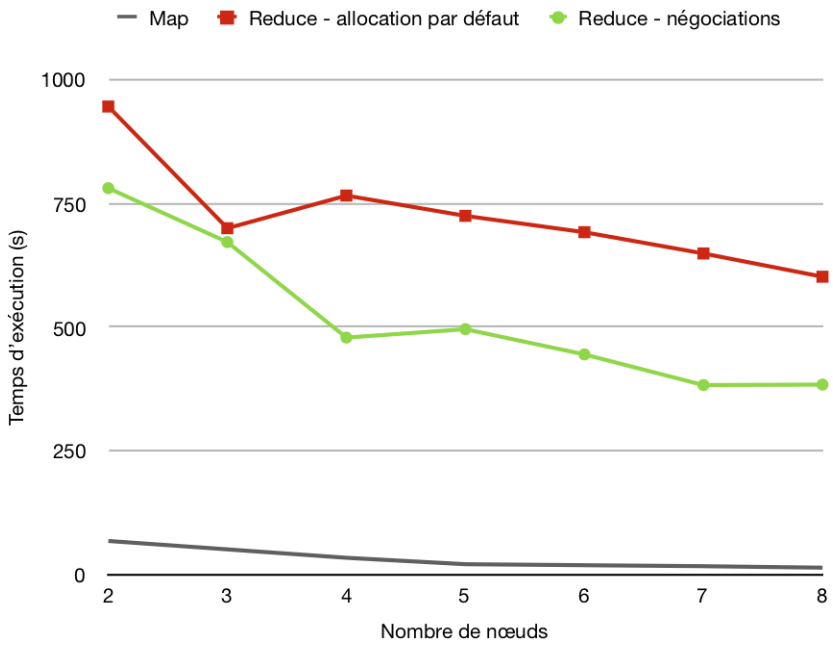


FIGURE 9.3 : Temps d'exécution moyens des différentes phases du job `CountByKeyword` en fonction du nombre de nœuds utilisés.

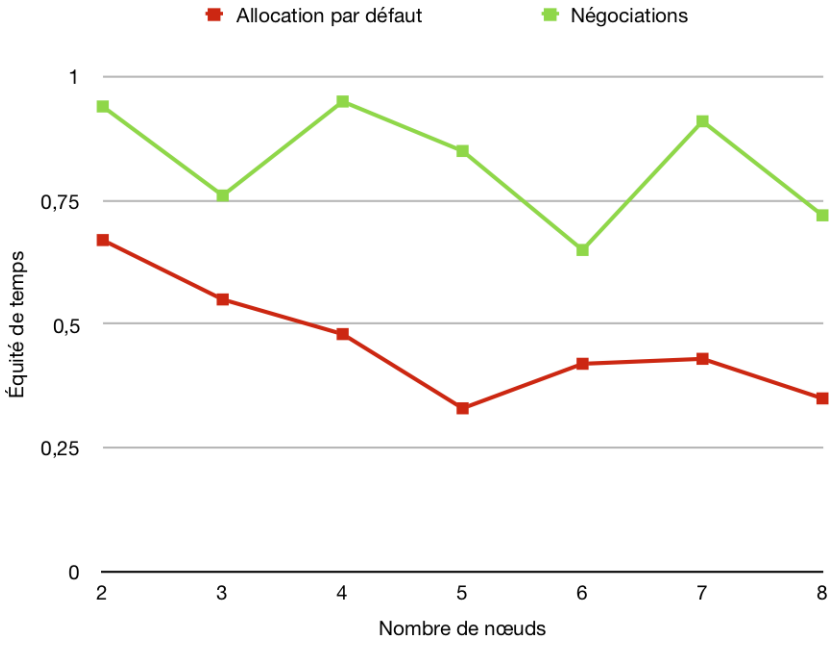


FIGURE 9.4 : Équité de temps des différentes exécutions du job `CountByKeyword` en fonction du nombre de nœuds utilisés.

représente la charge minimale et la courbe bleue représente la charge moyenne de l'ensemble des *reducers*. L'équité de charge, c'est-à-dire le rapport entre la charge minimale et la charge maximale du système, est également indiquée en gris [2].

La négociation des tâches entre les agents comble rapidement la différence entre la charge du *reducer* le plus chargé et celle du *reducer* le moins chargé. Ainsi, les délégations socialement rationnelles permettent de maintenir les charges de travail à un niveau similaire tout au long de l'exécution. De plus, la figure présente cette dynamique lorsque les agents sont individuellement engagés dans une seule enchère à la fois (à gauche), et lorsque, comme présenté dans le chapitre 5, ils sont engagés dans plusieurs enchères simultanément (à droite). Alors que le processus d'enchère unique exige 58 secondes pour atteindre une équité de charge supérieur à 0,70, le processus d'enchère multiples n'a besoin que de 3 secondes. Étant donné que le processus d'enchères multiples améliore la réactivité des agents, la phase de *reduce* avec un processus de multi-enchères est plus rapide de 33 secondes, soit une amélioration d'environ 12 %.

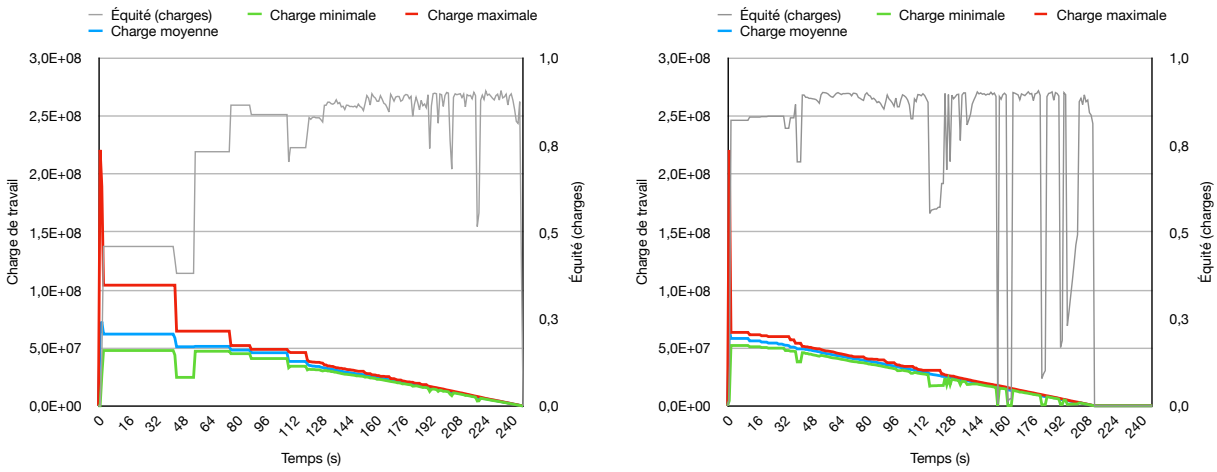


FIGURE 9.5 : Dynamique de l'évolution des charges de travail avec un processus de mono-enchère (à gauche) et de multi-enchères (à droite) pour le job `MusicEval`.

9.5 Hétérogénéité des nœuds de calcul

Grâce aux expériences présentées ensuite, je souhaite mettre en évidence le fait que la co-occurence du processus de réallocation et de l'exécution des tâches permet d'adapter l'allocation des tâches aux performances des agents.

Lors de ces expériences, les agents adoptent la stratégie de sélection de tâches k-éligible ($\mathtt{k} = 2$) et sont déployés de manière à être hétérogènes en terme de capacités de calcul. Cependant, cette hétérogénéité n'est pas explicitement représentée dans la fonction de coût. En d'autre termes, la fonction de coût est définie de manière générique, *a priori*, et sans connaissance préalable sur les capacités de calcul de chacun des nœuds. Ainsi, la fonction de coût utilisée reste la fonction de

2. Lorsque ces expériences ont été réalisées, je cherchais également à valider l'idée selon laquelle il n'est pas nécessaire de réallouer une tâche si le gain d'équité est inférieur à un certain seuil (ici, 0,1). Sans résultat supplémentaire, je ne présente pas ces expériences dans ce chapitre. Cependant, cela explique pourquoi l'équité de charge plafonne à 0,9.

coût par valeurs avec laquelle la coût d'une tâche est le même pour tous les agents du système. Pour vérifier que le processus de réallocation est adaptatif aux capacités des agents, observons leurs contributions, qui doivent être moins importantes pour les agents qui possèdent le moins de puissance de calcul.

Dans un premier temps, examinons les contributions des agents lorsqu'ils exécutent le job `CountByKeyword` sur deux nœuds hétérogènes : un PC Intel (R) Core (TM) i7 2,8 GHz avec 8 cœurs et 16 Go de RAM et un PC Intel (R) Core (TM) i5 3,3 GHz avec 4 cœurs et 8 Go de RAM. 5 *reducers* sont sur chacun de ces nœuds. La figure 9.6 présente les contributions des *reducers* sans réallocation (à gauche) et avec réallocation (à droite). Les 5 premiers *reducers* se trouvent sur le nœud le plus puissant et possèdent une contribution plus élevée. On peut constater que la réallocation des tâches permet d'adapter les charges de travail aux compétences de calcul des *reducers*.

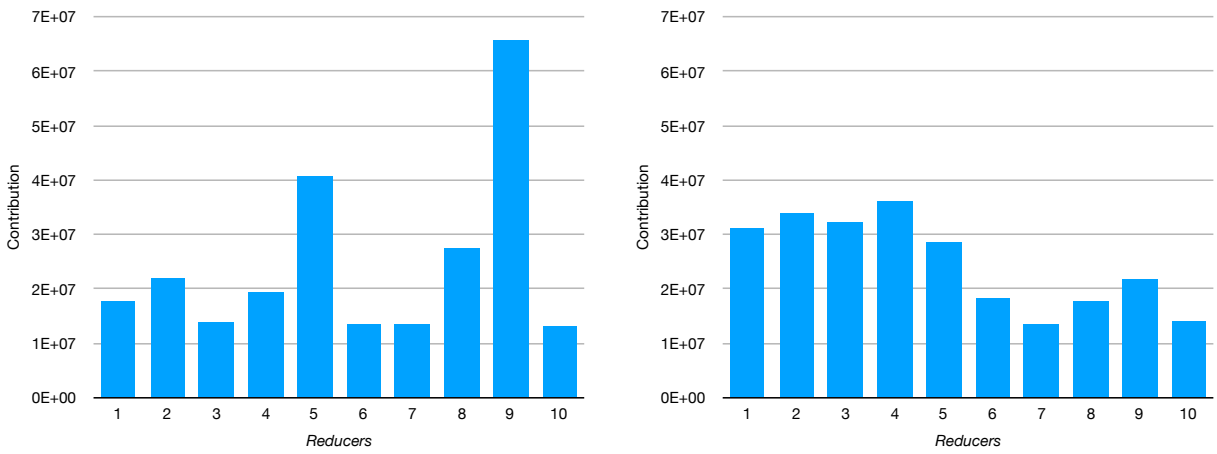


FIGURE 9.6 : Contributions des *reducers* pour une exécution MapReduce classique (à gauche) et une exécution MAS4Data (à droite) du job `CountByKeyword`. Les nœuds sont hétérogènes : les cinq *reducers* de gauche sont plus rapides que les cinq *reducers* de droite.

La figure 9.7 présente les contributions de 12 *reducers* qui exécutent le job `MusicEval`. L'environnement est hétérogène sur la figure de droite ; il est homogène sur la figure de gauche (pour comparaison). Cette fois, l'environnement hétérogène est constitué de sorte que 6 *reducers* aient chacun un nœud et que les 6 autres soient ensemble sur un septième et dernier nœud. On s'attend de nouveau à observer une contribution plus élevée pour les agents ayant bénéficier du plus de ressources de calcul.

Là encore, dans un environnement hétérogène, la réallocation dynamique des tâches adapte la répartition de la charge de travail en fonction des performances des agents. Je souligne également que l'équité de temps de l'allocation est de $0,99$, ce qui indique que les *reducers* terminent leur travail presque simultanément. Cela montre que la répartition de la charge de travail est efficace et qu'elle prend en compte les différences de capacités de calcul.

9.6 Découpe de tâche

Les prochaines expériences mettent en lumière l'intérêt de la découpe de tâche (cf. section 6.3) dans deux situations différentes.

Lorsque les tâches sont divisibles, la découpe peut d'abord servir pour parfaire une allocation stable [3]. Pour illustrer ce point, observons les temps d'exécution et les équités de temps lorsque les agents MAS4Data appliquent la découpe de tâche dans les mêmes conditions qu'en 9.3.2 (c'est-à-dire 10 agents qui réalisent le job `CountByKeyword` et utilisent la stratégie de sélection k-éligible). Pour cela, les figures 9.9 et 9.8 ajoutent une nouvelle courbe qui correspond à l'utilisation de la découpe de tâche. On y constate que la découpe permet d'améliorer l'équité de temps, ce qui indique un meilleur équilibre des charges (cf. figure 9.8). De plus, ce meilleur équilibre des charges se traduit par un meilleur temps d'exécution (cf. figure 9.9). Le temps d'exécution de la phase de *reduce* n'est pas pénalisé par les découpes de tâches car celles-ci ne sont réalisées que si nécessaire. Une découpe est menée uniquement si l'allocation est stable et que la découpe permet d'améliorer le *makespan* local de deux agents.

Dans la seconde situation, il est nécessaire de diviser les tâches car l'allocation initiale ne profite pas des ressources à disposition. Comme expliqué dans la section 9.2, le job `MusicEval` demande le traitement de seulement 5 tâches. De ce fait, un maximum de 5 *reducers* sont utilisés par la fonction de partition. Nous réalisons le job sur plus de 5 nœuds. Afin de profiter de l'ensemble des ressources de calcul et ainsi améliorer le temps d'exécution du job, les agents *reducers* qui recoivent initialement les tâches peuvent les découper pour les négocier avec leurs pairs. La figure 9.10 présente les temps d'exécutions du job `MusicEval` pour un nombre de nœuds compris entre 5 et 12. Les conditions d'exécution sont les mêmes qu'en 9.4 (c'est-à-dire 12 agents qui utilisent la stratégie de sélection de tâche k-éligible et la découpe de tâches). Cette figure montre que la découpe de tâche permet ici de profiter de l'ensemble des ressources disponibles et ainsi de faire décroître le temps d'exécution du job. On remarque en particulier que même avec 5 nœuds (et donc une tâche par nœud), les tâches représentent une différence de charges suffisamment importante pour que la découpe soit utile.

9.7 Comparaison des stratégies de sélection de tâche

Grâce à ces dernières expériences, je cherche à montrer qu'il est préférable pour un agent de prendre en compte la localité des ressources liées à une tâche lorsqu'il considère une délégation ou une exécution. En d'autres termes, je souhaite mettre en évidence le bien-fondé de la stratégie de sélection de tâches localisées.

Les exécutions suivantes ont été réalisées sur 16 PCs quadricœurs Intel(R) i7 avec 16 Go de RAM chacun. Huit nœuds servent durant la phase de *map*, 16 servent durant la phase de *reduce*. Ainsi, les *chunks* générés par un *mapper* sont locaux pour le *reducer* du même nœud, et huit *reducers* ne possèdent aucune ressource. Les données utilisées ont été générées de sorte que l'allocation initiale des tâches (cf. figure 9.12) soit déséquilibrée et de manière à répartir les clés inégalement parmi les *mappers*. Ainsi, tous les profils de tâches sont représentés. Certaines

3. Pour rappel, une allocation stable est une allocation qui ne peut plus être améliorée à l'aide d'une délégation socialement rationnelle.

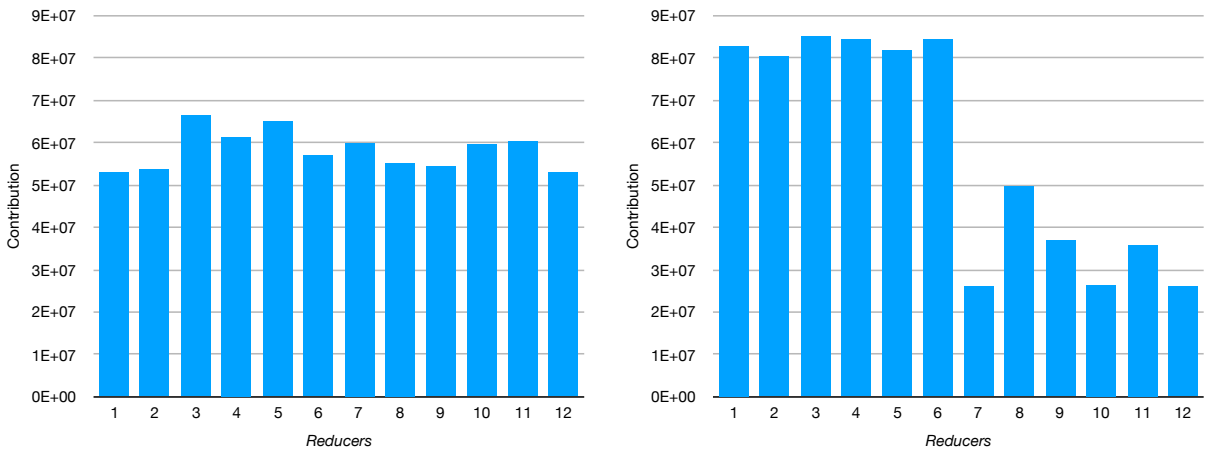


FIGURE 9.7 : Contributions des *reducers* pour une exécution MAS4Data du job `MusicEval` dans un environnement homogène (à gauche) et dans un environnement hétérogène (à droite).

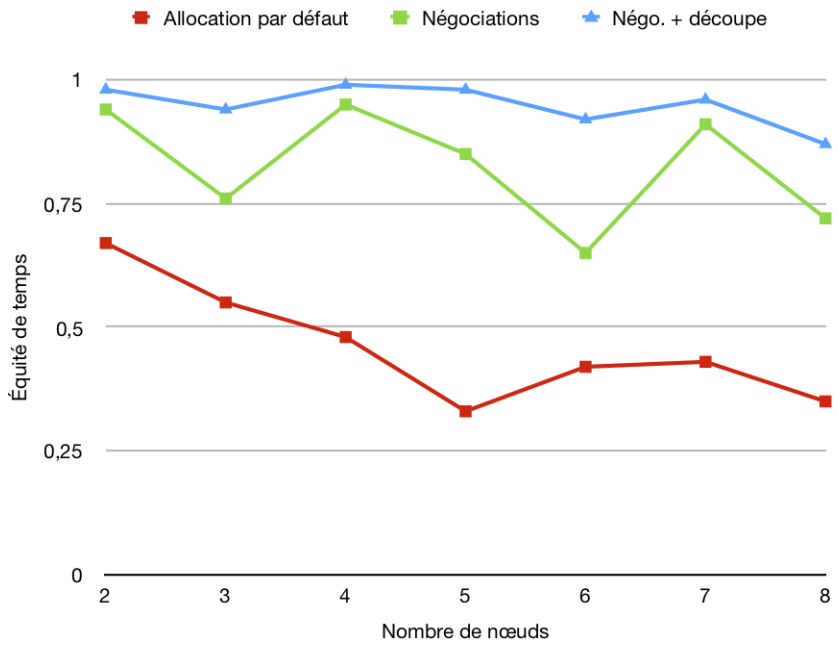


FIGURE 9.8 : Équité de temps moyenne des différentes exécutions du job `CountByKeyword` en fonction du nombre de nœuds utilisés.

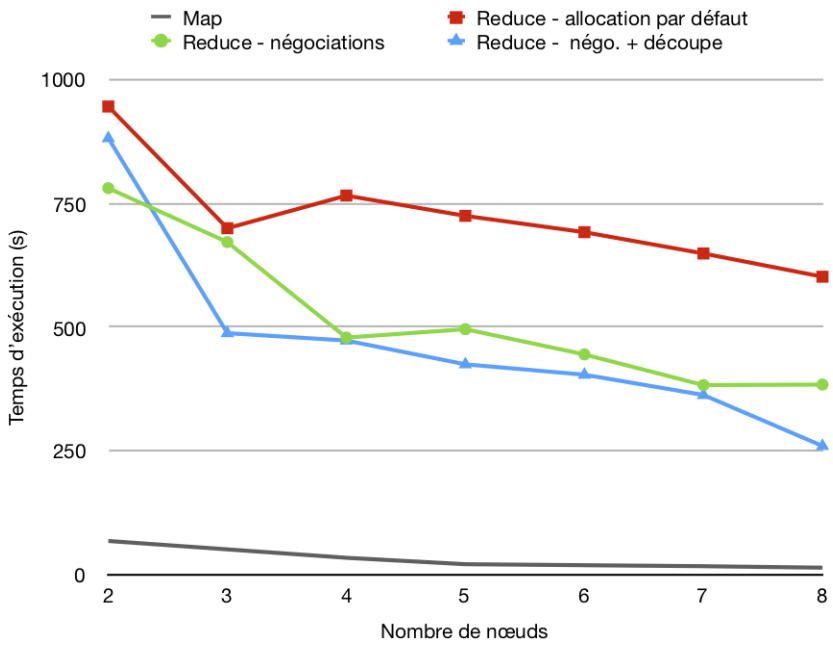


FIGURE 9.9 : Temps d'exécution moyens des différentes phases du job `CountByKeyword` en fonction du nombre de nœuds utilisés.

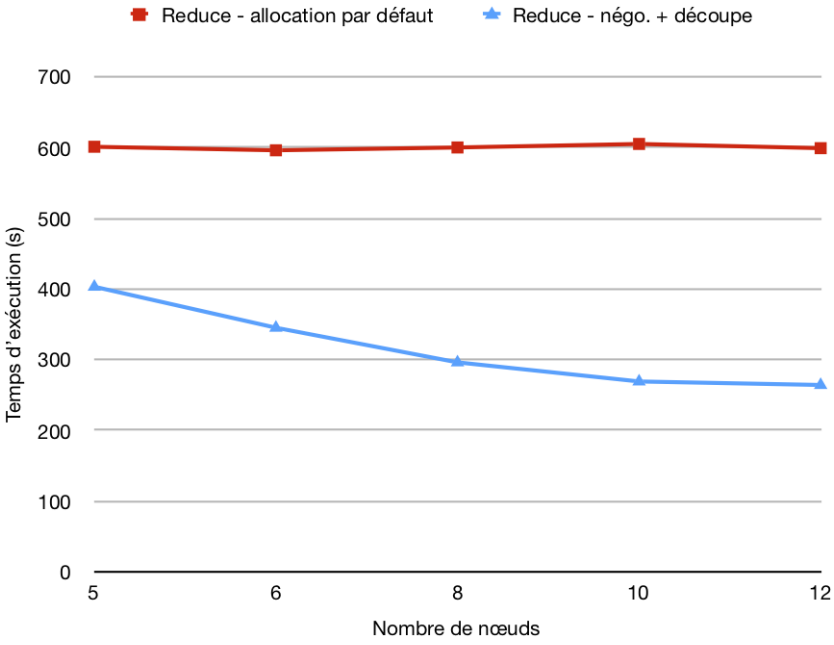


FIGURE 9.10 : Temps d'exécution moyens de la phase de *reduce* du job `MusicEval` en fonction du nombre de nœuds utilisés.

possèdent des *chunks* équitablement répartis sur tous les nœuds, d'autres sont fortement locales pour un *reducer* et donc fortement distantes pour ses pairs.

Par la suite, les agents appliquent la stratégie de sélection de tâches localisées (SSTL) ou une stratégie qui ne considère pas la localité des *chunks*, en l'occurence la stratégie de sélection de tâche par coûts ordonnés CS-DB (cf. chapitre 6). Ici, il est inenvisageable d'utiliser une stratégie de sélection de tâche k-éligible car les coûts des tâches sont différents pour chaque agents et que le nombre de tâches est conséquent, ce qui rend trop élevée la complexité de l'algorithme $\Big(O((m-1)\times|P_i|)\Big)$.

Afin de faire en sorte que le coût d'une tâche tienne compte de sa localité, les agents utilisent la fonction qui donne le coût de la tâche τ pour l'agent i de la manière suivante :

$$c_i(\tau) = \sum_{\rho\in[\rho_1^\tau;\rho_k^\tau]} c_i(\rho), \text{ avec } c_i(\rho) = \begin{cases} |\rho| \text{ si } l_\rho = l_i \\ \kappa \times |\rho| \text{ sinon} \end{cases} \tag{9.1}$$

Avec $k = |\mathcal{R}_\tau|$ le nombre de *chunks* nécessaire à l'exécution de la tâche τ, $|\rho|$ le nombre de valeurs que contient le *chunk* ρ et l_ρ sa localité. κ est un multiplicateur qui rend l'utilisation d'un *chunk* distant plus coûteuse. Lors des exécutions, la valeur de κ est arbitrairement fixée à $\kappa = 10$. Comme discuté dans la section 4.5, la définition d'une fonction de coût précise est un problème compliqué en pratique mais la réallocation dynamique et adaptative des tâches permet de compenser cette approximation.

À partir de dix exécutions, comparons les temps d'exécution médians lorsque les agents utilisent ou non la stratégie de sélection de tâches localisées (cf. figure 9.11). Nous observons que la stratégie de sélection de tâches localisées améliore significativement le temps d'exécution, d'environ $7,6\%$. Pour comparaison, le temps d'exécution de l'allocation par défaut sans réallocation est de $853s$ (environ $+100\%$). On observe de nouveau que la négociation permet de diminuer le temps d'exécution et que l'utilisation de la stratégie de sélection de tâches localisées est bénéfique.

Intéressons-nous maintenant aux contributions des *reducers*. En raison du non-déterminisme de l'exécution distribuée, je présente une exécution typique. La figure 9.12 compare les contributions des 16 *reducers* selon qu'ils aient utilisé la stratégie CS-DB ou la stratégie de sélection de tâches localisées. On peut remarquer que le *makespan* de l'allocation initiale vaut approximativement $3,3\times10^8$, qu'il est de $2,5\times10^8$ (-24%) en cas de négociation avec la stratégie CS-DB et de 2×10^8 (-30.7%) avec la stratégie de sélection de tâches localisées. L'usage de la stratégie de sélection de tâches localisées, permet d'améliorer l'équilibre des charges.

La figure 9.13 présente le nombre de tâches τ telles que $\alpha \leq o_i(\tau) < \alpha + 0,1$ quand l'agent i réalise τ. L'allocation initiale de tâches est mal équilibrée car il y a plus de $4,6\times10^4$ tâches pour lesquelles $o_i(\tau) = 0$. La stratégie de sélection de tâches localisées favorise le traitement des tâches qui sont les plus locales, i.e. $o_i(\tau) \geq 0,5$. Ce n'est pas le cas de la stratégie CS-DB. Par exemple, il y a 171 tâches qui ont été traitées par un agent qui possède 60 % des *chunks* (i.e. des ressources) avec la stratégie de sélection de tâches localisées, mais aucune avec la stratégie CS-DB. Puisqu'elle favorise l'exécution d'une tâche par un agent qui possède localement les ressources nécessaires, la stratégie de sélection de tâches localisées améliore l'équilibre des charges et diminue le temps d'exécution.

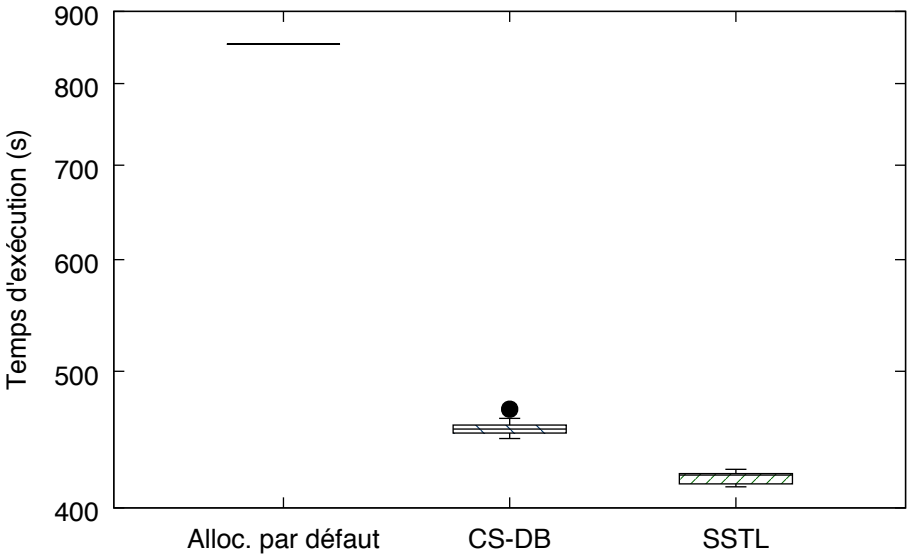


FIGURE 9.11 : Temps d'exécution médian de la phase de *reduce* pour l'allocation par défaut, pour l'allocation corrigée par des agents utilisant la stratégie de sélection de tâche CS-DB et pour l'allocation corrigée par des agents utilisant la stratégie de sélection de tâches localisées (SSTL).

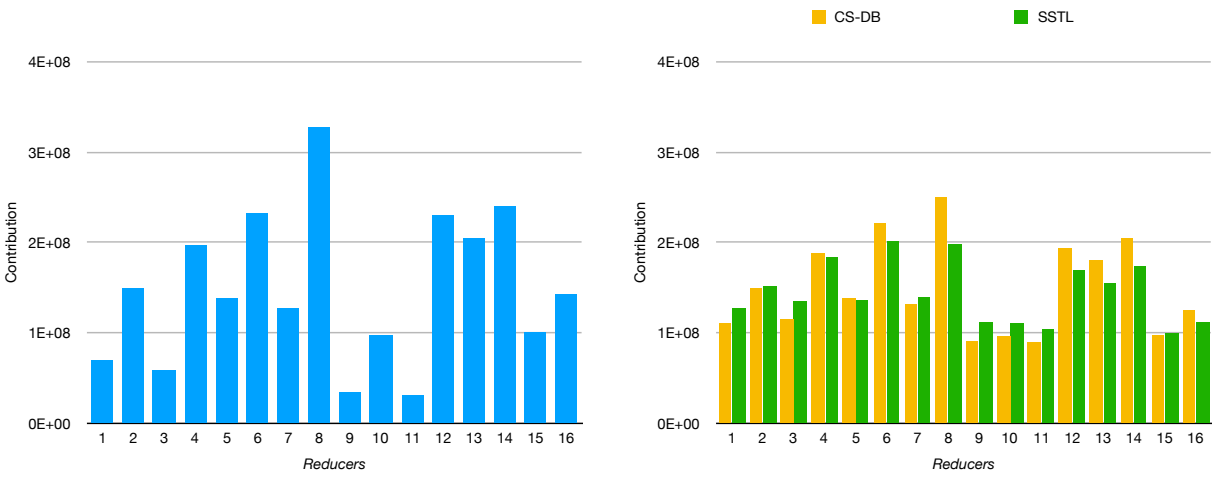


FIGURE 9.12 : Contributions des *reducers* pour l'allocation par défaut (à gauche) et pour l'allocation corrigée par MAS4Data (à droite) avec la stratégie de sélection de tâche CS-DB ou la stratégie de sélection de tâches localisées (SSTL).

9.8 Conclusion

9.8.1 Synthèse

Ce chapitre présente quelques-unes des expérimentations réalisées avec le prototype MAS4Data. Ces expériences ont été menées dans de multiples configurations matérielles sur trois jeux de données réelles et un jeu de données générées. Deux fonctions de coûts ont été utilisées. La fonction de coût par valeurs associe le coût d'une tâche au nombre de valeurs à traiter pour produire son résultat. Cette fonction est indifférente aux capacités de calcul des nœuds et à la localité des *chunks* mais elle permet d'utiliser la stratégie de sélection de tâches k-éligible et la stratégie de découpe de tâche décrites dans le chapitre 6. À l'inverse, la fonction de coût basée sur la localité considère que l'utilisation d'un *chunk* distant est plus coûteux que l'utilisation d'un *chunk* local. De ce fait, le coût des tâches dépend de la localité des *chunks* et les tâches sont susceptibles d'avoir un coût différent pour chaque agent du système. Cette fonction traduit l'hypothèse selon laquelle l'utilisation d'une ressource distante représente un surcoût. Elle autorise également l'utilisation de la stratégie de sélection de tâches localisées qui privilégie l'exécution des tâches coûteuses et locales et la délégation des tâches coûteuses et distantes.

Lors de l'analyse des résultats des différentes expériences, nous avons constaté que :

- la réallocation des tâches permet une meilleure répartition des charges de travail et d'adapter cette répartition aux performances des nœuds de *reduce* (cf. sections 9.3 et 9.5) ;
- les agents ajustent l'allocation de tâches tout au long de l'exécution afin de faire diminuer la charge de travail la plus élevée du système (cf. section 9.4) ;
- la stratégie de découpe de tâche permet de continuer à améliorer une allocation de tâches stable et de contrer le biais des clés coûteuses (cf. section 9.6) ;
- lors des négociations, il est préférable que les agents considèrent la localité des *chunks*, et donc qu'ils utilisent la stratégie de sélection tâches localisées (cf. section 9.7).

Notre objectif est de réduire le *makespan* d'une allocation de tâches grâce à des agents coopératifs qui négocient à l'aide de décisions locales. Au niveau d'une exécution MapReduce, cela se traduit par une diminution du temps d'exécution de la phase de *reduce* sans superviseur centralisé. À cet effet, au regard des différences de temps d'exécution, ces expériences nous apprennent qu'une fonction de coût valide et une stratégie de sélection de tâches pertinente ajoute à l'efficacité du mécanisme de réallocation des tâches. Elles nous apprennent également que le coût de la réallocation est systématiquement inférieur au gain qu'elle offre car le temps d'exécution est plus faible dès lors que les agents utilisent les délégations socialement rationnelles.

Enfin ce chapitre confirme l'intérêt d'appliquer la proposition de cette thèse au patron de conception MapReduce. En plus de proposer une solution aux deux biais de la phase de *reduce* (cf. chapitre 3), la réallocation dynamique et la découpe des tâches permettent d'ajuster l'allocation des tâches pendant la phase de *reduce*. Cela autorise l'utilisation de nœuds hétérogènes sans qu'on ait besoin de quantifier ou de spécifier les différences de performances. Mais, cela permet également de s'adapter à de potentielles (et imprévisibles) variations de performances des nœuds de calcul.

9.8.2 Perspectives

Beaucoup de paramètres entrent en jeux lorsqu'il s'agit de se prononcer sur l'efficacité du processus de réallocation : les données à traiter, le job à exécuter, les nœuds de calcul utilisés, la répartition des clés dans les fragments des *mappers*, l'allocation initiale offerte par la fonction de partition, la stratégie de sélection de tâches des agents, la précision de la fonction de coût, etc. En ce sens, les expériences menées dans ce chapitre ne représentent qu'un premier pas vers la validation expérimentale complète du modèle de réallocation proposé dans cette thèse. De nombreux autres cadres d'expériences sont à considérer, par exemple en évaluant l'impact de chaque couple (stratégie de sélection de tâche, fonction de coût).

Si l'on se concentre sur l'adaptation de l'allocation de tâche à l'hétérogénéité des nœuds de calcul, une perspective serait de provoquer cette hétérogénéité pendant la phase de *reduce*. Dans ce chapitre, les agents possèdent des capacités de calcul différentes dès le début de l'exécution. Ces différences de capacités ne sont pas explicitées et c'est bien le processus de réallocation qui adapte l'allocation aux performances des agents. Cependant, il serait intéressant d'observer la réaction du système multi-agents et le temps d'ajustement nécessaire pour retrouver une équité de charge intéressante dans le cadre de baisses de performances scénarisées.

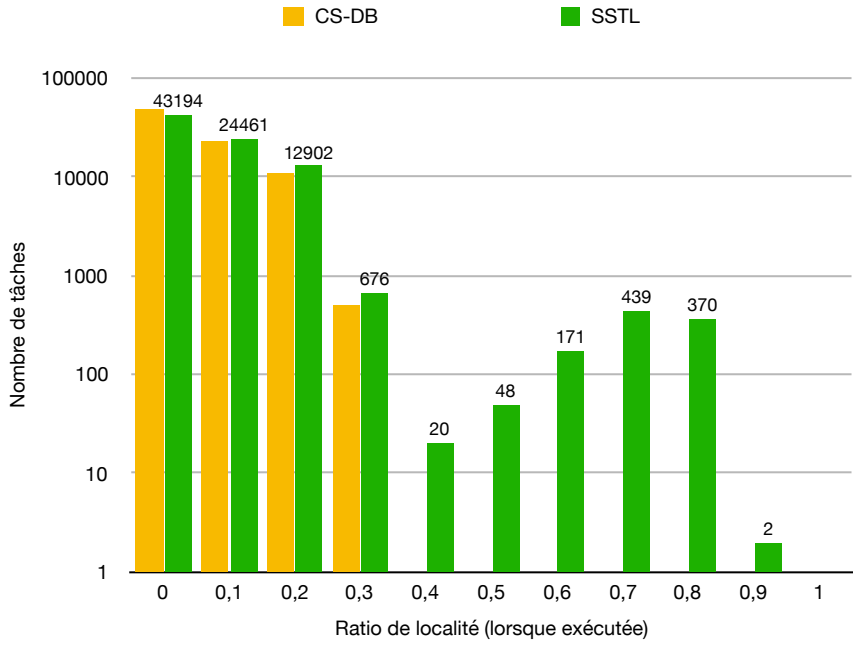


FIGURE 9.13 : Nombre de tâches en fonction du ratio de localité de l'agent exécutant (échelle logarithmique).

Conclusion

Dans ce document, je défends la thèse selon laquelle, dans le cadre d'un système multi-agents collaboratif et dans l'objectif de faire décroître le temps d'exécution d'un ensemble de tâches sujettes à perturbations, il est approprié de réallouer ces tâches dynamiquement et de façon décentralisée grâce à des décisions locales.

Le problème

Dans cette thèse, je me suis intéressé au problème de réallocation de tâches indépendantes, non divisibles et non préemptives. J'ai supposé que les ressources nécessaires à l'exécution des tâches sont partageables (c'est-à-dire qu'elles peuvent être transférées d'un agent à un autre) et qu'elles sont en nombre suffisant dans le système pour que toutes les tâches puissent être exécutées. J'ai également supposé que le système est fiable : tous les agents sont en mesure d'effectuer toutes les tâches et aucun des agents ne peut échouer.

J'ai donc proposé un mécanisme qui permet d'améliorer une allocation de n tâches à m agents hétérogènes. L'hétérogénéité des agents découle de deux faits :

1. les agents sont susceptibles d'avoir des capacités de calculs différentes ;
2. les ressources sont inégalement réparties et la collecte des ressources est coûteuse. Ainsi même si deux agents sont homogènes en terme de capacité de calcul, si l'un possède plus de ressources nécessaires à l'exécution d'une tâche que l'autre, alors la tâche sera moins coûteuse pour le premier que le second.

En tenant compte de ces observations, l'objectif est de redistribuer les tâches aux agents de sorte à minimiser le *makespan* (c'est-à-dire le temps d'exécution maximum du système). Ce problème est noté $R_m||C_{max}$ et il est connu pour être NP-difficile.

L'approche

L'approche que j'ai défendue est dynamique et décentralisée. Elle ne demande aucune connaissance préalable, ni aucun paramétrage spécifique aux tâches. Les agents sont collaboratifs et constituent un système distribué complet. Ils corrigent l'allocation tout au long de l'exécution des tâches pour équilibrer les charges de travail et ainsi améliorer le *makespan* de l'allocation. Ces corrections se font grâce à des ajustements locaux, résultats de négociations entre agents. Une telle réallocation dynamique et continue permet également de s'adapter aux variations de performances des agents et d'aligner leurs charges de travail avec leurs capacités de calculs en temps réel. Ainsi, la proposition défendue dans cette thèse permet de corriger une allocation de

tâches biaisée alors que les tâches sont en cours d'exécution afin d'améliorer le temps d'exécution de l'ensemble des tâches.

Les réponses proposées

Le formalisme MASTA

Le formalisme MASTA permet de représenter un problème de réallocation de tâches dans lequel les ressources et les agents sont situés sur plusieurs nœuds de calculs. Une instance MASTA est constituée d'un système distribué, d'un ensemble de tâches à exécuter, d'une allocation de ces tâches, et d'une fonction de coût qui donne une estimation du coût de l'exécution d'une tâche sur un nœud du système. Pour mesurer la charge de travail des agents, une représentation quantitative et cardinale des préférences est utilisée : chaque tâche représente un coût différent pour chaque agent. Le coût d'une tâche pour un agent dépend du volume intrinsèque de la tâche mais aussi de la localité des ressources nécessaires à son exécution.

Dans le but d'améliorer le *makespan* de l'allocation courante, les agents utilisent les délégations de tâches socialement rationnelles et consomment les tâches de manière concurrente. Une délégation de tâche socialement rationnelle est le don d'une tâche d'un agent à un autre qui permet de rééquilibrer les charges de travail des deux agents et fait donc diminuer leur *makespan* local. Une consommation de tâches se produit lorsqu'un agent a terminé d'exécuter une tâche. Celle-ci est alors supprimée de l'allocation courante et de l'ensemble des tâches. En appliquant itérativement ces opérations, les agents diminuent le temps nécessaire à l'exécution de l'ensemble de tâches alors qu'ils sont en train d'exécuter ces mêmes tâches.

Une négociation multi-agents pour la réallocation dynamique des tâches

Le protocole de négociation utilisé par les agents est le *Contract Net Protocol*. J'ai ajouté au CNP un ensemble de messages d'accusé de réception ainsi que des messages d'interruption temporelle qui rendent le processus de négociation robuste à la perte de messages. Cela a permis de produire des résultats empiriques dans des environnements véritablement distribués. Grâce à ce processus d'interaction, les agent effectuent des délégations de tâche socialement rationnelles pour améliorer le *makespan* de l'allocation de tâches courante. Comme les agents sont collaboratifs et qu'ils partagent le même objectif, ces derniers calquent le modèle de leurs pairs sur eux-mêmes. Pour générer une offre susceptible d'être acceptée, un agent se met à la place de son interlocuteur et se demande s'il accepterait l'offre dans sa situation.

Afin de s'assurer de la dynamicité du processus, les négociations se déroulent de manière concurrente, alors que les agents exécutent les tâches. En tant qu'enchérisseur, un agent peut être impliqué dans plus d'une négociation à la fois. Pour répondre au problème de l'enchérisseur enthousiaste, j'ai mis en place un mécanisme de charge de travail virtuelle. Cela assure qu'un enchérisseur est en mesure de remplir l'ensemble des engagements qu'il tient simultanément. De plus, chaque accord améliore le bien-être égalitaire des agents impliqués et le processus converge vers une solution stable.

Enfin, pour améliorer la complexité communicationnelle, un agent passe en état de pause lorsque,

selon ses croyances ou les réponses de ses pairs, il n'est plus en mesure d'initier une délégation de tâche qui soit socialement rationnelle. Dans cet état, un agent n'initie plus d'enchère.

Des stratégies de sélection de tâche

Afin d'optimiser l'ordre dans lequel ils délèguent/consomment leurs tâches, les agents ont recours à une stratégie de sélection de tâches. Plusieurs stratégies ont été proposées : les stratégies de sélection par coûts ordonnés, les stratégies de sélection de tâches k-éligibles, et enfin la stratégie de sélection de tâches localisées. Cette dernière prend en compte la localité des ressources pour choisir la tâche à déléguer/exécuter. En effet cette stratégie provient du constat que la seule fonction de coût pouvait être insuffisamment précise. Une charge de travail élevée peut signifier qu'un agent possède beaucoup de tâches à réaliser ou que certaines des tâches réclament l'utilisation de ressources dont ne dispose pas l'agent. De plus, même si plusieurs tâches ont le même coût pour un agent, il se peut que ce dernier possède plus de ressources que ses pairs pour l'une d'entre elles. Un agent qui utilise la stratégie de sélection de tâches localisées exécute en priorité les tâches locales et coûteuses et délègue en priorité les tâches distantes et coûteuses.

J'ai également défini une stratégie de découpe de tâche. Une allocation de tâches stable n'est pas nécessairement équilibrée. Si les tâches sont divisibles, un agent surchargé mais, dont aucune des tâches n'est délégable, peut envisager de découper sa tâche la plus coûteuse en sous-tâches délégables afin d'améliorer le *makespan* de l'allocation courante.

Une architecture composite d'agent

Motivé par le fait qu'un même agent possède plusieurs responsabilités (la gestion de son lot de tâches, l'initiation et la participation aux enchères, et l'exécution des tâches), j'ai proposé une structure d'agent composite. Un agent est ainsi composé d'un *manager* qui gère le lot de tâches, d'un *broker* qui s'implique dans les enchères comme initiateur ou enchérisseur et d'un *worker* qui exécute les tâches. Comme les agents ont une connaissance partielle du système, l'état d'un agent composite et de ses agents composants dépend de la représentation qu'ils ont de l'allocation courante et traduit les actions que l'agent peut réaliser selon sa charge de travail et ses croyances. J'ai spécifié l'ensemble des comportements de chaque agent composant à l'aide d'automates finis déterministes. Cette spécification se transpose naturellement lors de l'implémentation des agents.

Une résolution des biais de phase de *reduce* lors d'une exécution MapReduce

Le patron de conception MapReduce est utilisé comme cadre applicatif de ces travaux. En effet, je modélise la phase de *reduce* par une instance MASTA. Ainsi, l'objectif des agents *reducers* est d'exécuter l'ensemble des tâches de *reduce* le plus rapidement possible, c'est-à-dire de minimiser le *makespan*. Après la phase de *map*, les tâches de *reduce* sont allouées aux agents *reducers*. Ces derniers commencent alors à les exécuter. Ce faisant, ils négocient avec leurs pairs pour procéder à des délégations de tâches. Ces délégations de tâches permettent d'améliorer le *makespan* et de traiter le biais de partitionnement. J'ai également traité le biais des clés coûteuses grâce à

une stratégie de découpe de tâche. À ma connaissance, ces travaux sont les seuls à utiliser les propriétés des systèmes multi-agents pour aborder les biais de la phase de *reduce*.

Afin de pouvoir appliquer les concepts décrits dans cette thèse sur des données réelles, j'ai implémenté un prototype distribué appelé MAS4Data. Celui-ci déploie un système multi-agents qui autorise la réallocation des tâches de *reduce* pendant leur exécution grâce aux délégations de tâche socialement rationnelles. Ce prototype m'a permis de mener plusieurs expériences qui montrent que le processus de réallocation est pertinent et que le coût de la négociation est inférieur au gain de temps que procure un bon équilibre des charges de travail.

Perspectives

Cette section propose quelques perspectives pour prolonger les travaux présentés.

Étendre la portée du modèle MASTA

Une première possibilité pour étendre le modèle MASTA est de donner la possibilité aux agents d'exécuter d'autres opérations que les délégations socialement rationnelles. En effet, une allocation stable n'est pas forcément optimale. Pour continuer d'améliorer une allocation stable ou atteindre des allocations non atteignables avec des délégations socialement rationnelles, il faudrait que les agents considèrent d'autres formes d'accords. Une première possibilité est de permettre aux agents d'effectuer des échanges de tâches (ou *swap*). Une seconde possibilité est de permettre aux agents de déléguer plus d'une tâche à la fois. Avec ces nouveaux types d'accords, l'espace des solutions à explorer est plus large mais il est susceptible de contenir des optimums locaux qui possèdent un meilleur *makespan*.

Une seconde perspective est d'étendre le cadre couvert par le formalisme MASTA. Comme dans le cas de l'application pratique, j'ai considéré dans cette thèse des tâches sans date butoir, ni contrainte de précédence. De même, j'ai considéré qu'aucune ressource n'était rare. Le modèle demande à être étendu pour prendre en charge d'autres typologies de tâches, des contraintes différentes sur les ressources ou encore des topologies différentes de systèmes distribués. Étendre le formalisme MASTA pourra également ouvrir d'autres possibilités d'applications pratiques qui seront susceptibles de bénéficier d'une réallocation dynamique et décentralisée de tâches.

Par la suite, on pourra également considérer l'impact qu'aurait l'ajout d'un nouvel agent pendant l'exécution des tâches. J'estime que le processus peut facilement bénéficier de l'ajout d'un agent en cours d'exécution. En effet, grâce à la réallocation continue des tâches, ce nouvel agent pourra rapidement se voir attribuer des tâches et participer à améliorer le *makespan* de l'allocation courante. Pour cela, il suffira à l'agent d'annoncer sa présence à ses pairs avant de se voir participer aux enchères et attribuer des tâches. Le problème ici est plutôt de décider quand et pourquoi ajouter un agent au système.

Si l'on s'en tient à appliquer le modèle MASTA au patron de conception MapReduce, une première extension sera de prendre en charge des jobs concurrents. Dans ce cas, plusieurs ensembles de tâches sont traités simultanément par les agents et un nouvel ensemble de tâches peut entrer dans le système à tout moment. Si l'on considère qu'un ensemble de tâches correspond à la requête d'un utilisateur, le but des agents devient de traiter un maximum de tâches, provenant

d'un maximum d'utilisateurs, au plus vite. Cet objectif est similaire au *flowtime*. De nouvelles questions se poseront alors. Comment s'assurer que les jobs soumis le plus tôt soient traités suffisamment rapidement ? Comment intégrer cette notion de priorité aux interactions entre agents et aux stratégies de sélection de tâches ? Quel en est l'impact sur les comportements des agents composants ? Il est également commun d'enchainer les jobs MapReduce, c'est-à-dire qu'un job utilise comme entrées les sorties d'un autre job. Dans ce cas, les tâches possèdent les contraintes de précédence qu'il faudra intégrer au formalisme MASTA.

Rendre le processus de réallocation tolérant aux pannes

La décentralisation effective du protocole d'interaction réclame la mise en place d'interruptions temporelles aux points clés du protocole afin de prendre en compte les pertes ou retards de messages. Cependant, le protocole n'est pas adapté à la panne d'un nœud de calcul, c'est-à-dire à la perte de contact avec un agent pendant une délégation de tâche. Une perspective est donc de compléter le formalisme et le protocole afin qu'en plus d'être tolérant aux pertes/retards de messages, le système soit également robuste à la perte d'un nœud pendant le processus de délégation. Cette perspective nécessitera la prise en charge de plusieurs autres difficultés telles que la duplication des ressources et des informations de statut ou la réallocation des tâches allouées à l'agent de ce nœud pour redistribuer les tâches perdues lors de la panne d'un nœud de calcul.

Raffiner les stratégies de sélection de tâche

La mise en place de stratégies pour les différentes prises de décisions locales des agents est un point clé pour l'amélioration du processus de résolution d'un problème MASTA. Trouver de nouvelles stratégies de sélection de tâches appropriées aux contraintes du système reste donc une perspective.

Les stratégies décrites dans ce documents sont celles utilisées par les initiateurs des délégations de tâches. Une autre perspective est d'envisager des stratégies pour les enchérisseurs, c'est-à-dire pour les agents qui sont susceptibles de recevoir une tâche via une délégation socialement rationnelle. En effet, on peut se questionner sur l'intérêt d'une délégation si elle n'apporte qu'un gain minime de *makespan*. On pourra donc munir les agents d'une stratégie d'acceptabilité pour leur permettre de ne pas forcément faire une proposition dès l'instant qu'une délégation de tâche est socialement rationnelle. Par exemple, un enchérisseur ne répondra à un appel d'offre par une proposition que si le gain de *makespan* local est supérieur à 5 %.

Enfin, la stratégie de découpe de tâche repose pour l'instant sur des hypothèses qui ne tiennent pas compte de l'ensemble des caractéristiques d'une instance MASTA. Il faudra donc étendre l'heuristique de découpe de tâche pour y intégrer le fait que les tâches peuvent avoir un coût différent pour tous les agents du système ou encore pour prendre en compte la localité des ressources.

Améliorer le prototype MAS4Data

Il est tout d'abord nécessaire de continuer la validation expérimentale du modèle présenté dans cette thèse. Cela peut se faire en considérant de nouvelles données, de nouveaux paramètres et de nouvelles métriques. Par exemple, en scénarisant les ralentissements de certains nœuds de calcul, il serait intéressant de mesurer combien de temps est nécessaire au système pour corriger les écarts de charges induits par ces ralentissements.

Au niveau de l'architecture d'agent, on peut considérer un nouveau type d'agent composant dont le rôle sera d'anticiper la récupération des ressources avant l'exécution d'une tâche par le *worker*. Ainsi, pendant que le *worker* exécute une tâche, ce nouvel agent facilitera l'exécution de la prochaine tâche en réclamant les ressources manquantes à ses pairs.

Une autre perspective est d'explorer la possibilité d'intégrer MAS4Data à l'écosystème Hadoop. Ainsi, il sera possible de comparer une exécution qui utilise le processus de réallocation avec une exécution classique. En terme de modèle, cela demandera uniquement d'intégrer un système de fichiers distribué (par exemple HDFS), c'est-à-dire de considérer qu'une ressource est répliquée sur plusieurs nœuds. Cependant, cela représentera sûrement un travail conséquent d'ingénieurie pour gérer le transfert de tâches, l'accès aux *chunks* et les échanges entre *reducers*.

Annexes

Annexe A

Comportements des agents composants

Cette annexe contient les automates finis déterministes qui spécifient les comportements des agents composants décrits dans le chapitre 7. Les opérateurs suivants sont utilisés :

- `&&` : opérateur booléen ET ;
- `||` : opérateur booléen OU ;
- `not` : opérateur booléen NON ;
- `&` : opérateur qui délimite deux actions dans une séquence d'actions.

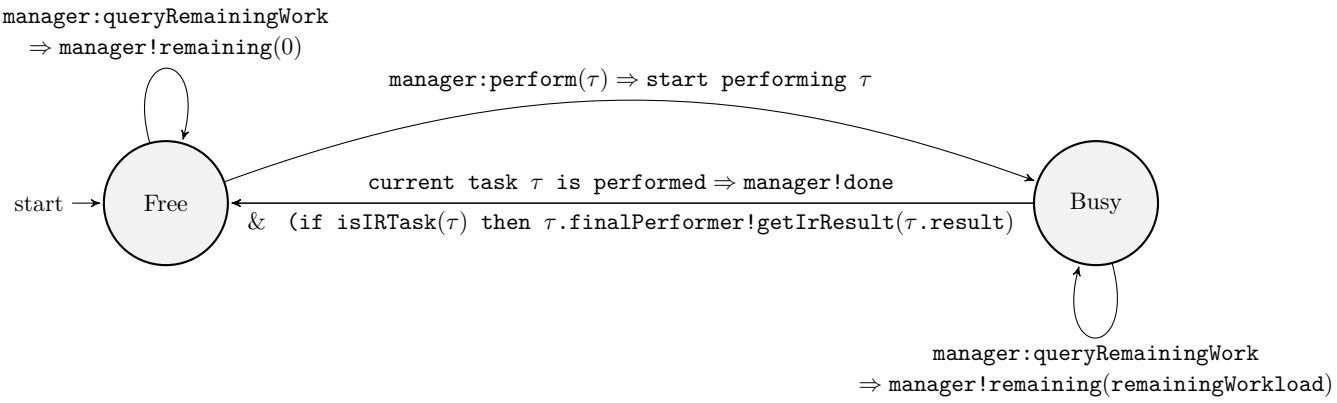


Figure A.1 : Comportement du *worker*. Pour répondre au message `queryRemainingWork` du *manager*, le *worker* doit être capable d'estimer la charge de travail que représente une tâche partiellement consommée. Sinon, on considère que la charge de travail d'une tâche reste constante jusqu'à ce qu'elle soit complétement exécutée.

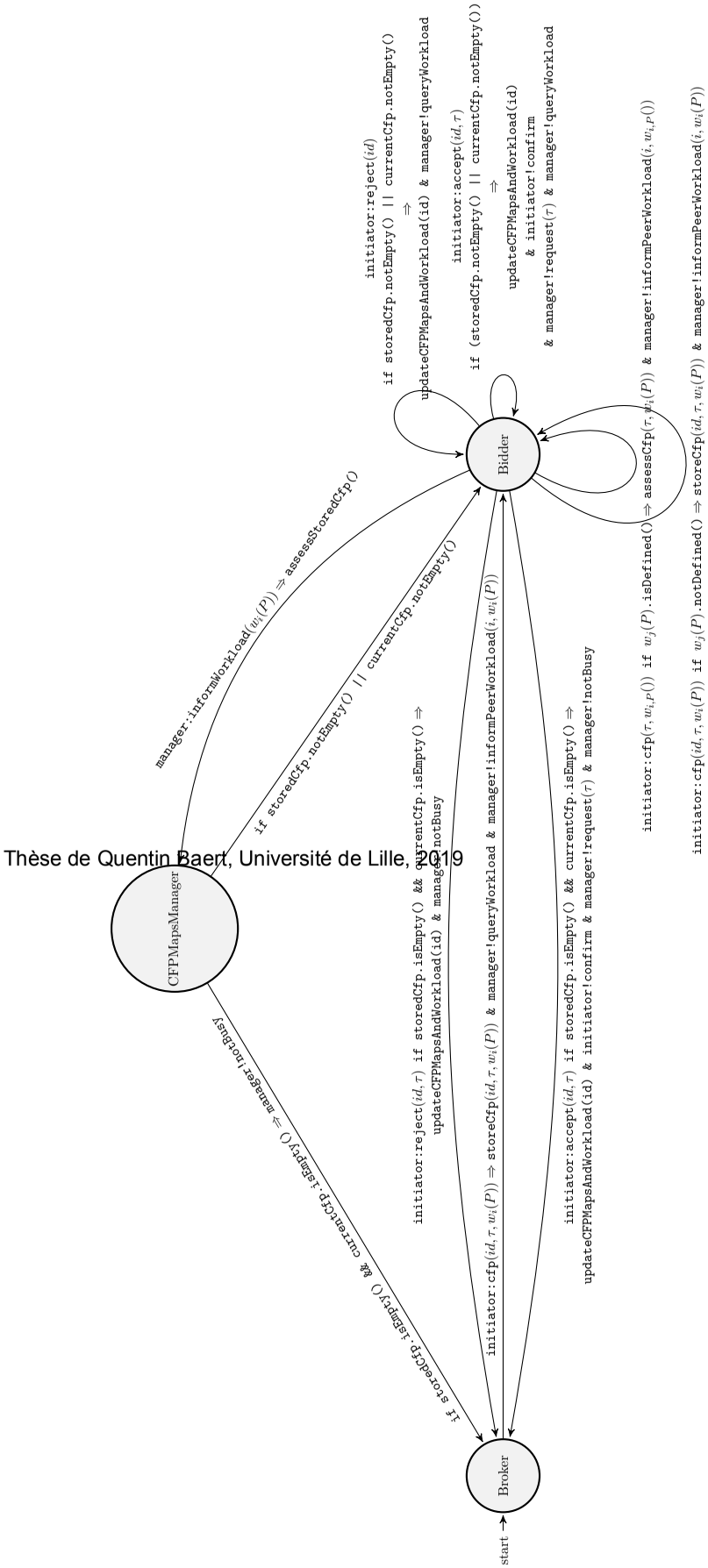


FIGURE A.2 : Comportement du *broker* en tant qu'enchérisseur (noté j). Les initiateurs (`initiator`) sont notés i. Les structures `storedCfp` et `currentCfp` contiennent respectivement les informations des `cfp` stockés et en cours de résolution. L'opération `storeCfp` stocke le `cfp` reçu. L'opération `assessCfp` consiste à répondre au `cfp` reçu par un message `propose`, `decline` ou à le stocker en fonction des charges de travail. L'opération `assessStoredCfp` consiste, pour chaque `cfp` stocké, à répondre par un message `propose`, *decline* ou à le garder stocker en fonction des charges de travail.

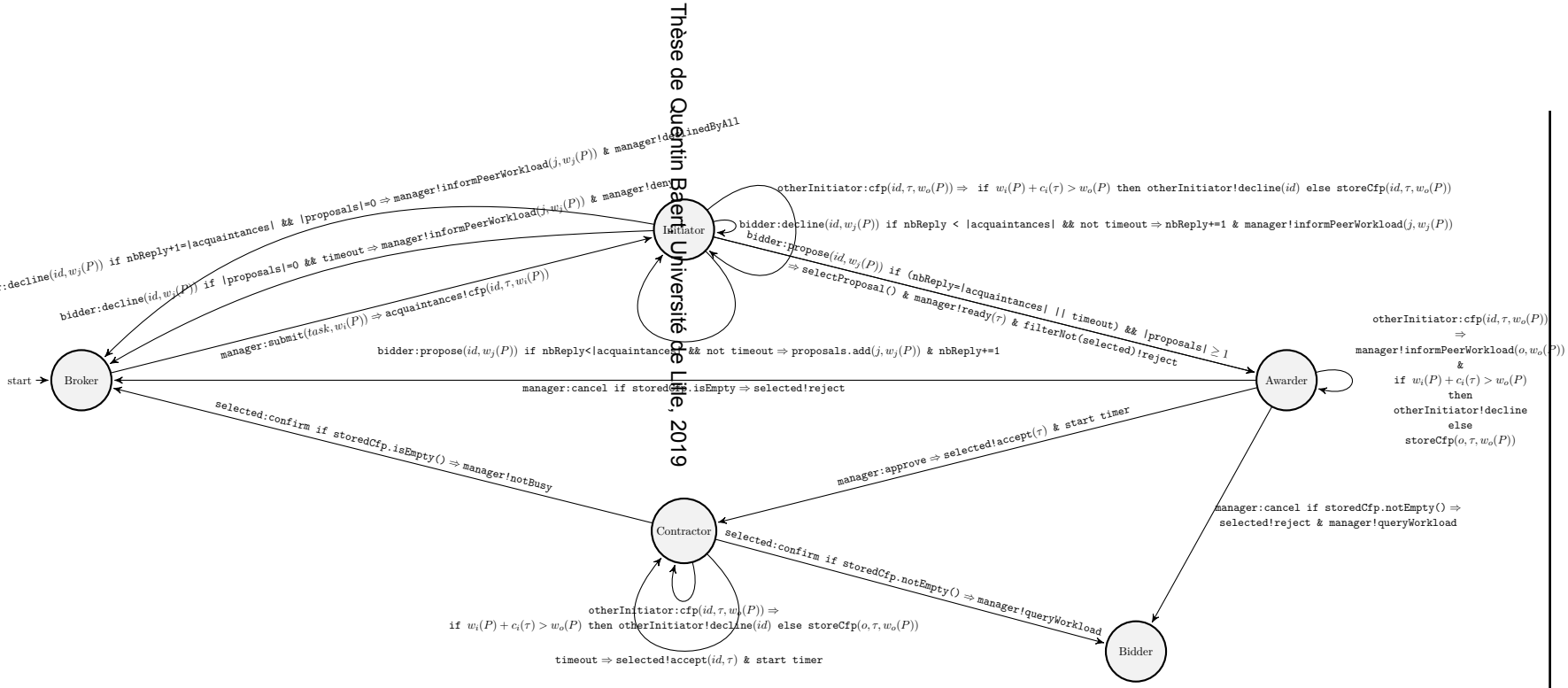


FIGURE A.3 : Comportement du *broker* en tant qu'initiateur (noté i). Les enchérisseurs (`bidder`) sont notés j. La variable `nbReply` contient le nombre de réponses reçues (proposition ou déclinaison) par l'initiateur. La structures `storedCfp` contient les informations des `cfp` stockés. La structure `proposals` contient les propositions reçues pour le `cfp` courant. La structure `acquaintances` contient l'ensemble des pairs de l'agent. L'opération `storeCfp` stocke le `cfp` reçu. L'opération `selectProposal` sélectionne l'enchérisseur qui gagne l'enchère parmi les réponses reçues.

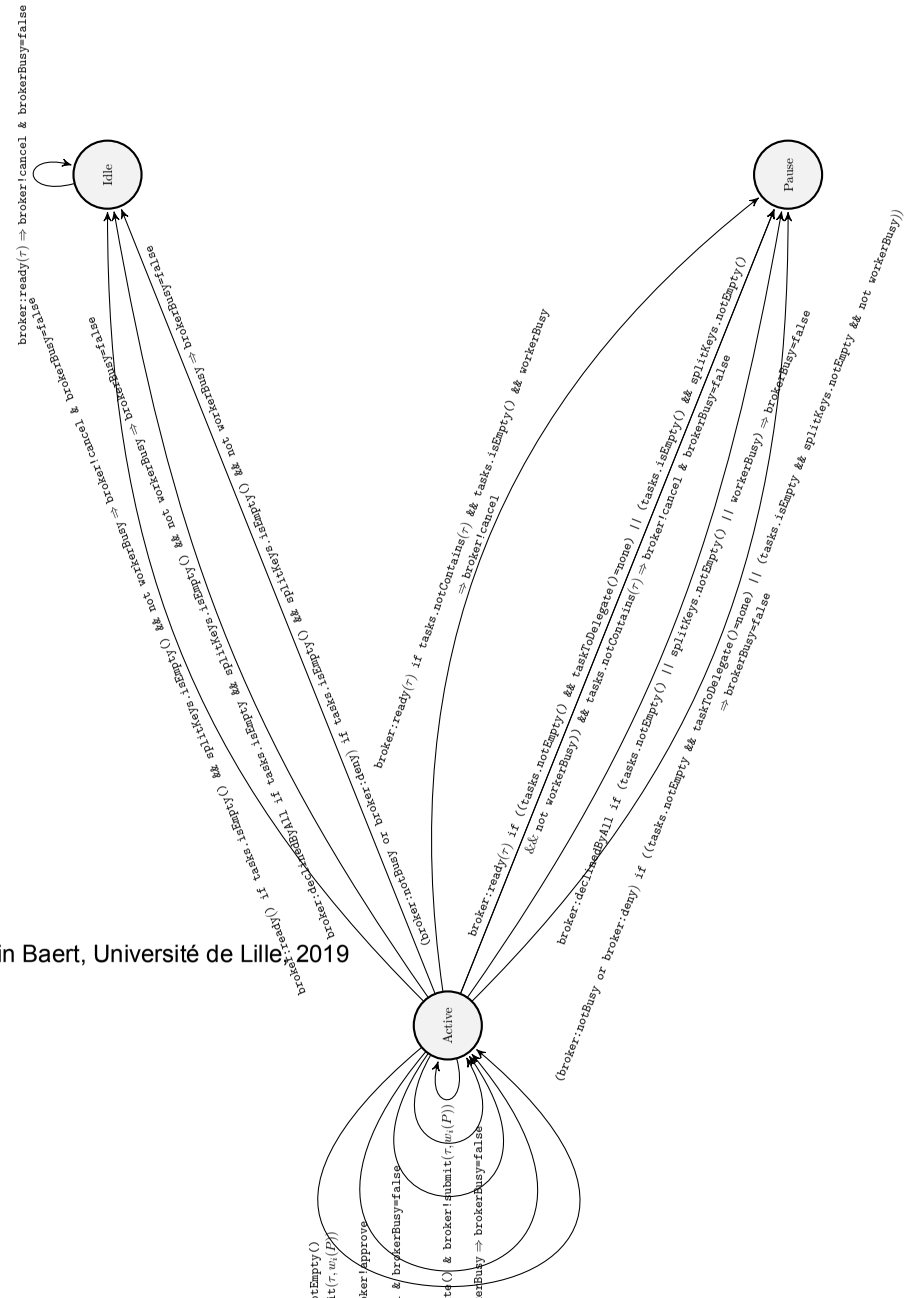


FIGURE A.4 : Comportement du *manager* lorsqu'il interagit avec le *broker* en tant qu'initiateur. La structure `tasks` contient l'ensemble des tâches de l'agent (P_i). La méthode `taskToDelegate()` retourne la prochaine tâche à déléguer selon la stratégie de sélection de tâche utilisée par l'agent. La variable `workerBusy` indique si le *worker* est en train d'exécuter une tâche. La variable `brokerBusy` indique si le *broker* est occupé. Cette variable permet au *manager* d'identifier quand il peut soumettre une tâche au *broker*.

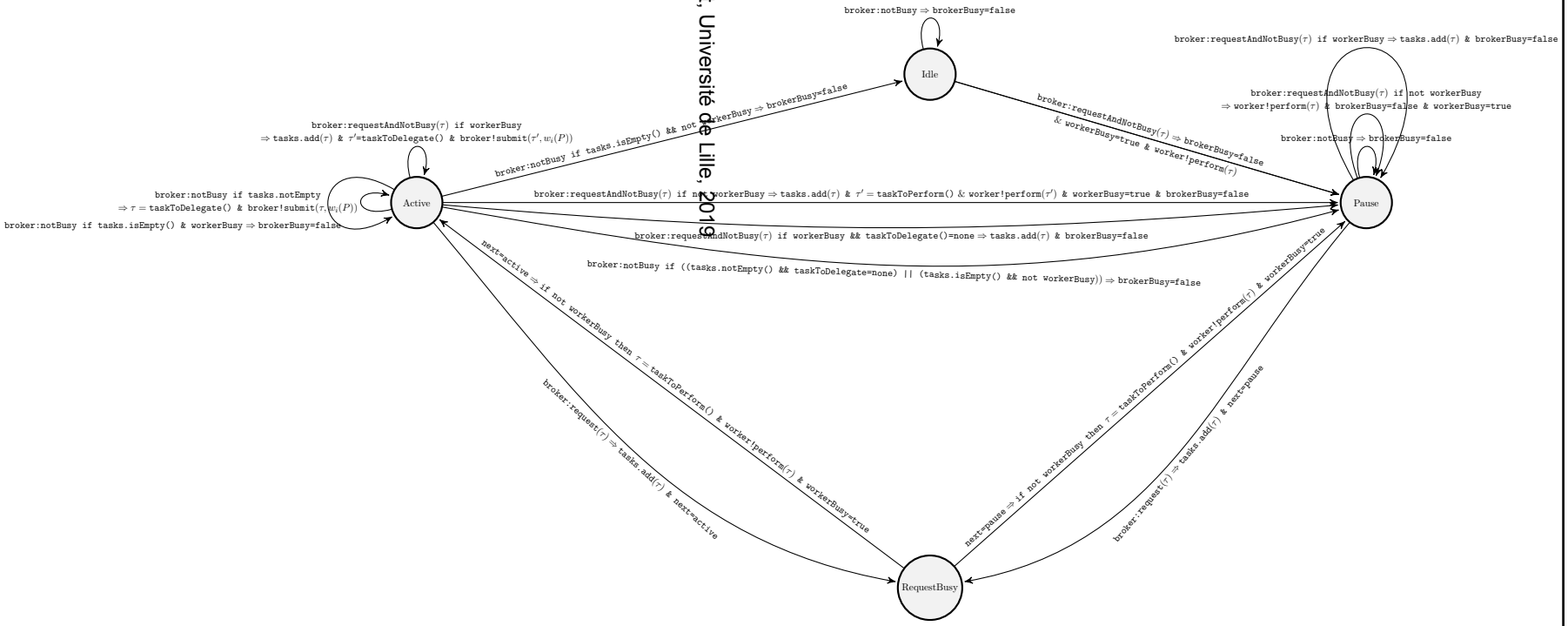


FIGURE A.5 : Comportement du *manager* lorsqu'il interagit avec le *broker* en tant qu'enchérisseur. La structure `tasks` contient l'ensemble des tâches de l'agent (P_i). La méthode `taskToPerform()` retourne la prochaine tâche à exécuter selon la stratégie de sélection de tâche utilisée par l'agent. La variable `workerBusy` indique si le *worker* est en train d'exécuter une tâche. La variable `brokerBusy` indique si le *broker* est occupé. Cette variable permet au *manager* d'identifier quand il peut soumettre une tâche au *broker*. Le message `requestAndNotBusy` combine la sémantique des messages `request` et `notBusy`. Si le *manager* reçoit un message `requestAndNotBusy`, cela signifie que le *broker* demande l'ajout d'une nouvelle tâche à l'ensemble de tâches et qu'il annonce ne plus être occupé.

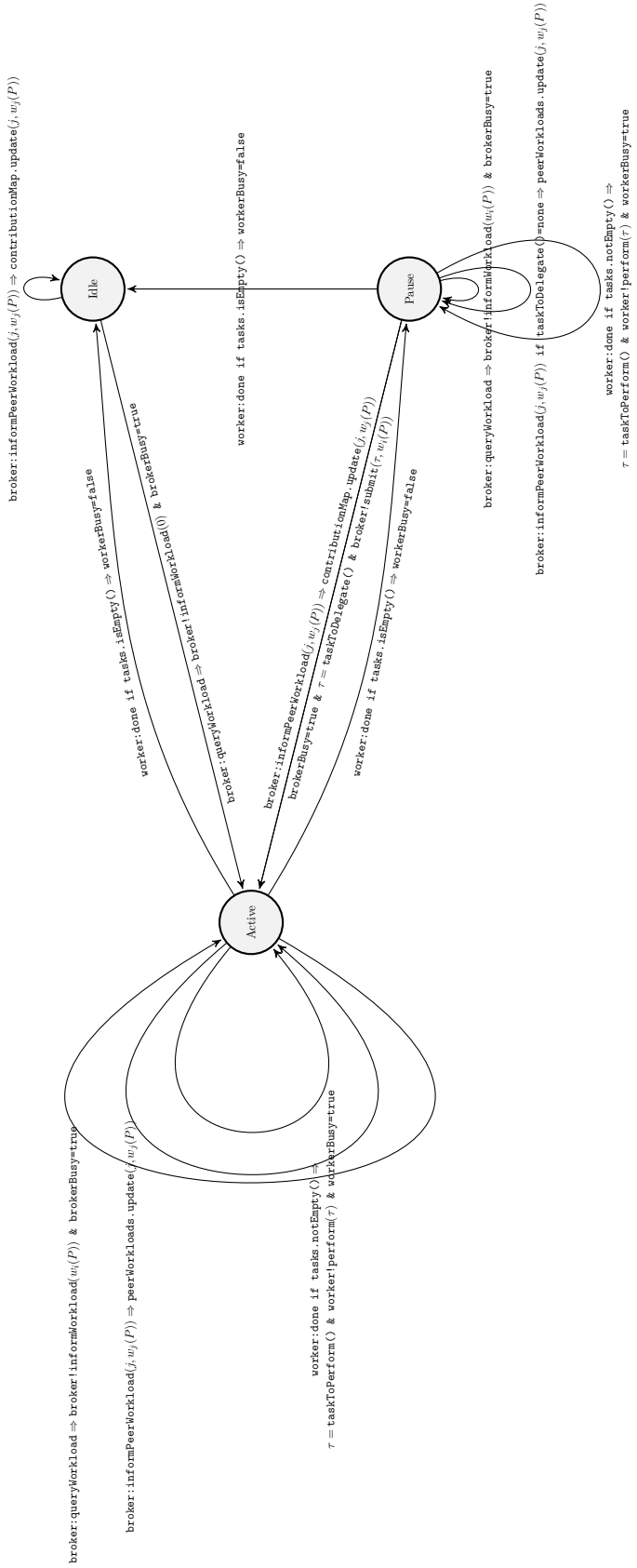


FIGURE A.6 : Comportement du *manager* lorsqu'il interagit avec le *worker*. La structure `tasks` contient l'ensemble des tâches de l'agent (P_i). La méthode `taskToPerform()` retourne la prochaine tâche à exécuter selon la stratégie de sélection de tâche utilisée par l'agent. La variable `workerBusy` indique si le *worker* est en train d'exécuter une tâche.

Annexe B

Découpe de tâche de *reduce*

Afin que les agents puissent appliquer le mécanisme effectif de découpe de tâches présenté dans cette annexe, il est nécessaire que l'hypothèse de l'heuristique de découpe du chapitre 6 soit toujours vérifiée : le coût d'une tâche doit être le même pour tous les agents du système.

B.1 Découpe effective de tâche

Dans le chapitre 6, j'ai présenté une heuristique de découpe de tâche qui ne prend pas en compte les contraintes liées aux ressources nécessaires à leur exécution. Dans le chapitre 8 la nature des tâches de *reduce* est décrite. Les *chunks* sont les ressources atomiques liées aux tâches de *reduce*. Diviser une tâche de *reduce* demande donc de répartir les *chunks* dans les différentes sous-tâches créées. Cette annexe présente la découpe effective des tâches *reduce* qui repose sur l'heuristique de la section 6.3.2 mais y intègre la répartition des *chunks*. Pour rappel, l'heuristique de découpe propose un calcul qui détermine une valeur k. Cette valeur correspond au nombre de sous-tâches à déléguer suite à la découpe de la tâche la plus coûteuse d'un agent. La valeur k est déterminée de sorte à minimiser la charge de travail de ce même agent si toutes les sous-tâches sous effectivement déléguées.

Un *chunk* est dit complet s'il n'est pas possible de lui ajouter des données, il est dit partiel sinon. Soient $\mathcal{C}$ la taille d'un *chunk* complet (par exemple $\mathcal{C} = 64\text{Mo}$) et $w_{\mathcal{C}}$ la charge de travail que représente le traitement d'un *chunk* complet. La charge de travail que représente un *chunk* partiel est inférieure à $w_{\mathcal{C}}$. Les *chunks* d'une tâche sont initialement créés par les *mappers* qui, pour chaque clé qu'ils traitent et selon les nombres de valeurs qui leur sont associés, produisent des *chunks* complets et un *chunk* partiel au plus. Une tâche de *reduce* dont x *mappers* ont produit la clé contient donc un certain nombre de *chunks* complet (la somme de tous les *chunks* complets construits par les x *mappers*) et au plus x *chunks* partiels.

Un agent i découpe sa tâche la plus coûteuse à condition qu'il soit en pause et qu'il existe m pairs moins chargés que lui selon ses croyances. Afin de créer des sous-tâches composées d'au moins un *chunk*, le calcul de k pour l'agent *reducer* i doit être revu de la façon suivante :

$$k = \underset{k \in [1;N]}{\operatorname{argmin}} \left(w_i(P) - \frac{k\Delta_i^k}{k+1} \right)$$

avec $N = \underset{n \in [1;m]}{\operatorname{argmax}}(\Delta_i^n \geq w_{\mathcal{C}})$.

Ensuite la découpe de la tâche τ pour le *reducer* i consiste à :

1. déterminer k en utilisant la formule ci-dessus ;
2. diviser les *chunks* de τ pour construire $k+1$ sous-tâches en répétant la routine suivante : parmi les *chunks* de données non attribués de τ, le *chunk* le plus coûteux est attribué à la sous-tâche de plus faible coût.

Si l'on estime que la découpe déterminée par l'heuristique de découpe de tâche est idéale, les $k+1$ sous-tâches construites grâce à ce mécanisme sont proches de leur coût idéal (c'est-à-dire $\frac{\Delta_i^k}{k+1}$ pour k d'entre elles et $c_\tau - \frac{k\Delta_i^k}{k+1}$ pour la dernière). En fait, pour une sous-tâche $task'$ de coût idéal $c^*_{\tau'}$ et de coût réel $c_{\tau'}$, on observe que $|c^*_{\tau'} - c_{\tau'}| \leq w_{\mathcal{C}}$.

B.2 Algorithme de découpe de tâche

Algorithme B.1 : Découpe, par l'agent i, de la tâche τ en $k+1$ sous-tâches en prenant les *chunks* en compte.

Input : τ, k, Δ_i^k

```
   /* subtasks est créée tel que subtasks.length=k                          */
 1 subtasks ← [emptyTask, ..., emptyTask];
   /* Les chunks a répartir en k+1 sous-tâches                              */
 2 chunks ← τ.chunks().copy();
   /* Coût total représenté par les chunks qui ont été répartis             */
 3 totalCost ← 0;
   /* Prochain chunk a ajouter à une sous-tâche                             */
 4 ρ ← chunks.head();
 5 while totalCost + c_ρ ≤ Δ_i^k do
       /* Supprime ρ des chunks a considérer par la suite                   */
 6     chunks ← chunks.tail();
       /* La sous-tâche avec le coût le plus bas reçoit chunToAdd           */
 7     subtasks.head().addChunk(ρ);
       /* Le coût de ρ est ajouté à totalCost                               */
 8     totalCost += c_ρ;
       /* subtasks est trié tel que les sous-tâches soient par ordre croissant de coût */
 9     subtasks.sortBy((μ_1^τ, μ_2^τ) ⇒ c_{μ_1^τ} < c_{μ_2^τ});
10 end
   /* Retourne les sous-tâches et une tâche supplémentaire qui contient les chunks restant,
      c'est-à-dire k+1 sous-tâches                                          */
11 return subtasks + new Task(chunks)
```

B.3 Exécution de tâches divisibles

À partir d'une tâche divisible, un *reducer* peut créer autant de sous-tâches que la tâche contient de *chunks*. Grâce à la nature de la fonction de *reduce*, il est donc possible de traiter individuellement chacun des *chunks* d'une tâche grâce à la fonction IR et de produire son résultat final

grâce à la fonction FR. Cette particularité permet d'envisager l'exécution d'une tâche divisible autrement que comme une action atomique exécutée par le *worker* d'un agent *reducer*.

Annexe C

Script Scala pour l'illustration du biais de partitionnement

Le script suivant utilise le langage de programmation Scala. Son exécution illustre le biais décrit dans l'exemple 3.4.

```
1 /* Contient l'ensemble des temperatures entre -50 et +50 degres par pas de 0.5
2  * degre */
3 val temps = -50.0 to 50.0 by 0.5
4 /* Pour chaque temperature ci-dessus, donne le numero du reducer auquel la tache
5  * est associee */
6 val reducerNumbers = temps map (temp => math.abs(temp.hashCode % 20))
7 /* Verifie qu'uniquement un reducer sur quatre se voit attribuer une tache */
8 reducerNumbers forall (reducerNumber => reducerNumber % 4 == 0)
```

Annexe D

Publications

Les travaux présentés dans cette thèse ont fait l'objet des publications suivantes.

Journal international

- Fair multi-agent task allocation for large datasets analysis. Knowledge and Information Systems (KAIS). [Baert *et al.* 2018]

Journal national

- Allocation équitable de tâches pour l'analyse de données massives : MapReduce et système multi-agent. Revue des Sciences et Technologies de l'Information - Série RIA : Revue d'Intelligence Artificielle. [Baert *et al.* 2017a]

Conférence internationale

- Adaptive Multi-agent System for Situated Task Allocation. 18th Conference on Autonomous Agents and MultiAgent Systems (AAMAS'19). *Short paper*. [Baert *et al.* 2019a]
- Negotiation Strategy of Divisible Tasks for Large Dataset Processing. 15th European Conference on Multi-Agent Systems (EUMAS'17). [Baert *et al.* 2017b]
- Fair Multi-agent Task Allocation for Large Data Sets Analysis. 14th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'16). [Baert *et al.* 2016b]

Conférence nationale

- Stratégie situationnelle pour l'équilibrage de charge. Journées Francophones sur les Systèmes Multi-Agents (JFSMA'19). [Baert *et al.* 2019b]
- Stratégie de découpe de tâche pour le traitement de données massives. Journées Francophones sur les Systèmes Multi-Agents (JFSMA'17). [Baert *et al.* 2017c]
- Allocation équitable de tâches pour l'analyse de données massives. Journées Francophones sur les Systèmes Multi-Agents (JFSMA'16). [Baert *et al.* 2016a]

Table des figures

Liste des tableaux

Bibliographie

[Abdallah & Lesser 2005] Sherief Abdallah et Victor R. Lesser. *Modeling task allocation using a decision theoretic model.* In 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands, pages 719–726, 2005. (Cité en pages 21 et 26.)

[Akka 2019] Akka. *Akka documentation*, 2019. https://akka.io/docs/. (Cité en page 106.)

[Alrayes *et al.* 2018] Bedour Alrayes, Özgür Kafali et Kostas Stathis. *Concurrent bilateral negotiation for open e-markets : the Conan strategy.* Knowl. Inf. Syst., vol. 56, no. 2, pages 463–501, 2018. (Cité en page 36.)

[An *et al.* 2010a] Bo An, Victor R. Lesser, David E. Irwin et Michael Zink. *Automated negotiation with decommitment for dynamic resource allocation in cloud computing.* In 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3, pages 981–988, 2010. (Cité en pages 23 et 26.)

[An *et al.* 2010b] Bo An, Victor R. Lesser, David E. Irwin et Michael Zink. *Automated negotiation with decommitment for dynamic resource allocation in cloud computing.* In 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3, pages 981–988, 2010. (Cité en page 36.)

[Arrow 1951] Kenneth J Arrow. *Social choice and individual values.(Cowles Commission Mongr. No. 12.).* 1951. (Cité en page 34.)

[Arrow 1958] Kenneth J Arrow. *Utilities, attitudes, choices : A review note.* Econometrica : Journal of the Econometric Society, pages 1–23, 1958. (Cité en page 29.)

[Attiya & Hamam 2006] Gamal Attiya et Yskandar Hamam. *Task allocation for maximizing reliability of distributed systems : A simulated annealing approach.* J. Parallel Distrib. Comput., vol. 66, no. 10, pages 1259–1266, 2006. (Cité en page 24.)

[Aydogan *et al.* 2014] Reyhan Aydogan, Víctor Sánchez-Anguix, Vicente Julián, Joost Broekens et Catholijn M. Jonker. *Guest Editorial : Computational Approaches for Conflict Resolution in Decision Making : New Advances and Developments.* Cybernetics and Systems, vol. 45, no. 3, pages 217–221, 2014. (Cité en page 28.)

[Baert *et al.* 2016a] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Allocation équitable de tâches pour l'analyse de données massives.* In Systèmes Multi-Agents et simulation - Vingt-quatrièmes journées francophones sur les systèmes multi-agents, JFSMA 16, Saint-Martin-du-Vivier (Rouen), France, Octobre 5-7, 2016., pages 55–64, 2016. (Cité en page 163.)

[Baert *et al.* 2016b] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Fair Multi-agent Task Allocation for Large Data Sets Analysis.* In Advances in Practical Applications of Scalable Multi-agent Systems. The PAAMS Collection - 14th International Conference, PAAMS 2016, Sevilla, Spain, June 1-3, 2016, Proceedings, pages 24–35, 2016. (Cité en page 163.)

[Baert *et al.* 2017a] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Allocation équitable de tâches pour l'analyse de données massives.* Revue d'Intelligence Artificielle, vol. 31, no. 4, pages 401–426, 2017. (Cité en page 163.)

[Baert *et al.* 2017b] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Negotiation Strategy of Divisible Tasks for Large Dataset Processing.* In Multi-Agent Systems and Agreement Technologies - 15th European Conference, EUMAS 2017, and 5th International Conference, AT 2017, Évry, France, December 14-15, 2017, Revised Selected Papers, pages 370–384, 2017. (Cité en page 163.)

[Baert *et al.* 2017c] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Stratégie de découpe de tâche pour le traitement de données massives.* In Cohésion : fondement ou propriété émergente - JFSMA 17 - Vingt-cinquièmes journées francophones sur les systèmes multi-agents, Caen, France, July 4-6, 2017., pages 65–74, 2017. (Cité en page 163.)

[Baert *et al.* 2018] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *Fair multi-agent task allocation for large datasets analysis.* Knowl. Inf. Syst., vol. 54, no. 3, pages 591–615, 2018. (Cité en page 163.)

[Baert *et al.* 2019a] Quentin Baert, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier et Kostas Stathis. *Adaptive Multi-agent System for Situated Task Allocation.* In Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019, pages 1790–1792, 2019. (Cité en page 163.)

[Baert *et al.* 2019b] Quentin Baert, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier et Kostas Stathis. *Stratégie situationnelle pour l'équilibrage de charge.* In Vingt-septièmes journées francophones sur les systèmes multi-agents, Distributed cooperative problem solving, pages 9–18, Toulouse, France, Juillet 2019. Cépaudès. (Cité en page 163.)

[Baert *et al.* 2019c] Quentin Baert, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. *MAS4Data*, 2019. `https://github.com/cristal-smac/mas4data`. (Cité en page 5.)

[Boutilier & Hoos 2001] Craig Boutilier et Holger H. Hoos. *Bidding Languages for Combinatorial Auctions.* In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001, pages 1211–1217, 2001. (Cité en page 29.)

[Boutilier *et al.* 2011] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos et David Poole. *CP-nets : A Tool for Representing and Reasoning withConditional Ceteris Paribus Preference Statements.* CoRR, vol. abs/1107.0023, 2011. (Cité en page 29.)

[Bruno *et al.* 1974] John L. Bruno, Edward G. Coffman Jr. et Ravi Sethi. *Scheduling Independent Tasks to Reduce Mean Finishing Time.* Commun. ACM, vol. 17, no. 7, pages 382–387, 1974. (Cité en page 18.)

[Chen *et al.* 2010] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng et Song Guo. *SAMR : A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment.* In 10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010, pages 2736–2743, 2010. (Cité en pages 47 et 50.)

[Chevaleyre *et al.* 2006] Yann Chevaleyre, Paul E. Dunne, Ulle Endriss, Jérôme Lang, Michel Lemaître, Nicolas Maudet, Julian A. Padget, Steve Phelps, Juan A. Rodríguez-Aguilar et Paulo Sousa. *Issues in Multiagent Resource Allocation.* Informatica (Slovenia), vol. 30, no. 1, pages 3–31, 2006. (Cité en pages 16 et 33.)

[Clinger 1981] William Douglas Clinger. *Foundations of actor semantics.* PhD thesis, Massachusetts Institute of Technology, 1981. (Cité en page 103.)

[Dean & Ghemawat 2008] Jeffrey Dean et Sanjay Ghemawat. *MapReduce : simplified data processing on large clusters.* Commun. ACM, vol. 51, no. 1, pages 107–113, 2008. (Cité en pages 39 et 40.)

[DeCandia *et al.* 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall et Werner Vogels. *Dynamo : amazon's highly available key-value store.* In Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 205–220, 2007. (Cité en page 39.)

[Delecroix *et al.* 2016] Fabien Delecroix, Maxime Morge, Thomas Nachtergaelle et Jean-Christophe Routier. *Multi-party negotiation with preferences rather than utilities.* Multiagent and Grid Systems, vol. 12, no. 2, pages 139–165, 2016. (Cité en page 28.)

[Delecroix 2015] Fabien Delecroix. *Dialoguer pour décider : Recommandation experte proactive et prise de de'cision multi-agents e'quitable. (From dialogue to decision / From dialogue to decision : Proactive expert recommendation and fair multi-agents negotiation).* PhD thesis, Lille University of Science and Technology, France, 2015. (Cité en page 34.)

[Durfee *et al.* 1989] Edmund H. Durfee, Victor R. Lesser et Daniel D. Corkill. *Trends in Cooperative Distributed Problem Solving.* IEEE Trans. Knowl. Data Eng., vol. 1, no. 1, pages 63–83, 1989. (Cité en page 28.)

[Endriss *et al.* 2006] Ulrich Endriss, Nicolas Maudet, Fariba Sadri et Francesca Toni. *Negotiating Socially Optimal Allocations of Resources.* J. Artif. Intell. Res., vol. 25, pages 315–348, 2006. (Cité en page 29.)

[Essa *et al.* 2014] Y. M. Essa, G. Attiya et A. El-Sayed. *Mobile Agent based New Framework for Improving Big Data Analysis.* International Journal of Advanced Computer Science and Applications, vol. 5, no. 3, pages 25–32, 2014. (Cité en page 49.)

[Faratin *et al.* 1998] Peyman Faratin, Carles Sierra et Nicholas R. Jennings. *Negotiation decision functions for autonomous agents.* Robotics and Autonomous Systems, vol. 24, no. 3-4, pages 159–182, 1998. (Cité en page 33.)

[Fatima *et al.* 2001] S. Shaheen Fatima, Michael J. Wooldridge et Nicholas R. Jennings. *Optimal Negotiation Strategies for Agents with Incomplete Information.* In Intelligent Agents VIII, 8th International Workshop, ATAL 2001 Seattle, WA, USA, August 1-3, 2001, Revised Papers, pages 377–392, 2001. (Cité en page 30.)

[Fischer *et al.* 1995] Klaus Fischer, Jörg P. Müller, Markus Pischel et Darius Schier. *A Model for Cooperative Transportation Scheduling.* In Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA, pages 109–116, 1995. (Cité en page 36.)

[Gale & Shapley 2013] D. Gale et L. S. Shapley. *College Admissions and the Stability of Marriage.* The American Mathematical Monthly, vol. 120, no. 5, pages 386–391, 2013. (Cité en page 28.)

[Grabisch 1997] Michel Grabisch. *k-order additive discrete fuzzy measures and their representation.* Fuzzy Sets and Systems, vol. 92, no. 2, pages 167–189, 1997. (Cité en page 30.)

[Graham 1969] Ronald L. Graham. *Bounds on Multiprocessing Timing Anomalies.* SIAM Journal of Applied Mathematics, vol. 17, no. 2, pages 416–429, 1969. (Cité en pages 18 et 128.)

[Gray *et al.* 1997] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow et Hamid Pirahesh. *Data Cube : A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals.* Data Min. Knowl. Discov., vol. 1, no. 1, pages 29–53, 1997. (Cité en page 121.)

[Greco *et al.* 2016] Salvatore Greco, J Figueira et M Ehrgott. Multiple criteria decision analysis. Springer, 2016. (Cité en page 29.)

[Grosu & Chronopoulos 2005] Daniel Grosu et Anthony T. Chronopoulos. *Noncooperative load balancing in distributed systems.* J. Parallel Distrib. Comput., vol. 65, no. 9, pages 1022–1034, 2005. (Cité en pages 22 et 26.)

[Grosu *et al.* 2002] Daniel Grosu, Anthony T. Chronopoulos et Ming-Ying Leung. *Load Balancing in Distributed Systems : An Approach Using Cooperative Games.* In 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings, 2002. (Cité en pages 20 et 26.)

[Gufler *et al.* 2011] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser et Alfons Kemper. *Handling Data Skew in MapReduce.* In CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 7-9 May, 2011, pages 574–583, 2011. (Cité en pages 46, 47 et 50.)

[Guttman *et al.* 1998] Robert H. Guttman, Alexandros Moukas et Pattie Maes. *Agent-mediated electronic commerce : a survey.* Knowledge Eng. Review, vol. 13, no. 2, pages 147–159, 1998. (Cité en page 27.)

[Habes *et al.* 2014] Mohamed Raouf Habes, Habiba Belleili-Souici et Laurent Vercouter. *A dynamic hybrid buyer bidding strategy for decentralized and cloud-based many-to-many negotiation*. Multiagent and Grid Systems, vol. 10, no. 3, pages 165–183, 2014. (Cité en page 36.)

[Hadoop 2019] Apache Hadoop. *Apache Hadoop*, 2019. https://hadoop.apache.org/. (Cité en page 122.)

[Hansson 1996] Sven Ove Hansson. *What is ceteris paribus preference ?* J. Philosophical Logic, vol. 25, no. 3, pages 307–332, 1996. (Cité en page 29.)

[Hariri & Potts 1991] A. M. A. Hariri et Chris N. Potts. *Heuristics for scheduling unrelated parallel machines*. Computers & OR, vol. 18, no. 3, pages 323–331, 1991. (Cité en pages 21 et 26.)

[Hewitt 1977] Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages*. Artif. Intell., vol. 8, no. 3, pages 323–364, 1977. (Cité en page 103.)

[Horn 1973] W. A. Horn. *Technical Note - Minimizing Average Flow Time with Parallel Machines*. Operations Research, vol. 21, no. 3, pages 846–847, 1973. (Cité en page 18.)

[Horowitz & Sahni 1976] Ellis Horowitz et Sartaj Sahni. *Exact and Approximate Algorithms for Scheduling Nonidentical Processors*. J. ACM, vol. 23, no. 2, pages 317–327, 1976. (Cité en page 14.)

[Ibarra & Kim 1977] Oscar H. Ibarra et Chul E. Kim. *Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors*. J. ACM, vol. 24, no. 2, pages 280–289, 1977. (Cité en page 21.)

[Ito *et al.* 2008] Takayuki Ito, Mark Klein et Hiromitsu Hattori. *A multi-issue negotiation protocol among agents with nonlinear utility functions*. Multiagent and Grid Systems, vol. 4, no. 1, pages 67–83, 2008. (Cité en page 30.)

[Jiang & Huang 2012] Yichuan Jiang et Zhichuan Huang. *The Rich Get Richer : Preferential Attachment in the Task Allocation of Cooperative Networked Multiagent Systems With Resource Caching*. IEEE Trans. Systems, Man, and Cybernetics, Part A, vol. 42, no. 5, pages 1040–1052, 2012. (Cité en pages 21 et 26.)

[Jiang & Li 2011] Yichuan Jiang et Zhaofeng Li. *Locality-sensitive task allocation and load balancing in networked multiagent systems : Talent versus centrality*. J. Parallel Distrib. Comput., vol. 71, no. 6, pages 822–836, 2011. (Cité en pages 21 et 26.)

[Jiang 2016] Yichuan Jiang. *A Survey of Task Allocation and Load Balancing in Distributed Systems*. IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 2, pages 585–599, 2016. (Cité en page 15.)

[Jonker *et al.* 2017] Catholijn M. Jonker, Reyhan Aydogan, Tim Baarslag, Katsuhide Fujita, Takayuki Ito et Koen V. Hindriks. *Automated Negotiating Agents Competition (ANAC)*. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA., pages 5070–5072, 2017. (Cité en page 33.)

[Kang *et al.* 2010] Qinma Kang, Hong He, Hui-Min Song et Rong Deng. *Task allocation for maximizing reliability of distributed computing systems using honeybee mating optimization.* Journal of Systems and Software, vol. 83, no. 11, pages 2165–2174, 2010. (Cité en pages 21 et 26.)

[Keeney & Raiffa 1993] Ralph L Keeney et Howard Raiffa. Decisions with multiple objectives : preferences and value trade-offs. Cambridge university press, 1993. (Cité en page 30.)

[Knabe *et al.* 2002] Tore Knabe, Michael Schillo et Klaus Fischer. *Improvements to the FIPA contract net protocol for performance increase and cascading applications.* In In International Workshop for Multi-Agent Interoperability at the German Conference on AI (KI-2002. Citeseer, 2002. (Cité en page 36.)

[Kraus *et al.* 2003] Sarit Kraus, Onn Shehory et Gilad Taase. *Coalition formation with uncertain heterogeneous information.* In The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings, pages 1–8, 2003. (Cité en pages 23 et 26.)

[Kraus 1997] Sarit Kraus. *Negotiation and Cooperation in Multi-Agent Environments.* Artif. Intell., vol. 94, no. 1-2, pages 79–97, 1997. (Cité en page 28.)

[Kuhn 2010] Harold W. Kuhn. *The Hungarian Method for the Assignment Problem.* In 50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art, pages 29–47. 2010. (Cité en page 14.)

[Kwon *et al.* 2011] YongChul Kwon, Magdalena Balazinska, Bill Howe et Jerome Rolia. *A study of skew in mapreduce applications.* Open Cirrus Summit, vol. 11, 2011. (Cité en pages 43 et 46.)

[Kwon *et al.* 2013] YongChul Kwon, Kai Ren, Magdalena Balazinska et Bill Howe. *Managing Skew in Hadoop.* IEEE Data Eng. Bull., vol. 36, no. 1, pages 24–33, 2013. (Cité en pages 47 et 50.)

[Lama & Zhou 2012] Palden Lama et Xiaobo Zhou. *AROMA : automated resource allocation and configuration of mapreduce environment in the cloud.* In 9th International Conference on Autonomic Computing, ICAC'12, San Jose, CA, USA, September 16 - 20, 2012, pages 63–72, 2012. (Cité en page 47.)

[Lenstra *et al.* 1990] Jan Karel Lenstra, David B. Shmoys et Éva Tardos. *Approximation Algorithms for Scheduling Unrelated Parallel Machines.* Math. Program., vol. 46, pages 259–271, 1990. (Cité en pages 20 et 26.)

[Li 1992] Wentian Li. *Random texts exhibit Zipf's-law-like word frequency distribution.* IEEE Trans. Information Theory, vol. 38, no. 6, pages 1842–1845, 1992. (Cité en page 46.)

[Lin 2009] Jimmy Lin. *The Curse of Zipf and Limits to Parallelization : A Look at the Stragglers Problem in MapReduce.* In Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009. (Cité en page 46.)

[Liroz-Gistau *et al.* 2016] Miguel Liroz-Gistau, Reza Akbarinia, Divyakant Agrawal et Patrick Valduriez. *FP-Hadoop : Efficient processing of skewed MapReduce jobs*. Inf. Syst., vol. 60, pages 69–84, 2016. (Cité en pages 47, 50 et 120.)

[Lopes *et al.* 2008] Fernando Lopes, Michael J. Wooldridge et Augusto Q. Novais. *Negotiation among autonomous computational agents : principles, analysis and challenges*. Artif. Intell. Rev., vol. 29, no. 1, pages 1–44, 2008. (Cité en page 27.)

[Martello *et al.* 1997] Silvano Martello, François Soumis et Paolo Toth. *Exact and Approximation Algorithms for Makespan Minimization on Unrelated Parallel Machines*. Discrete Applied Mathematics, vol. 75, no. 2, pages 169–188, 1997. (Cité en pages 20 et 26.)

[Ministère de l'éducation nationale et de la jeunesse 2019] Ministère de l'éducation nationale et de la jeunesse. *Liste des mots les plus fréquents de la langue française*, 2019. http://eduscol.education.fr/cid47916/liste-des-mots-classee-par-frequence-decroissante.html. (Cité en page 46.)

[Morge *et al.* 2008] Maxime Morge, Kostas Stathis et Laurent Vercouter. *Argumentation sur les motivations propres dans l'architecture V3A pour des agents auto-adaptatifs (présentation courte)*. In Systèmes Multi-Agents, Communautés virtuelles et naturelles - JFSMA 08 - Seizièmes journées francophones sur les systèmes multi-agents, Brest, France, October 15-17, 2008, pages 149–158, 2008. (Cité en page 107.)

[Morge 2018] Maxime Morge. *Négociation bilatérale par concession : un état de l'art*. In Distribution et Décentralisation - Vingt-sixièmes journées francophones sur les systèmes multi-agents, JFSMA 2018, Métabief, France, 10-12 Octobre 2018., pages 75–84, 2018. (Cité en page 28.)

[Météo France 2019] Météo France. *Données SYNOP essentielles OMM*, 2019. https://donneespubliques.meteofrance.fr/?fond=produit&id_produit=90&id_rubrique=32. (Cité en pages 45 et 126.)

[Najjar *et al.* 2017] Amro Najjar, Yazan Mualla, Olivier Boissier et Gauthier Picard. *AQUA-Man : QoE-driven cost-aware mechanism for SaaS acceptability rate adaptation*. In Proceedings of the International Conference on Web Intelligence, Leipzig, Germany, August 23-26, 2017, pages 331–339, 2017. (Cité en page 36.)

[Niu *et al.* 2018] Lei Niu, Fenghui Ren et Minjie Zhang. *Feasible Negotiation Procedures for Multiple Interdependent Negotiations*. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018, pages 641–649, 2018. (Cité en page 37.)

[Nongaillard & Mathieu 2011] Antoine Nongaillard et Philippe Mathieu. *Reallocation Problems in Agent Societies : A Local Mechanism to Maximize Social Welfare*. J. Artificial Societies and Social Simulation, vol. 14, no. 3, 2011. (Cité en page 29.)

[Parsons *et al.* 1998] Simon Parsons, Carles Sierra et Nicholas R. Jennings. *Agents That Reason and Negotiate by Arguing*. J. Log. Comput., vol. 8, no. 3, pages 261–292, 1998. (Cité en page 29.)

[Penmatsa & Chronopoulos 2011] Satish Penmatsa et Anthony T. Chronopoulos. *Game-theoretic static load balancing for distributed systems.* J. Parallel Distrib. Comput., vol. 71, no. 4, pages 537–555, 2011. (Cité en pages 20, 22 et 26.)

[Pinedo 2008] Michael L. Pinedo. Scheduling. theory, algorithms, and systems. third edition. Springer, 2008. (Cité en pages 3, 14 et 15.)

[Potiron *et al.* 2009] Katia Potiron, Patrick Taillibert et Amal El Fallah-Seghrouchni. *Agents autonomes : quelles conséquences sur les fautes ?* In Systèmes Multi-Agents, Génie logiciel multi-agents - JFSMA 09 - Dix Septièmes Journées Francophones sur les Systèmes Multi-Agents, Lyon, France, October 19-21, 2009, pages 23–34, 2009. (Cité en page 123.)

[Pruitt 1981] DG Pruitt. *Negotiation Behavior*, 1981. (Cité en page 27.)

[Robu *et al.* 2005] Valentin Robu, D. J. A. Somefun et Johannes A. La Poutré. *Modeling complex multi-issue negotiations using utility graphs.* In 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands, pages 280–287, 2005. (Cité en page 29.)

[Rosenschein & Zlotkin 1994] Jeffrey S. Rosenschein et Gilad Zlotkin. Rules of encounter - designing conventions for automated negotiation among computers. MIT Press, 1994. (Cité en page 31.)

[Rubinstein 1982] Ariel Rubinstein. *Perfect equilibrium in a bargaining model.* Econometrica : Journal of the Econometric Society, pages 97–109, 1982. (Cité en page 31.)

[Schelling 1980] Thomas C Schelling. *The Strategy of Conflict, 1960.* Harvard Business School Press : Boston, MA, 1980. (Cité en page 27.)

[Schillo *et al.* 2002] Michael Schillo, Christian Kray et Klaus Fischer. *The eager bidder problem : a fundamental problem of DAI and selected solutions.* In The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings, pages 599–606, 2002. (Cité en page 36.)

[Sen 1970] AK Sen. *Collective Choice and Social Welfare North-Holland.* 1970. (Cité en page 34.)

[Serugendo *et al.* 2003] Giovanna Di Marzo Serugendo, Noria Foukia, Salima Hassas, Anthony Karageorgos, Soraya Kouadri Mostéfaoui, Omer F. Rana, Mihaela Ulieru, Paul Valckenaers et Chris van Aart. *Self-Organisation : Paradigms and Applications.* In Engineering Self-Organising Systems, Nature-Inspired Approaches to Software Engineering [revised and extended papers presented at the Engineering Self-Organising Applications Workshop, ESOA 2003, held at AAMAS 2003 in Melbourne, Australia, in July 2003 and selected invited papers from leading researchers in self-organisation], pages 1–19, 2003. (Cité en page 49.)

[Shehory & Kraus 1998] Onn Shehory et Sarit Kraus. *Methods for Task Allocation via Agent Coalition Formation.* Artif. Intell., vol. 101, no. 1-2, pages 165–200, 1998. (Cité en pages 23 et 26.)

[Slagter *et al.* 2013] Kenn Slagter, Ching-Hsien Hsu, Yeh-Ching Chung et Daqiang Zhang. *An improved partitioning mechanism for optimizing massive data analysis using MapReduce.* The Journal of Supercomputing, vol. 66, no. 1, pages 539–555, 2013. (Cité en page 47.)

[Smith 1980] Reid G. Smith. *The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver.* IEEE Trans. Computers, vol. 29, no. 12, pages 1104–1113, 1980. (Cité en pages 31 et 68.)

[Stahl 1972] Ingolf Stahl. *Bargaining Theory (Stockholm School of Economics).* Stockholm, Sweden, 1972. (Cité en page 31.)

[Su *et al.* 2018] Jiafu Su, Meng Wei et Aijun Liu. *A Robust Predictive-Reactive Allocating Approach, Considering Random Design Change in Complex Product Design Processes.* Int. J. Comput. Intell. Syst., vol. 11, no. 1, pages 1210–1228, 2018. (Cité en pages 22 et 26.)

[Turner *et al.* 2018a] Joanna Turner, Qinggang Meng, Gerald Schaefer et Andrea Soltoggio. *Distributed Strategy Adaptation with a Prediction Function in Multi-Agent Task Allocation.* In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018, pages 739–747, 2018. (Cité en pages 23 et 26.)

[Turner *et al.* 2018b] Joanna Turner, Qinggang Meng, Gerald Schaefer et Andrea Soltoggio. *Fast consensus for fully distributed multi-agent task allocation.* In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, pages 832–839, 2018. (Cité en page 23.)

[Verma *et al.* 2011] Abhishek Verma, Ludmila Cherkasova et Roy H. Campbell. *ARIA : automatic resource inference and allocation for mapreduce environments.* In Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011, pages 235–244, 2011. (Cité en page 47.)

[Vernica *et al.* 2012] Rares Vernica, Andrey Balmin, Kevin S. Beyer et Vuk Ercegovac. *Adaptive MapReduce using situation-aware mappers.* In 15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings, pages 420–431, 2012. (Cité en page 49.)

[Verrons 2004] Marie-Hélène Verrons. *GeNCA : un modèle général de négociation de contrats entre agents. (GeNCA : a general contract-based negotiation between agents model).* PhD thesis, Lille University of Science and Technology, France, 2004. (Cité en page 31.)

[Wagner *et al.* 1999] Israel A. Wagner, Michael Lindenbaum et Alfred M. Bruckstein. *Distributed covering by ant-robots using evaporating traces.* IEEE Trans. Robotics and Automation, vol. 15, no. 5, pages 918–933, 1999. (Cité en pages 23 et 26.)

[Walsh & Wellman 1998] William E. Walsh et Michael P. Wellman. *A Market Protocol for Decentralized Task Allocation.* In Proceedings of the Third International Conference on Multiagent Systems, ICMAS 1998, Paris, France, July 3-7, 1998, pages 325–332, 1998. (Cité en pages 23 et 26.)

[White 2015] Tom White. Hadoop - the definitive guide : Storage and analysis at internet scale (4. ed., revised & updated). O'Reilly, 2015. (Cité en pages 39, 47 et 50.)

[Wolf *et al.* 2012] Joel L. Wolf, Andrey Balmin, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Sujay Parekh, Kun-Lung Wu et Rares Vernica. *On the optimization of schedules for MapReduce workloads in the presence of shared scans.* VLDB J., vol. 21, no. 5, pages 589–609, 2012. (Cité en page 49.)

[Yahoo ! 2019] Yahoo ! *Yahoo ! Webscope Datasets*, 2019. https://webscope.sandbox.yahoo.com. (Cité en page 127.)

[Yin *et al.* 2007] Peng-Yeng Yin, Shiuh-Sheng Yu, Pei-Pei Wang et Yi-Te Wang. *Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization.* Journal of Systems and Software, vol. 80, no. 5, pages 724–735, 2007. (Cité en pages 21, 24 et 26.)

[Zaharia *et al.* 2012] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker et Ion Stoica. *Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, pages 15–28, 2012. (Cité en page 48.)

[Zaharia *et al.* 2016] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker et Ion Stoica. *Apache Spark : a unified engine for big data processing.* Commun. ACM, vol. 59, no. 11, pages 56–65, 2016. (Cité en pages 39 et 48.)

[Zhang *et al.* 1997] Yongbing Zhang, Hisau Kameda et S-L Hung. *Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems.* IEE Proceedings-Computers and Digital Techniques, vol. 144, no. 2, pages 100–106, 1997. (Cité en page 20.)

Négociation multi-agents pour la réallocation dynamique de tâches

Résumé :

Le problème $R_m||C_{max}$ consiste à allouer un ensemble de tâches à m agents de sorte à minimiser le *makespan* de l'allocation, c'est-à-dire le temps d'exécution de l'ensemble des tâches. Ce problème est connu pour être NP-dur dès que les tâches sont allouées à deux agents ou plus ($m \geq 2$). De plus, il est souvent admis que le coût d'une tâche est précisément estimé pour un agent et que ce coût ne varie pas au cours de l'exécution des tâches. Dans cette thèse, je propose une approche décentralisée et dynamique pour l'amélioration d'une allocation de tâches. Ainsi, à partir d'une allocation initiale et pendant qu'ils exécutent les tâches, les agents collaboratifs initient de multiples enchères pour réallouer les tâches qui restent à exécuter. Ces réallocations sont socialement rationnelles, c'est-à-dire qu'un agent accepte de prendre en charge une tâche initialement allouée à un autre agent si la délégation de cette tâche bénéficie à l'ensemble du système en faisant décroître le *makespan*. De plus, le dynamisme du procédé permet d'améliorer une allocation malgré une fonction de coût peu précise et malgré les variations de performances qui peuvent survenir lors de l'exécution des tâches.

Cette thèse offre un cadre formel pour la modélisation et la résolution multi-agents d'un problème de réallocation de tâches situées. Dans un tel problème, la localité des ressources nécessaires à l'exécution d'une tâche influe sur son coût pour chaque agent du système. À partir de ce cadre, je présente le protocole d'interaction des agents et je propose plusieurs stratégies pour que les choix des agents aient le plus d'impact sur le *makespan* de l'allocation courante.

Dans le cadre applicatif de cette thèse, je propose d'utiliser ce processus de réallocation de tâches pour améliorer le patron de conception MapReduce. Très utilisé pour le traitement distribué de données massives, MapReduce possède néanmoins des biais que la réallocation dynamique des tâches peut aider à contrer. J'ai donc implémenté un prototype distribué qui s'inscrit dans le cadre formel et implémente le patron de conception MapReduce. Grâce à ce prototype, je suis en mesure d'évaluer l'apport du processus de réallocation et l'impact des différentes stratégies d'agent.

Mots clés : Système multi-agents, Réallocation de tâches, Négociation, Systèmes distribués

Multi-agent negotiation for dynamic task reallocation

Abstract :

The $R_m||C_{max}$ problem consists in allocating a set of tasks to m agents in order to minimize the makespan of the allocation, i.e. the execution time of all the tasks. This problem is known to be NP-hard as soon as the tasks are allocated to two or more agents ($m \geq 2$). In addition, it is often assumed that the cost of a task is accurately estimated for an agent and that this cost does not change during the execution of tasks. In this thesis, I propose a decentralized and dynamic approach to improve the allocation of tasks. Thus, from an initial allocation and while they are executing tasks, collaborative agents initiate multiple auctions to reallocate the remaining tasks to be performed. These reallocations are socially rational, i.e. an agent agrees to take on a task initially allocated to another agent if the delegation of this task benefits to the entire system by decreasing the makespan. In addition, the dynamism of the process makes it possible to improve an allocation despite an inaccurate cost function and despite the variations of performance that can occur during the execution of tasks.

This thesis provides a formal framework for multi-agent modeling and multi-agent resolution of a located tasks reallocation problem. In such a problem, the locality of the resources required to perform a task affects its cost for each agent of the system. From this framework, I present the interaction protocol used by the agents and I propose several strategies to ensure that the choices of agents have the greatest impact on the makespan of the current allocation.

In the applicative context of this thesis, I propose to use this tasks reallocation process to improve the MapReduce design pattern. Widely used for the distributed processing of massive data, MapReduce has biases that the dynamic tasks reallocation process can help to counter. I implemented a distributed prototype that fits into the formal framework and implements the MapReduce design pattern. Thanks to this prototype, I am able to evaluate the effectiveness of the reallocation process and the impact of the different agent strategies.

Keywords : Multi-agent System, Task reallocation, Negotiation, Distributed System