Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

cnrs
dépasser les frontières

Système multi-agents adaptatif pour l'équilibrage de charge centré utilisateur

Mémoire de thèse
en vue de l'obtention du titre de
Docteur en Informatique de l'Université de Lille
Spécialité : Informatique
présenté par :

Ellie Beauprez

Centre de Recherche
en Informatique, Signal et Automatique de Lille

Soutenue le 8 Juillet 2024
Composition du jury :

Zahia Guessoum	McF HdR., Université de Reims	Rapporteure
Gauthier Picard	Pr., ONERA	Rapporteur
René Mandiau	Pr., Université de Valenciennes	Président/Examinateur
Anne-Cécile Caron	McF, Université de Lille	Examinatrice
Maxime Morge	McF HdR, Université de Lille	Directeur
Jean-Christophe Routier	Pr., Université de Lille	Examinateur

Table des matières

Remerciements **vii**

1 Introduction **1**
1.1 Contexte scientifique 1
1.2 Contributions 3
1.3 Organisation du document 4

I État de l'art **7**

2 Problème d'allocation de tâches **9**
2.1 Introduction 9
2.2 Caractéristiques 11
2.2.1 Ressources 11
2.2.2 Tâches 12
2.2.3 Exécutants 13
2.3 Allocation 15
2.4 Métriques 18
2.5 Discussion 21

3 Méthodes d'allocation **25**
3.1 Introduction 25
3.2 Méthodes centralisées 26
3.2.1 Heuristiques génériques 26
3.2.2 Programmation linéaire 26
3.2.3 Heuristiques spécifiques 29
3.2.4 Processus adaptatifs 29
3.3 Méthodes décentralisées 30
3.3.1 Méthodes orientées marché et approche à base de consensus 30
3.3.2 Coalitions 32
3.3.3 Optimisation distribuée sous contraintes 32
3.3.4 MARL 36
3.4 Discussion et positionnement 36

II Contributions **39**

4 Introduction **41**
4.1 Vue d'ensemble . . . 41
4.2 Plan des contributions . . . 42

5 Formalisation **45**
5.1 Introduction . . . 45
5.2 Problème d'allocation continue de tâches situées (STAP) . . . 45
5.3 Opérations . . . 52
5.3.1 Consommation . . . 52
5.3.2 Réallocation . . . 54
5.4 Rationalité sociale et stabilité . . . 55
5.5 Synthèse . . . 56

6 Modèle multi-agents **59**
6.1 Introduction . . . 59
6.2 Modèle des pairs . . . 60
6.3 Règle d'acceptabilité . . . 62
6.4 Stratégie de consommation . . . 63
6.4.1 Ordonnancement orienté tâche . . . 63
6.4.2 Ordonnancement orienté job . . . 64
6.5 Protocole d'offres alternées . . . 68
6.6 Stratégie de négociation . . . 70
6.6.1 Stratégie d'offre . . . 70
6.6.2 Stratégie d'acceptation . . . 75
6.6.3 Stratégie de contre-offre . . . 75
6.7 Synthèse . . . 77

7 Mise en œuvre **79**
7.1 Introduction . . . 79
7.2 Architecture d'agents . . . 80
7.3 Interactions entre agents composants . . . 81
7.3.1 Gestionnaire-Exécutant . . . 81
7.3.2 Gestionnaire-Négociateur . . . 82
7.4 Comportements . . . 83
7.4.1 Superviseur . . . 85
7.4.2 Exécutant . . . 85
7.4.3 Gestionnaire . . . 86
7.4.4 Négociateur . . . 88
7.5 Implémentation . . . 92
7.6 Synthèse . . . 93

8 Expérimentations **97**
8.1 Introduction . . . 97
8.2 Prototype . . . 98
8.3 Réallocation instantanée . . . 98
8.3.1 Évaluation de la stratégie de contre-offre . . . 98
8.3.2 Évaluation de la stratégie de délégation n-aire . . . 100
8.3.3 Comparaison avec les méthodes d'enchères de référence . . . 100

8.3.4 Comparaison avec une méthode DCOP 106
8.4 Réallocation lors de la consommation 108
8.4.1 La stratégie de réallocation améliore le *flowtime*. 110
8.4.2 La stratégie de réallocation ne pénalise pas la consommation. 111
8.4.3 La stratégie de réallocation s'adapte aux aléas d'exécution. 111
8.4.4 La stratégie de réallocation s'adapte à la libération des jobs. 113
8.5 Synthèse 114

9 Conclusion **117**
9.1 Synthèse 117
9.2 Limites 119
9.3 Perspectives 119

Bibliographie **120**

Liste des notations **127**

Liste des figures **130**

Liste des tableaux **131**

III Annexes **133**

A Comportement des agents **135**
A.1 Gestionnaire 135
A.2 Négociateur 135
A.3 Superviseur 137

B Publications **143**

Résumé **145**

Abstract **146**

Remerciements

Cette thèse est le fruit de plusieurs années de travail réalisées au sein de l'équipe SMAC du laboratoire CRIStAL. De nombreuses personnes ont contribué, directement ou indirectement, à l'aboutissement de ce travail.

Je tiens dans un premier temps naturellement à exprimer toute ma gratitude à Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. Cela a été un réel plaisir et une fierté de travailler et d'apprendre avec eux au cours de ces quatre années de thèse sous leur encadrement. Je suis particulièrement reconnaissante pour leurs précieux conseils lors de nos échanges, leur disponibilité irréprochable, leur patience à toute épreuve, leur sympathie et pour la confiance qu'ils ont placée en moi. J'ai énormément appris à leurs côtés et je ne pourrais jamais assez les remercier pour cette très belle expérience.

Je remercie très chaleureusement les membres du jury pour avoir accepté d'évaluer mes travaux. Merci à Zahia Guessoum et Gauthier Picard pour avoir accepté de rapporter ma thèse, leurs retours détaillés et minutieux ayant contribué à l'enrichissement de ce manuscrit. De même, je remercie René Mandiau pour avoir examiné mes travaux et présidé le jury.

Je tiens également à remercier les membres de l'équipe SMAC. Je pense que je n'aurais pu espérer meilleur environnement de travail pour réaliser ma thèse. Je remercie ainsi notamment Philippe Mathieu, Antoine Nongaillard, ainsi que mes camarades doctorants Jarod Vanderlynden et Jules Bompard.

J'ai réalisé l'ensemble de mes études à l'Université de Lille, la plus grande partie au sein du FIL. Je remercie ainsi les enseignants que j'y ai rencontrés pour m'avoir transmis leur savoir et leur passion, et qui ont beaucoup contribué au parcours qui m'a menée ici aujourd'hui.

Je ne peux parler de mon parcours universitaire sans penser aux nombreuses amitiés qui s'y sont liées, bien que je ne puisse pas toutes les citer nominativement.

Je dois également énormément à l'ensemble de mes proches dont le soutien et l'enthousiasme ont été des plus précieux. Merci à mes parents Corinne et René. Merci à mon oncle Claude et à ma tante Virginie. Merci à Camille qui m'a accompagnée durant tout mon parcours. Et enfin, merci à Owen, pour tout.

Chapitre 1

Introduction

Sommaire

1.1 Contexte scientifique 1
1.2 Contributions 3
1.3 Organisation du document 4

1.1 Contexte scientifique

Les travaux exposés dans ce document ont été réalisés dans l'équipe SMAC (Systèmes Multi-Agents et Comportements), au sein du groupe thématique I2C (Interaction et Intelligence Collective) dans le laboratoire CRIStAL (Centre de Recherche en Informatique, Signal et Automatique de Lille) grâce aux ressources mises à disposition par le Mésocentre de Calcul de l'Université de Lille.

Les systèmes multi-agents sont un paradigme pour la modélisation et l'implémentation de systèmes informatiques permettant de s'attaquer à des problèmes difficiles ou impossibles à résoudre par un système monolithique. Toutefois, pour aboutir à une solution optimale qui combine des objectifs individuels incompatibles, le verrou scientifique réside dans la conception des comportements individuels joués de manière asynchrone par des agents, qui se coordonnent sans connaître l'état global du système mais disposant d'un modèle local et partiel de leurs homologues.

L'équipe SMAC s'intéresse depuis plusieurs années à la négociation à base d'agents pour résoudre des problèmes d'allocation. Antoine Nongaillard (2009) a étudié notamment l'adéquation entre les différentes notions de bien-être social (utilitaire, égalitaire, élitiste, produit de Nash) et les processus de négociation. Plus récemment, Quentin Baert (2019) a contribué à l'optimisation de la répartition des tâches sur un cluster de calcul afin d'en minimiser le temps d'exécution, grâce à une méthode multi-agents de réallocation. Dans la lignée des travaux de Quentin Baert, 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.

Les sciences des données exploitent de larges volumes de données sur lesquels des calculs sont effectués en parallèle par différents nœuds. Ces applications mettent à l'épreuve l'informatique distribuée en ce qui concerne l'allocation de tâches et l'équilibrage de charge. C'est le cas de la classe d'applications pratiques que je considère dans cette thèse : le modèle de traitement le plus répandu pour traiter des données massives sur une grappe de serveurs, c'est-à-dire le patron de conception MapReduce (Dean & Ghemawat, 2008).

En effet, introduit par le langage Common Lisp (« Lisp », 2019), le patron de conception MapReduce a été popularisé par son implémentation open source dans l'écosystème Hadoop (« Apache Hadoop », 2024) pour le traitement de données massives. Ce modèle permet la parallélisation des requêtes grâce à un cluster de calcul, où chaque nœud stocke et traite localement une partie de l'ensemble des données. Cette approche a été largement étudiée par la communauté scientifique des SGBD et a été commercialisée dans des systèmes tels que Teradata (« Teradata », 2024) dès la fin des années 80.

Des systèmes comme Hive (« Apache Hive », 2023) ou le framework de calcul distribué Spark (« Apache Spark », 2018) ont été développés pour pallier l'absence de schémas dans MapReduce, ce qui permet l'optimisation des plans d'exécution. Cependant, la mise en place d'un mécanisme d'équilibrage de charge dynamique, capable de s'adapter en temps réel aux variations imprévues de l'environnement d'exécution, demeure un défi non résolu. À ma connaissance, cette problématique n'est pas encore prise en compte dans les solutions commerciales actuelles d'analyse de données telles que BigQuery (« BigQuery », 2024) ou Snowflake (« Snowflake », 2024).

Je m'intéresse à l'allocation de plusieurs jobs concurrents, soumis au fil de l'eau par différents utilisateurs et pouvant arriver à des dates différentes. Les propriétés du problème étudié ici découlent de la classe d'applications pratiques évoquées précédemment. L'objectif est de minimiser la durée moyenne de réalisation de ces jobs, appelée *flowtime* moyen, qui est une mesure de la performance du système du point de vue des jobs et donc de leur commanditaire. Chaque job est composé de plusieurs tâches. Chacune des tâches, qui consiste à traiter des données, doit être assignée à un nœud de calcul chargé de son exécution. Les ressources nécessaires à l'exécution d'une tâche, c'est-à-dire les données à traiter, sont réparties, éventuellement répliquées, parmi les nœuds. Ces ressources sont immédiatement accessibles pour un nœud s'il s'agit de ressources locales à ce nœud. Dans le cas contraire, les ressources restent accessibles, mais la tâche est plus coûteuse en raison du temps nécessaire pour collecter les ressources.

L'approche centrée « individus » est appropriée pour aborder des problèmes d'allocation intraitables en pratique, de par la combinatoire de l'ordonnancement et dont la formulation est complexe. Le paradigme multi-agents, modulaire, décentralisé et adaptatif permet la distribution d'heuristiques qui passe à l'échelle. Intrinsèquement réactives, les méthodes multi-agents d'allocation continue s'adaptent, sans nécessiter d'apprentissage ou d'exploration préalable, aux estimations inexactes des temps d'exécution, aux perturbations (comme le ralentissement des exécutants) ainsi qu'à la consommation et la libération des tâches caractéristiques d'une allocation continue.

1.2 Contributions

Je propose dans cette thèse un modèle multi-agents d'assignation tâches-exécutants où les nœuds de calcul sont contrôlés par des agents collaboratifs, appelés agents-nœuds, qui négocient des réallocations locales pour aboutir à une meilleure répartition des tâches. Ces négociations se déroulent au fil de l'exécution des tâches. Grâce à leur modèle des pairs, les agents-nœuds sont capables d'identifier des opportunités au sein de l'allocation courante pour marchander des délégations voire des échanges de tâches avec leurs semblables. Pour améliorer la réactivité (*responsiveness*) de la stratégie multi-agents qui repose sur l'exécution asynchrone de comportements individuels en interaction, le processus de négociation s'appuie sur de multiples négociations bilatérales concurrentes plutôt que sur des enchères répétées synchronisées par des commissaires priseurs. Dénué de mécanismes de prise de décision, un agent superviseur est chargé de démarrer, surveiller et d'arrêter les processus de négociation et de consommation. Il cadence les changements de phases qui alternent successivement à l'initiative des agents-nœuds. Après une phase de recherche de délégations pour atteindre des allocations de qualité dans un temps raisonnable, si les agents-nœuds détectent des minima locaux, ils déclenchent une phase de recherche d'échanges pour s'en extraire.

Le travail présenté dans ce document apporte ainsi plusieurs contributions : théorique, algorithmique, technique et expérimentale.

Contribution théorique. Je formalise le problème d'allocation de jobs soumis au fil de l'eau et constitués de tâches situées dont les coûts diffèrent selon la localisation des ressources et qui sont traitées progressivement. De plus, j'introduis un critère de rationalité des échanges locaux qui garantit la terminaison du processus ainsi que les opérations de consommation et de réallocation.

Contribution algorithmique. Je propose un modèle multi-agents qui s'appuie sur un modèle des pairs, une règle d'acceptabilité et des stratégies de réallocation et de consommation. La stratégie de réallocation identifie en continu les agents limitants et les opportunités au sein de répartitions déséquilibrées pour déclencher des négociations concurrentes et bilatérales afin de déléguer voire d'échanger des tâches. Un donneur sélectionne un sous-lot de tâches pour le déléguer à un receveur qu'il a choisi. Ce dernier peut accepter, éventuellement en partie, cette offre, la refuser, voire contre-proposer une tâche à échanger.

Contribution technique. Je propose une architecture d'agent composite qui permet la co-occurence du processus de réallocation avec celui de la consommation, via l'implémentation d'un prototype. Afin de séparer les comportements de consommation et de négociation, chaque agent-nœud est composé de trois sous-agents composants : le premier est dédié à la consommation des tâches, le deuxième gère les négociations avec les pairs et un dernier coordonne les deux précédents.

Contribution expérimentale. Je présente un ensemble d'expérimentations qui permettent de mettre en valeur mes propositions et de les comparer à d'autres méthodes. Mes expérimentations montrent que, lorsqu'elle est exécutée de manière concurrente au processus de consommation, contrairement aux méthodes d'allocation et de réallocation fondées sur des enchères (Koenig et al., 2006), ma stratégie de réallocation ne

pénalise pas la consommation et permet d'améliorer le délai moyen de réalisation. Le système s'adapte à la libération de jobs, et aux aléas d'exécutions (i.e. le ralentissement de nœuds), bien que les agents aient alors une connaissance imparfaite de l'environnement d'exécution.

1.3 Organisation du document

Le document est organisé en deux parties.

La première partie décrit l'état de l'art lié aux problématiques abordées dans cette thèse.

Le chapitre 2 définit en détail les propriétés des caractéristiques constituant un problème d'allocation de tâches : ressources, exécutants et tâches. Il formalise le problème ainsi que les différentes métriques utilisées pour la mesure de performance. Ce chapitre positionne, dans une grille d'analyse, les références bibliographiques traitant de problèmes d'allocation avec leurs caractéristiques ainsi que la fonction objectif minimisée.

Le chapitre 3 donne un aperçu des différentes méthodes existantes centralisées et décentralisées pour la résolution du problème d'allocation de tâches. Comme le choix du modèle à utiliser dépend de la nature du problème ainsi que de la nature des tâches et des exécutants, ce chapitre dresse un récapitulatif des références déjà citées avec le type de méthodes utilisées.

La seconde partie présente la proposition défendue dans cette thèse. Ces trois chapitres constituent le cœur de la proposition et des contributions.

Le chapitre 4 présente une vue d'ensemble du modèle afin de positionner chacune de mes contributions détaillées dans les chapitre suivants.

Le chapitre 5 est axé sur la modélisation formelle du problème d'allocation de tâches situées étudié dans cette thèse. Il s'appuie sur la définition du problème d'allocation classique décrit dans le chapitre 2. J'y montre que les réallocations dites « socialement rationnelles », garantissent la terminaison du processus de réallocation vers une allocation stable.

Le chapitre 6 présente mon modèle multi-agents où des agents, chacun supervisant un nœud de calcul, collaborent pour aboutir à une meilleure répartition des tâches. Je propose ici une stratégie d'agent négociant pour détecter des opportunités de délégation et d'échange de tâches.

Le chapitre 7 est axé sur la mise en œuvre des mécanismes proposés dans les chapitres 5 et 6. Pour pouvoir gérer simultanément les consommations et les réallocations des tâches, les agents sont dotés d'une architecture composite basée sur la séparation des rôles. Ainsi, chaque agent-nœud est constitué de trois « sous-agents », appelés agents composants, gérant respectivement la consommation, la réallocation et la gestion du lot de tâches.

Le chapitre 8 décrit les campagnes d'expérimentations pour l'évaluation empirique de ma stratégie multi-agents de réallocation qui la comparent, notamment, aux méthodes centralisées et décentralisées de référence. J'évalue ma méthode de réallocation indépendamment du processus de consommation puis lorsqu'elle est exécutée de manière concurrente au processus de consommation.

Le chapitre 9 synthétise mes contributions, identifie les limites du travail réalisé et en dresse les perspectives.

Première partie

État de l’art

Chapitre 2

Problème d'allocation de tâches

Sommaire

2.1 Introduction 9
2.2 Caractéristiques 11
2.2.1 Ressources 11
2.2.2 Tâches 12
2.2.3 Exécutants 13
2.3 Allocation 15
2.4 Métriques 18
2.5 Discussion 21

2.1 Introduction

Un problème d'allocation, ou d'affectation, de tâches consiste à répartir un ensemble de **tâches** parmi des **exécutants**. Le but est de répartir ces tâches de la façon la plus pertinente possible, de sorte à satisfaire un ou plusieurs objectifs comme par exemple la minimisation du temps de réponse, ou la maximisation d'un gain. Pour être exécutées, les tâches nécessitent des **ressources** pouvant être de différentes natures (données, mémoire, etc.). Ainsi leur répartition, potentiellement inégale, parmi les exécutants se reflète dans le coût des tâches pour ces derniers.

Je décris, ci-dessous, deux scénarios différents d'application pour des problèmes d'allocation de tâches.

Scénario 2.1 (Transport de colis par des agents collaboratifs).
Le problème du transport de colis qui est notamment décrit par Morge *(2019),* Weyns, Helleboogh et Holvoet *(2005) est un problème jouet qui se situe dans un environnement qui est une grille de cellules (cf. figure 2.1). Chaque cellule contient au plus une entité, qu'elle soit passive (un objet) ou active (un agent). Les colis dispersés sur la grille ainsi que la destination où ils doivent être déposés sont des objets. Une entité active est soit le corps d'un agent livreur soit une équipe, c'est-à-dire un ensemble de corps*

d'agents. La taille d'une entité active, c'est-à-dire le nombre de corps qui la constituent, détermine le poids maximal des colis qu'elle peut transporter. Ainsi, un agent seul peut porter un colis de poids 1, un groupe de deux agents peut porter un colis de poids 2, etc.

À chaque pas de simulation, les actions des entités actives, éventuellement conflictuelles, sont simultanées. Les actions réalisables par une entité active sont :

1. *le déplacement vers l'une des cases adjacentes ;*
2. *le chargement d'un colis si l'entité en a la capacité ;*
3. *le dépôt d'un colis si l'entité chargée est adjacente à la destination ;*
4. *l'agrégation avec une autre entité active consentante pour former une coalition, ou sa dissolution.*

Une tâche consiste à acheminer un colis jusqu'à la destination. L'objectif est de réaliser l'ensemble des tâches en minimisant le nombre de pas de simulation.

La figure 2.1 illustre une instance d'un problème de ce type comportant trois entités actives : un livreur bleu, un livreur vert, et une équipe composée du livreur rouge et du livreur noir. Un colis ciblé par une entité est coloré selon la couleur de celle-ci. Ainsi, sur cette figure les entités bleue et verte ciblent chacune un colis de taille 1. Le colis de couleur bordeaux, qui est de taille 2, est ciblé par l'entité composée de deux livreurs.

Scénario 2.2 (Contrôle d'une constellation de satellites).
Le contrôle d'une constellation de satellites est une application réelle du problème d'allocation de tâches, développée par Bonnet et Tessier *(2009). Le problème consiste en un ensemble de satellites répartis sur différentes orbites, comme illustré sur la figure 2.2, et dont les tâches sont l'observation de certaines zones de la surface terrestre. Ces tâches, de priorités variables, ne sont pas toujours réalisables par un seul satellite et doivent alors être découpées en sous-tâches affectées à différents satellites. Les satellites collaborent pour réaliser l'ensemble des tâches. Ils doivent gérer leurs ressources, en l'occurence la mémoire nécessaire pour stocker les images, et éviter les redondances.*

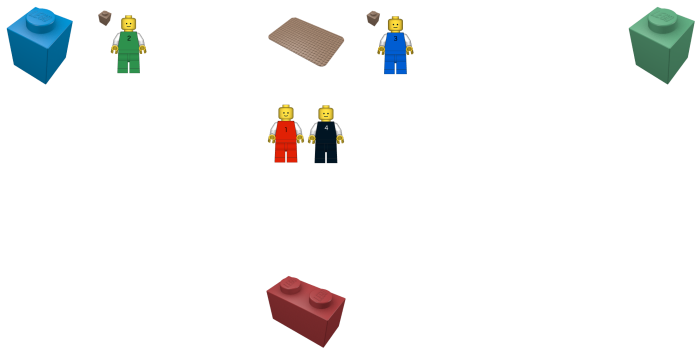


Figure 2.1 – Illustration du problème de transport de colis

Dans la section 2.2, je décris plus en détail les propriétés des caractéristiques constituant un problème d'allocation de tâches : ressources, exécutants et tâches. Je présente

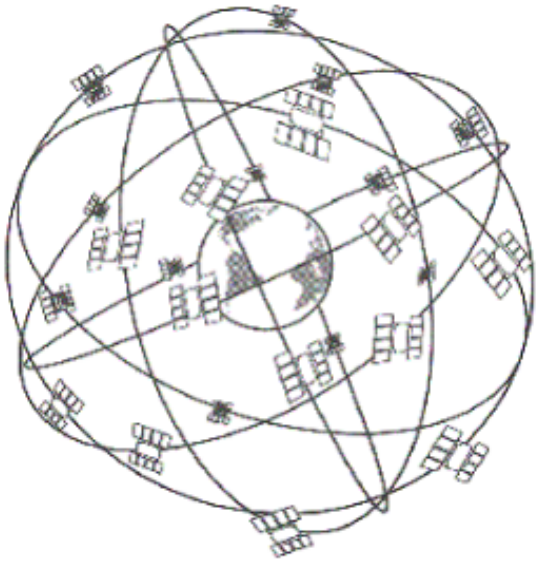


Figure 2.2 – Illustration du problème de contrôle d'une constellation de satellites (Bonnet & Tessier, 2009)

ensuite dans la section 2.3 une formalisation du problème ainsi que les différentes métriques utilisées pour la mesure de performance dans la section 2.4.

2.2 Caractéristiques

Je décris dans cette section les différentes caractéristiques composant un problème d'allocation, à savoir les ressources, les tâches et les exécutants.

2.2.1 Ressources

Afin d'être exécutée, une tâche nécessite généralement l'utilisation de ressources. Ces ressources sont présentes en quantité limitée dans le système et peuvent être des données, des capacités de calculs, de la mémoire, des objets physiques, de l'énergie, etc.

Les propriétés des ressources ont des impacts drastiques sur la nature du problème. Chevaleyre et al. (2006) s'intéressent au problème d'allocation de ressources. et dresse pour cela une typologie des propriétés des différents types de resources possibles.

- Une ressource est **discrète** ou **continue**. Dans ce dernier cas, elle est divisible selon une quantité. Par exemple, une information est une ressource discrète tandis qu'une ressource énergétique est continue.
- Les ressources peuvent être munies d'une **taille**, représentant par exemple la quantité pour de l'énergie ou des données, ou un poids pour des colis à transporter.
- Les ressources peuvent être présentes en un **unique** exemplaire ou **répliquées** en plusieurs exemplaires indistinguables. Par exemple, des données peuvent être répliquées, tandis que dans le problème du transport de colis, un colis existe en un seul exemplaire.

- Les ressources sont parfois périssables. Dans ce cas, leur taille est décroissante dans le temps. De la nourriture peut être un exemple de ressource périssable.
- Une ressource peut être **consommable**, c'est-à-dire qu'elle disparaît après avoir été utilisée par un exécutant, ou **statique** si elle ne disparaît pas et peut être réemployée pour une autre tâche. Le carburant est un exemple de ressource consommable. Des données, en revanche, sont des ressources statiques.
- Une ressource est **partageable** si elle peut être attribuée à plusieurs agents.
- Une ressource est **transférable** si elle est utilisable par un autre exécutant que celui qui la possède initialement.
- Selon la typologie du système, une ressource est **locale** à un exécutant s'il y accède directement. Alors qu'une ressource non locale et non partageable est inaccessible. Une ressource non locale mais partageable est dite **distante**.

Les ressources correspondant aux scénarios 2.1 et 2.2 sont décrites ci-dessous.

Exemple 2.1 (Transport de colis).
Dans le scénario 2.1, les colis sont des exemplaires uniques munis d'un poids qui représente leur taille. Ces ressources sont discrètes, indivisibles, non périssables. Elles sont consommables car après avoir été déposé sur la destination, un colis n'est plus accessible. Si une entité ne peut pas transmettre un colis à une autre entité et que l'on considère une équipe comme une seule et même entité alors les ressources ne sont ni partageables, ni transférables. Bien que toutes accessibles, la localité des ressources, qui se traduit par la position des colis par rapports aux entités, est variable pour les différents exécutants.

Exemple 2.2 (Contrôle d'une constellation de satellites).
Dans le scénario 2.2, les ressources sont la mémoire utilisée par les satellites pour prendre des images. La mémoire est libérée lorsque l'image est envoyée vers la station au sol. Chaque satellite disposant de sa propre mémoire, les ressources sont non partageables, non transférables et locales à chaque satellite.

2.2.2 Tâches

Les tâches sont l'élément central des problèmes d'affectation. Elles peuvent consister, par exemple, à traiter des données ou à explorer un environnement. Une tâche présente un coût pour l'exécutant chargé de la réaliser, par exemple un coût en temps.

La réalisation d'une tâche après sa libération vise à produire un résultat, éventuellement avant une date butoir. Formellement, une tâche se définit de la façon suivante :

Définition 2.1 (Tâche).
*Soit Res l'espace des résultats. Une **tâche** τ est une fonction qui associe un résultat à un ensemble de ressources $\mathcal{R}$:*

$$\tau : 2^{\mathcal{R}} \mapsto Res \tag{2.1}$$

$2^{\mathcal{R}}$ désigne l'ensemble des parties de $\mathcal{R}$.

Comme les ressources, les différentes propriétés des tâches ont un impact radical sur le problème, indépendamment de la méthode de résolution. Pinedo (2008), qui traite des différents types de problèmes d'allocation, fait les observations suivantes.

- L'**estimation du coût** d'une tâche peut être soumise à des incertitudes, des changements et des aléas. Par exemple, l'estimation du coût de tâches consistant à traiter des données par des nœuds de calcul peut être faussée par des ralentissements des nœuds de calcul.
- Lorsque des conditions préalables doivent être validées pour qu'elles soient réalisables, les tâches sont soumises à des contraintes de précédence induites comme cela peut être le cas dans un procédé de production. À l'inverse, les tâches sont dites **indépendantes** si leur résultat ne dépend pas de l'exécution des autres tâches.
- Une tâche est **préemptable** si son exécution peut être suspendue pour être reprise plus tard sans en altérer le résultat, éventuellement avec un surcoût, voire par un autre exécutant.
- Une tâche est soit atomique soit complexe, c'est-à-dire divisée en sous-tâches. On parle alors de **job**.

Je détaille ci-dessous les caractéristiques des tâches correspondant aux problèmes décrits par les scénarios 2.1 et 2.2.

Exemple 2.3 (Transport de colis).
Dans le scénario 2.1, chaque tâche consiste à transporter un colis, c'est-à-dire :

1. *atteindre une case adjacente à ce colis ;*
2. *charger le colis ;*
3. *atteindre une case adjacente à la destination du colis ;*
4. *déposer le colis.*

Comme certaines actions peuvent échouer, par exemple un déplacement vers une case occupée, ou être non sollicitées, comme une entité refoulée par une autre, le coût d'une tâche n'est qu'une estimation du nombre de pas de simulation nécessaires pour son accomplissement. Comme les entités ne peuvent porter qu'un colis à la fois et ne peuvent le déposer que sur la destination, les tâches sont atomiques, non préemptables et indépendantes.

Exemple 2.4 (Contrôle d'une constellation de satellites).
Dans le scénario 2.2, une tâche atomique consiste à enregistrer l'observation d'une zone donnée. Les tâches sont indépendantes et non préemptables. Certaines tâches ne peuvent être réalisées en une seule observation, par exemple pour couvrir une zone géographique très étendue. Il s'agit dans ce cas de tâches complexes, ou jobs, qui sont découpées en plusieurs sous-tâches atomiques.

2.2.3 Exécutants

Les exécutants sont les entités chargées d'exécuter les tâches. Les exécutants peuvent être par exemple des robots, des machines, ou des nœuds de calcul. Ils peuvent être compétents pour toutes les tâches ou seulement une partie d'entre elles.

Pinedo (2008) distingue les problèmes selon les capacités des exécutants dans la réalisation des tâches. Même si chaque exécutant a les compétences pour réaliser n'importe quelle tâche, leur efficacité dans la réalisation des tâches peut différer. On note

P_m les environnements homogènes où les exécutants présentent des capacités identiques et sont donc capables d'exécuter toutes les tâches avec la même efficacité. Dans le cas contraire, certaines spécificités font que les exécutants n'ont pas tous la même capacité pour exécuter les tâches, comme par exemple l'accès ou la proximité à des ressources nécessaires. Ce type d'environnement est noté R_m.

En résumé, le coût d'une tâche n'est pas une propriété intrinsèque, mais il peut varier d'un exécutant à un autre par exemple en fonction de l'accès aux ressources nécessaires à la tâche.

De plus, la **topologie** du réseau des exécutants est essentielle pour déterminer la faisabilité et la qualité d'une affectation. Cette topologie dépend du système auquel appartiennent les exécutants. Formellement :

Définition 2.2 (Système d'exécution).
*Le quintuplet $\mathcal{S} = \langle \Omega, \mathcal{V}, \mathcal{R}, \sigma \rangle$ est un **système** avec m exécutants $\Omega = \{\omega_1, \ldots, \omega_m\}$ dans un graphe d'accointances $\mathcal{V} \subset \Omega \times \Omega$ et muni de ressources $\mathcal{R}$ qui leur sont assignées selon une fonction $\sigma : \mathcal{R} \to 2^{\Omega}$.*
Quand les ressources sont réplicables, chacune peut être associée à un ou plusieurs exécutants.

Le **réseau d'accointances** définit quel exécutant peut interagir avec quels autres exécutants. Il peut, par exemple, être complet, en étoile, en ligne, en anneau, en grille, en arbre ou invariant d'échelle. La circulation des ressources peut être limitée, voire interdite, selon si les exécutants qui les possèdent peuvent interargir entre eux ou non, et également selon la distance les séparant.

Gerkey et Matarić (2004) présentent une taxonomie des problèmes multi-robots d'allocation de tâches. Ainsi, les tâches sont **mono-exécutant** si chacune d'elles est affectée à un seul exécutant ou **multi-exécutants** sinon. Réciproquement, les exécutants sont **mono-tâche** si chacun est affecté à une seule tâche ou **multi-tâches** si chacun réalise séquentiellement plusieurs tâches.

Je précise ci-dessous les caractéristiques des exécutants spécifiques aux problèmes décrits dans les scénarios 2.1 et 2.2.

Exemple 2.5 (Transport de colis).
Dans le scénario 2.1, une tâche de collecte est réalisable par une entité active dès l'instant où celle-ci a la taille requise, c'est-à-dire qu'elle comporte suffisamment de livreurs pour porter le colis concerné. Le coût d'une collecte dépend de la position de l'entité exécutante et varie donc au cours de la simulation. Comme elles peuvent enchaîner les collectes, les entités actives sont multi-tâches. Considérant qu'une équipe de livreurs est une seule et même entité, les tâches sont donc considérées comme mono-exécutant. Le coût d'une tâche pour un exécutant correspond ici au nombre de pas de simulation nécessaires pour récupérer le colis concerné et l'acheminer à la destination. Ce scénario décrit un problème R_m. En effet, le coût d'une tâche n'est pas le même pour toutes les entités exécutantes puisqu'il dépend de la position relative des exécutants et des colis. De plus, la capacité d'une entité exécutante à porter ou

non un colis dépend de la taille de l'entité et de la taille du colis.

Exemple 2.6 (Contrôle d'une constellation de satellites).
Dans le scénario 2.2, les exécutants sont les satellites chargés de réaliser les observations. Ces derniers ont la capacité de communiquer entre eux et d'échanger des informations à chaque fois qu'ils se rencontrent, au croisement de leurs orbites. Les exécutants sont ici hétérogènes puisque la capacité d'un satellite à effectuer une tâche dépend de son orbite et des coordonnées de la zone à observer. Ce scénario décrit donc un problème R_m. Les exécutants sont multi-tâches. Les tâches atomiques sont mono-exécutant. Les tâches multiples sont multi-exécutants.

2.3 Allocation

Après avoir décrit les ingrédients qui le constituent, je peux dans cette section introduire la notion de problème d'allocation.

Un problème d'allocation de tâches consiste à répartir des tâches parmi des exécutants en fonction de leur coût. Je me focalise ici sur les problèmes d'allocation où les tâches sont indépendantes, non préemptables, non divisibles et mono-agent.

La formalisation proposée pour un tel problème est la suivante :

Définition 2.3 (Problème d'allocation).
*Un **problème d'allocation** de tâches à des exécutants est un triplet $\mathcal{P} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ où :*

- $\mathcal{S} = \langle \Omega, \mathcal{V}, \mathcal{R}, \sigma \rangle$ *est un système d'exécution tel que défini dans la définition 2.2 ;*
- $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ *est un ensemble de n tâches ;*
- $c : \mathcal{T} \times \Omega \mapsto \mathbb{R}_+^*$ *est la fonction de coût des tâches.*

Le coût d'une tâche pour un exécutant se déduit de la localisation des ressources. Plus un exécutant a accès facilement à des ressources, moins le coût de la tâche est élevée pour celui-ci. Le coût de la tâche peut éventuellement être infini si elle n'est pas réalisable, si par exemple l'exécutant ne peut pas avoir accès aux ressources. Même si la plupart des travaux s'abstraient de la répartition des ressources qui est implicite dans la fonction de coût, cette répartition peut guider les heuristiques de (ré)allocation. Par exemple, Jiang et Li (2011) considèrent le cas d'un réseau d'accointances incomplet où il est moins coûteux d'accéder à une ressource pour un agent si ce dernier peut communiquer directement avec l'agent qui possède ces ressources. Baert et al. (2019) considère un modèle où la collecte de ressources distantes représente un coût non négligeable, et où le coût d'une même tâche sera plus faible pour un agent si les ressources nécessaires lui sont locales que si elles lui sont distantes. Dans leur taxonomie, Gerkey et Matarić (2004) font la distinction entre les problèmes d'allocation **instantanée** où toutes les tâches ont la même date de libération, et les problèmes d'allocation **continue** où des tâches peuvent être ajoutées au problème au cours du temps. Je m'intéresse ici aux problèmes d'allocation continue.

La solution d'un problème d'allocation constitué d'un système à m exécutants et d'un ensemble de n tâches, est une **allocation**, c'est-à-dire une répartition parmi les m exécutants des n tâches selon des lots, ordonnés ou non. Formellement,

Définition 2.4 (Allocation).
*Soit $\mathcal{P}$ un problème d'allocation de tâches. Une **allocation** à l'instant t est un vecteur de m lots de tâches $\overrightarrow{A_t} = (\vec{B}_{1,t}, \ldots, \vec{B}_{m,t})$ où $\vec{B}_{i,t} = (B_{i,t}, \prec_i)$ est un ensemble de tâches ($B_{i,t} \subseteq \mathcal{T}$) affectées à l'exécutant $\omega_i \in \Omega$ à l'instant t ordonné selon un ordre total strict ($\prec_i \subseteq \mathcal{T} \times \mathcal{T}$) tel que $\tau_j \prec_i \tau_k$ signifie que si $\tau_j, \tau_k \in B_{i,t}$ alors la tâche τ_j est exécutée avant la tâche τ_k par ω_i. Une allocation $\overrightarrow{A_t}$ vérifie que :*

$$\forall \tau \in \mathcal{T}\ \exists \omega_i \in \Omega,\ \tau \in B_{i,t} \tag{2.2}$$

$$\forall \omega_i \in \Omega, \forall \omega_j \in \Omega \setminus \{\omega_i\},\ B_{i,t} \cap B_{j,t} = \emptyset \tag{2.3}$$

Pour qu'une allocation soit valide, il faut que toutes les tâches soit allouées (cf équation 2.2), et que chaque tâche ne soit allouée qu'à un seul exécutant (cf équation 2.3).

Chaque tâche τ a une date de libération (*release date*) notée t^0_τ. La date à laquelle une tâche pourra être achevée dépend de l'ordre dans lequel les tâches sont traitées par les exécutants dans une allocation.
Supposant que le coût d'une tâche représente son temps d'exécution, le **délai d'attente** (*delay*) d'une tâche correspond aux temps d'exécution estimés (c'est-à-dire aux coûts) des tâches qui la précèdent dans le lot correspondant, en supposant que les exécutants ne sont jamais inactifs. Formellement :

Définition 2.5 (Délai d'attente).
*Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ à l'instant t. Soit τ la tâche affectée à l'exécutant ω_i dans l'allocation $\overrightarrow{A_t}$ pour le problème $\mathcal{P}$, le **délai d'attente** de τ est :*

$$\delta(\tau, \overrightarrow{A_t}) = \sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \omega_i) \tag{2.4}$$

La **durée de réalisation** (*completion time*) de la tâche correspond au temps d'attente avant que la tâche soit entamée plus l'estimation de son temps d'exécution. Contrairement au coût de la tâche, le délai d'attente, et donc la durée de réalisation, dépendent de l'ordre d'exécution. Formellement :

Définition 2.6 (Durée de réalisation).
*Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ à l'instant t. La **durée de réalisation** de la tâche τ libérée à la date $t^0_\tau < t$ et affectée à l'exécutant ω_i dans l'allocation $\overrightarrow{A_t}$ pour le problème $\mathcal{P}$ est :*

$$C_\tau(\overrightarrow{A_t}) = \delta(\tau, \overrightarrow{A_t}) + t - t^0_\tau + c(\tau, \omega_i) \tag{2.5}$$

Enfin, la **date d'achèvement** (*completion date*) d'une tâche est l'instant auquel elle est terminée.

Définition 2.7 (Date d'achèvement).
Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ à l'instant t. La ***date d'achèvement*** *de la tâche τ libérée à la date $t_\tau^0 < t$ et affectée à l'exécutant ω_i dans l'allocation $\overrightarrow{A_t}$ pour le problème $\mathcal{P}$ est :*

$$t_\tau^E(\overrightarrow{A_t}) = t_\tau^0 + C_\tau(\overrightarrow{A_t}) \tag{2.6}$$

Ces notions sont illustrées dans la figure 2.3 représentant le lot de tâches d'un exécutant à une date t. Les tâches ont toutes été libérées à la même date de libération t_τ^0. Les tâches qui ont été réalisées entre la date t_τ^0 et la date actuelle t sont hachurées en gris. Les tâches non hachurées représentent le lot de tâches de l'exécutant à l'instant t. Le délai d'attente de la tâche τ, représentée en rouge, est la durée entre la date actuelle t et la date à laquelle elle commencera à être traitée. Sa durée de réalisation est le temps écoulé entre la date de libération t_τ^0 et sa date d'achèvement estimée $t_\tau^E(\overrightarrow{A_t})$. Par la suite, on s'intéressera plutôt à la durée de réalisation qui est calculée relativement à la date de libération des tâches.

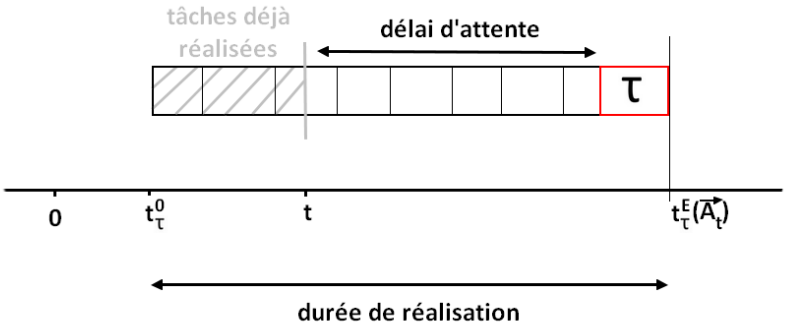


Figure 2.3 – Délai et durée de réalisation pour la tâche τ

Exemple 2.7 (Transport de colis).
La figure 2.4 représente une allocation $\overrightarrow{A_{t^0}}$ pour l'exemple de problème de transport de colis. Je considère que toutes les tâches ont la même date de libération t^0, date à laquelle on étudie l'allocation $\overrightarrow{A_{t^0}}$. Les barres représentent les lots de tâches des différentes entités exécutantes, chaque tâche consistant à acheminer un colis, et les ordonnées indiquent la charge de travail des exécutants, c'est-à-dire le nombre de pas de simulation pour réaliser l'ensemble des tâches. On observe que l'équipe constituée des livreurs n°1 et 4 est en charge du transport du colis n°5 pour un coût de 6 pas de simulation. Le livreur n°2 est chargé de transporter le colis n°1 puis le colis n°2 pour un coût total de 13 pas de simulation. Enfin le livreur n°3 transporte les colis n°3 puis n°4, pour un coût total de 11 pas de simulation.

Conformément à la définition 2.5, le délai d'attente de la tâche τ_2, consistant à transporter le colis n°2, correspond au coût des tâches assignées au livreur n°2 qui la

précédent. Cela correspond ici au nombre de pas réalisés pour transporter le colis n°1, soit $\delta(\tau_2, \overrightarrow{A_{t^0}}) = 2$.

Selon la définition 2.6, la durée de réalisation de τ_2 *correspond à son délai d'attente ainsi qu'au nombre de pas nécessaires pour sa réalisation, soit* $C_{\tau_2}(\overrightarrow{A_{t^0}}) = 2 + 11 = 13$.

Selon la définition 2.7, la date d'achèvement de τ_2 *est donc ici égale à la durée de réalisation :* $t_\tau^E(\overrightarrow{A_{t^0}}) = 13$.

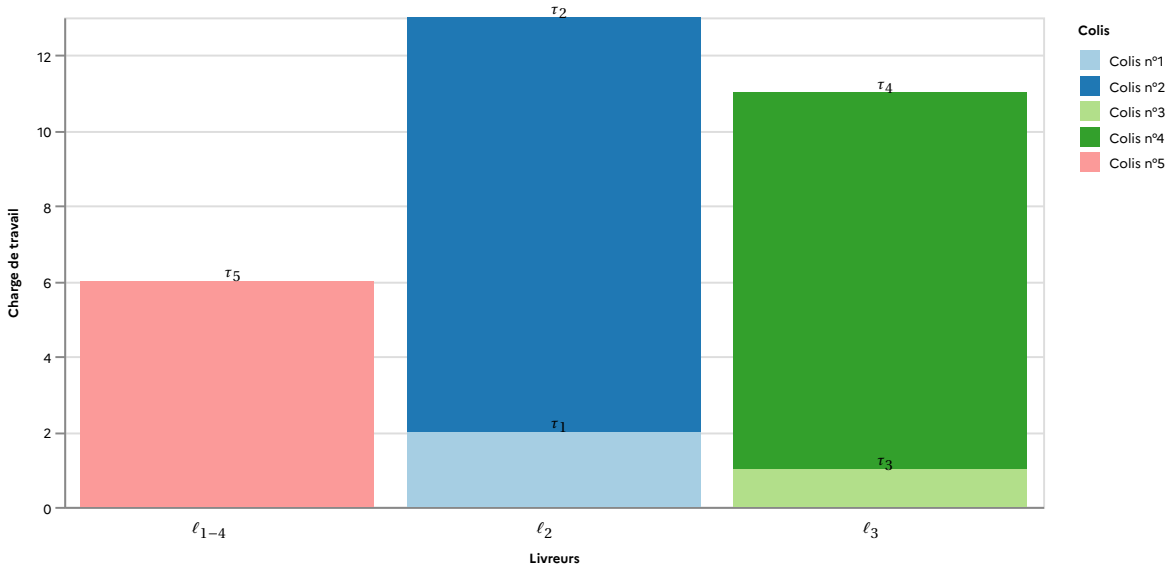


Figure 2.4 – Ordonnancement des tâches pour l'exemple 2.7

2.4 Métriques

Pour évaluer la qualité d'une allocation de tâches, je me focalise sur les mesures de performances envisageables.

Pentico (2007) et Jiang (2016) dressent un état de l'art des problèmes d'allocation de tâches et d'équilibrage de charge pour des systèmes distribués et proposent différentes métriques permettant d'évaluer la fiabilité du système comme le temps de réponse ou la durée globable de réalisation des tâches.

Les métriques peuvent être distinguées suivant différents critères. Elles peuvent prendre en compte ou non l'ordonnancement des tâches parmi les lots des exécutants. Les métriques peuvent représenter le point de vue des exécutants ou du donneur d'ordre. Elles représentent un point de vue individuel, si elles sont propres à une entité, ou collectif si elles se basent sur l'ensemble des entités. On distingue alors différents types d'objectifs. Ainsi, un objectif utilitaire visera à minimiser le coût collectif, tandis qu'un objectif égalitaire visera à faire en sorte qu'aucune entité ne soit pénalisée au profit des autres.

La charge de travail (*workload*) d'un exécutant correspond à la somme des coûts des tâches qui lui sont allouées.

Définition 2.8 (Charge de travail).
*Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ pour un instant t. La **charge de travail** d'un exécutant i pour l'allocation $\overrightarrow{A_t}$, à qui l'ensemble de tâches $B_{i,t} \subseteq \overrightarrow{A_t}$ a été alloué, est*

$$w_i(\overrightarrow{A_t}) = \Sigma_{\tau \in B_{i,t}} c(\tau, \omega_i) \ , \forall \omega_i \in \Omega \tag{2.7}$$

La charge de travail est une mesure individuelle du point de vue de l'exécutant. Elle ne prend pas en compte l'ordonnancement des tâches.

Le débit de réalisation (*throughput*) est équivalent au nombre de tâches réalisées par unité de temps.

Définition 2.9 (Débit de réalisation).
*Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ pour un instant t. Le **débit de réalisation** pour l'allocation $\overrightarrow{A_t}$ et où Ω est l'ensemble des exécutants est*

$$W(\overrightarrow{A_t}) = \Sigma_{\omega_i \in \Omega} w_i(\overrightarrow{A_t}) \tag{2.8}$$

Le débit de réalisation est une mesure utilitaire de la performance du système du point de vue des exécutants. Il ne dépend pas de l'ordonnancement.

La durée globale de réalisation (*makespan*) représente le temps nécessaire à la réalisation de l'ensemble des tâches.

Définition 2.10 (Durée globale de réalisation).
*Soit un problème $\mathcal{P}$ avec un ensemble de tâches $\mathcal{T}$ et une allocation $\overrightarrow{A_t}$ pour un instant t. La **durée globale de réalisation** pour l'allocation $\overrightarrow{A_t}$ est*

$$C_{max}(\overrightarrow{A_t}) = \max_{\tau \in \mathcal{T}} t_\tau^E(\overrightarrow{A_t}) \tag{2.9}$$

Dans le cas d'un problème d'allocation instantanée où toutes les tâches ont la même date de libération, cette durée correspond à la charge maximale des exécutants :

$$C_{max}(\overrightarrow{A_t}) = \max_{\omega_i \in \Omega} \{w_i(\overrightarrow{A_t})\} \tag{2.10}$$

Preuve 2.1.
Soit un problème $\mathcal{P}$ et une allocation $\overrightarrow{A_t}$ de tâches qui ont la même date de libération

$t^0 = 0$. *La durée globale de réalisation est :*

$$C_{max}(\overrightarrow{A_t}) = \max_{\tau \in \mathcal{T}} t_\tau^E(\overrightarrow{A_t}) \tag{2.11}$$

$$= \max_{\tau \in \mathcal{T}}(t_\tau^0 + C_\tau(\overrightarrow{A_t})) \text{ (selon la définition 2.7)} \tag{2.12}$$

$$= \max_{\tau \in \mathcal{T}} C_\tau(\overrightarrow{A_t}) \text{ car } t_\tau^0 = 0 \ \forall \tau_\in \mathcal{T} \tag{2.13}$$

$$= \max_{\tau \in \mathcal{T}}(\delta(\tau, \overrightarrow{A_t}) + c(\tau, \omega_i)) \text{ (selon la définition 2.6)} \tag{2.14}$$

$$= \max_{\tau \in \mathcal{T}}(\sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \omega_i) + c(\tau, \omega_i)) \text{ (selon la définition 2.5)} \tag{2.15}$$

$$= \max_{\tau \in \mathcal{T}}\{w_i(\overrightarrow{A_t})\} \tag{2.16}$$

La durée globale de réalisation est une mesure collective qui ne dépend pas de l'ordonnancement des tâches. L'objectif visant à minimiser la charge maximale de travail est un objectif égalitaire puisqu'il vise à minimiser la charge de l'exécutant le plus chargé. Pentico (2007) raffine cet objectif en considérant par exemple un ordre lexicographique sur les charges, la différence entre la charge maximale et la charge minimale ou la différence entre la charge maximale et la charge moyenne.

La durée moyenne de réalisation (*flowtime*) mesure le temps écoulé en moyenne entre la date de libération d'une tâche et sa date d'achèvement.

Définition 2.11 (Durée (moyenne) de réalisation).
*Soit un problème $\mathcal{P}$ avec m exécutants et n tâches et une allocation $\overrightarrow{A_t}$ pour un instant t. La **durée de réalisation** pour l'allocation $\overrightarrow{A_t}$ est :*

$$C(\overrightarrow{A_t}) = \Sigma_{\tau \in \mathcal{T}} C_\tau(\overrightarrow{A_t}) \tag{2.17}$$

*La **durée moyenne de réalisation** est :*

$$C_{mean}(\overrightarrow{A_t}) = \frac{C(\overrightarrow{A_t})}{n} \tag{2.18}$$

Contrairement à la durée globale de réalisation qui mesure l'équilibrage des charges, la durée de réalisation dépend de l'ordre d'exécution des tâches par chacun des exécutants.
C'est une mesure utilitaire de la performance du système du point de vue des tâches et donc du donneur d'ordre. C'est une mesure collective puisqu'elle s'appuie sur les temps de réalisation des tâches pour l'ensemble des exécutants. Cet objectif qui consiste à réduire les temps de réponse, c'est-à-dire les délais d'attente, peut être raffiné en distinguant les différents types de tâches.

Dans l'exemple ci-dessous, je m'appuie sur le problème de transport de colis du scénario 2.1 et sur l'instance présentée dans l'exemple 2.7 pour illustrer les métriques précédemment décrites.

Exemple 2.8 (Collecte de colis).
Suivant la définition 2.7, la charge de travail d'un exécutant correspond au coût total, c'est-à-dire ici le nombre de pas de simulation, pour réaliser l'ensemble de ses tâches. Suivant l'allocation $\overrightarrow{A_{t^0}}$ à l'instant t^0 où sont libérées les tâches présentée dans la figure 2.4, les charges de travail de chaque entité sont respectivement :

- $w_{l_{1-4}}(\overrightarrow{A_{t^0}}) = c(\tau_5, \overrightarrow{A_{t^0}}) = 6$;
- $w_{l_2}(\overrightarrow{A_{t^0}}) = c(\tau_1, \overrightarrow{A_{t^0}}) + c(\tau_2, \overrightarrow{A_{t^0}}) = 2 + 11 = 13$;
- $w_{l_3}(\overrightarrow{A_{t^0}}) = c(\tau_3, \overrightarrow{A_{t^0}}) + c(\tau_4, \overrightarrow{A_{t^0}}) = 1 + 10 = 11$.

Le débit de réalisation, calculé selon la définition 2.8, vaut $W(\overrightarrow{A_{t^0}}) = 6 + 13 + 11 = 30$.
La durée globale de réalisation, ou makespan, correspond selon la définition 2.9 au temps nécessaire pour réaliser l'ensemble des tâches. Cela équivaut ici au nombre de pas de simulation écoulés lorsque toutes les tâches seront terminées, ou à la charge de travail maximale des exécutants vu que toutes les tâches ont ici la même date de libération, soit $C_{max}(\overrightarrow{A_{t^0}}) = \max_{\omega_i \in \Omega}\{w_i(\overrightarrow{A_{t^0}})\} = 13$.

La durée moyenne de réalisation des tâches, ou flowtime, obtenue en calculant la moyenne des durées de réalisation des tâches selon la définition 2.17, vaut $C_{mean}(\overrightarrow{A_{t^0}}) = \frac{2+13+1+11+6}{5} = \frac{33}{5} = 6.6$.

2.5 Discussion

J'ai présenté et défini dans ce chapitre les notions nécessaires à l'étude du problème d'allocation de tâches. Les ingrédients constituant d'un problème d'allocation sont les **tâches**, les **exécutants** chargés de réaliser ces tâches, et les **ressources** nécessaires pour les réaliser. Les tâches, les ressources et les exécutants présentent chacun différentes caractéristiques, dépendantes du problème.

Pour évaluer la solution d'un problème d'allocation de tâches, différentes métriques peuvent être utilisées. Ces métriques, qui prennent ou non en compte l'ordonnancement des tâches, se placent du point de vue de l'exécutant ou du donneur d'ordre. Elles sont individuelles ou collectives. Dans ce dernier cas, un objectif utilitaire vise à minimiser le coût collectif, tandis qu'un objectif égalitaire vise à faire en sorte qu'aucune entité ne soit pénalisée au profit des autres.

Le tableau 2.1 liste plusieurs références traitant de problèmes d'allocation avec leurs caractéristiques ainsi que la fonction objectif minimisée. On peut remarquer que dans la plupart des travaux, la nature des ressources est implicite. Contrairement aux autres travaux mentionnés, Shehory et Kraus (1998) considèrent des tâches multi-exécutants dans leurs travaux sur la formation de coalitions. La majorité des travaux mentionnés adoptent le point de vue des exécutants ($W(\overrightarrow{A_t})$).

Dans le problème étudié dans cette thèse (noté STAP), je considère des jobs dont il est question de minimiser la durée de réalisation moyenne. Chaque job est composé de plusieurs tâches. Ces tâches sont mono-exécutant, indépendantes, non divisibles et non préemptables. Les exécutants sont multi-tâches et tous peuvent communiquer

	Ressources	**Tâches**	**Exécutants**	**Objectif**
Kuhn, 1955	—	n mono-exécutant	R_n mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Conway, Maxwell et Miller, 1967	—	n mono-exécutant	P_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Bruno, Coffman et Sethi, 1974	—	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$
Ibarra et Kim, 1977	—	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$
Mills-Tettey, Stentz et Dias, 2007	—	n mono-exécutant	R_n mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Su, Wei et Liu, 2018	—	n mono-exécutant	R_m mono-tâche	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$
Shehory et Kraus, 1998	—	n multi-exécutants	R_m mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Enright et Wurman, 2011	consommables répliquées	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$ $\oplus\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Baert et al., 2019	transférables duplicables	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$
Turner et al., 2018	limitées	n mono-exécutant	R_m en ligne multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
Schaerf, Shoham et Tennenholtz, 1995	—	n mono-exécutant	P_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$
STAP	transférables duplicables	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$

Table 2.1 – Grille d'analyse des caractéristiques pour différents problèmes d'allocation

entre eux. Chaque exécutant est capable de réaliser n'importe quelle tâche, mais son efficacité, qui se traduit par le coût d'une tâche pour un exécutant, dépend de la localité des ressources. Ces ressources sont partageables et duplicables. Ce problème, noté $R_m||C_{mean}$, consiste donc à allouer n tâches à m exécutants hétérogènes pour minimiser le coûts des jobs. Le point de vue adopté est utilitaire vis-à-vis des donneurs d'ordre, c'est-à-dire les utilisateurs soumettant les jobs, et égalitaire vis-à-vis des exécutants.

Après avoir défini les notions relatives au problème d'allocation de tâches, je présente dans les chapitres suivants les méthodes d'allocation et de réallocation existantes pour résoudre ce type de problème.

Chapitre 3

Méthodes d'allocation

Sommaire

3.1 Introduction 25
3.2 Méthodes centralisées 26
3.2.1 Heuristiques génériques 26
3.2.2 Programmation linéaire 26
3.2.3 Heuristiques spécifiques 29
3.2.4 Processus adaptatifs 29
3.3 Méthodes décentralisées 30
3.3.1 Méthodes orientées marché et approche à base de consensus 30
3.3.2 Coalitions 32
3.3.3 Optimisation distribuée sous contraintes 32
3.3.4 MARL 36
3.4 Discussion et positionnement 36

3.1 Introduction

De nombreuses méthodes ont été proposées pour allouer des tâches à des exécutants. Leur choix dépend de la nature du problème à traiter.

Les méthodes existantes peuvent être classées suivant deux grandes catégories : les méthodes centralisées et les méthodes décentralisées. Dans un processus **centralisé**, l'allocation est déterminée par une seule entité disposant de toutes les informations. Cela permet l'obtention d'une solution optimale, mais une telle méthode peut, en contre-partie, créer un goulot d'étranglement au niveau du système. Dans un processus **décentralisé**, l'allocation est le fruit d'interactions locales entre les agents, chacun disposant d'une partie des informations. Cela permet la distribution d'heuristiques pour des problèmes impraticables autrement et facilite la réactivité du système ainsi que le passage à l'échelle. En contre-partie, la solution obtenue est souvent sous-optimale, les agents ayant une connaissance partielle de l'environnement, et les coûts de communication entre les agents peuvent être non négligeables.

Les processus d'allocation se distinguent également par leur nature **instantanée** ou **continue**. Contrairement aux processus d'allocation instantanée, un processus d'allocation continue s'adapte à l'état courant du système au cours de l'exécution. De tels processus sont cependant plus complexes à mettre en place.

Il est important de distinguer les méthodes d'**allocation** et celles de **réallocation**. Dans le premier cas, il est question de générer une allocation pour résoudre un problème, dans le second il est question de modifier une allocation déjà existante pour améliorer la solution. Certaines méthodes présentées peuvent s'appliquer ou non aux deux cas. Il est à noter que réallocation et allocation continue sont deux notions différentes. Certaines méthodes utilisent la réallocation pour la génération de la solution, mais ne font pas forcément de l'allocation continue.

Je présente, dans un premier temps, différentes méthodes centralisées dans la section 3.2, puis les méthodes décentralisées dans la section 3.3. La portée et les limites de ces méthodes sont discutées dans la section 3.4.

3.2 Méthodes centralisées

J'aborde dans cette section des méthodes de programmation linéaire, différentes heuristiques, l'optimisation sous contraintes ainsi que plusieurs méthodes dynamiques.

3.2.1 Heuristiques génériques

Résoudre un problème d'allocation de tâches revient à la recherche d'un optimal dans un espace de solutions, ce dernier étant l'ensemble des allocations possibles. Un tel problème peut être résolu de manière approchée à l'aide d'heuristiques basées sur la recherche locale, telles que la méthode d'escalade ou le recuit simulé cité par Pinedo (2008). De telles méthodes consistent à explorer le voisinage de la solution courante (Russell & Norvig, 2009). Les voisins d'une allocation sont alors l'ensemble des réallocations possibles. Cependant, le grand nombre de réallocations possibles dans de tels problèmes, surtout quand le nombre de tâches est important, rend difficile l'application de ce type de méthodes qui ne passent pas l'échelle. L'avantage de ces méthodes est qu'elles peuvent s'appliquer à tout type de problème et produisent des solutions accetpables, mais elles sont donc limitées de par leur complexité élevée avec un temps d'ordonnancement rédhibitoire.

3.2.2 Programmation linéaire

Un problème d'allocation de tâches peut être formalisé par programmation linéaire (LP pour *Linear Programming*). Pour cela le problème est traduit par un ensemble de variables, une fonction objectif linéaire à minimiser, par exemple la durée, et par un ensemble de contraintes.

Définition 3.1.
Soit le problème d'allocation $\mathcal{P} =< \mathcal{S}, \mathcal{T}, c >$ constitué d'un système $\mathcal{S}$ avec un ensemble Ω de m exécutants et d'un ensemble $\mathcal{T}$ de n tâches consistant à minimiser la durée de réalisation (flowtime). La répresentation de $\mathcal{P}$ par programmation linéaire est constituée de :

1. *$n \times m$ variables de décision $x_{ik} \in \{0,1\}$ telles que*

$$x_{ik} = \begin{cases} 1 & \text{si la tâche } k \text{ est allouée à l'exécutant } i, \\ 0 & \text{sinon ;} \end{cases} \tag{3.1}$$

2. *m contraintes vérifiant que chaque tâche n'est allouée qu'à un seul exécutant,*

$$\Sigma_{i \in \Omega} x_{ik} = 1; \tag{3.2}$$

3. *la fonction objectif à minimiser :*

$$\sum_{i \in \Omega} \sum_{k \in \mathcal{T}} x_{ik} \times c_{ik} \tag{3.3}$$

Les travaux de Kuhn (1955) sont parmis ceux marquant le début des développements de méthodes pour le problème d'allocation classique. L'algorithme proposé, aussi appelé méthode hongroise, est un algorithme d'optimisation combinatoire pour le problème $R_n||W$ visant à minimiser le coût total de l'allocation, soit le débit de réalisation $\mathrm{W}(\overrightarrow{\mathrm{A}_t})$ (définition 2.9), de n tâches mono-exécutant à n exécutants mono-tâche. L'algorithme permet de trouver le couplage parfait de poids minimum dans un graphe biparti. La procédure itérative peut être représentée sous forme matricielle et consiste à transformer la matrice des coûts jusqu'à l'obtention d'une solution optimale. Les étapes pour la résolution, explicitées par Munkres (1957), sont les suivantes :

1. pour chaque ligne, soustraire la plus petite valeur à l'ensemble de la ligne. On obtient alors une matrice avec au moins un zéro par ligne ;
2. répéter la même opération pour les colonnes ;
3. rayer d'un trait les colonnes et/ou les lignes contenant un zéro de sorte à couvrir tous les zéros tout en traçant le moins de traits possible ;
4. si n traits ont été tracés, alors la solution optimale est constituée des associations tâches/exécutants correspondant à la position de n zéros indépendants ;
5. sinon le nombre de traits tracés est inférieur à n, trouver la plus petite valeur qui n'est pas rayée, soustraire cette valeur à toutes les lignes qui ne sont pas rayées, et ajouter cette même valeur à toutes les colonnes rayées ;
6. revenir à l'étape 3.

La résolution se fait en temps polynomial. L'exemple 3.1 montre le fonctionnement de la méthode sur un problème simple.

Exemple 3.1 (Méthode hongroise).
Soit le problème constitué de 3 tâches et de 3 exécutants correspondant à la matrice des coûts représentée par le tableau 3.1. Les étapes de résolution par la méthode hongroise, qui sont illustrées sur la figure 3.1, sont les suivantes :

- *Dans la figure 3.1b, on soustrait la plus petite valeur de chaque ligne à l'ensemble de la ligne. Ainsi, on soustrait 8 à la première ligne, 30 à la deuxième ligne et 13 à la troisième ligne ;*
- *Dans la figure 3.1c, on fait la même opération, cette fois-ci pour les colonnes. Ainsi, on soustrait 0 à la première colonne, 3 à la deuxième colonne et 19 à la troisième colonne ;*

- *Dans la figure 3.1d, on raye des lignes ou des colonnes de sorte à couvrir tous les zéros avec le moins de traits possible. Le nombre de traits est inférieur à 3, la procédure n'est donc pas encore terminée ;*
- *Dans la figure 3.1e, on soustrait la plus petite valeur non rayée de la table, en l'occurence 3, à l'ensemble des lignes non rayées ;*
- *Dans la figure 3.1f, on ajoute cette valeur à la colonne rayée ;*
- *Dans la figure 3.1g, on raye des lignes et des colonnes comme précédemment ;*
- *Dans la figure 3.1h, le nombre de traits est égal à 3, la recherche est donc terminée. La solution est donnée par les zéros positionnés de sorte à ce qu'il n'y en ait qu'un seul par ligne et par colonne. La solution optimale est l'allocation qui associe la tâche τ_3 à l'exécutant ω_1, τ_2 à ω_2 et τ_1 à ω_3.*

	τ_1	τ_2	τ_3
$c(\tau,\omega_1)$	8	16	27
$c(\tau,\omega_2)$	30	41	62
$c(\tau,\omega_3)$	13	54	81

Table 3.1 – Le coût des tâches pour chacun des exécutants

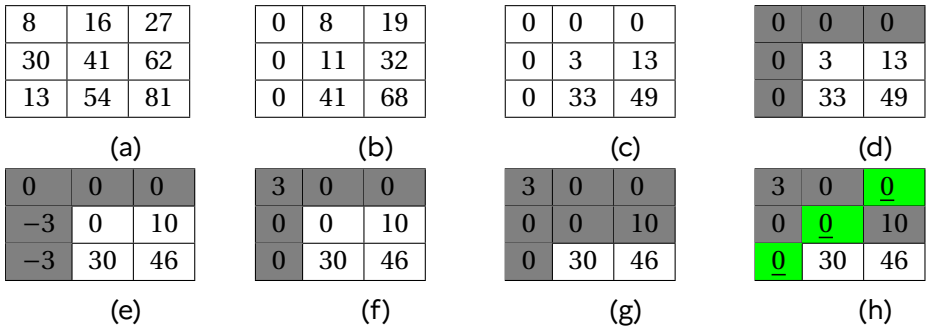


Figure 3.1 – Résolution du problème de l'exemple 3.1

Depuis ces travaux, traitant du problème où chaque exécutant est affecté à une seule tâche, de nombreuses variantes ont vu le jour pour, notamment, les problèmes où plusieurs tâches peuvent être assignées à un même exécutant ou inversement ou pour traiter des problèmes multi-objectifs.

Plus généralement, le problème $R_m||C$ consistant à minimiser la durée totale de réalisation ou *flowtime* (définition 2.11 page 20) avec m exécutants multi-tâches et n tâches mono-exécutant, dont les coûts dépendent de l'entité exécutante, se réduit à un problème d'appariement pondéré dans un graphe biparti avec n tâches et $n \times m$ positions. Comme précisé par Horn (1973), ce problème est polynomial. La complexité décrite par Bruno, Coffman et Sethi (1974) est $\mathcal{O}(max(mn^2, n^3))$.

D'une manière générale, les problèmes d'affectation sont des problèmes d'optimisation sous contraintes pour lesquels les résultats obtenus par des solveurs polyvalents tels que IBM® ILOG® CPLEX® ou Gurobi® sont médiocres car les contraintes du problème qui m'intéresse sont difficilement exprimables en fonctions linéaires, et il peut être préférable de s'orienter vers des méthodes approchées.

3.2.3 Heuristiques spécifiques

Les problèmes d'allocation de tâches peuvent être résolus de manière approchée à l'aide d'heuristiques spécialisées.

La règle du « plus court processus en premier » – *shortest processing time first* (SPT) – présentée par Conway, Maxwell et Miller (1967) est une méthode très simple pour le problème $P_1||W$ visant à minimiser le débit de réalisation pour un problème avec un seul exécutant multi-tâches et n tâches mono-exécutant.

Ce résultat se généralise au problème $P_m||W$ avec m exécutants multi-tâches si le coût d'une tâche est identique d'un exécutant à l'autre. Selon cet ordonnanceur, la plus petite tâche est affectée au premier exécutant, la seconde plus petite tâche au second exécutant, ..., la $(m+1)^{ime}$ plus petite tâche suit la plus petite tâche dans le lot du premier exécutant, la $(m+2)^{ime}$ plus petite tâche suit la seconde plus petite tâche dans le lot du deuxième exécutant, etc.

Le problème d'allocation $R_m||C_{max}$ qui consiste à minimiser la durée globale de réalisation ou *makespan* (définition 2.10 page 19) avec m exécutants multi-tâches et n tâches mono-exécutant dont les coûts dépendent de l'entité exécutante est NP-difficile d'après Horowitz et Sahni (1976).

Ibarra et Kim (1977) présentent différentes heuristiques pour le problème $R_m||C_{max}$. Notamment, « *Longest Processus Time first* » (LPT) est similaire à SPT mais consiste à affecter les tâches par ordre de coût décroissant. « *Earliest Completion Time* » (ECT) consiste à chaque étape à assigner une tâche parmi n à un exécutant de sorte que la tâche choisie ait la plus petite durée de réalisation possible. L'opération est répétée jusqu'à ce que toutes les tâches aient été allouées. Cette méthode est plus performante quand le nombre de tâches n est inférieur au nombre d'exécutants m.
Hariri et Potts (1991) présentent une heuristique à deux phases en réutilisant notamment l'heuristique ECT.

De manière générale, les heuristiques spécifiques sont faciles à mettre en place, mais peuvent produire des résultats éloignés de l'optimum.

3.2.4 Processus adaptatifs

Dans la pratique, il peut être nécessaire de s'adapter à d'éventuels changements subis par le système ou, par exemple, à des estimations de coût des tâches qui s'avèrent inexacts. Contrairement aux processus d'allocation instantanée où les données qui décrivent les caractéristiques du problème au début de l'exécution ne sont pas remises en question, des processus d'allocation continue sont dynamiques et permettent de réallouer en continu comme leur nom l'indique. Lorsque le processus est centralisé, l'allocation est produite par un exécutant ayant le rôle de superviseur. Ce superviseur reçoit des messages de ses pairs lui permettant de mettre à jour ses connaissances sur l'état du système et, ainsi, de décider d'une potentielle réallocation des tâches si nécessaire.

Ainsi, Mills-Tettey, Stentz et Dias (2007) proposent une version dynamique de l'algorithme hongrois applicable dans le cas de problèmes où les coûts sont sujets à changement. Leur méthode permet de trouver la solution optimale en réparant l'allocation

initialement obtenue avant changement en prenant en compte les nouveaux coûts.

Su, Wei et Liu (2018) proposent une approche prédictive et réactive pour le problème $R_m||C_{max}$ consistant à générer une allocation révisable à l'aide d'une heuristique puis à déterminer la séquence d'exécution grâce à une méthode prédictive basée sur des mesures de flexibilité afin de prévoir les éventuelles perturbations.

3.3 Méthodes décentralisées

Dans cette section, je passe en revue les méthodes décentralisées. Dans un premier temps les méthodes orientées marché sont présentées. Dans un deuxième temps, j'aborde les méthodes de formation de coalitions, dans le cas de problèmes où des tâches nécessitent plusieurs exécutants pour être réalisées. Je traite ensuite les méthodes d'optimisation distribuées sous contraintes avant d'évoquer les méthodes utilisant l'apprentissage par renforcement.

3.3.1 Méthodes orientées marché et approche à base de consensus

L'ordonnancement multi-agents a reçu une attention particulière pour aborder les problèmes d'équilibrage de charges dans les systèmes distribués (Jiang, 2016).
L'approche orientée marché s'inspire des théories économiques et la plupart des travaux tels que Walsh et Wellman (1998), Shehory et Kraus (1998), et Turner et al. (2018) modélisent le problème d'équilibrage de charges comme un jeu non coopératif.

Koenig et al. (2006) présentent des méthodes basées sur les enchères pour la coordination d'une équipe de robots dans le cadre d'un problème où les tâches consistent à explorer différentes zones avec des coordonnées données et où les robots cherchent à minimiser la distance parcourue.

Selon la méthode d'allocation fondée sur les enchères séquentielles à un seul article, appelée *Sequential Single-Item (SSI) Auctions*, toutes les tâches sont initialement non allouées et chaque exécutant fait une offre pour chaque tâche. L'exécutant avec la plus petite offre (en terme de coût) est affecté à la tâche correspondante. Le processus est répété jusqu'à ce que toutes les tâches soient allouées. Chaque exécutant calcule ensuite le plus court chemin pour effectuer l'ensemble des tâches qui lui sont allouées. Les robots ont des connaissances partielles et peuvent réallouer les tâches quand ils obtiennent de nouvelles informations.

Selon la méthode de réallocation fondée sur les enchères parallèles à un seul article, appelée *Parallel Single-Item (PSI) Auctions*, on considère que les tâches sont déjà allouées initialement et chaque exécutant place une offre sur chaque tâche en parallèle. Le robot possesseur d'une tâche informe le gagnant de l'enchère, c'est-à-dire le robot avec la plus petite offre. La complexité des méthodes fondées sur SSI et PSI est polynomiale selon le nombre de tâches et d'exécutants.

Choi, Brunet et How (2009) proposent des algorithmes décentralisés à base de consensus pour l'allocation de tâches. L'algorithme CBAA (pour *Consensus-Based Auction Algorithm*) pour des problèmes d'allocation $R_n||W$ avec une tâche par exécutant, se découpe en deux phases répétées itérativement :

1. un processus d'enchère où chaque agent sélectionne la tâche sur laquelle il souhaite placer une offre ;

2. un processus de consensus où les agents appliquent une stratégie de consensus pour converger vers la liste des offres validées et déterminer un gagnant en cas de conflits potentiels. Pour ce faire, chaque agent communique à ses pairs l'enchère qu'il a proposée. Les agents mettent ensuite leurs informations à jour selon le principe suivant : un agent maintient son enchère pour une tâche si son offre est la meilleure, ou il abandonne son offre dans le cas contraire.

Ces phases sont répétées jusqu'à ce que toutes les tâches aient été attribuées. Les auteurs proposent une version étendue applicable aux problèmes $R_m||W$ avec plusieurs tâches par agent, appelée CBBA (*Consensus Based Bundle Algorithm*). De la même manière, cet algorithme se découpe en deux phases répétées itérativement avec une première phase où les agents construisent le lot de tâches sur lequel ils veulent placer une enchère, et une deuxième phase dédiée à la résolution des conflits. La fonction objectif globale, étant une somme de coûts à minimiser, est utilitaire.

Dans la continuité, Turner et al. (2018) étudient un problème $R_m||W$ portant sur l'affectation en continu de tâches à une flotte de robots pour maximiser le débit de réalisation (équation 2.8) avec des ressources en carburant limitées. Grâce à l'apprentissage automatique supervisé, à partir des exécutions précédentes, les robots choisissent dynamiquement et de manière décentralisée la meilleure heuristique de sélection de tâche.

Plus récemment, Baert et al. (2019) visent un objectif égalitaire avec MASTA pour le problème $R_m||C_{max}$ en essayant de minimiser la durée globale de réalisation, c'est-à-dire le *makepan* de n tâches mono-exécutants parmi m exécutants multi-tâches hétérogènes. MASTA est un système multi-agents qui exploite la localité de ressources transférables pour réaffecter en continu des tâches afin de minimiser le *makespan*. Les agents sont des nœuds de calcul et les tâches sont des données à traiter. Le coût de ces dernières dépend de la localisation des ressources nécessaires aux tâches. Les nœuds ont accès à toutes les ressources mais une tâche sera plus coûteuse si les ressources correspondantes ne sont pas locales au nœud. Les agents effectuent des délégations de tâches de façon concurrentes à la consommation. Les négociations sont basées sur le *Contract Net Protocol* présenté par Smith (1980) et se déroulent suivant trois étapes de décision :

1. l'initiateur sélectionne une tâche à déléguer ;
2. chaque agent propose ou refuse la tâche selon que la délégation permet de faire décroître le *makespan* ou non ;
3. l'initiateur choisit comme vainqueur de l'enchère l'agent qui a la plus petite charge de travail.

Les agents prennent leurs décisions suivant leur base de connaissances partielles. Le choix de la tâche à déléguer et l'ordre selon lequel les tâches sont consommées peut se faire suivant différentes stratégies possibles. Par exemple, une approche possible est de consommer en priorité les grandes tâches locales et de déléguer les grandes tâches distantes.
À la différence de ces travaux, le cas que je traite dans ce manuscrit traite d'un problème centré utilisateur avec des jobs composés de plusieurs tâches soumis par différents utilisateurs et où l'objectif est la minimisation de la durée moyenne de réalisation des jobs, afin qu'aucun utilisateur ne soit pénalisé au profit des autres.

Contrairement aux travaux cités précédemment qui s'appuient sur le protocole *Contract Net* présenté par Smith (1980), le protocole d'offres alternées de Rubinstein (1982) sélectionne le receveur d'une tâche dès le début de l'interaction et ainsi de faire des propositions ciblées. Cette approche est idéale lorsque les agents ont connaissance du coût des tâches pour chaque agent et permet de réduire le champ de recherche et donc le coût en terme de calculs. Les propositions ciblées permettent aussi la concurrence de multiples négociations, ce qui améliore la réactivité.

3.3.2 Coalitions

Dans certains problèmes, une tâche peut nécessiter plusieurs exécutants pour être réalisée. Ce type de problème justifie alors la formation de coalitions, comme c'est le cas dans le scénario 2.1 présenté dans la section 2.1. Dans certains cas, le coût d'une tâche peut également diminuer avec le nombre d'exécutants qui lui sont affectés. En contre-partie, la formation de coalition engendre généralement un coût supplémentaire. C'est un problème complexe, comme étudié par Gueneron (2022).

Le problème de formation des coalitions pour l'allocation de tâches est équivalent à un problème de couverture ou de partitionnement par un ensemble de coût minimal. Ces problèmes sont NP-difficiles, car le nombre de coalitions potentielles est exponentiel (Cormen & Leiserson, 2010).

En particulier, Shehory et Kraus (1998) proposent des algorithmes décentralisés gloutons et *anytime* pour l'allocation de tâches via la formation de coalitions. Les auteurs supposent que les tâches sont multi-exécutants et soumises à des contraintes de précédence. Les capacités et les performances relatives des exécutants sont hétérogènes. Chaque agent est membre d'une seule coalition et une coalition est mono-tâche. Les algorithmes proposés procèdent en deux étapes :

1. le calcul distribué du coût des coalitions. L'algorithme permet à un agent de choisir l'ensemble des coalitions pour lesquelles il calcule le coût et en informe ses pairs ;
2. une procédure gloutonne, itérative et distribuée où les agents choisissent les coalitions qu'ils préfèrent et les forment. Afin de choisir les coalitions avec les coûts par agent minimaux, chaque agent cherche parmi les coalitions dont il a calculé les coûts celle dont le coût par agent est minimal et l'annonce à ses pairs, puis le groupe sélectionne une coalition. Comme un agent ne peut pas être impliqué dans une autre coalition tant que celle à laquelle il appartient n'a pas été dissoute, les coalitions doivent être disjointes et les agents impliqués sont exclus de la liste des coalitions potentielles à recalculer.

Le nombre de coalitions envisagées est réduit en limitant la taille maximale des coalitions, car les petites coalitions, moins coûteuses en communications, sont privilégiées. Les tâches du problème traité dans cette thèse ne nécessitant qu'un seul exécutant pour être traitée, les méthodes basées sur la formation de coalition n'ont pas été retenues par la suite.

3.3.3 Optimisation distribuée sous contraintes

Comme expliqué par Fioretto, Pontelli et Yeoh (2018), de nombreux problèmes d'allocation peuvent être formulés sous la forme d'un problème d'optimisation sous con-

traintes.

Un problème de satisfaction de contraintes, ou COP pour *Constraint Satisfaction Problem*, est un problème décisionnel consistant à affecter des valeurs à des variables en satisfaisant un ensemble de contraintes. Un problème d'optimisation sous contraintes (COP pour *Constraint Optimization Problem*) est une extension d'un CSP incluant une fonction objectif à optimiser, généralement la somme des coûts. Formellement :

Définition 3.2 (COP).
Un problème d'optimisation sous contraintes, ou COP, est représenté par un tuple $< X, D, C, f >$ *où :*

- $X = \{x_1, ..., x_n\}$ *est un ensemble fini de* n *variables ;*
- $D = \{D_1, ..., D_n\}$ *est l'ensemble des domaines finis de définition des variables de* X*, où* D_i *est l'ensemble des valeurs possibles pour la variable* x_i *;*
- $C = \{c_1, ..., c_m\}$ *est un ensemble fini de* m *contraintes chacunes définies sur le sous-ensemble de* k *variables* $x^i = \{x_{i1}, ...x_{ik}\}$ *telles que* $c_i : D_{i1} \times ... \times D_{ik} \rightarrow \mathbb{R}_+ \cup \infty$*;*
- f *une fonction objectif à minimiser* $f : \prod D_i \rightarrow \mathbb{R}$.

Les problèmes d'allocation ou de réallocation peuvent être représentés sous la forme d'un problème d'optimisation sous contraintes distribué (DCOP, pour *Distributed Constraint Optimization Problem*). Les DCOP sont une extension au cas multi-agents des COP évoqués dans la section 3.2. Selon l'approche multi-agents, les agents communiquent afin de décider localement de la valeur des variables qu'ils contrôlent.

La plupart des travaux se restreignent à des contraintes binaires et supposent que la fonction objectif est monotone, ce qui diffère du problème qui m'intéresse dans cette thèse.

Fioretto, Pontelli et Yeoh (2018) et Picard (2018) font une synthèse des nombreuses méthodes développées pour la recherche d'une solution optimale à un DCOP qui est un problème NP-difficile. Formellement :

Définition 3.3 (DCOP).
Un problème DCOP est défini par les auteurs par un tuple $< A, X, D, C, \mu, f >$ *où :*

- $A = \{a_1, ..., a_m\}$ *est un ensemble fini de* m *agents ;*
- $< X, D, C, f >$ *est un COP tel que défini en 3.2 ;*
- $\mu : X \rightarrow A$ *est la fonction associant chaque variable à un agent ;*
- $f : \prod D_i \rightarrow \mathbb{R}$ *est la fonction objectif à minimiser.*

L'exemple suivant illustre comment un problème d'allocation peut être formalisé par un DCOP.

Exemple 3.2 (Optimisation distribuée sous contraintes).
Le problème de l'exemple 3.1, consistant à affecter 3 tâches parmi 3 exécutants, peut être traduit sous la forme d'un problème d'optimisation distribuée sous contraintes

composé de :

- *l'ensemble de 3 agents* $A = \{a_k \mid i \in \{1,2,3\}\}$
- *l'ensemble de 9 variables* $X = \{x_{ik} \mid i \in \{1,2,3\},\ k \in \{1,2,3\}\}$*, chaque variable* x_{ik} *indiquant si la tâche* τ_k *est affectée à l'exécutant* ω_i*;*
- *l'ensemble des domaines de définition des variables* $D = \{D_{ik} | i \in \{1,2,3\}, k \in \{1,2,3\}\}$ *tel que* $D_{ik} = \{0,1\}\ \forall i,k$*. La variable* x_{ik} *a la valeur* 1 *si* τ_k *est affectée à l'exécutant* ω_i*,* 0 *sinon;*
- *l'ensemble des contraintes* $C = \{c_1, c_2, c_3\}$ *tel que :*

$$c_k = \begin{cases} 0 & \text{si } \sum_{i=0}^{3} x_{ik} = 1, \\ \infty & \text{sinon;} \end{cases} \tag{3.4}$$

  où chaque contrainte c_k *vérifie que la tâche* τ_k *est affectée à un et un seul exécutant;*

- *la fonction* μ *associant les variables aux agents telle que* $\forall x_{ik}\ \mu(x_{ik}) = a_k$*;*
- *la fonction objectif à minimiser représentant la durée de réalisation :*

$$\sum_{i=1}^{3} \sum_{k=1}^{3} x_{ik} \times c_{ik} \tag{3.5}$$

Dans un problème DCOP, les connaissances et la coordination entre les agents suivent les règles suivantes :

- une variable et son domaine de définition sont connus uniquement par l'agent qui contrôle cette variable et les voisins de cet agent ;
- chaque agent connaît seulement les fonctions de coût impliquant au moins l'une de ses variables ;
- chaque agent connaît et peut communiquer uniquement avec ses voisins.

Afin de traduire ces aspects, les problèmes sont souvent représentés sous forme d'un graphe de contraintes, d'un pseudo-arbre ou d'un graphe de facteurs.

Le choix de la méthode de résolution dépend de la nature du problème à résoudre. Comme résumé sur la figure 3.2, les algorithmes peuvent être :

- complets s'ils garantissent une solution optimale ou incomplets s'ils délivrent une solution en dessous de l'optimum au profit de meilleurs temps d'exécution ;
- synchrones s'ils nécessitent que les agents suivent des étapes simultanément et prennent leur décisions suivant un certain ordre, ou asynchrones si les agents progressent selon leur vue locale du problème et indépendamment des autres agents, comme détaillé par Lynch (1996) ;
- totalement ou partiellement décentralisés.

Les méthodes de résolution utilisées peuvent être basée sur la recherche, l'inférence ou l'échantillonnage.

Les méthodes par inférence étudient l'influence de la valeur de chaque variable sur la fonction objectif grâce à la propagation de messages de coûts. Les méthodes par recherche consistent, quant à elle, à explorer les affectations de valeurs possibles. Enfin, les méthodes par échantillonnage sont des algorithmes incomplets utilisant une approche statistique.

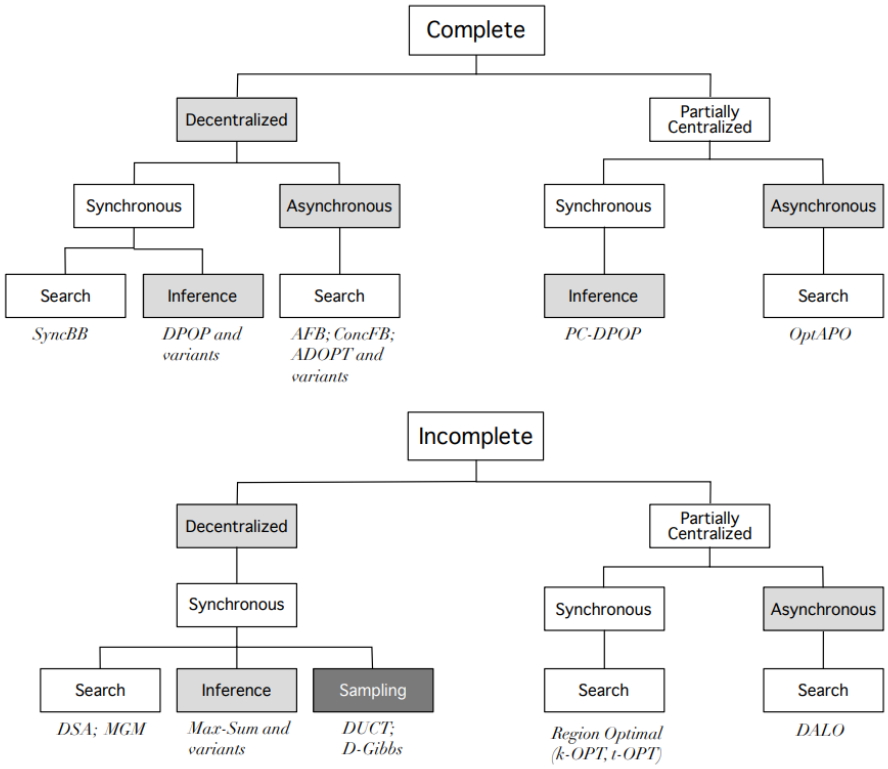


Figure 3.2 – Classification des algorithmes DCOP de Fioretto, Pontelli et Yeoh (2018)

L'algorithme DPOP (*Distributed Pseudo-tree Optimization Procedure*) proposé par Petcu et Faltings (2005a) est un algorithme décentralisé, complet, synchrone et basé sur l'inférence. Cet algorithme se déroule en trois phases :

1. les agents s'organisent dans un pseudo-arbre ;
2. Les messages contenant les vecteurs d'utilité sont propagés en partant des feuilles du pseudo-arbre ;
3. les agents calculent les valeurs optimales pour leurs variables et les propagent à leurs enfants.

L'algorithme MGM (*Maximum Gain Message*) de Maheswaran, Pearce et Tambe (2004) est un exemple d'algorithme basé sur la recherche. Il est décentralisé, incomplet et synchrone. Cet algorithme fonctionne en plusieurs étapes. Tout d'abord, chaque agent assigne des valeurs aléatoires à chacune de ses variables. Tant que la condition de terminaison n'est pas satisfaite, chaque agent répète les étapes suivantes :

1. si une nouvelle valeur est assignée, il l'envoie à ses voisins ;
2. il collecte des nouvelles valeurs de ses voisins s'il y en a ;
3. il calcule le gain, c'est-à-dire la diminution du coût, et le communique à ses voisins ;
4. il collecte les gains de ses voisins ;
5. s'il est celui qui présente le meilleur gain, il change sa valeur pour la valeur qui maximise ce gain.

Les auteurs présentent également l'algorithme MGM-2 qui est une version étendue de MGM permettant aux agents de coordonner leurs actions avec leurs voisins afin d'ob-

tenir de meilleurs résultats.

La plupart des méthodes DCOP sont des méthodes d'allocation instantanée, mais il existe tout de même des modèles d'allocation continue. Petcu et Faltings (2005b) présentent notamment une version dynamique de l'algorithme DPOP cité plus haut, nommée S-DPOP (pour *Self-stabilizing DPOP*).

La mise en œuvre des méthodes DCOP pour la réallocation de tâches présente des difficultés. Les principales sont les suivantes :

1. la représentation d'un problème réaliste sous la forme d'un DCOP, possiblement découpé en plusieurs sous-problèmes COP, nécessite une expertise de la méthode de résolution ;
2. il est parfois difficile de mettre en correspondance la fonction objectif avec une mesure de performance autre que le débit de réalisation. Par exemple, MGM2 suppose que cette fonction est monotone.

De plus, les fonctions objectif sont souvent monotones et les contraintes du problème que j'étudie ici sont difficilement exprimables dans le cadre d'un DCOP.

3.3.4 MARL

Les problèmes de réaffectation peuvent également être modélisés via des processus de décision markoviens, en particulier des processus de décision markoviens partiellement observables décentralisés – *Decentralized Partially Observed Markov Decision Process* (Dec-POMDP), abordés par Beynier et al. (2010). Le principe consiste à considérer l'environnement comme un ensemble d'états auxquels il est possible d'appliquer des actions. Ces actions ont un effet stochastique sur l'environnement qui transite alors vers un nouvel état associé à une récompense que les agents tentent de maximiser. L'optimisation d'un Dec-POMDP à horizon fini est un problème NEXPTIME. Les méthodes de résolution approchée ne peuvent être appliquées que sur de très petites instances de problèmes, elles ne passent pas à l'échelle. Au-delà de ces méthodes de planification hors-ligne, l'apprentissage multi-agents par renforcement (MARL pour *Multi-Agent Reinforcement Learning*) nécessite une connaissance parfaite de l'environnement et requiert une phase d'apprentissage, comme explicité par Schaerf, Shoham et Tennenholtz (1995). Cependant, une telle phase d'apprentissage n'est pas toujours possible, selon les problèmes. De plus, un grand volume de données peut rendre trop coûteux un prétraitement.

3.4 Discussion et positionnement

J'ai présenté dans ce chapitre un aperçu des différentes méthodes existantes centralisées et décentralisées pour la résolution du problème d'allocation de tâches. Le choix du modèle à utiliser dépend de la nature du problème ainsi que la nature des tâches et des agents.

Les méthodes centralisées présentent notamment comme limite le **passage à l'échelle**. Un contrôleur global constitue un goulot d'étranglement en matière de performance, car il doit collecter les informations d'état de l'ensemble du système en temps

réel. De plus, les problèmes d'allocation sont souvent non praticables à cause de la combinatoire des ordonnancements.

Les méthodes d'allocation instantanée, comme souvent utilisées pour les problèmes d'ordonnancement classiques présentent comme limite le manque de **réactivité**. L'estimation inexacte du temps d'exécution des tâches, aggravée par des perturbations telles que la consommation ou l'arrivée de tâches, le ralentissement des nœuds, etc., peuvent nécessiter d'importantes modifications de l'allocation existante pour qu'elle reste optimale. Plutôt que de recalculer en continu une allocation optimale, quelques modifications locales au cours de l'exécution des tâches peuvent améliorer la répartition des charges.

D'une manière générale, les problèmes d'allocation sont des problèmes d'optimisation sous contraintes pour lesquels les résultats obtenus par des solveurs polyvalents tel que IBM® ILOG® CPLEX® sont médiocres. Toutefois, ces problèmes peuvent être résolus de manière approchée par différentes heuristiques comme la méthode d'escalade ou le recuit simulé. La solution obtenue avec ces algorithmes d'approximation est acceptable mais les temps d'ordonnancement sont rédhibitoires avec un grand nombre de tâches.

L'approche décentralisée permet de surmonter la limite des solutions centralisées qu'est l'impossibilité de résoudre les problèmes à grande échelle. Le paradigme multi-agents permet notamment la distribution d'heuristiques pour des problèmes intractables à cause de la combinatoire des ordonnancements pour permettre le passage à l'échelle. De plus, l'approche centrée individu constitue un cadre conceptuel autorisant une représentation modulaire des problèmes d'allocation combinant les objectifs antagonistes des exécutants et des commanditaires (notés $C_{mean}(\overrightarrow{A_t}) \oplus W(\overrightarrow{A_t})$). Les systèmes multi-agents permettent de prendre en compte la topologie du réseau et la localisation des ressources et sont intrinsèquement réactifs.

Pour aborder les applications réelles, la difficulté réside dans la formulation de problèmes d'assignation tâches-exécutants qui sont divers et sophistiqués de par leurs ingrédients et leurs objectifs. Dans l'élaboration d'une méthode d'allocation décentralisée et adapatative, la conception des comportements individuels joués de manière asynchrone par les exécutants et les commanditaires doit aboutir à l'émergence d'allocations faisables qui combinent les objectifs des exécutants avec ceux des commanditaires.

Le tableau 3.2 dresse un récapitulatif des références déjà citées dans la section 2.1 avec le type de méthodes utilisées. Les méthodes d'allocation sont dans la partie haute du tableau, tandis que les méthodes de réallocation sont dans la partie basse. La majorité des méthodes pour des problèmes d'allocation continus utilisent l'approche décentralisée. La plupart de ces méthodes utilisent la réallocation, permettant d'être adaptatif quand de nouvelles informations reçues remettent en cause l'allocation courante.

Le cadre applicatif de l'objet de cette thèse implique de gros volumes de données avec un très grand nombre de tâches. On ne dispose pas de connaissances préalables sur ces tâches et leur coût n'est qu'une estimation qui peut s'avérer imprécise ou erronée. Pour ces raisons, cette thèse privilégie une approche réactive et décentralisée

	Ressources	Tâches	Exécutants	Objectif	Continu	Décentralisé	Technique/ Modèle
Kuhn, 1955	—	n mono-exécutant	R_n mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✗	✗	LP
Conway, Maxwell et Miller, 1967	—	n mono-exécutant	P_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✗	✗	heuristique
Bruno, Coffman et Sethi, 1974	—	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$	✗	✗	LP
Ibarra et Kim, 1977	—	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$	✗	✗	heuristique
Koenig et al., 2006	—	n mono-exécutant	R_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✗	✓	SSI
Mills-Tettey, Stentz et Dias, 2007	—	n mono-exécutant	R_n mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✗	LP + réparation d'allocation
Su, Wei et Liu, 2018	—	n mono-exécutant	R_m mono-tâche	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$	✓	✗	heuristique + approche prédictive
Shehory et Kraus, 1998	—	n multi-exécutants	R_m mono-tâche	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✓	coalition
Enright et Wurman, 2011	consommables répliquées	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$ $\oplus\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✓	marché
Koenig et al., 2006	—	n mono-exécutant	R_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✓	PSI
Baert et al., 2019	transférables duplicables	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$	✓	✓	enchères
Turner et al., 2018	limitées	n mono-exécutant	R_m en ligne multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✓	CBBA + classification
Schaerf, Shoham et Tennenholtz, 1995	—	n mono-exécutant	P_m multi-tâches	$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	✓	✓	MARL
SMASTA+	transférables duplicables	n mono-exécutant	R_m multi-tâches	$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$	✓	✓	négociation collaborative

Table 3.2 – Grille d'analyse des méthodes d'allocation (haut) et de réallocation (bas)

ne demandant aucune connaissance préalable car cela ne serait pas pertinent pour la classe d'application pratique qui nous concerne. La décentralisation permet d'éviter un goulot d'étranglement qui peut survenir dans un système centralisé en raison du volume important d'informations à traiter. Les agents collaboratifs négocient des réallocations de tâches pour corriger des allocations potentiellement déséquilibrées et pour s'ajuster aux aléas d'exécution. Ils utilisent une version à un tour du protocole d'offre alternées qui permet de faire des propositions ciblées et ainsi de limiter les coût de la négociation en terme de calculs et de permettre le déroulement de plusieurs négociations en parallèle.

Deuxième partie

Contributions

Chapitre 4

Introduction

Sommaire

4.1 Vue d'ensemble 41
4.2 Plan des contributions 42

4.1 Vue d'ensemble

Je traite dans ce manuscrit le problème d'allocation de jobs constitués de tâches situées dans le cadre du traitement de données massives.

Le problème consiste en plusieurs jobs concurrents, soumis au fil de l'eau par différents utilisateurs et pouvant arriver à des dates différentes, dont la durée moyenne de réalisation, ou *flowtime*, doit être minimisée. Chaque job est composé de plusieurs tâches. Ces tâches consistent à traiter des données et doivent être affectées à différents nœuds de calcul pour être exécutées.

Les ressources nécessaires aux tâches, c'est-à-dire les données à traiter, sont réparties parmi les nœuds. Ces ressources sont immédiatement accessibles pour un nœud si elles sont locales à ce nœud. Dans le cas contraire, les ressources restent accessibles, mais la tâche concernée coûte plus cher à réaliser pour ce nœud, en raison du temps nécessaire à ce dernier pour récupérer les ressources.

Ma thèse porte sur ce problème d'allocation continue de tâches situées. Je propose un modèle multi-agents où les nœuds de calcul sont contrôlés par des agents collaboratifs (on parlera d'**agents-nœuds**) qui négocient pour aboutir à une meilleure répartition des tâches. Le paradigme multi-agents, modulaire, décentralisé et adaptatif permet d'aborder des problèmes d'allocation dont la formulation est complexe, impraticables de par la combinatoire de l'ordonnancement, de manière adaptative sans apprentissage. Le système complexe d'assignation tâches-exécutants proposé repose sur la conception de comportements individuels des exécutants en interaction asynchrone. Mes stratégies d'agents permettent d'aboutir à des affectations faisables et performantes.

La figure 4.1 présente une vue d'ensemble du modèle proposé. On y voit l'ensemble des agents-nœuds, négocier les tâches issues de différents jobs. Chaque agent dispose de ressources qui lui sont locales, et une tâche coûtera ainsi moins cher à un agent si les ressources nécessaires sont locales à cet agent. Un agent peut déléguer une ou des tâches sans contrepartie ou échanger une tâche en échange d'une tâche de son interlocuteur. Le système dispose également d'un agent superviseur chargé de synchroniser les phases de négociation successives. Ces négociations se déroulent simultanément à l'exécution des tâches par les agents-nœuds, comme le montrent les tâches en train d'être consommées sur la figure.

Mes travaux consistent en la formalisation du problème d'allocation continue de tâches situées pour lequel je propose une stratégie multi-agents de négociation. Les agents sont capables d'identifier des opportunités dans l'allocation courante grâce à leur modèle des pairs et de déclencher des négociations bilatérales pour réassigner certaines tâches dans le but de réduire la durée moyenne de réalisation des jobs. Le processus de négociation multilatérale s'appuie sur un protocole bilatéral plutôt que sur des enchères nécessitant des points de synchronisation du système qui réduisent l'asynchronisme et donc la réactivité.

Le processus de négociation que je propose se déroule simultanément aux consommations, c'est-à-dire l'exécution des tâches par les nœuds de calcul. La consommation et la réallocation des tâches sont deux processus complémentaires. La réallocation sert à faciliter la consommation en réduisant le *flowtime* moyen des jobs, c'est-à-dire la durée moyenne pour que leurs tâches soient exécutées. Dans le même temps, la consommation de tâches modifie l'allocation courante, ce qui peut mettre en lumière de nouvelles opportunités de réallocation.

Pour pouvoir gérer simultanément ces deux mécanismes, je propose pour mes agents une architecture composite basée sur le principe de séparation des rôles. Ainsi chaque agent-nœud est un agent composite composé de trois sous-agents composants : le premier est dédié à la consommation des tâches, le deuxième gère les négociations avec les pairs, et un dernier coordonne les deux précédents.

Mes campagnes d'expérimentation permettent de valider empiriquement l'efficacité de la réactivité de ma stratégie multi-agents. En effet, ma méthode favorise un réordonnancement rapide des tâches, plutôt que la recherche de la solution optimale, ce qui permet une adaptation rapide. Mes expérimentations montre que ma stratégie de réallocation : (1) améliore le flowtime moyen grâce aux multiples réallocations bilatérales ; (2) est robuste aux aléas d'exécution comme le ralentissement de nœuds ; (3) s'adapte dès la libération de jobs.

4.2 Plan des contributions

L'ensemble de mes contributions sera organisé de la façon suivante dans ce manuscrit.

1. Dans le chapitre 5, je présente la formalisation du problème et des opérations de consommation et de réallocation. J'y définis également la notion de critère de rationalité des échanges locaux qui garantit la convergence du processus.

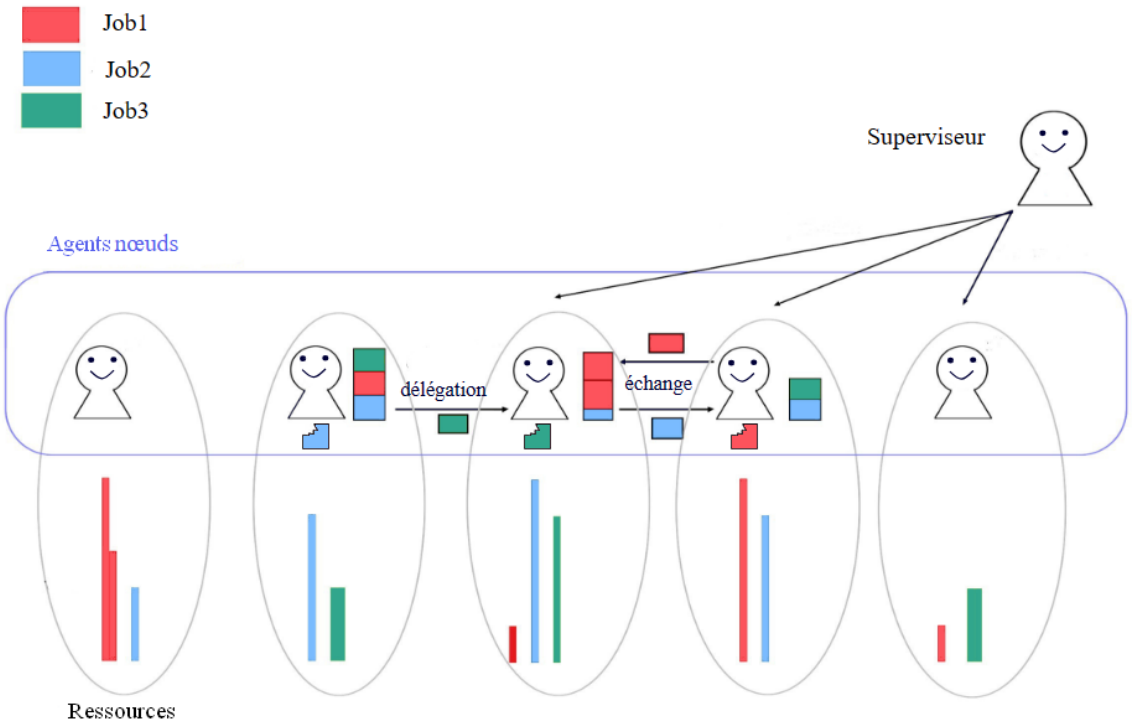


Figure 4.1 – Vue d'ensemble de l'architecture multi-agents

2. Dans le chapitre 6, je définis une stratégie d'agent négociant pour détecter des opportunités d'échange de tâches. Un donneur sélectionne un sous-lot de tâches pour le déléguer à un receveur qu'il a choisi. Ce dernier peut accepter, éventuellement en partie, cette offre, la refuser, voire contre-proposer une tâche à échanger.
3. Le chapitre 7 présente une architecture d'agent qui permet la co-occurence du processus de réallocation avec celui de la consommation. Chaque agent-nœud est composé d'une sous-agent gestionnaire qui ordonne les tâches au fur et à mesure des consommations locales par l'exécutant et des délégations émises ou reçues par le négociateur.
4. Le chapitre 8 présente une évaluation expérimentale de la stratégie qui la compare, notamment, aux méthodes centralisées et décentralisées de référence. J'évalue ma méthode de réallocation pour un problème d'allocation instantané, indépendamment du processus de consommation puis en considérant un problème d'allocation continue, lorsqu'elle est exécutée de manière concurrente au processus de consommation.

Chapitre 5

Formalisation

Sommaire

5.1 Introduction 45
5.2 Problème d'allocation continue de tâches situées (STAP) 45
5.3 Opérations 52
5.3.1 Consommation 52
5.3.2 Réallocation 54
5.4 Rationalité sociale et stabilité 55
5.5 Synthèse 56

5.1 Introduction

Ce chapitre est axé sur la modélisation formelle du problème d'allocation de tâches situées étudié dans cette thèse. Il s'appuie sur la définition du problème d'allocation classique décrit dans le chapitre 2. La particularité ici est l'introduction de la notion de jobs concurrents composés de tâches et soumis par différents utilisateurs. L'objectif consiste à minimiser la durée moyenne de réalisation de ces jobs, de sorte à ne pénaliser aucun des utilisateurs. Pour ce faire, les nœuds de calcul, contrôlés par des agents, réalisent non seulement des opérations de consommation des tâches, mais aussi des des opérations de réallocation de tâches afin de rééquilibrer leurs charges.

Dans un premier temps, je décris dans la section 5.2 la formalisation du problème d'allocation de tâches situées sur lequel repose mes contributions. La section 5.3 traite des opérations élémentaires sur les allocations. La sous-section 5.3.1 aborde l'opération de consommation représentant l'exécution des tâches par les nœuds. La sous-section 5.3.2 décrit les opérations de réallocation lors desquelles les nœuds échangent une ou plusieurs tâches. La section 5.4 se focalise sur les réallocations socialement rationnelles dont les propriétés sont démontrées.

5.2 Problème d'allocation continue de tâches situées (STAP)

Le problème d'allocation de tâches situées, en abrégé STAP pour *Situated Task Allocation Problem*, est défini par un système distribué composé de plusieurs nœuds de

calcul, un ensemble de jobs composés de plusieurs tâches et soumis par différents utilisateurs et une fonction de coût. Ces différents éléments sont définis dans cette section.

Un système distribué est composé d'un ensemble de nœuds de calcul capables de réaliser des tâches, appelés exécutants. Ces tâches requièrent des ressources, **transférables** et **non consommables**, réparties parmi différents nœuds de ressources, conformément à la classification proposée dans la section 2.2 du chapitre 2.

Définition 5.1 (Système distribué).
*Un **système distribué** est un quadruplet $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{V}, \mathcal{R} \rangle$ où :*

- *$\mathcal{P}$ est un ensemble de m nœuds de calcul ;*
- *$\mathcal{N}_r$ est un ensemble de r nœuds de ressources ;*
- *$\mathcal{V} : \mathcal{P} \times \mathcal{N}_r \to \{\top, \bot\}$ est une propriété de voisinage qui évalue si un nœud de calcul de l'ensemble $\mathcal{P}$ est local à un nœud de ressources dans $\mathcal{N}_r$;*
- *$\mathcal{R} = \{\rho_1, \ldots, \rho_k\}$ est un ensemble de ressources de tailles $|\rho_i|$. La localisation des ressources, qui sont éventuellement répliquées, est déterminée par la fonction :*

$$l : \mathcal{R} \to 2^{\mathcal{N}_r} \tag{5.1}$$

La principale différence avec le système d'exécution défini dans le chapitre 2 est ici la distinction entre l'ensemble $\mathcal{P}$ des nœuds de calcul et l'ensemble $\mathcal{N}_r$ des nœuds de ressources, tandis que dans la définition 2.2 on a un unique ensemble d'exécutants Ω, équivalent à $\mathcal{P}$ ici. Cela permet de rendre les ressources accessibles à tout instant, même si l'on souhaite désactiver certains nœuds de calcul selon les besoins, comme envisagé dans des travaux futurs.

Une ressource peut être locale ou distante d'un nœud de calcul, selon sa présence ou non sur un nœud de ressources dans le voisinage du nœud de calcul. À partir de la propriété de voisinage $\mathcal{V}$ et de la fonction de localité l, je définis le prédicat de localité :

$$\forall \nu_c \in \mathcal{P}, \forall \rho \in \mathcal{R}, \ \mathsf{local}(\rho, \nu_c) \text{ ssi } \exists \nu_r \in l(\rho) \text{ t.q. } \mathcal{V}(\nu_c, \nu_r)$$

Les ressources sont accessibles pour tous les nœuds de calcul, même celles sur les nœuds de ressources distants car la relation d'accointances est complète et totale.

Comme décrit dans la section 2.2.2 du chapitre 2, un job est un ensemble de tâches **indépendantes**, **non divisibles** et **non préemptables**. L'exécution de chaque tâche nécessite l'accès à des ressources distribuées sur les nœuds du système. L'exécution d'un job, sans date butoir, consiste à exécuter l'ensemble de ses tâches pour produire un résultat.

Définition 5.2 (Job/Tâche).
*Soit $\mathcal{D}$ un système distribué. On considère un ensemble de ℓ jobs $\mathcal{J} = \{J_1, \ldots, J_\ell\}$. Chaque **job** J_i, associé à la date de libération $t^0_{J_i}$, est un ensemble non vide de k_i tâches $J_i = \{\tau_1, \ldots, \tau_{k_i}\}$. Chaque **tâche** τ est une fonction qui associe un résultat à un ensemble de ressources : $\tau : 2^{\mathcal{R}} \mapsto Res$.*

La notion de tâche ici correspond à celle de la définition 2.1. $\mathcal{T} = \cup_{1 \leq i \leq \ell} J_i$ dénote l'ensemble des n tâches sous-jacentes à $\mathcal{J}$ et $\mathcal{R}_\tau \subseteq \mathcal{R}$ l'ensemble des ressources requises pour

la tâche τ. Par souci de concision, je noterai par la suite $\text{job}(\tau)$ le job contenant la tâche τ. Je fais l'hypothèse que le nombre de jobs est négligeable par rapport au nombre de tâches, $|\mathcal{J}| << |\mathcal{T}|$.

Le coût d'une tâche pour un nœud v_i est une estimation *a priori* de son temps d'exécution.

Définition 5.3 (Coût d'une tâche pour un nœud).
Soient $\mathcal{D}$ un système distribué et $\mathcal{T}$ un ensemble de tâches. La fonction de ***coût*** *$c : \mathcal{T} \times \mathcal{P} \mapsto \mathbb{R}_+^*$ est telle que :*

$$c(\tau, v_j) = \sum_{\rho_j \in \mathcal{R}_\tau} c(\rho_j, v_j) \tag{5.2}$$

où $c(\rho_j, v_j)$ est le coût de la collecte de la resssource ρ_j par le nœud v_j.

La fonction de coût que je considère dans ce manuscrit est telle que :

$$c(\rho_j, v_i) = \begin{cases} |\rho_j| \text{ si } \text{local}(\rho_j, v_i) \\ \kappa \times |\rho_j| \text{ avec } \kappa > 1 \text{ sinon.} \end{cases} \tag{5.3}$$

Comme la collecte de ressources distantes représente un surcoût, une tâche est plus coûteuse si les ressources nécessaires ne sont pas locales. Je considère la valeur κ comme étant arbitraire. Cette valeur sera discutée dans le chapitre 8 dédié aux expérimentations. La fonction de coût peut être étendue à un ensemble de tâches :

$$\forall \mathrm{T} \subseteq \mathcal{T},\ \mathrm{c}(\mathrm{T}, v_i) = \Sigma_{\tau \in \mathrm{T}} \mathrm{c}(\tau, v_j) \tag{5.4}$$

Après en avoir défini les ingrédients, je définis le problème d'allocation de jobs composés de tâches situées de la façon suivante.

Définition 5.4 (STAP).
Un ***problème d'allocation continue de tâches situées*** *à un instant t est un quadruplet STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ où :*

- *$\mathcal{D}$ est un système distribué avec m nœuds de calcul;*
- *$\mathcal{T}_t = \{\tau_1, \ldots, \tau_n\}$ est un ensemble de n tâches;*
- *$\mathcal{J}_t = \{J_1, \ldots, J_\ell\}$ est un partitionnement des tâches en ℓ jobs;*
- *$c : \mathcal{T}_t \times \mathcal{P} \mapsto \mathbb{R}_+^*$ est la fonction de coût.*

Cette définition est similaire au problème d'allocation défini en 2.3 du chapitre 2, à la différence que je considère que l'ensemble des tâches, qui sont regroupées dans des jobs, peut évoluer. Les tâches peuvent notamment être consommées. De fait, les ensembles de jobs et de tâches étant amenés à évoluer au cours du temps, ils seront indicés par le temps t par la suite.

Une allocation de tâches est une solution à un problème STAP. Elle consiste en une répartition des tâches dans des lots ordonnés. Le problème variant au cours du temps, en raison des consommations de tâches et des potentiels ajouts de nouveaux jobs, il est nécessaire de définir une allocation pour un instant t. Identiquement à la définition d'allocation 2.4 du chapitre 2, je définis formellement une allocation pour un problème STAP de la façon suivante :

Définition 5.5 (Allocation).
*Une **allocation** à l'instant t pour un problème STAP = $\langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ est un vecteur de m lots de tâches ordonnées $\overrightarrow{A_t} = ((B_{1,t}, \prec_1), \ldots, (B_{m,t}, \prec_m))$ où chaque lot $(B_{i,t}, \prec_i)$ est l'ensemble des tâches ($B_{i,t} \subseteq \mathcal{T}_t$) affectées au nœud ν_i à l'instant t, associé à un ordre total strict ($\prec_i \subseteq \mathcal{T}_t \times \mathcal{T}_t$). $\tau_j \prec_i \tau_k$ signifie que si $\tau_j, \tau_k \in B_{i,t}$ alors τ_j est exécutée avant τ_k par ν_i. L'allocation $\overrightarrow{A_t}$ vérifie pour l'instant t :*

$$\forall \tau \in \mathcal{T}_t,\ \exists \nu_i \in \mathcal{P},\ \tau \in B_{i,t} \tag{5.5}$$

$$\forall \nu_i \in \mathcal{P}, \forall \nu_j \in \mathcal{P} \setminus \{\nu_i\},\ B_{i,t} \cap B_{j,t} = \emptyset \tag{5.6}$$

L'équation 5.5 vérifie que toutes les tâches sont allouées et l'équation 5.6 vérifie que chacune n'est allouée qu'à un seul nœud. Par souci de concision, je noterai par la suite :

- $\vec{\mathsf{B}}_{i,t} = (\mathsf{B}_{i,t} \prec_i)$, le lot trié de ν_i ;
- $\min_{\prec_i} \mathsf{B}_{i,t}$, la prochaine tâche à exécuter par ν_i ;
- $\nu(\tau, \overrightarrow{\mathsf{A}_t})$, le nœud chargé de τ dans $\overrightarrow{\mathsf{A}_t}$.

L'objectif d'un tel problème est de réaliser au plus tôt les jobs, qui sont soumis par différents utilisateurs. Comme il est souhaitable qu'aucun job ne soit pénalisé au profit des autres, la qualité de l'allocation sera évaluée grâce à la durée moyenne de réalisation des jobs, ou *flowtime* moyen. Cette métrique mesure le temps moyen écoulé entre la date de libération des jobs et leur date d'achèvement. Afin de formaliser la durée de réalisation moyenne d'un job, il est nécessaire de définir au préalable les notions de délai d'attente et de durée de réalisation d'une tâche.

Comme dans la définition 2.5 proposée dans le chapitre 2, le délai d'attente correspond au délai d'attente à partir de l'instant courant t avant que la tâche τ ne soit traitée.

Définition 5.6 (Délai d'attente).
*Soient STAP un problème et $\overrightarrow{A_t}$ une allocation à l'instant t. Le **délai d'attente** d'une tâche $\tau \in \mathcal{T}_t$ dans le lot $\vec{B}_{i,t}$ est :*

$$\delta(\tau, \nu_i) = \sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \nu_i) \tag{5.7}$$

La durée de réalisation d'une tâche est la durée au bout de laquelle cette tâche est terminée. Elle est calculée à partir de son délai d'attente, de la date de libération de son job et de l'estimation de son temps d'exécution, c'est-à-dire son coût.

Définition 5.7 (Durée de réalisation d'une tâche).
*Soient STAP un problème et $\overrightarrow{A_t}$ une allocation à l'instant t. La **durée de réalisation** d'une tâche $\tau \in \mathcal{T}_t$ pour l'allocation $\overrightarrow{A_t}$ est :*

$$C_\tau(\overrightarrow{A_t}) = \delta(\tau, \nu(\tau, \overrightarrow{A_t})) + t - t^0_{job(\tau)} + c(\tau, \nu(\tau, \overrightarrow{A_t})) \tag{5.8}$$

Contrairement à la définition 2.6 du chapitre 2, je considère des jobs pouvant être ajoutés au fil de l'eau. Il est donc nécessaire de prendre en compte leur date de libération

dans le calcul de la durée de réalisation des tâches, comme indiqué dans l'équation 5.8.

La durée de réalisation d'un job correspond à la durée au bout de laquelle ce job est terminé, c'est-à-dire la durée de réalisation de sa tâche la plus tardive.

Définition 5.8 (Durée de réalisation d'un job).
*Soient STAP un problème et $\overrightarrow{A_t}$ une allocation à l'instant t. La **durée de réalisation** d'un job $J \in \mathcal{J}_t$ pour $\overrightarrow{A_t}$ est :*

$$C_J(\overrightarrow{A_t}) = \max_{\tau \in J}\{C_\tau(\overrightarrow{A_t})\} \tag{5.9}$$

La durée moyenne de réalisation, ou *flowtime* moyen, pour une allocation est la moyenne des durées de réalisation de l'ensemble des jobs.

Définition 5.9 (Durée moyenne de réalisation).
*Soient STAP un problème et $\overrightarrow{A_t}$ une allocation à l'instant t. La **durée moyenne de réalisation** pour $\overrightarrow{A_t}$,*

$$C_{mean}(\overrightarrow{A_t}) = \frac{1}{\ell} C(\overrightarrow{A_t})$$
$$\text{avec } C(\overrightarrow{A_t}) = \Sigma_{J \in \mathcal{J}_t} C_J(\overrightarrow{A_t}) \tag{5.10}$$

Ces notions sont illustrées à travers l'exemple suivant.

Exemple 5.1 (STAP).
Soit le problème STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ constitué d'un système distribué à 3 nœuds de calcul où doivent être réparties 9 tâches provenant de 3 jobs différents.
Le système distribué $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{V}, \mathcal{R} \rangle$ comporte 3 nœuds de calcul $\mathcal{P} = \{\nu_1, \nu_2, \nu_3\}$. Les ressources sont réparties sur 3 nœuds de ressources $\mathcal{N}_r = \{\nu_1^r, \nu_2^r, \nu_3^r\}$. La relation de voisinage entre les nœuds de calcul et les nœuds de ressources est telle que $\mathcal{V}(\nu_i, \nu_i^r) = \top$. Les ressources $\mathcal{R} = \{\rho_1, \ldots, \rho_9\}$ sont chacune répliquées sur 2 nœuds différents et réparties comme indiqué dans la figure 5.1a. Les 9 tâches $\mathcal{T}_t = \{\tau_1, \ldots, \tau_9\}$ du problème proviennent de 3 jobs $\mathcal{J}_t = \{J_1, J_2, J_3\}$ tous libérés à la date $t_0 = 0$ tels que :

- $J_1 = \{\tau_1, \tau_2, \tau_3\}$;
- $J_2 = \{\tau_4, \tau_5, \tau_6\}$;
- $J_3 = \{\tau_7, \tau_8, \tau_9\}$

La fonction de coût des tâches est donnée dans la table 5.1. Je suppose que le coût d'une tâche est proportionnel à la taille des ressources et qu'il est deux fois plus important si la ressource est non locale ($\kappa = 2$).

Je considère ici l'allocation $\overrightarrow{A_t}$ à l'instant $t = 0$, représentée sur la figure 5.1b, avec $\vec{B}_{1,t} = (\tau_5, \tau_8, \tau_3, \tau_2)$, $\vec{B}_{2,t} = (\tau_4, \tau_9)$ et $\vec{B}_{3,t} = (\tau_7, \tau_1, \tau_6)$. Selon la stratégie de consommation adoptée par les agents, chaque lot est trié par job, les jobs les moins coûteux étant prioritaires (les stratégies de consommation seront vues plus en détail dans le chapitre 6).

La durée de réalisation du job J_1, représenté en rouge sur les figures, est la durée au bout de laquelle toutes ses tâches sont terminées, ou autrement dit la durée de réalisation de sa tâche terminée le plus tardivement, en l'occurence la tâche τ_2. Le coût estimé de τ_2 pour le nœud ν_1 est $c(\tau_2, \nu_1) = 3$. Selon l'allocation proposée, τ_2 est placée dans le lot de tâches de ν_1 après les tâches τ_5, τ_8 et τ_3. Son délai d'attente est donc $\delta(\tau_2, \nu_1) = c(\tau_5, \nu_1) + c(\tau_5, \nu_1) + c(\tau_5, \nu_1) = 2 + 2 + 1 = 5$. On en déduit la durée de réalisation du job J_1 : $C_{J_1}(\overrightarrow{A_{t=0}}) = C_{\tau_2}(\overrightarrow{A_{t=0}}) = \delta(\tau_2, \nu_1) + t - t^0_{J_1} + c(\tau_2, \nu_1) = 5 + 0 - 0 + 3 = 8$.

De la même manière, les durées de réalisation des jobs J_2 et J_3 sont $C_{J_2}(\overrightarrow{A_{t=0}}) = C_{J_3}(\overrightarrow{A_{t=0}}) = 12$.

Le flowtime moyen correspondant à cette allocation est $C_{mean}(\overrightarrow{A_t}) = \frac{C_{J_1}(\overrightarrow{A_t}) + C_{J_2}(\overrightarrow{A_t}) + C_{J_3}(\overrightarrow{A_t})}{3} = \frac{8+12+12}{3} = \frac{32}{3} \simeq 10.67$.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9
$c(\tau, \nu_1)$	5	3	1	8	2	10	4	2	4
$c(\tau, \nu_2)$	10	3	2	4	2	5	2	2	8
$c(\tau, \nu_3)$	5	6	1	4	4	5	2	4	4

Table 5.1 – Le coût des tâches pour chaque nœud

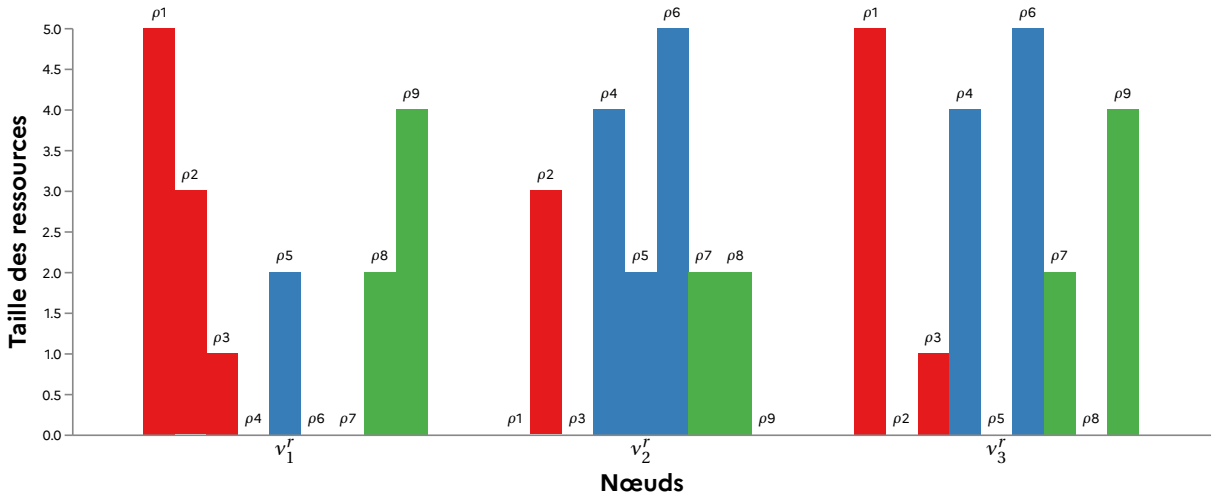
L'exemple 5.2 montre le cas d'un problème STAP où les jobs ont des dates de libérations différentes.

Exemple 5.2 (STAP avec plusieurs dates de libération).
Soit le problème STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ constitué d'un système distribué à 2 nœuds de de calcul où doivent être réparties 6 tâches provenant de 3 jobs différents. Le système distribué $\mathcal{D}$ comporte 2 nœuds de calcul $\mathcal{P} = \{\nu_1, \nu_2\}$. Les ressources sont réparties sur 2 nœuds de ressources $\mathcal{N}_r = \{\nu_1^r, \nu_2^r\}$. La relation de voisinage entre les nœuds de calcul et les nœuds de ressources est telle que $\mathcal{V}(\nu_i, \nu_i^r) = \top$. Les tâches $\mathcal{T}_t = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$ proviennent des jobs $\mathcal{J}_t = \{J_1, J_2, J_3\}$ tels que :

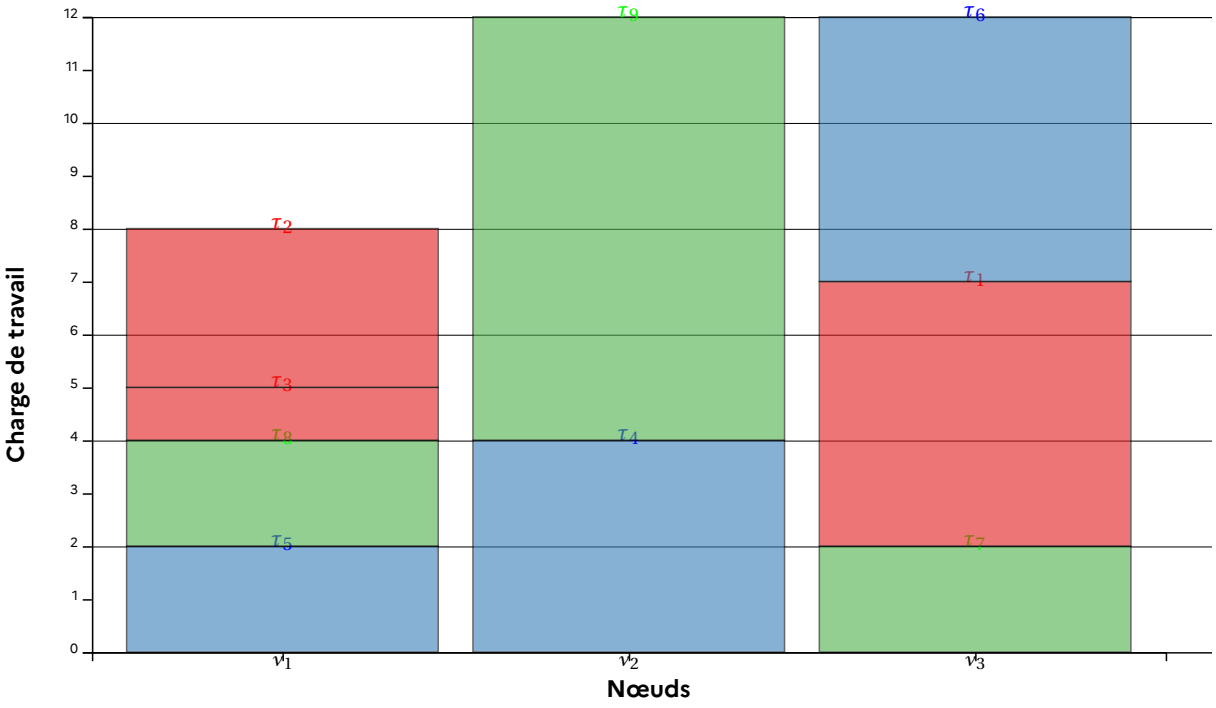
- $Job1 = \{\tau_1, \tau_2\}$ *libéré à la date* $t^0_{J_1} = 0$*;*
- $Job2 = \{\tau_3\}$ *libéré à la date* $t^0_{J_2} = 0$*;*
- $Job3 = \{\tau_4, \tau_5, \tau_6\}$ *libéré à la date* $t^0_{J_3} = 2$*;*

Chaque ressource est présente sur un nœud, et la répartition des ressources aboutit à la fonction de coût des tâches donnée dans le tableau 5.2. Comme dans l'exemple précédent, je suppose le coût d'une tâche proportionnel à la taille des ressources et qu'il est deux fois plus important si la ressource est non locale ($\kappa = 2$).

Je considère l'allocation $\overrightarrow{A_t}$ à l'instant $t = 4$ avec $\vec{B}_{1,t} = (\tau_1, \tau_2, \tau_4, \tau_5)$ et $\vec{B}_{2,t} = (\tau_3, \tau_6)$. L'allocation est représentée sous la forme d'un diagramme de Gantt dans la figure 5.2. Le job J_3 est libéré à la date $t^0_{J_3} = 2$. $\delta(\tau_5, \nu_1)$ est le temps restant avant que τ_5 ne soit traitée par ν_1, et $c(\tau_5, \nu_1)$ est son coût estimé. $C_{J_3}(\overrightarrow{A_t})$ est la durée de réalisation J_3, c'est-à-dire la durée entre sa date de libération $t^0_{J_3}$ et sa date de réalisation $t^E_{J_3}(\overrightarrow{A_{t=4}})$. $w_1(\overrightarrow{A_{t=4}})$ et $w_2(\overrightarrow{A_{t=4}})$ sont respectivement les charges de travail des nœuds ν_1 et ν_2 à l'instant



(a) Répartition des ressources



(b) Allocation de tâches

Figure 5.1 – Répartition des ressources et allocation des tâches aux nœuds pour notre exemple fil rouge où les jobs J_1, J_2 et J_3 sont respectivement représentés en rouge, bleu et vert.

> *$t = 4$. À $t = 4$, le nœud ν_2 commence à réaliser la tâche τ_6, tandis que le nœud ν_1 est toujours occupé avec la tâche τ_2 et doit d'abord la terminer avant de commencer à traiter τ_4 puis τ_5.*

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
$c(\tau, \nu_1)$	1	4	8	4	3	14
$c(\tau, \nu_2)$	2	2	4	2	6	7

Table 5.2 – Le coût des tâches pour chaque nœud

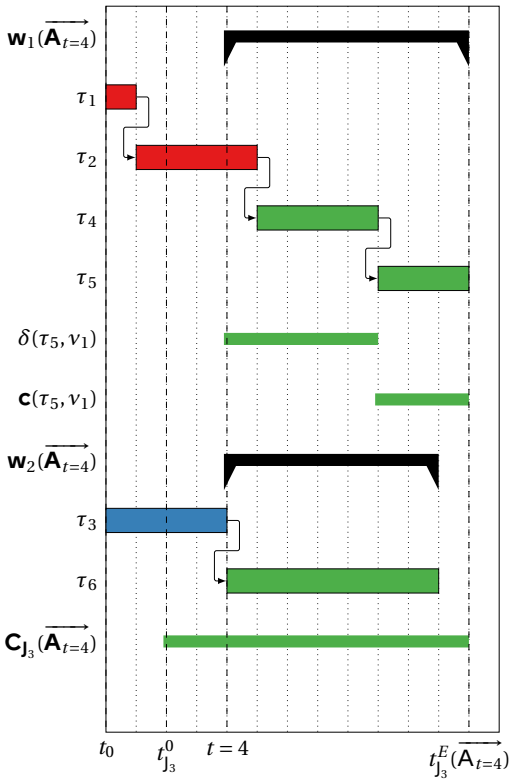


Figure 5.2 – Diagramme de Gantt illustrant l'allocation de l'exemple 5.2 à l'instant $t = 4$.

5.3 Opérations

Je formalise ici les opérations élémentaires et locales de transformation des allocations. Celles-ci permettent de passer d'une allocation à une autre. La consommation consiste à exécuter une tâche. Une réallocation modifie deux lots de tâches. Ces opérations élémentaires sont combinées dans des processus distribués et concurrents : le processus de réallocation et le processus de consommation.

5.3.1 Consommation

La **consommation** d'une tâche par un nœud consiste pour ce dernier à retirer cette tâche de son lot pour l'exécuter. Cette opération locale mène à une nouvelle instance de problème où cette tâche a été retirée. L'accomplissement d'une tâche a donc pour

effet de modifier non seulement l'allocation des tâches mais également le problème sous-jacent. Il est à noter qu'un nœud consomme toujours la première tâche de son lot, selon l'ordre dans lequel ses tâches sont ordonnées.

Définition 5.10 (Consommation de tâche).
Soient STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *un problème d'allocation continue de tâches et* $\overrightarrow{A_t}$ *une allocation. La consommation par un nœud consommateur* ν_i *à un instant* t*, dont le lot n'est pas vide (*$B_{i,t} \neq \emptyset$*), aboutit à l'allocation* $\overrightarrow{A_t}' = \lambda(\nu_i, \vec{B}_{i,t})$ *pour le problème STAP'* $= \langle \mathcal{D}, \mathcal{T}'_t, \mathcal{J}'_t, c \rangle$ *où :*

$$\mathcal{T}'_t = \mathcal{T}_t \setminus \{\min_{\prec_i} B_{i,t}\} \tag{5.11}$$

$$\mathcal{J}'_t = \begin{cases} \mathcal{J}_t \setminus \{job(\min_{\prec_i} B_{i,t})\} & si\ job(\min_{\prec_i} B_{i,t}) = \{\min_{\prec_i} B_{i,t}\} \\ \mathcal{J}_t & sinon \end{cases} \tag{5.12}$$

Dans ce dernier cas :

$$\forall J_j \in \mathcal{J}_t\ \exists J'_j \in \mathcal{J}'_t\ tel\ que\ J'_j = \begin{cases} J_j \setminus \{\min_{\prec_i} B_{i,t}\} & si\ job(\min_{\prec_i} B_{i,t}) = J_j \\ J_j & sinon \end{cases} \tag{5.13}$$

et $\overrightarrow{A_t}' = (\vec{B}'_{1,t}, ..., \vec{B}'_{m,t})$ *avec*

$$\vec{B}'_{j,t} = \begin{cases} \overrightarrow{B_{i,t} \ominus \min_{\prec_i} B_{i,t}} & si\ j = i \\ \vec{B}_{j,t} & sinon \end{cases} \tag{5.14}$$

Lorsqu'une tâche est consommée, elle est retirée du problème résultant non seulement dans l'ensemble des tâches (équation 5.11) mais également du job correspondant (équation 5.13). Ce dernier peut également être retiré si la tâche était la seule, c'est-à-dire la première et la dernière, du job (équation 5.12). L'allocation résultante est également modifiée, la tâche consommée étant retirée du lot où elle se trouvait (équation 5.14). Les tâches sont destinées à être consommées une à une jusqu'à atteindre l'allocation vide.

En théorie, la consommation d'une tâche ne peut pas augmenter le *flowtime* à un instant t. En effet, la consommation d'une tâche fait décroître localement le *flowtime*, à l'instant t,

$$\Sigma_{J \in \mathcal{J}_t} C_J(\lambda(\nu_i, \vec{B}_{i,t})) < \Sigma_{J \in \mathcal{J}_t} C_J(\vec{B}_{i,t}) \tag{5.15}$$

Cette propriété n'est cependant pas toujours vrai au cours du temps en pratique. En effet, comme nous le verrons dans le chapitre 8, les coûts effectifs des tâches peuvent être différents des coûts estimés (cf. définition 5.3). Si une tâche s'avère plus coûteuse que prévu lors de son exécution, le *flowtime* peut augmenter après sa consommation, comme dans l'exemple 5.3.

Exemple 5.3 (Consommation de tâches).
Soit le problème STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *avec :*

- $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{V}, \mathcal{R} \rangle$*, un système distribué avec un unique nœud de calcul* $\mathcal{P} = \{\nu_1\}$ *associé au seul nœud de ressources* $\mathcal{N}_r = \{\nu_1^r\}$*, tel que* $\mathcal{V}(\nu_1, \nu_1^r) = \top$ *et une unique ressource* $\mathcal{R} = \{\rho_1\}$ *sur le nœud de ressource* ν_1^r*;*
- *deux tâches* $\mathcal{T}_t = \{\tau_1, \tau_2\}$*;*

- *un unique job* $\mathcal{J}_t = \{J_1\}$ *libéré à* $t^0_{J_1} = 0$ *composé des deux tâches* $J_1 = \{\tau_1, \tau_2\}$*;*
- *la fonction de coût* c *telle que* $c(\tau_1, \nu_1) = 2$ *et* $c(\tau_2, \nu_1) = 4$.

L'allocation $\overrightarrow{A_0} = (\vec{B}_{1,t})$ *avec* $\vec{B}_{1,t} = (\tau_1, \tau_2)$*. Selon l'équation 5.10, le flowtime est* $C(\overrightarrow{A_0}) = C_{J_1}(\overrightarrow{A_0}) = C_{\tau_2}(\overrightarrow{A_0}) = \delta(\tau_2, \nu_1) + t - t^0_{J_1} + c(\tau_2, \nu_1) = c(\tau_1, \overrightarrow{A_t}) + 0 + 0 + c(\tau_2, \nu_1) = 2 + 4 = 6$.
Si l'exécution de la tâche τ_1 *s'avère plus coûteuse que prévu, par exemple si elle se termine à l'instant* $t_1 = 3$ *au lieu de* 2*, alors le flowtime de* $\overrightarrow{A_{t_1}} = (\vec{B}_{\nu_1,t_1})$ *avec* $B_{\nu_1,t_1} = (\tau_2)$ *est* $C_{mean}(\overrightarrow{A_{t_1}}) = C_{J_1}(\overrightarrow{A_{t_1}}) = t_1 + t^0_{J_1} + c(\tau_2, \nu_1) = 3 + 0 + 4 = 7 > C_{mean}(\overrightarrow{A_0})$.
En résumé, soient t_1 *le temps déjà écoulé au moment où une tâche* τ *commence à être consommée par un nœud* ν *et* t_2 *le temps écoulé à la fin de la consommation. Comme la fonction de coût utilisée est une estimation et non pas le coût « exact » réalisé, la durée effective* $t_2 - t_1$ *de la consommation de la tâche* τ *n'est pas toujours égale au coût estimé renvoyé par la fonction* $c(\tau, \nu)$.

Les opérations de réallocation se déroulent simultanément aux consommations. Les réallocations sont complémentaires à la consommation puisqu'elles permettent de la faciliter en réduisant le coût des jobs. Ces opérations font l'objet de la section suivante.

5.3.2 Réallocation

Une **réallocation bilatérale** est une opération élémentaire et locale qui modifie l'allocation courante. Elle consiste en un échange de tâches entre deux agents.

Définition 5.11 (Réallocation bilatérale).
Soit $\overrightarrow{A_t} = (\vec{B}_{1,t}, \ldots, \vec{B}_{m,t})$ *une allocation pour le problème STAP* $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$*. La réallocation bilatérale de la liste non vide de tâches* $\mathrm{T}_1 \subseteq B_{i,t}$ *allouées au proposant* ν_i *en échange de la liste de tâches* $\mathrm{T}_2 \subseteq B_{j,t}$ *allouées au répondant* ν_j *dans* $\overrightarrow{A_t}$ *aboutit à l'allocation* $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})$ *avec les* m *lots* $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \vec{B}_{k,t})$ *définis tels que :*

$$\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \vec{B}_{k,t}) = \begin{cases} \overrightarrow{B_{i,t} \ominus \mathrm{T}_1 \oplus \mathrm{T}_2} & \text{si } k = i, \\ \overrightarrow{B_{j,t} \ominus \mathrm{T}_2 \oplus \mathrm{T}_1} & \text{si } k = j, \\ \vec{B}_{k,t} & \text{sinon} \end{cases} \tag{5.16}$$

Il est à noter que la mise en œuvre des opérateurs d'ajout $\oplus$ et de suppression $\ominus$ d'une liste de tâches dans un lot dépend de la stratégie de consommation qui détermine l'ordonnancement du lot. Cette stratégie de consommation est définie dans la section 6.4.

Pour une réallocation bilatérale $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})$, je distingue deux cas :

- un **échange** où les deux listes de tâches sont non vides ($\mathrm{T}_1 \neq \emptyset \wedge \mathrm{T}_2 \neq \emptyset$), noté $\sigma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})$. Si $|\mathrm{T}_1| = |\mathrm{T}_2| = 1$, on parle d'échange unaire ;
- une **délégation** où un agent donne une partie de ses tâches à l'un de ses pairs sans contre-partie ($\mathrm{T}_2 = \emptyset$), notée $\delta(\mathrm{T}_1, \nu_i, \nu_j, \overrightarrow{A_t})$. Si $|\mathrm{T}_1| = 1$, on parle de délégation unaire. Dans le cas contraire, on parle délégation n-aire.

Nous verrons par la suite que la réallocation bilatérale de listes de tâches plutôt que d'ensembles permet de spécifier l'ordre selon lequel les tâches doivent être évaluées pour valider l'intérêt de tout ou partie de la transaction.

5.4 Rationalité sociale et stabilité

L'intérêt de réaliser des réallocations est d'améliorer une allocation existante. J'introduis donc la notion de réallocation bilatérale socialement rationnelle qui désigne une réallocation réduisant le *flowtime*, c'est-à-dire le délai de réalisation des jobs pour l'ensemble des nœuds.

Définition 5.12 (Réallocation bilatérale socialement rationnelle).
Soient STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *un problème d'allocation,* $\overrightarrow{A_t}$ *une allocation à un instant* t*. La réallocation bilatérale* $\gamma(\mathrm{T}_1, \mathrm{T}_2, v_i, v_j, \overrightarrow{A_t})$ *est socialement rationnelle vis-à-vis du flowtime si et seulement si cette réallocation fait décroître le flowtime,*

$$C(\gamma(\mathrm{T}_1, \mathrm{T}_2, v_i, v_j, \overrightarrow{A_t})) < C(\overrightarrow{A_t}) \tag{5.17}$$

Une allocation est dite **stable** s'il n'existe aucune réallocation bilatérale socialement rationnelle.

Afin de garantir la terminaison du processus, j'ai choisi un unique critère de rationalité globale plutôt que la combinaison de deux critères comme dans Beauprez et al. (2021).

La réduction du *flowtime* comme critère de rationalité permet de garantir la terminaison du processus.

Propriété 5.1 (Terminaison).
Soient STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *un problème d'allocation continue et* $\overrightarrow{A_t}$ *une allocation qui n'est pas stable vis-à-vis du flowtime. Toute séquence de réallocations bilatérales socialement rationnelles issue de* $\overrightarrow{A_t}$ *est finie car elle atteint une allocation stable.*

On peut noter que la preuve de terminaison du processus de réallocation est similaire à celle utilisée par Baert (2019) et découle du théorème de Endriss et al. (2006).

Preuve 5.1 (Terminaison).
Comme le nombre d'allocations est fini et $\Sigma_{J \in \mathcal{J}_t} C_J(\overrightarrow{A_t})$ *décroît strictement à chaque étape, il ne peut y avoir qu'un nombre fini de réallocations socialement rationnelles aboutissant à une allocation stable.*

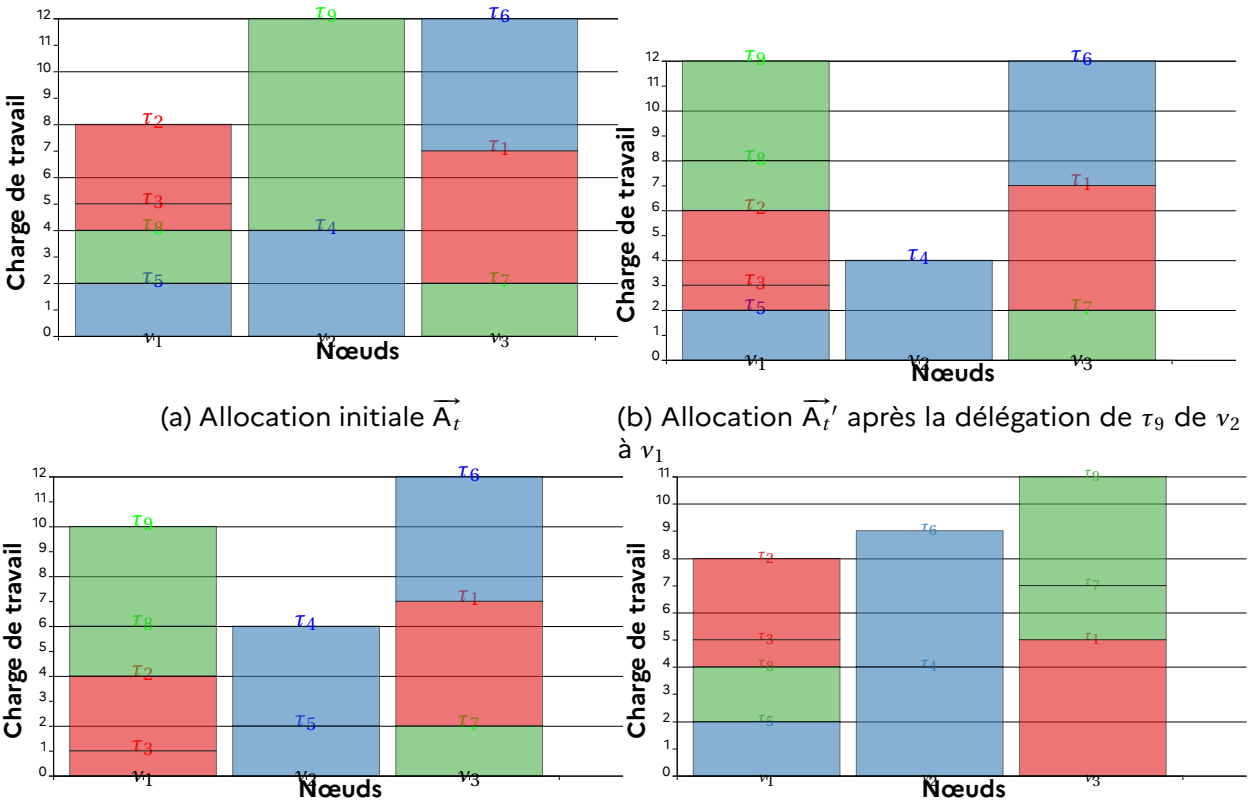
Les délégations comme les échanges peuvent être socialement rationnels. Bien que plus rares, ces derniers peuvent s'avérer plus efficaces pour améliorer le *flowtime* moyen, comme le montre l'exemple 5.4.

Exemple 5.4 (Réallocation socialement rationnelle).
Considérons le problème STAP de l'exemple 5.1, en particulier l'allocation $\overrightarrow{A_t}$ *représentée dans la figure 5.1b.*

La figure 5.3b représente l'allocation $\overrightarrow{A_t}'$ *obtenue après la délégation de la tâche* τ_9 *par le nœud* v_2 *au nœud* v_1*. La nouvelle allocation obtenue est notée* $\overrightarrow{A_t}' = \delta([\tau_9], v_2, v_1, \overrightarrow{A_t})$*. Cette délégation est socialement rationnelle car elle fait décroître le flowtime de* 32 *à* 31*.*

Toutefois, l'échange de $\tau_9 \in B_{2,t}$ contre $\tau_5 \in B_{1,t}$ entre les nœuds ν_2 et ν_1, qui aboutit à l'allocation $\overrightarrow{A_t}'' = \sigma([\tau_9],[\tau_5],\nu_2,\nu_1,\overrightarrow{A_t})$ représentée sur la figure 5.3c, est meilleur car il fait décroître le flowtime de 32 à 29.

Il existe également des cas où des échanges socialement rationnels existent à partir de délégations non socialement rationnelles. Par exemple, l'échange de la tâche τ_9 avec la tâche τ_6 à partir de l'allocation initiale aboutit à l'allocation $\overrightarrow{A_t}''' = \sigma([\tau_9],[\tau_6],\nu_2,\nu_3,\overrightarrow{A_t})$ représentée sur la figure 5.3d. Cet échange est socialement rationnel car il fait décroître le flowtime de 32 à 28, bien que ni la délégation $\delta([\tau_9],\nu_2,\nu_3,\overrightarrow{A_t})$ ni la délégation $\delta([\tau_6],\nu_3,\nu_2,\overrightarrow{A_t})$ ne sont socialement rationnelle.



(a) Allocation initiale $\overrightarrow{A_t}$

(b) Allocation $\overrightarrow{A_t}'$ après la délégation de τ_9 de ν_2 à ν_1

(c) Allocation $\overrightarrow{A_t}''$ après l'échange de τ_9 contre τ_5 entre ν_2 et ν_1

(d) Allocation $\overrightarrow{A_t}'''$ après l'échange de τ_9 contre τ_6 entre ν_2 et ν_3

Figure 5.3 – Allocations de l'exemple 5.4 résultant de réallocations bilatérales

5.5 Synthèse

J'ai présenté dans ce chapitre la formalisation du problème d'allocation continue de tâches situées, ou STAP pour *Situated Task Allocation Problem*. Contrairement au chapitre 2, je considère un problème dynamique où les tâches sont peu à peu consommées et les jobs sont soumis au fil de l'eau.

En résumé, un problème STAP est constitué :

(a) d'un système distribué avec m nœuds de calcul et autant de nœuds de ressources ;

(b) d'un ensemble de jobs concurrents, chacun constitué de plusieurs tâches dont les coûts pour les nœuds dépendent de la localité des ressources.

L'objectif est de minimiser la durée moyenne de réalisation de ces jobs. Les tâches sont indépendantes, non divisibles et non préemptables, et les ressources sont transférables, non consommables et sont accessibles à tous les nœuds.

La solution à un tel problème consiste en une allocation des tâches en des lots ordonnés parmi les différents nœuds de calcul qui sont chargés de consommer, c'est-à-dire d'exécuter, les tâches. Dans l'optique de réaliser l'ensemble des jobs le plus tôt possible, la qualité d'une allocation dépend de la durée moyenne de réalisation des jobs, aussi appelée *flowtime* moyen. Afin d'améliorer une allocation, les agents contrôlant les nœuds de calcul sont amenés à effectuer des opérations de réallocation, lors desquelles ils peuvent échanger une ou plusieurs tâches. J'ai montré que les réallocations dites « socialement rationnelles », garantissent la terminaison du processus de réallocation vers une allocation stable.

Consommations et réallocations sont complémentaires et se produisent simultanément. Le but de la réallocation est de faciliter la consommation en réduisant le délai moyen de réalisation des jobs.

Comme nous allons le voir dans le chapitre suivant, les agents négocient entre eux à l'aide d'un protocole d'offres alternées pour réallouer leurs tâches. Ils se basent sur leurs croyances à propos des lots de tâches de leurs pairs.

Chapitre 6

Modèle multi-agents

Sommaire

6.1 Introduction 59
6.2 Modèle des pairs 60
6.3 Règle d'acceptabilité 62
6.4 Stratégie de consommation 63
6.4.1 Ordonnancement orienté tâche 63
6.4.2 Ordonnancement orienté job 64
6.5 Protocole d'offres alternées 68
6.6 Stratégie de négociation 70
6.6.1 Stratégie d'offre 70
6.6.2 Stratégie d'acceptation 75
6.6.3 Stratégie de contre-offre 75
6.7 Synthèse 77

6.1 Introduction

Dans ce chapitre, je présente mon modèle multi-agents où des agents, chacun supervisant un nœud de calcul, collaborent pour aboutir à une meilleure répartition des tâches. Je propose ici une stratégie d'agent négociant pour détecter des opportunités d'échange de tâches.

Pour réallouer les tâches, les agents négocient en utilisant le protocole d'offres alternées (Rubinstein, 1982). Pour pouvoir négocier, un agent a besoin de plusieurs compétences présentées dans ce chapitre.

Un agent dispose d'un modèle des pairs, présenté dans la section 6.2, qui définit ses croyances sur les lots de tâches des autres agents, et sur lequel il s'appuie pour négocier, voire pour ordonnancer son propre lot. Il dispose également d'une règle d'acceptabilité, abordée dans la section 6.3, qui lui permet, à l'aide de ses croyances, de décider si une offre est acceptable ou non. La section 6.5 décrit le protocole d'offres alternées et les interactions entre agents dans le cas des délégations et dans le cas des échanges. La stratégie de négociation, détaillée dans la section 6.6 regroupe plusieurs

mécanismes de décision : les stratégies de délégations, unaire ou n-aire, permettent à l'agent de sélectionner une proposition de délégation ; la stratégie d'acceptabilité lui permet d'accepter une offre, partiellement ou dans son intégralité, ou de la refuser ; et la stratégie de contre-offre lui permet de proposer un échange lorsqu'une proposition de délégation est reçue.

Les agents peuvent réaliser des délégations et des échanges. Comme la recherche d'un échange pertinent est plus coûteux que la recherche d'une délégation, il n'est pas pertinent de chercher à réaliser systématiquement des échanges. Pour cette raison, j'adopte un processus de négociation en deux phases. Une première phase est dédiée aux délégations, tandis que comme nous le verrons dans le chapitre 8 dédié aux expérimentations la deuxième phase utilise les échanges pour s'extraire des minima locaux lorsqu'il n'y a plus de délégation pertinente à réaliser.

6.2 Modèle des pairs

Afin de non seulement déterminer sa stratégie d'offre mais également d'évaluer localement la rationalité sociale des délégations, un agent doit disposer au préalable d'un modèle des autres agents dans les éventuelles négociations. Disposer d'un modèle de ses pairs est également nécessaire à l'agent pour ordonnancer son lot de tâches selon certaines stratégies de consommation, qui seront étudiées dans la section 6.4.

Je considère ici la modélisation par un sujet $v_i \in \mathcal{P}$ d'une cible $v_j \in \mathcal{P}$. Celle-ci s'appuie sur :

- la base de croyances $\mathcal{B}_i$ du sujet, éventuellement partielle ou obsolète, construite à partir des informations transmises par la cible (le coût du job J pour la cible v_j $c(J, v_j)$, $\forall J \in \mathcal{J}_t$) ;
- la stratégie de consommation de la cible qui, pour les stratégies d'ordonnancement orientées job sur lesquelles je me focalise, se résume à la relation supposée par le sujet d'ordonnancement des jobs adoptée par la cible, notée $\blacktriangleleft_j^i$

De plus, l'agent connaît la fonction de coût ainsi que la localité des ressources pour toutes les tâches. Il peut donc déduire l'estimation du coût d'une tâche pour n'importe quel nœud.

Dans la suite, les croyances d'un sujet v_i sont indiquées par un exposant i dans les formulations. Elles sont construites à partir des informations contenues dans les messages échangés qui seront détaillés dans le chapitre 7.

Au-delà de ce que le sujet croit connaître du coût de chaque job pour la cible ($c^i(J, v_j)$, $\forall J \in \mathcal{J}_t$) et de ce que le sujet déduit de la charge de travail de la cible ($w_j^i(\overrightarrow{A_t})$), le sujet peut alors inférer :

- $C_J^{\,i}(\vec{B}_{j,t})$, une croyance sur la durée de réalisation de chaque job J pour la cible v_j dans l'allocation courante ;
- $C_J^{\,i}(\overrightarrow{B_{j,t} \oplus \tau})$, une croyance sur la durée de réalisation de chaque job pour la cible v_j pour laquelle une tâche τ est ajoutée à son lot ;
- $C_J^{\,i}(\overrightarrow{B_{j,t} \ominus \tau})$, une croyance sur la durée de réalisation de chaque job pour la cible v_j pour laquelle une tâche τ est supprimée de son lot.

Finalement, le sujet peut déduire des modèles de ses pairs une hypothèse sur la durée de réalisation de chaque job dans l'allocation :

$$C_J{}^i(\overrightarrow{A_t}) = \max_{\nu_k \in \mathcal{P}} C_J{}^i(\vec{B}_{k,t}) \text{ où } C_J{}^i(\vec{B}_{i,t}) = C_J(\vec{B}_{i,t}) \tag{6.1}$$

Ainsi, le sujet peut déduire pour chaque job quel nœud termine ses tâches le plus tardivement. Ce nœud est alors le **facteur limitant** du job concerné. Formellement,

Définition 6.1 (Facteur limitant).
Soit STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *un problème et* $\overrightarrow{A_t}$ *une allocation à l'instant* t*. Selon le sujet* $\nu_i \in \mathcal{P}$*, une cible* $\nu_j \in \mathcal{P}$ *est appelée facteur limitant pour un job J dans l'allocation* $\overrightarrow{A_t}$ *(noté* $\nu_j = nodeMax^i(\overrightarrow{A_t}, J)$*) si et seulement si la durée de réalisation du job pour la cible est la durée de réalisation de ce job dans l'allocation.*

$$\nu_j = nodeMax^i(\overrightarrow{A_t}, J) \Leftrightarrow C_J{}^i(\overrightarrow{A_t}) = C_J{}^i(\vec{B}_{j,t}) \tag{6.2}$$

L'agent ν_i informe ses pairs du coût que représente chaque job pour lui ($c(J, \nu_i)\ \forall J \in \mathcal{J}_t$) avant le processus de négociation et après chaque réallocation bilatérale qu'il propose ou accepte. Comme le nombre de jobs est négligeable par rapport au nombre de tâches, comme mentionné dans la section 5.2, la taille des messages est insignifiante par rapport à la description d'un lot.

J'illustre ci-dessous ces notions à l'aide de l'exemple fil rouge.

Exemple 6.1 (Modèle des pairs).
Considérons le problème STAP et l'allocation $\overrightarrow{A_t}$ *de l'exemple 5.1. Le modèle des pairs de l'agent contrôlant le nœud de calcul* ν_1 *contient ses croyances concernant les lots de ses pairs* ν_2 *et* ν_3*. La figure 6.1 montre l'allocation* $\overrightarrow{A_t}$ *telle que perçue par l'agent* ν_1*. Pour rappel, les jobs* J_1*,* J_2 *et* J_3 *sont représentés respectivement en rouge, bleu et vert sur la figure. S'il connaît en détail le contenu de son propre lot, les croyances de* ν_1 *concernant les lots de ses pairs portent uniquemement sur le coût de leurs jobs.*

- $c^1(J_1, \nu_2) = 0$ *le coût supposé du job* J_1 *pour le nœud* ν_2*;*
- $c^1(J_2, \nu_2) = 4$ *le coût supposé du job* J_2 *pour le nœud* ν_2*;*
- $c^1(J_3, \nu_2) = 8$ *le coût supposé du job* J_3 *pour le nœud* ν_2*;*
- $c^1(J_1, \nu_3) = 5$ *le coût supposé du job* J_1 *pour le nœud* ν_3*;*
- $c^1(J_2, \nu_3) = 5$ *le coût supposé du job* J_2 *pour le nœud* ν_3*;*
- $c^1(J_3, \nu_3) = 2$ *le coût supposé du job* J_3 *pour le nœud* ν_3*.*

Comme l'agent connaît la stratégie de consommation de ses pairs, c'est-à-dire l'ordre des jobs dans leur lot, il peut en déduire une hypothèse sur la durée de réalisation de chacun des jobs pour ses pairs :

- $C_{J_1}{}^1(\vec{B}_{2,t}) = 0$ *la durée de réalisation supposée du job* J_1 *pour le nœud* ν_2*;*
- $C_{J_2}{}^1(\vec{B}_{2,t}) = 4$ *la durée de réalisation supposée du job* J_2 *pour le nœud* ν_2*;*
- $C_{J_3}{}^1(\vec{B}_{2,t}) = 12$ *la durée de réalisation supposée du job* J_3 *pour le nœud* ν_2*;*

- $C_{J_1}^1(\vec{B}_{3,t}) = 7$ *la durée de réalisation supposée du job* J_1 *pour le nœud* v_3 *;*
- $C_{J_2}^1(\vec{B}_{3,t}) = 12$ *la durée de réalisation supposée du job* J_2 *pour le nœud* v_3 *;*
- $C_{J_3}^1(\vec{B}_{3,t}) = 2$ *la durée de réalisation supposée du job* J_3 *pour le nœud* v_3*.*

Le sujet peut alors identifier le facteur limitant de chacun des jobs :

- $v_3 = nodeMax^1(\overrightarrow{A_t}, J_1)$ *est le facteur limitant du job* J_1 *représenté en rouge;*
- $v_3 = nodeMax^1(\overrightarrow{A_t}, J_2)$ *est le facteur limitant du job* J_2 *représenté en bleu;*
- $v_2 = nodeMax^1(\overrightarrow{A_t}, J_3)$ *est le facteur limitant du job* J_3 *représenté en vert.*

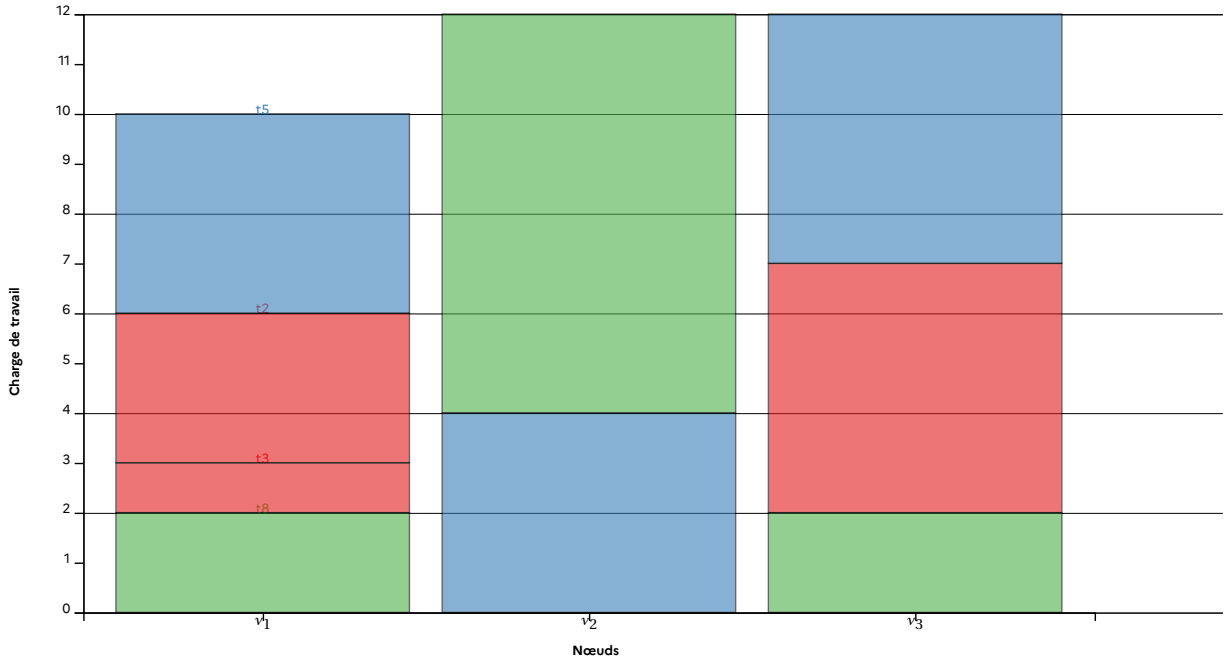


Figure 6.1 – Allocation $\overrightarrow{A_t}$ de l'exemple 6.1 vue par l'agent v_1 qui ignore la description du lot de ses pairs

6.3 Règle d'acceptabilité

La **règle d'acceptabilité** est une règle de décision locale prise par un agent impliqué dans une réallocation bilatérale, à partir de ses connaissances et du modèle de ses pairs. Elle détermine si la réallocation qui lui est proposée est socialement rationnelle, au sens de la définition 5.12. Ainsi, un agent n'accepte une proposition que si la réallocation permet de faire décroître le *flowtime* supposé de l'allocation courante.

Définition 6.2 (Acceptabilité).
Soient STAP $= \langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *un problème d'allocation continue et* $\overrightarrow{A_t}$ *une allocation à l'instant* t*. La réallocation bilatérale* $\gamma(\mathrm{T}_1, \mathrm{T}_2, v_i, v_j, \overrightarrow{A_t})$ *est acceptable par l'agent* $v_k \in$

$\{v_i, v_j\}$ *vis-à-vis du flowtime si et seulement si l'agent croit que le flowtime décroît,*

$$\Sigma_{J\in\mathcal{J}_t} \max_{\forall v_o\in\mathcal{P}\setminus\{v_i,v_j\}} (C_J{}^k(\overrightarrow{B_{i,t}\ominus \mathrm{T}_1\oplus \mathrm{T}_2}), C_J{}^k(\overrightarrow{B_{j,t}\ominus \mathrm{T}_2\oplus \mathrm{T}_1}), C_J{}^k(B_{o,t})) < C^{\,k}(\overrightarrow{A_t}) \tag{6.3}$$

Il est important de noter que cette règle de décision locale ne nécessite pas de connaître la description des lots des autres nœuds. Le coût des jobs pour les pairs et leur ordonnancement dans ces lots sont suffisants et nécessaires pour déterminer l'acceptabilité (ou non) d'une réallocation bilatérale.

6.4 Stratégie de consommation

Nous avons vu précédemment que chaque agent consomme la première tâche de son lot qui est ordonné. Je propose ici plusieurs stratégies de consommation, c'est-à-dire des heuristiques de tri pour le lot de tâches. Par la suite, une stratégie de consommation est donc définie par une relation d'ordre stricte et totale sur les tâches.

Comme chaque tâche (respectivement chaque job, chaque nœud) possède son propre identifiant qui est un nombre entier, je considère $<$ l'ordre sur les tâches (respectivement les jobs, les nœuds) induit par l'ordre naturel sur leurs identifiants. Pour aller au-delà de ces ordres arbitraires, je présente dans la section 6.4.1 des heuristiques qui prennent en considération les coûts ou durées de réalisation des tâches, et dans la section 6.4.2 des heuristiques qui prennent en considération ceux des jobs.

De plus, je fais la distinction entre les stratégies de consommation dîtes **locales**, où le nœud ordonne ses tâches en considérant uniquement les coûts des tâches ou des jobs pour son propre lot, et les stratégies dîtes **globales**, où le nœud ordonne ses tâches en considérant les coûts pour l'ensemble des nœuds.

6.4.1 Ordonnancement orienté tâche

La première stratégie de consommation, appelée *Locally Cheapest Task First*, consiste à ordonner les tâches par coût croissant dans le lot de tâches, sans tenir compte du job auquel chacune appartient.

Définition 6.3 (LCTF).
La stratégie "Locally Cheapest Task First" LCTF, adoptée par l'agent v_i consiste à trier son lot de tâches selon la relation d'ordre stricte et totale $\prec_i$ sur $\mathcal{T}_t$ définie par :

$$\begin{aligned}&\forall \tau_j, \tau_k \in B_{i,t}, \tau_j \prec_i \tau_k \Leftrightarrow\\ &c(\tau_j, v_i) < c(\tau_k, v_i) \vee (c(\tau_j, v_i) = c(\tau_k, v_i) \wedge \tau_j < \tau_k)\end{aligned} \tag{6.4}$$

Selon cette stratégie, les tâches les moins coûteuses sont exécutées avant les plus coûteuses. Comme l'ordre naturel sur les identifiants des tâches départage les ex æquo, la relation $\prec_i$ est stricte.

La stratégie LCTF permet à un nœud de minimiser localement la durée de réalisation des tâches pour son lot de tâches.

Théorème 6.1 (LCTF).
Soit $\vec{B}_{i,t}$ ordonné selon la stratégie "Locally Cheapest Task First" adoptée par l'agent ν_i.

$$\forall \sigma \in S(\vec{B}_{i,t}),\ \Sigma_{\tau \in B_{i,t}} C_\tau(\vec{B}_{i,t}) \leq \Sigma_{\tau \in B_{i,t}} C_\tau(\sigma(\vec{B}_{i,t})) \tag{6.5}$$

où $S(\vec{B}_{i,t})$ dénote l'ensemble des permutations du lot $\vec{B}_{i,t}$.

Preuve 6.1 (LCTF).
*Cette stratégie de consommation des tâches correspond à la règle appelée "Shortest Processing Time first" (SPT) en théorie de l'ordonnancement et évoquée dans le chapitre 3. Le théorème 3.1.1 du chapitre 3 dans (*Pinedo*, 2008) permet de déduire que la stratégie LCTF est optimale quand on souhaite minimiser la somme des durées de réalisation des tâches dans $B_{i,t}$.*

Il est important de noter que cette stratégie ne s'appuie que sur la connaissance par l'agent du problème et de son propre lot de tâches.

6.4.2 Ordonnancement orienté job

Les stratégies d'ordonnancement des jobs consistent à agencer le lot d'abord par job puis par tâche au sein d'un même job.

Définition 6.4 (Stratégie d'ordonnancement des jobs).
Soient $\lhd_i$ et $\blacktriangleleft_i$ deux relations d'ordre strictes et totales sur $\mathcal{T}_t$ et sur $\mathcal{J}_t$ respectivement. Une stratégie d'ordonnancement des jobs basée sur $(\lhd_i, \blacktriangleleft_i)$ permet d'ordonner le lot $B_{i,t}$ selon la relation d'ordre stricte et totale $\prec_i$ définie par :

$$\begin{aligned} &\forall \tau_j, \tau_k \in B_{i,t}\ \tau_j \prec_i \tau_k \Leftrightarrow \\ &job(\tau_j) \blacktriangleleft_i job(\tau_k) \vee (job(\tau_j) = job(\tau_k) \wedge \tau_j \lhd_i \tau_k) \end{aligned} \tag{6.6}$$

En adoptant ces stratégies, les tâches d'un même job sont consécutives dans le lot de tâches. Au-delà de l'ordre sur les tâches au sein d'un même job, qui peut être l'ordre naturel sur les identifiants ou similaire à la stratégie LCTF telle que décrite dans la définition 6.3, les stratégies d'ordonnancement des jobs se distinguent les unes des autres principalement par la relation $\blacktriangleleft_i$ utilisée pour agencer les jobs. $J_1 \blacktriangleleft_i J_2$ signifie que les tâches dans J_1 sont plus prioritaires que celles dans J_2.

Locally Cheapest Job First.

Une première famille de stratégies d'ordonnancement de jobs consiste à les ordonner dans le lot par coût croissant.

Définition 6.5 (LCJF).
Soit $\lhd_i$ une relation d'ordre stricte et totale sur $\mathcal{T}$. Une stratégie d'ordonnancement des jobs basée sur $(\lhd_i, \blacktriangleleft_i)$ est dite "Locally Cheapest Job First" (LCJF) ssi la relation $\blacktriangleleft_i$

sur les jobs vérifie :

$$\begin{aligned}&\forall J_j, J_k \in \mathcal{J}_t,\ J_j \blacktriangleleft_i J_k \Leftrightarrow \\ &c(J_j, v_i) < c(J_k, v_i) \vee (c(J_j, v_i) = c(J_k, v_i) \wedge J_j < J_k)\end{aligned} \tag{6.7}$$

D'après une telle stratégie, les tâches des jobs les moins coûteux sont positionnées avant celles des jobs les plus coûteux. L'ordre sur les jobs $\blacktriangleleft_i$ s'appuie uniquement sur la connaissance par l'agent du problème et de son propre lot de tâches. De plus, cette relation est totale et stricte car la relation $<$ sur les jobs départage les ex æquo grâce à leurs identifiants. Comme la relation $\lhd_i$ est supposée stricte et totale, la relation $\prec_i$ sur les tâches est complètement définie, totale et stricte.

Une stratégie LCJF permet à un nœud v_i de minimiser localement la durée de réalisation des jobs pour son lot de tâches. Il n'existe pas de permutation des jobs qui lui sont affectés, appliquée au lot de tâches, qui permette de faire décroître strictement la somme de ses durées de réalisation des jobs.

Théorème 6.2 (LCJF).
Soit une stratégie d'ordonnancement des jobs LCJF basée sur $(\lhd_i, \blacktriangleleft_i)$.

$$\forall \sigma \in S(jobs(B_{i,t})),\ \Sigma_{J \in jobs(B_{i,t})} C_J(\vec{B}_{i,t}) \leq \Sigma_{J \in jobs(B_{i,t})} C_J(\sigma(\vec{B}_{i,t})) \tag{6.8}$$

où $S(jobs(B_{i,t}))$ *dénote l'ensemble des permutations de l'ensemble des jobs qui sont assignés au nœud* v_i.

Preuve 6.2 (LCJF).
Comme toutes les tâches d'un même job sont consécutives dans le lot, la preuve de ce théorème est identique à celle du théorème 6.1 en faisant abstraction des tâches pour ne considérer que les jobs.

Globally Cheapest Job First.

Une seconde famille de stratégies consiste à ordonner les jobs par coût maximal croissant sur l'ensemble des nœuds.

Définition 6.6 (GCJF).
Soit $\lhd_i$ *une relation d'ordre stricte et totale sur* $\mathcal{T}$. *Une stratégie d'ordonnancement des jobs basée* $(\lhd_i, \blacktriangleleft_i)$ *est dite "Globally Cheapest Job First" (GCJF) si et seulement si la relation* $\blacktriangleleft_i$ *sur les jobs vérifie :*

$$\begin{aligned}&\forall J_j, J_k \in \mathcal{J}_t,\ J_j \blacktriangleleft_i J_k \Leftrightarrow \\ &\Big[\max_{v_\ell \in \mathcal{P}} c^i(J_j, v_\ell) < \max_{v_\ell \in \mathcal{P}} c^i(J_k, v_\ell) \\ &\vee (\max_{v_\ell \in \mathcal{P}} c^i(J_j, v_\ell) = \max_{v_\ell \in \mathcal{P}} c^i(J_k, v_\ell) \wedge J_j < J_k)\Big]\end{aligned} \tag{6.9}$$

D'après une telle stratégie, les tâches des jobs dont les coûts maximaux sont moindres sont positionnées avant celles des jobs dont les coûts maximaux sont plus importants.

Pour les mêmes raisons que celles évoquées précédemment, la relation sur les jobs $\blacktriangleleft_i$ définie ici est totale et stricte. De même, la relation $\prec_i$ sur les tâches est complètement définie, totale et stricte. Contrairement aux stratégies précédentes, une stratégie GCJF s'appuie sur les croyances de l'agent à propos de ses pairs. De plus, on peut noter que les ensembles ordonnés $(\mathcal{J}_t, \blacktriangleleft_i)$ sont, à croyance égale, identiques pour tous les agents ν_i.

Most Expensive Job of Most Loaded Node Last.

Une dernière famille de stratégies intitulée "*Most Expensive Job of Most Loaded Node Last*" consiste à exécuter en dernier le job le plus coûteux pour le nœud le plus chargé, puis exécuter en avant-dernier le job restant le plus coûteux pour le nœud le plus chargé sans prendre en compte le job précédent, etc. À cette intention, l'algorithme 6.1 permet à un agent, grâce à ses connaissances et ses croyances, d'ordonner les jobs du dernier à exécuter au premier. On peut noter qu'à la ligne 4 (resp. à la ligne 5), l'ordre $<$ sur les nœuds (resp. sur les jobs) départage les ex aequo. En conséquence, la relation $\blacktriangleleft_i$ est stricte et totale sur $\mathcal{J}_t$. Comme précédemment, ces ordres sont, à croyance égale, identiques pour tous les agents.

Définition 6.7 (MEJMLNL).
Soit $\lhd_i$ une relation d'ordre stricte et totale sur $\mathcal{T}_t$. Une stratégie d'ordonnancement des jobs basée sur $(\lhd_i, \blacktriangleleft_i)$ est dite "Most Expensive Job of Most Loaded Node Last" (MEJMLNL) ssi la relation $\blacktriangleleft_i$ est établie par l'algorithme 6.1.

Comme pour les autres stratégies, la relation $\prec_i$ sur les tâches est complètement définie, totale et stricte.

Algorithme 6.1 : Ordonnancement des jobs du dernier à exécuter au premier pour la stratégie MEJMLNL adoptée par ν_i.

Entrées : STAP un problème d'allocation,
$\mathcal{B}_i$ la base de croyance du nœud
Sorties : $(\mathcal{J}_t, \blacktriangleleft_i)$ l'ensemble ordonné des jobs

1 Jobs = $\mathcal{J}_t$;
2 LateJobs = $\emptyset$;
3 **tant que** *Jobs* $\neq \emptyset$ **faire**
4 $\quad \nu_{max} = \text{argmax}_{\nu_k \in \mathcal{N}} \Sigma_{\mathsf{J} \in \mathsf{Jobs}} \mathsf{c}^i(\mathsf{J}, \nu_k)$; `// nœud le plus chargé en ne considérant que les jobs dans Jobs`
5 $\quad \mathsf{J}_{max} = \text{argmax}_{\mathsf{J} \in \mathsf{Jobs}} \mathsf{c}^i(\mathsf{J}, \nu_{max})$;
$\quad$ `// job le plus coûteux pour le nœud le plus chargé en ne considérant que les jobs dans Jobs`
6 $\quad \forall \mathsf{J} \in$ LateJobs, $\mathsf{J}_{max} \blacktriangleleft_i \mathsf{J}$; `// L'ordre est mis à jour`
7 $\quad$ LateJobs.append(J_{max}) ;
8 $\quad$ Jobs.remove(J_{max}) ;
9 **fin**
10 **retourner** *LateJobs*

Pour mémoire, le *flowtime* est la mesure de performance que je considère dans ce manuscrit. Contrairement au *makespan* qui ne dépend pas de l'ordre d'exécution des

tâches, le *flowtime* résulte de la durée de réalisation des jobs plutôt que celles des tâches. C'est la raison pour laquelle je me focalise par la suite sur les stratégies d'ordonnancement orientées job plutôt que sur les stratégies d'ordonnancement orientées tâche.

L'exemple suivant illustre les différentes stratégies d'ordonnancement des jobs.

Exemple 6.2 (Ordonnancement orienté job).
Soit le problème STAP défini dans l'exemple 5.1. Pour rappel, on considère un ensemble de 3 nœuds de calcul $\mathcal{P} = \{\nu_1, \nu_2, \nu_3\}$ *et un ensemble de 3 nœuds de ressources* $\mathcal{N}_r = \{\nu_1^r, \nu_2^r, \nu_3^r\}$ *liés par la relation de voisinage telle que* $\mathcal{V}(\nu_i, \nu_i^r) = \top$. *L'ensemble des jobs* $\mathcal{J}_t = \{J_1, J_2, J_3\}$ *tous libérés à la date* $t_0 = 0$ *est tel que* $J_1 = \{\tau_1, \tau_2, \tau_3\}$, $J_2 = \{\tau_4, \tau_5, \tau_6\}$ *et* $J_3 = \{\tau_7, \tau_8, \tau_9\}$. *Les coûts des tâches pour les nœuds, qui dépendent de la localité des ressources, est rappelé dans le tableau 6.1. La répartition des tâches est telle que :*

- τ_2, τ_3, τ_5 *et* τ_8 *sont affectées au nœud* ν_1 ;
- τ_4 *et* τ_9 *sont affectées au nœud* ν_2 ;
- τ_1, τ_6 *et* τ_7 *sont affectées au nœud* ν_3.

En adoptant la stratégie "Locally Cheapest Job First" (LCJF) illustrée sur la figure 6.2a, les agents ordonnent dans leur lot les jobs par coût croissant. Ainsi, les lots de tâches sont :

- $\vec{B}_{1,t} = (\tau_5, \tau_8, \tau_3, \tau_2)$ *car*

$$\begin{aligned} (c(J_2, \nu_1) = c(\tau_5, \nu_1)) = \\ (c(J_3, \nu_1) = c(\tau_8, \nu_1)) < \\ (c(J_1, \nu_1) = c(\tau_2, \nu_1) + c(\tau_3, \nu_1)) \end{aligned} \tag{6.10}$$

- $\vec{B}_{2,t} = (\tau_4, \tau_9)$ *car*

$$\begin{aligned} (c(J_1, \nu_2) = 0) < \\ (c(J_2, \nu_2) = c(\tau_4, \nu_2)) < \\ (c(J_3, \nu_2) = c(\tau_9, \nu_2)) \end{aligned} \tag{6.11}$$

- $\vec{B}_{3,t} = (\tau_7, \tau_1, \tau_6)$ *car*

$$\begin{aligned} (c(J_3, \nu_3) = c(\tau_7, \nu_3)) = \\ (c(J_1, \nu_3) = c(\tau_1, \nu_3)) = \\ (c(J_2, \nu_3) = c(\tau_6, \nu_3)) \end{aligned} \tag{6.12}$$

Le flowtime moyen de cette allocation est $C_{mean}(\overrightarrow{A_t}) \simeq 10.67$.

En adoptant la stratégie "Globally Cheapest Job First" (GCJF) illustrée sur la figure 6.2b, les agents ordonnent les jobs par coût maximal sur l'ensemble des nœuds croissant. Si on suppose que les agents ont tous des croyances mises à jour, alors les

ordres sur les jobs sont identiques, c'est-à-dire $J_1 \blacktriangleleft_i J_2 \blacktriangleleft_i J_3$ car :

$$\begin{aligned} \max_{\nu_\ell \in \mathcal{P}} c^i(J_1, \nu_\ell) &= c(\tau_1, \nu_3) = 5 \\ \max_{\nu_\ell \in \mathcal{P}} c^i(J_2, \nu_\ell) &= c(\tau_6, \nu_3) = 5 \\ \max_{\nu_\ell \in \mathcal{P}} c^i(J_3, \nu_\ell) &= c(\tau_9, \nu_2) = 8 \end{aligned} \tag{6.13}$$

Le flowtime moyen de cette allocation est $C_{mean}(\overrightarrow{A_t}) = 9$.

En adoptant la stratégie "Most Expensive Job of Most Loaded Node Last" (MEJMLNL) representée sur la figure 6.2c, les agents ordonnent en dernier le job le plus coûteux pour le nœud le plus chargé puis, en avant-dernier le job restant le plus coûteux pour le nœud le plus chargé sans prendre en compte le job précédent, et ainsi de suite. Si on suppose que les agents ont tous des croyances exactes, alors les ordres sur les jobs sont identiques, c'est-à-dire $J_2 \blacktriangleleft_i J_1 \blacktriangleleft_i J_3$ car :

1. *Les nœuds ν_2 et ν_3 sont les plus chargés, ν_2 est donc prioritaire suivant l'ordre lexicographique. Le job le plus coûteux pour ν_2 est J_3, c'est donc lui qui sera placé en dernier dans les lots de tâches.*
2. *Le nœud le plus chargé en ne considérant que J_1 et J_2 est ν_3. Les jobs J_1 et J_2 ont le même coût pour ν_3, J_1 est donc choisi suivant l'ordre lexicographique.*

Le flowtime moyen de cette allocation est $C_{mean}(\overrightarrow{A_t}) = 9$.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9
$c(\tau, \nu_1)$	5	3	1	8	2	10	4	2	4
$c(\tau, \nu_2)$	10	3	2	4	2	5	2	2	8
$c(\tau, \nu_3)$	5	6	1	4	4	5	2	4	4

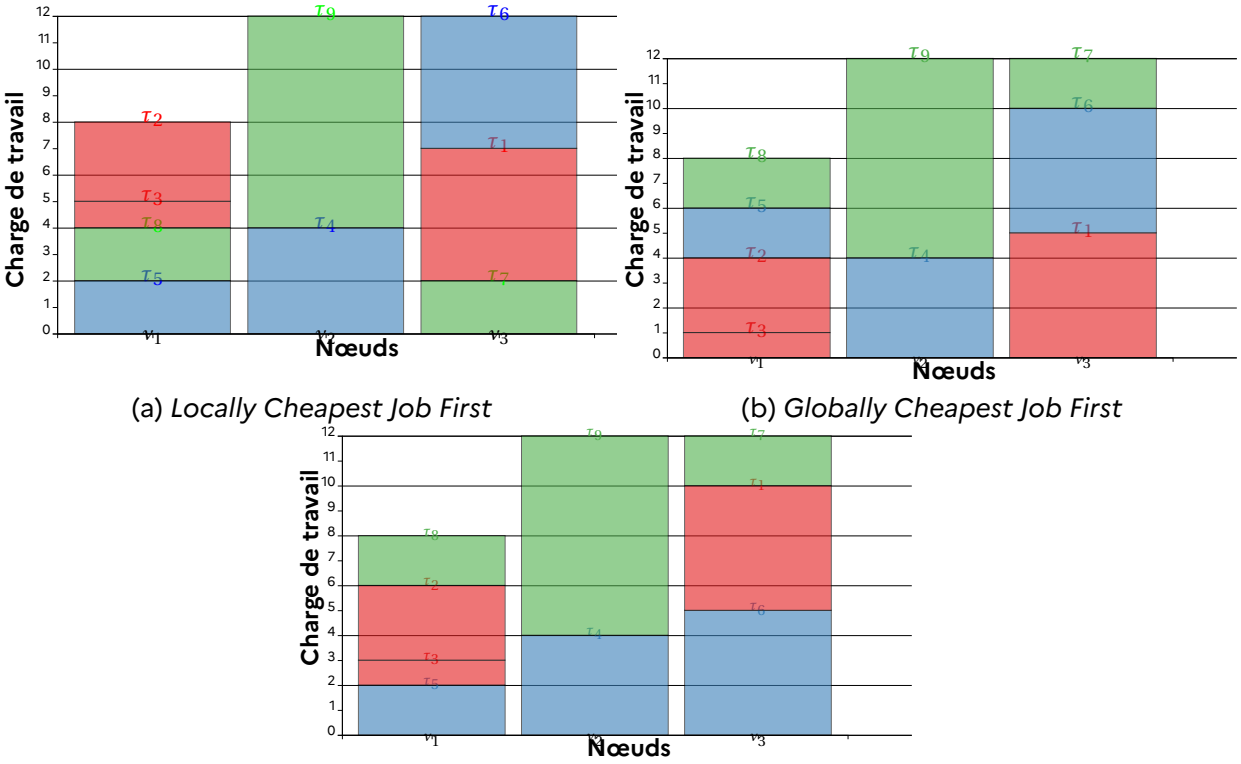
Table 6.1 – Le coût des tâches pour chaque nœud dans l'exemple 6.2

Comme selon les stratégies dites « globales » GCJF et MEJMLNL, les jobs sont ordonnancés dans le même ordre pour tous les nœuds, ces stratégies sont plus efficaces pour réduire le *flowtime* moyen. Cependant, comme elles ne s'appuient pas uniquement sur les connaissances locales du lot de l'agent mais également sur les croyances à propos de ses pairs, il est nécessaire pour tous les agents de mettre à jour leur lot de tâches dès que leur base de croyances est mise à jour, c'est-à-dire à chaque réallocation et consommation même s'ils ne sont pas impliqués. De fait, les coûts communicationnel et computationnel induit par ces stratégies de consommation sont rédhibitoires. C'est pourquoi j'adopte par la suite la stratégie LCJF, cette dernière constituant un compromis entre l'efficacité de l'ordonnancement et le coût de la stratégie multi-agents.

6.5 Protocole d'offres alternées

Les agents réallouent leurs tâches en réalisant des négociations bilatérales à un tour. Chaque négociation reposent sur un protocole d'offres alternées entre le proposant et le répondant et inclut trois étapes de décision :

1. le donneur utilise sa stratégie d'offre pour sélectionner une délégation à un potentiel receveur ;



(a) *Locally Cheapest Job First*

(b) *Globally Cheapest Job First*

(c) *Most Expensive Job of Most Loaded Node Last*

Figure 6.2 – Allocations dans l'exemple 6.2 où tous les agents adoptent une stratégie LCJF, une stratégie GCJF, ou une stratégie MEJMLNL.

2. le receveur peut accepter, refuser la délégation ou proposer une contre-offre ;
3. en cas d'acceptation ou de proposition de contre-offre, le donneur confirme la transaction ou l'annule si elle est obsolète.

Le processus de négociation se déroule en deux phases :

1. lors de la première phase, les agents proposent des délégations socialement rationnelles à leurs pairs, qui peuvent accepter ou refuser une offre de délégation, jusqu'à ce qu'il n'y ait plus de délégation socialement rationnelle identifiée dans l'allocation courante ;
2. lors de la deuxième phase, les agents peuvent proposer des délégations même si elles ne sont pas socialement rationnelles, dans l'espoir de déclencher des échanges si le répondant trouve une contre-offre pertinente à proposer.

La figure 6.3 montre les interactions entre un donneur et un receveur durant la négociation d'une délégation lors de la première phase. La stratégie d'offre du donneur sélectionne une délégation, c'est-à-dire une liste de tâches de son lot, socialement rationnelle et la soumet au receveur avec un message `Propose`. Le receveur applique la règle d'acceptabilité et décide s'il accepte (message `Accept`) ou rejette cette offre (message `Reject`), selon si elle est socialement rationnelle ou non d'après ses croyances. Il est à noter qu'il peut acccepter le lot d'offre en intégralité ou en partie. Si l'offre a été acceptée, le donneur confirme la délégation avec un message `Confirm`. Il l'annule avec un message `Withdraw` si elle est obsolète.

La figure 6.4 montre les interactions entre un donneur et un receveur durant une délégation pouvant aboutir à un échange lors de la deuxième phase. Le proposant sélectionne à l'aide de sa stratégie d'offre une délégation, qui n'est pas nécessairement socialement rationnelle et la soumet au receveur (message `Propose`). Le receveur applique sa stratégie de contre-offre. Si cette dernière renvoie une contre-offre pertinente, c'est-à-dire une tâche à donner en contre-partie permettant d'aboutir à une réallocation socialement rationnelle et avec un gain meilleur en terme de *flowtime* que la délégation seule, le receveur répond au donneur par une proposition d'échange avec un message `CounterPropose`. Le donneur accepte la contre-proposition (message `Confirm`) ou l'annule (message `Withdraw`) si la réallocation est obsolète. Si le receveur n'identifie pas de contre-proposition pertinente, les interactions qui font suite à la réception de la proposition sont les mêmes que celles de la première phases décrite précédemment.

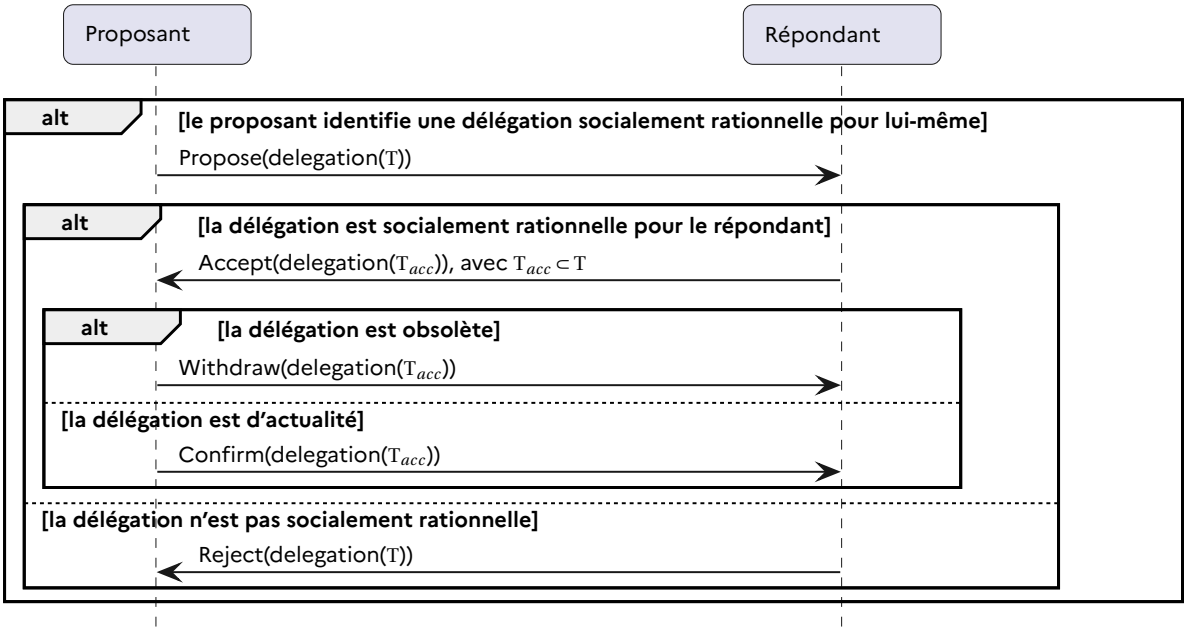


Figure 6.3 – Protocole de négociation bilatérale entre un agent « proposant » et un agent « répondant » pour la délégation de lots de tâches.

6.6 Stratégie de négociation

La stratégie de négociation, qui interface la rationalité de l'agent (la règle d'acceptabilité) et le protocole pour générer et interpréter les offres est constituée d'une stratégie d'offre, d'une stratégie de contre-offre et d'une stratégie d'acceptation. Cette stratégie de négociation repose sur les connaissances de l'agent à propos du lot de tâches du nœud qu'il supervise et des croyances issues de son modèle des pairs.

6.6.1 Stratégie d'offre

La stratégie d'offre permet à l'agent de sélectionner une délégation à soumettre, c'est-à-dire sélectionner une ou des tâches et un receveur, afin de réduire la durée de réalisation d'au moins un de ses jobs. Plus précisément, la stratégie d'offre pour un agent $\nu_i \in \mathcal{P}$ a pour but de sélectionner son lot d'offre, c'est-à-dire une ou plusieurs tâches à

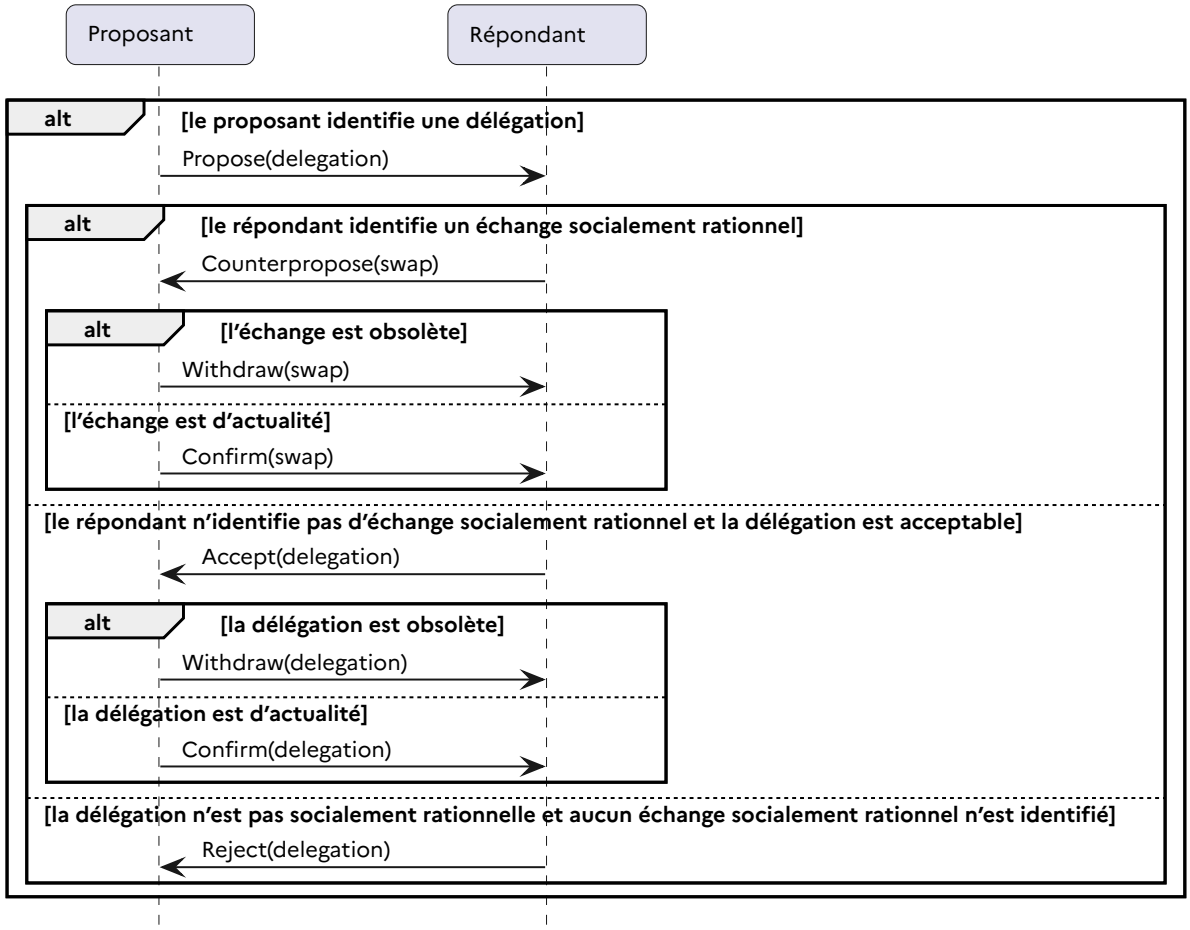


Figure 6.4 – Protocole de négociation bilatérale entre un agent « proposant » et un agent « répondant » pour l'échange de tâches.

déléguer, à un receveur choisi parmi un ensemble $\mathcal{P}' \subseteq \mathcal{P} \setminus \{v_i\}$ afin de réduire le temps de réalisation d'un job sélectionné parmi un ensemble $\mathcal{J}' \subseteq \mathcal{J}_t$, avec initialement $\mathcal{P}' = \mathcal{P}$ et $\mathcal{J}' = \mathcal{J}_t$.

Plus formellement, une stratégie d'offre se découpe en trois étapes décrites ci-dessous.

1. **Sélection d'un job.** L'objectif du donneur est de réduire la durée de réalisation des jobs dont il est responsable. Pour cela, il identifie le job le plus prioritaire parmi ceux dont il pense être facteur limitant, afin d'essayer de réduire non seulement sa durée de réalisation, mais également celle des jobs qui suivent dans son lot.

$$\mathsf{J}_* = \min_{\blacktriangleleft_i}\{\mathsf{J} \in \mathsf{jobs}(\mathsf{B}_{i,t}) \cap \mathcal{J}' \mid v_j = \mathsf{nodeMax}^i(\overrightarrow{\mathsf{A}_t}, \mathsf{J}) = v_i\} \tag{6.14}$$

2. **Sélection d'un receveur.** Le donneur sélectionne un receveur pour la délégation. Afin de ne pas augmenter le *flowtime* des jobs impactés, il sélectionne un receveur v_* pour qui la somme des différences entre la durée de réalisation pour l'allocation et celle pour l'agent est la plus grande,

$$v_* = random(\{\underset{v_j \in \mathcal{P}'}{\mathrm{argmax}}\, \Sigma_{\mathsf{J}_* \blacktriangleleft_j^i \mathsf{J}} (\mathsf{C_J}^i(\overrightarrow{\mathsf{A}_t}) - \mathsf{C_J}^i(\vec{\mathsf{B}}_{j,t}))\}) \tag{6.15}$$

   où $random$ est une fonction de choix pseudo-aléatoire qui à un ensemble de nœuds associe un nœud de cet ensemble.

3. **Sélection du lot d'offre.** Le donneur sélectionne une ou des tâches de son lot appartenant à J_* ou à un job précédent J_* dans son lot et dont la délégation réduirait le *flowtime* moyen. Je distingue dans cette section deux types de stratégie d'offre :
   - une **stratégie de délégation unaire** qui permet au donneur de proposer une offre constituée d'une unique tâche ;
   - une **stratégie de délégation n-aire** qui permet au donneur de proposer un lot d'offre sous la forme d'une liste de plusieurs tâches.

   Ces stratégies sont détaillées dans les sous-sections 6.6.1.1 et 6.6.1.2.

Si aucune délégation n'est déclenchée à l'issue de l'étape 3, l'agent retourne à l'étape 2 pour choisir un autre receveur ($\mathcal{P}' \leftarrow \mathcal{P}' \setminus \{v_*\}$). En cas d'échec, il revient à l'étape 1 pour sélectionner un autre job ($\mathcal{J}'_t \leftarrow \mathcal{J}'_t \setminus \{\mathsf{J}_*\}$). Si à l'issue de ces étapes aucune délégation socialement rationnelle de son point de vue n'a été identifiée, l'agent passe en état de pause et ne fera plus de délégation jusqu'à ce que sa base de croyances soit mise à jour.

Selon l'approche adoptée ici, il est à noter que les stratégies d'offre proposées ne concernent que des délégations. Les échanges peuvent être déclenchés si le receveur propose une contre-offre en retour, ce qui sera explicité dans la section 6.6.3.

6.6.1.1 Délégations unaires

Une **stratégie de délégation unaire** permet au donneur de proposer des offres constituées d'une unique tâche. Il choisit de déléguer la tâche distante dont la délégation réduira le plus le coût estimé d'exécution. En cas d'égalité, il choisit celle qui est la plus prioritaire dans son lot,

$$\forall \mathcal{T}' \subseteq \mathcal{T}_t, \tau_* = \min_{\lhd_i}\{\underset{\tau \in \mathcal{T}' \cap \mathsf{B}_{i,t} \cap \{\mathsf{J} \mid \mathsf{J} = \mathsf{J}_* \vee (\mathsf{J} \blacktriangleleft_i \mathsf{J}_*)\}}{\mathrm{argmax}} \mathsf{c}(\tau, v_i) - \mathsf{c}(\tau, v_*)\} \tag{6.16}$$

Dans le cas des délégations unaires, je distingue deux types de stratégie différentes pour chacune des deux phases du processus de négociation évoquées dans la section 6.5 :

- avec une stratégie **conservatrice**, le donneur ne soumet la délégation que si cette dernière est socialement rationnelle selon ses croyances ;
- avec une stratégie **libérale**, le donneur peut soumettre la délégation même si elle n'est pas socialement rationnelle selon ses croyances. Même si le receveur n'acceptera pas la délégation telle quelle si elle n'est pas rationnelle, cela peut déclencher un échange socialement rationnel si le receveur trouve une contre-offre pertinente à proposer. Il est à noter qu'une stratégie libérale serait trop coûteuse en temps de calcul dans le cas de délégations n-aires en raison du trop grand nombre de possibilités, c'est pourquoi je ne considère ce type de stratégie que dans le cas des délégations unaires.

6.6.1.2 Délégations n-aires

Une **stratégie de délégation n-aire** permet au donneur de proposer plusieurs tâches à la fois. Pour cela, il construit itérativement un lot d'offre sous la forme d'une liste de tâches T_* que le receveur pourra accepter dans son entièreté ou en partie. Comme pour la stratégie unaire, le donneur privilégie les tâches distantes dans le but de réduire leur coût d'exécution lors de la délégation. La stratégie de sélection du lot d'offre est détaillée dans l'algorithme 6.2. Il s'agit d'un algorithme où le donneur ajoute itérativement à chaque étape la tâche dont la délégation réduit le plus le coût d'exécution, tant que l'ajout de la tâche dans le lot d'offre permet de diminuer le *flowtime* moyen. Passons en revue les différentes étapes de l'algorithme.

1. Les lignes 2 et 3 de l'algorithme sélectionnent comme tâches candidates les tâches distantes appartenant à J_* ou aux jobs le précédent dans le lot du donneur et dont la délégation permet de diminuer le coût d'exécution. Ces tâches candidates sont rangées dans une liste par gain de coût d'exécution croissant dans une liste T'.
2. Tant qu'il reste des tâches candidates (ligne 5), on vérifie si l'ajout de la première tâche candidate de T' dans le lot d'offre permettrait d'améliorer le *flowtime* (ligne 8). Si oui, cette tâche est ajoutée au lot d'offre T_* (ligne 9) et retirée de la liste des tâches candidates T'. Sinon, la recherche s'arrête et l'algorithme renvoie le lot d'offre T_* (ligne 13).
3. Si toutes les tâches candidates ont été étudiées et que la recherche n'a pas été interrompue avant, l'algorithme renvoie le lot d'offre T_* (ligne 16). Il est à noter que si la recherche ne s'est pas terminée lors de la boucle *Tant que* (lignes 5 à 14), cela signifie que toutes les tâches candidates de T' ont été ajoutées au lot d'offre.

Cette stratégie est illustrée dans l'exemple 6.3 suivant.

Exemple 6.3 (Stratégie de négociation).
Soit le problème STAP de l'exemple 5.1 avec l'allocation initiale $\overrightarrow{A_t}$ rappelée sur la figure 6.5a telle que $\vec{B}_{1,t} = (\tau_5, \tau_1)$, $\vec{B}_{2,t} = (\tau_3, \tau_2, \tau_7, \tau_8, \tau_9)$ et $\vec{B}_{3,t} = (\tau_4, \tau_6)$. Le flowtime est $C(\overrightarrow{A_t}) = 7+9+17 = 33$. Je considère ici que les agents ont des croyances qui sont exactes. L'agent ν_2 applique la stratégie de délégation n-aire de la façon suivante pour sélectionner une offre :

1. *il sélectionne le job le plus prioritaire pour lequel il est limitant selon l'équation 6.14, $J_* = J_3$;*
2. *il sélectionne le receveur qui est le moins limitant pour J_3 selon l'équation 6.15. Comme ni ν_1 ni ν_3 ne possèdent de tâche de J_3, l'agent ν_2 choisit aléatoirement,*

Algorithme 6.2 : Construction du lot d'offre par l'initiateur ν_i

Entrées : J_* le job sélectionné dans l'étape 1,
ν_* le receveur sélectionné dans l'étape 2
Sorties : T_* le lot d'offre sélectionné

```
1  T* ← empty_stack;
2  T = {τ | job(τ) = J* ∨ (job(τ) ◄_i J*)};
3  T' ← (...,τ^k,...,τ^l,...) | τ^i ∈ T ∧ (k < l ⇔ c(τ^k,ν_i) − c(τ^k,ν*) > c(τ^l,ν_i) − c(τ^l,ν*)) /* la liste des
       tâches par gain de coût d'exécution décroissant                                              */
4  bestFlowtime = C^i(A_t);
5  tant que T' ≠ ∅ faire
6  |  τ* ← head(T');
7  |  T' ← tail(T');
8  |  si C^i(δ(T* ∪ {τ*}, ν_i, ν*, A_t)) < bestFlowtime alors
9  |  |  T*.push(τ*);
10 |  |  bestFlowtime ← C^i(δ(T*, ν_i, ν*, A_t));
11 |  fin
12 |  sinon
13 |  |  Retourner T* ;
14 |  fin
15 fin
16 Retourner T* ;
```

$\nu_* = \nu_1$;

3. *l'algorithme 6.2 permet à l'agent ν_2 de sélectionner son lot d'offre :*
   (a) *les tâches candidates, c'est-à-dire les tâches de J_3 ou les précédentes dans son lot de tâches, sont triées par gain de coût décroissant,* $T' = [\tau_9, \tau_3, \tau_2, \tau_8, \tau_7]$;
   (b) *la délégation de la tâche τ_9, aboutissant à l'allocation représentée sur la figure 6.5b améliore la durée de réalisation,*

$$C^2(\delta([\tau_9], \nu_2, \nu_1, \overrightarrow{A_t})) = 11 + 9 + 9 = 29 < 33 \tag{6.17}$$

   La tâche τ_9 est donc ajoutée au lot d'offre, $T_* = [\tau_9]$,
   (c) *la délégation des tâches τ_3 et τ_9 aboutissant à l'allocation représentée sur la figure 6.5c améliore la durée de réalisation,*

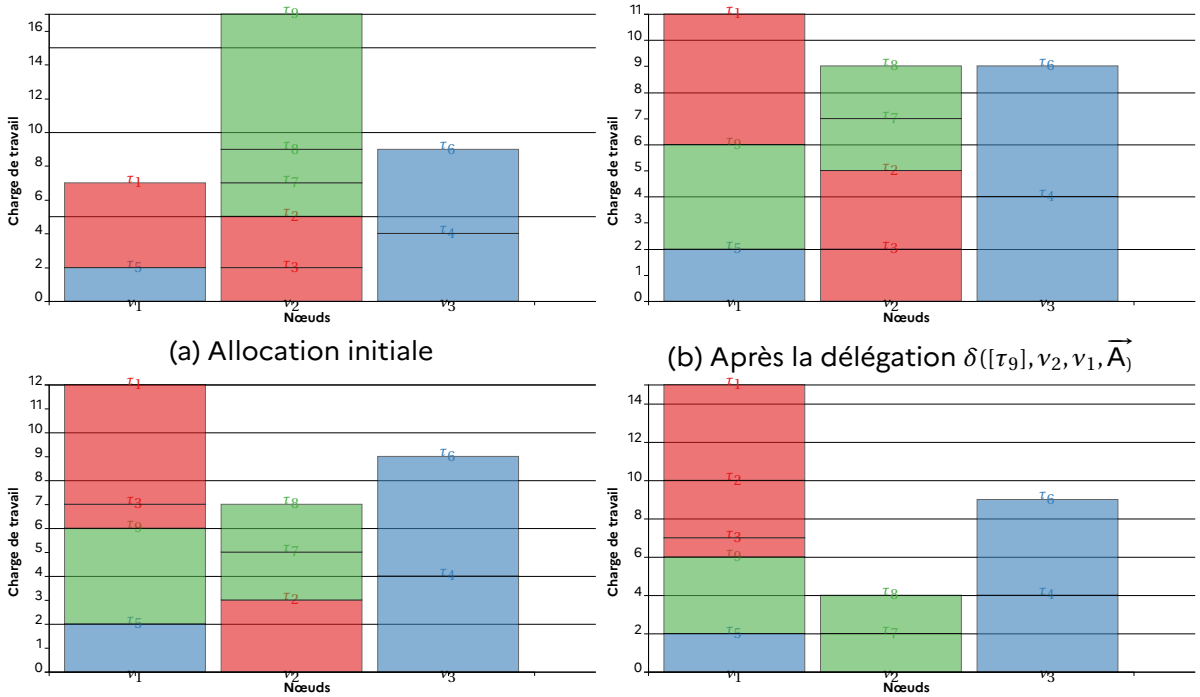
$$C^2(\delta([\tau_9, \tau_3], \nu_2, \nu_1, \overrightarrow{A_)}) = 12 + 9 + 7 = 28 < 29 \tag{6.18}$$

   La tâche τ_3 est ajoutée au lot d'offre, $T_* = [\tau_9, \tau_3]$,
   (d) *la délégation des tâches τ_2, τ_3 et τ_9 n'améliore pas la durée de réalisation (cf. figure 6.5c),*

$$C^2(\delta([\tau_9, \tau_3, \tau_2], \nu_2, \nu_1, \overrightarrow{A_t})) = 15 + 9 + 6 = 30 > 28 \tag{6.19}$$

   Le lot d'offre sélectionné est $T_* = [\tau_9, \tau_3]$.

En résumé, l'agent ν_2 propose à ν_1 la délégation $\delta([\tau_9, \tau_3], \nu_2, \nu_1, \overrightarrow{A_t})$.



(a) Allocation initiale

(b) Après la délégation $\delta([\tau_9], \nu_2, \nu_1, \overrightarrow{A})$

(c) Après la délégation $\delta([\tau_9, \tau_3], \nu_2, \nu_1, \overrightarrow{A})$

(d) Après la délégation $\delta([\tau_9, \tau_3, \tau_2], \nu_2, \nu_1, \overrightarrow{A})$

Figure 6.5 – Allocations de l'exemple 6.3

6.6.2 Stratégie d'acceptation

La stratégie d'acceptation permet au receveur d'une proposition de délégation de décider si la délégation doit être acceptée ou rejetée. La stratégie d'acceptation repose sur les croyances de l'agent et sur la règle d'acceptabilité définis dans les sections 6.2 et 6.3. Autrement dit, le receveur accepte une proposition de délégation lorsque cette dernière permet, selon ses croyances, d'améliorer le *flowtime*, c'est-à-dire si elle est socialement rationnelle.

Dans le cas de la stratégie unaire et où les échanges ne sont pas envisagés, le receveur accepte la délégation si elle vérifie la règle d'acceptabilité définie dans l'équation 6.3. Si la règle n'est pas vérifiée, il rejette l'offre.

Dans le cas d'une délégation n-aire, le receveur vérifie dans un premier temps l'acceptabilité du lot d'offre dans son entièreté. Si la délégation du lot d'offre vérifie la règle d'acceptabilité, le receveur accepte le lot d'offre dans son intégralité. Dans le cas contraire, il dépile une à une les tâches du lot d'offre et accepte le sous-lot dès lors que la transaction lui semble rationnelle. Cette stratégie est décrite formellement dans l'algorithme 6.3.

6.6.3 Stratégie de contre-offre

La stratégie de contre-offre permet au receveur de compléter une proposition de délégation $\delta(\tau_1, \nu_i, \nu_j, \overrightarrow{A_t})$ par une proposition d'échange. Elle consiste pour le receveur ν_j à sélectionner une tâche en échange de celle proposée par le donneur ν_i et à vérifier si l'échange ainsi envisagé lui semble acceptable et meilleur, c'est-à-dire qu'il « améliore

Algorithme 6.3 : Sélection par le receveur ν_* du sous-lot parmi le lot d'offre reçu

Entrées : T_* le lot d'offre reçu
Sorties : T_{acc} le lot d'offre accepté

```
T_acc ← T_* ;
tant que T_acc ≠ ∅ et δ(T_acc, ν_i, ν_*, A_t) n'est pas acceptable par l'agent ν_* faire
    T_acc.pop ;
fin
Retourner T_acc ;
```

plus le *flowtime* », que la délégation proposée.

1. **Étape de sélection d'une tâche.** L'agent ν_j sélectionne la tâche non locale la plus prioritaire selon sa stratégie de consommation, dans le but de réduire son coût d'exécution, parmi l'ensemble de tâches $\mathcal{T}'$, où $\mathcal{T}'$ est initialement l'ensemble des tâches de ν_j,

$$\tau_* = \min_{\lhd_j} \{\tau \in \mathcal{T}' \cap \vec{\mathsf{B}}_{j,t} \mid \mathsf{c}(\tau, \nu_j) - \mathsf{c}(\tau, \nu_i) > 0\} \tag{6.20}$$

   Le critère vérifiant que le coût de la tâche pour le receveur ν_j est supérieur à celui pour le donneur ν_i traduit une meilleure localité pour le donneur.

2. **Étape de validation.** L'agent ν_j vérifie en se basant sur ses croyances que :
   - l'échange obtenu lui semble acceptable, c'est-à-dire qu'il respecte la règle d'acceptabilité définie dans la section 6.3 ;
   - l'échange lui semble meilleur que la délégation initialement proposée, c'est-à-dire que la réduction du *flowtime* induite est plus importante qu'avec la délégation seule,

$$\begin{aligned} &\Sigma_{\mathsf{J}\in\mathcal{J}_t} \max_{\forall \nu_o \in \mathcal{N}\setminus\{\nu_i,\nu_j\}} (\mathsf{C_J}^j(\overrightarrow{\mathsf{B}_{i,t} \ominus \tau_1 \oplus \tau_*}), \mathsf{C_J}^j(\overrightarrow{\mathsf{B}_{j,t} \ominus \tau_* \oplus \tau_1}), \mathsf{C_J}^k(\mathsf{B}_{o,t})) \\ &< \Sigma_{\mathsf{J}\in\mathcal{J}_t} \max_{\forall \nu_o \in \mathcal{N}\setminus\{\nu_i,\nu_j\}} (\mathsf{C_J}^j(\overrightarrow{\mathsf{B}_{i,t} \ominus \tau_1}), \mathsf{C_J}^j(\overrightarrow{\mathsf{B}_{j,t} \oplus \tau_1}), \mathsf{C_J}^j(\mathsf{B}_{o,t})) \end{aligned} \tag{6.21}$$

Si l'étape de validation échoue, l'agent retourne à l'étape 1 pour choisir une autre tâche dans l'ensemble $\mathcal{T}' \leftarrow \mathcal{T}' \setminus \{\tau_*\}$.

Exemple 6.4 (Stratégie de contre-offre).
Considérons l'allocation $\overrightarrow{A_t}$ pour le problème STAP de l'exemple 5.1 où l'agent ν_1, pour lequel on suppose les croyances à jour, reçoit de la part de l'agent ν_2 la proposition de la délégation $\delta(\tau_9, \nu_2, \nu_1, \overrightarrow{A_t})$. L'agent ν_1 sélectionne la tâche avec le gain le plus important (cf. l'équation 6.20), $\tau_* = \tau_5$. L'échange est acceptable, le flowtime global décroît strictement et il est meilleur que celui après la seule délégation (cf. figures 5.3b et 5.3c). En résumé, le répondant ν_1 propose τ_5 en contre-partie de la tâche τ_9 pour atteindre l'allocation $\overrightarrow{A_t}' = \sigma(\tau_9, \tau_5, \nu_2, \nu_1, \overrightarrow{A_t})$.

6.7 Synthèse

Dans ce chapitre, j'ai proposé mon modèle multi-agents où des agents, chacun supervisant un nœud de calcul, collaborent pour aboutir à une meilleure répartition des tâches. J'ai notamment proposé une stratégie d'agent négociant pour détecter des opportunités d'échange de tâches. J'ai présenté ici les différents ingrédients nécessaires à la négociation entre les agents.

Pour négocier, ainsi que pour ordonner leurs lots de tâches, les agents s'appuient sur leur modèle des pairs construit à partir des croyances sur les lots de tâches de leurs pairs. Ainsi, un agent possède des croyances sur les coûts des jobs de ses pairs pour en déduire leurs durées de réalisation.

Les agents ordonnent leurs lots de tâches en s'appuyant sur leur stratégie de consommation. Le *flowtime* moyen que je cherche à minimiser dépend de l'ordre d'exécution des jobs, et non des tâches. Ainsi, les stratégies orientées jobs, qui ordonnent d'abord les lots par jobs et regroupent les tâches d'un même job au sein d'un lot, sont plus efficaces et sont donc privilégiées dans cette thèse.

Le processus de négociation est subdivisé en deux phases où la stratégie multi-agents :

1. recherche des délégations socialement rationnelles ;
2. recherche des échanges socialement rationnels quitte à envisager des délégations qui ne le sont pas.

Les agents négocient en suivant un protocole d'offres alternées à un tour. Ils adoptent une stratégie de négociation constituée :

- d'une stratégie d'offre qui permet de proposer des délégations ;
- d'une stratégie d'acceptation qui permet d'accepter une offre en intégralité, en partie ou de la refuser ;
- d'une stratégie de contre-offre qui permet de répondre à une proposition de délégation par une proposition d'échange ;
- d'une règle d'acceptablité qui se base sur le critère de rationalité sociale.

Afin de détecter des opportunités de délégations n-aires, le donneur propose un lot d'offre que le receveur peut accepter, éventuellement en partie, ou refuser.
Que la proposition reçue soit une délégation unaire ou n-aire, elle doit être considérée comme socialement rationnelle par le receveur pour être acceptée. Afin de détecter des opportunités d'échanges unaires, le proposant émet une offre de délégation unaire, socialement rationnelle ou non, à l'aide de la stratégie de délégation libérale. Le receveur peut alors proposer une meilleure contre-offre ou refuser. Il est à noter que la stratégie d'offre libérale, ainsi que la stratégie de contre-offre qu'elle est susceptible de déclencher, ont un coût computationnel important. C'est la raison pour laquelle elles sont déployées dans une seconde étape, lorsque plus aucune délégation socialement rationnelle n'est détectée.

Le chapitre suivant est dédié à la mise en œuvre des mécanismes décrits ici, notamment grâce à l'architecture des agents nœuds qui permet la réalisation en parallèle des consommations et des négociations ainsi que la mise en œuvre des comportements d'agents lors des négociations.

Chapitre 7

Mise en œuvre

Sommaire

7.1 Introduction 79
7.2 Architecture d'agents 80
7.3 Interactions entre agents composants 81
7.3.1 Gestionnaire-Exécutant 81
7.3.2 Gestionnaire-Négociateur 82
7.4 Comportements 83
7.4.1 Superviseur 85
7.4.2 Exécutant 85
7.4.3 Gestionnaire 86
7.4.4 Négociateur 88
7.5 Implémentation 92
7.6 Synthèse 93

7.1 Introduction

Ce chapitre est axé sur la mise en œuvre des mécanismes étudiés dans les chapitres 5 et 6. Pour pouvoir gérer simultanément les consommations et les réallocations des tâches, les agents sont dotés d'une architecture composite basée sur la séparation des rôles. Ainsi, chaque agent-nœud est constitué de trois « sous-agents », appelés **agents composants**, gérant respectivement la consommation, la réallocation et la gestion du lot de tâches.

Comme mentionné dans le chapitre 6, le processus de réallocation se déroule en deux phases :

1. une première phase où les agents réalisent uniquement des délégations, appliquant une stratégie d'offre conservatrice, comme définie dans la section 6.6, et suivant le protocole de négociation décrit sur la figure 6.3 ;
2. une deuxième phase où les agents appliquent une stratégie d'offre libérale et suivent le protocole de négociation représenté sur la figure 6.4. pour réaliser des échanges, plus coûteux que les délégations mais utiles pour s'extraire des minima locaux

lorsque plus aucune possibilité de délégation socialement rationnelle n'est trouvée.

Pour cela, en plus des agents-nœuds, le système dispose d'un agent superviseur, comme illustré sur la figure 4.1 dont le rôle est de synchroniser ces deux phases, en plus de démarrer le processus et de l'arrêter lorsque toutes les tâches ont été consommées.

Mon modèle de comportement d'agent est une machine à états finis où chaque transition est étiquettée par l'événement qui la déclenche, par exemple la réception d'un message, et par les actions qu'elle amorce comme l'émission d'une réponse.

La section 7.2 décrit l'architecture composite des agents-nœuds constitués de trois agents composants tandis que la section 7.3 est dédiée aux interactions entre ces différents agents composants. La section 7.4 détaille les comportements des agents. Enfin, la section 7.5 décrit comment le comportement d'agent est implémenté.

7.2 Architecture d'agents

Pour concevoir un agent-nœud, j'ai adopté une architecture modulaire qui permet la concurrence des négociations et des consommations, ainsi que la séparation entre les comportements de communication et de décision.

Un agent-nœud est un agent composite constitué de trois agents composants, comme représenté sur la figure 7.1, chacun ayant un rôle limité :

- l'**exécutant** exécute, c'est-à-dire consomme les tâches, comme décrit dans la section 5.3.1 ;
- le **négociateur** s'appuie sur un modèle des pairs, telle que décrit dans la section 6.2 pour négocier des tâches en appliquant les stratégies de négociation détaillées dans la section 6.6 ;
- le **gestionnaire** gère le lot de tâches du nœud de calcul en y ajoutant ou supprimant les tâches selon les réallocations bilatérales marchandées par le négociateur. Il ordonnance les tâches selon les stratégies de consommation présentées dans la section 6.4, définissant dans quel ordre il va les confier à l'exécutant pour leur consommation.

Dès que le gestionnaire est informé que l'exécutant est libre, il fournit à ce dernier la prochaine tâche à exécuter conformément à la stratégie de consommation, quitte à annuler la réallocation de cette tâche en cours de négociation, afin de prioriser la consommation. Cette tâche n'est alors plus éligible pour une potentielle réallocation. En effet, le but de la réallocation est de rendre la consommation plus efficace. Elle ne doit donc pas la ralentir en retardant la consommation d'une tâche pour la réallouer.

L'agent composite sert d'interface pour ses agents composants qui ne communiquent pas directement avec les agents composants des autres nœuds. C'est donc lui qui reçoit les messages venant des autres agents-nœuds et qui se charge de les transmettre à l'agent composant concerné. L'agent composant négociateur est le seul qui a besoin d'interagir avec les pairs, comme le montre la figure 7.1. Cependant, en pratique, les messages sont reçus d'abord par l'agent-nœud composite qui sert d'intermédiaire, comme par exemple lors de la réception d'une proposition de délégation qui est transmise à l'agent négociateur. L'agent composite sert également d'intermédiaire entre les

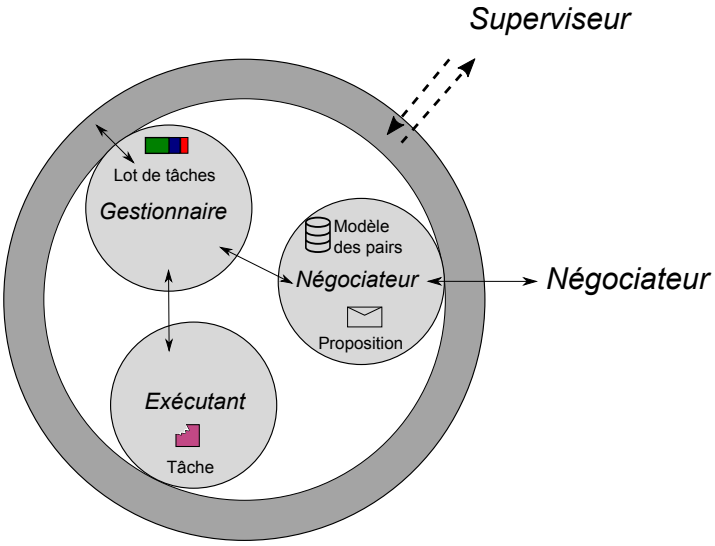


Figure 7.1 – Architecture composite d'un agent-nœud

agents composants et l'agent superviseur. Au sein d'un agent composite, l'exécutant et le négociateur ne communiquent pas entre eux. Ils échangent des messages uniquement avec le gestionnaire qui les coordonne en maintenant le lot de tâches.

À la différence de l'architecture proposée par Baert (2019), la base de croyances n'est pas associée au gestionnaire mais au négociateur qui est autonome vis-à-vis du gestionnaire pour conduire l'ensemble des fils de négociations bilatérales.

7.3 Interactions entre agents composants

Cette section résume les interactions entre les agents composants au sein d'un même agent composite. Comme précisé dans la section précédente, l'exécutant et le négociateur ne communiquent pas entre eux. Ils interagissent uniquement avec le gestionnaire, qui sert d'intermédiaire en gérant le lot de tâches.

Les interactions entre le gestionnaire et l'exécutant sont abordées dans la sous-section 7.3.1 tandis que la section 7.3.2 décrit les interactions entre le gestionnaire et le négociateur. Dans les deux cas, ces interactions sont représentées sur la forme de diagrammes d'interaction, (voir figures 7.2a, 7.2b, 7.3 et 7.4). Les flèches pleines représentent des appels synchrones, les flèches ouvertes des messages asynchrones et les lignes en pointillés des messages de réponse.

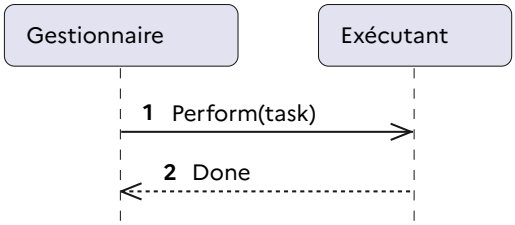
7.3.1 Gestionnaire-Exécutant

L'exécutant est chargé de consommer les tâches. Il interagit avec le gestionnaire qui lui indique quelle tâche consommer et il peut aussi signaler à ce dernier le temps restant qu'il estime pour la tâche qu'il est en train de traiter.

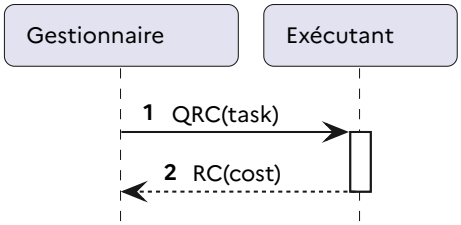
Le gestionnaire peut confier une tâche à l'exécutant. Il envoie alors un message `Perform`. Quand la tâche est accomplie, l'exécutant répond par un message `Done` pour signaler au gestionnaire qu'il a terminé et qu'il est disponible pour recevoir une nouvelle

tâche à consommer. Cette interaction est représentée sur la figure 7.2a.

Pour raffiner son estimation du délai d'attente des tâches dans son lot (cf. définition 5.6), le gestionnaire peut demander à l'exécutant une estimation du temps d'exécution restant pour la tâche en cours avec un message `QRC(task)`, pour *Query Remaining Cost* (cf. figure 7.2b). L'exécutant répond alors avec un message RC, pour *Remaining Cost*. Contrairement à l'interaction `Perform`/`Done` précédemment décrite, cette interaction est synchrone : le gestionnaire attend la réponse de l'exécutant et ne fait rien d'autre en attendant.



(a) Le gestionnaire donne une tâche à l'exécutant, qui répond par un message `Done` quand il a terminé d'exécuter la tâche. Cette interaction est asynchrone.



(b) Le gestionnaire demande à l'exécutant le coût estimé restant pour la tâche en cours de consommation. Cette interaction est synchrone.

Figure 7.2 – Interactions entre le négociateur et l'exécutant

7.3.2 Gestionnaire-Négociateur

Les interactions entre le gestionnaire et le négociateur lors de la première phase de négociation dédiée aux délégations, sont représentées sur la figure 7.3 dans le cas d'une délégation unaire. Le protocole correspond à celui présenté dans la figure 6.3 du chapitre 6. Lorsqu'une réallocation bilatérale est négociée, le négociateur demande de façon synchrone à son gestionnaire de mettre à jour le lot de tâches, pour retirer ou ajouter les tâches réallouées. La réponse du gestionnaire lui permet de mettre à jour sa base de croyances pour pouvoir confirmer la transaction et s'engager plus tard dans de nouvelles négociations. Lorsqu'il met à jour le lot de tâches, le gestionnaire demande de manière synchrone à l'exécutant le temps estimé restant pour la tâche en cours d'exécution afin d'avoir une estimation plus fine du délai d'attente des tâches du lot, comme spécifié dans la section précédente 7.3.1. Le gestionnaire peut donc ensuite répondre au négociateur pour lui transmettre les nouveaux coûts estimés pour le lot de tâches à l'aide d'un message `GiveUpdatedBundle`.
Lors d'une seconde phase de négociation, suivant le protocole d'interactions 6.4 du chapitre 6, les agents marchandent des échanges de tâches. Les interactions entre le gestionnaire et le négociateur, représentées sur la figure 7.4 sont similaires à celles dans le cas des délégations de la figure 7.3 : lors de la transaction, le négociateur attend de manière synchrone la mise à jour du lot de tâches par le gestionnaire, qui demande de façon synchrone à l'exécutant le coût estimé restant de la tâche en cours.

Au-delà du protocole de négociation spécifié dans la figure 6.4 du chapitre 6, la concurrence du processus de réallocation avec le processus de consommation néces-

site la mise en œuvre d'un double acquittement. Quand la réallocation est un échange, elle doit être non seulement confirmée par le proposant pour garantir la disponibilité de la tâche à déléguer mais également par le répondant pour garantir la disponibilité de la contre-partie.

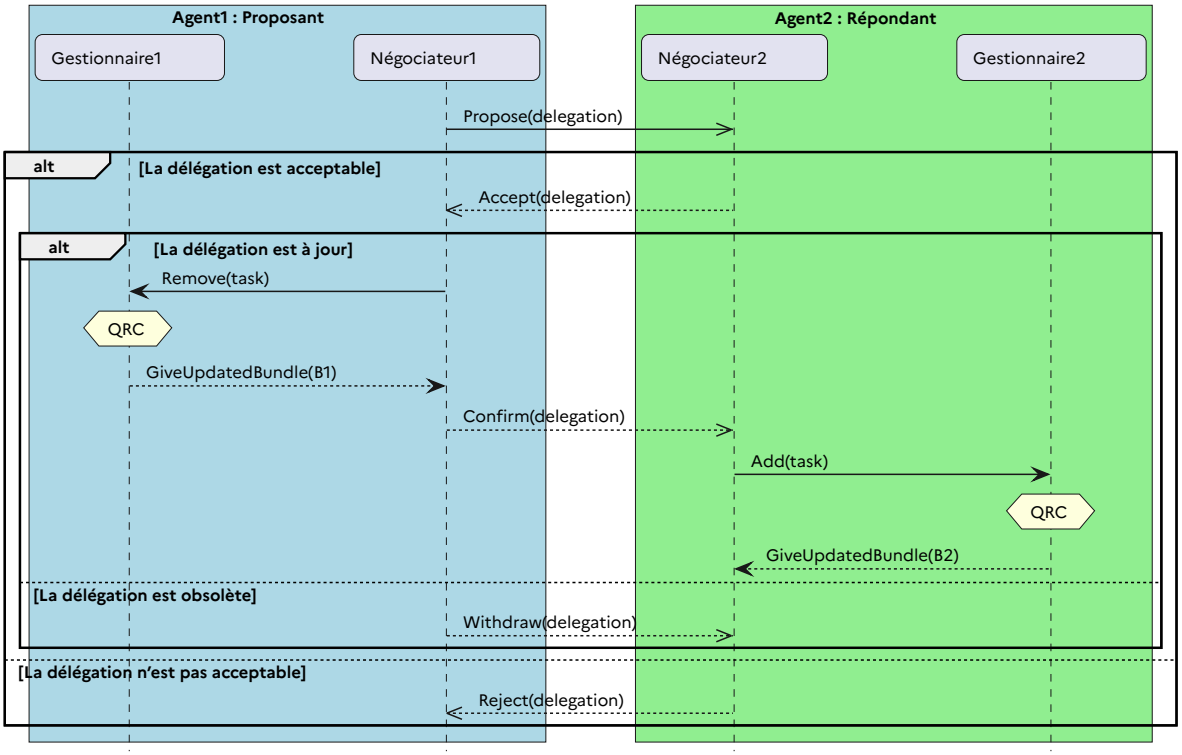


Figure 7.3 – Interactions entre le gestionnaire et le négociateur lors de la première phase de négociation dans le cas d'une délégation unaire. Les étiquettes QRC indiquent que le gestionnaire interagit avec l'exécutant selon le protocole de la figure 7.2b pour prendre en compte le temps d'exécution restant pour la tâche en cours.

7.4 Comportements

Les comportements des agents sont spécifiés sous la forme d'automates finis. Un agent peut être dans différents états et va se comporter différemment selon l'état dans lequel il se trouve, le premier message dans sa file et certaines conditions, représentées généralement par des variables d'état dans son état mental. Selon ces conditions réunies, l'agent peut réagir par diverses actions en réponse, comme envoyer des messages ou mettre à jour son état mental. Chaque type d'agent possède un comportement qui lui est propre. Cette section est dédiée à la description de ces comportements.

Pour faciliter la lecture, les automates présentés dans ce chapitre sont sous forme simplifiée, mais leurs versions complètes sont disponibles dans le chapitre A de l'Annexe. Dans ces automates, la réception d'un message envoyé par un agent `exp` est notée « `exp:message` ». L'envoi d'un message asynchrone à un agent `dest` est noté « `dest!message` », tandis que l'envoi d'un message synchrone est noté « `dest?message` ».

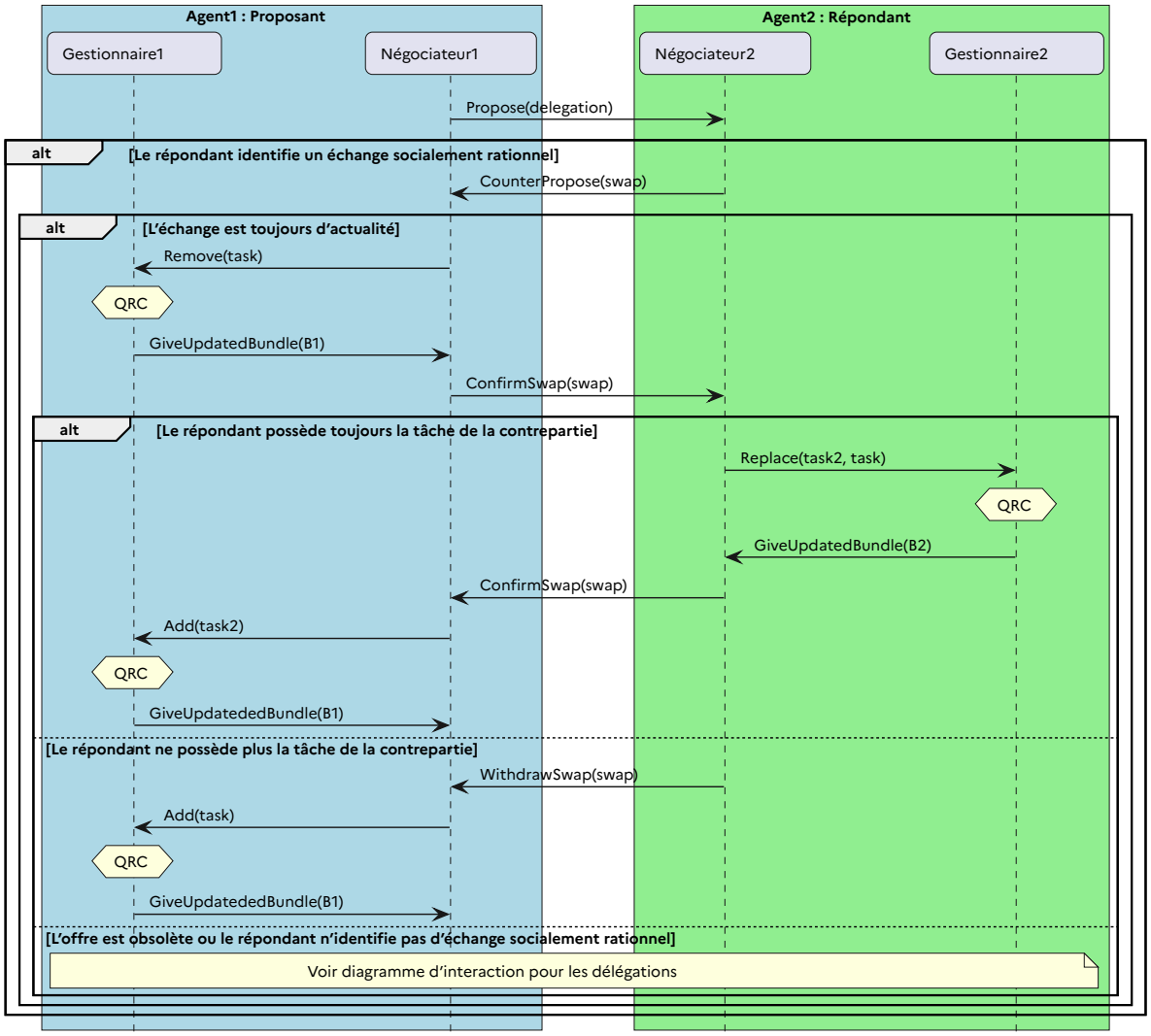


Figure 7.4 – Interactions entre le gestionnaire et le négociateur lors de la deuxième phase de négociation. Dans le cas où le répondant n'identifie pas d'échange socialement rationnel, les interactions sont les mêmes que dans le cas des délégations (cf. figure 7.3).

7.4.1 Superviseur

Dénué de mécanismes de prise de décision, le superviseur a pour rôle de distribuer les lots de tâches aux agents nœuds au début du processus et de synchroniser les différentes phases de négociation.

Comme évoqué dans le chapitre 6, les négociations se déroulent selon deux phases différentes, desquelles dépendent la stratégie appliquée par les agents :

1. une première phase où les proposants adoptent la stratégie d'offre conservatrice et les répondants acceptent une délégation s'ils la considèrent acceptable ou la rejettent dans le cas contraire ;
2. une deuxième phase où les proposants adoptent la stratégie d'offre libérale et les répondants appliquent la stratégie de contre-offre pour déclencher des échanges.

Ces phases alternent jusqu'à la fin du processus, c'est-à-dire lorsque toutes les tâches ont été consommées, comme illustré sur la figure 7.5. Tant qu'il reste des tâches à consommer, le superviseur re-déclenche la première phase à l'issue de la deuxième phase.

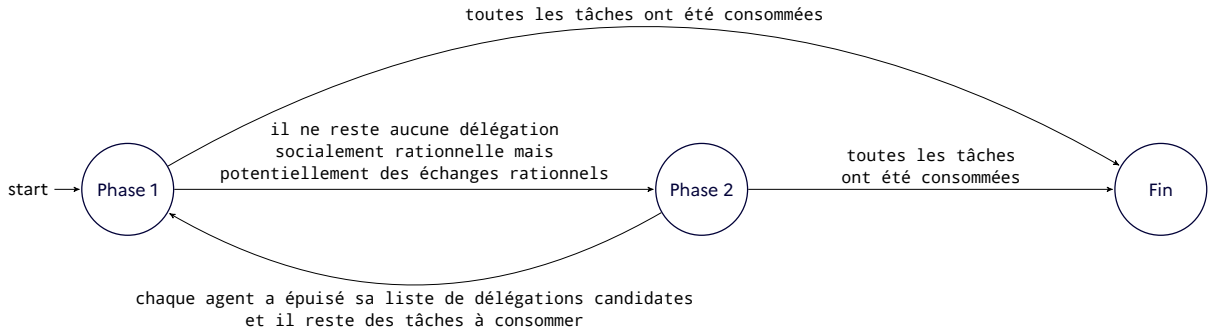


Figure 7.5 – Alternance des phases de négociation

Quand il est informé qu'aucun agent-nœud ne détecte d'opportunité de réallocation, le superviseur enclenche le changement de phase de négociation. Il collecte les métriques nécessaire à la surveillance (*monitoring*) et à l'évaluation empirique de la stratégie multi-agents (cf. chapitre 8) puis stoppe le processus lorsque toutes les tâches ont été consommées.

Le comportement du superviseur est détaillé en Annexe sur la figure A.4.

7.4.2 Exécutant

L'exécutant est un agent composant de l'agent-nœud. Son comportement est représenté sur la figure 7.6.

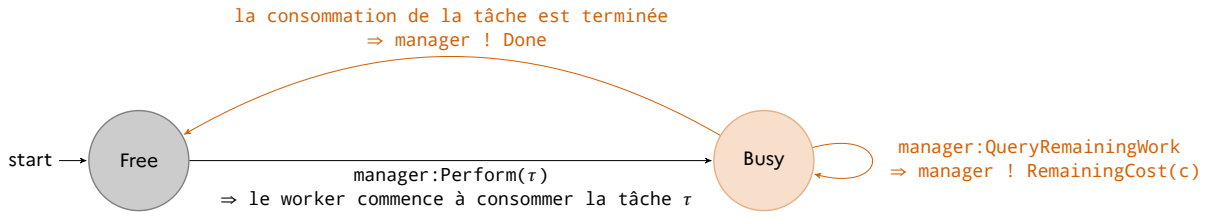


Figure 7.6 – Comportement de l'exécutant

L'agent composant exécutant est soit disponible, soit en train d'exécuter une tâche et il peut alors estimer le temps d'exécution restant pour la tâche en cours. Il dispose donc de deux états différents : un état *Free* lorsqu'il attend de recevoir une tâche à consommer et un état *Busy* lorsqu'il est en train de consommer une tâche.

7.4.3 Gestionnaire

L'agent composant gestionnaire gère le lot de tâches et coordonne les opérations de consommation des tâches réalisées par l'exécutant avec celles de réallocations marchandées par le négociateur. Quand ce dernier lui signale qu'il n'a plus de délégation socialement rationnelle à proposer et qu'il attend les propositions des autres agents de nœud, le gestionnaire en informe le superviseur. Il continue également de distribuer les tâches à exécuter à l'exécutant jusqu'à ce que son lot soit vide.

Le gestionnaire, dont le comportement est représenté de manière simplifiée dans la figure 7.7, peut être dans l'un des états suivants :

- l'état *Initial* quand le gestionnaire attend de recevoir le lot de tâches de la part du superviseur ;
- l'état *Running* quand il y a encore des tâches à consommer et que les négociations sont en cours ;
- l'état *Inactive* quand il n'y a plus de tâches à consommer mais que les négociations sont encore en cours ;
- l'état *EndStage* quand les négociations pour la phase en cours sont terminées ;
- l'état *TransitionStage* quand une nouvelle phase de négociation est sur le point de commencer et que le gestionnaire attend le signal du superviseur.

Le gestionnaire commence dans l'état *Initial*, dans l'attente de recevoir son lot de tâches de la part du superviseur (`supervisor:Give(`$B_{i,t}$`)`). Si le lot contient des tâches, il entre dans l'état *Running* et confie à l'exécutant la tâche la plus prioritaire selon la stratégie de consommation (`worker!Perform(nextTask)`). Si le lot est vide, il entre l'état *Inactive*.

Lorsqu'il est dans l'état Running, le gestionnaire communique à la fois avec l'exécutant pour la consommation des tâches et avec le négociateur pour la mise à jour du lot suite aux réallocations. Lorsque l'exécutant lui signale la fin de la consommation de la tâche qui lui a été confiée (`worker:Done`), le gestionnaire lui confie la tâche suivante, s'il y en a une (`worker!Perform(nextTask)`). Si le lot est vide, il passe dans l'état *Inactive*. Lorsque le négociateur signale une réallocation (`negotiator:Replace(`τ_{swap}, τ`)` ou `negotiator:Add(`τ`)` ou `negotiator:Remove(`τ`)`), le gestionnaire met à jour le lot et demande également à l'exécutant le temps restant de la tâche en cours (`worker?QueryRemainingCost(`τ_{exec}`)` où τ_{exec} désigne la tâche en cours d'exécution).

Dans l'état *Inactive*, le gestionnaire repasse dans l'état *Running* quand le négociateur lui demande d'ajouter au lot une nouvelle tâche reçue lors d'une réallocation (`negotiator:Add(`τ`)`). Le gestionnaire confie alors la dîte tâche à l'exécutant (`worker!Perform(`τ`)`).

Le gestionnaire peut recevoir dans l'état *Running* ou *Inactive* un message du négociateur l'informant de la fin de la phase actuelle de négociation

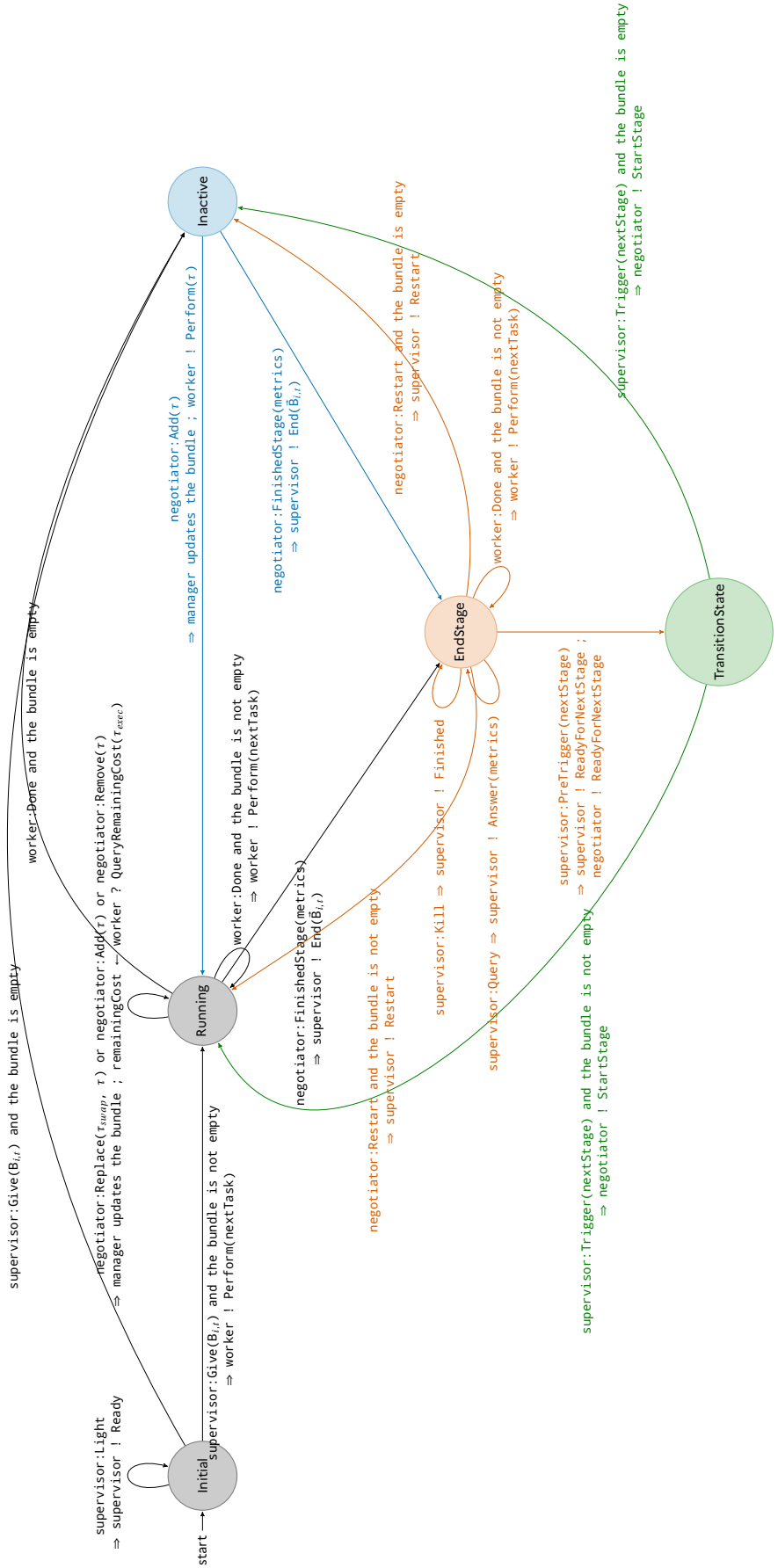


Figure 7.7 – Comportement simplifié du gestionnaire

(`negotiator:FinishedStage(metrics)`). Il passe alors dans l'état *EndStage* et informe le superviseur (`supervisor!End(`$\vec{\mathrm{B}}_{i,t}$`)`).

Il est à noter qu'à chaque mise à jour du lot de tâches, le gestionnaire envoie les nouvelles informations du lot au négociateur (`negotiator!GiveUpdatedBundle(`$\vec{\mathrm{B}}_{i,t}$`)`) et les nouveaux coûts des jobs aux pairs (`peers!Inform`). Ces messages n'apparaissent pas sur l'automate simplifié pour faciliter la lecture mais sont présents sur la version complète présentée dans l'annexe A.

Dans l'état *EndStage*, le gestionnaire attend soit :

- le message du superviseur annonçant la prochaine phase de négociation (`supervisor:PreTrigger(nextStage)`), il passe alors dans l'état *TransitionState* ;
- un message du négociateur annonçant la reprise de la phase actuelle (`negotiator:Restart`), il repasse alors dans l'état *Running* ou *Inactive* selon s'il reste des tâches à consommer ou non.

Comme dans l'état *Running*, il continue également d'interagir avec l'exécutant en lui confiant la prochaine tâche (`worker!Perform(nextTask)`), s'il y en a une, quand ce dernier a terminé la précédente (`worker:Done`). Enfin, il envoie au superviseur ses métriques (`supervisor!Answer(metrics)`) contenant les informations sur les durées de réalisation des tâches quand ce dernier le lui demande (`supervisor:Query`), et il se met en arrêt quand le superviseur l'informe de la fin du processus (`supervisor:Kill`).

La version complète de l'automate représentant le comportement du gestionnaire est disponible dans la figure A.1 de l'Annexe.

7.4.4 Négociateur

L'agent composant négociateur est chargé, en s'appuyant sur son modèle des pairs, de marchander les réallocations de tâches avec les agents négociateurs représentant les autres nœuds.

Le négociateur répond aux propositions de ses pairs et met à jour sa base de croyances, ce qui lui permet de détecter des opportunités de réallocation. Conformément aux diagrammes d'interactions 7.3 et 7.4, après avoir proposé une délégation, il attend, avant une date butoir, une acceptation, un refus ou une contre-proposition. Lorsque le négociateur a accepté une proposition ou fait une contre-proposition, il attend la confirmation ou l'abandon de son interlocuteur (dans le cas où la tâche aurait été consommée depuis). Lorsque l'agent a confirmé son acceptation d'une contre-offre, il attend également la double-confirmation de son homologue. Quand la stratégie d'offre ne suggère aucune délégation, la base de croyances est mise à jour jusqu'à ce qu'une nouvelle opportunité soit trouvée.

Le comportement simplifié du négociateur, noté ν_i, est représenté sur les figures 7.9 et 7.8. Son comportement peut être divisé en plusieurs parties. Une partie des états servent à la négociation, c'est-à-dire le « cœur de métier » de l'agent, tandis que d'autres servent à à la synchronisation des différentes phases ou à l'initialisation du processus.

La figure 7.8 présente le comportement simplifié du négociateur pour gérer les négociations ainsi que les transitions de phases.

Négociation.

Les états du négociateur dédiés à la négociation sont les suivants :

- l'état *Responder* quand le négociateur est disponible pour recevoir les propositions de délégation de ses pairs ;
- l'état *Contractor* quand le négociateur attend une confirmation ou une annulation concernant une proposition qu'il a acceptée ou à laquelle il a répondu par une contre-proposition ;
- l'état *Proposer* quand le négociateur a envoyé une proposition à un pair et qu'il attend une acceptation, une contre-proposition ou un refus ;
- l'état *Swapper* quand le négociateur a accepté une contre-proposition et qu'il attend la double confirmation de la part du répondant.

Dans l'état *Responder*, le négociateur peut se mettre lui-même en recherche d'une offre à proposer, ou répondre aux propositions des pairs. Lors de la réception d'un message `Next`, si sa stratégie d'offre ne donne pas de résultat ($\overline{\mathcal{O}_i} = \theta$), il passe dans l'état *Pause* en s'envoyant un message `Close`. Si une offre est trouvée, il l'envoie au pair concerné ($\texttt{recipient}(\mathcal{O}_i)\texttt{!Propose}(\mathcal{O}_i)$) et entre dans l'état *Proposer*. Lorsqu'il reçoit une proposition ($v_j : \texttt{Propose}(\delta(\tau, v_j, v_i, \overrightarrow{\mathsf{A}_\emptyset}))$ qui est acceptable ou pour laquelle une contre-offre est possible, il passe dans l'état *Contractor*, et répond en conséquence ($v_j\texttt{!Accept}(\delta(\tau, v_j, v_i, \overrightarrow{\mathsf{A}_t}))$ ou $v_j\texttt{!CounterPropose}(\mathcal{O}_i(v_j, \tau))$). Dans le cas contraire, il refuse l'offre ($v_j\texttt{!Reject}(\delta(\tau, v_j, v_i, \overrightarrow{\mathsf{A}_t}))$) et cherche de nouvelles opportunités ($v_i\texttt{!Next}$).

Dans l'état *Contractor*, le négociateur attend la confirmation d'une transaction qu'il a acceptée ou pour laquelle il a proposé une contre-offre. Lorsqu'il reçoit la confirmation d'une délégation ($v_j : \texttt{Confirm}(\delta(\tau, v_j, v_i, \overrightarrow{\mathsf{A}_t}))$), il demande au gestionnaire d'ajouter la ou les tâches au lot ($\texttt{manager?Add}(\tau)$). Lorsqu'il reçoit la confirmation d'un échange toujours valide ($v_j : \texttt{ConfirmSwap}(\sigma(\tau, \tau_{swap}, v_j, v_i, \overrightarrow{\mathsf{A}_t}))$), il demande au gestionnaire de remplacer la tâche qu'il cède par la tâche reçue ($\texttt{manager?Replace}(\tau_{swap}, \tau)$) et répond à son tour par un message `ConfirmSwap`. Si l'échange n'est plus valide, il répond par un message `WithdrawSwap`. Dans tous ces cas de figure, le négociateur retourne ensuite dans l'état *Responder*.

Dans l'état *Proposer*, si le négociateur reçoit une acceptation ($v_j : \texttt{Accept}(\delta(\tau, v_i, v_j, \overrightarrow{\mathsf{A}_t}))$) pour une offre toujours valide, il demande au gestionnaire de retirer la ou les tâches cédées ($\texttt{manager?Remove}(\tau)$) et confirme la délégation auprès du receveur
($v_j\texttt{!Confirm}(\delta(\tau, v_i, v_j, \overrightarrow{\mathsf{A}_t}))$). S'il reçoit une acceptation ou une contre-proposition pour une offre qui n'est plus valide (par exemple si l'une des tâches devant être cédée a été confiée à l'exécutant pour consommation), il répond par une annulation ($v_j\texttt{!Withdraw}(\delta(\tau, v_i, v_j, \overrightarrow{\mathsf{A}_\emptyset}))$). Dans ces deux cas, ainsi que lors de la réception d'un refus d'une offre ($v_j : \texttt{Reject}(\mathcal{O}_i)$), le négociateur retourne ensuite dans l'état *Responder*. Lors de la réception d'une contre-proposition acceptable pour une offre toujours valide, le négociateur demande au gestionnaire de retirer du lot la tâche cédée et répond à son pair par un message `ConfirmSwap` avant de passer dans l'état *Swapper*.

Dans l'état *Swapper*, le négociateur attend de recevoir la confirmation ou l'annulation d'un échange en cours. S'il reçoit une confirmation

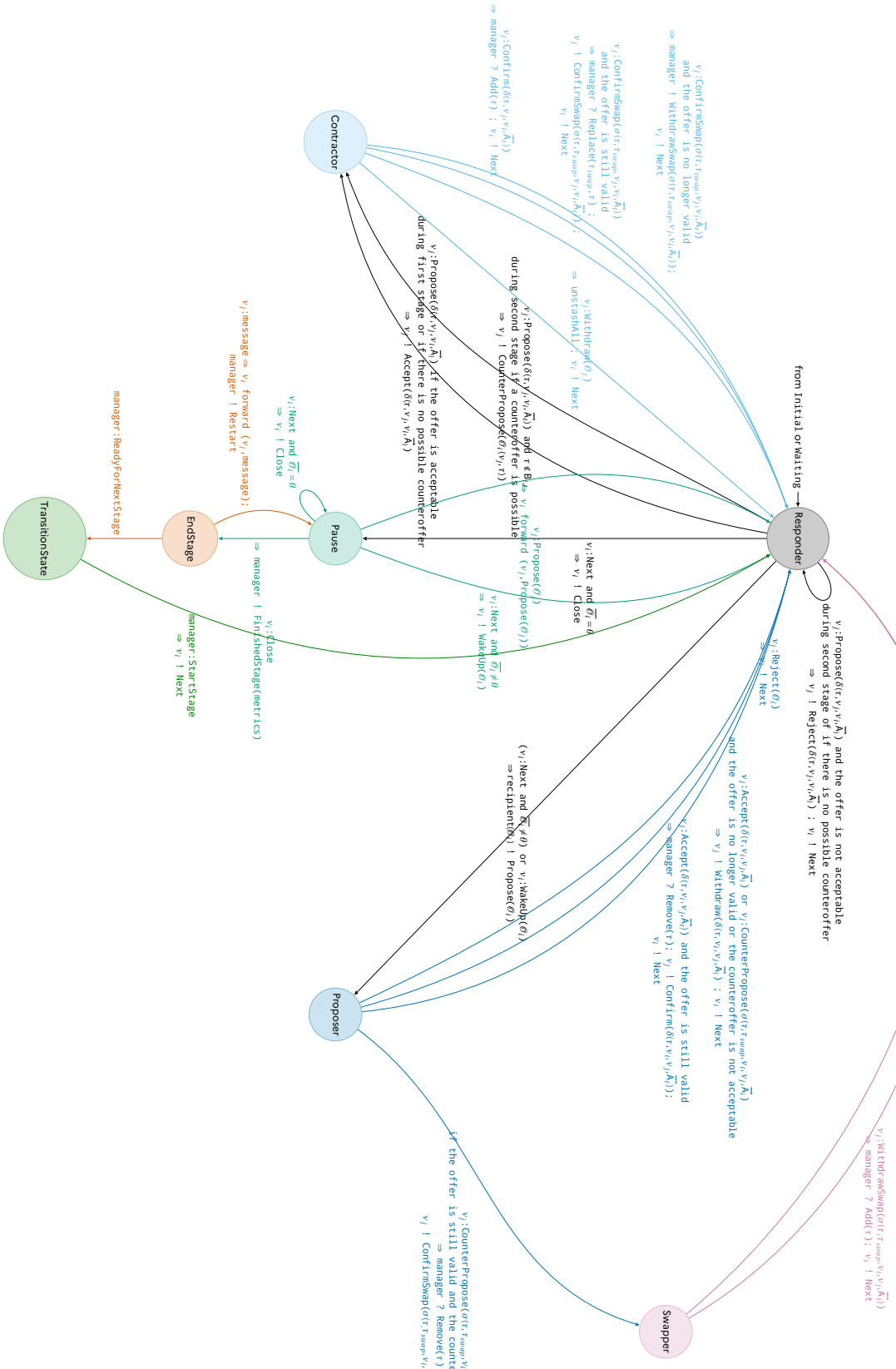


Figure 7.8 – Comportement simplifié du négociateur depuis l'état *Responder*

(v_j : ConfirmSwap($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_t})$)), il demande au gestionnaire d'ajouter dans le lot la tâche reçue (manager ? Add(τ)). S'il reçoit une annulation (v_j : WithdrawSwap($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_t})$)), il demande au gestionnaire de remettre dans le lot la tâche qu'il devait céder et qu'il a déjà retirée du lot quand il était dans l'état *Proposer*. Dans les deux cas, le gestionnaire retourne dans l'état *Responder* suite à ces opérations.

Il est à noter qu'avant d'entrer dans l'état *Responder* depuis un autre état, le négociateur s'envoie toujours un message Next pour déclencher la recherche d'une offre avec la stratégie de délégation. Ces messages n'apparaissent sur les automates simplifiés pour faciliter la lecture. De même, la réception des messages GiveUpdatedBundle($\vec{B}_{i,t}$) provenant du gestionnaire ou des messages Inform provenant des pairs ne sont pas indiqués sur les automates simplifiés mais entraînent la mise à jour de la base de croyances. Toute proposition reçue en dehors de l'état *Responder* est gardée en mémoire lors du retour à l'état correspondant. Toute acceptation ou contre-proposition reçue correspondant à une ancienne offre devenue obsolète se voit répondre par un abandon (message Withdraw).

Transition entre les phases de négociation.

Les états suivants du négociateur permette de gérer les transitions entre les phases de négociation successives :

- l'état *Pause* quand la stratégie de délégation du négociateur ne suggère plus de délégation à proposer ;
- l'état *EndStage* quand les négociations pour la phase en cours sont terminées ;
- l'état *TransitionState* quand une nouvelle phase de négociation est sur le point de commencer et que le négociateur attend le signal du superviseur.

Le négociateur se trouve dans l'état *Pause* quand il n'a plus d'offre à proposer ($\overline{\mathcal{O}_i} = \theta$) et qu'il attend que la mise à jour de sa base de croyances déclenche de nouvelles opportunités ou la fin de la phase actuelle. S'il reçoit un message Next et qu'il n'a toujours pas d'offre à proposer, il s'envoie un message Close. Si une offre est détectée, il retourne dans l'état *Responder* en s'envoyant un message WakeUp qui déclenchera l'envoi de la proposition. À la réception du message Close, le gestionnaire entre dans l'état *EndStage* en prévenant son gestionnaire (manager ! FinishedStage(metrics)).

Dans l'état *EndStage*, le négociateur attend de recevoir le signal du gestionnaire pour entrer dans l'état *TransitionState* (manager : ReadyForNextStage), ou la réception d'un autre message susceptible d'ouvrir la voie à de nouvelles opportunités. Dans ce dernier cas, il retourne dans l'état *Pause* et en informe le gestionnaire avec un message Restart.

Dans l'état *TransitionState*, le négociateur attend d'être informé du début de la phase de négociation suivante par le gestionnaire (manager : StartStage) pour retourner dans l'état *Responder* en s'envoyant un message Next pour déclencher la recherche d'une potentielle nouvelle offre.

Initialisation.

Avant d'entrer dans l'état *Responder* pour la première fois, le négociateur débute dans une phase d'initialisation, lors de laquelle lui ainsi que ses pairs attendent de recevoir les informations sur leurs lots de tâches respectifs pour pouvoir commencer les négociations. Le négociateur peut alors se trouver dans les états suivants :

- l'état *Initial* quand le négociateur attend le signal de départ du superviseur ;
- l'état *Waiting* quand le négociateur a reçu le signal de départ mais que ses pairs ne sont pas encore prêts.

La figure 7.9 présente le comportement simplifié du négociateur depuis ces deux états.

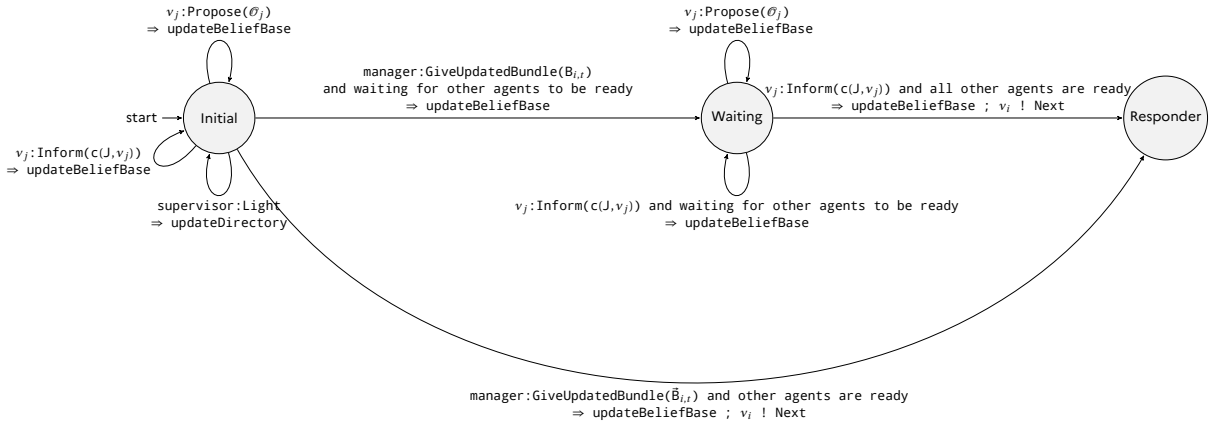


Figure 7.9 – Comportement simplifié du négociateur depuis les états *Initial* et *Waiting*

Le négociateur est au début dans l'état *Initial*. À l'origine du processus, il reçoit un message du superviseur l'informant de sa liste d'accointances (`supervisor!Light`). Lorsqu'il reçoit les informations concernant le lot de tâches de la part du gestionnaire (`manager!GiveUpdatedBundle`), il passe dans l'état `Responder` si tous les pairs ont déjà reçu le leur également, ou passe dans l'état *Waiting* dans le cas contraire. Une fois dans l'état *Waiting*, il passe dans l'état *Responder* quand tous les autres agents sont prêts également, c'est-à-dire quand il a reçu les informations sur les coûts des jobs dans leurs propres lots (v_j : `Inform(c(J,`v_j`))`) pour tous ses pairs v_j. Dans l'état *Waiting* tout comme dans l'état *Initial*, le négociateur peut recevoir des messages `Inform` de la part de ses pairs lui permettant de mettre à jour sa base de croyances, ainsi que des propositions de délégation (v_j : `Propose(`$\mathcal{O}_j$`)`) qui sont gardées dans la file d'attente de messages en attendant d'être traitées dans l'état *Responder* et qui entraînent également la mise à jour de la base de croyances.

La version complète des automates représentant le comportement du négociateur est disponible dans les figures A.2 et A.3 de l'Annexe.

7.5 Implémentation

Le prototype `SMASTA+` (Beauprez, Caron & Morge, 2020) est implémenté avec le langage de programmation `Scala` et la bibliothèque `Akka` (Lightbend, 2020), qui implémente le modèle d'acteur (Hewitt, 1977), adaptée aux applications orientées messages,

fortement concurrentes, distribuées et robustes. À la différence de l'implementation proposée par Baert (2019), la spécification des comportements par les automates finis est directe au travers du patron de conception FSM (*Finite State Machine*).

Je suppose ici que :

1. le délai de transmission des messages est arbitraire mais non négligeable ;
2. l'ordre des messages par paire émetteur/récepteur est préservé ;
3. la distribution des messages est garantie.

Chaque agent-nœud est représenté par un objet `NodeAgent` qui est une sous-classe de `Actor`. Chaque acteur `NodeAgent` est parent de trois sous-acteurs `WorkerBehaviour`, `ManagerBehaviour` et `NegotiatorBehaviour`, représentant les comportements des trois agents composants. Chacun de ces trois sous-acteurs est implémenté en un acteur FSM (pour *Finite State Machine*) et nécessite la définition des différents états possibles pour l'acteur et d'un état mental (*mind*). Les acteurs sont instanciés dans un `ActorSystem`, représentant le système multi-agents.

Pour illustrer l'implémentation directe du modèle d'agent, le comportement de l'exécutant décrit par l'automate 7.6 est présenté dans l'extrait de code 7.10. Les différents états qu'il peut prendre sont définis dans `WorkerState` (figure 7.11). La classe `WorkerMind` définit l'état mental. L'agent exécutant est initialement dans l'état *Free* (ligne 5). Le comportement relatif à chaque état est traité dans un bloc `when(State)`. Pour chaque état, les événements possibles comme la réception de messages sont traités à l'aide de `case Event(message, mind)`. À la fin du traitement d'un événement, il est spécifié si l'agent change d'état (`goto(nextState)`) ou non (`stay()`) ainsi que son nouvel état mental. Par exemple, dans l'état *Free* (lignes 7 à 11), l'exécutant peut recevoir un message `Perform` (ligne 7) et se rend alors dans l'état *Busy* (ligne 10). L'envoi de message s'écrit avec l'opérateur « ! », de la même manière que dans les automates, comme par exemple aux lignes 17 et 22.
Comme le montre cet exemple, l'élégance de l'implémentation réside dans sa correspondance directe avec l'automate, ce dernier se traduisant de façon immédiate avec le patron de conception utilisé.

7.6 Synthèse

Dans ce chapitre, j'ai proposé une architecture d'agents permettant la mise en œuvre des mécanismes étudiés dans les chapitre 5 et 6. Ainsi, pour pouvoir réaliser simultanément réallocations et consommations, un agent-nœud est représenté comme un agent composite constitué de trois agents composants, selon le principe de séparation des rôles. En résumé, un agent-nœud est constitué de trois agents composants :

- un exécutant qui consomme les tâches ;
- un négociateur qui marchande les réallocations en s'appuyant sur son modèle des pairs (cf. section 6.2) et sur les stratégies de négociation proposée dans la section 6.6 ;
- un gestionnaire qui met à jour le lot de tâches en fonction des transactions effectuées par le négociateur et qui choisit quelle tâche donner à l'exécutant selon une des stratégies de consommation introduites dans la section 6.4.

```
class WorkerBehaviour(stap: STAP, node: ComputingNode)
  extends Worker(stap, node) with FSM[WorkerState, WorkerMind]
  with Stash {

  startWith(Free, new WorkerMind(node, stap))

  when(Free) {
    case Event(Perform(task), mind)  =>
      startConsumeTask(task)
      goto(Busy) using mind
  }


  when(Busy) {
    case Event(QueryRemainingWork(), mind)  =>
      val c = stap.cost(task, node) - taskProgression(task)
      manager ! RemainingCost(c)
      stay() using mind


    case Event(TaskDone(), mind) =>
      manager ! Done
      goto(Free) using mind
  }

}
```

Figure 7.10 – Extrait de code SMASTA+ du comportement de l'exécutant représenté par l'automate 7.6

```
trait WorkerState extends State
case object Free extends WorkerState
case object Busy extends WorkerState
```

Figure 7.11 – Extrait de code SMASTA+ définissant tous les états de l'exécutant

Les réallocations se déroulent en alternant deux phases de négociation jusqu'à ce que toutes les tâches soient consommées :

- lors de la première phase, les négociateurs réallouent les tâches en proposant des délégations n-aires socialement rationnelles, sans contre-partie ;
- lors de la deuxième phase, les négociateurs proposent des délégations unaires, même si elles se sont pas rationnelles, susceptibles d'aboutir en un échange socialement rationnel.

Ces phases sont synchronisées par un agent superviseur.

Les comportements des agents composants et du superviseur sont spécifiés sous la forme d'automates à états finis. La difficulté réside dans la complexité de ces compor-

tements, comme en attestent les versions complètes des automates présentées en annexe A et le tableau 7.1, reprenant les nombres d'états et de transitions entre états pour chaque type d'agent, ainsi que le nombre de lignes de code correspondantes dans l'implémentation. Cette complexité est importante notamment pour la spécification du comportement du négociateur. Ce dernier applique des stratégies de négociation qui sont complexes, en raison du nombre élévé d'interactions possibles et des événements disruptifs pouvant survenir. D'une part la programmation asynchrone rend l'ordre de réception des messages non déterministe. D'autre part, la concurrence entre l'exécutation des tâches et leur réallocation par négociation nécessite un mécanisme de double acquittement pour la réalisation des échanges. Enfin, les changements de phase de négociation nécessitent la mise en place d'un mécanisme de transition, permettant de s'assurer que la phase en cours est bien terminée ou de redémarrer les négociations dans le cas où une nouvelle opportunité survient.

L'architecture choisie se basant sur la séparation des rôles permet de réduire cette complexité, comparativement au cas où un seul agent aurait à gérer simultanément l'exécution des tâches et la négociation des réallocations.

Agent	Nombre d'états	Nombre de transitions	Nombre de lignes
Exécutant	2	7	173
Gestionnaire	5	23	465
Négociateur	9	74	1306
Superviseur	9	69	626

Table 7.1 – Complexité des comportements

Chapitre 8

Expérimentations

Sommaire

8.1 Introduction 97
8.2 Prototype 98
8.3 Réallocation instantanée 98
8.3.1 Évaluation de la stratégie de contre-offre 98
8.3.2 Évaluation de la stratégie de délégation n-aire 100
8.3.3 Comparaison avec les méthodes d'enchères de référence 100
8.3.4 Comparaison avec une méthode DCOP 106
8.4 Réallocation lors de la consommation 108
8.4.1 La stratégie de réallocation améliore le *flowtime*. 110
8.4.2 La stratégie de réallocation ne pénalise pas la consommation. 111
8.4.3 La stratégie de réallocation s'adapte aux aléas d'exécution. 111
8.4.4 La stratégie de réallocation s'adapte à la libération des jobs. 113
8.5 Synthèse 114

8.1 Introduction

Je décris dans ce chapitre les campagnes d'expérimentation pour l'évaluation empirique de ma stratégie multi-agents de réallocation.

Dans un premier temps, en considérant un problème d'allocation instantané, je montre que ma méthode de réallocation, indépendamment du processus de consommation, atteint une durée de réalisation proche de celle obtenue par les méthodes de référence et réduit significativement le temps de réordonnancement.

Dans un second temps, en considérant le problème d'allocation continue, je montre que lorsqu'elle est exécutée de manière concurrente au processus de consommation, ma stratégie multi-agents ne pénalise pas la consommation contrairement aux méthodes de référence fondées sur les enchères. Ma méthode permet d'améliorer la durée moyenne de réalisation jusqu'à 29 %. La stratégie multi-agents s'adapte à la libération de jobs et aux aléas d'exécution (comme le ralentissement de nœuds) bien que les agents aient une connaissance imparfaite de l'environnement d'exécution.

8.2 Prototype

Comme mentionné dans la section 7.5 du chapitre 7, le prototype SMASTA+ (Beauprez, Caron & Morge, 2020) est implémenté avec le langage de programmation Akka. Pour rappel, je suppose que :

- le modèle de communication entre agents est asynchrone, le délai de transmission des messages est arbitraire mais non négligeable ;
- l'ordre des messages par paire émetteur-récepteur est préservé, donc si un agent envoie deux messages à un autre agent, alors le premier message est reçu avant le second ;
- la réception en temps fini des messages émis est garantie. Cette hypothèse simplifie considérablement la conception des protocoles d'interactions qui, contrairement à Baert (2019), ne nécessitent pas d'introduire des accusés de réception autres que ceux nécessaires à la logique métier.

Les expériences ont été réalisées sur la plateforme data de l'Université de Lille, sur une lame de la grappe de calcul munie de 20 cœurs Intel(R) Xeon(R) Silver avec 512Go de RAM.

Afin de déterminer le coût d'une tâche, je considère qu'une ressource distante au nœud est deux fois plus onéreuse à traiter qu'une ressource locale car ces données doivent être au préalable rapatriées. En d'autres termes, j'ai fixé $\kappa = 2$ dans l'équation 5.3. Cette valeur a été fixée empiriquement lors d'une campagne d'expérimentation précédente restituée par Baert (2019) dans la même configuration matérielle. Il est à noter que cette valeur n'est cependant pas vérifiée pour des clusters au sein desquels les nœuds seraient éloignés géographiquement.

8.3 Réallocation instantanée

Cette section est une première étape dans l'évaluation de la stratégie multi-agents, puisque je compare ici le calcul d'une réallocation, c'est-à-dire la résolution d'un problème STAP à l'instant t (cf. définition 5.2). Même si la stratégie de consommation est nécessaire pour trier les lots de tâches de chaque agent, je ne considère pas le processus de consommation.

Je compare ici mes différents types de stratégies de négociation. Dans un premier temps j'évalue la valeur ajoutée de la stratégie de contre-offre (cf. section 8.3.1) et de la stratégie de délégation n-aire (cf. section 8.3.2). Je compare ensuite ma méthode de réallocation avec des méthodes orientées enchères (cf. section 8.3.3) et avec une méthode d'optimisation distribuée sous contrainte (cf. 8.3.4).

8.3.1 Évaluation de la stratégie de contre-offre

J'évalue ici l'apport de la stratégie de contre-offre décrite dans la section 6.6.3 permettant aux agents de réaliser des échanges, comparativement à la stratégie d'acceptation présentée dans la section 6.6.2.

Objectif. La stratégie de contre-offre permet d'améliorer le *flowtime* moyen, au prix d'un surcoût computationnel raisonnable. Afin de vérifier cette hypothèse, je compare ici les performances de la stratégie d'acceptation décrite dans la section 6.6.2 avec celles de la stratégie de contre-offre décrite dans la section 6.6.3.

Protocole expérimental. Je compare, pour la résolution d'un problème d'allocation et à partir d'allocations initiales aléatoires, le temps de réordonnancement et la durée moyenne de réalisation de ma méthode multi-agents dans deux configurations différentes :

1. Stratégie sans échange : dans un processus de négociation à une phase, les agents adoptent une stratégie de délégation unaire pour émettre des offres (cf. section 6.6.1.1) et la stratégie d'acceptation (cf. section 6.6.2) pour y répondre ;
2. Stratégie avec échanges : dans un processus de négociation à deux phases, tel que présenté dans la section 7.4.1, où la première phase est identique à celle ci-dessus, et où lors de la deuxième, les agents adoptent la même stratégie de délégation pour émettre des offres et la stratégie de contre-offre 6.6.3 pour y répondre.

Les résultats sont également comparés avec une méthode d'escalade (cf. section 3.2.1) qui part d'une allocation initiale aléatoire et qui, à chaque étape, sélectionne parmi tous les échanges possibles celui qui minimise la durée de réalisation moyenne.

Je considère des instances de STAP avec $m \in [2;12]$ agents-nœuds, $\ell = 4$ jobs, $n = 3 \times \ell \times m$ tâches et 10 ressources par tâche. Chaque ressource ρ_i est répliquée 3 fois. La taille de chacune des ressources est générée indépendamment de manière pseudo-aléatoire dans l'intervalle $[0, 100]$ selon une distribution de loi uniforme ($|\rho_i| \in [0;100]$). Je génère aléatoirement 10 instances de STAP, et pour chacune sont générées aléatoirement 10 allocations initiales.

Par la suite, et par souci de simplification, je parlerai de *flowtime* moyen pour désigner la durée moyenne de réalisation (cf. 5.9).

Résultats. Les figures 8.1a et 8.1b comparent respectivement la durée de réalisation et le temps de réordonnancement de la stratégie sans échange avec ceux de la stratégie de délégation avec échanges ainsi qu'avec la méthode d'escalade. Les trois méthodes débutent avec la même allocation initiale générée aléatoirement.

La figure 8.2 représente, pour une exécution quelconque avec 16 nœuds, l'évolution du *flowtime* moyen lors du processus incluant une phase d'échanges. la stratégie de délégation opère 87 délégations (où une délégation désigne une offre ayant aboutie) dans la phase 1. La stratégie de contre-offre poursuit le processus de négociation en opérant 32 échanges dans la phase 2 qui, à leur tour, permettent de déclencher 23 délégations puis 14 échanges.

Analyse. Nous observons sur la figure 8.1a que la stratégie avec échanges atteint des solutions de qualité similaire aux solutions obtenues par la méthode d'escalade et, comme attendu, légèrement meilleure que la stratégie sans échange. On observe également que, si le temps de réordonnancement (cf. figure 8.1b) de la stratégie avec échanges est légèrement moins bon que pour la stratégie sans échange, il reste approximativement identique et donc largement meilleur que celui de la méthode d'escalade

qui croît exponentiellement avec le nombre de nœuds, c'est pourquoi la méthode d'escalade n'est testée que jusqu'à 8 nœuds ici.

Grâce au processus de négociation en deux phases, la stratégie d'échange prolonge la stratégie de délégation en atteignant une allocation similaire avec le même temps de réordonnancement tout en permettant de l'améliorer par la suite.

8.3.2 Évaluation de la stratégie de délégation n-aire

Contrairement à la stratégie de délégation unaire décrite dans la section 6.6.1.1, la stratégie de délégation n-aire décrite dans la section 6.6.1.2 propose des offres constituées non pas d'une unique tâche mais d'un lot d'offre.

Hypothèse. La stratégie de délégation n-aire permet de réduire le temps de réordonnancement sans dégrader significativement le *flowtime* moyen.

Protocole expérimental. Pour comparer ces deux stratégies, je considère des instances de STAP avec $m \in [2;12]$ agents-nœuds, $\ell = 4$ jobs, $n = 3 \times \ell \times m$ tâches et 10 ressources par tâche. Chaque ressource ρ_i est répliquée 3 fois. La taille de chacune des ressources est générée indépendamment de manière pseudo-aléatoire dans l'intervalle $[0, 100]$ selon une distribution de loi uniforme ($|\rho_i| \in [0;100]$). Je génère aléatoirement 10 instances de STAP, et pour chacune sont générées aléatoirement 10 allocations initiales.

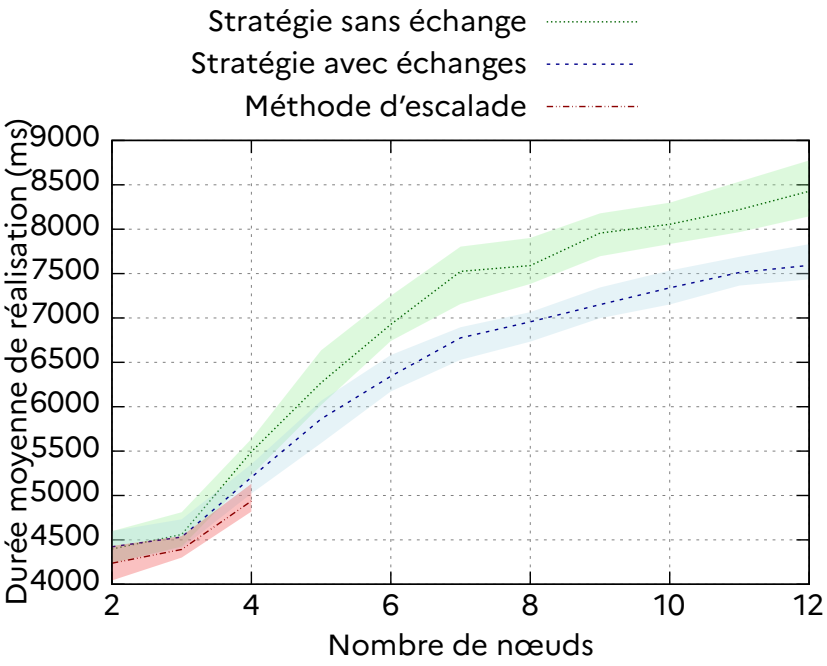
Résultats. Les figures 8.3a et 8.3b comparent respectivement la moyenne et l'écart-type caractéristique du *flowtime* moyen et du temps de réordonnancement de la stratégie de délégation unaire avec celle de la stratégie de délégation n-aire. On observe que, si le *flowtime* moyen atteint par la stratégie de délégation n-aire est légèrement moins bonne d'environ 600 millisecondes par rapport à la stratégie de délégation unaire, le gain en terme de temps de réordonnancement est en revanche très avantageux. En effet, la sélection d'une délégation n-aire réduit également le temps de réordonnancement, ici $0,35$ seconde plutôt que $1,45$ secondes pour 12 nœuds.

La figure 8.4 représente l'évolution de la durée moyenne de réalisation de ces deux stratégies d'offre pour une instance quelconque non équilibrée de problème de réallocation à 16 nœuds. Nous observons sur la figure 8.4a que la sélection d'une délégation n-aire réduit le nombre de délégations (ici 40 plutôt que 66) ainsi que le temps d'exécution sans détériorer de façon significative le *flowtime* moyen.

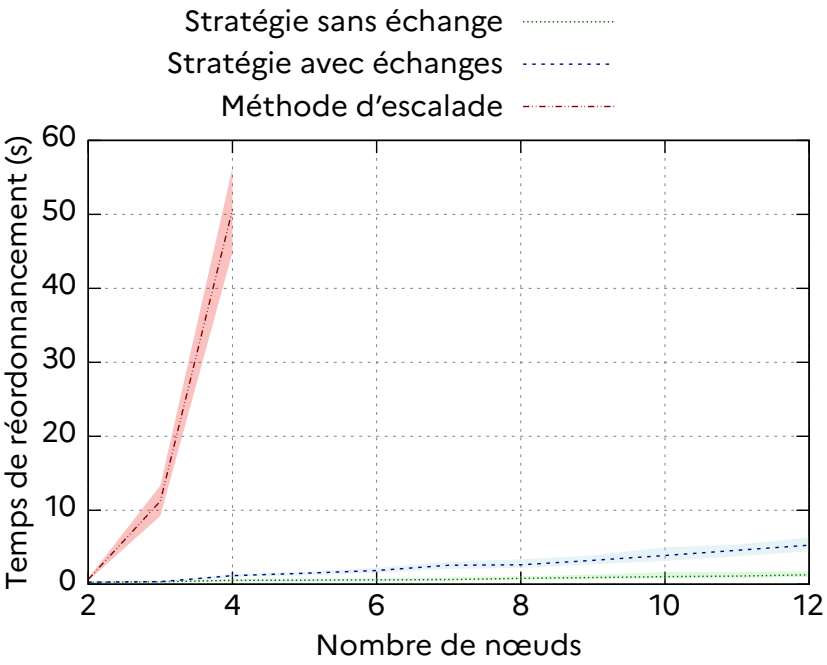
Analyse. La stratégie de délégation n-aire atteint une allocation dont le *flowtime* moyen est légèrement moins bon que celui atteint par la stratégie de délégation unaire. En effet, la délégation de lots peut conduire à un minimum local pour le *flowtime* moyen. Toutefois, la stratégie de délégation n-aire permet d'atteindre une allocation stable plus rapidement et donc réduire le temps de réordonnancement.

8.3.3 Comparaison avec les méthodes d'enchères de référence

Dans la suite des expérimentations, et compte-tenu des résultats décrits dans les sections 8.3.1 et 8.3.2, ma méthode de réallocation utilise un processus de négociation à deux phases, où la première phase se base sur la stratégie de délégation n-aire et où



(a) Durée moyenne de réalisation



(b) Temps de réordonnancement

Figure 8.1 – Comparaison des stratégies sans et avec échanges

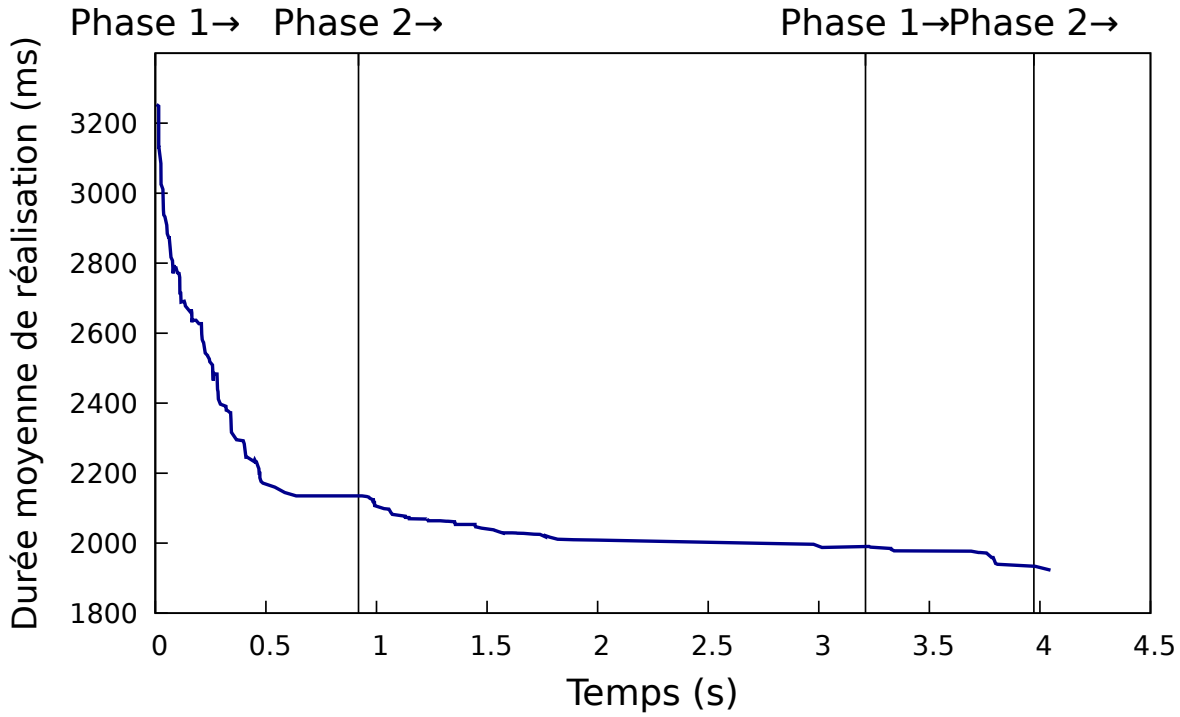


Figure 8.2 – *Flowtime* moyen au cours du temps lors de l'exécution de la stratégie avec échanges sur une instance caractéristique

la deuxième utilise la stratégie avec échanges.

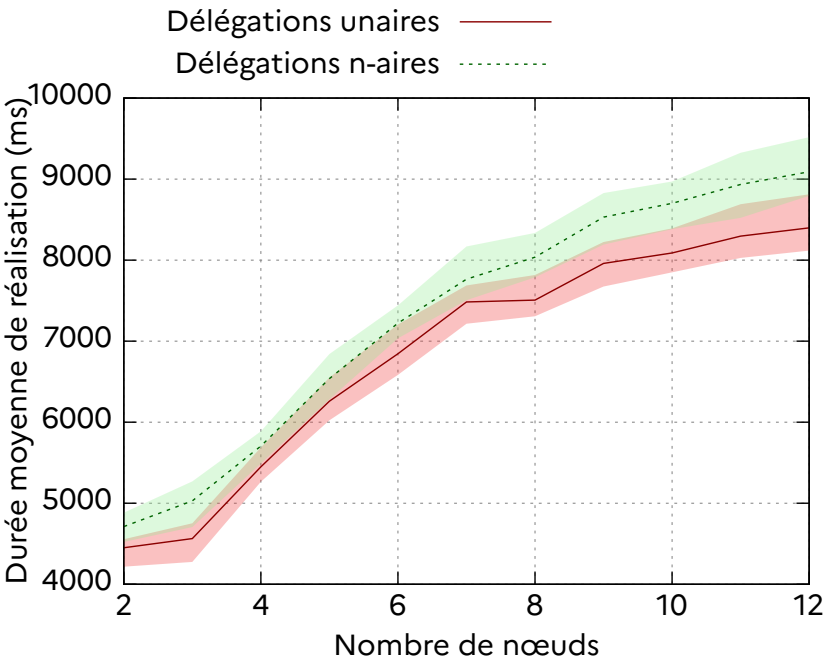
Hypothèse. La stratégie de réallocation réduit significativement le temps de réordonnancement, en la comparant avec des méthodes multi-agents d'enchères de référence.

Protocole expérimental. Afin d'évaluer le processus de réallocation indépendamment du processus de consommation des taches, je considère $m \in [2;16]$ nœuds, $\ell = 4$ jobs, $n = 12 \times m$ tâches avec 10 ressources par tâche. Chaque ressource ρ_i est répliquée 3 fois. La taille de chacune des ressources est générée indépendamment de manière pseudo-aléatoire dans l'intervalle $[0,100]$ selon une distribution de loi uniforme ($|\rho_i| \in [0;100]$).

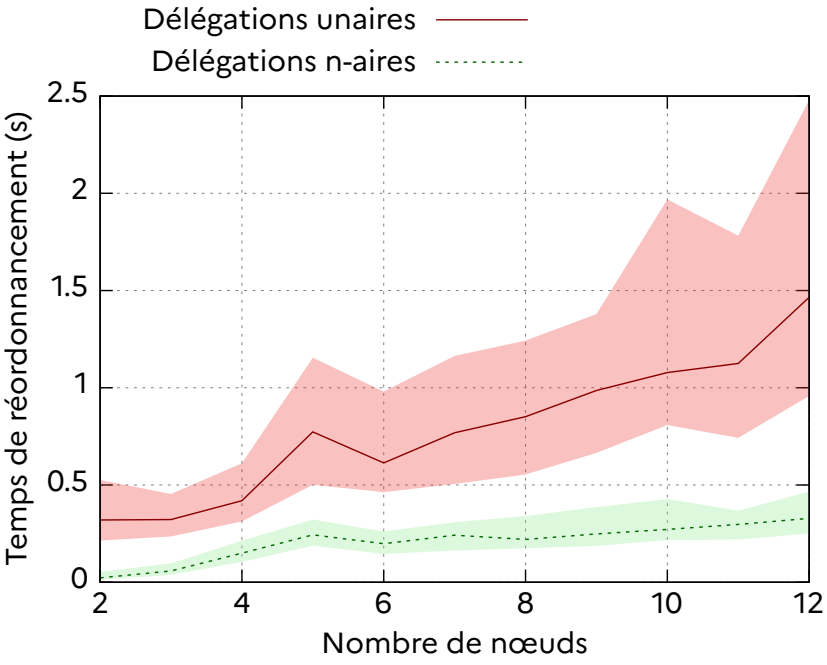
Je génère 10 instances de STAP, et pour chacune sont générées aléatoirement 10 allocations initiales. J'évalue les médianes et les écarts types de trois métriques : la durée de réalisation (équation 5.10), le temps de réordonnancement et le taux de disponibilité locale qui mesure la proportion des ressources traitées localement.

Définition 8.1 (Taux de disponibilité locale).
Soient STAP un problème et $\overrightarrow{A_t}$ une allocation à l'instant t. le taux de disponibilité locale de $\overrightarrow{A_t}$ est :

$$L(\overrightarrow{A_t}) = \Sigma_{\tau \in \mathcal{T}} \frac{\Sigma_{\rho \in \mathcal{R}_\tau,\ local(\rho, v(\tau, \overrightarrow{A_t}))} |\rho|}{\Sigma_{\rho \in \mathcal{R}_\tau} |\rho|} \tag{8.1}$$

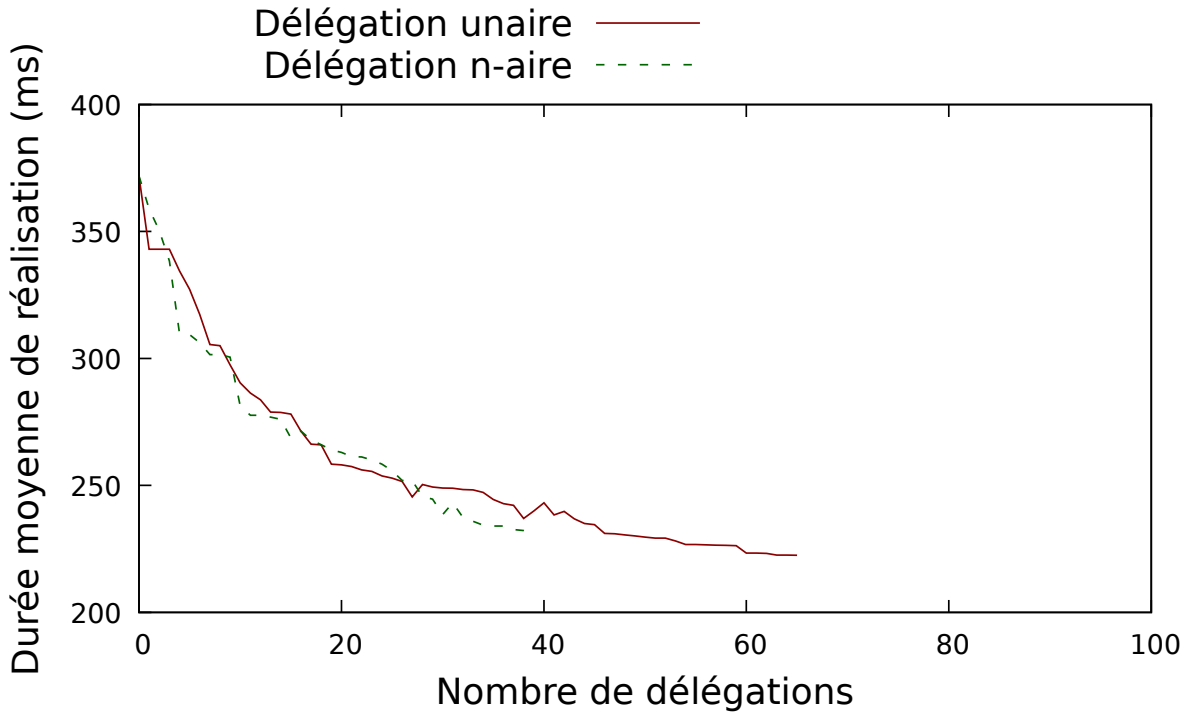


(a) Durée moyenne de réalisation

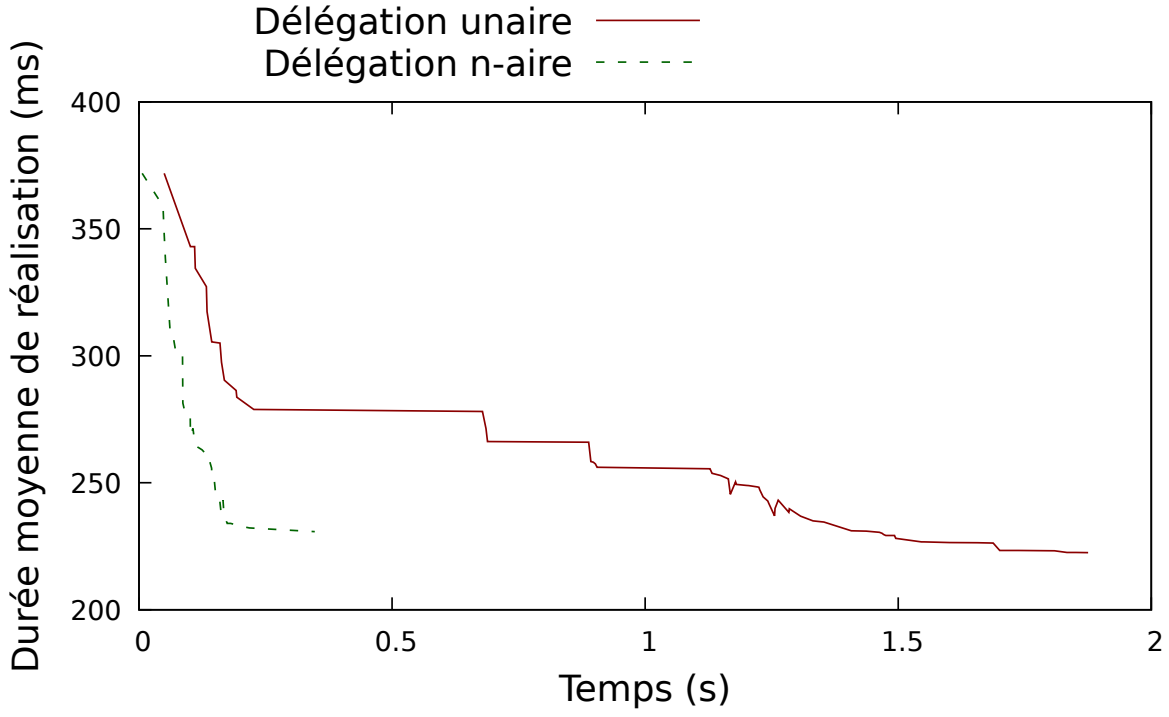


(b) Temps de réordonnancement

Figure 8.3 – Comparaison des stratégies de délégation unaire et n-aire



(a) *Flowtime* moyen de réalisation des stratégies de délégation unaire et n-aire au cours des délégations



(b) *Flowtime* moyen de la stratégie de délégation unaire et n-aire au cours du temps

Figure 8.4 – Évolution du *flowtime* moyen pour une instance caractéristique non équilibrée avec 16 agents-nœuds

Plus le taux de disponibilité locale d'une allocation est élevé, plus le volume de ressources distantes nécessaires à l'exécution des tâches est faible.

Je compare les résultats obtenus avec ceux de méthodes d'enchères de références telles que :

- une méthode d'allocation fondée sur des enchères séquentielles à un seul article (ASSI), où une offre correspond au *flowtime* si la tâche est attribuée à l'enchérisseur ;
- une méthode de réallocation fondée sur des enchères séquentielles à un seul article (RSSI) exécutée de manière concurrente au processus de consommation où les agent-noeuds, chacun à leur tour, proposent de déléguer une de leurs tâches en attente.

Ces deux méthodes sont des adaptations pour notre problème de la méthode SSI (Koenig et al., 2006) présentée dans le chapitre 3.

Résultats. La figure 8.5 montre les médianes et les écarts types du *flowtime* moyen, du temps d'ordonnancement et du taux de disponibilité locale en fonction du nombre de nœuds. Nous pouvons observer que ma stratégie de réallocation améliore le *flowtime* moyen de l'allocation initiale mais qu'il est supérieur à celui atteint par les méthodes fondées sur des enchères séquentielles (cf. figure 8.5a). La méthode d'allocation, qui considère le meilleur nœud pour chaque tâche mise aux enchères, permet d'obtenir un *flowtime* moyen meilleur que celui obtenu par les méthodes de réallocation qui sont pénalisées par l'allocation initiale aléatoire à rééquilibrer. Malgré le fait que certaines délégations, susceptibles de réduire le *flowtime* moyen, ne sont pas prises en compte par le processus de négociation, ma stratégie multi-agents s'avère efficace : le *flowtime* moyen atteint par ma stratégie est meilleur que celui de l'allocation initiale.
Bien que le taux de disponibilité soit inférieur à celui atteint par la méthode d'allocation ASSI (cf. figure 8.5c), il reste similaire à celui obtenu par la méthode de réallocation basée sur des enchères RSSI.
Comparativement aux méthodes basées sur des enchères qui évaluent toutes les délégations possibles, notre stratégie multi-agents présente un temps d'ordonnancement nettement inférieur (cf. figure 8.5b). Par exemple, ce temps est mille fois plus court pour 8 nœuds. Il est intéressant de noter que l'écart entre le temps de réordonnancement entre une méthode fondées sur des enchères et ma stratégie de négociation croît exponentiellement avec le nombre de nœuds, tandis que l'écart pour le *flowtime* moyen reste globalement constant.

Analyse. Ma stratégie d'offre sélectionne efficacement les tâches distantes dont la délégation réduit le coût, dans le but d'améliorer le taux de disponibilité locale. Bien que ce taux soit inférieur à celui atteint par la méthode d'allocation, il reste similaire à celui obtenu par la méthode de réallocation basée sur des enchères.

Ma stratégie aboutit à un *flowtime* moyen supérieur à celui obtenu par les méthodes d'enchère. Elle permet toutefois de diminuer le *flowtime* moyen de l'allocation initiale en sélectionnant efficacement les tâches distantes dont la délégation réduit le coût afin d'améliorer le taux de disponibilité locale, avec un temps de réordonnancement nettement meilleur. Ainsi, même lorsque le nombre de nœuds est limité, le bénéfice obtenu sur le *flowtime* moyen grâce aux enchères séquentielles est contrebalancé et annulé par le coût supplémentaire lié au temps de réordonnancement. Or, comme cela sera

abordé dans la section 8.4, ce gain en terme de temps de réordonnancement permet de mieux s'adapter à la consommation.

8.3.4 Comparaison avec une méthode DCOP

Hypothèse. Le temps de réordonnancement de notre stratégie de négociation est plus faible que celui obtenu par des techniques d'optimisation distribuée sous contraintes (DCOP) présentées dans la section 3.3.3 et qu'elle permet d'atteindre des durées moyennes de réalisation plus faibles.

Méthode de référence. Le problème qui nous concerne, à savoir trouver l'allocation optimale qui minimise le *flowtime* moyen de ℓ jobs constitués de n tâches sur m nœuds indépendants, peut être modélisé mathématiquement par :

1. n variables de décision x_i telles que,

$$x_i = (o-1) \times n + k \text{ si } \tau_i \text{ est la } k^{ieme} \text{ tâche à partir de la fin dans } \vec{\mathsf{B}}_{o,t} \tag{8.2}$$

2. n^2 contraintes pour que chaque tâche soit affectée à une unique position

$$\forall i \in [1,n] \forall j \in [1,n] \setminus \{i\} \; x_i \neq x_j; \tag{8.3}$$

3. la fonction objectif à minimiser est $\mathsf{C}(\overrightarrow{\mathsf{A}_t})$.

$$\begin{aligned} \mathsf{C}(\overrightarrow{\mathsf{A}_t}) &= \Sigma_{\mathsf{J} \in \mathcal{J}_t} \mathsf{C}_{\mathsf{J}}(\overrightarrow{\mathsf{A}_t}) \\ &= \Sigma_{\mathsf{J} \in \mathcal{J}_t} \max_{v_i \in \mathcal{N}} C_{last(v_i,\mathsf{J})}(\overrightarrow{\mathsf{A}_t}) \end{aligned} \tag{8.4}$$

où $last(v_i, \mathsf{J})$ est la dernière tâche du job J dans le lot de tâches $\vec{\mathsf{B}}_{i,t}$:

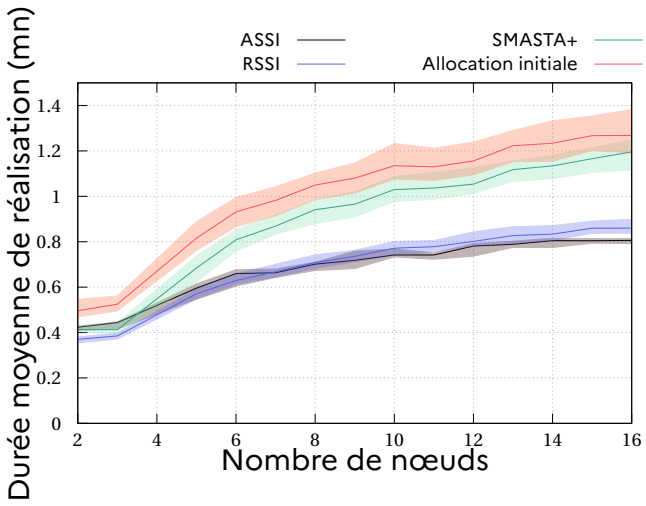
$$last(v_i, \mathsf{J}) = argmax_{\tau_k \in \mathsf{J} \cap \mathsf{B}_{i,t}} \{card(\{\tau \in \vec{\mathsf{B}}_{i,t} \mid \tau \prec_i \tau_k\})\} \tag{8.5}$$

Pour résoudre ce problème, je considère l'algorithme MGM-2 (*Maximum Gain Message*) proposé par Maheswaran, Pearce et Tambe (2004), décrit dans la section 3.3.3, comme le plus adapté car c'est un algorithme distribué de recherche locale qui est approché et asynchrone. J'ai utilisé la bibliothèque pyDCOP (Rust & Picard, 2021; Rust, Picard & Ramparany, 2018).

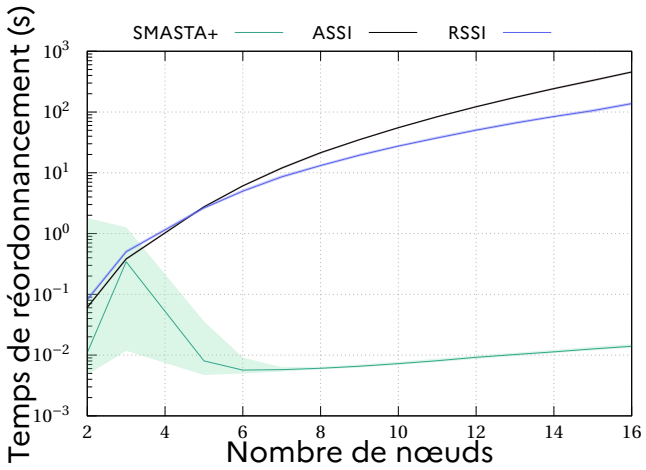
Protocole expérimental. Je considère 100 problèmes de réallocation pour des instances de STAP avec $m = 2$ nœuds, $\ell = 4$ jobs et $n = 3 \times \ell \times m = 24$ tâches, avec une ressource par tâche et où chaque ressource est répliquée 3 fois.

Résultats. La figure 8.6 compare les durées moyennes de réalisation pour des réallocations obtenues par notre stratégie en 0.4 millisecondes (en moyenne) avec l'algorithme MGM2 dont le *timeout* est respectivement 2, 5 et 10 secondes. On peut noter que l'algorithme MGM-2 propose systématiquement une réallocation quand le *timeout* est de 5 ou 10 secondes mais jamais avec un *timeout* de 2 secondes. Dans ce cas, je considère que l'allocation initiale générée de manière pseudo-aléatoire est conservée.

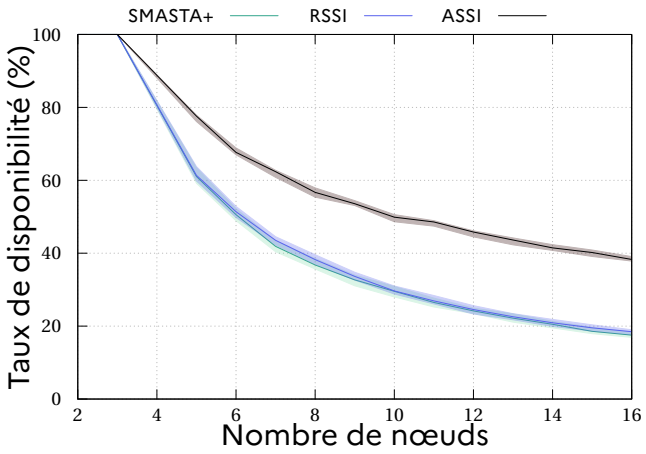
Analyse. Au-delà des temps de réordonnancement qui peuvent s'expliquer par le fait que MGM2 est implémenté en Python et que la stratégie s'exécute sur la machine virtuelle Java (Fulgham & Gouy, 2021), les expériences montrent que même si le *timeout*



(a) Durée moyenne de réalisation



(b) Temps de réordonnancement selon une échelle de temps logarithmique



(c) Taux de disponibilitié

Figure 8.5 – Réallocation instantanée

est de 5 secondes, l'algorithme MGM-2 retourne une réallocation où la durée moyenne de réalisation est supérieure à celle obtenue par notre stratégie. Accroître le *timeout* ne permet pas d'améliorer la durée moyenne de réalisation des allocations retournées par MGM-2. Notons que l'algorithme MGM-2 ne propose jamais de réallocation avec $m = 3$ nœuds, $\ell = 5$ jobs et $n = 3 \times \ell \times m = 45$ tâches quand le *timeout* est de 15 minutes. Cet algorithme ne passe donc pas l'échelle sur ce type de problème.

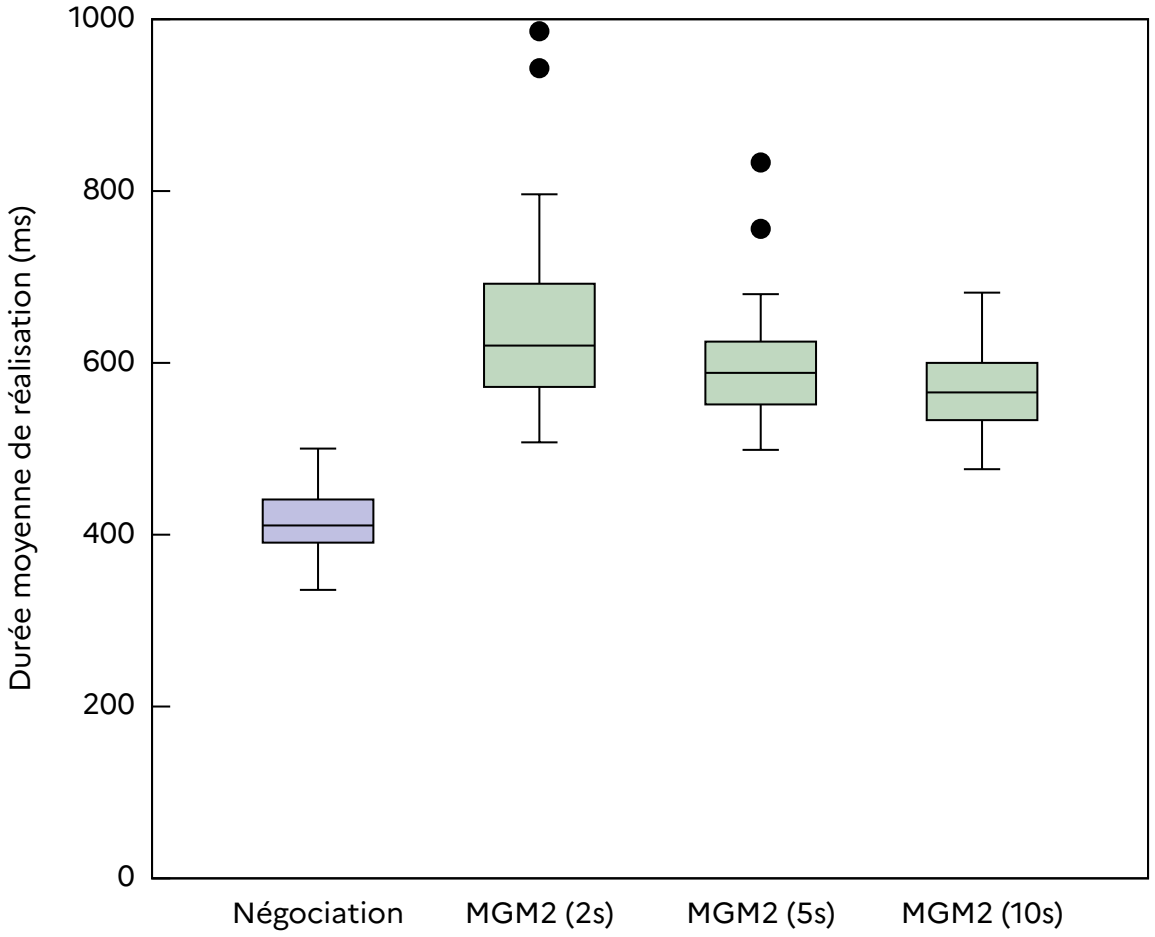


Figure 8.6 – Diagrammes en boîte à moustaches des durées moyennes de réalisation pour les réallocations obtenues par notre stratégie en 0.4 millisecondes (en moyenne) avec l'algorithme MGM-2 dont le *timeout* est respectivement 2, 5 et 10 secondes

8.4 Réallocation lors de la consommation

Les expériences présentées dans cette section visent à valider et évaluer l'architecture d'agents présentée dans le chapitre 7 lors de l'exécution concurrente des processus de consommation et de réallocation.

Les expérimentations de la section précédente ont permis de raffiner ma stratégie de négociation. Ainsi, dans cette section je conserve la stratégie définie dans la section 8.3.3, à savoir un processus de négociation à deux phases, où la première phase

se base sur la stratégie de délégation n-aire et où la deuxième utilise la stratégie avec échanges.

Hypothèses. Les expérimentations sont menées de façon à vérifier quatre hypothèses différentes :

1. la stratégie de réallocation permet d'améliorer le *flowtime* ;
2. la stratégie de réallocation ne pénalise pas la consommation ;
3. la stratégie de réallocation est robuste aux aléas d'exécution, comme notamment le ralentissement de nœuds ;
4. la stratégie de réallocation s'adapte à la libération de nouveaux jobs.

Méthodes de références. Pour comparer la stratégie de réallocation exécutée de manière concurrente au processus de consommation, je considère les mêmes méthodes d'enchère de référence que dans la section 8.3.3, à savoir :

- une méthode d'allocation fondée sur des enchères séquentielles à un seul article (ASSI), où une offre correspond au *flowtime* si la tâche est attribuée à l'enchérisseur. Le processus d'allocation précède le processus de consommation. Lors de la libération de nouveaux jobs, les tâches supplémentaires sont affectées de la même manière ;
- une méthode de réallocation fondée sur des enchères séquentielles à un seul article (RSSI) exécutée de manière concurrente au processus de consommation où les agent-noeuds, chacun à leur tour, proposent de déléguer une de leurs tâches en attente. Lors de la libération de nouveaux jobs, les tâches supplémentaires sont réaffectées de la même manière.

Métriques. Les expérimentations sont réalisées en simulant la consommation des tâches. C'est pourquoi, pour définir les métriques, en plus du temps estimé d'exécution des tâches par les nœuds (cf. définition 5.3), je considère le **coût simulé** $c^S(\tau, \nu)$ comme le temps que l'agent exécutant du nœud ν consacre à la tâche τ :

- avec une connaissance parfaite de l'environnement d'exécution,

$$c^{S_E}(\tau, \nu_i) = c(\tau, \nu_i) \tag{8.6}$$

- avec le ralentissement de la moitié des nœuds (ceux de rang impair),

$$c^{S_H}(\tau, \nu_i) = \begin{cases} 2 \times c(\tau, \nu_i) & \text{si } i \bmod 2 = 1 \\ c(\tau, \nu_i) & \text{sinon.} \end{cases} \tag{8.7}$$

Au-delà de la durée moyenne de réalisation, ou *flowtime* moyen, je distingue ainsi :

- le ***flowtime* simulé** $C^S_{mean}(\overrightarrow{\mathsf{A}_t})$ calculé à partir d'une allocation $\overrightarrow{\mathsf{A}_t}$ selon les coûts simulés, comme si un oracle calculait la réallocation en temps constant ;
- le ***flowtime* réalisé** $C^R_{mean}(\overrightarrow{\mathsf{A}_t})$ calculé à partir des temps de réalisation des tâches effectivement mesurés.

Je définis le **taux d'amélioration de performance** :

$$\Gamma = \frac{C^R_{mean}(\overrightarrow{\mathsf{A}_0}) - C^R_{mean}(\overrightarrow{\mathsf{A}_e})}{C^R_{mean}(\overrightarrow{\mathsf{A}_0})} \tag{8.8}$$

où $\overrightarrow{A_e}$ est l'allocation des tâches au moment de leur exécution et $\overrightarrow{A_0}$ est l'allocation initiale. Il est à noter que si aucune réallocation n'a lieu pendant le processus, $\overrightarrow{A_e} = \overrightarrow{A_0}$. Le taux d'amélioration est positif si le *flowtime* réalisé de l'allocation atteinte par le processus de réallocation est meilleur (c'est-à-dire plus faible) que celui de l'allocation initiale. À ces métriques, s'ajoutent le nombre de tâches déjà réalisées par le processus de consommation et le nombre de tâches déjà déléguées pour un processus de réallocation, que cela soit pour la méthode d'enchère ou pour ma stratégie.

Protocole expérimental. Le protocole expérimental consiste, pour les différentes expériences, à générer aléatoirement 25 allocations initiales pour des problèmes d'allocation distincts. Je considère $m = 8$ nœuds, $\ell_0 = 4$ jobs, $n \in [40;320]$ tâches avec 10 ressources par tâche. Chaque ressource ρ_i est répliquée 3 fois et $|\rho_i| \in [0;500]$. Afin de ne pas déclencher des négociations inutiles dues à l'asynchronisme des opérations de consommations, je considère dans les expérimentations qu'une réallocation bilatérale est socialement rationnelle si elle fait décroître d'au moins une seconde le *flowtime*.

8.4.1 La stratégie de réallocation améliore le *flowtime*.

Hypothèse. Je souhaite vérifier que la stratégie de réallocation permet d'améliorer le *flowtime* moyen. Pour cela, je mesure les différentes métriques présentées ci-dessus pour la stratégie, ainsi que pour les méthodes d'enchères ASSI et RSSI à partir d'une allocation initiale aléatoire.

Résultats. La figure 8.7 montre les médianes et les écarts types des différents *flowtimes* en fonction du nombre de tâches. Nous observons que le *flowtime* réalisé atteint par la méthode d'allocation ASSI est pire que le *flowtime* réalisé de l'allocation aléatoire initiale. Même si le *flowtime* simulé de ASSI est très bon, cette méthode diffère le processus de consommation afin d'obtenir préalablement une allocation équilibrée et donc pénalise le *flowtime* réalisé. À l'inverse, le *flowtime* réalisé des méthodes de réallocation appliquées au cours du processus de consommation, que cela soit notre stratégie ou qu'elle soit fondée sur des enchères (RSSI), est meilleur que le *flowtime* réalisé de l'allocation initiale aléatoire. De plus, le *flowtime* réalisé des méthodes de réallocation est borné par le *flowtime* simulé de la réallocation si un oracle calcule la réallocation en temps constant.

La figure 8.8 montre selon une échelle de temps logarithmique l'évolution des médianes et écarts-types de nos métriques au cours de la consommation de 128 tâches. Elle confirme que le meilleur *flowtime* réalisé est celui obtenu par ma stratégie lorsqu'elle est exécutée de manière concurrente au processus de consommation.

Analyse. Contrairement à ASSI, les deux autres méthodes améliorent le *flowtime* en réallouant au cours du processus de consommation les tâches non locales dont la délégation réduit le coût. Comme le nombre de délégations de ma stratégie est supérieur au nombre de délégations de la méthode de réallocation fondée sur des enchères, le taux d'amélioration de performance (Γ) de ma stratégie est meilleur (entre 13 % et 23 %) que le taux d'amélioration de performance de la méthode de référence (entre 0 % et 15 %).

La réactivité (*responsiveness*) de ma stratégie s'explique par le fait qu'avant même que le processus de consommation des tâches ne commence (après $0,1$ seconde), près

de 30 tâches sont déjà réallouées grâce à de multiples négociations bilatérales.

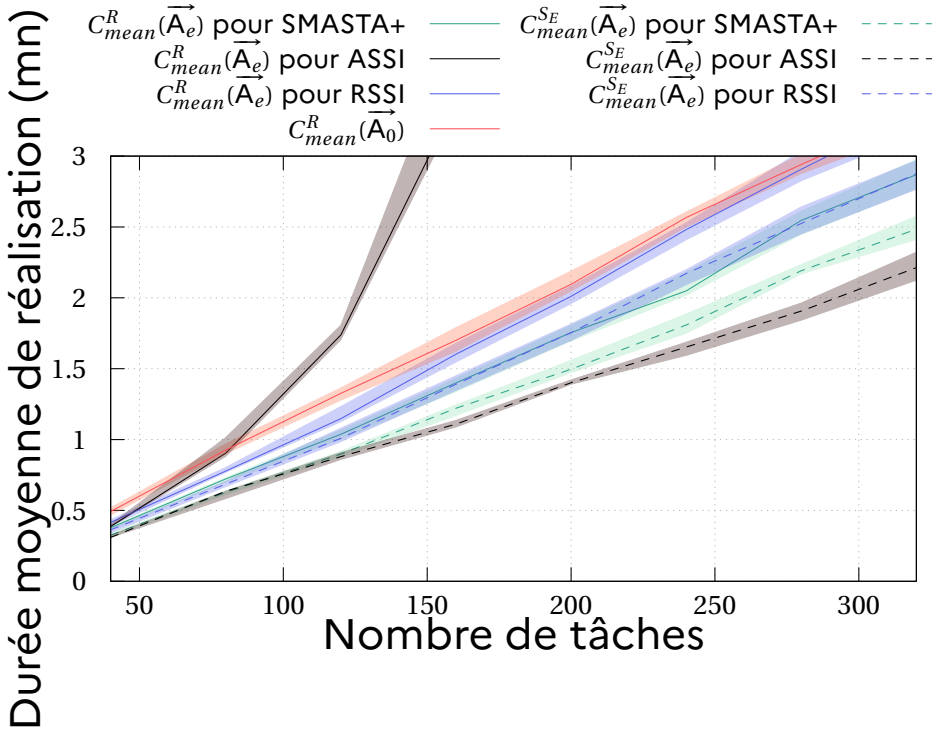


Figure 8.7 – Réallocation continue depuis une allocation aléatoire en fonction du nombre de tâches

8.4.2 La stratégie de réallocation ne pénalise pas la consommation.

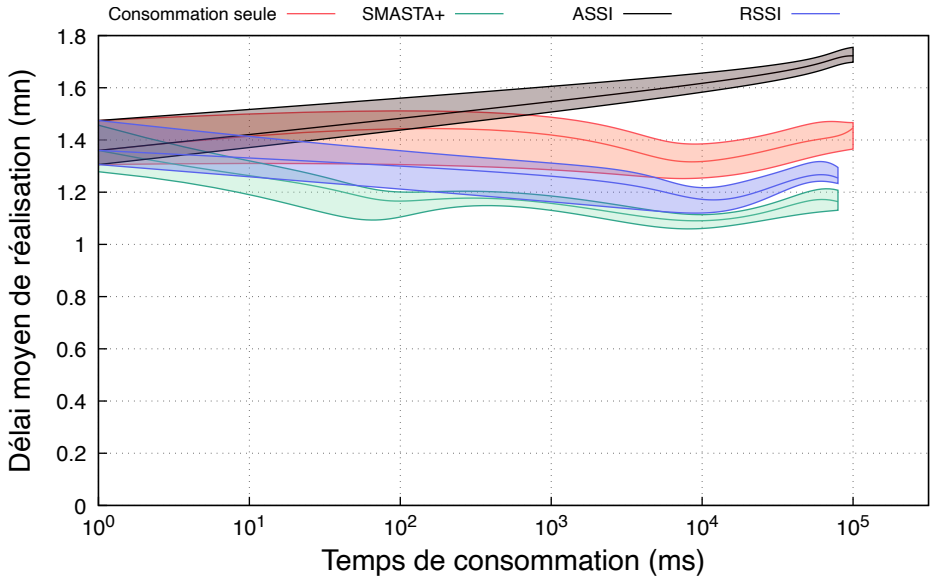
Hypothèse. Je vérifie dans cette section que la stratégie de réallocation ne pénalise pas la consommation. Pour cela, je considère ici uniquement des allocations initiales stables, c'est-à-dire qui ne peuvent pas être améliorées par le processus de réallocation.

Résultats. Dans la figure 8.9, le *flowtime* réalisé des méthodes de réallocation, mesuré en fonction du nombre de tâches, est similaire au *flowtime* réalisé de l'allocation initiale.

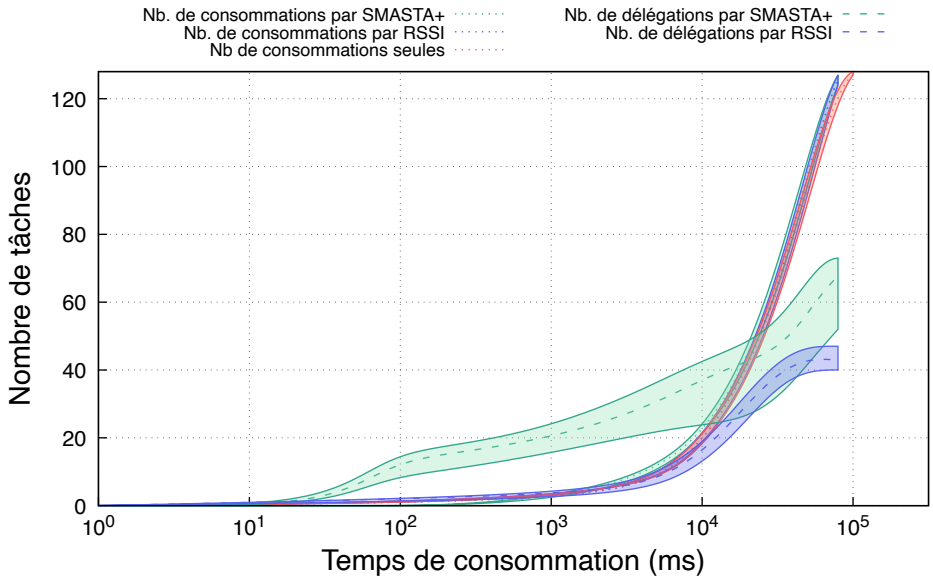
Analyse. Le surcoût de la négociation, que cela soit des enchères ou des marchandages bilatéraux, est négligeable car le processus de négociation est concurrent au processus de consommation et, dans le cas de notre stratégie, aucune négociation n'est déclenchée lorsque les agents estiment que l'allocation est stable.

8.4.3 La stratégie de réallocation s'adapte aux aléas d'exécution.

Hypothèse. Je vérifie dans cette section que ma stratégie de réallocation s'adapte aux aléas d'exécution. Pour cela, je considère ici le coût qui simule le ralentissement de



(a) Durée moyenne de réalisation



(b) Nombre de tâches consommées/déléguées

Figure 8.8 – Réallocation continue au cours de la consommation de 128 tâches

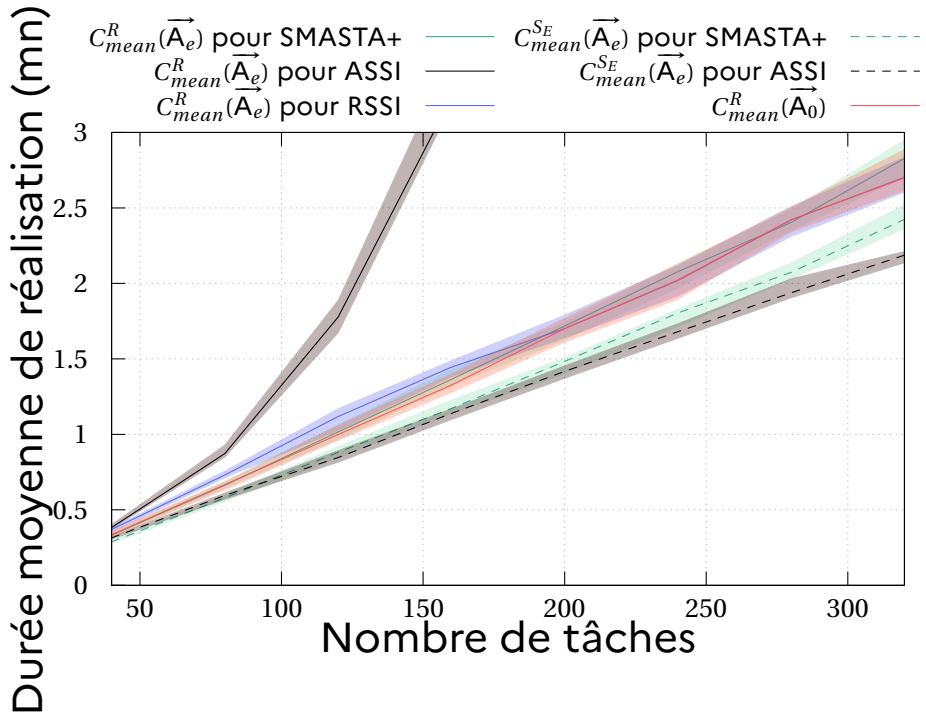


Figure 8.9 – Réallocation continue depuis une allocation stable

la moitié des nœuds (cf. équation 8.7).

Résultats. On observe dans la figure 8.10 que les délais moyens de réalisation ont doublé à cause des aléas d'exécution. Le *flowtime* réalisé et le *flowtime* simulé de la méthode d'allocation fondée sur des enchères (ASSI), qui ne prend pas en compte le ralentissement de la moitié des nœuds, sont supérieurs aux mêmes configurations sans ralentissement (cf. figure 8.7).

Analyse. Contrairement à la méthode de réallocation fondée sur des enchères (RSSI), le *flowtime* réalisé de ma stratégie reste meilleur que le *flowtime* réalisé de l'allocation initiale aléatoire et ce malgré une connaissance imparfaite de l'environnement d'exécution des agents, ces derniers se basant sur les coûts estimés pour réallouer. Prendre en compte les temps d'exécution effectifs des tâches déjà réalisées permet au taux d'amélioration de performance Γ de se situer entre 17 % et 29 %.

8.4.4 La stratégie de réallocation s'adapte à la libération des jobs.

Hypothèse. Je vérifie dans cette section que ma stratégie de réallocation s'adapte à la libération de nouveaux jobs au fil de l'eau.

Protocole expérimental. Je considère ici à nouveau que les allocations initiales sont aléatoires et que les agents ont une connaissance parfaite de l'environnement d'exécution (cf. équation 8.6). J'examine la consommation de 3 jobs libérés à t_0, chacun constitué de 32 tâches. Un quatrième job, également constitué de 32 tâches, est libéré 10 secondes après le début de la consommation des autres jobs.

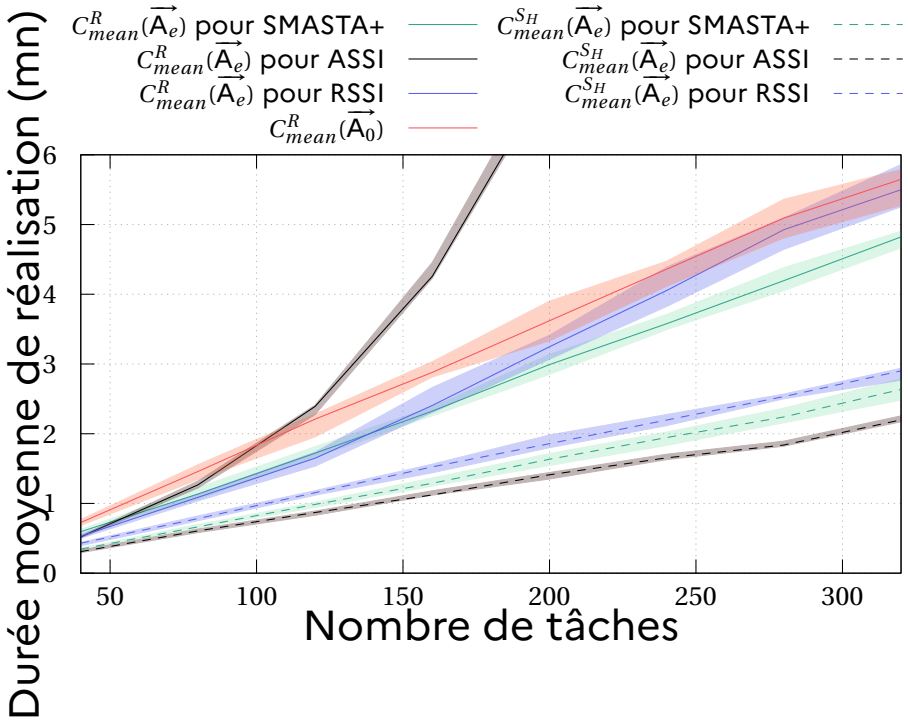


Figure 8.10 – Réallocation continue avec des aléas d'exécution

Résultats. La figure 8.11 montre selon une échelle de temps logarithmique l'évolution des médianes et écarts-types de nos métriques au cours de la consommation.La figure 8.11a révèle une nette amélioration du *flowtime* moyen avec notre stratégie. La figure 8.11b met en évidence le nombre plus important de délégations réalisées au cours du processus par notre stratégie.

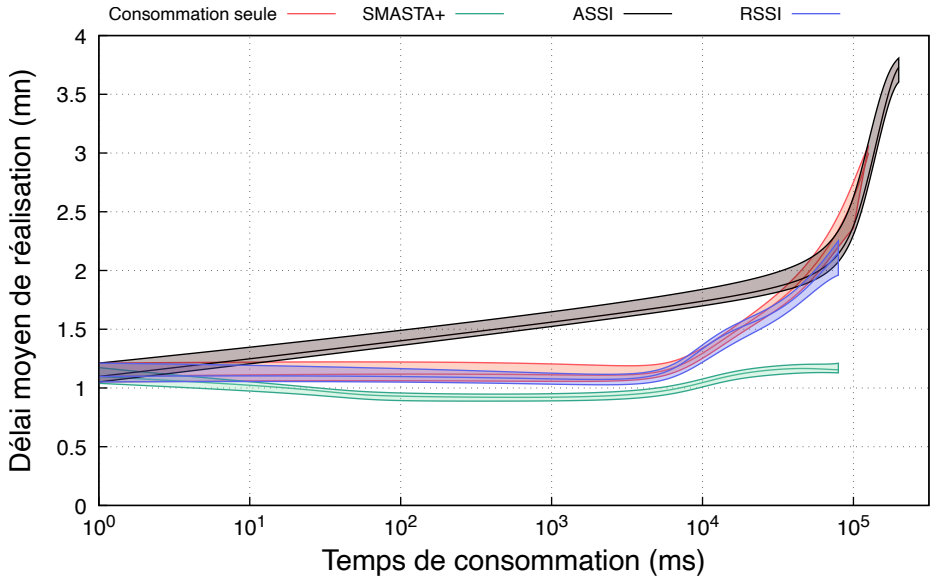
Analyse. Ma stratégie de réallocation améliore nettement le *flowtime*, mais elle permet aussi d'obtenir un temps total d'exécution (*makespan*) bien meilleur que le temps de consommation sans réallocation ou avec l'allocation (ASSI), et du même ordre que le temps d'exécution avec réallocation (RSSI). Contrairement à ASSI et RSSI, ma stratégie réaffecte les tâches d'un job dès qu'il est libéré afin d'améliorer le *flowtime* réalisé et de réduire le temps de consommation.

8.5 Synthèse

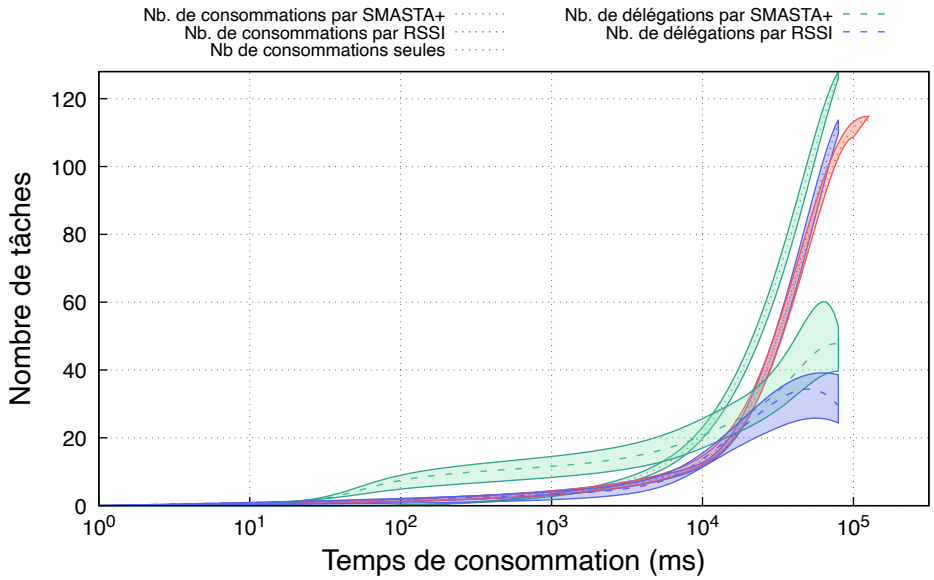
J'ai présenté dans ce chapitre les expérimentations permettant de valider le modèle présenté dans les chapitres 6 et 7.

Les expérimentations basées sur la réallocation instantanée ont permis de valider les points suivants.

1. La stratégie de contre-offre, utilisée pour s'extraire des minima locaux pouvant être atteints par la stratégie de délégation seule, permet d'améliorer le *flowtime* moyen, aux prix d'un surcoût computationnel acceptable.
2. La stratégie de délégation n-aire permet de réduire le temps de réordonnancement



(a) Délai moyen de réalisation



(b) Nombre de tâches consommées/déléguées

Figure 8.11 – Réallocation continue après la libération d'un nouveau job

en atteignant une allocation stable plus rapidement, sans détériorer significativement le *flowtime* moyen.

3. La stratégie de réallocation permet de réduire significativement le temps de réordonnancement, comparativement aux méthodes d'enchères de référence, tout en fournissant un *flowtime* moyen acceptable.
4. La stratégie de réallocation fournit également de meilleurs résultats qu'une méthode DCOP qui ne passe pas l'échelle pour ce type de problème.

Ces deux derniers points résultent du fait que ma méthode favorise un réordonnancement rapide des tâches, plutôt que la recherche de la solution optimale, ce qui permet une adaptation rapide.

En résumé, même si dans le cas de la réallocation instantanée, le *flowtime* moyen fourni par ma stratégie est moins bon que celui des méthodes de référence, cela est largement compensé par le gain de temps de réordonnancement. Ce gain de temps de réordonnancement permet de faciliter la consommation des tâches, comme le vérifient les expérimentations dédiées à la réallocation continue au cours de la consommation. Ces expérimentations ont en effet validés les hypothèses suivantes :

- La stratégie de réallocation permet d'améliorer le *flowtime* moyen grâce aux multiples réallocations bilatérales, dont une partie survient avant même que le processus de consommation ne commence.
- La stratégie de réallocation ne pénalise pas la consommation, les agents ne déclenchant pas de négociation lorsque l'allocation est stable.
- La stratégie de réallocation est robuste aux aléas d'exécution comme le ralentissement de nœuds.
- La stratégie de réallocation s'adapte dès la libération de jobs.

Même si une analyse de sensibilité pour étudier l'influence du facteur de réplication, du surcoût induit par la récupération des ressources non locales (κ) ou du délai d'annulation de la négociation (*timeout*) mériterait une étude approfondie, mes campagnes d'expérimentation permettent de valider empiriquement l'efficacité de la réactivité de ma stratégie multi-agents.

Chapitre 9

Conclusion

Sommaire

9.1 Synthèse **117**
9.2 Limites **119**
9.3 Perspectives **119**

9.1 Synthèse

Problème et approche. Dans cette thèse, je me suis intéressée au problème d'allocation de jobs constitués de tâches situées dans le cadre du traitement de données massives. Plus précisément, le problème étudié consiste en plusieurs jobs concurrents soumis par différents utilisateurs et pouvant arriver au fil de l'eau et dont la durée moyenne de réalisation, doit être minimisée.

Un job est un ensemble de tâches indépendantes, non divisibles et non préemptables qui sont réparties sur différents nœuds de calcul pour être exécutées. Les ressources nécessaires aux tâches, qui sont les données à traiter, sont transférables et non consommables. Elles sont réparties parmi les nœuds. Chaque ressource reste accessible à n'importe quel nœud, mais le coût d'une tâche dépend de si ses ressources nécessaires sont locales ou distantes au nœud de calcul chargé de l'exécuter.

Pour résoudre ce problème, j'ai proposé un modèle multi-agents où les nœuds de calcul sont contrôlés par des agents, dits agents-nœuds, qui négocient pour réallouer les tâches afin d'aboutir à une meilleure répartition tandis que les tâches sont consommées simultanément.

Je défends la thèse selon laquelle l'approche centrée « individus » est appropriée pour aborder des problèmes d'allocation intraitables en pratique, de par la combinatoire de l'ordonnancement, et dont la formulation est complexe. Le paradigme multi-agents, modulaire, décentralisé et adaptatif permet la distribution d'heuristiques qui passe à l'échelle. Intrinsèquement réactives, les méthodes multi-agents d'allocation continue s'adaptent, sans nécessiter d'apprentissage ou d'exploration préalable, aux estimations inexactes des temps d'exécution, aux perturbations (comme le ralentissement

des exécutants) ainsi qu'à la consommation et la libération des tâches caractéristiques d'une allocation continue.

Formalisation. J'ai proposé une formalisation du problème d'allocation continue de tâches situées, ou STAP pour *Situated Task Allocation Problem*. Une instance d'un problème STAP est constituée d'un système distribué de m nœuds de calcul et d'autant de nœuds de ressources, et d'un ensemble de jobs concurrents chacun formés de plusieurs tâches dont les coûts pour les nœuds dépendent de la localité des ressources.

Afin de minimiser la durée moyenne de réalisation des jobs, ou *flowtime* moyen, l'objectif est d'améliorer une allocation de tâches donnée. Pour cela les agents effectuent des opérations de réallocation « socialement rationnelles », qui leur permettent de déléguer ou d'échanger des tâches afin d'atteindre une allocation stable.

Modèle multi-agents. J'ai proposé un modèle multi-agents où les agents disposent de stratégie pour détecter les opportunités de réallocations pertinentes. Chaque agent négocie en s'appuyant sur un modèle des pairs construit à partir des croyances sur les lots de tâches de ses pairs.

Les agents disposent d'une stratégie de consommation spécifiant dans quel ordre ils exécutent les tâches. Ils négocient suivant un protocole d'offres alternées à un tour et appliquent une stratégie de négociation constituée d'une stratégie d'offre, d'une stratégie d'acceptation, d'une stratégie de contre-offre et d'une règle d'acceptabilité basée sur le critère de rationalité.

Mise en œuvre. Afin de mettre en œuvre ces mécanismes, j'ai proposé une architecture multi-agents basée sur le principe de séparation des rôles où chaque agent-nœud est constitué de trois agents composants, de façon à pouvoir réaliser simultanément consommations et réallocations. Ainsi, un agent-nœud est constitué :

- d'un exécutant chargé de consommer les tâches ;
- d'un négociateur s'appuyant sur le modèle des pairs pour négocier les réallocations ;
- d'un gestionnaire gérant le lot de tâches.

Les réallocations se déroulent suivant deux phases de négociation synchronisées par un agent superviseur et alternant jusqu'à ce que toutes les tâches soient consommées :

- lors d'une première phase, les agents réallouent les tâches en effectuant des délégations n-aires socialement rationnelles, sans contre-partie ;
- lors d'une seconde phase, les agents proposent des délégations unaires, socialement rationnelles susceptibles d'aboutir à des échanges grâce à la stratégie de contre-offre, afin de s'extraire des minima locaux pouvant être atteints lors de la première phase.

Le comportement des agents est représenté sous la forme d'automates à états finis, qui se transposent naturellement lors de l'implémentation avec le langage de programmation `Scala` et la bibliothèque `Akka`.

Expérimentations. Les expérimentations menées ont permis de valider empiriquement l'efficacité de la réactivité de ma stratégie multi-agents. En effet, ma stratégie favorise un réordonnancement rapide, au lieu de rechercher une solution optimale, ce qui

permet une adaptation efficace et facilite la consommation des tâches. Ainsi, la stratégie de réallocation, appliquée simultanément au processus de consommation, permet d'améliorer le *flowtime* moyen en réallouant les tâches, sans pénaliser la consommation. Elle permet de s'adapter aux aléas d'exécution, comme le ralentissement de nœuds et à la libération de jobs au fil de l'eau.

9.2 Limites

La classe d'application concrète que j'examine dans cette thèse consiste à mettre en œuvre le modèle de conception MapReduce pour le traitement de données massives sur une grappe de serveurs. De cette classe d'application découlent plusieurs hypothèses concernant la formalisation du problème. Celles-ci peuvent s'avérer trop restrictives pour d'autres applications pratiques telles que l'équilibrage de charge dans une fédération de clusters, la planification de missions pour une flotte de robots mobiles ou la gestion d'une constellation de satellites.

J'ai supposé qu'une ressource pouvait être soit locale, soit distante pour un nœud de calcul (cf. section 5.2). Cette hypothèse, basée sur la présence ou l'absence d'une ressource dans le voisinage du nœud, s'avère inadaptée pour une grappe géographique où les nœuds sont déployés sur plusieurs sites dispersés géographiquement, voire au sein de réseaux étendus (WAN) couvrant plusieurs pays. Dans un tel contexte, il serait nécessaire d'associer une métrique numérique à la distance entre une ressource et un nœud, plutôt qu'une simple valeur booléenne. De plus, j'ai considéré que la propriété de voisinage était statique. Cependant, si les ressources sont non seulement distantes, mais également mutualisées entre les tâches, il pourrait être intéressant d'affiner le coût des tâches en tenant compte des ressources préalablement rapatriées pour exécuter les tâches précédentes et mises en mémoire tampon.

J'ai considéré que les relations d'accointances entre les agents-nœuds suivaient la topologie du réseau des exécutants. Dans le cadre de cette thèse, chaque agent dispose d'un modèle de tous ses pairs (cf. section 6.2). Concrètement, chaque agent-nœud peut communiquer de manière fiable avec n'importe quel autre agent-nœud. La réception en temps fini des messages émis est garantie (cf. section 8.2). Cependant, cette hypothèse d'un réseau complètement connecté et d'une fiabilité de communication élevée peut s'avérer inadaptée dans d'autres contextes applicatifs.

Garantir une borne théorique de la performance d'une méthode multi-agents ou du moins évaluer à partir de cas d'étude réalistes sa robustesse vis-à-vis de la connectivité du graphe sous-jacent (qu'il soit statique ou dynamique) ou lorsque les communications sont intermittentes, représente un problème complexe qui dépasse le cadre de cette thèse.

9.3 Perspectives

Dans la continuité de la thèse et dans le cadre du traitement de données massives où plusieurs jobs sont exécutés simultanément sur une réserve de nœuds - typique de l'informatique en nuage - les travaux futurs visent à étudier l'approvisionnement et le désapprovisionnement en ressources de calcul en fonction des besoins. Le verrou scientifique réside dans la conception d'une stratégie multi-agents capable d'identifier en

continu les menaces au sein d'une allocation sous-provisionnée et les opportunités au sein d'une allocation sur-provisionnée pour déclencher l'activation et la libération des nœuds, puis la réallocation des tâches pour l'équilibrage de la charge tout en tenant compte de la localité des données.

L'étape suivante consiste à adapter notre modèle multi-agents pour qu'il fonctionne efficacement dans un environnement dynamique, de sorte que le système s'adapte de manière optimale aux besoins des utilisateurs au moment de l'exécution, tout en équilibrant la charge en tenant compte de la localité des données. L'idée consiste à considérer la négociation multi-agents pour l'approvisionnement élastique de ressources en fonction des demandes faites sur ces ressources par les tâches des utilisateurs (Al-Dhuraibi et al., 2018).

Cette idée peut être formulée techniquement comme un problème d'optimisation de contraintes dans lequel, étant donné un ensemble de nœuds de calcul disponibles, il faut minimiser le nombre de nœuds qui doivent être activés afin d'atteindre une fourchette acceptable autour d'un objectif prédéfini axé sur le temps de réponse moyen pour accomplir les tâches. Il serait également nécessaire de minimiser le temps de réaction pour ajuster le nombre de nœuds, c'est-à-dire l'intervalle de temps entre le moment où une (re)configuration des nœuds a été déclenchée et le moment où l'adaptation a été achevée.

Il s'agirait donc de proposer une stratégie multi-agents qui présente un comportement élastique permettant d'augmenter ou de réduire automatiquement l'infrastructure en ajoutant ou en supprimant suffisamment de nœuds de calcul dédiés à l'application afin de s'adapter au temps de réponse attendu par les utilisateurs. Afin de fournir à la volée juste assez de ressources informatiques pour respecter le SLA (*Service Level Agreement*), une première étape consisterait à envisager un agent superviseur qui surveille notre système en flux tendu et soit déclenche l'activation de nœuds lorsqu'il est en sous-provisionnement, soit libère des nœuds lorsque le système est en sur-provisionnement. Les travaux futurs devraient étendre le processus d'approvisionnement des nœuds à un processus itéré, dynamique et continu, qui se déroule en même temps que le système.

Bibliographie

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. & Merle, P. (2018). Elasticity in Cloud Computing : State of the Art and Research Challenges. *IEEE Transactions on Services Computing*, *11*(2), 430-447 (cf. p. 120).

Apache Hadoop. (2024). https://hadoop.apache.org. (Cf. p. 2)

Apache Hive. (2023). https://hive.apache.org. (Cf. p. 2)

Apache Spark. (2018). https://spark.apache.org. (Cf. p. 2)

Baert, Q. (2019). *Négociation multi-agents pour la réallocation dynamique de tâches et application au patron de conception MapReduce* (thèse de doct.). (Cf. p. 1, 55, 81, 93, 98)
Thèse de doctorat dirigée par Routier, Jean-Christophe Caron, Anne-Cécile et Morge, Maxime.

Baert, Q., Caron, A.-C., Morge, M., Routier, J.-C. & Stathis, K. (2019). Stratégie situationnelle pour l'équilibrage de charge, In *Actes des 27ièmes journées francophones sur les systèmes multi-agents*, Cépaduès Editions. (Cf. p. 15, 22, 31, 38).

Beynier, A., Charpillet, F., Szer, D. & Mouaddib, A.-I. (2010). DEC-MDP / DEC-POMDP. In O. S. Olivier Buffet (Éd.), *Markov Decision Processes in Artificial Intelligence* (p. 277-313). Wiley-ISTE. (Cf. p. 36).

BigQuery. (2024). https://cloud.google.com/bigquery. (Cf. p. 2)

Bonnet, G. & Tessier, C. (2009). Evaluation d'un système multirobot. Cas d'une constellation de satellites. *Revue d'Intelligence Artificielle*, *23*, 565-592 (cf. p. 10, 11).

Bruno, J., Coffman, E. G., Jr. & Sethi, R. (1974). Scheduling Independent Tasks to Reduce Mean Finishing Time. *Commun. ACM*, *17*(7), 382-387 (cf. p. 22, 28, 38).

Chevaleyre, Y., Dunne, P., Endriss, U., Lang, J., Lemaitre, M., Maudet, N., Padget, J., Phelps, S., Rodrguez-Aguilar, J. & Sousa, P. (2006). Issues in Multiagent Resource Allocation. *Informatica*, *30*, 3-31 (cf. p. 11).

Choi, H.-L., Brunet, L. & How, J. P. (2009). Consensus-based decentralized auctions for robust task allocation. *IEEE transactions on robotics*, *25*(4), 912-926 (cf. p. 30).

Conway, R., Maxwell, W. & Miller, L. (1967). *Theory of Scheduling*. Addison- Wesley, Reading, MA. (Cf. p. 22, 29, 38).

Cormen, T. & Leiserson, C. (2010). *Algorithmique : cours avec 957 exercices et 158 problèmes* (3e édition). Paris, Dunod. (Cf. p. 32).

Dean, J. & Ghemawat, S. (2008). MapReduce : simplified data processing on large clusters. *Commun. ACM*, *51*(1), 107-113 (cf. p. 2).

Endriss, U., Maudet, N., Sadri, F. & Toni, F. (2006). Negotiating Socially Optimal Allocations of Resources. *Journal of Artificial Intelligence Research*, *25*, 315-348 (cf. p. 55).

Enright, J. & Wurman, P. R. (2011). Optimization and Coordinated Autonomy in Mobile Fulfillment Systems, In *Automated Action Planning for Autonomous Mobile Robots, Papers from the 2011 AAAI Workshop, San Francisco, California, USA, August 7, 2011*, AAAI. (Cf. p. 22, 38).

Fioretto, F., Pontelli, E. & Yeoh, W. (2018). Distributed constraint optimization problems and applications : A survey. *Journal of Artificial Intelligence Research*, *61*, 623-698 (cf. p. 32, 33, 35).

Fulgham, B. & Gouy, I. (2021). The Computer Language Benchmarks Game. Récupérée 22 avril 2021, à partir de https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. (Cf. p. 106)

Gerkey, B. P. & Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The international journal of robotics research*, *23*(9), 939-954 (cf. p. 14, 15).

Gueneron, J. (2022). *Formation de coalitions répétée dans un contexte stochastique : protocoles et expérimentations* (thèse de doct.). Université de Caen Normandie. (Cf. p. 32)
Thèse de doctorat dirigée par Bonnet, Grégory.

Hariri, A. M. A. & Potts, N., Chris. (1991). Heuristics for scheduling unrelated parallel machines. *Computers & operations research*, *18*(3), 323-331 (cf. p. 29).

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence*, *8*(3), 323-364 (cf. p. 92).

Horn, W. (1973). Minimizing average flow time with parallel machines. *Operations Research*, *21*(3), 846-847 (cf. p. 28).

Horowitz, E. & Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, *23*(2), 317-327 (cf. p. 29).

Ibarra, O. H. & Kim, C. E. (1977). Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of ACM*, *24*(2), 280-289 (cf. p. 22, 29, 38).

Jiang, Y. (2016). A survey of task allocation and load balancing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, *27*(2), 585-599 (cf. p. 18, 30).

Jiang, Y. & Li, Z. (2011). Locality-sensitive task allocation and load balancing in networked multiagent systems : Talent versus centrality. *J. Parallel Distrib. Comput.*, *71*(6), 822-836 (cf. p. 15).

Koenig, S., Tovey, C., Lagoudakis, M., Markakis, V., Kempe, D., Keskinocak, P., Kleywegt, A., Meyerson, A. & Jain, S. (2006). The power of sequential single-item auctions for agent coordination, In *Proceedings of the 21st AAAI Conference on Artificial Intelligence*, AAAI. (Cf. p. 3, 30, 38, 105).

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval research logistics quarterly*, *2*(1-2), 83-97 (cf. p. 22, 27, 38).

Lightbend. (2020). Akka is the implementation of the Actor Model on the JVM. http://akka.io. (Cf. p. 92)

Lisp. (2019). https://common-lisp.net. (Cf. p. 2)

Lynch, N. A. (1996). *Distributed Algorithms*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (Cf. p. 34).

Maheswaran, R. T., Pearce, J. P. & Tambe, M. (2004). Distributed Algorithms for DCOP : A Graphical-Game-Based Approach, In *Proc. of the 17th International Conference on Parallel and Distributed Computing Systems (ISCA)*, ISCA. (Cf. p. 35, 106).

Mills-Tettey, G. A., Stentz, A. & Dias, M. B. (2007). The dynamic hungarian algorithm for the assignment problem with changing costs. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27* (cf. p. 22, 29, 38).

Morge, M. (2019). Répartition des tâches pour la collecte de colis : démonstration [Poster]. Cépuadès. (Cf. p. 9).

Munkres, J. (1957). Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, *5*(1), 32-38. https://doi.org/10.1137/0105003 (cf. p. 27)

Nongaillard, A. (2009). *An Agent-Based Approach for Distributed Resource Allocations* (thèse de doct.). Université Lille 1 et Université Concordia (Montréal, Canada). (Cf. p. 1)
Thèse de doctorat dirigée par Mathieu, Philippe et Jaumard, Brigitte.

Pentico, D. W. (2007). Assignment problems : A golden anniversary survey. *European Journal of Operational Research*, *176*(2), 774-793 (cf. p. 18, 20).

Petcu, A. & Faltings, B. (2005a). DPOP : A Scalable Method for Multiagent Constraint Optimization, Professional Book Center. (Cf. p. 35).

Petcu, A. & Faltings, B. (2005b). Superstabilizing, Fault-Containing Distributed Combinatorial Optimization., AAAI Press. (Cf. p. 36).

Picard, G. (2018). Optimisation sous contraintes distribuée : une introduction au domaine, In *Actes des 26ièmes journées francophones sur les systèmes multi-agents*, Cépaduès. (Cf. p. 33).

Pinedo, M. L. (2008). *Scheduling. Theory, Algorithms, and Systems. Third Edition*. Springer. (Cf. p. 12, 13, 26, 64).

Rubinstein, A. (1982). Perfect Equilibrium in a Bargaining Model. *Econometrica*, *50*(1), 97-102 (cf. p. 32, 59).

Russell, S. J. & Norvig, P. (2009). *Artificial Intelligence : a modern approach* (3e éd.). Pearson. (Cf. p. 26).

Rust, P. & Picard, G. (2021). pyDCOP is a python library for Distributed Constraints Optimization. https://github.com/Orange-OpenSource/pyDcop. (Cf. p. 106)

Rust, P., Picard, G. & Ramparany, F. (2018). Mise en place d'une décision collective résiliente sur une infrastructure IoT à l'aide du framework PyDCOP (démonstration), In *Actes Vingt-sixièmes journées francophones sur les systèmes multi-agents, JFSMA 2018, Métabief, France, 10-12 Octobre 2018*, Cépaduès. (Cf. p. 106).

Schaerf, A., Shoham, Y. & Tennenholtz, M. (1995). Adaptive Load Balancing : A Study in Multi-Agent Learning. *J. Artif. Intell. Res.*, *2*, 475-500 (cf. p. 22, 36, 38).

Shehory, O. & Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artificial Intelligence*, *101*(1-2), 165-200 (cf. p. 21, 22, 30, 32, 38).

Smith, R. G. (1980). The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on computers*, *29*(12), 1104-1113 (cf. p. 31, 32).

Snowflake. (2024). https://www.snowflake.com. (Cf. p. 2)

Su, J., Wei, M. & Liu, A. (2018). A Robust Predictive-Reactive Allocating Approach, Considering Random Design Change in Complex Product Design Processes. *Int. J. Comput. Intell. Syst.*, *11*(1), 1210-1228 (cf. p. 22, 30, 38).

Teradata. (2024). https://www.teradata.com. (Cf. p. 2)

Turner, J., Meng, Q., Schaefer, G. & Soltoggio, A. (2018). Distributed Strategy Adaptation with a Prediction Function in Multi-Agent Task Allocation, In *Proc. of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, International Foundation for Autonomous Agents ; Multiagent Systems. (Cf. p. 22, 30, 31, 38).

Walsh, W. E. & Wellman, M. P. (1998). A market protocol for decentralized task allocation, In *Proc. of ICMAS*, IEEE. (Cf. p. 30).

Weyns, D., Helleboogh, A. & Holvoet, T. (2005). The Packet-World : A Test Bed for Investigating Situated Multi-Agent Systems. In *Software Agent-Based Applications, Platforms and Development Kits* (p. 383-408). Springer. (Cf. p. 9).

Liste des notations

Terme	Description	Pages
$\mathcal{D}$	Système distribué constitué de nœuds de calcul et de nœuds de ressources	46
$\mathcal{P}$	Ensembles des nœuds de calcul du système distribué	46
$\mathcal{N}_r$	Ensembles des nœuds du système	46
$\mathcal{V}$	Propriété de voisinage sur les nœuds de calcul et les nœuds de ressources	46
$\mathcal{R}$	Ensemble de ressources nécessaires à l'exécution des tâches	46
$\mathsf{local}(\rho, \nu_c)$	Prédicat de localité de la ressource ρ sur le nœud ν_c	46
$\mathcal{R}_\tau$	Ensemble de ressources nécessaires à l'exécution de la tâche τ	46
$\mathcal{T}$	Ensemble des tâches	46
$\mathcal{J}$	Ensemble des jobs	46
$t^0_{\mathsf{J}_i}$	Date de libération du job	46
$\mathsf{c}(\tau, \nu_j)$	Coût de la tâche τ pour le nœud ν_j	47
STAP	Problème d'allocation de tâches situées	47
$\overrightarrow{\mathsf{A}_t}$	Allocation à l'instant t	48
$\mathsf{B}_{i,t}$	Ensemble des tâches assignées au nœud ν_i à l'instant t	48
$\vec{\mathsf{B}}_{i,t}$	Liste ordonnée des tâches assignées au nœud ν_i à l'instant t	48
$\prec_i$	Relation d'ordre stricte et totale sur les tâches adoptée par le nœud ν_i	48
$\min_{\prec_i} \mathsf{B}_{i,t}$	La prochaine tâche à réaliser pour le nœud ν_i à l'instant t	48
$\mathsf{job}(\tau)$	Job contenant la tâche τ	47
$\nu(\tau, \overrightarrow{\mathsf{A}_t})$	Nœud auquel est affectée la tâche τ dans l'allocation $\overrightarrow{\mathsf{A}_t}$	48
$\delta(\tau, \nu_i)$	Délai de la tâche τ pour le nœud ν_i	48
$\mathsf{C}_\tau(\overrightarrow{\mathsf{A}_t})$	Durée de réalisation de la tâche τ dans l'allocation $\overrightarrow{\mathsf{A}_t}$	48

Terme	Description	Pages
$\mathsf{C}_\mathsf{J}(\overrightarrow{\mathsf{A}_t})$	Durée de réalisation du J dans l'allocation $\overrightarrow{\mathsf{A}_t}$	49
$\mathsf{C}_\mathsf{J}(\vec{\mathsf{B}}_{i,t})$	Durée de réalisation du J dans le lot de tâches $\vec{\mathsf{B}}_{i,t}$	53
$\mathsf{C}(\overrightarrow{\mathsf{A}_t})$	Durée de réalisation de l'allocation $\overrightarrow{\mathsf{A}_t}$	49
$\mathsf{C}_{mean}(\overrightarrow{\mathsf{A}_t})$	Durée moyenne de réalisation de l'allocation $\overrightarrow{\mathsf{A}_t}$	22
$\mathsf{C}_{max}(\overrightarrow{\mathsf{A}_t})$	*Makespan* moyen de l'allocation $\overrightarrow{\mathsf{A}_t}$	22
$\mathsf{W}(\overrightarrow{\mathsf{A}_t})$	Débit de réalisation pour l'allocation $\overrightarrow{\mathsf{A}_t}$	22
$\lambda(\nu_i, \vec{\mathsf{B}}_{i,t})$	Consommation d'une tâche par le nœud ν_i dans le lot de tâches $\vec{\mathsf{B}}_{i,t}$	53
$\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{\mathsf{A}_t})$	Réallocation de la liste de tâches T_1 du donneur ν_i au receveur ν_j en échange de la liste de tâches T_2 à partir de l'allocation $\overrightarrow{\mathsf{A}_t}$	54
$\overrightarrow{\mathsf{B}_{i,t} \ominus \mathrm{T}_1 \oplus \mathrm{T}_2}$	Ensemble des tâches $\mathsf{B}_{i,t} \cup \mathrm{T}_2 \setminus \mathrm{T}_1$ ordonnées selon $\prec_i$	54
$\sigma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{\mathsf{A}_t})$	Échange des listes non vides de tâches T_1 et T_2 entre les nœuds ν_i et ν_j à partir de l'allocation $\overrightarrow{\mathsf{A}_t}$	54
$\delta(\mathrm{T}_1, \nu_i, \nu_j, \overrightarrow{\mathsf{A}_t})$	Délégation d'une liste de tâches T_1 du donneur ν_i au receveur ν_j à partir de l'allocation $\overrightarrow{\mathsf{A}_t}$	54
$\mathcal{B}_i$	Base de croyances $\mathcal{B}_i$ de l'agent ν_i	60
$\blacktriangleleft_i$	Relation d'ordre totale et stricte sur les jobs adoptée par le nœud ν_i	64
$\blacktriangleleft_j^i$	Relation d'ordre totale et stricte sur les jobs adoptée par le nœud ν_j selon les croyances du nœud ν_i	60
$\lhd_i$	Relation d'ordre totale et stricte sur les tâches adoptée par le nœud ν_i	64
$<$	Ordre naturel sur les identifiants des tâches, des jobs ou des nœuds	63
$\mathsf{c}^i(\mathsf{J}, \nu_j)$	Ce que l'agent ν_i croit connaître du coût de chaque job $\mathsf{J} \in \mathcal{J}$ pour son pair ν_j	60
$\mathsf{w}_j^i(\overrightarrow{\mathsf{A}_t})$	Ce que l'agent ν_i déduit de la charge de travail du nœud ν_j dans l'allocation $\overrightarrow{\mathsf{A}_t}$	60
$\mathsf{C}_\mathsf{J}{}^i(\vec{\mathsf{B}}_{j,t})$	Ce que l'agent ν_i croit connaitre de la durée de réalisation du job J dans le bundle $\vec{\mathsf{B}}_{j,t}$	60
$\mathsf{C}_\mathsf{J}{}^i(\overrightarrow{\mathsf{A}_t})$	Ce que l'agent ν_i croit connaitre de la durée de réalisation du job J dans l'allocation $\overrightarrow{\mathsf{A}_t}$	61
$\nu_j = \mathsf{nodeMax}^i(\overrightarrow{\mathsf{A}_t}, \mathsf{J})$	Selon le sujet ν_i, la cible ν_j est un facteur limitant pour un job J dans l'allocation $\overrightarrow{\mathsf{A}_t}$	61
$\mathsf{L}(\overrightarrow{\mathsf{A}_t})$	Ratio de disponibilité locale de l'allocation $\overrightarrow{\mathsf{A}_t}$	102
$C^S_{mean}(\overrightarrow{\mathsf{A}_t})$	*Flowtime* simulé calculé à partir de l'allocation $\overrightarrow{\mathsf{A}_t}$	109
$C^R_{mean}(\overrightarrow{\mathsf{A}_t})$	*Flowtime* réalisé calculé à partir des temps de réalisation des tâches effectivement mesurés	109
Γ	Taux d'amélioration de performance	109

Liste des figures

2.1 Illustration du problème de transport de colis 10
2.2 Illustration du problème de contrôle d'une constellation de satellites (Bonnet & Tessier, 2009) 11
2.3 Délai et durée de réalisation pour la tâche τ 17
2.4 Ordonnancement des tâches pour l'exemple 2.7 18

3.1 Résolution du problème de l'exemple 3.1 28
3.2 Classification des algorithmes DCOP de Fioretto, Pontelli et Yeoh (2018) . . 35

4.1 Vue d'ensemble de l'architecture multi-agents 43

5.1 Répartition des ressources et allocation des tâches aux nœuds pour notre exemple fil rouge où les jobs J_1, J_2 et J_3 sont respectivement représentés en rouge, bleu et vert. 51
5.2 Diagramme de Gantt illustrant l'allocation de l'exemple 5.2 à l'instant $t = 4$. 52
5.3 Allocations de l'exemple 5.4 résultant de réallocations bilatérales 56

6.1 Allocation $\overrightarrow{A_t}$ de l'exemple 6.1 vue par l'agent v_1 qui ignore la description du lot de ses pairs 62
6.2 Allocations dans l'exemple 6.2 où tous les agents adoptent une stratégie LCJF, une stratégie GCJF, ou une stratégie MEJMLNL. 69
6.3 Protocole de négociation bilatérale entre un agent « proposant » et un agent « répondant » pour la délégation de lots de tâches. 70
6.4 Protocole de négociation bilatérale entre un agent « proposant » et un agent « répondant » pour l'échange de tâches. 71
6.5 Allocations de l'exemple 6.3 75

7.1 Architecture composite d'un agent-nœud 81
7.2 Interactions entre le négociateur et l'exécutant 82
7.3 Interactions entre le gestionnaire et le négociateur lors de la première phase de négociation dans le cas d'une délégation unaire. Les étiquettes QRC indiquent que le gestionnaire interagit avec l'exécutant selon le protocole de la figure 7.2b pour prendre en compte le temps d'exécution restant pour la tâche en cours. 83

7.4 Interactions entre le gestionnaire et le négociateur lors de la deuxième phase de négociation. Dans le cas où le répondant n'identifie pas d'échange socialement rationnel, les interactions sont les mêmes que dans le cas des délégations (cf. figure 7.3). 84
7.5 Alternance des phases de négociation 85
7.6 Comportement de l'exécutant 85
7.7 Comportement simplifié du gestionnaire 87
7.8 Comportement simplifié du négociateur depuis l'état *Responder* 90
7.9 Comportement simplifié du négociateur depuis les états *Initial* et *Waiting* 92
7.10 Extrait de code SMASTA+ du comportement de l'exécutant représenté par l'automate 7.6 94
7.11 Extrait de code SMASTA+ définissant tous les états de l'exécutant 94

8.1 Comparaison des stratégies sans et avec échanges 101
8.2 *Flowtime* moyen au cours du temps lors de l'exécution de la stratégie avec échanges sur une instance caractéristique 102
8.3 Comparaison des stratégies de délégation unaire et n-aire 103
8.4 Évolution du *flowtime* moyen pour une instance caractéristique non équilibrée avec 16 agents-nœuds 104
8.5 Réallocation instantanée 107
8.6 Diagrammes en boîte à moustaches des durées moyennes de réalisation pour les réallocations obtenues par notre stratégie en 0.4 millisecondes (en moyenne) avec l'algorithme MGM-2 dont le *timeout* est respectivement 2, 5 et 10 secondes 108
8.7 Réallocation continue depuis une allocation aléatoire en fonction du nombre de tâches 111
8.8 Réallocation continue au cours de la consommation de 128 tâches 112
8.9 Réallocation continue depuis une allocation stable 113
8.10 Réallocation continue avec des aléas d'exécution 114
8.11 Réallocation continue après la libération d'un nouveau job 115

A.1 Comportement complet du gestionnaire 136
A.2 Comportement complet du négociateur pour les états *Init* et *Waiting* . . . 138
A.3 Comportement complet du négociateur pour les autres états 139
A.4 Comportement du superviseur 142

Liste des tableaux

2.1 Grille d'analyse des caractéristiques pour différents problèmes d'allocation 22

3.1 Le coût des tâches pour chacun des exécutants 28
3.2 Grille d'analyse des méthodes d'allocation (haut) et de réallocation (bas) . 38

5.1 Le coût des tâches pour chaque nœud 50
5.2 Le coût des tâches pour chaque nœud 52

6.1 Le coût des tâches pour chaque nœud dans l'exemple 6.2 68

7.1 Complexité des comportements 95

Troisième partie

Annexes

Chapitre A

Comportement des agents

Cette annexe contient la spécification complète du comportement des agents sous la forme de machines à états finis qui ont été présentées sans les détails dans le chapitre 7.

A.1 Gestionnaire

La figure A.1 représente le comportement complet de l'agent gestionnaire, dont la version simplifiée a été présentée dans la section 7.4.3. À chaque mise à jour du lot de tâches, que ce soit lorsque l'exécutant termine la consommation d'une tâche ou lorsqu'une réallocation a lieu, le gestionnaire informe à la fois les pairs (`peers!Inform`) et le négociateur (`negotiator!GiveUpdatedBundle(`$\vec{\mathrm{B}}_{i,t}$`)`). Les messages de l'exécutant concernant la consommation d'une tâche reçus dans l'état *TransitionState* sont mis en attente (`worker:Done` $\Rightarrow$ `stash`), jusqu'à ce que le gestionnaire retourne dans l'état *Running* (`unstashAll`).

A.2 Négociateur

Les figures A.2 et A.3 représente le comportement complet du négociateur, déjà présenté de façon simplifié dans la section 7.4.4.

Au-delà de ses connaissances et croyances, le négociateur dispose dans son état mental de plusieurs variables lui permettant de gérer le processus de négociation :

- *informers* est l'ensemble des pairs dont il a reçu un message `Inform` pour l'initialisation du modèle des pairs ;
- *lastOffer* est la délégation qui a été suggerée lors de la dernière proposition ;
- *isProactive* est vraie si le comportement pro-actif de l'agent en tant que proposant est activé ;
- *isFirstStage* est vraie si la phase de négociation en cours est la première phase, et est fausse si la phase en cours est la seconde.

Chaque réception d'un message `GiveUpdatedBundle` de la part du gestionnaire entraîne une mise à jour de la base de croyances (`updateBeliefBase`). Toute proposition reçue en dehors de l'état *Responder* est mise en attente (`stash`) jusqu'au retour du négociateur dans cet état (`unstashAll`). Le négociateur dispose également d'un minuteur, lui

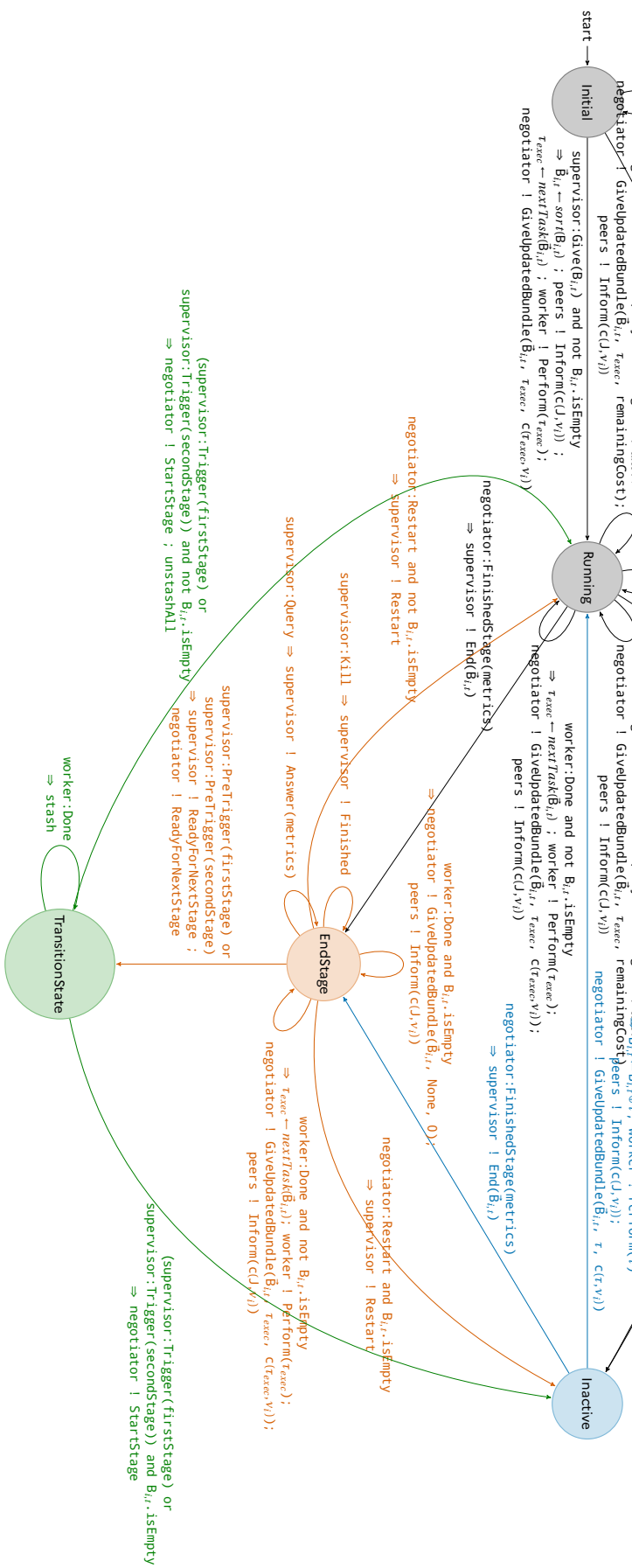


Figure A.1 – Comportement complet du gestionnaire

permettant de considérée une offre comme étant rejetée s'il n'obtient pas de réponse après une certaine temporisation.

A.3 Superviseur

Le comportement du superviseur est représenté sur l'automate de la figure A.4.
Le superviseur peut se trouver dans l'un des états suivants :

- l'état *Initial* dans lequel il attend que les agents soient prêts et leur communique leurs lots de tâches ;
- l'état *RunningFirstStage* dans lequel il attend la fin du déroulement de la première phase de négociation ;
- l'état *EndFirstStage* dans lequel il entre quand la première phase est terminée et où il attend de pouvoir amorcer la transition vers la phase suivante ou vers la fin du processus ;
- l'état *FirstToSecond* où il attend que les agents soient prêts pour démarrer la deuxième phase ;
- l'état *RunningSecondStage* dans lequel il attend la fin du déroulement de la deuxième phase de négociation ;
- l'état *EndSecondStage* dans lequel il entre quand la seconde phase est terminée et où il attend de pouvoir amorcer la transition vers la phase suivante ou vers la fin du processus ;
- l'état *SecondToFirst* où il attend que les agents soient prêts pour démarrer une nouvelle première phase ;
- l'état *TransitionalEnd* quand les négociations sont terminées et où il attend soit la fin du processus, soit le redémarrage des phases de négociation ;
- l'état *TerminalStage* dans lequel il entre quand le processus est terminé.

Le superviseur maintient plusieurs variables d'état à jour, lui permettant savoir ce qu'il doit faire :

- *allocation* représente l'allocation courante ;
- *isTwoStageProcess* est un booléen qui est vrai si le processus de négociation se déroule en deux phases, ou faux s'il n'y a qu'une seule phase ;
- *sumReady* est le nombre d'agents-nœuds ayant déjà reçu leur lot de tâches initial ;
- *desperatedNodes* est l'ensemble des agents qui n'ont plus de réallocation potentielles à proposer ;
- *sumReadyForNext* est le nombre d'agents prêts à commencer la phase de négociation suivante ;
- *answerNodes* est l'ensemble des agents ayant envoyés leurs métriques au superviseur ;
- *sumKilled* est le nombre d'agents qui ont accusé réception du message signalant la fin du processus ;
- *lastQueryId* est l'identifiant de la dernière requête pour la collecte des métriques ;

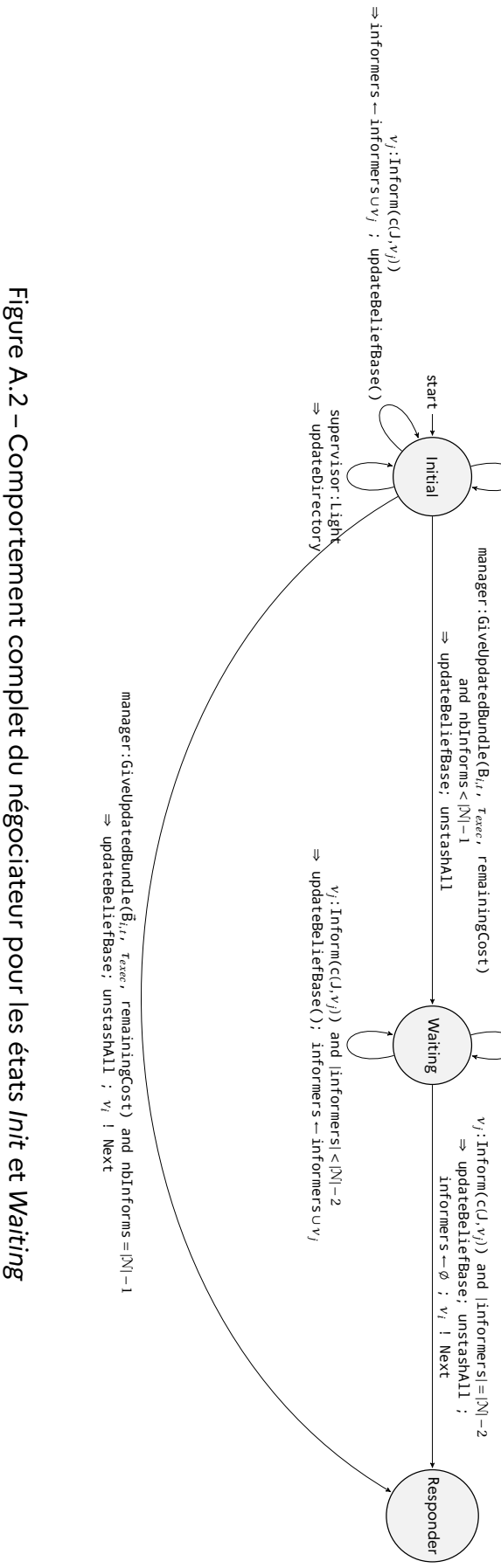


Figure A.2 – Comportement complet du négociateur pour les états *Init* et *Waiting*

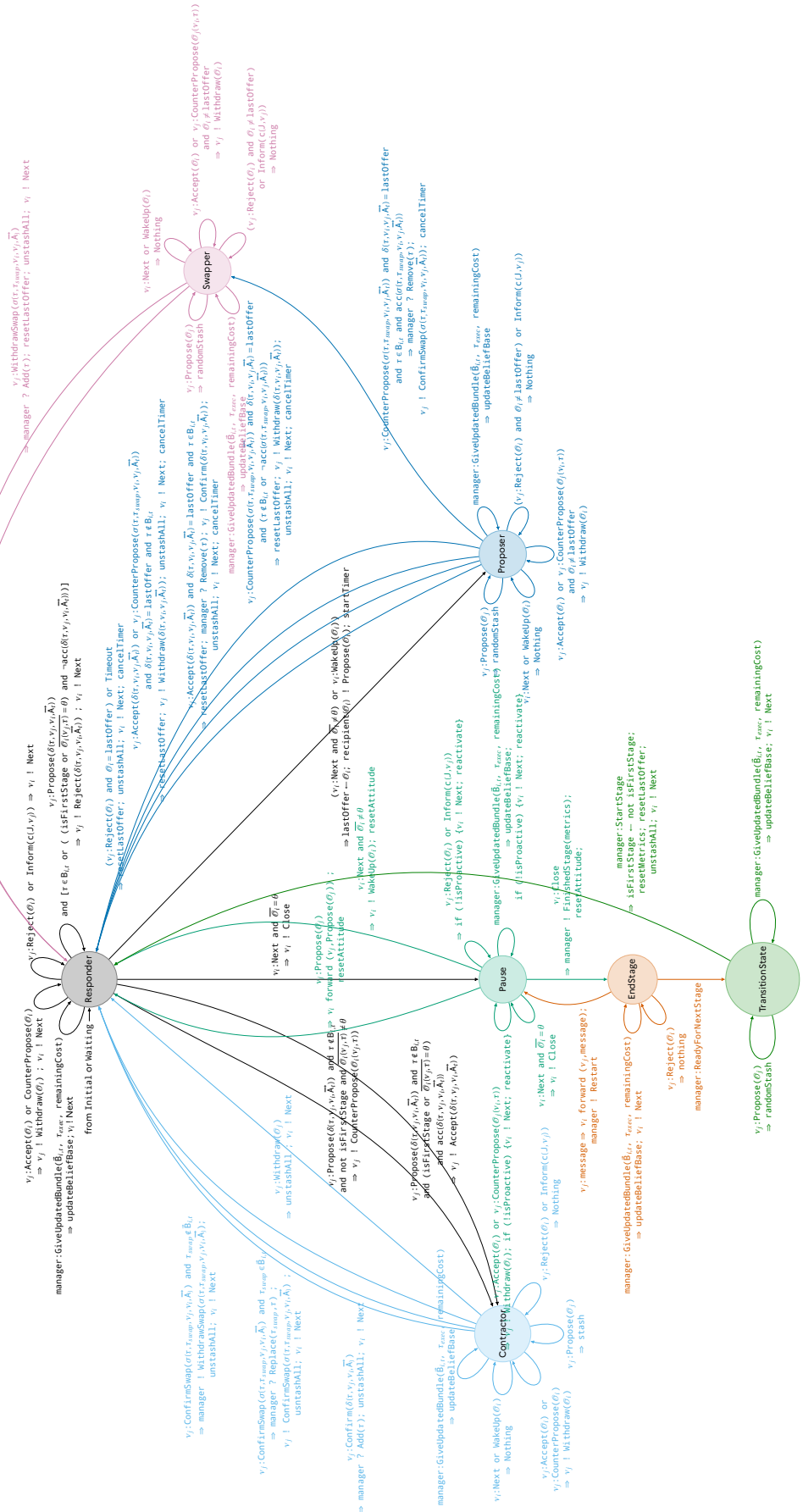


Figure A.3 – Comportement complet du négociateur pour les autres états

- *metrics* contient des mesures relatives au processus de négociation telles que l'allocation courante, le nombre de propositions et de contre-propositions envoyées, le nombre de délégations et d'échanges confirmés, le nombre de *timeouts*, le nombre de tâches déléguées et le nombre de transactions lors de la phase courante.

Dans l'état *InitialState*, le superviseur peut recevoir les messages suivants messages suivants :

- un message d'initialisation `Init(`$\overrightarrow{A_t}$`)` de l'équilibreur de charge lui donnant l'allocation initiale. En réponse, le superviseur réveille les agents ;
- un message `Ready` venant d'un des agents-nœuds qui entraîne l'incrémentation de la variable *sumReady*. De plus, si tous les agents sont prêts, le superviseur leur envoie leur lot initial (`Give(`$\vec{B}_{i,t}$`)`), ainsi que les messages qu'il avait précédemment empilés et passe dans l'état *RunnningFirstStage* ;
- un message `End` de la part d'un des agents-nœuds qui est stocké.

Dans l'état *RunningFirstStage*, le superviseur peut recevoir les messages suivants :

- un message `End` d'un des agents-nœuds qui entraîne la mise à jour de l'allocation courante et l'ajout de l'agent dans l'ensemble *desperatedNodes*. De plus, si tous les agents ont terminé, le superviseur s'envoie un message `Close` et passe dans l'état *EndFirstStage* ;
- un message `Restart` de la part d'un agent-nœud qui conduit le superviseur à le retirer de l'ensemble *desperatedNodes* ;
- un message `Answer` ou `Close` obsolète qui est ignoré.

Dans l'état *EndFirstStage*, le superviseur peut recevoir les messages suivants :

- un message `Close` envoyé par lui-même qui l'amène à envoyer une requête (`Query`) à tous les agents, leur demandant de communiquer leurs nouvelles métriques. Le superviseur garde en mémoire l'identifiant de la requête dans *lastQueryId* ;
- un message `Restart` de la part d'un agent-nœud qui conduit le superviseur à réinitialiser *desperateNodes*, *answerNodes* et *lastQuery*, et à revenir dans l'état *RunningFirstStage* ;
- une réponse `Answer` d'un agent-nœud. Si son identifiant ne correspond pas à *lastQueryId*, cette réponse est ignorée. Si l'identifiant correspond, le superviseur met à jour les métriques et ajoute l'agent à l'ensemble *answerNodes*. Si les réponses de tous les agents ont déjà été reçues, si une seconde phase de négociation est définie et s'il reste encore des tâches à consommer, le superviseur initie la transition vers la phase suivante en envoyant un message `PreTrigger(secondStage)` à tous les agents et il réinitialise *sumReady*, *sumReadyForNext* et *desperatedNodes* avant de passer dans l'état *FirstToSecond*. Si toutes les tâches ont été consommées, le superviseur met à jour les métriques, envoie un message `Kill` à tous les agents et passe dans l'état *TransitionalEnd*.

Dans l'état *FirstToSecond*, le superviseur peut recevoir les messages suivants :

- un message `ReadyForNextStage` d'un agent-nœud. Si tous les agents sont prêts, le superviseur déclenche la phase suivante en envoyant un message `Trigger(secondStage)` à tous les agents, il réintialise la variable *sumReadyForNext* et passe dans l'état *RunningSecondStage*. Si tous les agents ne sont pas prêts, le superviseur reste dans l'état *FirstToSecond* et incrémente la variable *sumReadyForNext* ;

- un message `Answer` ou `Close` obsolète qui est ignoré.

Dans l'état *RunningSecondStage*, le superviseur peut recevoir les messages suivants :

- un message `End` d'un agent-nœud qui entraîne la mise à jour de l'allocation courante et l'ajout de l'agent dans l'ensemble *desperatedNodes*. De plus, si tous les agents ont terminé, le superviseur s'envoie un message `Close` et passe dans l'état *EndSecondStage* ;
- un message `Restart` de la part d'un agent-nœud qui conduit le superviseur à le retirer de l'ensemble *desperatedNodes* ;
- un message `Answer` ou `Close` obsolète qui est ignoré.

Dans l'état *EndSecondStage*, le superviseur peut recevoir les messages suivants :

- un message `Close` envoyé par lui-même qui l'amène à envoyer une requête (`Query`) à tous les agents, leur demandant de communiquer leurs nouvelles métriques. Le superviseur garde en mémoire l'identifiant de la requête dans *lastQueryId* ;
- un message `Restart` de la part d'un agent-nœud qui conduit le superviseur à réinitialiser *desperateNodes*, *answerNodes* et *lastQuery*, et à revenir dans l'état *RunningSecondStage* ;
- une réponse `Answer` d'un agent-nœud. Si son identifiant ne correspond pas à *lastQueryId*, cette réponse est ignorée. Si l'identifiant correspond, le superviseur met à jour les métriques et ajoute l'agent à l'ensemble *answerNodes*. Si les réponses de tous les agents ont déjà été reçues et s'il reste encore des tâches à consommer, le superviseur initie la transition vers la phase suivante en envoyant un message `PreTrigger(firstStage)` à tous les agents et il réinitialise les variables *sumReady*, *sumReadyForNext* et *desperatedNodes* avant de passer dans l'état *SecondToFirst*. Si tous les lots de tâches sont vides, le superviseur met à jour les métriques, envoie un message `Kill` à tous les agents et passe dans l'état *TransitionalEnd*.

Dans l'état *SecondToFirst*, le superviseur peut recevoir les messages suivants :

- un message `ReadyForNextStage` d'un agent-nœud. Si tous les agents sont prêts, le superviseur déclenche la phase suivante en envoyant un message `Trigger(firstStage)` à tous les agents, il réintialise la variable *sumReadyForNext* et passe dans l'état *RunningFirstStage*. Si tous les agents ne sont pas prêts, le superviseur reste dans l'état *SecondToFirst* et incrémente la variable *sumReadyForNext* ;
- un message `Answer` ou `Close` obsolète qui est ignoré.

Dans l'état *TransitionalEnd*, le superviseur peut recevoir les messages suivants :

- un message `Finished` de la part d'un agent-nœud. Si tous les autres agents ont ont également terminés, le superviseur envoie le résultat *Outcome* à l'équilibreur de charge et passe dans l'état *TerminalState*. Si tous les agents ne sont pas prêts, il reste dans l'état *TransitionalEnd* et incrémente la variable *sumKilled* ;
- un message `Answer` obsolète qui est ignoré.

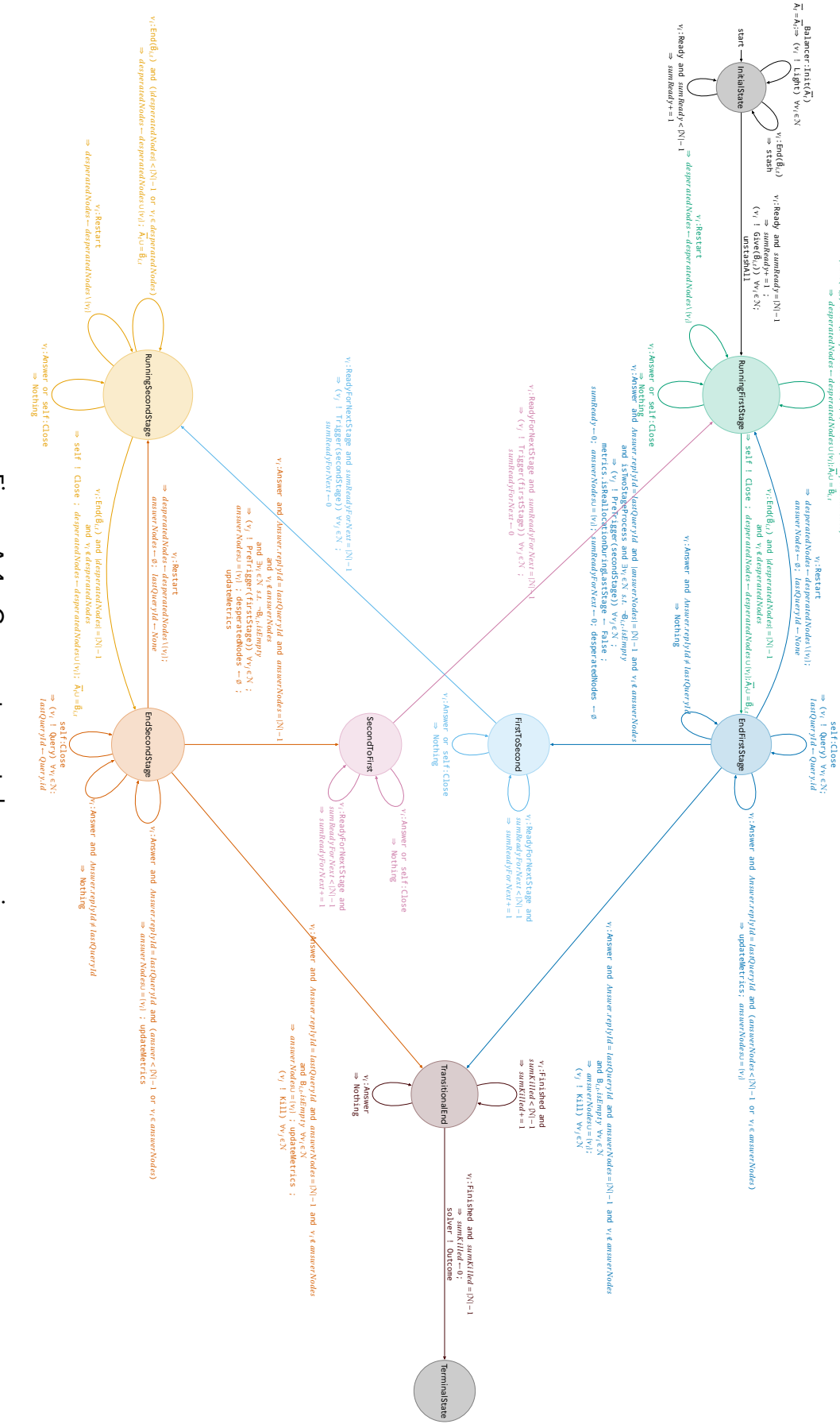


Figure A.4 – Comportement du superviseur

Chapitre B

Publications

Les travaux présentés dans cette thèse ont fait l'objet des publications suivantes.

Articles dans des conférences internationales

Beauprez, E., Caron, A.-C., MORGE, M. & Routier, J.-C. (2023). Adaptive Consumption by Continuous Negotiation (P. Mathieu, F. Dignum, P. Novais & F. D. la Prieta, Éd.). In P. Mathieu, F. Dignum, P. Novais & F. D. la Prieta (Éd.), *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics*, Springer Nature Switzerland. PAAMS. https://doi.org/10.1007/978-3-031-37616-0_3

Lecture Notes in Computer Science

Beauprez, E., Caron, A.-C., Morge, M. & Routier, J.-C. (2022b). Task Bundle Delegation for Reducing the Flowtime. In A. P. Rocha, L. Steels & J. van den Herik (Éd.), *Agents and Artificial Intelligence, 13th International Conference, ICAART 2021, Online streaming, February 4-6, 2021, Revised Selected Papers* (p. 22-45). Springer, Cham.

Articles dans des revues nationales

Beauprez, E., Caron, A.-C., Morge, M. & Routier, J.-C. (2023b). Délégation de lots de tâches pour la réduction de la durée moyenne de réalisation. *Revue Ouverte d'Intelligence Artificielle*, *4*(2), 193-221. https://doi.org/10.5802/roia.62

Articles dans des conférences nationales

Beauprez, E., Caron, A.-C., Morge, M. & Routier, J.-C. (2023a). Consommation adaptative par négociation continue (M. Morge, Éd.). In M. Morge (Éd.), *Explicabilité des systèmes multi-agents - Trente-et-unièmes journées francophones sur les systèmes multi-agents, JFSMA 2023, Strasbourg, France, July 5-7, 2023*, Cépaduès.

Beauprez, E., Caron, A.-C., Morge, M. & Routier, J.-C. (2022a). Échange de tâches pour la réduction de la durée moyenne de réalisation, In *Actes des Trentièmes journées francophones sur les systèmes multi-agents (JFSMA)*, Saint-Étienne, France, Cépaudès.

Beauprez, E., Bigand, L., Caron, A.-C., Morge, M. & Routier, J.-C. (2021). Réaffectation de tâches de la théorie à la pratique : état de l'art et retour d'expérience, In *Actes des Vingt-neuvièmes journées francophones sur les systèmes multi-agents (JFSMA)*, Bordeaux, France, Cépaudès.

Beauprez, E., Caron, A.-C., Morge, M. & Routier, J.-C. (2021). Une stratégie de négociation multi-agents pour réduire la durée moyenne de réalisation, In *Actes des Vingt-neuvièmes journées francophones sur les systèmes multi-agents (JFSMA)*, Bordeaux, France, Cépaudès.

Logiciels

Beauprez, E., Caron, A.-C. & Morge, M. (2020). *SMASTA+ - Scala implementation of the Extended Multi-agents Situated Task Allocation*. Récupérée 18 février 2023, à partir de https://gitlab.univ-lille.fr/maxime.morge/smastaplus

Résumé

Mes travaux s'intègrent aux recherches menées dans l'équipe SMAC de CRIStAL en Intelligence Artificielle Distribuée.

Les sciences des données exploitent de larges volumes de données sur lesquelles des calculs sont effectués en parallèle par différents nœuds. Ces applications mettent à l'épreuve l'informatique distribuée en ce qui concerne l'allocation de tâches et l'équilibrage de charge. J'étudie dans cette thèse le problème de l'allocation continue de jobs concurrents, composés de tâches situées, sous-jacent au déploiement d'applications de traitement de données massives sur une grappe de serveurs. L'objectif est de minimiser le délai moyen de réalisation de ces jobs, appelée *flowtime*.

Je propose dans ce document un modèle multi-agents d'assignation tâches-exécutants où les nœuds de calcul sont contrôlés par des agents collaboratifs, appelés agents-nœuds, qui négocient des réallocations locales pour aboutir à une meilleure répartition des tâches. Ces négociations se déroulent au fil de l'exécution des tâches. Grâce à leur modèle des pairs, les agents-nœuds sont capables d'identifier des opportunités au sein de l'allocation courante pour marchander des délégations voire des échanges de tâches avec leurs semblables. Pour améliorer la réactivité (*responsiveness*) de la stratégie multi-agents qui repose sur l'exécution asynchrone de comportements individuels en interaction, le processus de négociation s'appuie sur de multiples négociations bilatérales concurrentes.

Mes campagnes d'expérimentation permettent de valider empiriquement l'efficacité de la réactivité de ma stratégie multi-agents. En effet, ma méthode favorise un réordonnancement rapide des tâches, plutôt que la recherche de la solution optimale, ce qui permet une adaptation rapide. Mes expérimentations montrent que, lorsqu'elle est exécutée de manière concurrente au processus de consommation, notre stratégie de réallocation : (1) réduit significativement le temps de réordonnancement ; (2) améliore le délai moyen de réalisation ; (3) ne pénalise pas la consommation ; (4) est robuste aux aléas d'exécution ; et (5) s'adapte à la libération de jobs.

Mots clés : Système multi-agents, Résolution collective de problèmes, Négociation multi-agents, Allocation de tâches.

Abstract

My work is part of the research done by the SMAC team in the laboratory CRIStAL in Distributed Artificial Intelligence.

Data sciences exploit large datasets on which computations are performed in parallel by different nodes. These applications challenge distributed computing in terms of task allocation and load-balancing. In this thesis, I study the problem of continuous allocation of concurrent jobs, composed of situated tasks, underlying the deployment of massive data processing applications on a cluster of servers. The objective is to minimise the mean flowtime of these jobs.

In this document, I propose a multi-agent task-worker assignment model where computing nodes are controlled by collaborative agents, called node agents, which negotiate local task reallocations to achieve a better task distribution. These negotiations take place during the tasks execution. Thanks to their peer modelling, node agents are able to identify opportunities within the current allocation to negotiate task delegations or even swaps with their peers. To improve the responsiveness of the multi-agent strategy, which is based on the asynchronous execution of interacting individual behaviours, the negotiation process is based on multiple concurrent bilateral negotiations.

My experimental campaigns allow me to empirically validate the efficiency of the reactivity of my multi-agent strategy. This is because my method encourages rapid reordering of tasks, rather than the search for the optimum solution, which allows responsiveness. My experiments show that, when executed concurrently with the consumption process, our reallocation strategy : (1) significantly reduces the rescheduling time ; (2) improves the flowtime ; (3) does not penalise the consumption ; (4) is robust to execution hazards ; and (5) adapts to the release of jobs.

Keywords : Multi-Agents System, Distributed Problem Solving, Agent-based Negotiation, Task Allocation.