Consommation adaptative par négociation continue

Ellie Beauprez, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier

Consommation adaptative par négociation continue

Ellie Beauprez
Ellie.Beauprez@univ-lille.fr

Anne-Cécile Caron
Anne-Cecile.Caron@univ-lille.fr

Maxime Morge
Maxime.Morge@univ-lille.fr

Jean-Christophe Routier
Jean-Christophe.Routier@univ-lille.fr

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Résumé

Nous étudions ici le problème de l'allocation de jobs concurrents, composés de tâches situées, sous-jacent au déploiement distribué du patron de conception MapReduce sur une grappe de serveurs. Afin de mettre en œuvre notre stratégie multi-agents qui vise à minimiser le délai moyen de réalisation des jobs (flowtime), nous proposons une architecture composite d'agent qui permet la concurrence des négociations et des consommations. Nos expérimentations montrent que, lorsqu'elle est exécutée en continue lors du processus de consommation, notre stratégie de réallocation : (1) améliore le flowtime ; (2) ne pénalise pas la consommation ; (3) est robuste aux aléas d'exécution.

Mots-clés : *Résolution collective de problèmes, Négociation multi-agents, Architecture d'agent*

Abstract

In this paper, we study the problem of allocating concurrent jobs, composed of situated tasks, underlying the distributed deployment of the MapReduce design pattern on a cluster. In order to implement our multi-agent strategy that aims at minimizing the mean flowtime of jobs, we propose a composite agent architecture that allows negotiation and consumption concurrency. Our experiments show that, when executed continuously during the consumption process, our reallocation strategy : (1) improves the flowtime ; (2) does not penalise the consumption ; (3) is robust to execution hazards.

Keywords: *Distributed Problem Solving, Agent-based Negotiation, Agent Architecture*

1 Introduction

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. C'est le cas de l'application pratique que nous considérons dans cet article : 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 [10]. Les jobs, qui doivent être exécutés le plus tôt possible, sont composés d'un ensemble de tâches, exécutées par les différents nœuds. L'exécution d'une tâche consiste à traiter des ressources situées sur les nœuds. Comme plusieurs ressources sont requises pour réaliser une tâche sur un nœud, son exécution peut nécessiter de récupérer des ressources disponibles sur d'autres nœuds, ce qui induit un surcoût.

De nombreux travaux adoptent le paradigme multi-agents pour aborder le problème de la réallocation de tâches et de l'équilibrage de charge parmi des exécutants multiples [2]. L'approche centrée individus permet la distribution d'heuristiques pour des problèmes impraticables à cause de la combinatoire des ordonnancements, autorisant ainsi le passage à l'échelle. De plus, intrinsèquement réactives, les méthodes multi-agents de réaffectation s'adaptent aux estimations inexactes des temps d'exécution et aux perturbations (consommation/libération de tâches, ralentissement des exécutants, etc.).

Dans [3], nous avons proposé une stratégie multi-agents de réallocation de tâches pour un ensemble de jobs devant être exécutés le plus tôt possible. Afin de minimiser le délai moyen de réalisation des jobs (*flowtime*), les agents, qui sont coopératifs, négocient pour déterminer les prochaines tâches à déléguer, voire à échanger. Cette stratégie nécessite le déploiement distribué d'agents autonomes qui consomment les tâches et échangent de manière continue certaines d'entre elles pour équilibrer l'allocation courante. Dans cet article, nous formalisons les opérations de consommation de tâches et celles de réallocation et nous proposons une architecture composite d'agent qui permet la concurrence des négociations et des consommations. Selon le principe de la séparation des préoccu-

pations, un premier agent composant est dédié à la consommation (i.e. l'exécution) des tâches, un second aux négociations des réallocations et un troisième à la coordination locale de ces opérations à travers la gestion du lot de tâches. La difficulté réside dans la conception des comportements des agents composants qui ne partagent pas un état global du système (e.g. l'allocation) mais disposent de connaissances locales et partielles. Nos expérimentations montrent que, lorsqu'elle est exécutée en continue lors du processus de consommation, notre stratégie de réallocation ne pénalise pas la consommation et peut améliorer le *flowtime* jusqu'à 37 %, même lorsque les agents ont une connaissance imparfaite de l'environnement d'exécution comme lors d'aléas (i.e. le ralentissement de nœuds).

Après un aperçu des travaux connexes dans la section 2, nous rappelons dans la section 3 la formalisation du problème d'allocation de jobs composés de tâches situées. La section 4 formalise les opérations de consommation/réallocation. Nous décrivons ensuite, dans la section 5, comment le processus de consommation et celui de réallocation sont entrelacés, puis nous détaillons notre architecture d'agents dans la section 6. La section 7 présente nos résultats expérimentaux. La section 8 résume notre contribution et présente nos perspectives.

2 Travaux connexes

De nombreux travaux ont abordé le problème de la réallocation de tâches parmi de multiples exécutants. L'approche centrée individus permet de surmonter les limites des solutions centralisées : l'impossibilité de résoudre les problèmes à grande échelle et la faible réactivité aux changements [2]. Les problèmes d'allocation dynamique des tâches nécessitent notamment de proposer des processus dynamiques qui s'ajustent en permanence aux changements de l'environnement d'exécution ou de performance des exécutants [11]. La plupart de ces travaux s'appuie sur l'algorithme à base de consensus (CBBA – *Consensus Based Bundle Algorithm*) [7] qui est une méthode multi-agents d'assignation en deux phases qui consiste à : (a) sélectionner les tâches à négocier à travers un processus d'enchère ; (b) déterminer les offres qui remportent ces enchères en résolvant les conflits potentiels. En particulier, notre architecture d'agent modulaire s'inspire largement de [1]. Toutefois, nos agents ne visent pas à minimiser le *makespan* (i.e. le temps d'exécution global) mais le *flowtime*. De plus, nous avons préféré ici un protocole de négociation bilatérale qui permet, en sélectionnant l'interlocuteur, de faire des propositions ciblées et donc de réduire les coûts computationel et communicationnel liés à la négociation. Finalement, la simulation de l'environnement d'exécution nous permet d'en contrôler les perturbations.

Chen et *al.* envisagent des problèmes d'allocation dynamique de tâches où les tâches sont libérées à des moments incertains [6]. Ils proposent d'ajuster l'allocation des tâches de façon continue en combinant le réordonnancement local des agents avec la réallocation des tâches entre agents. De manière similaire, notre stratégie multi-agents s'appuie sur une stratégie de consommation pour définir l'ordonnancement local des tâches et sur une stratégie de négociation des tâches à réallouer. Contrairement à [6], nous faisons ici l'hypothèse que l'ensemble des jobs sont initialement connus, mais nos agents sont susceptibles d'avoir une connaissance imparfaite de l'environnement d'exécution.

La plupart des travaux qui considèrent que les perturbations de l'environnement d'exécution font varier le coût des tâches s'appuient sur des techniques de recherche opérationnelle comme l'analyse de sensibilité pour évaluer la robustesse des optima aux perturbations [15], des méthodes incrémentales pour réparer l'allocation optimale initiale lorsque les coûts changent [14] ou l'optimisation combinatoire pour exploiter les mesures de dégradation [13]. De manière similaire, notre stratégie mesure notamment l'écart entre les progrès attendus et ceux observés en vue de modifier l'allocation. Toutefois, notre approche centrée individus permet de résoudre des problèmes à grande échelle.

Creech et *al.* aborde le problème de l'allocation des ressources et de la hiérarchisation des tâches dans les systèmes multi-agents distribués pour des environnements dynamiques [8]. Ils proposent un algorithme d'optimisation de l'allocation des ressources multi-groupes (MG-RAO) qui combine des algorithmes de mise à jour et de priorisation et qui utilise des techniques d'apprentissage par renforcement. À l'inverse des techniques d'apprentissage, notre solution ne nécessite aucun modèle préalable, ni des données, ni de l'environnement, et aucune phase d'exploration car cela ne serait pas pertinent pour l'application pratique qui nous concerne. En effet, le volume de données rend les pré-traitements et l'exploration trop coûteux. De plus, la variabilité des données les rend rapidement obsolète.

Nos précédentes expérimentations ont montré que la durée moyenne de réalisation atteinte par notre stratégie est meilleure que celle obtenue avec les techniques d'optimisation sous contraintes distribuée (DCOP) et reste proche de celle obtenue avec une heuristique classique, avec dans tous les cas un temps de réordonnancement significativement réduit [4]. Nous montrons dans cet article comment déployer cette stratégie de manière continue au cours du processus de consommation.

3 Problème

Cette section présente la formalisation, introduite dans [3], du problème d'allocation des jobs concurrents composés de tâches situées.

Un système distribué est composé d'un ensemble de nœuds de calcul capables d'exécuter des tâches. Ces tâches requièrent des ressources, transférables et non consommables, réparties parmi différents nœuds de ressources.

Définition 1 (Système distribué)**.** *Un **système distribué** est un quadruplet $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{E}, \mathcal{R} \rangle$ où :*
- *$\mathcal{P}$ est un ensemble de p nœuds de calcul ;*
- *$\mathcal{N}_r$ est un ensemble de r nœuds de ressource ;*
- *$\mathcal{E} : \mathcal{P} \times \mathcal{N}_r \to \{\top, \bot\}$ est une propriété de voisinage qui évalue si un nœud de calcul de l'ensemble $\mathcal{P}$ est local à un nœud de ressource dans $\mathcal{N}_r$;*
- *$\mathcal{R} = \{\rho_1, \ldots, \rho_k\}$ est un ensemble de ressources ayant des tailles $|\rho_i|$. La localisation des ressources, qui sont éventuellement répliquées, est déterminée par la fonction :*

$$l : \mathcal{R} \to 2^{\mathcal{N}_r} \tag{1}$$

Une ressource peut être locale ou distante d'un nœud de calcul, selon sa présence ou non sur un nœud de ressource dans le voisinage du nœud de calcul. À partir des fonctions $\mathcal{E}$ et l, nous définissons le prédicat de localité :

$$\begin{aligned} &\forall \nu_c \in \mathcal{P}, \forall \rho \in \mathcal{R},\ \mathrm{local}(\rho, \nu_c) \text{ ssi} \\ &\exists \nu_r \in l(\rho) \text{ t.q. } \mathcal{E}(\nu_c, \nu_r) \end{aligned} \tag{2}$$

Les ressources sont accessibles pour tous les nœuds de calcul, même celles sur les nœuds de ressource distants.

Un job est un ensemble de tâches indépendantes, non divisibles et non-préemptives. 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 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 $J_i = \{\tau_1, \ldots, \tau_{k_i}\}$.*

On note $\mathcal{T} = \cup_{1 \le i \le \ell} J_i$ 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, on note $\mathrm{job}(\tau)$ le job contenant la tâche τ. Nous faisons 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 ν_i est une estimation *a priori* de son temps d'exécution.

Définition 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 $c : \mathcal{T} \times \mathcal{N} \mapsto \mathbb{R}^*_+$ est telle que :*

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

$$\text{avec } c(\rho_j, \nu_i) = \begin{cases} |\rho_j| \text{ si } local(\rho_j, \nu_i) \\ \kappa \times |\rho_j| \text{ avec } \kappa > 1 \text{ sinon.} \end{cases} \tag{3}$$

Comme la collecte de ressources distantes représente un surcoût, une tâche est plus coûteuse si les ressources nécessaires sont « moins locales ». La fonction de coût peut être étendue à un ensemble de tâches :

$$\forall T \subseteq \mathcal{T},\ c(T, \nu_i) = \Sigma_{\tau \in T} c(\tau, \nu_i) \tag{4}$$

En substance, nous considérons le problème d'allocation de jobs composés de tâches situées.

Définition 4 (STAP)**.** *Un problème d'allocation de tâches situées est un quadruplet STAP $= \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c \rangle$ où :*
- *$\mathcal{D}$ est un système distribué de m nœuds ;*
- *$\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ est un ensemble de n tâches ;*
- *$\mathcal{J} = \{J_1, \ldots, J_\ell\}$ est un partitionnement des tâches en ℓ jobs ;*
- *$c : \mathcal{T} \times \mathcal{N} \mapsto \mathbb{R}^*_+$ est la fonction de coût.*

Une allocation de tâches est une répartition des tâches dans des lots ordonnés.

Définition 5 (Allocation)**.** *Une allocation pour un problème STAP à l'instant t 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* ($B_{i,t} \subseteq \mathcal{T}$) *affectées au nœud* ν_i *à l'instant t, associé à un ordre total strict* ($\prec_i \subseteq \mathcal{T} \times \mathcal{T}$). $\tau_j \prec_i \tau_k$ *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 :*

$$\forall \tau \in \mathcal{T},\ \exists \nu_i \in \mathcal{N},\ \tau \in B_{i,t} \quad (5)$$

$$\forall \nu_i \in \mathcal{N}, \forall \nu_j \in \mathcal{N} \setminus \{\nu_i\},\ B_{i,t} \cap B_{j,t} = \emptyset \quad (6)$$

Toutes les tâches sont allouées (Eq. 5) et chacune n'est allouée qu'à un seul nœud (Eq. 6). Par souci de concision, on note :

- $\overrightarrow{B_{i,t}} = (B_{i,t} \prec_i)$, le lot trié de ν_i ;
- $\min_{\prec_i} B_{i,t}$, la prochaine tâche à exécuter par ν_i.

Pour évaluer la qualité d'une allocation de tâches, nous considérons le délai moyen de réalisation (*flowtime*), qui mesure le temps moyen écoulé entre la date de libération des jobs et leur date d'achèvement.

Définition 6 (*Flowtime*)**.** *Soient STAP un problème et* $\overrightarrow{A_t}$ *une allocation à l'instant t. On définit :*

- *le délai d'attente de la tâche* τ *dans le lot* $\overrightarrow{B_{i,t}}$,

$$\Delta(\tau, \nu_i) = \sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \nu_i) \quad (7)$$

- *la durée de réalisation de la tâche* $\tau \in \mathcal{T}$ *pour l'allocation* $\overrightarrow{A_t}$,

$$C_\tau(\overrightarrow{A_t}) = \Delta(\tau, \nu(\tau, \overrightarrow{A_t})) + t - t^0_{job(\tau)} + c(\tau, \nu(\tau, \overrightarrow{A_t})) \quad (8)$$

- *la durée de réalisation de* $J \in \mathcal{J}$ *pour* $\overrightarrow{A_t}$,

$$C_J(\overrightarrow{A_t}) = \max_{\tau \in J} \{C_\tau(\overrightarrow{A_t})\} \quad (9)$$

- *le délai moyen de réalisation de* $\mathcal{J}$ *pour* $\overrightarrow{A_t}$,

$$C_{mean}(\overrightarrow{A_t}) = \frac{1}{\ell} C(\overrightarrow{A_t})$$
$$avec\ C(\overrightarrow{A_t}) = \Sigma_{J \in \mathcal{J}} C_J(\overrightarrow{A_t}) \quad (10)$$

Plus particulièrement le délai d'attente (équation 7) correspond au délai d'attente à partir de l'instant courant t avant que la tâche τ ne soit traitée. Il est à noter que les temps de réalisation, et par conséquent le *flowtime*, dépendent de l'ordre d'exécution des tâches sur chacun des nœuds.

4 Opérations

Nous formalisons ici les opérations de consommation et de réallocation de tâches.

La **consommation** d'une tâche par un nœud consiste pour ce dernier à retirer cette tâche de son lot pour l'exécuter. Cette opération mène à une nouvelle instance de problème où cette tâche a été retirée. En d'autres termes, l'accomplissement d'une tâche est un évènement disruptif qui modifie non seulement l'allocation des tâches mais également le problème sous-jacent.

Définition 7 (Consommation de tâche)**.** *Soient* $STAP = \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c \rangle$ *un problème d'allocation 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* $\overrightarrow{A_t}' = \lambda(\nu_i, \overrightarrow{B_{i,t}})$ *pour le problème* $STAP' = \langle \mathcal{D}, \mathcal{T}', \mathcal{J}', c \rangle$ *où :*

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

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

Dans ce dernier cas :

$$\forall J_j \in \mathcal{J}\ \exists J'_j \in \mathcal{J}'\ t.q\ J'_j = \begin{cases} J_j \setminus \{\min_{\prec_i} B_{i,t}\} \\ si\ job(\min_{\prec_i} B_{i,t}) = J_j \\ J_j\ sinon \end{cases}$$

et $\overrightarrow{A_t}' = (\overrightarrow{B_{1,t}}', \ldots, \overrightarrow{B_{m,t}}')$ *avec*

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

Lorsqu'une tâche est consommée, elle est retirée du problème résultant non seulement dans l'ensemble des tâches mais également du job correspondant. Ce dernier peut également être retiré si la tâche était la seule (la dernière) du job. L'allocation résultante est également modifiée. La tâche est retirée du lot où elle se trouvait.

Les tâches sont destinées à être consommées une à une jusqu'à atteindre l'allocation vide.

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

$$\Sigma_{J\in\mathcal{J}} C_J(\lambda(\nu_i, \overrightarrow{B_{i,t}})) < \Sigma_{J\in\mathcal{J}} C_J(\overrightarrow{B_{i,t}}) \quad (11)$$

Cela n'est cependant pas toujours vrai au cours du temps car les coûts effectifs des tâches peuvent être différents des coûts estimés. Si une tâche s'avère plus coûteuse que prévu lors de son exécution, le *flowtime* peut augmenter après sa consommation, comme dans l'exemple 1.

Exemple 1. *Soit le problème STAP* $= \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c\rangle$ *avec :*

- $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{E}, \mathcal{R}\rangle$*, un système distribué avec un unique nœud de calcul* $\mathcal{P} = \{\nu_1\}$ *associé au seul nœud de ressource* $\mathcal{N}_r = \{\nu_1^r\}$*, tel que* $\mathcal{E}(\nu_1, \nu_1^r) = \top$ *et une unique ressource* $\mathcal{R} = \{\rho_1\}$ *sur le nœud de ressource* ν_1^r *;*
- *deux tâches* $\mathcal{T} = \{\tau_1, \tau_2\}$ *;*
- *un unique job* $\mathcal{J} = \{J_1\}$ *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 $\overrightarrow{A_0} = (\overrightarrow{B_{1,t}})$ *avec* $\overrightarrow{B_{1,t}} = (\tau_1, \tau_2)$*. Selon l'équation 10, le flowtime est* $C_{mean}(\overrightarrow{A_0}) = C_{J_1}(\overrightarrow{A_0}) = C_{\tau_2}(\overrightarrow{A_0}) = \Delta(\tau_2, \nu_1) + t + t^0_{J_1} + c(\tau_2, \nu_1) = c(\tau_1, \overrightarrow{A_t}) + 0 + 0 + c(\tau_2, \nu_1) = 2 + 4 = 6$.

Si la consommation de τ_1 *se termine à l'instant* $t_1 = 3$*, cela signifie que cette tâche s'avère plus coûteuse que prévu lors de son exécution. Par conséquence, le flowtime de* $\overrightarrow{A_{t_1}} = (\overrightarrow{B_{\nu_1,t_1}})$ *avec* $B_{\nu_1,t_1} = (\tau_2)$ *est* $C_{mean}(\overrightarrow{A_{t_1}}) = C_{J_1}(\overrightarrow{A_{t_1}}) = t_1 + t^0_{J_1} + c(\tau_2, \nu_1) = 3 + 0 + 4 = 7 > C_{mean}(\overrightarrow{A_0})$.

Une **réallocation bilatérale** est une opération lors de laquelle une ou plusieurs tâches sont déplacées d'un lot à un autre.

Définition 8 (Réallocation bilatérale)**.** *Soit* $\overrightarrow{A_t} = (\overrightarrow{B_{1,t}}, \ldots, \overrightarrow{B_{m,t}})$ *une allocation pour le problème STAP* $= \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c\rangle$ *à l'instant t. La réallocation bilatérale de la liste non vide de tâches* T_1 *allouées au proposant* ν_i *en échange de la liste de tâches* T_2 *allouées au répondant* ν_j *dans* $\overrightarrow{A_t}$ *(*$T_1 \subseteq B_{i,t}$ *et* $T_2 \subseteq B_{j,t}$*) aboutit à l'allocation* $\gamma(T_1, T_2, \nu_i, \nu_j, \overrightarrow{A_t})$ *avec les m lots* $\gamma(T_1, T_2, \nu_i, \nu_j, \overrightarrow{B_{k,t}})$ *définis tels que :*

$$\gamma(T_1, T_2, \nu_i, \nu_j, \overrightarrow{B_{k,t}}) = \begin{cases} \overrightarrow{B_{i,t} \ominus T_1 \oplus T_2} & \text{si } k = i, \\ \overrightarrow{B_{j,t} \ominus T_2 \oplus T_1} & \text{si } k = j, \\ \overrightarrow{B_{k,t}} & \text{sinon} \end{cases} \quad (12)$$

Si T_2 *est vide, on parle de délégation. Sinon, une réallocation bilatérale est un échange de tâches.*

Nous nous restreignons ici aux échanges mais les réallocations multilatérales mériteraient d'être explorées.

Contrairement à la plupart des autres travaux (e.g. [9]), nos agents ne sont pas individuellement rationnels mais ils ont un but commun qui prime sur leur intérêt individuel : réduire le *flowtime*.

Définition 9 (Réallocation bilatérale socialement rationnelle)**.** *Soit* $\overrightarrow{A_t}$ *une allocation à l'instant t pour le problème STAP* $= \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c\rangle$*. La réallocation bilatérale* $\gamma(T_1, T_2, \nu_i, \nu_j, \overrightarrow{A_t})$ *est socialement rationnelle ssi le flowtime décroît,*

$$C(\gamma(T_1, T_2, \nu_i, \nu_j, \overrightarrow{A_t})) < C(\overrightarrow{A_t}) \quad (13)$$

Une allocation est dite **stable** s'il n'existe aucune réallocation bilatérale socialement rationnelle. Dans [4], nous avons démontré la terminaison du processus qui itère ce type de réallocations.

5 Processus

Afin d'exécuter de manière concurrente le processus de consommation et celui de réallocation, nous considérons deux types d'agents : (a) les agents de nœud, chacun d'entre eux représentant un nœud de calcul en gérant son lot de tâches (cf. section 6) ; (b) le superviseur qui synchronise les phases du processus de négociation.

Le processus de consommation se résume à l'exécution concurrente ou séquentielle des différentes tâches par les nœuds de calcul sous la supervision de leur agent. Le processus de réallocation est constitué de multiples réallocations locales qui sont le résultat de négociations bilatérales entre agents de nœud, réalisées de manière séquentielle ou concurrente. Ces processus sont complémentaires. Tandis que les consommations se déroulent en continu, les agents négocient leurs lots de tâches jusqu'à atteindre une allocation stable. La consommation d'une tâche

peut rendre instable une allocation et ainsi déclencher de nouvelles négociations. Le processus de consommation se termine quand toutes les tâches sont exécutées. L'allocation finale, qui est vide, met un terme au processus.

Il est important de noter que ce système multi-agents est intrinsèquement adaptatif. En effet, si le coût d'une tâche s'avère plus élevé que prévu, parce que le temps d'exécution a été sous-estimé ou parce que le nœud l'exécutant est ralenti, alors le processus de réallocation qui se déroule en continu permet de corriger l'allocation en prenant en compte le temps effectivement mesuré après la réalisation des tâches.

La **stratégie de consommation** des agents, détaillée dans [3], spécifie l'ordonnancement des tâches, au sein du lot de tâches, pour leur exécution par le nœud dont ils ont la charge. Afin de réduire la durée de réalisation des jobs, cette stratégie exécute les tâches des jobs les moins coûteux avant celles des jobs les plus coûteux.

La **stratégie de négociation** des agents de nœud, également détaillée dans [3], s'appuie sur un modèle des pairs, notamment une base de croyances, construit à partir des messages échangés et grâce auquel elle détermine si une réallocation est socialement rationnelle, selon les croyances de l'agent. Les agents disposent : (a) d'une stratégie d'offre qui propose des réallocations bilatérales ; (b) d'une règle d'acceptabilité qui évalue si une proposition reçue est ou non socialement rationnelle avant de l'accepter ou de la refuser ; et (c) d'une stratégie de contre-offre qui sélectionne une contre-partie à une délégation afin de proposer un échange de tâches.

Le processus de négociation se décompose en deux phases successives : (1) les agents proposent des délégations qu'ils croient socialement rationnelles et qui sont acceptées ou refusées par leurs pairs ; (2) les agents proposent des délégations qui ne sont pas nécessairement socialement rationnelles mais qui sont susceptibles de déclencher des contre-offres et ainsi des échanges socialement rationnels. Les phases de négociations s'alternent successivement, de façon concurrente à la consommation.

6 Architecture

Pour concevoir un agent de nœud, nous avons adopté une architecture modulaire qui permet la concurrence des négociations et des consommations.

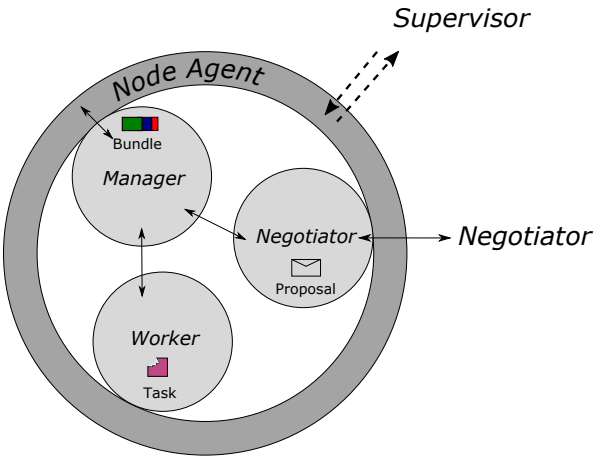

FIGURE 1 – Architecture d'un agent de nœud

Un agent de nœud est un agent composite constitué de trois agents composants (cf. figure 1), chacun ayant un rôle limité :

- le ***worker*** exécute (consomme) les tâches ;
- le ***negotiator*** maintient une base de croyances pour négocier des tâches avec ses pairs ;
- le ***manager*** gère le lot de tâches du nœud de calcul pour ordonnancer leur exécution par le *worker* en y ajoutant ou supprimant les tâches selon les réallocations bilatérales marchandées par le *negotiator*.

Afin de prioriser la consommation des tâches, le *manager*, dès qu'il est informé que le *worker* est libre, 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 négociation. Cette tâche n'est alors plus éligible pour une potentielle réallocation.

Nous représentons ici les interactions entre les agents composants sous la forme de diagrammes d'interaction où les flèches pleines représentent des appels synchrones (comme dans la figure 2b), les flèches ouvertes des messages asynchrones (comme dans la figure 2a) et les lignes pointillées des messages de réponse.

Après que le *manager* a confié au *worker* une tâche, ce dernier signale au *manager* quand elle est accomplie (cf. figure 2a). Pour prioriser la la consommation plutôt que la négociation, la demande de la prochaine tâche à accomplir par le *worker* au *manager* et prioritaire et préempte les interactions de ce dernier avec le *negotiator*. Pour raffiner son estimation du délai d'attente des tâches dans son lot, le *manager* peut demander au *worker* une estimation du temps d'exécution restant pour la tâche en cours (cf. figure 2b).

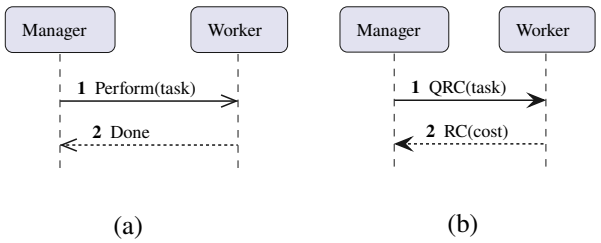

FIGURE 2 – Interactions *manager*/*worker*

Lors d'une première phase de négociation, les agents marchandent des délégations (cf. figure 3). Afin de confirmer une telle réallocation bilatérale, le *negotiator* de l'agent proposant demande de manière synchrone au *manager* de mettre à jour le lot de tâches afin que lui-même puisse mettre à jour sa base de croyances avant de s'engager dans de nouvelles négociations. Après cette confirmation, le *negotiator* du répondant en fait de même. Les étiquettes `QRC` indiquent que le *manager* interagit avec le *worker* selon le protocole de la figure 2b pour prendre en compte le temps d'exécution restant pour la tâche en cours. Lors d'une seconde phase de négociation, les agents marchandent des échanges de tâches et les interactions sont similaires.

Malgré notre architecture modulaire, la principale difficulté subsiste dans la conception des comportements des agents, qui sont spécifiés dans [5] par des automates [1], et dont la complexité est mesurée dans la table 1 en nombre d'états, de transitions et de lignes de code.

Agent	États	Transitions	Lignes
worker	2	7	173
manager	5	23	465
negotiator	9	74	1306
superviseur	9	69	626

TABLE 1 – Complexité des comportements

Le ***worker*** est : soit disponible ; soit en train d'exécuter une tâche et il peut donc estimer le temps d'exécution restant pour la tâche en cours.

Le ***manager*** gère le lot de tâches et coordonne les opérations de consommation des tâches réalisées par le *worker* avec celles de réallocations marchandées par le *negotiator*. 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 *manager* en informe le superviseur. Il continue également de distribuer les tâches à exécuter au *worker* jusqu'à ce que son lot soit vide. Informé qu'aucun agent de nœud ne détecte d'opportunité de réallocation, le superviseur enclenche le changement de phase de négociation. Enfin, le superviseur clôt le processus quand il apprend par les *managers* que toutes les tâches ont été consommées.

Le ***negotiator*** répond aux propositions de ses pairs et met à jour sa base de croyances, ce qui lui permet de détecter des opportunités de réallocation. Après avoir proposé une délégation, il attend, avant une date butoir, une acceptation, un refus ou une contre-proposition. Lorsque le *negotiator* a accepté une proposition ou fait une contre-proposition, il attend la confirmation ou l'abandon de son interlocuteur (si la tâche a 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.

7 Expérimentations

Nos expériences visent à valider que, lorsqu'elle est exécutée en continue lors du processus de consommation, la stratégie de réallocation : (1) améliore le *flowtime* ; (2) ne pénalise pas la consommation ; (3) est robuste aux aléas d'exécution (i.e. le ralentissement de nœuds). Nous présentons ici nos métriques, notre protocole expérimental et nos résultats [2].

Plutôt que le temps estimé d'exécution des tâches par les nœuds (cf. équation 3), nous considérons le **coût simulé** $c^S(\tau,\nu)$ comme le coût effectif de la réalisation de la tâche τ par le nœud ν :

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

$$c^{S_E}(\tau,\nu_i) = c(\tau,\nu_i) \tag{14}$$

- avec le ralentissement de la moitié des nœuds,

$$c^{S_H}(\tau,\nu_i) = \begin{cases} 2\times c(\tau,\nu_i) & \text{si } i \bmod 2 = 1 \\ c(\tau,\nu_i) & \text{sinon.} \end{cases} \tag{15}$$

C'est pourquoi nous distinguons :

- le ***flowtime* simulé** $C^S_{mean}(\overrightarrow{A_t})$ calculé à partir d'une allocation $\overrightarrow{A_t}$ selon les coûts simulés ;

1. https://gitlab.univ-lille.fr/maxime.morge/smastaplus/-/tree/worker/doc/specification

2. Ces expérimentations sont reproductibles à partir des instructions suivantes : https://gitlab.univ-lille.fr/maxime.morge/smastaplus/-/tree/master/doc/experiments

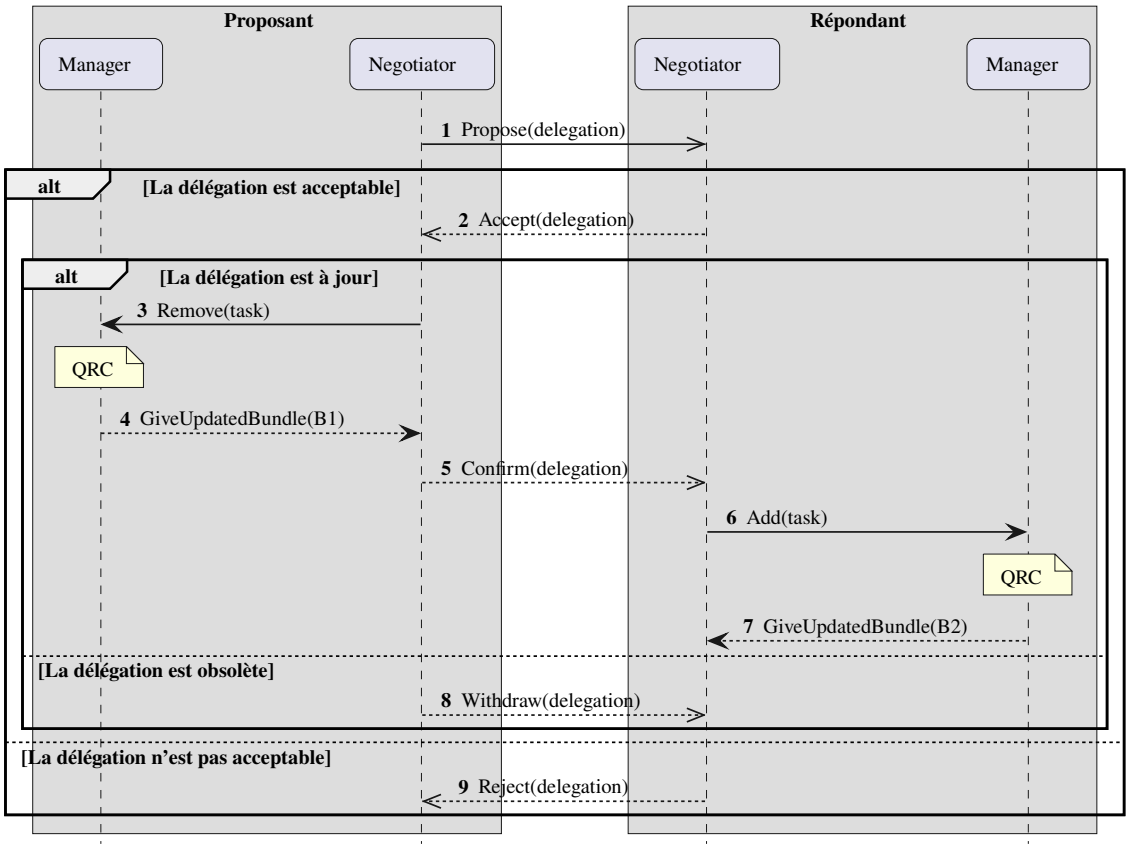

FIGURE 3 – Interactions entre le *manager* et le *negotiator* lors de la première phase de négociation

- le ***flowtime* réalisé** $C^R_{mean}(\overrightarrow{A_t})$ calculé à partir des temps de réalisation des tâches effectivement mesurés.

On définit le **taux d'amélioration de performance** :

$$\Gamma = \frac{C^R_{mean}(\overrightarrow{A_0}) - C^R_{mean}(\overrightarrow{A_e})}{C^R_{mean}(\overrightarrow{A_0})} \tag{16}$$

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.

Notre prototype [5] est implémenté avec le langage de programmation Scala et la bibliothèque Akka [12] adaptée aux applications orientées messages, fortement concurrentes, distribuées et robustes. Les expériences ont été réalisées sur une lame munie de 20 CPUs avec 512Go de RAM. Le fait que, dans nos expériences, la différence entre le *flowtime* réalisé et le *flowtime* simulé de l'allocation initiale ($C^R_{mean}(\overrightarrow{A_0}) - C^S_{mean}(\overrightarrow{A_0})$), qui mesure le coût de l'infrastructure, est négligeable, conforte ce choix technologique. Le protocole expérimental consiste, pour les différentes expériences, à générer aléatoirement 25 allocations initiales pour des problèmes STAP distincts. Nous avons fixé empiriquement $\kappa = 2$ comme une valeur réaliste pour capturer le surcoût induit par la récupération des ressources non locales dans un réseau homogène. Nous considérons $m = 8$ nœuds, $l = 4$ 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, nous considérons dans nos expérimentations qu'une réallocation bilatérale est socialement rationnelle si elle fait décroître d'au moins une seconde le *flowtime*.

Hypothèse 1 : la stratégie de réallocation améliore le *flowtime*. Nous considérons ici que les allocations initiales sont aléatoires et que les agents ont une connaissance parfaite de l'environnement d'exécution (c^{S_E}). La figure 4 montre les médianes et les écarts types de nos métriques

en fonction du nombre de tâches. Nous observons que le *flowtime* réalisé de la réallocation est meilleur que le *flowtime* réalisé de l'allocation initiale et borné par le *flowtime* simulé de la réallocation (si un oracle calcule la réallocation en temps constant). Notre stratégie améliore le *flowtime* en réallouant en continu lors du processus de consommation les tâches non locales dont la délégation réduit le coût. Le taux d'amélioration de performance (Γ) se situe entre 20 % et 37 %.

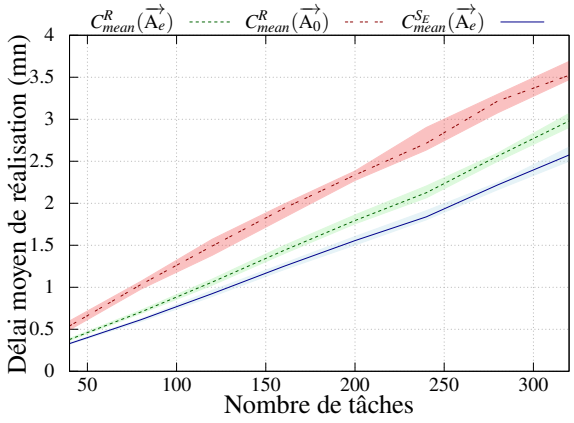

FIGURE 4 – Depuis une allocation aléatoire

Hypothèse 2 : la stratégie de réallocation ne pénalise pas la consommation. Nous considérons ici que les allocations initiales sont stables. Dans la figure 5 le *flowtime* réalisé de la réallocation est similaire au *flowtime* réalisé de l'allocation initiale. Le surcoût de la négociation est négligeable car aucune négociation n'est déclenchée lorsque les agents estiment que l'allocation est stable.

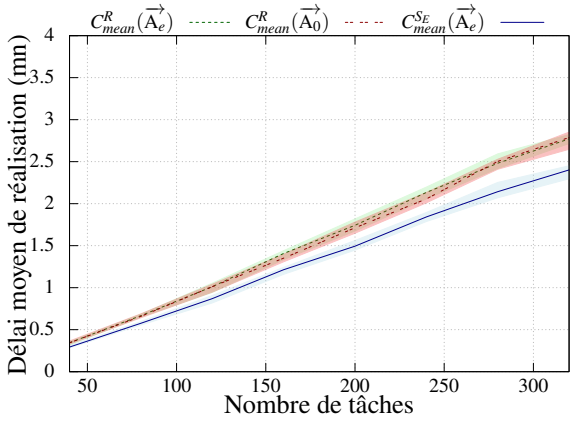

FIGURE 5 – Depuis une allocation stable

Hypothèse 3 : la stratégie de réallocation s'adapte aux aléas d'exécution. Nous considérons ici le coût effectif des tâches qui simule le ralentissement de la moitié des nœuds, c^{S_H}. Nous observons dans la figure 6 que les délais moyens de réalisation ont doublé à cause des aléas d'exécution. De plus, le *flowtime* réalisé de la réallocation reste meilleur que le *flowtime* réalisé de l'allocation initiale malgré une connaissance imparfaite de l'environnement d'exécution des agents. Prendre en compte les temps d'exécution effectifs des tâches déjà réalisées permet au taux d'amélioration de performance (Γ) de se situer entre 30 % et 37 %.

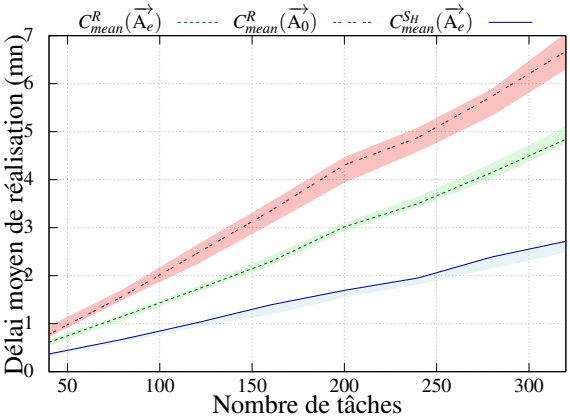

FIGURE 6 – Avec aléas d'exécution

8 Discussion

Afin de concevoir des agents autonomes qui réalisent de manière concurrente des opérations de consommation et de réallocation, nous avons proposé une architecture modulaire d'agent composé de trois agents composants : le *worker* qui exécute les tâches ; le *negotiator* qui marchande des réallocations avec ses pairs ; et le *manager* qui coordonne localement ces opérations en gérant le lot de tâches. Sans pour autant connaître l'état global du système, i.e. l'allocation, ces agents disposent de connaissances locales (e.g. la tâche courante, le lot de tâches) et de croyances qui guident leur comportement dans les interactions.

Nos expérimentations montrent que le taux d'amélioration de performance dû à notre stratégie de réallocation, lorsqu'elle est exécutée en continue lors du processus de consommation, peut atteindre 37 %. De plus, le surcoût lié aux négociations est négligeable car, lorsque l'allocation est stable, elles sont suspendues. De plus, même si un ou plusieurs nœuds sont ralentis, notre stratégie de réallocation s'adapte au contexte d'exécution en distribuant plus de tâches aux nœuds qui ne sont pas ralentis, car

elle tient compte du temps d'exécution effectif des tâches déjà réalisées, sans pour autant nécessiter de phase d'apprentissage.

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*) va au-delà de la portée de cet article, mais mériterait une étude approfondie. Nous souhaitons également évaluer la réactivité de notre stratégie face à la libération de *jobs* au fil de l'eau.

Plus généralement, nos travaux futurs porteront sur l'intégration de la réallocation des tâches dans un processus d'approvisionnement qui ajoute ou supprime des nœuds de calcul au cours de l'exécution en fonction des besoins des utilisateurs afin de proposer une stratégie multi-agents élastique de passage à l'échelle.

Remerciements. Nous adressons nos remerciements aux relecteurs pour leur travail minutieux et leurs précieux conseils.

Références

[1] Quentin Baert, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier et Kostas Stathis. « Un système multi-agent adaptatif pour la réallocation de tâches au sein d'un job MapReduce ». In : *Revue Ouverte d'Intelligence Artificielle* (2022), p. 557-585.

[2] Ellie Beauprez, Luc Bigand, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. « Réaffectation de tâches de la théorie à la pratique : état de l'art et retour d'expérience ». In : *Actes des JFSMA*. Cépaduès, 2021, p. 51-60.

[3] Ellie Beauprez, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. « Échange de tâches pour la réduction de la durée moyenne de réalisation ». In : *Actes des JFSMAs*. Cépaduès, 2022, p. 19-28.

[4] Ellie Beauprez, Anne-Cécile Caron, Maxime Morge et Jean-Christophe Routier. « Délégation de lots de tâches pour la réduction de la durée moyenne de réalisation ». In : *ROIA* (2023). À paraître, p. 1-29.

[5] Ellie Beauprez et Maxime Morge. *Scala implementation of the Extended Multi-agents Situated Task Allocation*. `https://gitlab.univ-lille.fr/maxime.morge/smastaplus`. 2020.

[6] Yin Chen, Xinjun Mao, Fu Hou, Qiuzen Wang et Shuo Yang. « Combining re-allocating and re-scheduling for dynamic multi-robot task allocation ». In : *Proc. of SMC*. 2016, p. 395-400.

[7] Han-Lim Choi, Luc Brunet et Jonathan P How. « Consensus-based decentralized auctions for robust task allocation ». In : *IEEE transactions on robotics* 25.4 (2009), p. 912-926.

[8] Niall Creech, Natalia Criado Pacheco et Simon Miles. « Resource allocation in dynamic multiagent systems ». In : *CoRR* abs/2102.08317 (2021).

[9] Anastasia Damamme, Aurélie Beynier, Yann Chevaleyre et Nicolas Maudet. « The Power of Swap Deals in Distributed Resource Allocation ». In : *Proc. of AAMAS*. 2015, p. 625-633.

[10] J. Dean et S. Ghemawat. « MapReduce : Simplified Data Processing on Large Clusters ». In : *Proc. of OSDI*. 2004, p. 137-150.

[11] Kristina Lerman, Chris Jones, Aram Galstyan et Maja J Matarić. « Analysis of dynamic task allocation in multi-robot systems ». In : *The International Journal of Robotics Research* 25.3 (2006), p. 225-241.

[12] Lightbend. *Akka is the implementation of the Actor Model on the JVM*. `http://akka.io`. 2020.

[13] Siddharth Mayya, Diego S D'antonio, David Saldaña et Vijay Kumar. « Resilient task allocation in heterogeneous multi-robot systems ». In : *IEEE Robotics and Automation Letters* 6.2 (2021), p. 1327-1334.

[14] G Ayorkor Mills-Tettey, Anthony Stentz et M Bernardine Dias. « The dynamic hungarian algorithm for the assignment problem with changing costs ». In : *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27* (2007).

[15] Changjoo Nam et Dylan A. Shell. « When to do your own thing : Analysis of cost uncertainties in multi-robot task allocation at run-time ». In : *Proc. of ICRA*. 2015, p. 1249-1254.