Négociation pour la consommation adaptative d'allocation continue

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

▶ **To cite this version:**

Ellie Beauprez, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier. Négociation pour la consommation adaptative d'allocation continue. Revue Ouverte d'Intelligence Artificielle, 2024, Post-actes des Journées Francophones sur les Systèmes Multi-Agents (JFSMA 2023), 5 (4), pp.9-35. 10.5802/roia.85 . hal-04826283

HAL Id: hal-04826283
https://hal.science/hal-04826283v1

Submitted on 9 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

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

Négociation pour la consommation adaptative d'allocation continue

Volume 5, nº 4 (2024), p. 9-35.

https://doi.org/10.5802/roia.85

© Les auteurs, 2024.

Cet article est diffusé sous la licence
Creative Commons Attribution 4.0 International License.
http://creativecommons.org/licenses/by/4.0/

La Revue Ouverte d'Intelligence Artificielle est membre du Centre Mersenne pour l'édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : 2967-9672

Revue Ouverte d'Intelligence Artificielle
Volume 5, n° 4, 2024, 9-35

Négociation pour la consommation adaptative d'allocation continue

Ellie Beauprez[a], Anne-Cécile Caron[a], Maxime Morge[b], Jean-Christophe Routier[a]

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

[b] Univ Lyon, UCBL, CNRS, INSA Lyon, UMR 5205 LIRIS, F-59622 Villeurbanne, France

E-mail : Ellie.Beauprez@univ-lille.fr, Anne-Cecile.Caron@univ-lille.fr, Maxime.Morge@univ-lyon1.fr, Jean-Christophe.Routier@univ-lille.fr.

RÉSUMÉ. — Nous étudions ici le problème de l'allocation continue 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 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, Architecture d'agent.

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 [11].

Nous nous intéressons à l'allocation de plusieurs jobs concurrents, soumis au fil de l'eau par différents utilisateurs et pouvant arriver à des dates différentes. L'objectif est de minimiser le délai moyen de réalisation de ces jobs, appelée *flowtime*, 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 nœuds 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 onéreuse en raison du temps nécessaire pour collecter les ressources.

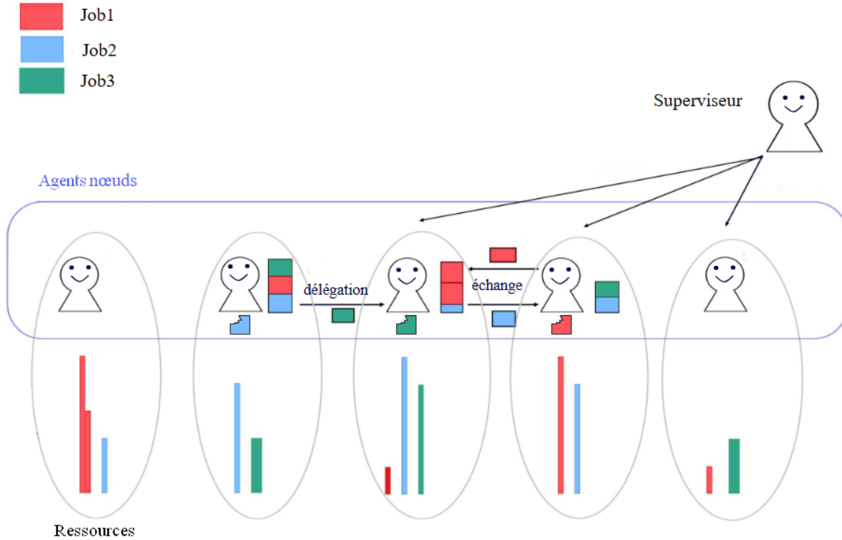


FIGURE 1.1 – Vue d'ensemble de l'architecture multi-agents où un agent superviseur surveille l'exécution et les échanges des tâches (rectangles) réalisés par des agents-nœuds pour lesquels les ressources (barres) éventuellement répliquées sont locales (dans l'ellipse) ou distantes.

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 [2].

Nous proposons dans cet article 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 (cf. figure 1.1). 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.

Au delà de nos travaux préliminaires exposés dans [4], nos contributions sont les suivantes.

(1) Nous introduisons un cadre formel pour l'allocation continue de tâches situées qui sont traitées progressivement et où les jobs sont soumis au fil de l'eau. De plus, nous introduisons un critère de rationalité des échanges locaux qui garantit la terminaison du processus.
(2) Nous proposons une architecture d'agent composite qui permet la co-occurence du processus de réallocation avec celui de la consommation. 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.
(3) Nos 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 [13], notre stratégie de réallocation ne pénalise pas la consommation et peut améliorer le délai moyen de réalisation jusqu'à 29 %. 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.

Après un aperçu des travaux connexes dans la section 2, nous rappelons dans la section 3 la formalisation du problème d'allocation continue 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. Nous esquissons notre architecture d'agent dans la section 6 et nous détaillons le comportement des agents-nœuds en particulier celui de négociation dans la section 7. La section 8 analyse nos résultats expérimentaux. La section 9 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 continus qui s'ajustent en permanence aux changements de l'environnement d'exécution ou de performance des exécutants [14].

La plupart de ces travaux s'appuie sur des mécanismes d'enchère pour répartir les tâches (*e.g.* [13, 8, 1]). Les agents font des offres pour des tâches uniques ou des lots de tâches et les gagnants sont déterminés en fonction de la valeur des offres. Une méthode d'allocation fondée sur des enchères séquentielles à un seul article – *Sequential Single-Item (SSI) auctions*, consiste à considérer initialement les tâches comme non-allouées. Un unique commissaire-priseur annonce les tâches une par une. Les enchérisseurs émettent une offre en connaissant l'allocation des tâches précédentes. Le processus est répété jusqu'à ce que toutes les tâches soit allouées. De manière alternative, chaque exécutant enchérit sur des lots de tâches dans une enchère combinatoire à un seul tour. Toutefois, il convient de noter que, dans une telle enchère combinatoire, le calcul par les enchérisseurs de leurs offres est NP-difficile car le nombre de lots possibles est exponentiel selon le nombre de tâches. De même, le problème de détermination du gagnant est soit de taille exponentielle, soit NP-difficile. Une méthode de réallocation fondée sur des enchères parallèles à un seul article – *Parallel Single-Item (PSI) auctions* consiste à considérer autant de commissaires-priseurs que d'exécutants qui annonce le lot de tâches qu'il lui est assigné. La complexité des méthodes fondées sur SSI et PSI est polynomiale selon le nombre de tâches et d'exécutants, mais, contrairement à la méthode fondée sur PSI, celle qui s'appuie sur SSI offre des garanties sur la qualité de la solution [13].

Contrairement aux méthodes centralisées précédentes, l'algorithme à base de consensus (CBBA – *Consensus Based Bundle Algorithm*) [8] est une méthode multi-agents d'allocation en deux phases : (a) un processus d'enchère où chaque agent sélectionne les tâches sur lesquelles il souhaite placer une offre ; (b) un processus décentralisé de résolution des conflits fondé sur un consensus parmi les agents portant sur les valeurs des offres gagnantes. L'objectif consiste à minimiser le débit de réalisation (*throughput*), *i.e.* une mesure de la performance du système du point de vue des exécutants qui est agnostique vis-à-vis de l'ordonnancement.

Dans [1], un processus multi-enchères permet à un agent de participer à plusieurs négociations concurrentes dans le but d'améliorer le processus de réallocation des tâches. Ici, 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 permet la concurrence de multiples marchangages qui améliore la réactivité (*responsiveness*) de la méthode. 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 [7]. 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 [7], 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 [18], des méthodes incrémentales pour réparer l'allocation optimale initiale lorsque les coûts changent [17] ou l'optimisation combinatoire pour exploiter les mesures de dégradation [16]. 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 [9]. 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 obsolètes.

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 [5]. 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 du problème d'allocation continue de jobs concurrents composés de tâches situées. Nous considérons ici un problème dynamique où les tâches sont peu à peu consommées et les jobs sont soumis au fil de l'eau.

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 3.1 (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 ;*

- $\mathcal{E} : \mathcal{P} \times \mathcal{N}_r \rightarrow \{\top, \perp\}$ *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 :*

$$l\ : \mathcal{R} \longrightarrow 2^{\mathcal{N}_r} \tag{3.1}$$

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 :

$$\forall \nu_c \in \mathcal{P}, \forall \rho \in \mathcal{R},\ \text{local}(\rho, \nu_c) \text{ ssi } \exists \nu_r \in l(\rho) \text{ t.q. } \mathcal{E}(\nu_c, \nu_r) \tag{3.2}$$

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

Libérés au cours temps, les jobs sont des ensembles de tâches sans date butoir, 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 consiste à exécuter l'ensemble de ses tâches pour produire un résultat.

Définition 3.2 (Job/Tâche). — *Soit* $\mathcal{D}$ *un système distribué. On considère un ensemble de* ℓ *jobs* $\mathcal{J} = \{J_1, \ldots, J_\ell\}$*. Chaque job* $J_i \in \mathcal{J}$*, 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* $\mathcal{T} = \cup_{1 \leqslant i \leqslant \ell} J_i$ *l'ensemble des n tâches sous-jacentes à* $\mathcal{J}$*.*

Comme les jobs sont libérés à des dates différentes, on peut s'intéresser aux jobs libérés avant un temps donné t. Ainsi, $\mathcal{J}_t$ est l'ensemble des ℓ_t jobs libérés avant l'instant t ($\forall \mathrm{J} \in \mathcal{J}_t,\ t^0_{\mathrm{J}} \leqslant t$) et $\mathcal{T}_t$ est l'ensemble des tâches sous-jacentes à $\mathcal{J}_t$. $\mathcal{R}_\tau \subseteq \mathcal{R}$ dénote l'ensemble des ressources requises pour la tâche τ. Par souci de concision, on note $\text{job}(\tau)$ le job contenant la tâche τ. Nous faisons l'hypothèse que le nombre de jobs est négligeable par rapport au nombre de tâches, $|\mathcal{J}| \ll |\mathcal{T}|$.

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

Définition 3.3 (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 :*

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

$$\textit{avec } c(\rho_j, \nu_i) = \begin{cases} |\rho_j| & \textit{si local}(\rho_j, \nu_i) \\ \kappa \times |\rho_j| \textit{ avec } \kappa > 1 & \textit{sinon.} \end{cases} \tag{3.3}$$

Comme la collecte de ressources distantes représente un surcoût, une tâche est plus coûteuse si les ressources nécessaires sont « moins locales ». La valeur de κ sera discutée dans la section 8.

La fonction de coût peut être étendue à un ensemble de tâches :

$$\forall \mathrm{T} \subseteq \mathcal{T},\ \mathrm{c}(\mathrm{T}, \nu_i) = \Sigma_{\tau \in \mathrm{T}} \mathrm{c}(\tau, \nu_i) \tag{3.4}$$

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

Définition 3.4 (STAP). — *Un problème d'allocation continue de tâches situées à l'instant t est un quadruplet STAP =* $\langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ *où :*

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

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

Définition 3.5 (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* ($B_{i,t} \subseteq \mathcal{T}_t$) *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 :*

$$\forall \tau \in \mathcal{T}_t,\ \exists \nu_i \in \mathcal{N},\ \tau \in B_{i,t} \tag{3.5}$$

$$\forall \nu_i \in \mathcal{N}, \forall \nu_j \in \mathcal{N} \setminus \{\nu_i\},\ B_{i,t} \cap B_{j,t} = \emptyset \tag{3.6}$$

Toutes les tâches des jobs libérés sont allouées (3.5) et chacune n'est allouée qu'à un seul nœud (3.6). Par souci de concision, on note :

- $\overrightarrow{\mathrm{B}_{i,t}} = (\mathrm{B}_{i,t} \prec_i)$, le lot trié de ν_i ;
- $\min_{\prec_i} \mathrm{B}_{i,t}$, 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 3.6 (*Flowtime*). — *Soient STAP un problème et* $\overrightarrow{A_t}$ *une allocation à l'instant t. On définit :*

- *le délai d'attente d'une tâche* $\tau \in \mathcal{T}_t$ *dans le lot* $\overrightarrow{B_{i,t}}$,

$$\Delta(\tau, \nu_i) = \sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \nu_i) \tag{3.7}$$

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

$$C_\tau(\overrightarrow{A_t}) = \Delta(\tau, \nu(\tau, \overrightarrow{A_t})) + t - t^0_{\mathrm{job}(\tau)} + c(\tau, \nu(\tau, \overrightarrow{A_t})) \tag{3.8}$$

- *la durée de réalisation d'un job* $J \in \mathcal{J}_t$ *pour* $\overrightarrow{A_t}$,

$$C_J(\overrightarrow{A_t}) = \max_{\tau \in J} \{C_\tau(\overrightarrow{A_t})\} \tag{3.9}$$

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

$$C_{\text{mean}}(\overrightarrow{A_t}) = \frac{1}{\ell_t} C(\overrightarrow{A_t}) \quad avec \quad C(\overrightarrow{A_t}) = \Sigma_{J \in \mathcal{J}_t} C_J(\overrightarrow{A_t}) \tag{3.10}$$

Plus particulièrement le délai d'attente (3.7) 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 locales de transformation des allocations : les consommations et les réallocations de tâches. Ces opérations élémentaires sont combinées dans des processus distribués et concurrents : le processus de consommation et le processus de réallocation.

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 4.1 (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 à l'instant t. La consommation par un nœud consommateur* ν_i *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* $STAP' = \langle \mathcal{D}, \mathcal{T}_t', \mathcal{J}_t', c \rangle$ *où :*

$$\mathcal{T}_t' = \mathcal{T}_t \setminus \{\min_{<_i} B_{i,t}\} \tag{4.1}$$

$$\mathcal{J}_t' = \begin{cases} \mathcal{J}_t \setminus \{\text{job}(\min_{<_i} B_{i,t})\} & si \ \text{job}(\min_{<_i} B_{i,t}) = \{\min_{<_i} B_{i,t}\} \\ \mathcal{J}_t & sinon \end{cases} \tag{4.2}$$

Dans ce dernier cas :

$$\forall J_j \in \mathcal{J} \ \exists J_j' \in \mathcal{J}' \ t.q \ J_j' = \begin{cases} J_j \setminus \{\min_{<_i} B_{i,t}\} & si \ \text{job}(\min_{<_i} B_{i,t}) = J_j \\ J_j & sinon \end{cases} \tag{4.3}$$

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

$$\overrightarrow{B_{j,t}}' = \begin{cases} \overrightarrow{B_{i,t} \ominus \min_{<_i} B_{i,t}} & si \ j = i \\ \overrightarrow{B_{j,t}} & sinon \end{cases} \tag{4.4}$$

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.

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

$$\Sigma_{\mathrm{J} \in \mathcal{J}_t} \mathrm{C}_{\mathrm{J}}(\lambda(\nu_i, \overrightarrow{\mathrm{B}_{i,t}})) < \Sigma_{\mathrm{J} \in \mathcal{J}_t} \mathrm{C}_{\mathrm{J}}(\overrightarrow{\mathrm{B}_{i,t}}) \quad (4.5)$$

Cela n'est cependant pas toujours vrai en pratique. En effet, comme nous le verrons dans la section 8, les coûts effectifs des tâches peuvent être différents des coûts estimés (cf. définition 3.3). Si une tâche s'avère plus coûteuse que prévu lors de son exécution, le *flowtime* peut augmenter après sa consommation, comme dans l'exemple 4.2.

Exemple 4.2. — Soit le problème STAP = $\langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c\rangle$ avec :

- $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{E}, \mathcal{R}\rangle$, 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 $\mathcal{T}_t = \{\tau_1, \tau_2\}$;
- un unique job $\mathcal{J}_t = \{\mathrm{J}_1\}$ libéré à $t^0_{\mathrm{J}_1} = 0$ composé des deux tâches $\mathrm{J}_1 = \{\tau_1, \tau_2\}$;
- la fonction de coût c telle que $\mathrm{c}(\tau_1, \nu_1) = 2$ et $\mathrm{c}(\tau_2, \nu_1) = 4$.

L'allocation $\overrightarrow{\mathrm{A}_0} = (\overrightarrow{\mathrm{B}_{1,t}})$ avec $\overrightarrow{\mathrm{B}_{1,t}} = (\tau_1, \tau_2)$. Selon l'équation 3.10, le *flowtime* est $\mathrm{C}_{\mathrm{mean}}(\overrightarrow{\mathrm{A}_0}) = \mathrm{C}_{\mathrm{J}_1}(\overrightarrow{\mathrm{A}_0}) = \mathrm{C}_{\tau_2}(\overrightarrow{\mathrm{A}_0}) = \Delta(\tau_2, \nu_1) + t + t^0_{\mathrm{J}_1} + \mathrm{c}(\tau_2, \nu_1) = \mathrm{c}(\tau_1, \overrightarrow{\mathrm{A}_t}) + 0 + 0 + \mathrm{c}(\tau_2, \nu_1) = 2 + 4 = 6$.
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 $\overrightarrow{\mathrm{A}_{t_1}} = (\overrightarrow{\mathrm{B}_{\nu_1,t_1}})$ avec $\overrightarrow{\mathrm{B}_{\nu_1,t_1}} = (\tau_2)$ est $\mathrm{C}_{\mathrm{mean}}(\overrightarrow{\mathrm{A}_{t_1}}) = \mathrm{C}_{\mathrm{J}_1}(\overrightarrow{\mathrm{A}_{t_1}}) = t_1 + t^0_{\mathrm{J}_1} + \mathrm{c}(\tau_2, \nu_1) = 3 + 0 + 4 = 7 > \mathrm{C}_{\mathrm{mean}}(\overrightarrow{\mathrm{A}_0})$.

Une **réallocation bilatérale** est une opération élémentaire lors de laquelle une ou plusieurs tâches sont déplacées. Elle consiste en un échange local de tâches entre deux exécutants.

DÉFINITION 4.3 (Réallocation bilatérale). — *Soit $\overrightarrow{A_t} = (\overrightarrow{B_{1,t}}, \dots, \overrightarrow{B_{m,t}})$ une allocation pour le problème STAP = $\langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c\rangle$ 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 $\overrightarrow{A_t}$ ($\mathrm{T}_1 \subseteq B_{i,t}$ et $\mathrm{T}_2 \subseteq B_{j,t}$) aboutit à l'allocation $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})$ avec les m lots $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{B_{k,t}})$ définis tels que :*

$$\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{B_{k,t}}) = \begin{cases} \overrightarrow{B_{i,t} \ominus \mathrm{T}_1 \oplus \mathrm{T}_2} & si\ k = i, \\ \overrightarrow{B_{j,t} \ominus \mathrm{T}_2 \oplus \mathrm{T}_1} & si\ k = j, \\ \overrightarrow{B_{k,t}} & sinon \end{cases} \quad (4.6)$$

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

Comme nous privilégions un protocole de négociation bilatérale plutôt qu'un protocole d'enchère, nous nous restreignons ici aux échanges bilatéraux et nous excluons les

réallocations multilatérales. Par la suite, $\delta(\tau, \nu_i, \nu_j, \overrightarrow{\mathrm{A}_t})$ et $\sigma(\tau_1, \tau_2, \nu_i, \nu_j, \overrightarrow{\mathrm{A}_t})$ dénotent respectivement une délégation unaire et un échange unaire.

Contrairement à la plupart des autres travaux (*e.g.* [10]), nos agents ne sont pas individuellement rationnels mais ils ont un but commun qui prime sur leur intérêt individuel : réduire le *flowtime* de l'allocation courante.

Définition 4.4 (Réallocation bilatérale socialement rationnelle). — *Soit $\overrightarrow{A_t}$ une allocation pour le problème STAP = $\langle \mathcal{D}, \mathcal{T}_t, \mathcal{J}_t, c \rangle$ à l'instant t. La réallocation bilatérale $\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})$ est socialement rationnelle si et seulement si cette réallocation fait décroître le* flowtime*,*

$$C(\gamma(\mathrm{T}_1, \mathrm{T}_2, \nu_i, \nu_j, \overrightarrow{A_t})) < C(\overrightarrow{A_t}) \tag{4.7}$$

Une allocation est dite **stable** s'il n'existe aucune réallocation bilatérale socialement rationnelle. Dans [5], nous avons démontré la terminaison du processus qui itère ce type de réallocations. La preuve est identique à celles dans [12, 1].

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-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-nœud, réalisées de manière séquentielle ou concurrente. Ces processus sont complémentaires. Tandis que les consommations se déroulent au fur et à mesure, 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 permet progressivement 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-nœud, détaillée dans [5], 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 stratégie d'acceptabilité qui évalue si toute ou partie d'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. 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 5.1. Tant que l'un des agents-nœuds a encore des tâches à consommer, le superviseur re-déclenche la première phase à l'issue de la deuxième phase. 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. section 8) puis stoppe le processus lorsque toutes les tâches ont été consommées.

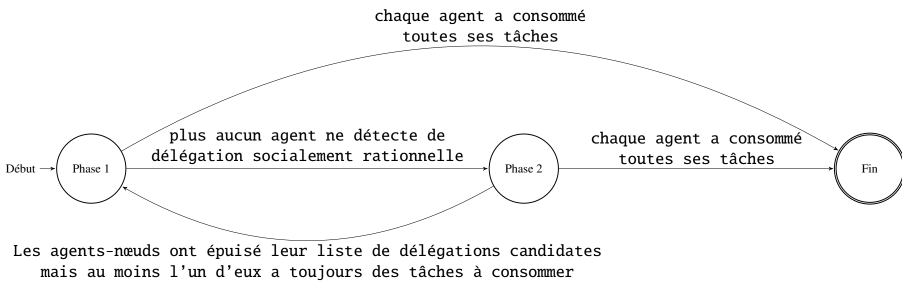


FIGURE 5.1 – Comportement global du système cadencé par le superviseur

6. ARCHITECTURE

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

Un agent-nœud est un agent composite constitué de trois agents composants (cf. figure 6.1), chacun ayant un rôle limité :

- l'***exécutant*** consomme les tâches ;
- le ***négociateur*** s'appuie sur un modèle des pairs pour négocier des tâches ;

- 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. La stratégie de consommation détermine dans quel ordre il va confier les tâches à l'exécutant pour leur consommation.

Afin de prioriser la consommation des tâches, le gestionnaire, dès qu'il est informé que l'exécutant est libre, fournit à ce dernier la prochaine tâche à exécuter, quitte à annuler la réallocation de cette tâche en cours de négociation. Cette tâche n'est alors plus éligible pour une potentielle réallocation. À la différence de l'architecture proposée dans [1], le modèle des pairs (y compris de lui-même) n'est pas associé au gestionnaire mais au négociateur pour qu'il soit autonome vis-à-vis du gestionnaire dans la conduite de l'ensemble des fils de négociations bilatérales.

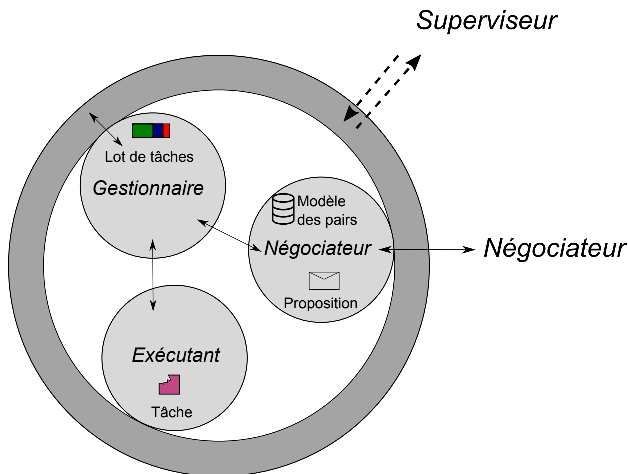
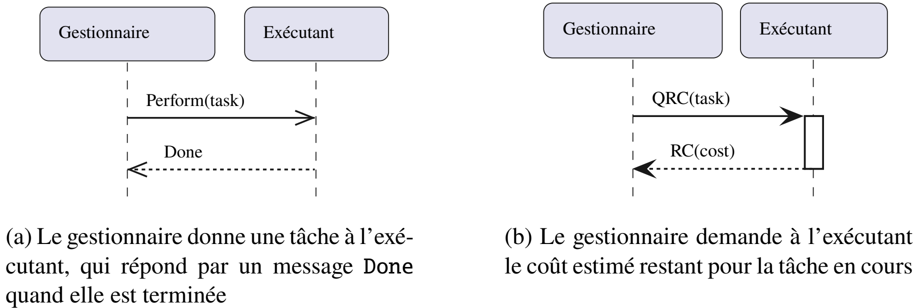


FIGURE 6.1 – Architecture composite d'un agent-nœud qui sert d'interface entre ses agents composants et les agents composants des autres nœuds et d'intermédiaire avec l'agent superviseur

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 6.2b), les flèches ouvertes des messages asynchrones (comme dans la figure 6.2a) et les lignes pointillées des messages de réponse.

Après que le gestionnaire a confié à l'exécutant une tâche, ce dernier signale au gestionnaire quand elle est accomplie (cf. figure 6.2a). Pour prioriser la consommation plutôt que la négociation, la demande de la prochaine tâche à accomplir par l'exécutant au gestionnaire est prioritaire et préempte les interactions de ce dernier avec le négociateur. Pour raffiner son estimation du délai d'attente des tâches dans son lot, le gestionnaire peut demander à l'exécutant une estimation du temps d'exécution restant pour la tâche en cours – `QRC(task)`, *Query Remaining Cost* (cf. figure 6.2b).



(a) Le gestionnaire donne une tâche à l'exécutant, qui répond par un message `Done` quand elle est terminée

(b) Le gestionnaire demande à l'exécutant le coût estimé restant pour la tâche en cours

FIGURE 6.2 – Interactions entre le gestionnaire et l'exécutant

Les interactions entre le gestionnaire et le négociateur lors de la première phase de négociation dédiée aux délégations, sont représentées sur la figure 6.3 dans le cas d'une délégation unaire. Pour opérer une réallocation bilatérale, le négociateur demande de façon synchrone à son gestionnaire de mettre à jour le lot de tâches afin de retirer ou d'ajouter les tâches réallouées. La réponse du gestionnaire lui permet de mettre à jour son modèle des pairs pour pouvoir confirmer la transaction et s'engager

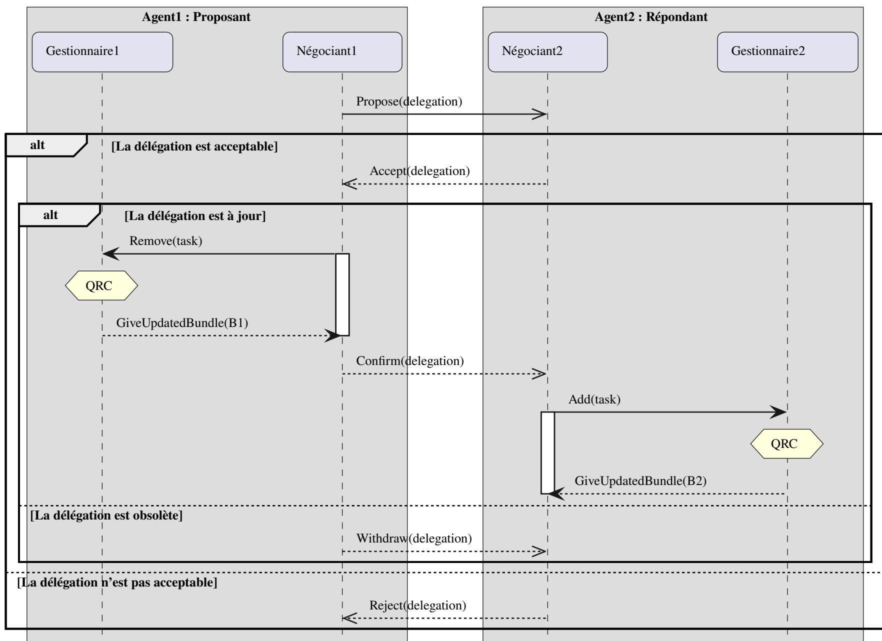


FIGURE 6.3 – Interactions entre le gestionnaire et le négociateur lors de la première phase de négociation pour la délégation d'une unique tâche. Les étiquettes QRC font référence à l'interaction de la figure 6.2b

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. Le gestionnaire peut donc ensuite répondre au négociateur pour lui transmettre les nouveaux coûts estimés des jobs pour le lot de tâches à l'aide d'un message `GiveUpdatedBundle`.

Lors d'une seconde phase de négociation, les agents marchandent des échanges de tâches. Les interactions entre le gestionnaire et le négociateur, représentées sur la figure 6.4 sont similaires à celles dans le cas des délégations. Toutefois, la concurrence

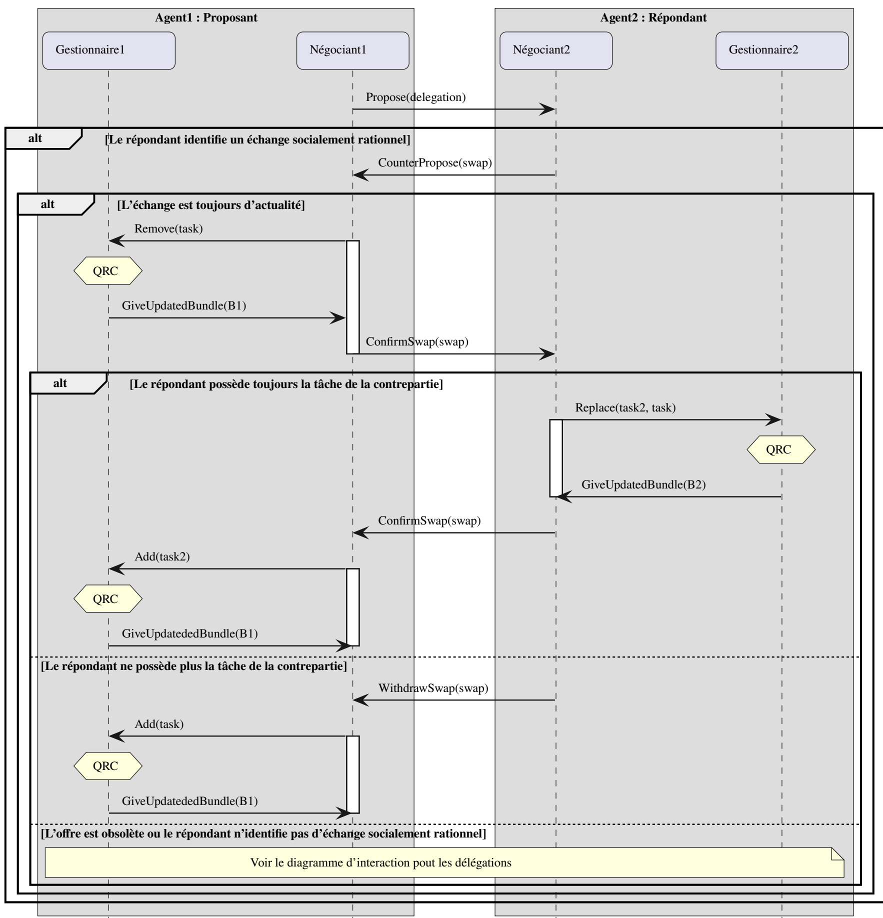


FIGURE 6.4 – Interactions entre le gestionnaire et le négociateur lors de la deuxième phase de négociation. Dans le cas où le répondant n'identifie pas d'échange socialement rationnel, les interactions sont les mêmes que dans le cas des délégations

du processus de réallocation avec le processus de consommation nécessite 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.

7. Comportements

Notre modèle de comportement d'agent est une machine à états finis où chaque transition est étiquetée 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 ou la mise à jour de l'état mental de l'agent. L'implémentation du prototype dans [6] est la directe mise en œuvre de la spécification sous la forme d'automates[(1)]

L'exécutant est soit disponible lorsqu'il attend de recevoir une tâche à consommer, soit en train d'exécuter une tâche et il peut alors estimer le temps d'exécution restant pour la tâche en cours.

Le 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-nœuds, le gestionnaire en informe le superviseur. Il continue également à confier à l'exécutant des tâches à réaliser jusqu'à ce que son lot soit vide. Informé qu'aucun agent-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 gestionnaires que toutes les tâches ont été consommées.

Le négociateur répond aux propositions et met à jour son modèle des pairs, ce qui lui permet de détecter des opportunités de réallocation. Conformément aux diagrammes d'interactions 6.3 et 6.4, après avoir proposé une délégation, il attend, avant une date butoir, une acceptation, un refus ou une contre-proposition.
L'essentiel du comportement du négociateur, noté ν_i, est représenté de manière simplifiée dans la figure 7.1.

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

(1) `https://gitlab.univ-lille.fr/maxime.morge/smastaplus/-/tree/master/doc/specification`.

v_j:ConfirmSwap($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_i})$)
$\Rightarrow$ manager ? Add(τ_{swap}); v_i ! Next

v_j:WithdrawSwap($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_i})$)
$\Rightarrow$ manager ? Add(τ); v_i ! Next

v_j:Propose($\delta(\tau, v_j, v_i, \overrightarrow{A_j})$) et l'offre n'est pas acceptable
et il n'y a pas de contre-offre acceptable au cours de la seconde phase
$\Rightarrow$ v_j ! Reject($\delta(\tau, v_j, v_i, \overrightarrow{A_j})$); v_i ! Next

Après initialisation → Répondant

v_j:Reject(O_i)
$\Rightarrow$ v_i ! Next

v_j:Accept($\delta(\tau, v_i, v_j, \overrightarrow{A_j})$) ou v_j:CounterPropose($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_j})$)
et (l'offre n'est plus valide ou la contre-offre n'est pas acceptable)
$\Rightarrow$ v_j ! Withdraw($\delta(\tau, v_i, v_j, \overrightarrow{A_j})$); v_i ! Next

v_j:Propose(O_j)
$\Rightarrow$ v_i forward (v_j,Propose(O_j))

v_j:ConfirmSwap($\sigma(\tau, \tau_{swap}, v_j, v_i, \overrightarrow{A_i})$)
et l'offre n'est plus valide
$\Rightarrow$ manager ! WithdrawSwap($\sigma(\tau, \tau_{swap}, v_j, v_i, \overrightarrow{A_i})$);
v_i ! Next

v_j:Accept($\delta(\tau, v_i, v_j, \overrightarrow{A_i})$) et l'offre est toujours valide
$\Rightarrow$ manager ? Remove(τ); v_j ! Confirm($\delta(\tau, v_i, v_j, \overrightarrow{A_i})$);
v_i ! Next

Troqueur

v_j:ConfirmSwap($\sigma(\tau, \tau_{swap}, v_j, v_i, \overrightarrow{A_i})$)
et l'offre est toujours valide
$\Rightarrow$ manager ? Replace(τ_{swap}, τ);
v_j ! ConfirmSwap($\sigma(\tau, \tau_{swap}, v_j, v_i, \overrightarrow{A_i})$);
v_i ! Next

v_j:CounterPropose($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_i})$)
et l'offre est toujours valide et la contre-offre est acceptable
$\Rightarrow$ manager ? Remove(τ);
v_j ! ConfirmSwap($\sigma(\tau, \tau_{swap}, v_i, v_j, \overrightarrow{A_i})$)

v_j:Withdraw(O_j)
$\Rightarrow$ v_i ! Next

v_i:Next et $\overline{O_i} = \theta$
$\Rightarrow$ v_i ! Close

v_j:Propose($\delta(\tau, v_j, v_i, \overrightarrow{A_j})$) et $\tau \notin B_{i,t}$
et une contre-offre est acceptable cours de la seconde phase
$\Rightarrow$ v_j ! CounterPropose($O_i(v_j, \tau)$)

(v_i:Next et $\overline{O_i} \neq \theta$) ou v_i:WakeUp(O_i)
$\Rightarrow$ recipient(O_i) ! Propose(O_i)

v_j:Confirm($\delta(\tau, v_j, v_i, \overrightarrow{A_i})$)
$\Rightarrow$ manager ? Add(τ); v_i ! Next

v_i:Next et $\overline{O_i} \neq \theta$
$\Rightarrow$ v_i ! WakeUp(O_i)

v_j:Propose($\delta(\tau, v_j, v_i, \overrightarrow{A_j})$) et (l'offre est acceptable
au cours de le première phase ou il n'y a pas de contre-offre acceptable)
$\Rightarrow$ v_j ! Accept($\delta(\tau, v_j, v_i, \overrightarrow{A_j})$)

Contractant

Pause

Proposant

manager:StartStage
$\Rightarrow$ v_i ! Next

v_i:Next et $\overline{O_i} = \theta$
$\Rightarrow$ v_i ! Close

v_i:Close
$\Rightarrow$ manager ! FinishedStage(metrics)

v_j:message $\Rightarrow$ v_i forward (v_j,message);
manager ! Restart

manager:ReadyForNextStage

FinDePhase

Transistion

FIGURE 7.1 – Comportement simplifié du négociateur

Dans l'état `Contractant`, 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 (ν_j`:Confirm(`$\delta(\tau, \nu_j, \nu_i, \overrightarrow{\mathtt{A}_t})$`)`), il demande au gestionnaire d'ajouter la tâche au lot (`manager?Add(`τ`)`). Lorsqu'il reçoit la confirmation d'un échange toujours valide (ν_j`:ConfirmSwap(`$\sigma(\tau, \tau_{\text{swap}}, \nu_j, \nu_i, \overrightarrow{\mathtt{A}_t})$`)`), il demande au gestionnaire de remplacer la tâche déléguée par sa contrepartie (`manager?Replace(`τ_{swap}`,`τ`)`) 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 `Répondant`.

Dans l'état `Proposant`, si le négociateur reçoit une acceptation (ν_j`:Accept(`$\delta(\tau, \nu_i, \nu_j, \overrightarrow{\mathtt{A}_t})$`)`) pour une offre toujours valide, il demande au gestionnaire de retirer la tâches cédée (`manager?Remove(`τ`)`) et confirme la délégation auprès du receveur (ν_j`!Confirm(`$\delta(\tau, \nu_i, \nu_j, \overrightarrow{\mathtt{A}_t})$`)`). S'il reçoit une acceptation ou une contre-proposition pour une offre qui n'est plus valide (*i.e.* la tâche a été confiée à l'exécutant pour consommation), il répond par une annulation (ν_j`!Withdraw(`$\delta(\tau, \nu_i, \nu_j, \overrightarrow{\mathtt{A}_t})$`)`). Dans ces deux cas, ainsi que lors de la réception d'un refus d'une offre (ν_j`:Reject(`O_i`)`), le négociateur retourne ensuite dans l'état `Répondant`. 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 `Troqueur`.

Dans l'état `Troqueur`, le négociateur attend de recevoir la confirmation ou l'annulation d'un échange en cours. S'il reçoit une confirmation (ν_j`:ConfirmSwap(`$\sigma(\tau, \tau_{\text{swap}}, \nu_i, \nu_j, \overrightarrow{\mathtt{A}_t})$`)`), il demande au gestionnaire d'ajouter dans le lot la tâche reçue (`manager ? Add(`τ`)`). S'il reçoit une annulation (ν_j`:WithdrawSwap(`$\sigma(\tau, \tau_{\text{swap}}, \nu_i, \nu_j, \overrightarrow{\mathtt{A}_t})$`)`), le négociateur demande au gestionnaire de remettre dans le lot la tâche précédemment retirée. Dans les deux cas, le gestionnaire retourne dans l'état `Répondant`.

8. Expérimentations

Nos expériences visent à valider que, lorsqu'elle est exécutée de manière concurrente au processus de consommation, la stratégie de réallocation : (1) réduit significativement le temps de réordonnancement ; (2) améliore le *flowtime* ; (3) ne pénalise pas la consommation ; (4) est robuste aux aléas d'exécution (*i.e.* le ralentissement de nœuds) ; et (5) s'adapte à la libération de jobs. Nous présentons ici nos méthodes de référence, nos métriques, notre protocole expérimental et nos résultats[(2)].

Nous considérons ici le processus de réalisation des tâches où notre stratégie de consommation spécifie leurs ordonnancements au sein du lot de chaque nœud (cf. section 5). Afin de comparer notre stratégie de réallocation exécutée manière concurrente à ce processus de consommation, nous considérons :

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

- 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-nœuds, 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.

Nos expérimentations sont réalisées en simulant la consommation des tâches. C'est pourquoi, pour définir nos métriques, plutôt que le temps estimé d'exécution des tâches par les nœuds (cf. définition 3.3), nous considérons le **coût simulé** $c^S(\tau, \nu)$ comme le temps que l'exécutant du nœud ν_i consacre à la tâche τ :

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

$$c^{S_E}(\tau, \nu_i) = c(\tau, \nu_i) \tag{8.1}$$

- avec le ralentissement de la moitié des nœuds,

$$c^{S_H}(\tau, \nu_i) = \begin{cases} 2 \times c(\tau, \nu_i) & \text{si } i \bmod 2 = 1 \\ c(\tau, \nu_i) & \text{sinon.} \end{cases} \tag{8.2}$$

C'est pourquoi nous distinguons :

- le ***flowtime* simulé** $C^S_{\text{mean}}(\overrightarrow{A_t})$ calculé à partir d'une allocation $\overrightarrow{A_t}$ selon les coûts simulés ;
- le ***flowtime* réalisé** $C^R_{\text{mean}}(\overrightarrow{A_t})$ calculé à partir des temps de réalisation des tâches effectivement mesurés.

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

$$\Gamma = \frac{C^R_{\text{mean}}(\overrightarrow{A_0}) - C^R_{\text{mean}}(\overrightarrow{A_e})}{C^R_{\text{mean}}(\overrightarrow{A_0})} \tag{8.3}$$

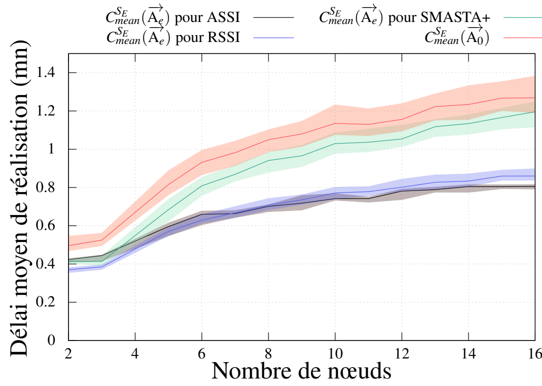
où $\overrightarrow{A_e}$ est l'allocation des tâches au moment de leur exécution et $\overrightarrow{A_0}$ est l'allocation initiale. Il est à noter que si aucune réallocation n'a lieu pendant le processus, $\overrightarrow{A_e} = \overrightarrow{A_0}$. Le taux d'amélioration est positif si le *flowtime* 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 temps de réordonnancement, le taux de disponibilité locale qui mesure la proportion des ressources traitées localement, 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 RSSI ou notre propre stratégie.

Notre prototype SMASTA+ [6] est implémenté avec le langage de programmation Scala et la bibliothèque Akka [15] 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. 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. Notre protocole expérimental consiste, pour les différentes expériences, à générer aléatoirement 25 allocations initiales pour des problèmes d'allocation distincts.

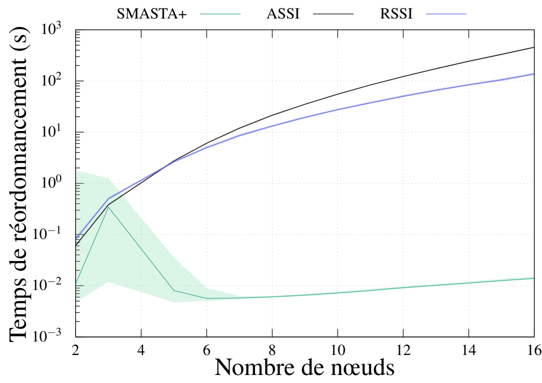
Afin d'évaluer notre processus de réallocation indépendamment du processus de consommation des taches, nous considérons $m \in [2; 16]$ nœuds, $\ell_0 = 4$ jobs, $n = 12 \times m$ tâches avec 10 ressources par tâche. Chaque ressource ρ_i est répliquée 3 fois et $|\rho_i| \in [0; 100]$.

Hypothèse 1 : la stratégie de réallocation réduit significativement le temps de réordonnancement. — Nous considérons ici que les allocations initiales sont aléatoires.

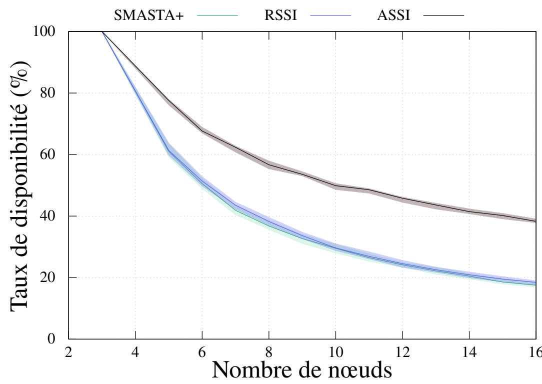
La figure 8.1 montre les médianes et les écarts types de nos métriques, le *flowtime* simulé, le temps d'ordonnancement et le taux de disponibilité, en fonction du nombre de nœuds. Nous observons que notre stratégie de réallocation améliore le *flowtime* de l'allocation initiale mais qu'il est supérieur à celui atteint par les méthodes fondées sur des enchères séquentielles. La méthode d'allocation, qui considère le meilleur nœud pour chaque tâche mise aux enchères, permet d'obtenir un *flowtime* simulé 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*, ne sont pas prises en compte par le processus de négociation, notre stratégie multi-agents s'avère efficace : le *flowtime* simulé atteint par notre stratégie est meilleur que celui de l'allocation initiale, et le surcoût par rapport à une méthode d'allocation basée sur des enchères séquentielles est en moyenne de 40 %. En effet, notre 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.
Comparativement aux méthodes basées sur des enchères qui évaluent toutes les délégations possibles, notre stratégie multi-agents présente un temps d'ordonnancement nettement inférieur. 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 notre stratégie de négociation croît exponentiellement avec le nombre de nœuds, tandis que l'écart pour le *flowtime* reste globalement constant. Ainsi, même lorsque le nombre de nœuds est limité, le bénéfice obtenu sur le *flowtime* grâce aux enchères séquentielles sera contrebalancé et annulé par le coût supplémentaire lié au temps de réordonnancement. Comme nous l'explorerons ultérieurement, ce surcoût a un impact négatif sur l'équilibrage continu des charges pendant l'exécution des tâches.



(a) Délai moyen de réalisation



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



(c) Taux de disponibilitié

FIGURE 8.1 – Réallocation sans consommation

Afin d'évaluer le processus de réallocation exécuté de manière concurrente à la consommation des taches, nous considérons $m = 8$ 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, nous considérons dans nos expérimentations qu'une réallocation bilatérale est socialement rationnelle si elle fait décroître d'au moins une seconde le *flowtime*. Le fait que, dans nos expériences, la différence entre le *flowtime* réalisé et le *flowtime* simulé de l'allocation initiale ($C^R_{\text{mean}}(\overrightarrow{A_0}) - C^S_{\text{mean}}(\overrightarrow{A_0})$), qui mesure le coût de l'infrastructure, est négligeable, conforte ce choix technologique.

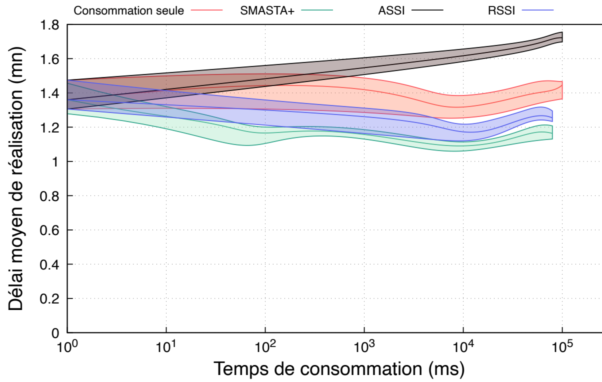
Hypothèse 2 : la stratégie de réallocation améliore le flowtime. — Nous considérons ici que les allocations initiales sont aléatoires et que les agents ont une connaissance parfaite de l'environnement d'exécution (cf. équation 8.1).

La figure 8.3a montre les médianes et les écarts types des différents *flowtime* en fonction du nombre de tâches. Nous observons que le *flowtime* réalisé atteint par la méthode d'allocation est pire que le *flowtime* réalisé de l'allocation aléatoire initiale. Même si le *flowtime* simulé est très bon, la méthode d'allocation 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, 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). Ces 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 notre stratégie est supérieur au nombre de délégations de la méthode de réallocation fondée sur des enchères, le taux d'amélioration de performance (Γ) de notre stratégie est meilleur (entre 13 % et 23 %) que le taux d'amélioration de performance de la méthode de référence (entre 0 % et 15 %).

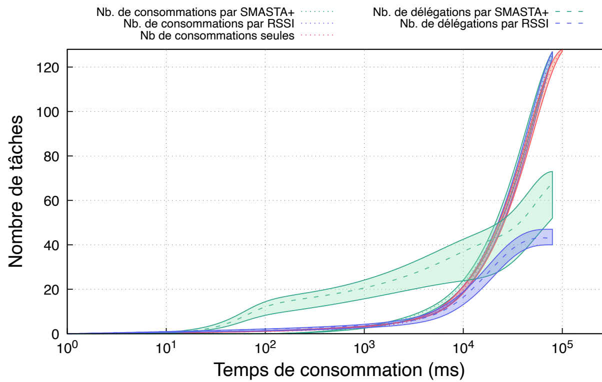
La figure 8.2 montre l'évolution des médianes et écarts-types de nos métriques au cours de la consommation de 128 tâches selon une échelle de temps logarithmique. Elle confirme que le meilleur *flowtime* réalisé est celui obtenu par notre stratégie lorsqu'elle est exécutée de manière concurrente au processus de consommation. La réactivité (*responsiveness*) de notre stratégie

Hypothèse 3 : la stratégie de réallocation ne pénalise pas la consommation. — Nous considérons ici uniquement des allocations initiales stables, c'est-à-dire pour lesquelles il n'existe aucune réallocation bilatérale socialement rationnelle.

Dans la figure 8.3b le *flowtime* réalisé des méthodes de réallocation est similaire au *flowtime* réalisé de l'allocation initiale. 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.



(a) Délai moyen de réalisation

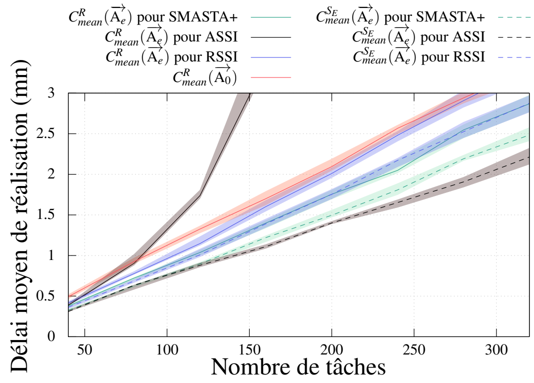


(b) Nombre de tâches consommées/déléguées

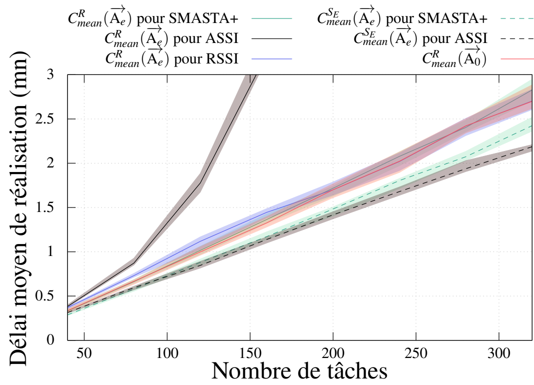
FIGURE 8.2 – Au cours de la consommation selon une échelle de temps logarithmique

Hypothèse 4 : la stratégie de réallocation s'adapte aux aléas d'exécution. — Nous considérons ici le coût qui simule le ralentissement de la moitié des nœuds (cf. équation 8.2).

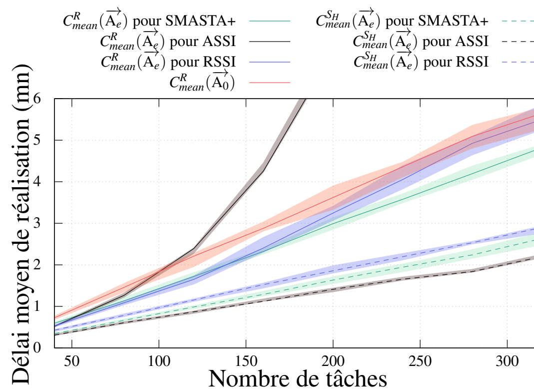
Nous observons dans la figure 8.3c que les délais moyens de réalisation ont doublé à cause des aléas d'exécution. Le *flowtime* réalisé et le *flowtime* simulé de la méthode d'allocation fondée sur des enchères, qui ne prend pas en compte le ralentissement de la moitié des nœuds, sont supérieurs aux mêmes configurations sans ralentissement (cf. figure 8.3a). Contrairement à la méthode de réallocation fondée sur des enchères, le *flowtime* réalisé de notre 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. 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 %.



(a) Depuis une allocation aléatoire



(b) Depuis une allocation stable

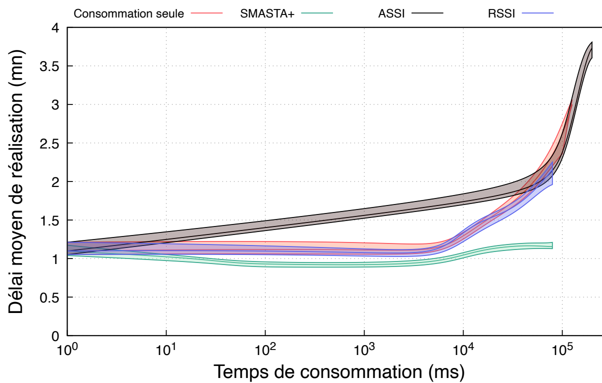


(c) Avec aléas d'exécution

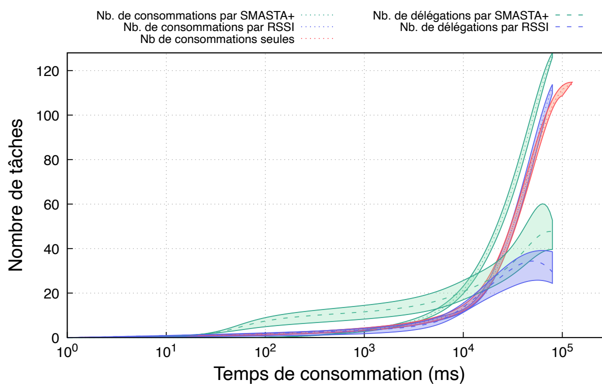
FIGURE 8.3 – Hypothèses 2, 3 et 4 : la stratégie de réallocation améliore le *flowtime*, ne pénalise pas la consommation et s'adapte aux aléas d'exécution

Hypothèse 5 : la stratégie de réallocation s'adapte à la libération de jobs. — Nous considérons ici à nouveau que les allocations initiales sont aléatoires et que les agents ont une connaissance parfaite de l'environnement d'exécution (cf. équation 8.1). Nous examinons 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.

La figure 8.4 montre l'évolution des médianes et écarts-types de nos métriques au cours de la consommation selon une échelle de temps logarithmique. La figure 8.4a révèle une nette amélioration du *flowtime* moyen avec notre stratégie. La figure 8.4b met en évidence le nombre plus important de délégations réalisées au cours du processus par notre stratégie. Il est intéressant de noter que, non seulement notre 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



(a) Délai moyen de réalisation



(b) Nombre de tâches consommées/déléguées

FIGURE 8.4 – Lors de la libération selon une échelle de temps logarithmique

réallocation ou avec une méthode d'allocation, et du même ordre que le temps d'exécution avec une méthode de réallocation fondée sur des enchères. Contrairement aux méthodes de référence, notre 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.

9. 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 : l'exécutant qui exécute les tâches ; le négociateur qui marchande des réallocations avec ses pairs ; et le gestionnaire 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 d'un modèle de leurs homologues qui guident leur comportement dans les interactions.

L'architecture modulaire que nous proposons, en se basant sur la séparation des rôles, permet de réduire la complexité de conception des comportements des agents, comparativement au cas où un seul agent aurait à gérer simultanément l'exécution des tâches et la négociation des réallocations. Cette complexité est mesurée dans la table 9.1 en nombre d'états, de transitions et de lignes de code.

Agent	Nb. d'états	Nb. de transitions	Nb. de lignes
Superviseur	9	69	626
Exécutant	2	7	173
Gestionnaire	5	23	465
Négociateur	9	74	1306
Total agents composants	16	104	1944

Table 9.1 – Mesure de complexité des comportements

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 29 %. En sélectionnant efficacement les délégations et les échanges qui atténuent le coût d'exécution des tâches, le marchandage réduit significativement le temps de réordonnancement. De plus, le surcoût lié aux négociations est négligeable car, lorsque l'allocation est stable, elles sont suspendues. 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. Nous avons également évalué notre stratégie face à la libération de jobs au fil de l'eau. La réactivité (*responsiveness*) de notre stratégie, qui se démarque des méthodes d'allocation et de réallocation de référence fondées

sur des enchères, s'explique par la réallocation de lots de tâches grâce à de multiples négociations bilatérales concurrentes entre-elles et avec le processus de consommation.

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.

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 qui nous ont permis d'améliorer cet article. Nous tenons également à remercier le Mésocentre de Calcul Scientifique Intensif de l'Université de Lille pour la mise à disposition des ressources informatiques.

Bibliographie

[1] Q. Baert, « Négociation multi-agents pour la réallocation dynamique de tâches », Thèse de doctorat, Université de Lille, Septembre 2019.

[2] E. Beauprez, L. Bigand, A.-C. Caron, M. Morge & J.-C. Routier, « Réaffectation de tâches de la théorie à la pratique : état de l'art et retour d'expérience », in *Actes des Journées Francophones sur les Systèmes Multi-Agents (JFSMA)*, Cépaduès, 2021, p. 51-60.

[3] E. Beauprez, A.-C. Caron, M. Morge & J.-C. Routier, « Échange de tâches pour la réduction de la durée moyenne de réalisation », in *Actes des Journées Francophones sur les Systèmes Multi-Agents (JFSMA)*, Cépaduès, 2022, p. 19-28.

[4] ———, « Consommation adaptative par négociation continue », in *Actes des Journées Francophones sur les Systèmes Multi-Agents (JFSMA)*, Cépaduès, 2023, p. 21-30.

[5] ———, « Délégation de lots de tâches pour la réduction de la durée moyenne de réalisation », *ROIA* **4** (2023), nº 2, p. 193-221.

[6] E. Beauprez, M. Morge & A.-C. Caron, « Scala implementation of the Extended Multi-agents Situated Task Allocation », `https://gitlab.univ-lille.fr/maxime.morge/smastaplus`, 2020.

[7] Y. Chen, X. Mao, F. Hou, Q. Wang & S. Yang, « Combining re-allocating and re-scheduling for dynamic multi-robot task allocation », in *Proc. of IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016, p. 395-400.

[8] H.-L. Choi, L. Brunet & J. P. How, « Consensus-based decentralized auctions for robust task allocation », *IEEE Transactions on Robotics* **25** (2009), nº 4, p. 912-926.

[9] N. Creech, N. Criado Pacheco & S. Miles, « Resource allocation in dynamic multiagent systems », `https://arxiv.org/abs/2102.08317`, 2021.

[10] A. Damamme, A. Beynier, Y. Chevaleyre & N. Maudet, « The Power of Swap Deals in Distributed Resource Allocation », in *Proc. of International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2015, p. 625-633.

[11] J. Dean & S. Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », in *Proc. of The Sixth Symposium on Operating System Design and Implementation*, 2004, p. 137-150.

[12] U. Endriss, N. Maudet, F. Sadri & F. Toni, « Negotiating Socially Optimal Allocations of Resources », *Journal of Artificial Intelligence Research* **25** (2006), p. 315-348.

[13] S. Koenig, C. Tovey, M. Lagoudakis, V. Markakis, D. Kempe, P. Keskinocak, A. Kleywegt, A. Meyerson & S. Jain, « The power of sequential single-item auctions for agent coordination », in *Proc. of The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI)*, vol. 2006, 2006, p. 1625-1629.

[14] K. Lerman, C. Jones, A. Galstyan & M. J. Matarić, « Analysis of dynamic task allocation in multi-robot systems », *The International Journal of Robotics Research* **25** (2006), nº 3, p. 225-241.

[15] Lightbend, « Akka is the implementation of the Actor Model on the JVM », `http://akka.io`, 2020.

[16] S. Mayya, D. S. D'antonio, D. Saldaña & V. Kumar, « Resilient task allocation in heterogeneous multi-robot systems », *IEEE Robotics and Automation Letters* **6** (2021), nº 2, p. 1327-1334.

[17] G. A. Mills-Tettey, A. Stentz & M. B. Dias, « The dynamic hungarian algorithm for the assignment problem with changing costs », *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27* (2007), p. 19.

[18] C. Nam & D. A. Shell, « When to do your own thing: Analysis of cost uncertainties in multi-robot task allocation at run-time », in *Proc. of IEEE International Conference on Robotics and Automation*, 2015, p. 1249-1254.

Abstract. — In this paper, we study the problem of continous allocation of 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 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 Systems, Distributed Problem Solving, Agent-based Negotiation, Agent Architecture.