Rapport de stage de troisième année

Encadrant industriel : Martin QUINSON

Encadrant académique : Malika SMAIL-TABBONE

Parallel and Distributed Simulation of Large-Scale Distributed Applications

Guillaume SERRIERE

8 avril 2013 - 27 septembre 2013

Laboratoire lorrain de recherche en informatique et ses applications

Villers-lès-Nancy

Résumé

SimGrid est un framework de simulation développé par plusieurs équipes de recherche qui a pour but de fournir un ensemble d'outils pour la simulation d'applications distribuées. Très récemment, une parallélisation du noyau de simulation a été efféctuée, mais elle n'a pas apporté les gains de temps escomptés. Mon travail sera donc de trouver pourquoi cette mise en parallèle n'a pas été aussi efficace qu'attendue et de trouver également de nouvelles pistes pour permettre au framework de se rapprocher de sa performance optimale.

Dans une première partie, je ferai une analyse complète de la parallèlisation actuelle à l'aide de divers outils, puis je proposerai plusieurs pistes pour améliorer les performances. Enfin, je testerai la validité de l'approche de l'exécution conditionnelle.

Remerciements

Je tiens tout d'abord à remercier mon encadrant de stage Martin Quinson pour m'avoir accueilli de nouveau pour mon stage de troisième année ainsi que pour m'avoir permis de découvrir une seconde facette du métier de la recherche.

Je remercie ensuite l'équipe AlGorille dans son ensemble pour son accueil, ainsi que pour avoir patiemment pris le temps de répondre à mes questions même les plus tordues. De même, je les remercie de m'avoir intégré à la session plénière du projet, et de m'avoir permis à la fois de rencontrer des membres d'autres centres Inria, mais aussi d'avoir rencontré de nombreux utilisateurs du projet SimGrid originaires de plusieurs endroits dans le monde.

Je remercie également l'équipe Mescal pour m'avoir accueilli à Grenoble durant une semaine. Je remercie tout particulièrement Augustin Degomme pour m'avoir permis de prendre en main le logiciel de profilage VTune ainsi que de m'avoir fourni certaines des pistes explorées pour la réalisation de mon objectif de stage. Je remercie également Arnaud Legrand pour m'avoir fait découvrir les outils Org mode, PlantUML et Sweave qui m'ont permis de structurer de manière rapide et efficace mon travail.

Je remercie pour finir le Loria pour m'avoir accueilli en son sein, et notamment Delphine Hubert , l'assistante universitaire de l'équipe, pour sa patience ainsi que son travail dans la préparation de ma mission à Grenoble.

Table des matières

Table des figures

Liste des tableaux

1 Présentation du stage

Dans le cadre de ma troisième année à Télécom Nancy, j'ai dû effectuer un stage en qualité d'ingénieur. J'ai décidé de faire celui-ci dans l'équipe de recherche AlGorille[1] sous la direction de Martin Quinson, qui avait accepté de m'encadrer pendant le stage de niveau technicien, en seconde année, effectué pendant l'été 2012. Le stage se déroule à Nancy au Loria[2] du 8 avril au 27 septembre 2013.

1.1 Présentation de l'institution d'accueil

Le Loria, laboratoire LOrrain de Recherche en Informatiques et ses Applications est une unité mixte de recherche (UMR) commune à 3 entités différentes :

- le CNRS, Centre Nationale de Recherche Scientifique,
- l'INRIA, Institut National de Recherche en Informatique et Automatique,
- l'Université de Lorraine.

Créé en 1997, le Loria a pour mission la recherche fondamentale et appliquée dans le domaine des sciences informatiques. Il est membre de la Fédération Charles Hermite qui regroupe trois laboratoires de recherche en mathématiques et en STIC (le CRAN, l'IECL et le Loria)

Le Loria emploie plus de 500 personnes réparties ainsi :

- 105 enseignants-chercheurs,
- 63 chercheurs,
- 14 personnels administratifs,
- 130 doctorants,
- 105 post-doctorants,
- 50 ingénieurs sur contrat,
- 50 stagiaires.

Il est séparé en cinq grands départements depuis 2010 :

- algorithmique, calcul, image et géométrie,
- méthodes formelles,

– réseaux, systèmes et services,
– traitement automatique des langues et des connaissances,
– systèmes complexes et intelligence artificielle.

Au niveau des instances dirigeantes, le Loria possède deux conseils. Le premier est un conseil scientifique. Il est composé du directeur du laboratoire, des deux directeurs adjoints et des scientifiques responsables des cinq départements du laboratoire. Il a pour rôle d'assister le directeur dans ses décisions et la mise en œuvre de celles-ci.

Le second est le conseil de laboratoire. Celui-ci est composé de membres élus et de membres nommés. Il a une vocation consultative sur les questions concernant l'UMR : politique scientifique, organisation du laboratoire, budget, recrutement de personnels, formation, règlement intérieur, aménagement du temps de travail, conditions de travail, consultation des personnels.

1.2 Présentation de l'équipe

J'ai effectué mon stage dans l'équipe AlGorille crée en 2007 qui fait partie du département "Réseaux, systèmes et services" . Elle est composée de 4 chercheurs dont 3 enseignants chercheurs, de 5 ingénieurs, et de deux doctorants. Elle est dirigée depuis sa création par Jens GUSTEDT. L'équipe poursuit des recherches dans le cadre des systèmes distribuées. Ils étendent leur recherche sur trois axes thématiques majeurs :

– structuration des applications pour l'évolutivité,
– gestion de ressources transparentes,
– validation expérimentale.

La principale contribution de l'équipe est le framework [1] de simulation SimGrid. Elle a également développé de nombreux outils tels que Kadeploy3[3] ou Distem[4]. L'équipe a aussi à sa charge le cluster Nancéien de la plate-forme de calcul scientifique Grid'5000[5].

1. Un framework est un ensemble cohérent de composants logiciels qui sert à créer un autre logiciel.

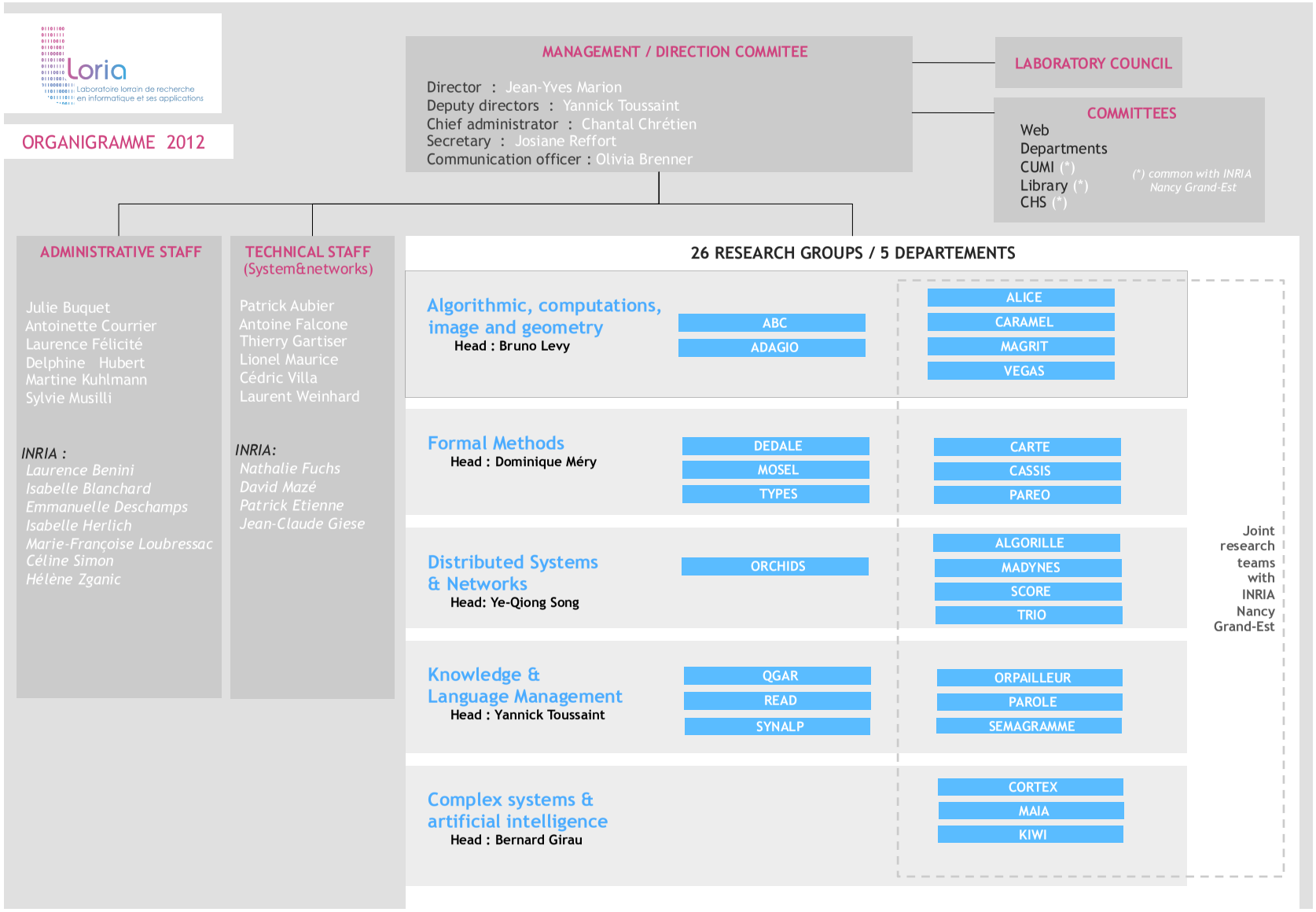


FIGURE 1.1 – Organigramme du Loria.

1.3 SimGrid

SimGrid est un framework multi plate-forme destiné à la simulation d'algorithmes distribués. Il est développé dès le début des années 2000 par Henri Casanova[6]. Initialement, le simulateur avait été développé pour ses recherches, mais faisant le constat que d'autres personnes devraient au final réaliser le même travail, il mit en place une API [2] pour son simulateur et en commença la diffusion. Le projet est aujourd'hui toujours très actif, soutenu par plusieurs équipes, réparties dans la France, mais aussi à Hawaï où se trouve actuellement Henri Casanova. Il possède un financement ANR via le projet SONGS et il est sous la coordination de Martin Quinson, Arnaud Legrand et Frédéric Suter.

1.3.1 Présentation du projet

Un système distribué est un système dont les ressources ne sont pas concentrées en un seul point, mais "distribuées" entre plusieurs entités. En informatique, la distribution concerne principalement deux ressources, la puissance de calcul et les données. Aujourd'hui, ces systèmes sont de plus en plus répandus et connaissent des formats allant du système très simple (deux machines en réseau local) à des formats plus complexes impliquant plusieurs milliers de machines hétérogènes reliées par un réseau lui aussi hétérogène. Les applications qui utilisent ces systèmes sont également très variées. On aura par exemple :

- des applications pair-à-pair de partage de données comme eMule[7] ou encore les applications utilisant le protocole BitTorrent[8],
- des applications de calcul distribué comme le projet coopératif BOINC[9] incluant des projets comme LHC@home[10] du CERN ou encore le projet SETI@home[11],
- des systèmes plus classiques d'un couple serveur web / serveur de base de données réparti sur deux machines.

Bien que très répandus, ces systèmes posent des difficultés à ceux qui développent les applications destinées à les utiliser. Il est en effet parfois difficile voire impossible de faire des tests sur celles-ci. Il existe trois différentes façons d'effectuer des tests sur ce type d'application : le test réel, la simulation et l'émulation.

Test réel Le test réel est la solution qui apparaît comme la plus "naturelle". Elle consiste à déployer [3] l'application sur le système pour laquelle elle a été développée puis de faire les tests directement dessus. Cette solution a un avantage évident, elle permet de valider l'application sur la véritable plate-forme. Toutefois, elle possède plusieurs inconvénients :

2. Une API est une interface de programmation d'un composant logiciel qui les autres vont devoir interagir avec lui.
3. Lancer.

– Il faut posséder l'architecture pour faire le test, et si on est dans le cas du LHC et que la plate-forme utilisée fait plusieurs dizaines de milliers de machines, il est impossible d'avoir à disposition un tel système juste pour effectuer des tests. Une solution peut être alors d'utiliser la plate-forme de destination si elle existe déjà pour faire des tests mais cela peut entraîner l'interruption du service ce qui n'est pas toujours possible.
– Le contrôle environnemental est très complexe. En effet, si jamais vous souhaitez reproduire des comportements sur certaines machines ou sur le réseau, vous devez intervenir dessus parfois physiquement ce qui n'est pas toujours possible.

Simulation La simulation consiste à modéliser votre application sous forme d'algorithmes puis de faire évoluer ceux-ci dans un environnement simulé. Cette solution a l'avantage de permettre le test des modèles sur une grande variété de plates-formes puisqu'elle n'est pas physique. Elle permet aussi d'avoir un contrôle total de l'environnement et autorise ainsi des scénarios assez poussés sur les algorithmes. Enfin, on peut aussi assurer la reproductibilité d'une condition expérimentale du fait du parfait contrôle du déroulement des tests.

Toutefois, cette solution possède un inconvénient majeur, on ne teste que les modèles. On ne peut donc que valider le bon fonctionnement des algorithmes de l'application, mais elle ne permet pas de détecter les erreurs dans l'implémentation comme le permet un test réel.

Émulation Elle est à la fois la solution la plus intéressante, mais aussi la plus complexe à mettre en place. L'émulation consiste à utiliser l'application réelle, le fichier binaire par exemple, pour la faire interagir avec un monde simulé. L'avantage de cette méthode est que l'on peut utiliser l'application définitive et donc valider l'implémentation finale, et comme on utilise une plate-forme simulée, on peut tester des architectures très vastes et très diverses sans devoir posséder celles-ci.

Le principal inconvénient de cette technique réside dans la complexité de mise en œuvre. Il faut en effet être capable d'intercepter le comportement de l'application ainsi que de lui communiquer les réactions de son environnement. L'autre problème majeur réside dans le fait qu'il faut effectuer les calculs que devra réaliser l'application. Dans le cas d'application de type BitTorrent, cela ne pose pas de problème majeur, cependant dans le cas d'application de calcul distribué cela peut devenir rapidement handicapant en allongeant de manière démesurée la durée des tests.

SimGrid permet pour sa part d'effectuer principalement des simulations.

On lui communique la plate-forme distribuée via un fichier au format XML qui contient la description de chaque hôte mais aussi les différents liens qui forment le réseau du système.

Le déploiement de l'application est fourni dans un second fichier, lui aussi au format XML qui indique quelles parties de l'application doivent être lancées sur les différents hôtes de la plate-forme. On peut indiquer le délai entre le début de la simulation et le lancement du processus, mais aussi communiquer des arguments. En réalité, ce que l'on fournit au fichier de déploiement sont les noms des routines qui seront exécutées, et qui seront donc l'équivalent de la fonction main de chaque processus. Ces routines doivent être ensuite reliées à des fonctions lors de l'initialisation du simulateur.

Le framework comporte également une variété de modèles que ce soit pour décrire le comportement du ou des processeurs des hôtes, mais aussi des modèles pour gérer les communications par le réseau. De part son côté libre, il est aussi possible pour un utilisateur de rajouter ses propres modèles. Toutefois cette dernière opération est relativement complexe d'où la volonté actuelle de l'équipe de simplifier cette création afin de permettre un développement plus rapide.

Développé en C, il est possible d'utiliser SimGrid avec les langages Java et Ruby via des bindings [4] réalisés par l'équipe.

1.3.2 Architecture du projet

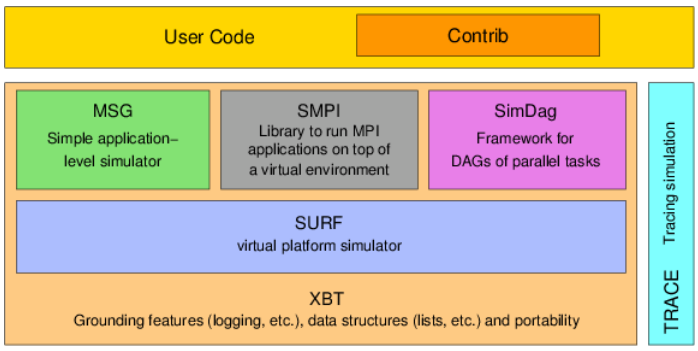


FIGURE 1.2 – Schéma de l'architecture des différentes couche de SimGrid.

La figure 1.2 montre les différentes API regroupées dans les différentes couches qui composent SimGrid. Voici une brève description de celles-ci.

XBT XBT fournit toutes les structures de données basiques utilisées dans SimGrid, mais fournit aussi un gestionnaire de log très puissant, ainsi qu'un gestionnaire d'erreur capable

4. Un binding est le fait de fournir une interface à une bibliothèque dans un langage différent.

notamment d'afficher une pile d'appel lors d'un crash du simulateur par exemple. La plupart des entités de cette API sont accessibles de l'extérieur ce qui permet à l'utilisateur de pouvoir utiliser des tableaux dynamiques, des files mais aussi des dictionnaires ainsi que plusieurs autres fonctionnalités dont le gestionnaire de log.

C'est aussi cette couche qui fournit l'ensemble des interfaces permettant de garantir la portabilité du framework.

SURF SURF est l'API qui contient le noyau de simulation. C'est ici que se trouve les différents modèles (CPU, réseau, ...) qui seront utilisés pour la résolution des actions. Elle n'est accessible que via deux interfaces minimalistes, une permettant d'injecter des interactions dans le monde simulé, et une seconde pour lancer la simulation. En fonctionnement normal, l'utilisateur n'est pas censé utiliser directement cette couche.

SIMIX Cette API n'est pas présente sur le schéma précédent. Celle-ci est une surcouche de SURF et n'est en pratique pas utilisée directement par l'utilisateur. C'est dans cette API que sont stockées toutes les informations des processus. C'est elle qui est le point d'entrée du noyau de simulation via une interface semblable à celle des appels systèmes sous GNU/Linux, l'interface des simcalls. C'est également dans cette couche que se trouve l'ordonnanceur du framework ainsi que le mécanisme d'exécution des processus simulés.

MSG C'est cette API qui est utilisée majoritairement par les utilisateurs pour interagir avec le monde simulé. L'interface possède un grand nombre de fonctions interfacées avec SIMIX permettant d'effectuer des actions à simuler mais aussi d'obtenir des informations sur l'environnement comme le temps du monde simulé, ou des informations sur l'hôte ou encore d'autres processus.

SMPI Comme son nom l'indique, cette API sert à la simulation d'applications développées en utilisant MPI[12]. Toute l'interface n'est pas implémentée, toutefois, cela n'est pas pénalisant parce que les fonctions les plus souvent utilisées sont présentes dans l'API.

SimDag Cette API est un peu particulière. Celle-ci ne passe pas par SIMIX mais est directement reliée au noyau SURF et permet d'injecter des actions dans l'environnement ainsi que de lancer la résolution de celle-ci. Cette interface permet notamment de coupler le simulateur à une autre application lui permettant ainsi d'utiliser le noyau de simulation. Cela peut être utile notamment pour le développement d'émulateurs.

TRACE Cette API est très utile car elle permet de rejouer des simulations déjà effectuées. En effet, lors de la simulation via MSG par exemple, on peut demander au simulateur de garder une trace des actions effectuées. Puis une fois terminée, il ne reste plus qu'à prendre les traces générées et à les réinjecter dans un deuxième simulateur qui dérivera du premier.

1.3.3 Fonctionnement général

Pour la suite de ce document, nous nous concentrerons sur le fonctionnement de SimGrid lorsqu'on utilise l'API MSG et donc le mode avec contexte. La boucle principale du programme est relativement simple.

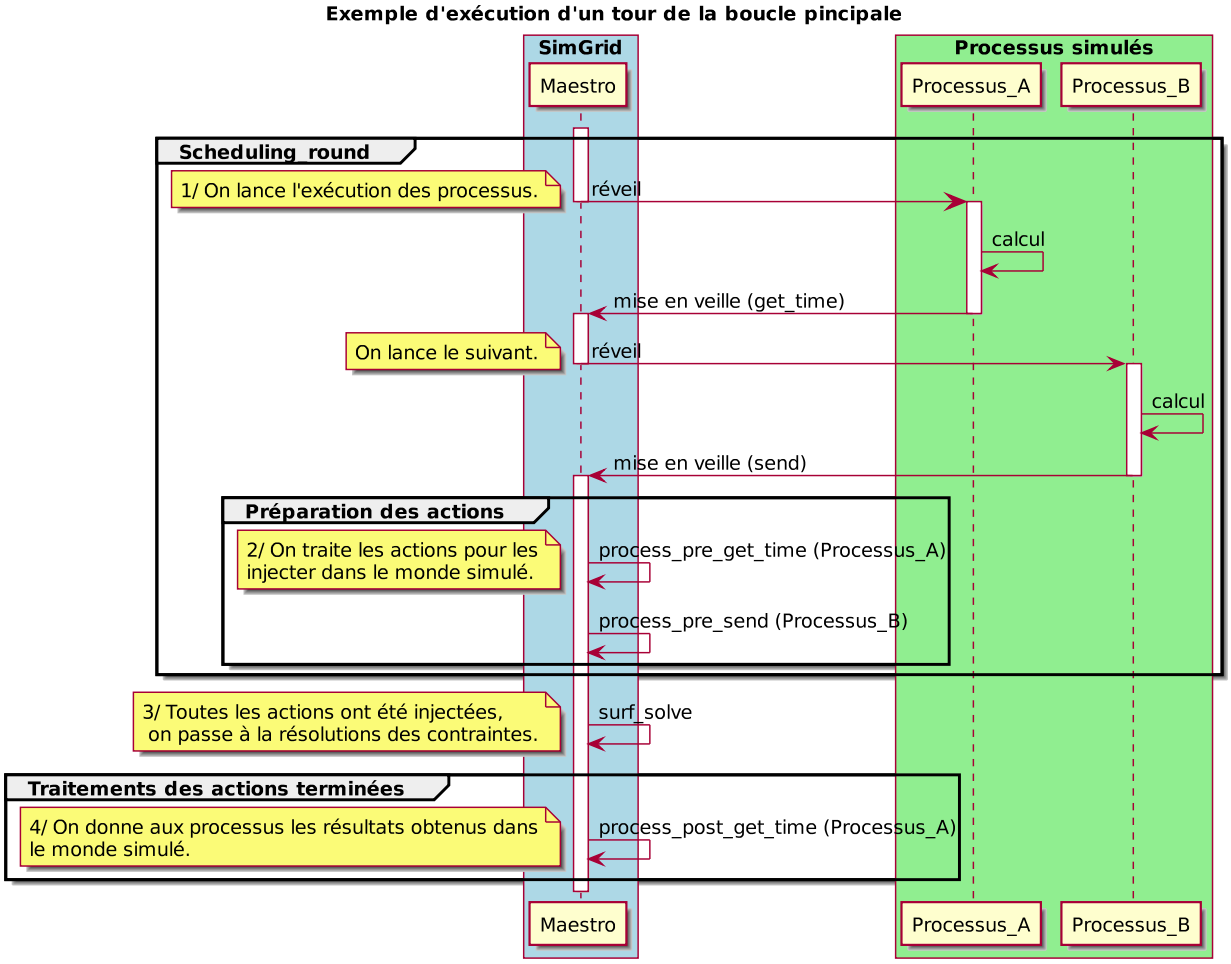


FIGURE 1.3 – Diagramme de séquence de la boucle principale.

1. L'ordonnanceur de SIMIX commence l'exécution des processus qui doivent être réveillés. Une fois qu'un processus effectue une action qui nécessite de passer par SURF ou SIMIX, celui-ci est mis en veille.

2. Les actions générées lors des exécutions sont injectées dans le noyau de simulation après une étape de formatage.
3. Le noyau de simulation est appelé pour résoudre les actions en cours dans le monde simulé. Il résout donc les différentes contentions pour trouver le ou les processus qui ont terminés leur précédente action et qui doivent donc être redémarrés. Les contentions correspondent aux différentes contraintes qui sont imposés à chaque élément du système distribué (processeur, réseau, ...). La résolution de ses contentions permet de déterminer qu'elles sont les actions qui termineront le plus vite (en fonction des différentes charges).
4. On donne aux processus les résultats des actions dans le monde simulé.

Par convention, la phase située entre deux appels au noyau de simulation pour la résolution des contentions sera nommée "scheduling round" pour la suite du document.

Par souci d'optimisation, la partie où opère l'ordonnanceur est en réalité une boucle. En effet, il existe des actions auxquelles on peut apporter une réponse immédiatement sans devoir passer par une résolution des contentions du système et donc par le noyau de simulation. Ces actions sont, par exemple, une demande du temps ou de certaines informations sur la machine hôte. Ainsi au moment de formater les actions pour les injecter dans le noyau de simulation, on regarde si l'on ne peut pas directement donner une réponse et le cas échéant remettre le processus dans la liste de ceux qui doivent être exécutés. Et donc à la fin de la phase de formatage des actions, s'il y a des processus à qui on a répondu directement, on relance l'exécution de ceux-ci. Par convention, les phases internes à une *scheduling round* pendant lesquelles les processus sont exécutés seront nommées "sub-scheduling round" dans la suite du document.

On obtient alors le diagramme de séquence 1.4.

Chaque processus possède son propre contexte qui comprend à la fois les informations nécessaires pour SimGrid, mais aussi une zone mémoire servant de pile pour l'exécution de celui-ci. Lorsqu'un processus est exécuté, il est en réalité mis en route sur le processeur, soit par le biais de threads, soit par le biais d'un changement de contexte (voir annexe A). Ainsi, c'est le code de l'application elle-même qui est tourné, jusqu'à ce que celle-ci soit amenée à être mettre mise en veille ce qui entraînera un nouveau changement de contexte pour basculer sur un nouveau processus ou revenir au processus maître de SimGrid. Dans la suite de ce document, le *thread* qui à la charge de faire tourner les parties importantes du simulateur (noyau de simulation, traitement des actions, ...) sera dénommé "Maestro".

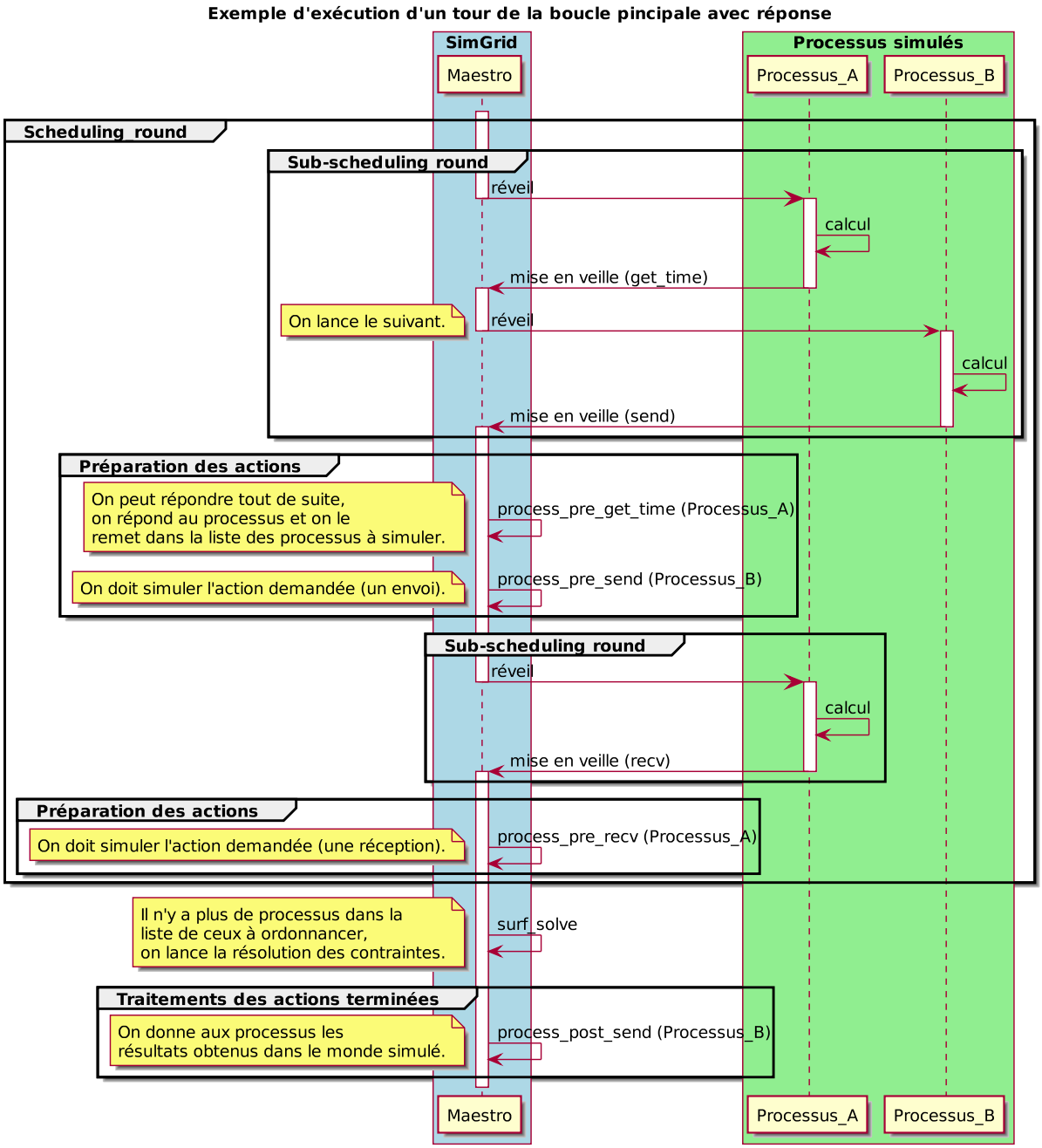


FIGURE 1.4 – Diagramme de séquence avec réponse immédiate.

Cette utilisation des contextes a l'avantage de permettre à chaque processus d'avoir sa propre pile et donc de s'exécuter comme en temps normal, avec ses propres données, tout en gardant la possibilité de partager des données globales avec les autres. Cette accès reste toutefois dangereux car un partage d'informations sans utiliser le monde simulé entre deux processus peut finir par créer des erreurs conséquentes dans la simulation (il y a des commu-

nications qui devraient exister mais qui ne sont pas simulées car elles passent par la mémoire partagée). Cependant, cet accès peut être un avantage car il permet notamment de factoriser le code et de limiter ainsi la place en mémoire.

1.3.4 Travaux de parallèlisation du code

L'équipe a eu l'idée de paralléliser les sub-scheduling round. Ainsi au lieu de n'avoir que Maestro pour faire tourner les processus à tour de rôle, on dispose d'un groupe de *threads* destinés à effectuer les exécutions en parallèle. En dehors de cette phase, tous les threads, à l'exception de Maestro, seront en pause. Par convention, on nommera ces *threads* des "workers".

On utilise deux barrières de synchronisation et une zone mémoire à accès restreint afin de faire la coordination entre tous les threads.

La première barrière se trouve au début des *sub-scheduling round* au moment où Maestro initialise la parmap avec les données nécessaires, et qu'il réveille les *workers* en sommeil. La parmap est la structure de données qui contient toutes les informations nécessaires à la synchronisation. La seconde se situe à la fin de la scheduling round. Cette barrière sert à attendre que tous les processus soient exécutés avant de quitter la parmap. Maestro ne peut franchir cette barrière qu'une fois tout achevé.

La zone mémoire à accès restreint est un entier qui correspond à l'indice du prochain processus à simuler dans le tableau des processus à ordonnancer. Il est incrémenté grâce à une opération atomique qui permet d'effectuer une post-incrémentation [5]. Cette opération garantit dans le même temps l'exclusion mutuelle des *threads* empêchant ainsi deux *threads* d'accéder à la zone mémoire au même moment.

Un exemple d'exécution est fourni à la figure 1.5

Le nombre de *worker* dans la parmap est fixé par un argument passé en ligne de commande. Un dispositif est intégré dans le framework pour détecter la quantité de cœurs disponible afin de faire correspondre automatiquement la quantité de *thread* et les possibilités offertes par la machine hôte. Au moment de la création de la parmap avec un mode parallèle demandé avec n threads, SimGrid créera n-1 *threads* qui seront considérés comme *workers* et Maestro, au moment de la sub-scheduling round, prendra le rôle du nième *worker* une fois qu'il aura franchi la première barrière de synchronisation.

5. La post-incrémentation est une opération qui consiste à incrémenter une variable et à renvoyer la valeur de celle-ci avant l'opération.

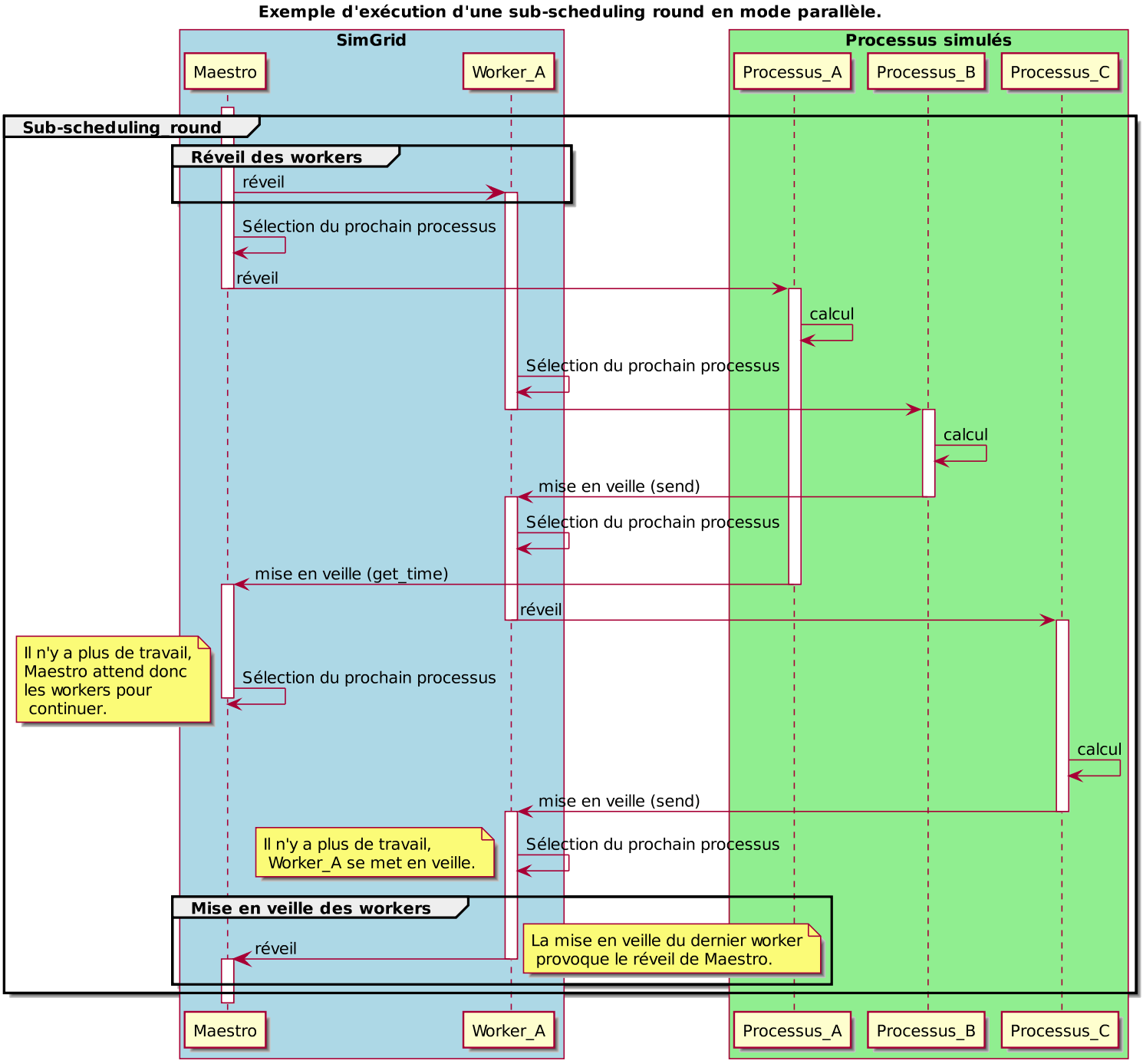


FIGURE 1.5 – Exemple d'exécution d'une *sub-scheduling round* parallélisée.

Un système de seuil statique a été mis en place afin de pouvoir combiner le mode séquentiel et le mode parallèle. On peut l'initialiser via la ligne de commande et si jamais le nombre de processus à lancer dans la *sub-scheduling round* est supérieur au seuil, on utilise le lancement parallèle, sinon on utilise le séquentiel.

Toutefois, cette parallélisation a une très grosse limite, elle permet de gagner du temps uniquement sur les très grosses expérimentations (plus de 300 000 nœuds). Le graphique 1.6

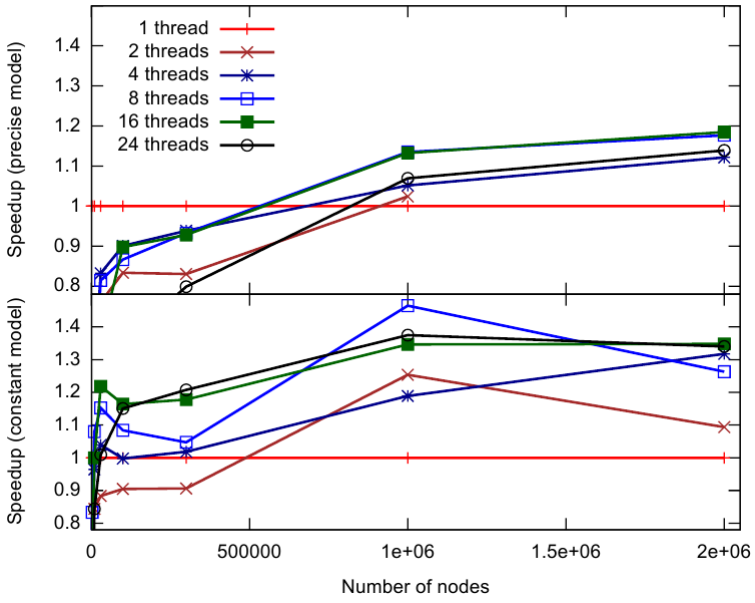


FIGURE 1.6 – Gain de temps sur les modes "précis" et "constant".

montre bien que les performances ne sont pas au rendez-vous, le gain de temps étant plus que décevant.

1.4 Sujet de Stage

1.4.1 Analyse de l'existant et amélioration de celui-ci

Comme vu précédemment au 1.3.4, une première version parallèle existe. Toutefois, l'article[13], montre que l'amélioration en terme de temps d'exécution est relativement éloignée de ce que l'on pourrait espérer. De plus, les mécanismes mis en œuvre dans le processus de parallélisation ne sont pas pleinement maîtrisés.

Dans le même temps, une erreur expérimentale a été commise dans cet article. En effet, lors du test comparatif des différents simulateurs, ce n'est pas le code du protocole Chord[14] qui a été exécuté mais le celui du protocole Pastry[15]. Mon travail dans cette partie se résumera en plusieurs points :

– faire une analyse complète de l'existant pour comprendre son fonctionnement mais aussi pour trouver ce qui cause la déperdition de temps,

– trouver des pistes pour optimiser le niveau de parallélisation,
– faire les travaux nécessaires à la rédaction de l'article de synthèse sur les travaux,
– rédiger l'article de conclusion assisté par Martin.

Cette partie sera la plus longue de mon stage.

1.4.2 Augmentation de la capacité de parallèlisation

Cette partie de mon stage, consiste à mettre en œuvre une nouvelle approche pour optimiser le parallélisme et en évaluer les performances. L'idée est de les *scheduling round* compresser en faisant redémarrer le plus tôt possible les processus qui peuvent l'être afin d'en réduire le nombre. Le mécanisme est expliqué de façon plus détaillée dans la partie 3.

Ce travail concernera la dernière partie de mon stage et consistera en la livraison de la validation technique ainsi que de la vérification de l'intérêt de la méthode.

1.4.3 Distribution de SimGrid

SimGrid a atteint un niveau de performance plus que satisfaisant en comparaison des concurrents. Aujourd'hui, la principale limitation du framework se situe au niveau de la mémoire physique des machines hôtes. En effet, pour chaque processus une pile est créée, et pour des programmes plus conséquents que les applications pair-à-pair, comme les grosses applications MPI utilisées dans l'industrie, on arrive très rapidement à la saturation de la RAM. On a donc envisagé dans un premier temps une distribution de SimGrid pour répartir les processus sur différents hôtes afin de gagner plus de place. Cependant, cela aurait entraîné une profonde modification du code.

Lucas Nussbaum a trouvé une solution alternative avec ramFS qui permet de monter la mémoire RAM d'autres machines et de pouvoir les utiliser comme une simple extension de la mémoire RAM de la machine hôte. D'après les tests préliminaires qu'il a effectués, on obtient un débit vers la RAM distante suffisamment important pour ne pas occasionner une baisse trop importante des performances. Toutefois, les tests ont été faits sur un cluster [6] muni d'Infiband[16] ce qui laisse présager des performances moindres sur les grilles dotées d'un réseau moins efficace.

Cependant du fait de la solution technique déjà existante trouvée par Lucas Nussbaum nous avons décidé de ne pas poursuivre ce volet de mon stage.

6. Un cluster est un système informatique composé d'unités de calcul (micro-processeurs, cœurs, unités centrales) autonomes qui sont reliées entre elles à l'aide d'un réseau de communication

2 Analyse de l'existant et amélioration de la parallélisation

2.1 Mise en place de tests de régression

Depuis l'origine du projet, l'équipe de SimGrid a pris le soin de rédiger de nombreux tests qui servent à la structure d'intégration continue dont bénéficie le projet. Toutefois, ces différents tests ne valident que la correction du code, et en aucun cas la continuité de la performance de celui-ci.

De plus, étant donné que j'ai été amené à faire de grosses modifications du code au niveau du parallélisme du noyau de simulation, j'ai dû mettre en place dans la première partie de mon stage un ensemble de test de régression. Ceux-ci sont écrits en shell et permettent de faire une comparaison entre deux versions récupérées directement sur le dépôt git du projet. Les scripts ont été développés pour être exécutés sur la plate-forme Grid'5000.

2.2 Analyse de l'existant

Toute cette partie de mon travail a consisté à faire une analyse complète et aussi exhaustive que possible du parallélisme du noyau.

J'ai travaillé sur l'exécution du code du protocole Chord fourni dans la partie exemple du dépôt source de SimGrid. Chord est un protocole pair-à-pair permettant de structurer un réseau de manière décentralisée. Chaque nœud de ce réseau possède un identifiant unique de N bits. Chaque nœud connaît son successeur direct, son prédécesseur ainsi qu'un ensemble de nœuds nommés *fingers*. Ceux-ci sont retenus dans la *finger table*. La stabilisation consiste à vérifier qu'il n'existe pas un successeur plus proche dans l'anneau que celui que l'on connaît déjà.

Le scénario est le suivant. À l'instant t=0s, tous les nœuds sont démarrés et l'un d'entre eux initie la création de l'anneau, les autres demandant à le rejoindre. Puis dans la suite de l'expérience, chaque nœud effectuera une stabilisation toutes les 20 secondes, une mise à

jour de la *finger table* toutes les 120 secondes et une recherche de pair aléatoire toutes les 10 secondes. De façon à rendre toutes les exécutions parfaitement reproductible, tous les nœuds feront une recherche de pair sur le même identifiant choisi de manière arbitraire.

2.2.1 Instrumentation du code

Afin de comprendre la différence entre l'amélioration théorique attendue et celle pratique observée lors des expérimentations, j'ai dû comprendre et donc observer les origines du problème. Pour cela, j'ai eu recours à deux outils de mesure différents.

Benchmark des scheduling round

Afin de comprendre les différences majeures obtenues entre les deux exécutions j'ai décidé de commencer par instrumenter le code au niveau des sub-scheduling round. Puisque SimGrid permet la reproductibilité des expérimentations et que le code du protocole Chord intégré dans les exemples de SimGrid est parfaitement déterministe, la charge des *sub-scheduling round* en nombre de processus mais aussi en code exécuté sera parfaitement égal. Ainsi on pourra comparer directement, de *sub-scheduling round* à *sub-scheduling round*, la différence de temps d'exécution.

Puisque l'on souhaite diminuer le temps global pris par SimGrid pour effectuer la simulation, on mesurera le walltime [1]. Comme vu au 1.3.4, la seule différence entre le mode parallèle et le mode séquentiel se situe dans la manière de faire tourner les processus en attente. On placera donc la mesure du temps au début de la *sub-scheduling round* et on la terminera lorsque Maestro franchira la dernière barrière pour sortir de celle-ci. Afin d'obtenir des données exploitables à travers différents tests, tous les temps seront stockés à chaque fois avec le nombre de processus exécutés lors de la sub-scheduling round.

Lors des premiers tests effectués, j'avais utilisé la fonction time de GNU/Linux. Toutefois, devant le manque de précision de celle-ci (certaine *scheduling round* étant en dessous des 1 µs), j'ai dû très rapidement basculer vers l'interface clock_gettime disponible dans la norme POSIX qui permet d'utiliser une horloge beaucoup plus précise.

Profilage du code

Pour profiler le code, j'ai eu recours à deux logiciels différents. Le premier est l'extension Cachegrind du logiciel de débogage Valgrind[17] et le second est l'outil de profilage développé par Intel, Vtune[18]. Chacun est présenté ici avec ses avantages et ses inconvénients.

1. Le walltime correspond au temps réel.

Cachegrind Il a l'avantage d'être libre et gratuit, mais aussi de fournir de façon absolue différentes statistiques sur les appels de fonction, ou sur les sauts. De plus, il est aujourd'hui livré avec de nombreux outils permettant notamment la simulation de cache, ainsi que des estimations sur la prédiction de branchement.

Toutefois, il a de très nombreux défauts :

- il ne prend pas en compte le coût d'une instruction (en cycle), ni l'impact des défauts de cache sur le temps d'exécution d'une fonction,
- il se fait perdre par les changements de contexte opérés dans la fabrique de contexte raw,
- il a un très gros overhead ce qui rend excessivement longue toute séquence de tests.

Pour toute ses raisons, Cachegrind n'a pas été utilisé comme profileur permanent, mais uniquement pour obtenir une vue des différents éléments du code, notamment pour sa représentation très claire des chemins d'exécution.

Vtune Vtune est le profileur de code développé par Intel. Il peut être utilisé dans une version de démonstration complète mais limitée dans le temps. VTune possède une vaste gamme d'analyses disposant chacune de plusieurs niveaux de profondeur. Pour ma part, je me suis concentré sur trois types différents d'analyse :

- celle générale qui indique le temps passé dans chaque fonction ainsi que ligne par ligne, le temps d'attente du processeur ainsi que l'activité *thread* par *thread*,
- celle sur la contention mémoire, et sur les défauts de cache,
- celle sur la prédiction lors des sauts.

Vtune offre également une très bonne performance au niveau de ses analyses basiques et permet donc de faire tourner de façon rapide un nombre conséquent de configurations.

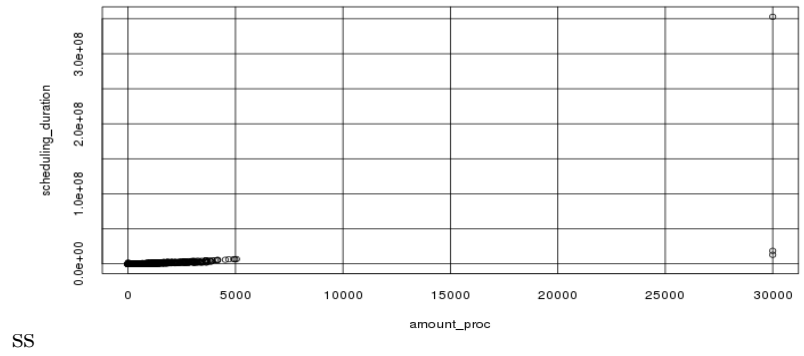
Toutefois, VTune possède ses propres limites. À la différence de Cachegrind, celui-ci ne fait pas une analyse exacte du code et de son exécution mais se contente, à intervalle régulier, de stopper le processeur et de regarder ce qu'il est en train de faire puis il extrapole les différents résultats pour en sortir des statistiques. Ceci pose un problème majeur parce que la comparaison stricte entre deux exécutions différentes ne permet pas d'en tirer des résultats absolus que ce soit en se basant sur les temps obtenus mais aussi sur le nombre d'instructions exécuté par fonction.

De plus, les affichages générés par VTune sont très difficilement compréhensibles et donc interprétables. Ils multiplient les données si bien que l'on se demande parfois quel est la différence entre plusieurs colonnes successives. Étant donné la complexité de l'affichage mais aussi de la répartition entre plusieurs onglets de ceux-ci, les résultats obtenus via ce profileur ne seront qu'évoqués mais ne bénéficieront d'aucun graphique.

Du fait des nombreuses limitations de chaque profileur, ceux-ci serviront pour les analyses globales du code et non pour détecter les optimisations pures, celles-ci seront observées par le biais d'analyses statistiques sur plusieurs exécutions.

2.2.2 Analyse des résultats des benchmarks

Dans toute cette partie, les temps sont fournis en ns. L'expérience a porté sur la simulation d'un système à 30000 nœuds utilisant le protocole Chord selon le schéma décrit avant. Durant cette phase d'analyse, j'ai utilisé la fabrique de contexte raw et la synchronisation *futex* (configuration de base). Commençons par analyser les résultats obtenus en séquentiel. Voici l'affichage rapide en nuage de points :



SS

FIGURE 2.1 – Nuage de points des résultats de l'exécution séquentielle.

La figure 2.1 permet de voir plusieurs choses intéressantes. Tout d'abord, la majorité des sub-scheduling round possèdent un nombre de processus à tourner inférieur à 5000 processus. Ensuite, les temps d'exécution sont très courts, de l'ordre de la milliseconde.

Une analyse rapide des données permet de détecter que le nombre de *sub-scheduling round* avec un nombre de processus à simuler supérieur à 200 représente en réalité 12784 *sub-scheduling round* sur les 1 304 010 qui composent la totalité de la simulation soit moins de 0.01%. Par conséquent, il sera plus intéressant de reprojeter l'analyse que sur les *sub-scheduling round* faisant tourner moins de 200 processus (figure 2.2).

Maintenant, faisons la même analyse avec le mode parallèle. Par soucis de comparaison les différentes données seront affichées en juxtaposition avec les données séquentielles.

Une simple lecture de la figure 2.3 permet de relever la différence qui existe entre les deux modes au niveau du temps d'exécution des trois *sub-scheduling round* les plus chargées. De même, le nuage de points semble plus tassé du côté du mode parallèle que du côté séquentiel. En revanche, ce qui se passe pour les *sub-scheduling round* maigres ne peut être observé sur

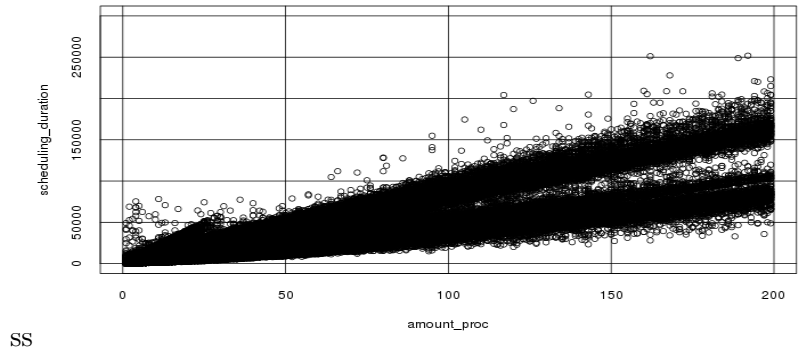


FIGURE 2.2 – Zoom sur les *sub-scheduling round* de moins de 200 processus.

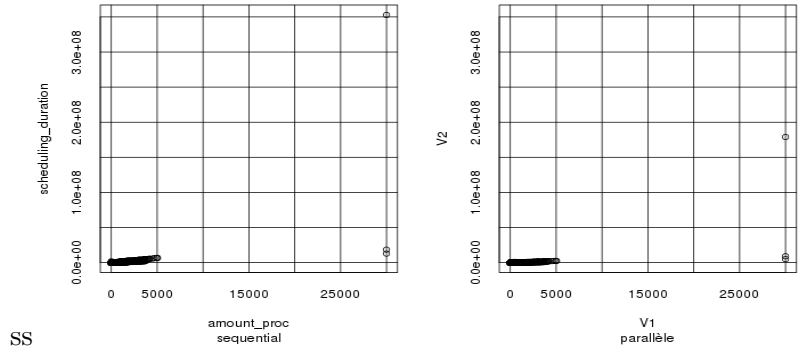


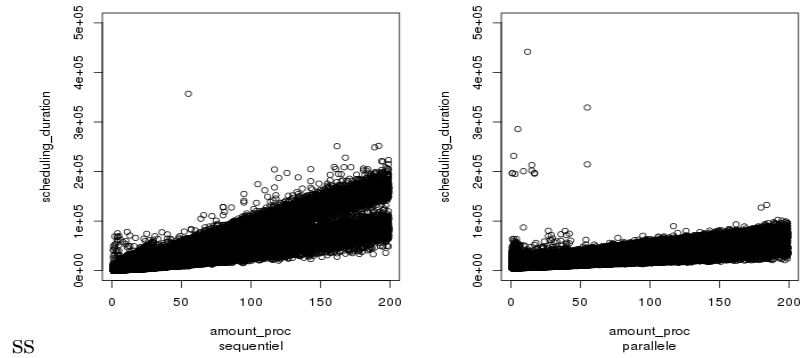
FIGURE 2.3 – Comparaison sur des deux modes d'exécutions.

ce schéma. Agrandissons donc les graphiques pour n'afficher plus que celles ayant moins de 200 processus à exécuter (figure 2.4).

Ici, les différences entre le mode parallèle et le mode séquentiel sont beaucoup plus criantes et visibles. Dans les *sub-scheduling round* maigres, le séquentiel est légèrement plus rapide que le mode parallèle. En revanche, lorsque celles-ci deviennent chargées, le mode multi-threadé devient beaucoup plus performant. Cela provient de plusieurs facteurs tous inhérents à la synchronisation entre les différents *workers* lors de l'exécution des processus simulés.

Le premier est le coût du réveil et de l'attente en début et fin de cycle. En effet, au début de l'algorithme, Maestro doit réveiller tous les *workers* avant de commencer le travail. De même, à la fin, il devra attendre que tous les *thread* de la parmap aient terminé leur travail avant de pourvoir continuer.

Le second est le coût qui provient de la contention mémoire lors de l'accès à la variable commune, dans la parmap qui permet de retenir le prochain processus à lancer. Bien que



SS

FIGURE 2.4 – Comparaison sur des deux modes d'exécutions sur les *sub-scheduling round* à moins de 200 processus.

l'incrémentation et la lecture soient atomiques et réduites à une instruction processeur, celles-ci provoquent une exclusion mutuelle et donc des ralentissements lors de l'accès à cette zone.

Par conséquent, le mode parallèle n'est pas directement avantageux sur les *sub-scheduling round* très maigre. Il faut donc le privilégier sur les simulations comportant un grand nombre de processus ou sur celles faisant intervenir une charge de calcul au niveau de l'application simulée suffisamment importante. Toutefois, il serait plus avantageux de trouver un moyen de combiner les deux modes, pour en tirer le meilleur temps possible. Une première tentative a été faite (section 1.3.4) mais possède deux problèmes majeurs.

Le premier, évident, est qu'il faut donner une valeur à ce seuil. Une manière simple de le faire serait de refaire exactement ce qui vient d'être fait, de tourner deux simulations, une dans chaque mode, en instrumentant le code à chaque fois, puis de calculer de manière statique le seuil, et enfin de le donner à SimGrid au moment de l'initialisation. Toutefois, cette méthode présente l'immense désavantage de nécessiter de simuler au moins deux fois avant d'obtenir une simulation performante. De plus, il faudrait aussi retourner ces benchmarks à chaque modification sensible de la charge applicative, ce qui en phase de développement, se produit souvent.

Le second est que la charge applicative à exécuter n'est pas non plus nécessairement stable dans le temps. En effet, suivant les processus des applications on pourrait passer d'une phase avec peu de calculs comme par exemple une phase d'envoi de données d'un serveur aux applications, à une phase de calculs intensive avec peu d'interruptions, puis à une phase de calcul à nouveau faible.

Ces deux éléments tendent à discréditer l'idée du seuil statique. C'est pourquoi, afin de pouvoir marier les deux modes de façon plus satisfaisante, j'ai développé un algorithme

adaptatif et dynamique qui permet de mettre à jour continuellement le seuil de manière dynamique sans devoir faire intervenir l'humain.

2.2.3 Algorithme de seuillage automatique

Cet algorithme doit posséder plusieurs éléments plus ou moins prépondérants :
- il doit pouvoir se mettre à jour très rapidement,
- il doit minimiser le nombre d'erreurs de mesure,
- il doit permettre de gagner plus de temps que ce qu'il n'en fait perdre lors de ses mesures.

Description de l'algorithme

Dans un premier temps, je me suis penché sur les différentes composantes des temps dans les deux différents modes. Dans le mode séquentiel, le temps est linéaire. celui-ci se compose ainsi :

$$temps\_sequentiel = \sum_{0}^{\#proc} (simulation\_time + changement\_contexte) + changement\_contexte$$

$$\Rightarrow temps\_sequentiel = \sum_{0}^{\#proc} (temps\_moyen\_processus\_sequentiel) + changement\_contexte$$

On a donc une composante fixe qui est le tout premier changement de contexte lorsque Maestro commencera à travailler sur les processus à simuler. $Temps\_moyen\_processus\_sequentiel$ correspond au temps moyen que prendra Maestro pour effectuer le changement de contexte et pour exécuter le processus. Le temps de changement de contexte étant très faible, on peut assimiler le temps en séquentiel à une fonction linéaire ne dépendant que du nombre de processus et du temps moyen d'exécution

On cherche donc un moyen de trouver aussi dans le mode parallèle une fonction du type $temps\_parallele = nombre\_processus * temps\_moyen\_processus\_parallele$.

La difficulté est donc de déterminer le temps moyen pris pour l'exécution d'un processus en parallèle. Malheureusement, ce temps n'est pas accessible à partir de n'importe quelle mesure. Dans le mode séquentiel, on peut raisonnablement estimer que le temps moyen mesuré pour une *sub-scheduling round* de 20 processus restera cohérent avec celui mesuré pour une de 60 processus. Cependant dans le mode parallèle, il est impossible de prendre le temps d'une *sub-scheduling round* à 2000 processus pour extrapoler le temps pris à 30 processus.

Toutefois, on doit trouver un moyen de comparer les temps d'exécution des deux modes pour pouvoir en déduire un seuil. De plus, il faut trouver un moyen de faire de manière rapide les mesures nécessaires à la mise à jour du seuil. Ce qu'on cherche en réalité, c'est à détecter la taille à partir de laquelle une *sub-scheduling round* est plus rapide en mode parallèle qu'en mode séquentiel. On va donc vérifier juste à la frontière le temps que prennent les *sub-scheduling round* (pour chaque mode) puis on ajuste le seuil en fonction des temps mesurés.

Cette méthode pose cependant un très gros problème, il faut avoir des mesures au niveau du seuil, ce qui entraîne une plus grande durée avant la mise à jour du seuil. On fera donc une approximation de linéarité sur une petite fenêtre, en considérant que le temps en mode parallèle est linéaire sur cette fenêtre.Toutefois, cela pose le problème de la correspondance de la mesure, puisqu'une mesure à seuil-3 processus sera forcément inférieur à une mesure à seuil+3 processus. On utilisera donc comme élément pour la comparaison le temps moyen d'exécution par processus. Voici une description en pseudo code de cet algorithme.

```
Initilialisation:
  seuil = 2    //on fixe le seuil intial à deux -> il est toujours intéressant
       //d'exécuter en parallèle si on a plus d'un processus à simuler
Variable:
  tps_seq = 0  //accumulateur du temps séquentiel
  nbr_seq = 0  //somme du nombre de processus exécuté dans la fenêtre séquentielle
  moy_seq      //temps moyen d'exécution d'un processus en séquentiel
  tps_par = 0  //accumulateur du temps parallèle
  nbr_par = 0  //somme du nombre de processus exécute dans la fenêtre parallèle
  moy_par      //temps moyen d'exécution d'un processus en parallèle
  fenêtre      //dimension de la demi-fenêtre
  seuil        //seuil actuel
  #proc        //nombre de processus à exécuter
Code:
  si #proc < seuil - fenêtre
    exécution_séquentiel

  sinon si #proc > seuil + fenêtre
    exécution_parallèle

  sinon si #proc < seuil
    lancement du chronomètre
    exécution_séquentielle
    arrêt du chronomètre
    tps_seq += chronomètre
    nbr_seq += #proc

  sinon
    lancement du chronomètre
```

```
    exécution_parallèle
    arrêt du chronomètre
    tps_par += chronomètre
    nbr_par += #proc

  si assez de mesure
    //on met à jour le seuil
    moy_seq = tps_seq / nbr_seq
    moy_par = tps_par / nbr_par

    si moy_par < moy_seq
      seuil--
    sinon
      seuil++

    tps_seq = 0
    nbr_seq = 0
    tps_par = 0
    nbr_par = 0
```

Cet algorithme possède déjà deux paramètres à fixer. Le premier est le nombre de mesures à effectuer avant de mettre à jour le seuil. En effet, il faut trouver un compromis entre la stabilité du système pour éviter les sursauts du seuil qui tendraient à rendre le tout instable, mais un nombre trop grand d'échantillons provoquerait une trop grande lourdeur dans la mise à jour du seuil et empêcherait toute flexibilité du système. Pour le moment, ce nombre d'échantillon a été fixé à 5 pour chacun des deux modes. Bien évidemment, on continue à intégrer les nouveaux échantillons à l'ensemble si jamais ce seuil est dépassé d'un côté ou de l'autre.

Le second paramètre est celui de la taille de la fenêtre. Le problème est un peu plus complexe parce que les paramètres se multiplient. Limiter la fenêtre donne un avantage évident, celui de rendre plus juste l'approximation de linéarité et donc de donner une valeur beaucoup plus juste au seuil. De plus, limiter la fenêtre entraîne une diminution du nombre de chronométrage et donc le temps pris par l'algorithme lui-même. En revanche, réduire la fenêtre de trop limiterait grandement les mesures et donc diminuerait de façon dramatique la rapidité de mise à jour du seuil. Cette fenêtre a été fixée à 5 de chaque côté du seuil.

Plusieurs points ont variés entre la version de base de l'algorithme et la version finale.

– La méthode de calcul de la moyenne dans la première version de l'algorithme a été abandonnée au profit de la moyenne géométrique qui permet de donner une prédominance aux mesures récentes et donc de prendre en compte une variation sur la période très proche de la mise à jour du seuil malgré la multiplication des mesures.
– La mise à jour du seuil est très lente, et même si celle-ci intervient très rapidement, on se retrouve à devoir attendre un long moment avant que le seuil ne se stabilise. Aussi j'ai eu l'idée d'utiliser une mise à jour qui prenne en compte l'écart de temps entre le séquentiel et le parallèle. Ainsi, le seuil est actualisé en utilisant un coefficient arbitraire que l'on va multiplier avec le rapport des temps (fait de telle sorte qu'il soit supérieur à 1). Ainsi la variation du seuil est bien plus importante lorsque l'écart est important et devient plus fine, voire nulle lorsque la différence est stabilisée.
– Afin de permettre au seuil d'atteindre très rapidement son niveau de "croisière", une phase d'initialisation rapide a été imaginée pour permettre une mise à jour plus rapide au tout début sans nous obliger à utiliser un coefficient de mise à jour trop grand et donc provoquer des mises à jour beaucoup trop brutales. Ainsi, la solution envisagée est d'utiliser au tout début un coefficient plus important puis lui donner une valeur plus faible permettant ainsi d'atteindre une certaine stabilité. J'ai imaginé plusieurs critères d'arrêt, celui que j'ai retenu serait de considérer la fin du démarrage lorsque la courbe s'inverse pour la première fois, c'est-à-dire lorsque le seuil doit diminuer pour la toute première fois.
– Une sécurité a été ajoutée à l'algorithme pour éviter que le seuil ne tombe en dessous de deux ce qui empêcherait la poursuite de celui-ci (plus aucune mesure ne serait effectuée pour le séquentiel). Ainsi, quoiqu'il arrive le seuil ne peut tomber pas plus bas que deux.

Résultats

Les premiers résultats obtenus en exécutant l'algorithme sur les fichiers de logs générés par les benchmarks sont très encourageants. On obtiendrait ainsi :

Algorithme	Temps séquentiel	Temps parallèle	Temps hybride	Meilleur temps
Basique	17.33	21.18	12.12	11.64
Évolué	17.33	21.18	12.05	11.64

TABLE 2.1 – Résultats du test de l'algorithme adaptatif sur les logs.

La colonne meilleur temps donne le temps que l'on aurait eu si on avait toujours exécuté le meilleur temps.

2.2.4 Validation de la configuration de "Maestro as a worker"

Durant ma mission à Grenoble, nous avons fait un inventaire assez profond de la configuration actuelle du mode parallèle avec Augustin. Parmi cette configuration nous avons décidé de vérifier le gain obtenu en utilisant Maestro comme un worker. En effet, (cf 1.3.4) lorsque Maestro a fini de réveiller tous les workers, celui-ci commence le travail en tant que simple worker. Si cela permet d'éviter un appel système au lancement des exécutions en ne réveillant que n-1 *threads* pour n demandés dans la ligne de commande, on peut se demander si cela n'est pas préférable d'avoir un processus prêt à être lancé au moment où le dernier *worker* termine son travail.

Après plusieurs séquences d'expérimentation, on obtient les résultats présentés dans le tableau (les tests ont été effectués sur ma machine de travail)

Nombre *thread*	Maestro configuration		
	As a *worker*	En veille et remplacé	En veille et non remplacé
2	335.96	940.3	1029.395
3	332.375	894.015	923.93
4	332.83	848.665	943.585

TABLE 2.2 – Résultats des tests sur les différentes configurations de Maestro.

2.2.4

Cela permet donc de valider l'utilisation de Maestro comme un worker.

2.2.5 Analyse des modes de synchronisation

En réalité, SimGrid propose trois modes de synchronisation accessible via l'argument de ligne de commande *–cfg=contexts/synchro*. Chaque mode est présenté ici avec ses faiblesses et ses avantages, puis une analyse des résultats sera faite dans une seconde partie.

Futex Le mode *futex* est la synchronisation par défaut utilisée par SimGrid. Celle-ci se base sur l'appel système *futex* du noyau Linux. Grâce à celui-ci, on peut obtenir une mise en veille du processus qui sera réveillé par un autre. Les *futex* permettent aussi une modulation sur la quantité de *thread* qui se réveille et aussi d'éventuellement éviter de passer par le noyau Linux puisque plusieurs tests sont effectués sur le paramètre d'attente avant la mise en veille effective du thread.

Toutefois, cette méthode a deux inconvénients majeurs. Le premier, est que reposant sur un appel système de GNU/Linux, le code devient alors dépendant du système d'exploitation. Or l'équipe tente d'exporter le code sur différentes plates-formes (Windows, Mac OS) et l'utilisation des *futex* sur les autres plates-formes n'est pas envisageable. De même, bien que

fait pour la vitesse, les coûts de réveil des *workers* ainsi que de l'attente finale sont relativement élevés.

Posix Cette synchronisation est basée sur une bibliothèque POSIX et repose sur l'utilisation de mutex qui sont nettement plus répandus que les *futex* (en l'occurrence elle utilise l'implémentation des mutex fournis par la librairie pthread). Cette méthode bénéficie de la possibilité de mise en veille des *threads* tout en permettant une plus grande portabilité (une surcouche garantissant cette portabilité a été développée par l'équipe).

Comme la synchronisation par *futex* , celle par les mutex posix possède le désavantage d'avoir des coûts de synchronisation pouvant être élevés et qui devront être comparés à ceux des *futex* .

Busy_wait Cette synchronisation là ne repose sur aucune synchronisation particulière parce qu'il va ici s'agir d'utiliser une attente active plutôt que de mettre en veille les threads. Le seul avantage que pourrait offrir cette méthode est la diminution des coûts de synchronisation, puisque les *workers* ne sont jamais en veille et testent en permanence la présence de travail.

Toutefois, cette méthode présente un désavantage certain qui est la surconsommation énergétique de l'hôte. En effet, au lieu d'avoir Maestro qui travaille la plupart du temps, et des *workers* qui se lancent épisodiquement, on a un ensemble de *thread* utilisant le maximum des capacités du processeur.

J'ai effectué des expériences sur 100 000 nœuds afin de comparer les différents mode de synchronisation. Les résultats obtenus sur le cluster graphene [2] sont affichés dans le tableau 2.3.

Nombre *thread*	Mode de synchronisation		
	Busy_wait	Futex	Posix
Séquentiel	310.8700		
2	309.2475	329.9875	332.4400
3	299.7725	328.6800	337.9225
4	294.2475	318.9800	336.1175

TABLE 2.3 – Résultats des tests sur les différents modes de synchronisation proposés par SimGrid.

La première analyse que l'on peut pourrait effectuer se situe sur la faiblesse énorme du mode posix sur le mode *futex*. Cela valide donc bien son statut de mode de réserve, pour

2. Machine du cluster Nancéien de Grid'5000

garantir une certaine portabilité.

La seconde analyse se porte elle sur la domination du busy_wait sur le *futex* . En effet, celui-ci permet d'obtenir une amélioration grâce à la parallélisation là où le *futex* ne permet pas d'égaler le temps en séquentiel.

Ces résultats tendent à prouver que le *futex* n'est pas forcément le plus désigné pour être le type de synchronisation dans SimGrid. Toutefois, l'utilisation de l'attente active entraîne une explosion du temps CPU et de la consommation.

Cette constation a donné lieu à une réflexion sur la possible utilisation de l'attente active comme synchronisation par défaut. Toutefois, il faudrait dans un premier temps parvenir à limiter le nombre de *worker* en attente active en fonction de la quantité de travail. Cela reviendrait à maintenir une partie des *threads* en attente réelle et à ne les faire basculer en attente active pour la réalisation de travail que lorsque la charge s'en fait ressentir. En gardant ainsi juste une partie des *threads* en attente active, on économise l'énergie tout en gardant des performances acceptables. Étant donné que la quantité de travail n'est pas prévisible d'une *scheduling round* à une autre, il a été décidé d'appliquer un principe de développement système, à savoir que le futur ressemblera au présent. Ainsi on allumera et éteindra des *threads* uniquement quand le besoin s'en fera sentir et non par anticipation.

Cette partie de mon travail n'étant pas fini au moment de la rédaction de ce rapport, je n'ai pas pu intégrer les résultats de cette idée.

2.3 Maestro éteint la lumière

Dans la perspective d'amélioration du mode parallèle et de la chasse aux appels systèmes coûteux, l'équipe a imaginé une solution pour éviter la synchronisation finale. Le principe est simple, si le dernier processus à terminer une exécution est Maestro, il n'y a pas besoin d'effectuer la dernière synchronisation. La difficulté réside dans le fait qu'il faut être sûr que le dernier *worker* est bel et bien Maestro. Comme le montre l'appendice A, cet algorithme ne peut être implémenté que dans les fabrique de contexte "raw" et "ucontext".

Il est en réalité impossible sans effectuer de synchronisation d'assurer que le *thread* qui effectue l'exécution du dernier processus à simuler, soit Maestro. En revanche, il est possible de faire en sorte que le dernier *thread* devienne Maestro. En effet, à chaque changement de contexte, on met à jour le *thread* avec une nouvelle pile et des nouveaux registres. Or c'est ce couple (pile, registre) qui déterminera en réalité ce qu'est un thread. N'importe quel *thread* qui possédera le couple (pile, registre) de Maestro sera Maestro peut importe ce qu'il était avant le changement de contexte. On ne considère pas ici l'espace mémoire puisqu'il est com-

mun (voir A.3).

Un premier algorithme simple consisterait donc à faire en sorte que tous les *workers* sauf le dernier (Maestro est ici considéré comme un *worker* jusqu'à ce qu'il retrouve son contexte) restaurent une identité quelconque de worker et le dernier doit lui récupérer le contexte de Maestro devenant ainsi ce dernier et bouclant alors la sub-scheduling round en se passant de la barrière finale puisqu'il est effectivement le dernier. On aurait donc l'algorithme suivant :

- au moment du premier changement de contexte, il faut stocker le contexte des *workers* et de Maestro de telle manière à pouvoir distinguer Maestro des autres,
- lors de la restauration du contexte, on doit donner au premier les contextes des workers, et au dernier on lui donne le contexte de Maestro.

Malheureusement cet algorithme possède un immense problème : deux *threads* ne peuvent pas utiliser la même pile et cela pour des raisons évidentes de sécurité mais aussi pour des questions de cohérence. Par conséquent, il faut être certain que la pile que l'on va restaurer n'est effectivement plus utiliser par le *worker* dont on prend la place. Cela n'est possible que si l'on rajoute une synchronisation ce qui est évidemment le contraire de ce qui est recherché. Une seconde méthode a donc été imaginée.

Le problème du premier algorithme reposait sur le fait que les *workers* et Maestro s'échangeaient leur contexte. Prenons le problème d'une autre façon. Ce que l'on désire en réalité c'est que les *threads* soient des *workers* et que l'un d'entre eux en fonction des circonstances soit Maestro. De ce point de vue apparaît une toute autre solution : il suffirait de munir chaque *thread* d'une identité de worker, Maestro y compris et ainsi à la fin de la *sub-scheduling round* lorsque viendrait le moment de restaurer leur contexte, ceux-ci restaureraient soit leur propre identité de *worker* sans risquer de prendre une pile non libérée, soit de restaurer Maestro pour le dernier thread. La dernière situation problématique reste est le cas où Maestro n'a pas le temps d'effectuer un changement de contexte avant que les *workers* ne terminent le dernier travail. Ainsi, on devra quand même passer par une synchronisation finale.

C'est à la parmap que revient la responsabilité de gérer les contextes. En se basant sur le schéma d'exécution pour le moment utilisé, il a été décidé de rajouter des cases vides pour indiquer à un travailleur qu'il doit reprendre son identité de worker, et renvoie l'identité de Maestro lorsqu'on arrive au dernier. Aussi faut-il que Maestro indique à la parmap son contexte avant de faire son premier changement.

2.4 Binding de processus

Au cour de mon séjour à Grenoble, nous avons, avec Augustin, découvert grâce à VTune que certaines fonctions obscures prenaient du temps CPU que se soit en mode séquentiel ou en mode parallèle. Après une rapide investigation, ces fonctions s'avéraient être des fonctions servant lors de la migration d'un processus entre les différents cœurs du processeur. J'ai alors décidé de tester l'affectation de *thread* sur cœur pour voir l'impact de ces déplacements sur les performances de la simulation.

J'ai testé deux configurations différentes pour obtenir une comparaison :

- la première qui est la configuration actuelle et qui consiste à laisser l'ordonnanceur du système d'exploitation choisir où sont exécutés les processus,
- la seconde qui consiste à affecter chaque *thread* à un cœur différent.

Nombre *thread*	Futex		Busy_wait	
	Sans affectation	Avec affactation	Sans affectation	Avec affactation
2	335.96	330.05	317.525	312.56
3	332.375	333.13	371.5	389.75
4	332.83	328.975	383.04	384.78

TABLE 2.4 – Résultats des tests sur les affectations.

Les données montrent que les meilleurs résultats sont atteints lorsque l'on affecte manuellement les *workers* sur les cœurs du processeur plutôt que de laisser l'ordonnanceur faire. De plus, bien que nécessitant une couche de portabilité cette option peut être utilisée sur différentes plates-formes puisque tous les systèmes d'exploitation récents possèdent cet option.

Les résultats proposés plus haut montrent de singulière variation en comparaison des résultats visible dans le tableau 2.3. Cela provient en réalité de l'architecture des machines. Celle sur laquelle je travaille possède une architecture appelée hyperthreading qui consiste à simuler la présence de deux cœurs sur un seul cœur physique. Dans le cadre d'une utilisation classique, cela permet d'obtenir des performances plus élevées qu'avec un seul cœur. Toutefois, dans le cas d'application faisant beaucoup de calcul et multipliant les lectures/écritures en mémoire, cela provoque des chutes importantes de performance. En comparaison, une machine qui n'aura pas l'hyperthreading aura un comportement plus intuitif (visible dans les résultats du tableau 2.3).

De plus, ma machine possède une charge à vide (interface graphique, programmes, ...) beaucoup plus importante que graphene, ce qui, lorsque que le nombre de *thread* augmente, vient perturber les temps.

Toutefois, les conclusions présentées dans ce rapport sont vérifiables quelque soit le type d'architecture (hyperthreadée ou non) et les expériences n'ont pas été retournées sur graphene pour pouvoir économiser du temps.

2.5 Rédaction de l'article

L'objectif de cette partie de mon stage était de préparer la rédaction d'un article de conclusion sur les travaux de parallélisation du noyau de simulation de SimGrid. C'est dans ce but qu'ont été réalisées toutes mes analyses ainsi que mes recherches pour comprendre les points noirs de l'implémentation actuelle.

2.5.1 Calcul de la loi d'Amdahl

Le loi d'Amdahl est une loi présentée en 1967 pour permettre de calculer l'amélioration théorique maximale que l'on peut obtenir en utilisant un système multi-processeur. Cette loi prend en compte que seule une partie du programme peut être parallélisée. La loi est la suivante :

$$T(n) = T(1) * (B + \frac{(1-B)}{n})$$

où $T(n)$ représente le temps pris par le programme pour n thread, et où B est la fraction de programme qui est parallélisée. Reste donc à trouver ce facteur de parallélisme.

Afin de trouver ce seuil, il a fallu instrumenter le code une nouvelle fois pour faire un calcul du temps passé dans les sections séquentielles et dans les sections parallélisées. Reste le problème des *sub-scheduling round* à un seul processus. D'un point de vue strict, il devrait être compté dans la partie parallèle puisque l'exécution des processus simulés est parallélisée, mais d'un autre côté, il n'y en a qu'un, donc l'exécution ne pourra jamais être parallèle. Aussi j'effectuerai les deux mesures, une en excluant le temps des *sub-scheduling round* à un processus du temps parallélisable, l'autre en l'incluant.

Je n'ai pas encore pu faire cette partie de mon stage.

2.5.2 Implémentation du protocole dans une API concurrente pour la comparaison

Lors de l'écriture[13] l'équipe a souhaité, dans un souci d'honnêteté fournir l'ensemble des codes sources utilisés dans les différentes expériences qui ont permis de faire la figure de comparaison avec la concurrence. Malheureusement, une erreur a été faite lors de ces expérimentations, et au lieu d'exécuter le code du protocole Chord fournit par Peersim[19], ils ont utilisé le code de Pastry, un autre protocole de DHT lui aussi fournit par Peersim.

Bien qu'une mesure de la charge par l'équipe a permis d'établir que les résultats n'étaient pas totalement erronés et restaient dans le même ordre de grandeur puisque la charge était équivalente, ils ont décidé tout de même de retourner les expériences avec le code du protocole Chord fournit par Peersim.

J'ai dû me charger de retourner ces expériences pour pouvoir obtenir les vrais temps pour la comparaison. Après une rapide inspection du code, j'ai constaté que seulement une infime partie du protocole était implémentée et devant la complexité du code fourni, j'ai décidé de réécrire complètement le code pour coller au protocole tel qu'il a été décrit de façon à correspondre à la charge que nous imposons à notre propre simulateur.

Cependant après avoir effectué les premiers tests, je me suis retrouvés avec un code plus performants que celui de SimGrid malgré la génération de plate-forme aléatoire incluse dans le code Peersim. On obtient ainsi les temps suivants visibles dans le tableau 2.5.

Quantité de nœuds	Simgrid	Peersim
10000	19.90	6.06
30000	74.24	27.58
100000	330.05	198.73

TABLE 2.5 – Résultats comparatif de Peersim.

Cette différence s'explique pour une raison très simple, les deux simulateurs n'utilisent pas le même paradigme de simulation. Alors que SimGrid utilise des contextes qu'il va ensuite faire tourner comme le ferait un système d'exploitation, Peersim utilise le paradigme "event-driven". Les processus en eux-même ne font que répondre à des événements programmés dans le temps et en programment d'autres, les processus ne récupèrent pas la main une fois leurs actions effectuées, mais ne sont réveillés que par un événement. De plus, le moteur de simulation permet de faire plusieurs choses étranges.

Dans un premier temps, tous les nœuds du réseau ont une capacité infinie dans le sens où ils peuvent recevoir plusieurs communications simultanément et répondre à toutes ces communications dans le même temps. Contrairement à SimGrid, il n'y a aucune notion de contention, de partage de ressource, ce qui rend nécessairement le moteur plus rapide.

Cela pose donc un problème pour la comparaison avec SimGrid. Bien qu'il soit sur le papier plus rapide que lui, Peersim permet de faire beaucoup moins, et ne présente en aucun cas les réalités des contentions sur un réseau mais aussi sur une machine ce qui pose un problème au niveau du réalisme diminuant ainsi l'intérêt de ce simulateur pour le développement algorithmique.

J'ai aussi effectué une étude sur le code source du protocole Chord fourni par l'équipe Peersim qui obtient des performances moindres que mon code, mais aussi sur le code de Pastry qui avait été utilisé lors de l'article afin de comprendre d'où vient l'écart de performance. Le problème vient du fait que les versions fournis par l'équipe développant Peersim gardent en mémoire tous les messages afin de faire des statistiques ce qui à terme provoque une explosion de la place en mémoire, mais aussi cause une augmentation sensible du temps nécessaire à la simulation de l'algorithme.

3 Compression des scheduling round

Cette partie de mon stage a pour but d'augmenter les capacités de parallélisation du noyau de simulation de SimGrid en tentant de regrouper les sub-scheduling round. Cette idée a été trouvée par Martin. Bien que n'ayant pas encore pu faire un test réel du concept, je présenterai dans cette partie les travaux préliminaires que j'ai déjà effectués. Toutefois, je n'ai pas encore pu aborder de manière poussée cette partie de mon stage.

3.1 Problématique

L'idée naît d'une constatation simple : si l'amélioration due au parallélisme n'est pas intéressante, c'est parce que la plupart des *sub-scheduling round* sont maigres et ne permettent donc pas d'obtenir des gains lors de l'exécution de celles-ci en parallèle. Ceci est très visible à la figure 3.1. L'idée est donc de trouver un moyen de faire grossir les listes de processus à faire tourner ce qui entraînerait dans le même temps une diminution du nombre de *sub-scheduling round* ce qui pourrait également être bénéfique dans le temps.

Pour le moment, le moteur de simulation traite chaque processus de la façon suivante :

- le processus fait partie de la liste des processus à exécuter et il est donc lancé,
- il s'exécute jusqu'à ce qu'il rencontre une action nécessitant un passage par le noyau (communication, simulation de calcul ...),
- le processus est mis en veille jusqu'à ce que l'action se termine dans le monde simulé,
- l'action est injectée après traitement dans le noyau,
- le processus est de nouveau ajouté à la liste de la prochaine *sub-scheduling round* lorsque l'action est terminée dans le monde simulé.

Dans la partie 1.3.3, on a vu que certaines actions obtenaient une réponse immédiate. De la même façon, il existe des actions auxquelles on peut répondre avant que celles-ci ne soient terminées dans le monde simulé. Prenons l'exemple d'une communication. Le schéma classique est présenté à la figure C.1.

Toutefois, dès l'instant où le processus fait son envoi, on sait quelles données recevra le receveur et quelle valeur de retour obtiendra l'émetteur. Ainsi, on peut largement envisager

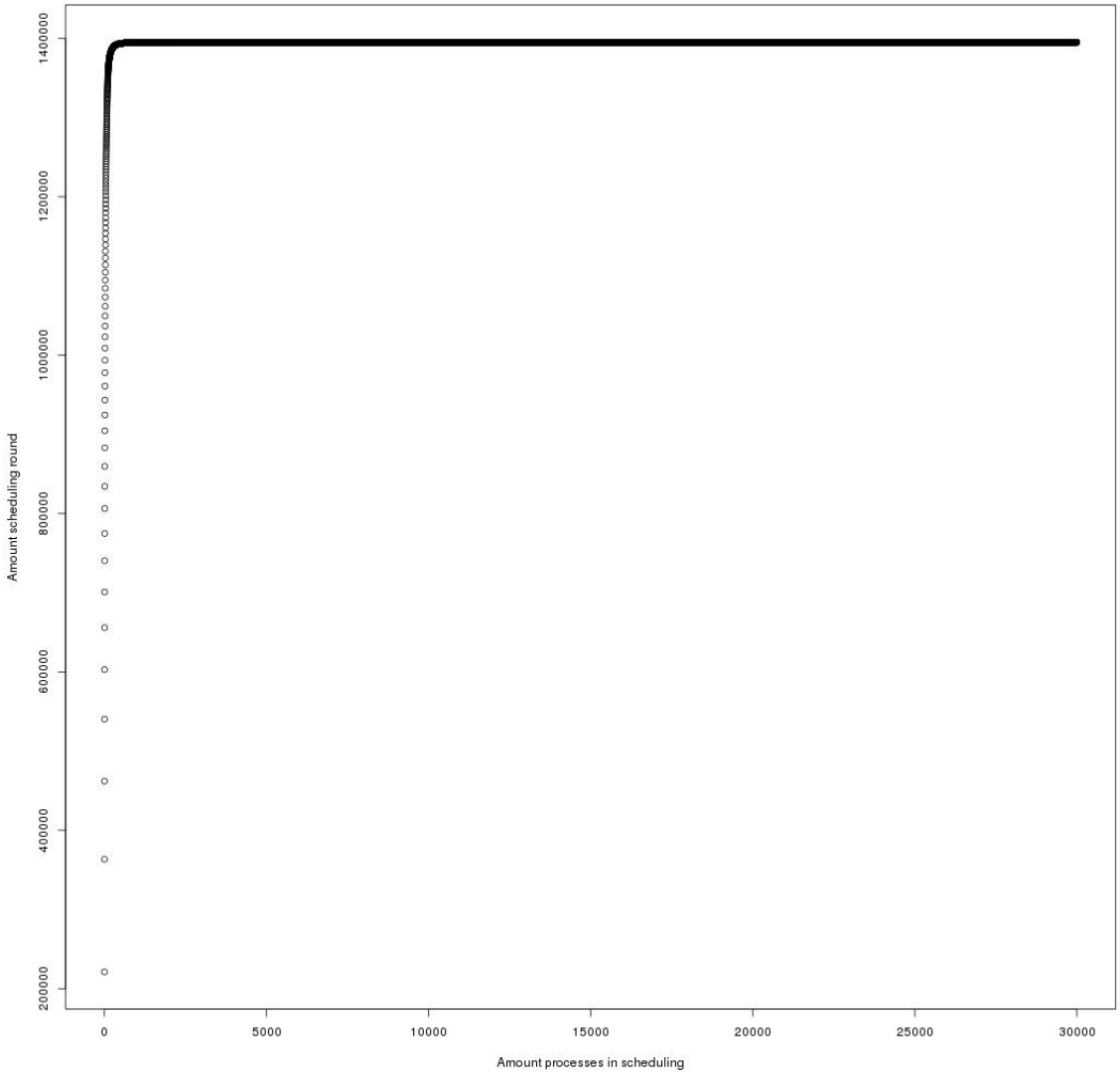


FIGURE 3.1 – Somme cumulative du nombre de *sub-scheduling round* par taille.

d'ordonnancer de nouveau ces processus sans attendre la fin de la simulation de la communication. Cela permettrait de retirer ces processus des sub-scheduling rounds où ils auraient du être pour les insérer dans une autre.

La difficulté de la compression consiste à trouver les actions qui permettent une anticipation de l'état futur d'un processus mais aussi de trouver un moyen de marquer les processus qui sont en exécution anticipée. En effet, lorsqu'un processus est réveillé en avance, toutes ses interactions avec son environnement ne se produiront que dans le futur. Il faut donc pouvoir faire en sorte que les actions générées par un processus dans cet état ne soient pas répercutées

dans le noyau directement ou alors en tenant compte du délai.

3.2 Compression de sub-scheduling round

Afin de résoudre ce problème j'ai imaginé une solution simple pour mettre en place l'exécution conditionnelle. Une fois une situation amenant à une possibilité d'anticipation détectée suite à une action A, on marque le processus comme étant un exécution conditionnelle, puis on le réinjecte dans la *sub-scheduling round* suivante. Le processus est exécuté et tombe sur une action B. À ce moment là, le processus qui est en état conditionnel se met en veille quelque soit l'action effectuée, et il mémorise l'action B. Arrive enfin le moment où la première action, l'action A, se termine. À ce moment là, le processus est toujours marqué en état d'exécution conditionnelle. On prend donc l'action mémorisée dans le contexte du processus pour l'injecter dans le noyau de simulation et on lève le caractère conditionnel du processus. Un exemple de compression est montré dans la figure C.2. Celui-ci correspond à la compression de l'exemple précédent. On obtient alors le diagramme d'activité visible sur la figure B.2

Plusieurs améliorations relativement simples peuvent être apportées afin de rendre l'algorithme plus performant :

- faire le tri des actions entre celles qui deviennent réellement bloquantes et celles auxquelles on peut répondre tout de suite malgré l'exécution anticipée,
- vérifier le caractère conditionnel de l'action B pour voir si on ne peut pas laisser le processus en exécution conditionnelle et le réinjecter dans l'ordonnanceur.

La principale difficulté de la compression réside dans la détection des actions auxquelles on peut répondre de manière anticipée sans risquer de se tromper.

La première conséquence de cette compression, est qu'il faut faire la supposition qu'une machine ne tombera jamais en panne, puisque cet état interrompt brutalement l'exécution d'un processus. Par exemple dans le cas d'une transmission, si jamais le récepteur tombe en panne, l'émetteur recevra un message d'erreur si cela se produit pendant la transmission, mais si elle se produit juste après la fin de la transmission, il ne sera jamais au courant. Il faudrait donc pouvoir prévoir la fin d'une transmission ce qui est par essence impossible puisque cette transmission dépend notamment de la charge réseau qui est imprévisible. Il faut donc faire l'hypothèse suivante, une machine ne tombe pas en panne.

D'autres contraintes importantes se posent de la même façon sur les communications. Si jamais une communication possède un timeout [1], il n'est pas possible de prévoir l'issue de ce délai. En effet, même si la transmission a commencé, cela ne garanti pas qu'elle se terminera dans les délais fixés, et en cas de dépassement la transmission renverrai une erreur. On ne pourra donc pas faire d'anticipation sur des transmissions possédant une limite dans le temps.

Le travail se découpera donc en deux parties.

- Dans un premier temps, il me faudra faire un prototype fonctionnel en faisant toutes les modifications au niveau du code existant pour intégrer l'exécution conditionnelle.
- Dans un second temps, une fois le prototype vérifié, il me faudra faire une liste de toutes les situations qui peuvent mener à une exécution anticipée et ensuite les implémenter.
- Enfin, faire une phase de tests afin de vérifier l'utilité de la compression.

Cependant, au vue de la contrainte sur le système distribué, il y a de fortes chances pour que le mécanisme ne puisse que rester une option et qu'il ne puisse être utilisé que si les contraintes sont respectées.

La situation servant à faire les tests sera une simple transmission entre deux processus via les appels send [2] et recv [3] qui effectueront les transmissions sans timeout.

1. Un délai d'expiration
2. Envoi de données.
3. Réception de données.

4 Conclusion

Mes travaux ont permis à l'équipe d'obtenir une meilleure vue sur les problématiques de la parallélisation du noyau de simulation. J'ai pu faire une analyse complète et poussée pour à la fois valider les choix de conception mais aussi pour trouver des idées nouvelles pour améliorer les performances du noyau. J'ai également pu mettre au point un algorithme pour la mise à jour du seuil. Enfin, j'ai fourni l'ensemble nécessaire à la rédaction d'un article.

Ce stage a été très intéressant dans le cadre de ma formation. Il m'a permis de me perfectionner dans plusieurs domaines mais aussi de découvrir de nombreux outils qui pourront m'aider lors de mes futurs travaux quelque soit mon domaine de travail.

Tout d'abord ce stage m'a permis de perfectionner mes connaissances en système, notamment lors du profilage du code ou j'ai pu réellement appréhender le fonctionnement d'un système d'exploitation dans une de ses parties les plus primordiales, l'ordonnanceur. J'ai aussi pu toucher de prêt aux problématiques de gestion de cache, ainsi qu'à des problèmes de synchronisation et d'accès concurrent.

Ce stage a été aussi pour moi l'occasion de découvrir deux outils :

- R[20] et Sweave[21]
- Org mode[22] et PlantUML[23].

Chacun de ces outils a sa propre utilité mais chacun d'eux m'a donné la possibilité de travailler avec une meilleure organisation, mais aussi de présenter le résultat de mes travaux de manière concrète.

R est à la fois un environnement et un langage d'analyse statistique. Il permet de manipuler des quantités de données importantes (les fichiers de trace sur lesquels j'ai travaillé pendant mon stage (section 2.2.2) faisaient plus d'un million de lignes) de manière robuste et rapide. La difficulté toutefois est de trouver de la documentation. À l'instar de ses homologues comme Matlab, il est fourni avec un ensemble de fonctions optimisées permettant le traitement rapide notamment sur les grosses matrices. Cependant, trouver ces fonctions exige de trouver des documentations qui sont très rares et souvent assez peu complètes. On se retrouve donc

souvent à utiliser des boucles et des structures conditionnelles (if...then...else) très lentes, alors qu'une fonction existe déjà.

Sweave lui est une fonctionnalité du langage R qui permet d'intégrer du code R dans un document LaTeX[24]. Celui-ci est ensuite converti par la commande Sweave dans R. Cette commande provoque l'exécution des blocs de code R puis intègre les sorties des instructions ainsi que les figures.

Ces deux éléments m'ont permis de rédiger des documents intermédiaires au cours de mon stage pour pouvoir expliquer mon travail, mais aussi de faire une analyse des données générées par les expérimentations menées.

Le second outil que ce stage m'a permis de découvrir est Org mode. C'est une extension pour Emacs qui permet de faire des documents structurés, mais qui permet aussi de faire des agendas, de prendre des notes et de les organiser, et de générer automatiquement des documents à partir des notes. Pour ma part, je m'en suis servi de journal, en stockant au fur et à mesure les observations de mon stage, mais aussi en y planifiant les travaux que je devais faire ce qui m'a permis de gagner du temps. PlantUML est un utilitaire pour la génération de diagramme UML (séquence, activité, classe, ...) qui est intégrable dans Org mode, cette outil m'a lui permis d'intégrer des diagrammes clairs dans les documents où je faisais mes analyses et m'a donc permis de mieux appréhender le mécanisme d'ordonnancement de SimGrid.

Ce stage m'a aussi permis de devenir plus autonome. Du fait que je devais effectuer un rapport hebdomadaire, mais aussi de l'absence de mon maître de stage pendant une longue période, j'ai dû organiser mon travail seul, afin de respecter les deadlines imposées. J'ai aussi eu l'occasion d'effectuer deux déplacements dont un tout seul, mais aussi de travailler avec des personnes d'origines différentes et de partager nos cultures et nos expériences.

Enfin ce stage me permettra de découvrir l'exercice que représente la rédaction d'un article scientifique. Voulant faire de la recherche mon métier et souhaitant poursuivre en thèse, cela sera une expérience très profitable pour moi.

Pour finir, il m'a permis de me confronter à de la recherche, alors que le stage de l'année précédente était plus proche du travail d'un ingénieur en recherche. Il m'a vraiment montré qu'elles étaient les particularités du métier de chercheur. Il m'a permis également, en étant présent pendant des périodes en dehors des vacances scolaires de découvrir en parlant avec les membres de l'équipe de certaines réalités du métier d'enseignant chercheur pour la partie enseignement.

A Les différentes fabriques de contexte de SimGrid

SimGrid possède trois façons de retenir et gérer les contextes. Chacune d'entre elle a ses avantages et ses inconvénients qui les rendent indipensables. Par défaut, le framework utilisera les contextes "raw", parce qu'ils sont les plus performants. On peut sélectionner la fabrique [1] qui fournira les contextes à utiliser via l'option de la ligne de commande :

```
--cfg=contexts/factory:[thread, ucontext, raw]
```

A.1 Fabrique thread

La première solution qui a été implémentée était l'utilisation de thread. Cette solution est la plus intuitive. En effet, la création d'un processus simulé ressemble à la création de thread. Lorsque l'on crée un thread, on crée une pile qui lui sera propre, en gardant un accès complet à l'espace mémoire de celui qui l'a créé, exactement comme les processus simulés dans SimGrid.

L'utilisation de *threads* pour le contexte sont multiples :

- toutes les fonctions pour interrompre ou relancer un *thread* existent déjà,
- tous les systèmes d'exploitation possèdent une interface pour thread,
- les outils de débogage et de profilage sont suffisamment développés pour permettre de les utiliser.

Malheurseument il y a aussi des inconvénients :

- il y a un nombre limité de *threads* qui peuvent être créés sur un système et même si cette limite est extensible on arrive rapidement à saturer le système d'exploitation,
- les mécanismes de synchronisation sont plus lourds que dans les deux autres fabriques.

1. Une fabrique est un patron de conception qui a pour but de fournir une abstraction pour la création d'entité (`http://fr.wikipedia.org/wiki/Fabrique_%28patron_de_conception%29`).

A.2 Fabrique ucontext

La fabrique ucontext repose sur le mécanimse de changement de contexte présent dans les environnements de type System V. Ceux-ci fournissent au développeur quatres fonctions qui permettent de créer un contexte, de le modifier, de le récupérer ou de l'échanger. Cette méthode permet plusieurs choses intéressantes :

- puisque le contexte est créé dans le même espace mémoire, on garde un accès aux variables globales,
- on n'a pas besoin d'avoir plus de *thread* que le nombre de *worker* demandé par l'utilisateur,
- il n'y a pas besoin de mécanisme de synchronisation, puisque si le contexte est changé une nouvelle fois, le processus est de ce fait stoppé.

Il reste cependant des inconvénients majeurs à cette fabrique. Dans un premier temps, celle-ci n'est pas portable, puisqu'on utilise une particularité des environnements System V. De plus, bien que plus rapide que la synchronisation entre les threads, le changement de contexte reste quand même long.

A.3 Fabrique raw

Cette fabrique est la plus performante des trois. Elle part d'une constatation simple. En fait ce qui caractérise un processus est le trio (espace mémoire, pile, registre). Ce que l'on souhaite, c'est avoir plusieurs processus utilisant le même espace mémoire mais possédant leur propre pile ainsi que des registres cohérents avec leur état d'exécution. La création d'état se résume donc à créer une pile et des registres cohérents avec celle-ci. Puis, au moment de faire le changement de contexte, on n'a plus qu'à sauvegarder les registres de l'ancien processus (qui contiennent l'adresse mémoire de la pile de celui-ci) puis à mettre à la place de ceux-ci les registres du nouveau processus qui contiendront eux l'adresse de la nouvelle pile. On effectue ainsi un changement complet du couple (pile, registre). Cette méthode cumule les avantages de la fabrique ucontext mais permet une bien meilleure performance car le nombre d'instruction est réduit à son stricte minimum.

Cependant, à l'instar de la fabrique ucontext, elle possède un problème de portabilité. Mais ici, il n'est plus au niveau du système d'exploitation mais au niveau du jeu d'instruction du processeur car les fonctions de création de contexte mais aussi de changement de contexte sont écrites en assembleur et sont donc clairement dépendantes de la machine hôte. Toutefois, les fonctions ont été implémentées pour les jeux x86_64 et i386 qui représentent l'immense

majorité des architectures vendues, et l'extension à de nouvelles plates-formes est relativement aisée puisqu'on utilise uniquement des fonctions très basiques dans ce code (empilement, dépilement principalement).

Un autre inconvénient, plus gênant celui-là est que les outils de débogage sont complètement perdus au moment des changements de contexte. Si une erreur se produit dans le changement de contexte, celui n'arrive plus à remonter les fonctions et donc à indiquer où se situe l'erreur de manière précise.

B Diagramme d'activité d'un processus

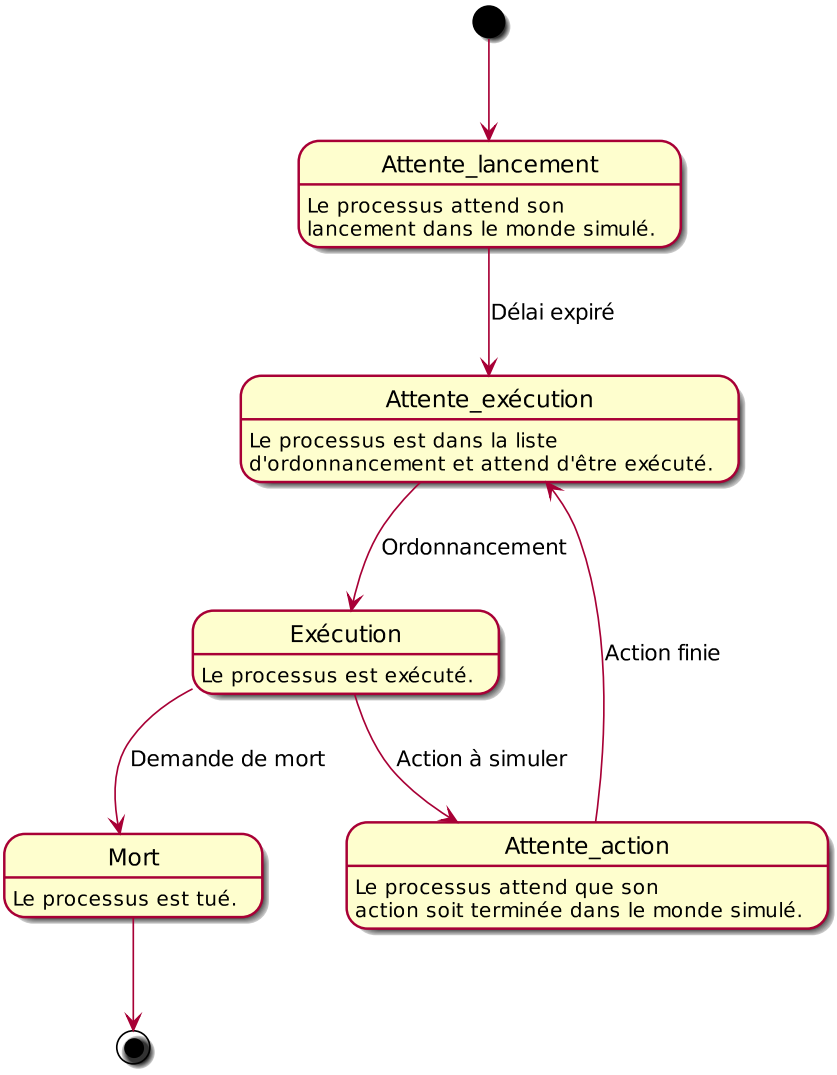


FIGURE B.1 – Diagramme d'état classique.

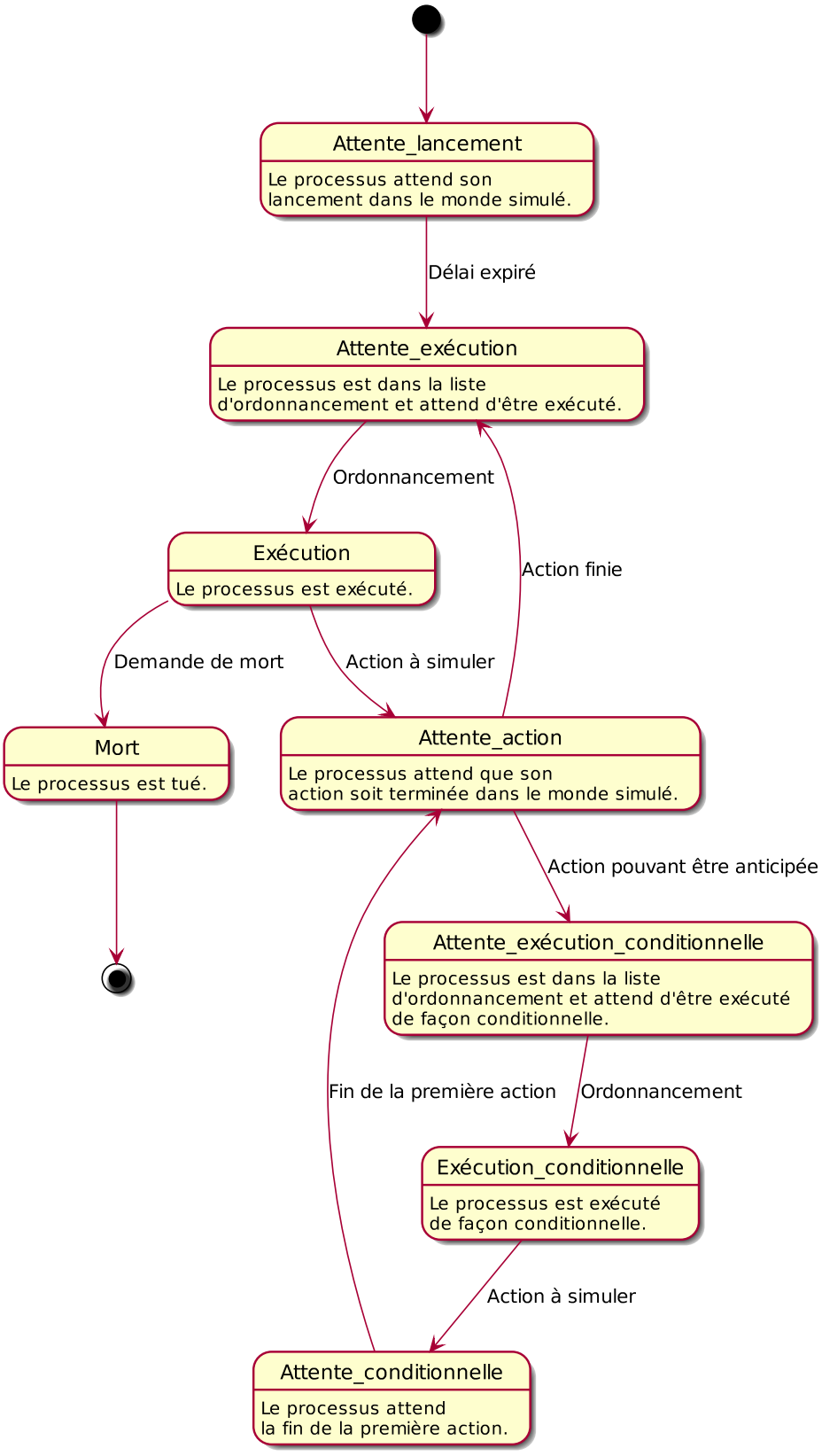


FIGURE B.2 – Diagramme d'état avec compression..

C Exemple d'exécution avec le principe de compression

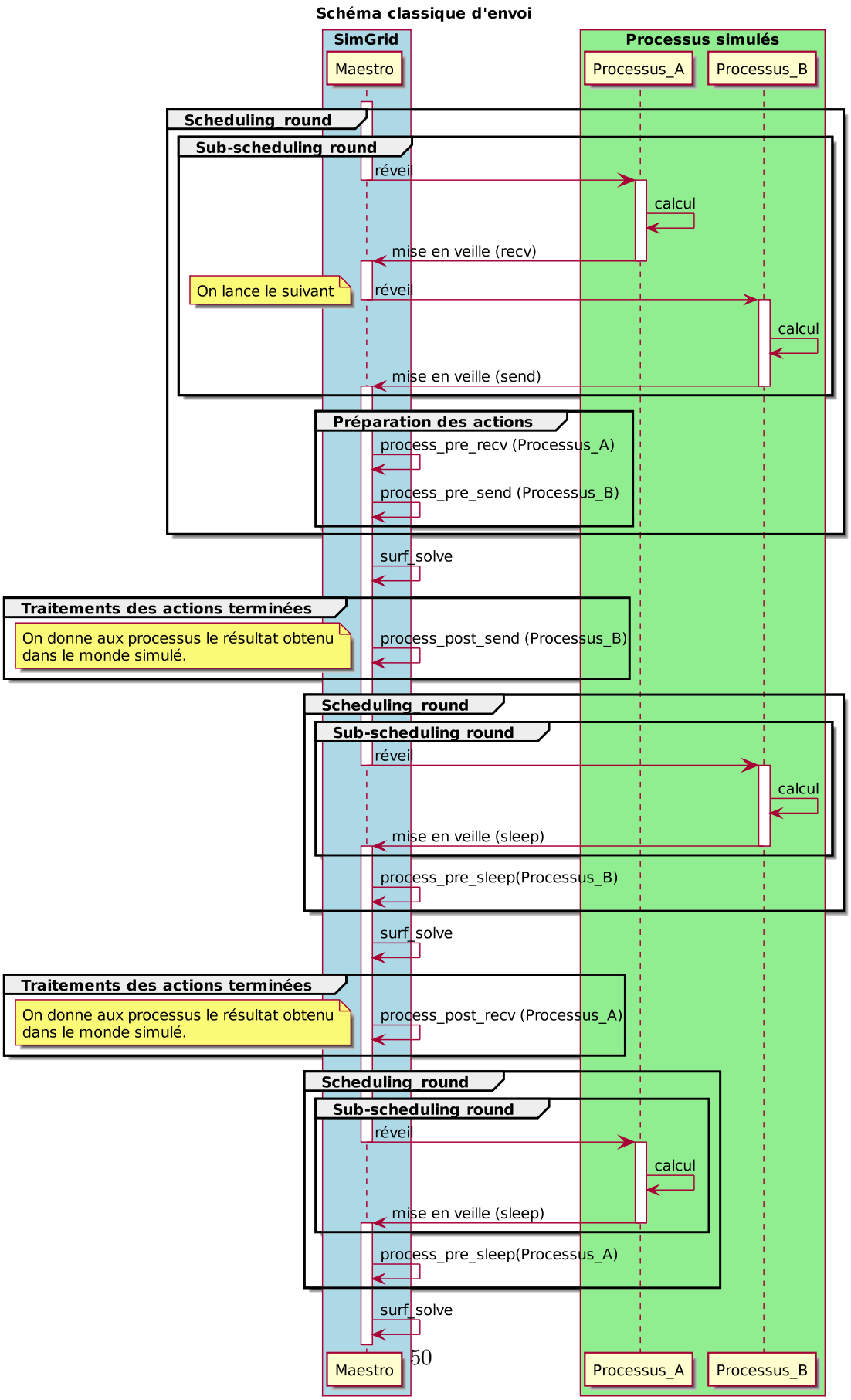


FIGURE C.1 – Exemple d'exécution d'une transmission de données de B vers A.

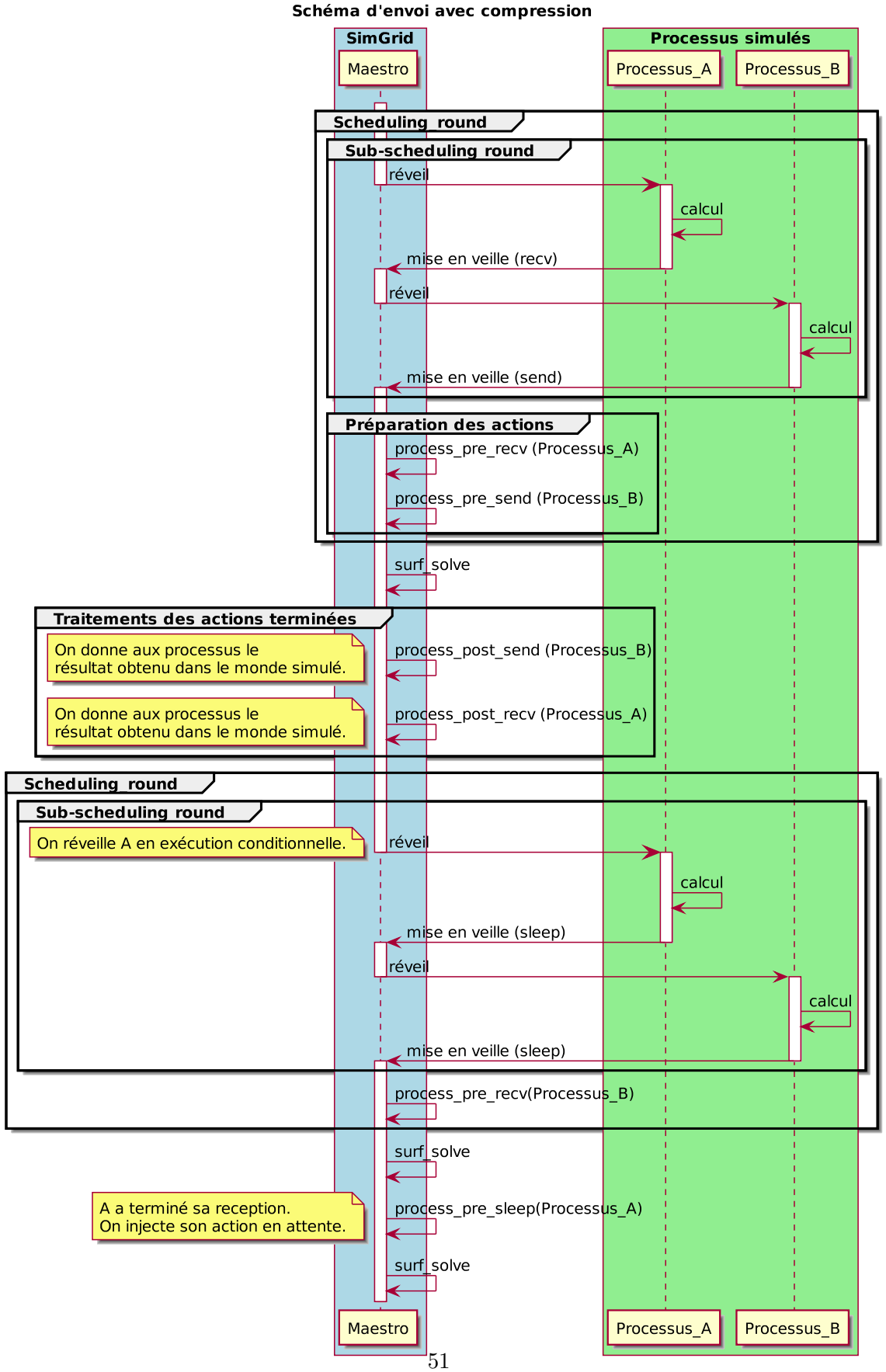


FIGURE C.2 – Exemple d'exécution d'une transmission de données de B vers A avec compression.

Bibliographie

[1] AlGorille. http://algorille.loria.fr/index.html.

[2] Loria (laboratoire lorrain de recherche en informatique et ses applications). http://www.loria.fr/les-actus.

[3] Kadeploy3, efficient and scalable operating system provisioning. http://kadeploy3.gforge.inria.fr/.

[4] Distem, DISTributed systems EMulator. http://distem.gforge.inria.fr.

[5] Grid'5000, efficient and scalable operating system provisioning. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

[6] Henri Casanova. Simgrid : A toolkit for the simulation of application scheduling. *Cluster Computing and the Grid, IEEE International Symposium on*, 0 :430, 2001.

[7] eMule. http://www.emule-project.net/home/perl/general.cgi?l=13.

[8] BitTorrent. http://www.bittorrent.org/beps/bep_0003.html.

[9] BOINC. http://boinc.berkeley.edu/.

[10] LHC@home. http://lhcathome.web.cern.ch/.

[11] SETI@home. http://setiathome.berkeley.edu/.

[12] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[13] Martin Quinson, Cristian Rosa, and Christophe Thiery. Parallel Simulation of Peer-to-Peer Systems. In *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 668–675, Ottawa, Canada, May 2012. IEEE.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[15] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, nov 2001.

[16] Infiniband. http://www.infinibandta.org/.

[17] Valgrind. http://valgrind.org/.

[18] VTune. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[19] Peersim : a peer-to-peer simulator. http://peersim.sourceforge.net/.

[20] The R project for statistical computing. http://www.r-project.org/.

[21] Sweave. http://www.stat.uni-muenchen.de/~leisch/Sweave/.

[22] Org mode. http://orgmode.org/.

[23] Plantuml, opensource tool in Java to draw UML diagram. http://plantuml.sourceforge.net/.

[24] Latex – a document preparation system. http://www.latex-project.org/.