Mise en place de tests de non régression de performance

Florian Popek

13 août 2016

RICM 4

Mise en place de tests de non régression de performance

Florian Popek

13 août 2016

Stage réalisé au Laboratoire d'Informatique de Grenoble

Sous la supervision d'Arnaud Legrand (équipe POLARIS)

Août 2016

Résumé

De tout temps la communauté scientifique n'a cessé de croître grâce aux échanges, permettant aux scientifiques de baser leurs travaux sur ceux préalablement établis par d'autres. Ces échanges permettent également de crédibiliser les résultats trouvés en décrivant les protocoles expérimentaux, afin que quiconque puisse de son côté les vérifier. L'informatique ne déroge pas à ce principe, néanmoins, le nombre de paramètres à prendre en compte et les différences entre machines rendent ce processus de partage épineux, autant pour celui qui souhaite décrire précisément son expérience que pour celui qui souhaite la reproduire.

Dans ce document, nous abordons le problème de la reproductibilité des expériences, en particulier dans le domaine HPC (High Performance Computing), selon deux aspects qui sont : le contrôle de l'environnement et le pilotage de l'expérience, en plus de quelques considérations autour de la reproductibilité en général. On y présentera notre solution basée sur des technologies développées à l'Inria permettant de déployer un environnement afin d'y conduire des expériences. L'objectif final est d'évaluer les performances (*benchmarker*) de StarPU tout au long de son évolution afin d'y déceler de potentielles régressions entre deux versions, tout en ayant un contrôle parfait de l'environnement et de l'expérience.

Remerciements

Je remercie tout particulièrement Arnaud Legrand, Michael Mercier, Vinícius Garcia Pinto, et Luka Stanisic pour leur aide et expérience dans l'aboutissement de ce projet.

Table des matières

— 1 —
Introduction

L'informatique est devenue le 3ème pilier de la science, indispensable aux sciences modernes permettant à la fois des études en simulation (*in silico*) et la fouille de donnée massive (*big data*). Pour répondre aux besoins de calcul sans cesse croissant tout en respectant les contraintes technologiques (finesse de gravure des composants) et physiques (vitesse de la lumière, dissipation de chaleur et besoin énergétique), l'informatique s'est orientée vers des architectures de plus en plus complexes, massivement multi-coeurs, embarquant parfois dans un même noeud de calcul des processeurs de technologies différentes (ARM, x86, ...) ainsi que des accélérateurs (GPUs, Xeon Phi, ...). L'exploitation de telles architectures nécessite de revisiter complétement la façon de programmer. L'approche actuelle la plus efficace consiste à utiliser un *runtime* utilisant le paradigme des tâches. Au lieu d'expliciter séquentiellement comment le calcul doit être fait, on décrit le calcul par un graphe de tâches et c'est le runtime qui décidera dynamiquement sur quelles ressources de calcul placer telle ou telle tâche et quand transférer les données entre ces ressources. L'ordonnancement optimal d'un tel graphe de tâche étant un problème difficile, ces runtimes utilisent des heuristiques (de type ordonnancement de liste ou vol de travail) plus ou moins évoluées. S'assurer que de tels runtimes soient efficaces sur une grande variété de plateformes et de graphes est donc extrêmement difficile. Il est habituel de mettre en place des tests de non régression afin de s'assurer qu'on n'introduit pas de bug lors du cycle de vie du runtime. Dans un contexte de calcul où les performances sont très importantes, il serait essentiel d'avoir des tests de non régression de performance afin de s'assurer que telle ou telle amélioration du code sur telle plateforme ne conduit pas en réalité à une régression des performances sur d'autres plateformes. Ce problème de non-régression de performance a été quelque peu abordé dans le contexte du noyau Linux ou dans la communauté python mais quasiment pas dans le contexte du calcul hautes performances.

Dans mon stage, je me suis donc intéressé à la mise en place d'une infrastructure de test de non régression de performance légère et adaptée au runtime High Performance Computing (HPC) StarPU, dont l'objectif est l'exploitation efficace d'architectures hybrides.

Ce rapport est organisé de la façon suivante : dans la section intitulée *Contexte* on y expliquera les enjeux et problématiques du calcul hautes performances. Dans *État de l'art*, nous présenterons des outils ensentiellement conçus à l'Inria en abordant la repoductibilité des recherches. La partie *Méthodologie* concerne la façon dont j'ai travailler tout au long de ce stage, et la partie *Ma contribution* parlera du fonctionnement de cette infrastructure de tests de non régression de performance. Enfin la dernière section intitulée *Conclusion* traitera des connaissances qui ont été acquises à mon sens, ainsi que les évolutions possibles de ce projet, suivi d'une conclusion plus générale.

— 2 —
Contexte

2.1 Le LIG

Le Laboratoire Informatique de Grenoble (LIG) rassemble plus de 500 chercheurs, enseignants-chercheurs, doctorants et autres personnels accompagnant la recherche, répartis sur deux sites : Grenoble et Montbonnot. En partenariat académique avec le CNRS, l'UGA, Inria Grenoble Rhône-Alpes et Grenoble INP, ses 24 équipes contribuent au développement de l'informatique fondamentale (modèles, méthodes, langages, algorithmes). C'est un laboratoire aux activités diverses qui oeuvre pour développer une synergie entre les défis technologiques, conceptuels et sociétaux entourant cette discipline.

J'ai intégré l'équipe POLARIS, qui s'intéresse à la compréhension et modélisation mais aussi l'analyse des performances des systèmes distribués à très grande échelle. Pour modéliser de tels systèmes (des centaines de machines, réparties géographiquement, reliées entre elles par des réseaux à hautes performances), ils ont conçus en partenariat avec les équipes de recherche d'Inria Bordeaux, un certains nombre de simulateurs (SimGrid) et d'outils permettant de faciliter le travail d'expérimentation selon différents aspects tels que le déploiement d'environnement sur les machines distantes ou la conduite d'expérience.

Par exemple pour la gestion d'environnement, ils ont conçu Kameleon : un outil d'exécution de scripts et de commandes Shell basé sur une syntaxe YAML, capable de déployer et de construire une image (Debian par exemple), d'y installer un certain nombre de paquets/logiciels pour ensuite sauvegarder l'image sous forme d'une archive pouvant être plus tard redéployée à volonté afin d'y conduire des expériences.

Pour la conduite d'expériences, des moteurs d'expériences tels que Expo ont pour but de simplifier ce travail en offrant des abstractions telles que des tâches, des listes de tâches et des ressources, ainsi que tout un tas de fonctionnalités intéressantes (capture constante de logs, modèle client-serveur, ...)

Développement durable

Mon stage s'est déroulé dans le nouveau bâtiment de l'IMAG, situé sur le campus universitaire de Grenoble. Ce bâtiment centralise les équipes situées aux alentours de Grenoble (l'année dernière, j'avais effectué un stage à Montbonnot avec le même tuteur) au sein du même emplacement.

Ce bâtiment, tout récent, possède certaine particularité s'inscrivant dans le développement durable mais aussi au confort de ses occupants.

Ce bâtiment est *intelligent* : par exemple, les volets électriques des fenêtres s'ajustent automatiquement pour obtenir une luminosité adéquate dans les différents bureaux. Le bâtiment est aussi équipé de l'air conditionné. Chaque étage (au nombre de 4) est composé de deux salles à manger équipées (machine à café, frigo, couverts, ...) permettant aux gens d'y prendre leur repas, et contient également un certain nombre de douches disséminées à différents endroits.

Salle de réunion, salle de rencontres informelles, salles de détente, le confort des occupants est une préoccupation motivée par tous : j'ai pu assister à une réunion rassemblant l'équipe DATAMOVE et mon équipe POLARIS au sujet de l'état actuel de notre partie d'étage et de l'organisation. Cette réunion prônait l'ouverture et le partage entre équipe (mise en place d'un trombinoscope, liste/groupe de mails pour recevoir les dernières informations, BBQs à venir, ...) et faisait aussi mention des améliorations envisageable (mise en place de tableaux d'affichage dans les couloirs, portes coupe-feu moins dures à ouvrir à l'intention des personnes à mobilité réduite, suggestions d'exploitation de certaines salles encore vides, ...).

Le bâtiment s'inscrit également dans une démarche environnementale forte : le datacenter est refroidi par le *freecooling* : un système innovant de récupération de chaleur sur le datacenter utilisant la géothermie, et la récupération de l'eau de pluie (nappe phréatique sous le bâtiment) pour les sanitaires. L'isolation est aussi renforcée, permettant de réduire les besoins en chauffage et rafraîchissement.

2.2 Le Calcul Hautes Performances

Le domaine du HPC s'intéresse aux architectures multi-coeurs et muli-GPUs et à l'ordonnancement des tâches afin d'approcher les performances théoriques offertes par ce genre d'architecture, traitant des centaines de noeuds sur différentes machines connectées entre elles. De telles architectures sont difficiles à programmer (concevoir une application mono-threadée avec de bonnes performances est déjà un problème) : la gestion des threads et des transferts de données entre les différentes unités de calculs n'est pas un problème simple même avec des technologies censées simplifier ce travail comme CUDA (Compute Unified Device Architechture).

Aujourd'hui, ce sont les processeurs graphiques (GPU) qui sont les plus aptes à effectuer des calculs en masses, couplées aux processeurs (CPU) communiquant avec ces premières. La différence d'architecture et de fréquence de ces deux types d'unité de calcul (˜1 GHz pour les GPU et ˜3 GHz) nécessite un niveau d'abstraction supplémentaire afin de pouvoir les utiliser efficacement. Cette programmation de plus haut niveau (*DAG*) s'appuie sur l'exécution dynamique opportuniste dont le principe est de distribuer la charge de travail aux différentes unités de calcul, ce qu'offre StarPU mais aussi d'autres (Quark, Parsec). Pour des raisons historiques, les applications actuelles sont codées directement en threads/OpenMP et CUDA et comportent donc une façon de programmer complexe et sujette à l'erreur pour l'expérimentateur.

Les technologies évoluent si vite qu'obtenir un code avec de bonnes performances pour les différents types d'architecture et quasiment impossible sans y consacrer énormément de temps. Ajourd'hui un bon nombre d'application est en train de passer à un modèle de programmation à plus haut niveau.

2.3 L'écosysteme StarPU

Pour gérer cette distribution dynamique des tâches (runtime), les laboratoires de Bordeaux ont conçus StarPU : un support exécutif original qui fournit un modèle d'exécution unifié afin d'exploiter l'intégralité de la puissance de calcul tout en s'affranchissant des difficultés liées a la gestion des données, et offre par ailleurs la possibilité de concevoir facilement des stratégies d'ordonnancement portables et efficaces.

StarPU permet aux développeurs de décomposer le travail en tâches qui pourront être dynamiquement distribuées afin d'optimiser la charge de travail sur les unités de calcul selon l'architecture, tout en optimisant également le transfert de données entre la mémoire principale (RAM) et les autres mémoires discretes (cache, mémoire GPU, ...).

StarPU base ses modèles de calculs sur les BLAS (Basic Linear Algebra Subprograms) et MKL (Math Kernel Library) qui sont des librairies mathématiques, pour le calcul - à haute performances - de matrices par exemple, ou la résolution de systèmes mathématiques.

Enfin, StarPU utilise MORSE (Matrices Over Runtime Systems) dont le but est d'exploiter au maximum les ressources disponibles afin d'aboutir le plus rapidement possible à une solution, et Chameleon qui est un sous projet de MORSE specifiquement dédié à l'algèbre linéaire dense. Par exemple, Chameleon est capable de résoudre une factorisation de Cholesky à un débit de 80 TFlop/s d'une matrice dense d'ordre 400000 en 4 minutes.

2.4 Plateformes expérimentales

Plafrim est une plateforme située à Bordeaux dédiée à la recherche, aux modélisations et simulations, et à l'expérimentation des mathématiques appliquées. Il s'agit d'une plateforme classique telle que l'on peut retrouver typiquement en HPC, et constitue une plateforme de production, c'est à dire une plateforme dédiée pour le calcul spécialisé.

Grid5000 est une plateforme nationale dédiée à l'expérimentation mais n'est pas une plateforme de production. C'est un projet lancé en 2003 dont le but était de mettre en place une grille informatique expérimentale répartie sur 10 sites en France. Aujourd'hui Grid5000 est constituée de milliers de CPU et de GPU, mis à disposition des chercheurs informatiques. C'est une plateforme plus homogène (de type grille ou cloud) mais dans laquelle se trouve quand même quelques noeuds un peu exotiques.

C'est sur cette plateforme que j'ai travaillé : j'y déployais des environnements Debian afin d'y tester StarPU. L'avantage de Grid5000 est que l'utilisateur est *root* : il peut faire ce qu'il veut de la machine pendant son temps de réservation et y déployer son propre environnement.

2.5 La conduite d'expérience

Conduire une expérience n'est pas chose aisée : contrôler chacun des paramètres présents dans une machine est pratiquement impossible tant ils sont nombreux et tant le comportement d'une machine moderne actuelle est imprédictible. Cet indéterminisme est dû à beaucoup de facteurs qui peuvent jouer sur les performances d'un code :

- L'architecture même de la machine : non homogénéité du cluster et de ses composants (processeurs de constructeurs différents avec des caractéristiques différentes)

- Compilateur (GCC, MSV, Intel, . . . , ainsi que leur version)
- Bilbliothèques utilisées (versions de CUDA, mais aussi des BLAS, de la MKL, versions des paquets installés, . . .)
- Système d'exploitation (sa version, son scheduleur, ses paramètres actuels), gouverneur DVFS

Ainsi, sur des plateformes comme Plafrim, les expérimentateurs et leurs expériences sont tributaires des mises à jour de l'administrateur, en bien ou en mal, qui s'arrange pour faire le mieux pour l'ensemble de la communauté.

2.6 Tests de non régression de performance

Lors de la conception de logiciel tel que StarPU, il est fréquent d'instaurer des tests de non régression de performance, afin de s'assurer que l'introduction d'une nouvelle fonctionnalité ou la modification du code existant n'implique pas une baisse de performance.

Pour StarPU, chaque nuit des tests automatiques ont lieu afin de tester la révision courante, sur Plafrim. Ces tests consistent à exécuter une transformation de Cholesky sur une matrice de taille donnée, pour ensuite comparer les résultats du tests avec ceux des versions antérieures.

Ces tests de non régression de performance sont le plus souvent décentralisés, confiés à une machine qui se chargera de lancer l'expérience et d'en afficher les résultats sur une page Web consultable par ceux intéressés.

— 3 —
État de l'art

3.1 Non régression de performance

Aujourd'hui, un certain nombre de frameworks écrits dans différents langages permettent la mise en place de ces tests plus ou moins facilement. Pratiquement tous ont opté pour le modèle client-serveur, où les résultats sont rendus *public* et sont décentralisés pour des raisons de praticité et d'accessibilité. Tous ces résultats sont ensuite affichés sous forme de courbes, de graphes avec barres, . . . , en offrant des moyens simples et visuels de comparer et classifier les résultats afin de rendre cet amas de données visible.

On retrouve par exemple, Codespeed écrit en python, Chromium : Bisecting Performance Regressions, Phoronix, Jenkins-Perf Module ou bien CollectiveMind.

Dans le contexte du développement d'un logiciel et plus particulièrement dans le contexte de la recherche, certains de ces outils sont plus adaptés que d'autres : ceux qui nécessitent de modifier même légèrement le code pour se plier aux règles du framework seront à éviter (on pensera à Phoronix qui oblige à ne retourner que trois valeurs par expérience : le résultat final (la moyenne), le minimum et le maximum (Phoronix possède des fonctionnalités utiles dans d'autres contexte, mais est moins adapté au notre et aux travaux préalablement établis concernant les tests de StarPU)).

CollectiveMind est orienté recherche et dévelloppement reproductible et s'abstrait de tout langage déjà existant en proposant une syntaxe qui lui est propre, permettant ainsi à quiconque de l'utiliser, sans avoir à disposer d'une quelconque expérience dans un langage (très souvent orienté objet).

Il est difficile de se prononcer sur qui est le meilleur, tant l'attente du résultat final peut être différente selon le contexte (un industriel aura sûrement moins d'éxigences qu'un chercheur quant à la visualisation des données, puisque celle ci jouera un rôle moins essentiel dans son activité).

3.2 Recherche reproductible

La reproductibilité est une préoccupation récente motivée en particulier par les chercheurs en informatique traduisant une volonté de transparence et de clarté des expériences et des résultats (graphes clairs, titré, avec une bonne échelle, présence des unités, problèmes rencontrés, . . .).

Elle est issue d'un phénomène courant : lorsqu'un article paraît présentant des résultats, récupérer le code de l'expérience auprès de leurs auteurs est rarement possible pour des raisons

variées :

- L'auteur n'a pas l'intention de fournir le code
- Le code n'est pas disponible pour des raisons commerciales
- Le code est la propriété d'un organisme ou d'une université
- La personne en charge de ce code ne fait plus parti de l'organisme
- Problème de version : le code récupéré n'est pas exactement le même (il a évolué depuis)
- Mauvaise pratique de back-up : le code est perdu / n'est plus accessible
- Le code sera bientôt disponible (il est en phase de *nettoyage*, et très souvent ne sera pas délivré)
- Le code n'est pas partageable : trop d'investissement serait nécessaire pour qu'il le soit

D'autres phénomènes peuvent également biaiser les résultats d'une expérience la rendant non-reproductible :

- Un article devra quasi-systématiquement afficher des résultats positifs (esprit de compétitivité)
- Biais de l'expérimentateur : celui-ci se débrouillera pour concurrencer les résultats d'autres chercheurs puisqu'il fera tout pour (ces derniers pourraient en faire autant) : problème de contexte d'exécution et de point de vue de l'expérimentateur
- L'erreur humaine : problème de manipulation de données, ou erreurs de programmation qui au final affichent de bons résultats
- Pas de volonté de rendre l'expérience partageable de toute façon : manque d'outils, le matériel et le logiciel changent constamment (ces excuses ne sont plus valables aujourd'hui)
- Voire même problème de fraude

Tous ces problèmes ont poussé les chercheurs à entreprendre des recherches reproductibles, dont la définition ne se limite pas à pouvoir ré-exécuter du code. Il s'agit de pouvoir répliquer à l'identique ce qu'un tiers a pu faire (réplicabilité) afin de recréer l'esprit des précédents travaux par soi-même pour en arriver à des conclusions identiques (reproductibilité).

Plusieurs points clés définissent le processus de recherche reproductible :

- Noter son activité
- Noter et enregistrer ses résultats
- Organiser ses résultats
- Contrôler son environnement
- Contrôler son expérience

- Tout le monde peut refaire l'expérience

Ainsi, l'objectif principal des recherches reproductibles est de noter absolument tout ce que l'expérimentateur entreprend et collecte, tant de bons que de mauvais résultats, mais aussi les motivations de ce qu'il fait. De nombreux outils permettent de faciliter ce travail de prise de note (Org-mode par exemple, que nous présenterons), et d'autres outils permettent de mieux définir une expérience en terme de workflow, mais aussi en terme d'avancement / de back-up avec les logiciels de gestion de versions tels que Git par exemple.

D'autres méthodes commencent à émerger telles que la programmation littérale (qui consiste mixer du code traditionnel et des descriptions en langage naturel du code et des enjeux (ex : RMarkdown)), ou bien les containers (Docker) permettant de virtualiser un environnement tout en profitant directement des ressources fournies par le système d'exploitation.

3.3 Contrôle d'environnement

Le controle d'environnement est un élément clé pour la reproductibilité : il consiste à pouvoir décrire précisément l'état d'une machine (tous les paquets/logiciels installés, leur version, la configuration matérielle de la machine, ... (la liste est longue)), mais aussi de pouvoir revenir à un tel état. Peut-on installer un programme/paquet à partir d'une version précise sans tomber dans un enfer de dépendances ?

Une solution brutale, mais efficace, consisterait à enregistrer l'environnement dans sa globalité pour obtenir une image (un .tgz par exemple) que l'on pourrait redéployer : on obtiendrait un environnement identique et, auquel cas, des expériences identiques.

Grid5000 est basé sur ce principe de déploiement d'image : à l'aide `tgz-g5k`, il est possible d'enregistrer son environnement pour ensuite le redéployer avec `Kadeploy`, un outil développé à l'Inria.

Cette solution s'appuie sur le principe de bonnes pratiques, c'est à l'expérimentateur de s'assurer qu'une image de son expérience est disponible. Pour des raisons de stockage (une image peut peser plusieurs GB), cette solution peut ne pas être systématiquement appliquée, en particulier lorsqu'un logiciel à benchmarker possède des milliers de révisions.

Plutôt que d'enregistrer chacun des environnements, des solutions telles que Kameleon permettent de les construire à volonté selon des *recettes* : une image Debian pourra être construite pour ensuite y installer un certain nombre de paquets ou de logiciels, et constituer l'environnement d'une expérience. Kameleon n'est pas juste un outil qui exécutera successivement des commandes Shell. Son gros avantage (outre sa simplicité exemplaire) est la possibilité de créer des recettes basées sur d'autres recettes, à la manière d'héritage tel qu'il est proposé par les langages orientés objets. Une recette Kameleon pourra ainsi reprendre une recette construisant une image Debian basique sans se soucier de ce qu'elle contient, mais aussi bénéficier de blocs de code déjà prêt permettant des interactions avec Grid5000.

Pour ce qui concerne les dépendances dynamiques (celles qui changeront au fur et à mesure de l'expérience), l'outil Spack résoud ce problème en abstrayant toutes ces dépendances et leur installation à l'utilisateur. Ce dernier pourra ainsi installer StarPU avec tel ou tel BLAS, pour une version donnée, et/ou une version de compilateur, etc... sans se soucier du téléchargement des paquets nécessaires et de leur installation.

Ces deux outils, à eux seuls, permettent de largement simplifier le travail de l'expérimentateur en ce qui concerne le gestion de son environnement et illustrent le besoin

croissant des enjeux de la reproductibilité : des outils simples permettant de définir précisément et clairement les dépendances d'une machine. Bien que ce ne soient pas des solutions miracles (installer StarPU avec Spack nécessite de fournir une fois pour toute les règles de compilation de ce premier), une recette Kameleon utilisant Spack en interne pourra être partagée sans problème et réutilisée à volonté.

3.4 Moteur d'expérimentation

Les moteurs d'expérimentation permettent de décrire avec un niveau d'abstraction le déroulement d'une expérience, participant ainsi à sa lisibilité et à son partage.

On retrouve par exemple XPFlow écrit en Ruby avec une syntaxe très haut niveau, permettant de décrire les tâches qui doivent être exécutées séquentiellement ou bien parallèlement, et offre par ailleurs tout un tas de blocs d'instructions permettant de clarifier l'écriture en guise de sucre syntaxique (instructions *switch* sous plusieurs vairantes, avec des tests à l'intérieur, avec prise en charge du parallélisme ou non, . . .).

Execo, écrit en Python, offre tout un tas de classes permettant entre autre de déployer des tâches via SSH tout en récupérant les sorties standards et sorties d'erreurs simplement.

Expo, en Ruby, propose des mécaniques similaires à XPFlow basées sur l'abstraction des tâches et des ressources et est construit en deux parties : une partie client et une autre serveur.

Parmis ces moteurs d'expérimentation, on retiendra Execo qui est plus abouti mais plus bas niveau. C'est celui-ci que l'on utilisera, même si l'utilité que l'on en fait est moindre comparée aux possiblités offertes.

— 4 —
Méthodologie

4.1 Journal

Dès mon arrivée, mon tuteur m'a conseillé d'utiliser un journal pour noter mon avancement à titre personnel, dans un fichier Org. Lui-même en utilise un dans son travail de chercheur, quotidiennement.

L'intêret est de noter chronologiquement tous les problèmes rencontrés, les solutions tentées, si elles ont échouées ou au contraire réussies, y résumer les points essentiels des réunions, les objectifs à entreprendre, les commandes pour installer telle ou telle chose, ...

Pour tenir mon journal, j'ai utilisé Org-mode sous Emacs qui une fois bien configuré, possède un - gros - tas de raccourcis permettant d'étiquetter le journal, faire référence à des fichiers en local, insérer des suites de commandes pouvant être directement lancées depuis emacs, insérer des sorties standards, et bien d'autres. L'organisation hierarchique est également très pratique, les informations sont triées par titre, sous-titre (avec autant de niveaux que l'on souhaite) et peuvent être dépliées / repliées pour ne laisser paraître que ce dont on a besoin, tout en ayant l'intégralité du journal à disposition.

Tenir un journal n'est pas un travail supplémentaire, en ce sens ce n'est absolument pas fastidieux, au contraire. Même si une journée peut consister à inlassablement tenter de résoudre un problème en vain, le journal grossira et prodiguera une sensation de travail accompli même si rien n'a été produit. Pour moi qui ne suis absolument pas organisé, cet outil a su me séduire par sa praticité tant sur la prise en main que sur les résultats apportés par un journal.

Mon journal est disponible en ligne, accessible depuis la section *Bibliographie*.

4.2 Gestion du projet

Ce projet s'articule linéairement selon plusieurs parties indépendantes : le déploiement, l'installation de StarPU, son exécution, et la visualisation des données.

Mon travail s'est d'abord focalisé sur l'usage de Spack, en même temps que l'aprentissage de Grid5000, pour ensuite m'interresser à Kameleon, puis à l'execution de StarPU. La visualisation des données s'est quant à elle faite, en premier lieu, à partir de données quelconque (dans un fichier CSV).

Toutes ces parties ont grandies et ont été paufinées au fur et à mesure du stage, cadencées par les différentes personnes compétentes sur chaqu'une de ces technologies, selon leur présence

au sein du laboratoire et les délais de réponse des mails. Le projet a évolué selon ce qui été prévu, les délais réels et estimés par moi-même étaient très proches, bien qu'en fin de stage quelques problématiques (problème de compilation de StarPU, Grid5000 down, ...) alliées au départ en vacances de la majorité du personnel m'ont mis des bâtons dans les roues.

Travailler sur Grid5000 demande une certaine patience : déployer un environnement prend quelques minutes, l'installation de StarPU en prend une dizaine, tout comme l'exécution de chacun de ses tests. Concevoir et tester les différents scripts utilisés nécessite donc de privilégier au maximum les tests sur la machine locale, en créant parfois des repertoires et sous répertoires similaires à ceux qui seront crées sur Grid5000 afin de simuler l'environnement.

Toutefois, j'ai réservé des centaines de machines sur Grid5000 et une telle attente nécessite d'organiser son travail (remplir son journal pendant le déploiement par exemple) et il m'est arrivé trop souvent de subir cette attente.

— 5 —
Ma contribution

5.1 Préliminaires

Avant d'arriver à la solution finale de mon travail, les premières semaines furent consacrées à l'appopriation du sujet et consistaient en une recherche documentaire sur la reproductibilité principalement.

Mon tuteur a tenu ces derniers mois des conférences intitulées Webinar abordant des thèmes variés en lien avec la recherche informatique comme le contrôle d'environnement, la prise de note des expériences et des résultats, la reproductibilité numérique ou bien la production d'articles dont le contenu peut être reproduit (d'autres sujets sont également prévus). Je me suis ainsi familiarisé avec les problématiques, le contexte général, les outils et pratiques qui pouvaient être utilisées, etc...

Mes recherches se sont ensuite tournées vers des outils déjà existant pour répondre à notre objectif, ceux cités plus haut (Phoronix, Codespeed, ...), et après avoir fait le tour de ces outils, la solution la plus convainquante fut de créer un outil maison basé sur Spack et Kameleon, mais aussi Execo. C'est ici que mon travail principal débute.

5.2 Déployer, piloter, rappatrier avec Kameleon

Une fois l'obtention d'un compte Grid5000 auprès d'un administrateur, il suffit de se connecter à l'un des 10 sites (*frontend* ou *frontale* en français) via SSH pour ensuite réserver un certain nombre de machines à l'aide de la commande `oarsub` pour ensuite déployer une image sur celles-ci avec Kadeploy.

De base, plusieurs images Debian (parmis d'autres) sont disponibles depuis la frontale possédant plus ou moins d'outils préinstallés. Le dossier utilisateur de la frontale sert de dépôt qui persistera dans le temps, ce qui n'est pas le cas des machines réservées : celles-ci serviront à d'autres plus tard et seront reformarttées au grès de leurs utilisateurs. On se servira alors de ce dépôt pour y stocker nos images afin de les redéployer à volonté.

Le déploiement peut aussi être fait depuis Kameleon qui contient de base des recettes permettant le déploiement sur Grid5000 : nous utiliserons ces recettes en interne aux nôtres afin d'exploiter cette fonctionnalité.

Par ailleurs, Kameleon permet de gérer simplement trois contextes d'exécution qui sont : la machine locale (celle où l'on lance Kameleon, contexte *local*), la frontale (contexte *out*), et les machines réservées (contexte *in*). Une recette Kameleon constitue donc un bon moyen de

piloter une expérience en spécifiant les commandes à utiliser sur tel ou tel contexte (voir annexe intitulée *Principe d'une recette Kameleon*).

Enfin, Kameleon permet d'échanger des informations entre contexte et nous permettra ainsi de copier des scripts depuis la machine locale vers les machines réservées, mais aussi l'inverse : à la fin de l'expérience, tous les résultats et données seront transférés vers la machine locale.

5.3 Architecture logicielle

L'infrastructure de test de non régression de performance est découpées en deux parties distinctes : la création d'un envrionnement et l'exploitation de cet environnement pour y tester StarPU.

Création d'un environnement

La création d'un environnement consiste à déployer une image Debian vierge, pour y installer toutes les dépendances statiques de StarPU, ainsi que Spack et Execo.

Pour ce faire, on utilise donc Kameleon en spécifiant une recette constituée essentiellement de commandes `apt-get install` (pour les dépendances de StarPU, et de quelques outils), de `pip` pour Execo et d'un `git clone` pour Spack.

Toutes les commandes et scripts utilisés par Kameleon sont enregistrés dans leur contexte d'exécution. Ces informations seront récupérées par nos soins et constitue une trace de la création de l'environnement. On en profitera pour récupérer des informations concernant la réservation Grid5000 stockée sur la frontale, ainsi que des informations sur la machine en elle-même (ses caractéristiques hardware en particulier le CPU (fréquence, cache, gouverneur, . . .)). Ce script m'a été fourni par l'équipe.

La fin de la recette consiste à enregistrer l'image sur la frontale à l'aide de `tgz-g5k` (une recette toute prête existe déjà pour cette opération dans Kameleon).

En définitive, une telle recette Kameleon ne contient qu'une vingtaine de lignes et la création d'un environnement ainsi que son suivi sont considérablement simplifiés par cet outil (voir annexe intitulée *Recette Kameleon de création d'environnement*).

Tests de StarPU

Une fois l'image d'expérimentation créée, on crée une deuxième recette Kameleon afin de déployer cette image, et d'y exécuter des scripts avant de récupérer les résultats de la même manière que pour la création de l'environnement (voir annexe intitulée *Recette Kameleon pour l'exécution de StarPU*).

1. Préparation des tests à effectuer Une expérience consistera à tester StarPU pour une révision SVN donnée (on choisit une révision de StarPU et une révision de Chameleon, ainsi que l'URL de leur branche respective) et une commande donnée (l'exécution d'un script interne à StarPU, réalisant une transformation de Cholesky d'une matrice d'ordre 48000 dans notre cas) (voir annexe intitulée *Définition des expériences à mener*).

   Ces données (révision StarPU, branche StarPU, révision Chameleon, branche Chameleon, commande exécutées) sont entrées dans un fichier CSV dont chaque ligne consti-

tuera une expérience. Pour des raisons de lisibilité, les branches et les commandes seront des alias dont les correspondances figureront dans d'autres fichiers CSV.

2. Exécutions des tests La première partie de la recette Kameleon consistera à générer un fichier CSV sans alias (bien plus pratique à manipuler) à l'aide d'un script Python, qui sera donné à la machine réservée.

   Identiquement à la création d'environnement, on utilisera le script qui m'a été fourni afin de récupérer des informations sur la machine sur laquelle s'est déroulés les tests.

   a) Modification des packages Spack Installer StarPU et Chameleon depuis une révision particulière nécessite de modifier quelque peu les *packages* Spack de StarPU et de Chameleon (un package Spack d'un logiciel indique comment celui-ci doit être compilé et fournis à l'utilisateur des spécifications sur les versions qu'il peut installer, exemple : StarPU v1.2).

      Il nous faudra donc spécifier chacune des installations en fonction de notre fichier CSV d'entrée afin qu'elle puisse être installées avec Spack (voir annexe intitulée *Package Spack*). Ceci est réalisé avec deux scripts Python qui modifieront les packages de StarPU et de Chameleon préexistant.

   b) Exécution de StarPU avec Execo Une fois les packages Spack modifiés il ne reste plus qu'à utiliser Spack afin d'installer la version souhaitée de StarPU pour exécuter la commande associée.

      Pour cela, on utilise Execo en Python qui nous permettra de gérer les sorties standards et sorties d'erreurs simplement.

      Pour chaque expérience spécifiée dans le fichier CSV d'entrée, on utilisera Spack pour installer la version adéquate de StarPU avant d'exécuter la commande correspondante.

5.4 Résultats

Organisation

Pour chaque installation de StarPU, un répertoire est crée contenant un fichier de la sortie standard de l'installation ainsi qu'un fichier pour chaque expérience menée sur cette installation.

Chacun des fichiers résultats est ensuite parsé (toujours à l'aide de scripts Python) afin de générer un fichier CSV global récapitulant tous les résultats des expériences (ce fichier pourra ensuite être utilisé avec R ou d'autres outils afin de les visualiser sous formes de courbes).

Entre temps, pour chaque installation, un fichier Org est crée permettant de récapituler les résultats de ces expériences (on y fait référence aux caractéristiques de la machine où se sont exécutés ces tests, les résultats finaux, les révisions et branches utilisées pour ces tests, ...). Ces fichiers permettent à l'expérimentateur de facilement prendre conscience de la globalité des expériences menées (voir annexe intitulée *Fichier récapitulatif*).

Affichage des résultats

Une fois les résultats résumés dans un fichier CSV, on utilise R pour les visualiser. On se servira de **Flexdashboard** qui est un paquet R permettant d'obtenir des graphes interactifs (zoom, selection de points, . . .) (voir annexe intitulée *Résultats des expériences*).

— 6 —
Conclusion

6.1 Connaissances acquises

Ce stage m'a permis de définitivement me familiariser avec les environnements Linux (ayant l'habitude de coder exclusivement sur Windows, et voyant plutôt Linux comme étant un environnement de travail imposé dans mes études).

La gestion d'un environnement mais aussi le parsing de fichier, le tout en commandes Shell n'a plus rien de mystérieux pour moi. Je me suis également familiarisé avec les connexions SSH et l'usage de Grid5000 m'a permis d'élargir ma vision des plateformes distantes en général.

L'utilisation de Org-mode dans ma prise de note est un outil que j'utiliserai dorénavant très souvent, fournissant un organisation précieuse en particulier dans mon travail où les tâches étaient parsemées selon plusieurs aspects très différents.

6.2 Améliorations / Perspectives futures

De nombreuses améliorations mineures qui sont liées à l'utilisation des outils sont possibles (par exemple, installer plusieurs paquets au sein de Kameleon avec une seule commande `apt` pose des problèmes si un des paquets n'existent plus (les autres ne seront pas téléchargés)), et relèvent plus du paufinage qu'autre chose.

En revanche, d'autre aspects permettent d'augmenter la réplicabilité de cette infrastructure de mise en place de tests. Parmis ces aspects, l'utilisation du script récoltant toutes les données hardwares peut être paramétré pour fonctionner avec StarPU (permettant ainsi de récupérer des informations liées à sa compilation, etc ...)., ou encore l'utilisation du chache persistant de Kameleon, fonctionnalité qui permet d'enregistrer tous les paquets téléchargés et évite les problèmes de versions (la version téléchargée du paquet sera ainsi toujours disponible pour notre usage).

Une autre amélioration possible serait la mise en place des résultats en ligne à la manière de Codespeed ou de Phoronix (ici les résultats sont rappatriés sur la machine locale).

Enfin, les résultats des tests sont différents de ceux préalablement fait à Bordeaux tout au long du développement de StarPU, cela doit être dû à un manque de configuration de StarPU lors de son exécution. Ce point fut mis à l'écart pendant le développement de cette plateforme de tests car il n'en empêchait pas le développement.

Tous ces points auraient pu être abordés mais la durée du stage (3 mois) était bien trop courte pour arriver à cette finalité.

6.3 Bilan

Tous les outils qui ont permis la mise en oeuvre de ce projet ont été conçus à l'Inria dans le cadre de l'expérimentation sur la grille de calcul Grid5000. Ce projet constitue en somme une utilisation agencée de ces outils, inscris dans le contexte de la recherche reproductible. L'utilisation finale de tels outils semble selon moi relativement simple puisque leurs fonctionnalités répondent plus ou moins directement à ce dont j'avais besoin.

Ces outils sont encore en développement et ne sont pas à l'abris de bugs (quelques *issues* ont été créées par mes soins sur GitHub) mais restent globalement d'une accessibilité exemplaire. Leur simplicité d'utilisation ainsi que les possibilités offertes entament largement le problème de la reproductibilité en proposant une description simple de l'expérience menée tout en prodiguant un traçage total de ce qui a été réellement exécuté.

La situation actuelle de cette infrastructure de tests de non régression de performance est très satisfaisante et ne fera qu'évoluer avec le temps lorsque les outils mis en jeu évolueront eux aussi.

— 7 —
Bibliographie

1. Webinar sur la recherche reproductible : `https://github.com/alegrand/RR_webinars`
2. StarPU : `http://starpu.gforge.inria.fr/`
3. Grid5000 : `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`
4. Kameleon : `http://kameleon.gforge.inria.fr/`
5. Spack : `http://software.llnl.gov/spack/`
6. Execo : `http://execo.gforge.inria.fr/doc/latest-stable/`
7. Mon journal : `https://github.com/FlorianPO/Performance-Regression-Testing/blob/master/Journal.org`

— 8 —
Annexes

8.1 Principe d'une recette Kameleon

Une recette Kameleon consiste en l'exécution de commandes dans le contexte correspondant. Il est possible d'hériter d'une autre recette et d'en écraser les variables afin de fournir, par exemple, à la recette Grid5000 son login.

Ces recettes sont divisées en 3 sections : *boostrap* (lors du déploiment de la machine), *setup* (le corps principal de la recette) et *export* qui constitue les dernières instructions lorsque l'expérience principale est terminée (le moment où l'on récupère les données généralement).

Une section est composée d'étapes, de micro étapes et des commandes dans le contexte choisis.

Pour copier des fichiers d'un contexte à un autre on utilise l'instruction *pipe* en se servant de la commande `cat`. Bien d'autres instructions sont fournis par Kameleon, en particulier *breakpoint* qui permet de prendre la main sur les différents contextes, afin d'en vérifier le contenu ou bien d'exécuter des instructions à la main.

```
---
extend: relative_path_to_recipe/recipe.yaml # inherit a recipe

global: # variables definition (variables can be overrided)
  my_g5k_site: grenoble
  my_g5k_user: fpopek

bootstrap: # first commands (right after deployment)
  - "@base" # bootstrap commands of the inherited recipe

setup: # main experiment commands
  - "@base" # setup commands of the inherited recipe
  - first_step:
    - some_packages:
      - exec_in: | # commands in in context (deployed machine)
          apt-get update
          apt-get install -y python2.7-dev python-httplib2 python-pip
      - exec_out: | # commands in out context (frontend)
          ...
```

```
      - exec_local: | # commands in local context (my machine)
          ...
    - my_experimentation:
      - pipe: # used to copy data from one context to another
        - exec_local: cat my_local_file.txt
        - exec_in: cat > same_file_to_in.txt
      - exec_in: |
          ...
      - breakpoint: "Breakpoint reached..."

export: # when the experiment end
  - g5k_custom:
    - in_to_local:
      - pipe:
        - exec_in: cat results.tar
        - exec_local: cat > results.tar

  - save_appliance_from_g5k: # this one is defined in the inherited recipe
                             # works like a function call
```

8.2 Package Spack

Dans un package Spack on retrouve la fonction `install` qui constitue le moyen de configurer le makefile de StarPU en fonction de la version souhaitée.

Chacune des lignes intitulée `version` constitue une version de StarPU installable avec Spack côté utilisateur. Dans notre cas, il suffira d'ajouter ce genre de ligne afin d'installer StarPU depuis une révision :

```
version('svn-trunk-<experiment name>', svn='<branch>', revision=<revision number>)
```

A titre d'exemple, voici le package Spack (épuré) de StarPU :

```
from spack import *
import subprocess
import os
import platform
import spack

class Starpu(Package):
    """offers support for heterogeneous multicore architecture"""
    homepage = "http://starpu.gforge.inria.fr/"

    version('1.2.0rc5', '5ee228354d0575c53e631ed359054cfd',
            url="http://starpu.gforge.inria.fr/files/starpu-1.2.0rc5.tar.gz")
    version('1.2.0rc4', '9509fa4cd2790bc51b164103f2c87f3c',
            url="http://starpu.gforge.inria.fr/files/starpu-1.2.0rc4.tar.gz")
```

```
...
version('git-1.2', git='https://bitbucket.org/jeromerobert/starpu.git/starpu.git'
version('svn-trunk', svn='https://scm.gforge.inria.fr/anonscm/svn/starpu/trunk')

pkg_dir = spack.repo.dirname_for_package_name("fake")
# fake tarball because we consider it is already installed
version('exist', '7b878b76545ef9ddb6f2b61d4c4be833',
        url = "file:"+join_path(pkg_dir, "empty.tar.gz"))
version('src')

variant('debug', default=False, description='Enable debug symbols')
variant('shared', default=True, description='Build STARPU as a shared library')
variant('fxt', default=False, description='Enable FxT tracing support')
...

depends_on("hwloc")
depends_on("mpi", when='+mpi')
depends_on("cuda", when='+cuda')

def install(self, spec, prefix):

    if os.path.isfile("./autogen.sh"):
        subprocess.check_call("./autogen.sh")

    config_args = ["--prefix=" + prefix]
    config_args.append("--disable-build-doc")
    config_args.append("--disable-starpu-top")

    if spec.satisfies('+debug'):
        config_args.append("--enable-debug")
    ...

    config_args.append("--with-hwloc=%s" % spec['hwloc'].prefix)

    if not spec.satisfies('+mpi'):
        config_args.append("--without-mpicc")
    ...

    configure(*config_args)

    # On OSX, deactivate glpk
    if platform.system() == 'Darwin':
        filter_file('^#define.*GLPK.*', '', 'src/common/config.h', 'include/starp

    make()
    make("install", parallel=False)
```

```
    # to use the existing version available in the environment: STARPU_DIR environmen
    @when('@exist')
    def install(self, spec, prefix):
        if os.getenv('STARPU_DIR'):
            starpuroot=os.environ['STARPU_DIR']
            if os.path.isdir(starpuroot):
                os.symlink(starpuroot+"/bin", prefix.bin)
                os.symlink(starpuroot+"/include", prefix.include)
                os.symlink(starpuroot+"/lib", prefix.lib)
            else:
                raise RuntimeError(starpuroot+' directory does not exist.'+' Do you r
        else:
            raise RuntimeError('STARPU_DIR is not set, you must set this environment
```

8.3 Définition des expériences à mener

Le fichier CSV global (avec les alias) des tests de StarPU à menés est de cette forme :

experiment_name	chameleon_branch	chameleon_revision	starpu_branch	starpu_revision	command
svn_1	trunk	2808	trunk	17204	cmd1
svn_2	trunk	2808	trunk	17200	cmd1
svn_3	trunk	2808	trunk	17186	cmd1
svn_4	trunk	2804	trunk	17168	cmd1
svn_5	trunk	2790	trunk	17008	cmd1
svn_6	trunk	2781	trunk	17004	cmd1
svn_7	trunk	2780	trunk	16997	cmd1
svn_8	trunk	2775	trunk	16989	cmd1
svn_9	trunk	2773	trunk	16952	cmd1
svn_10	trunk	2772	trunk	16944	cmd1
svn_11	trunk	2770	trunk	16921	cmd1
svn_12	trunk	2770	trunk	16920	cmd1
svn_13	trunk	2770	trunk	16916	cmd1
svn_14	trunk	2768	trunk	16901	cmd1
svn_15	trunk	2768	trunk	16899	cmd1

Le fichier CSV des branches est :

chameleon_branch_name	chameleon_branch
trunk	`https://scm.gforge.inria.fr/anonscm/svn/morse/trunk/chameleon`

Celui des commandes est :

command_name	command
cmd1	”export STARPU_WORKER_STATS=1 export STARPU_CALIBRATE=2 ./timing/time_spotrf_tile –warmup –gpus=3 –threads=9 –nb=960 –ib=96 –n_range=48000

8.4 Recette Kameleon de création d'environnement

Cette recette consiste à installer les dépendances statiques de StarPU et d'autres programmes (la suite d'instruction `apt`). La partie *export* contient un certain nombre de *pipes* afin de rappatrier sur la machine locale un certain nombre d'information.

Enfin, la toute dernière ligne est une étape contenue dans les recettes héritées qui permettra d'enregistrer l'environnement crée sous forme d'une archive *tgz* sur la frontale.

```
#==============================================================================
# vim: softtabstop=2 shiftwidth=2 expandtab fenc=utf-8 cc=81 tw=80
#==============================================================================
#
# DESCRIPTION: Environment creation
#
#==============================================================================

---
extend: default/grid5000/debian7.yaml

global:
  # You can see the base template ‘default/grid5000/debian7.yaml‘ to know the
  # variables that you can override
  my_g5k_site: grenoble
  my_g5k_user: fpopek
  my_g5k_property: -p \"gpu='YES'\"
  my_g5k_nodes: 1
  my_g5k_walltime: "1:00:00"
  filename: kameleonImage
  filesystem: ext4

bootstrap:
  - "@base"

setup:
  - "@base"
  - first_step:
    - some_packages:
      - exec_in: |
          apt-get update
          apt-get install -y python2.7-dev python-httplib2 python-pip
```

```
          apt-get install -y vim emacs
          apt-get install -y curl patch
          apt-get install -y git subversion mercurial
          apt-get install -y build-essential gfortran
          apt-get install -y autoconf automake cmake cmake-data doxygen texinfo
          apt-get install -y libtool
          apt-get install -y libboost-dev
          apt-get install -y gawk
          apt-get install -y bison flex
          apt-get install -y binutils-dev libelf-dev libiberty-dev
          apt-get install -y libz-dev
          apt-get install -y libqt4-dev freeglut3-dev
          apt-get install -y environment-modules
          apt-get install -y hwloc libhwloc-dev
    - spack_execo:
      - pipe: # copy data info script in node
        - exec_local: cat ../../scripts/get_info.sh
        - exec_in: cat > get_info.s
      - exec_in: |
          # NODE INFO
          bash get_info.sh node_info.org

          # SPACK
          git clone https://github.com/fpruvost/spack
          cd spack/
          git checkout morse

          # EXECO
          pip install --user execo

export:
  - g5k_custom:
    - in_to_local:
      - exec_in: | # Kameleon scripts
          tar -cvf kameleon_scripts_in.tar kameleon_scripts/in/
      - pipe: # PIPE Kameleon scripts
        - exec_in: cat kameleon_scripts_in.tar
        - exec_local: cat > kameleon_scripts_in.tar
      - pipe: # PIPE Node info
        - exec_in: cat node_info.org
        - exec_local: cat > node_info.org
      - exec_local: |
          tar -xvf kameleon_scripts_in.tar
          rm kameleon_scripts_in.tar

    - out_to_local:
```

```
    - exec_out: |
        # Kameleon scripts
        tar -cvf kameleon_scripts_out.tar kameleon_scripts/out/
        # OAR logs
        tar -cvf OAR_logs.tar $(find . -name "OAR.*.$(cat job_id).*")
    - pipe: # PIPE Kameleon scripts
      - exec_out: cat kameleon_scripts_out.tar
      - exec_local: cat > kameleon_scripts_out.tar
    - pipe: # PIPE OAR logs
      - exec_out: cat OAR_logs.tar
      - exec_local: cat > OAR_logs.tar
    - pipe: # PIPE ssh_config
      - exec_out: cat ssh_config
      - exec_local: cat > ssh_config_out
    - exec_local: |
        tar -xvf kameleon_scripts_out.tar
        rm kameleon_scripts_out.tar
        tar -xvf OAR_logs.tar
        rm OAR_logs.tar
    - exec_out: |
        rm kameleon_scripts_out.tar
        rm OAR_logs.tar

- save_appliance_from_g5k:
```

8.5 Recette Kameleon pour l'exécution de StarPU

Cette recette consiste à transférer un certain nombre de scripts dans la machine déployée. Ces scripts constitueront l'expérience principale.

Enfin, dans la section *export*, on utilise également des *pipes* afin de récupérer des données vers la machine locale.

```
#==============================================================================
# vim: softtabstop=2 shiftwidth=2 expandtab fenc=utf-8 cc=81 tw=80
#==============================================================================
#
# DESCRIPTION: StarPU execution
#
#==============================================================================

---
extend: default/grid5000/debian7.yaml

global:
  # You can see the base template ‘default/grid5000/debian7.yaml‘ to know the
  # variables that you can override
```

```
  my_g5k_site: grenoble
  my_g5k_user: fpopek
  my_g5k_property: -p \"gpu='YES'\"
  my_g5k_nodes: 1
  my_g5k_walltime: "2:30:00"
  my_g5k_env: ~/kameleon_workdir/debian_g5k/kameleonImage.env

bootstrap:
  - "@base"

setup:
  - first_step:
    - null_step: # fatal error otherwise
      - exec_in: echo "null_step"
    - local_scripts: # Local scripts
      - exec_local: |
          python ../../scripts_local/csv_filler.py ../../input_data/revisions.csv ../
          cp ../../input_data/revisions.csv ../../scripts/revisions_abstract.csv
          cp ../../scripts_local/csv_reader.py ../../scripts/csv_reader.py
    - scripts_transfer: # Copy every scripts and data files from local to in
      - exec_local: |
          export DIRECTORY=$PWD
          cd ../../scripts/
          tar -cvf $DIRECTORY/scripts.tar *
      - pipe:
        - exec_local: cat scripts.tar
        - exec_in: cat > scripts.tar
      - exec_local: | # Clean local
          rm scripts.tar
          rm ../../scripts/revisions_abstract.csv
          rm ../../scripts/csv_reader.py
      - exec_in: |
          tar -xvf scripts.tar
          rm scripts.tar
      - Breakpoint: "Break..."
    - execo_step: # Main experiment
      - exec_in: |
          # SPACK ENVIRONMENT
          export SPACK_ROOT=$PWD/../debian_g5k/spack/
          export PATH=$SPACK_ROOT/bin:$PATH

          # NODE INFO
          bash get_info.sh node_info.org

          # SPACK PACKAGES
          python chameleon_package_builder.py revisions.csv revisions_abstract.csv $S
```

```
          python starpu_package_builder.py revisions.csv revisions_abstract.csv $SPAC

          # EXECO
          python execo_script.py revisions.csv revisions_abstract.csv

export:
  - g5k_custom:
    - in_to_local:
      - exec_in: |
          # Kameleon scripts
          tar -cvf kameleon_scripts_in.tar kameleon_scripts/in/
          # StarPU results
          tar -cvf starpu_results.tar starpu_results/
      - pipe: # PIPE Kameleon scripts
        - exec_in: cat kameleon_scripts_in.tar
        - exec_local: cat > kameleon_scripts_in.tar
      - pipe:  # StarPU results
        - exec_in: cat starpu_results.tar
        - exec_local: cat > starpu_results.tar
      - exec_local: |
          tar -xvf kameleon_scripts_in.tar
          rm kameleon_scripts_in.tar
          tar -xvf starpu_results.tar
          rm starpu_results.tar
      - pipe: # PIPE Node info
        - exec_in: cat node_info.org
        - exec_local: cat > starpu_results/node_info.org

    - out_to_local:
      - exec_out: |
          # Kameleon scripts
          tar -cvf kameleon_scripts_out.tar kameleon_scripts/out/
          # OAR logs
          tar -cvf OAR_logs.tar $(find . -name "OAR.*.$(cat job_id).*")
      - pipe: # PIPE Kameleon scripts
        - exec_out: cat kameleon_scripts_out.tar
        - exec_local: cat > kameleon_scripts_out.tar
      - pipe: # PIPE OAR logs
        - exec_out: cat OAR_logs.tar
        - exec_local: cat > OAR_logs.tar
      - pipe: # PIPE ssh_config
        - exec_out: cat ssh_config
        - exec_local: cat > ssh_config_out
      - exec_local: |
          tar -xvf kameleon_scripts_out.tar
          rm kameleon_scripts_out.tar
```

```
          tar -xvf OAR_logs.tar
          rm OAR_logs.tar
      - exec_out: |
          rm kameleon_scripts_out.tar
          rm OAR_logs.tar

    - data_csv: # Retrieving StarPU data in a CSV file
      - exec_local: |
          python ../../scripts_local/org_builder.py ../../scripts/revisions.csv ../..
          python ../../scripts_local/data_csv.py starpu_results/
          rm ../../scripts/revisions.csv
          rm ../../scripts_local/csv_reader.pyc
```

8.6 Fichier récapitulatif des résultats

```
#+TITLE: Experiment results
#+DATE: 05/08/2016 13:56:11
#+AUTHOR: root
#+MACHINE: adonis-9.grenoble.grid5000.fr
#+FILE: chameleon_trunk_2914_starpu_trunk_17971.org

* Host information
[[file:../node_info.org]]
* Software revisions
** Chameleon
#+BEGIN_EXAMPLE
chameleon_rev: 2914
chameleon_bch: https://scm.gforge.inria.fr/anonscm/svn/morse/trunk/chameleon
#+END_EXAMPLE
** StarPU
#+BEGIN_EXAMPLE
starpu_rev: 17971
starpu_bch: https://scm.gforge.inria.fr/anonscm/svn/starpu/trunk
#+END_EXAMPLE
* Compilation
[[file:./compil_chameleon_trunk_2914_starpu_trunk_17971]]
* Experimental results
** XP1
*** Command
#+begin_src sh :results output :exports both
export STARPU_WORKER_STATS=1
export STARPU_CALIBRATE=2
./timing/time_spotrf_tile --warmup --gpus=3 --threads=9 --nb=960 --ib=96 --n_range=48
#+end_src
*** Standard output
```

```
#+BEGIN_EXAMPLE
#
# CHAMELEON 0.9.1, ./timing/time_spotrf_tile
# Nb threads: 9
# Nb GPUs:    3
# NB:         960
# IB:         96
# eps:        5.960464e-08
#
#     M       N  K/NRHS   seconds   Gflop/s Deviation
  48000   48000       1   256.086    143.96 +-   0.00

#+END_EXAMPLE
*** Standard error
#+BEGIN_EXAMPLE
[starpu][starpu_initialize] Warning: StarPU was configured with --with-fxt, which slo
[starpu][_starpu_bind_thread_on_cpu] Warning: both workers 0 and 8 are bound to the s

#---------------------
Worker stats:
CPU 0
3356 task(s)
CPU 1
6085 task(s)
CPU 2
6070 task(s)
CPU 3
6092 task(s)
CPU 4
6078 task(s)
CPU 5
6074 task(s)
CPU 6
6073 task(s)
CPU 7
6061 task(s)
CPU 8
3311 task(s)
#---------------------

#+END_EXAMPLE
*** Result
#+BEGIN_EXAMPLE
XP1_Flops: 143.96
#+END_EXAMPLE
```

8.7 Résultats des expériences menées

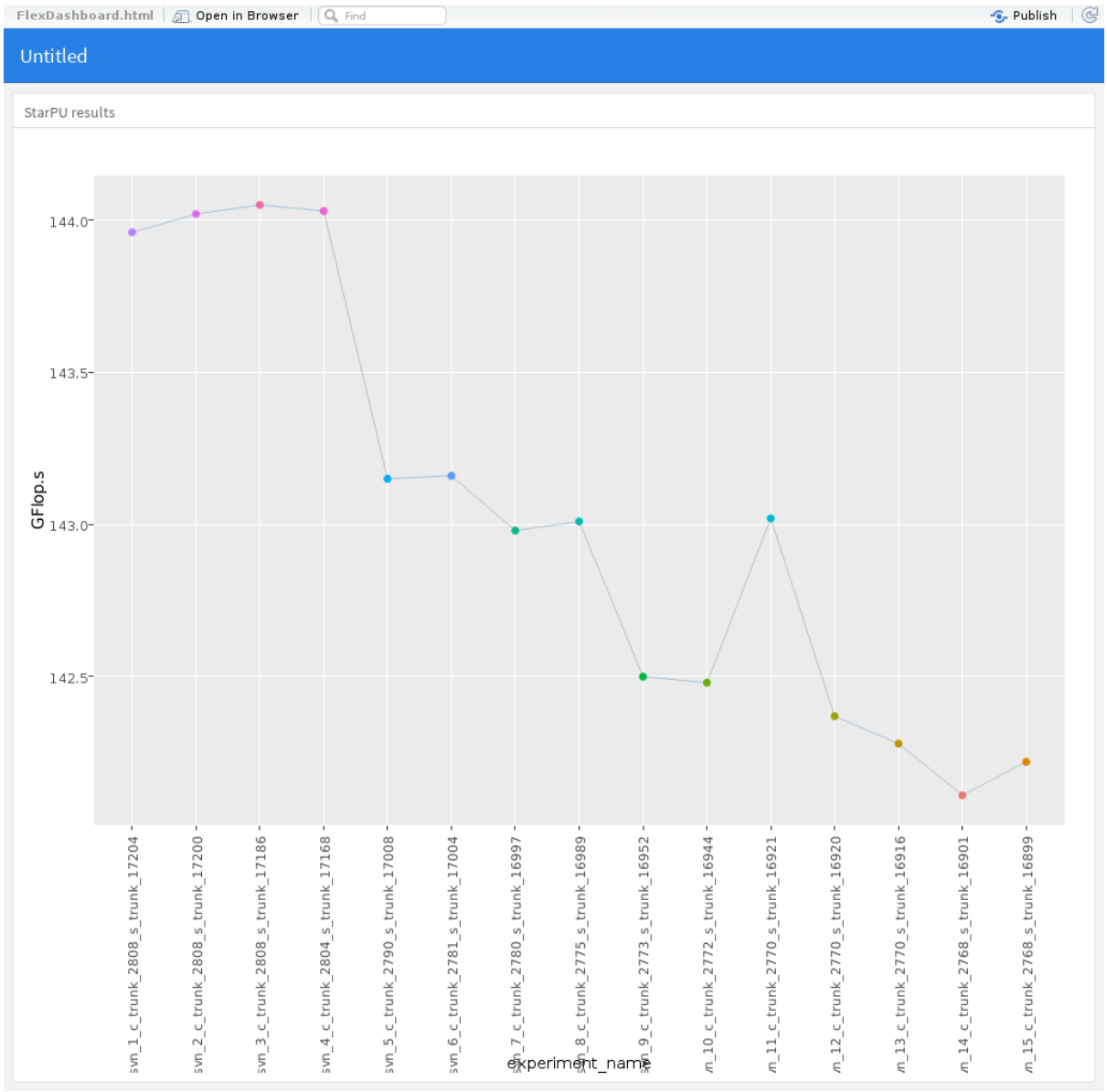

FIGURE 8.1 – Les résultats des expériences StarPU avec Flexdashboard en R

Etudiant : Florian Popek	**Année d'étude dans la spécialité :** 4ème
Entreprise : Laboratoire Informatique de Grenoble **Adresse complète :** 700 Avenue Centrale, 38400 Saint-Martin-d'Hères **Théléphone :** 04 57 42 22 42	
Responsable administratif : M.GROS Patrick **Téléphone :** non renseigné **Courriel :** non renseigné	
Tuteur de stage : M. LEGRAND Arnaud **Téléphone :** 06 51 49 12 21 **Courriel :** arnaud.legrand@imag.fr	
Enseignant référant : M. PALIX Nicolas **Téléphone :** 04 76 51 46 27 **Courriel :** nicolas.palix@ujf-grenoble.fr	

Titre : Mise en place de tests de non régression de performance

Résumé :

Dans ce document, nous abordons le problème de la reproductibilité des expériences, en particulier dans le domaine HPC (High Performance Computing), selon deux aspects qui sont: le contrôle de l'environnement et le pilotage de l'expérience, en plus de quelques considérations autour de la reproductibilité en général. On y présentera notre solution basée sur des technologies développées à l'Inria permettant de déployer un environnement afin d'y conduire des expériences. L'objectif final est d'évaluer les performances (benchmarker) de StarPU tout au long de son évolution afin d'y déceler de potentielles régressions entre deux versions, tout en ayant un contrôle parfait de l'environnement et de l'expérience.

La mise en place de cette infrastructure de tests est réalisé à l'aide d'outils réalisés à l'Inria tels que Kameleon, Spack, ou encore Execo. Ces tests seront exécutés sur la grille de calcul Grid5000. Ce document décrit comment, à l'aide de ces outils, la chaîne complète allant de la création d'un environnement jusqu'à son déploiement et son utilisation est réalisée.