Portage de StarPU sur la bibliothèque de communication NewMadeleine

Guillaume Beauchamp

Mémoire de stage : Portage de StarPU sur la bibliothèque de communication NewMadeleine

Beauchamp Guillaume

Université de Bordeaux

Encadré par Alexandre Denis
dans l'équipe Inria TADaaM.

13 septembre 2017

Table des matières

1 Introduction

Dans le cadre du calcul haute performance, il est généralement préférable d'utiliser la totalité des cœurs disponible sur une ou des machines, afin d'augmenter la puissance de calcul totale. Les noeuds modernes ont également fréquemment une architecture hétérogène ayant de multiples cœurs ainsi que des accélérateurs(GPU,etc). Il serait ainsi souhaitable qu'un calcul soit répartit entre les accélérateurs et les cœurs mais cela implique de réécrire des parts du programme parallélisable dans différents langages, certains périphériques ayant des contraintes particulière, ainsi que faire des transferts de données entres ces périphériques et les cœurs. De plus ces accélérateurs et les cœurs n'ont généralement pas la même efficacité sur les différentes parties d'un programme, ce qui rend difficile d'effectuer une répartition optimale du travail. Enfin de nombreuses librairies, comme le solveur Chameleon développé à l'Inria, ne peuvent pas être développées a destination d'une architecture spécifique, et doivent donc adapter la répartition des données aux ressources disponibles.

Un ordonnanceur de tache comme StarPU [1] , utilisé par Chameleon[2], permet de simplifier le développement de tels programmes parallèles, en gérant la répartition du calcul et d'effectuer les communication nécessaires, les applications devant seulement découper leur applications en taches indiquant les données en entrées et en sorties. L'ordonnanceur organise ensuite les taches de manière efficace et en effectue les transferts de données nécessaires à leur exécution. Bien qu'il tente de réduire leur nombre et de les recouvrir(exécuter un calcul simultanément a un transfert de données au lieu de devoir attendre la fin du transfert pour reprendre le calcul), la durée et l'efficacité du recouvrement de ces transferts ont une influence importante sur le temps de calcul.

StarPU possède une interface StarPU-MPI permettant d'effectuer des communications nœuds a nœuds en se basant sur la norme de librairie de communication MPI. L'implémentation de StarPU-MPI actuelle l'utilise pour faire des transferts de données entre plusieurs machines. Cependant bien qu'étant les librairies de communication les plus utilisées leur modèle envoi/réception n'est pas vraiment adapté aux besoins de StarPU, n'assurant pas la progression des communications hors des appels MPI. De plus l'implémentation actuelle est peu efficace avec un grand nombre de requêtes.

La librairie NewMadeleine qui permet d'utiliser un modèle basé sur des appels distants de procédures(RPC) en plus du modèle envoi/réception pourrait permettre de résoudre ces problèmes, garantissant sans appels de l'application la progression des communi-

cation. Elle fournit de plus une interface MadMPI qui pourrait, dans un premier temps, faciliter la transition. Nous avons donc développé une nouvelle implémentation de StarPU-MPI qui serra appelée StarPU-NMad basant ses communications inter-nœuds d'abord sur Mad-MPI puis dans un second temps sur NewMadeleine.

Nous présenterons dans un premier temps, StarPU, MPI et NewMadeleine. Puis nous décrirons notre port d'une implémentation basée sur MPI de StarPU-MPI a une implémentations basée sur NewMadeleine. Enfin nous étudierons les impacts de ce port sur la performance de StarPU.

2 Contexte

2.1 StarPU

La librairie StarPU [3] [1] est une librairie de calcul haute performance développée à l'INRIA par l'équipe STORM. Elle permet de paralléliser sur un seul nœud (mémoire partagée) un programme décrit a l'aide de taches. Elle est notablement utile pour paralléliser des programmes ayant des dépendances de données complexes ou sur un support hétérogène (accélérateurs/GPU/cœurs).

Les taches sont définies par une ou plusieurs implémentations destinées à un/des supports d'exécutions, ainsi qu'une liste de paramètres ayant différent mode d'accès (lecture, écriture, lecture-écriture).

StarPU détecte les supports d'exécutions disponibles puis y distribue les taches, de manière à assurer la cohérence de données. Ainsi pour une donnée, l'ordre d'exécution entre deux écriture et entre une écriture et une lecture soit le même que lors de l'ajout de la tache. L'ordre entre lectures n'est pas important, et plusieurs taches exécutées simultanément peuvent avoir une même donnée en lecture. StarPU effectuera les transferts de données nécessaires.

StarPU tente de faire les transferts de données dès qu'elles sont disponibles pour réduire le coût des communications.

2.2 MPI

MPI est une interface de communication par passage de messages. Elle est le moyen le plus utilisé pour paralléliser des application sur plusieurs nœuds (mémoire distribuée).

Un communicateur représente un groupe de processus MPI (généralement sur des nœuds différents) avec lesquels on peut communiquer. Un outil (mpirun/mpiexec) est fournit par les implémentations d'MPI qui permet de lancer des processus exécutant un même programme sur différents nœuds. Ces différentes instances du programmes ont un communicateur en commun qui leur permet de connaître le nombre d'instance et leur rang parmi elles.

Pour effectuer une communication l'émetteur du message doit poster un send, et le récepteur un receive. Les send/receive ont pour paramètre le communicateur utilisé, un tag pour distinguer différents message, et le rang de la source/destination. Les paramètres ANY_TAG, ANY_SOURCE peuvent être utilisé lors d'une réception pour recevoir un message ayant n'importe quel tag ou venant de n'importe quel source. Le receveur fera correspondre les messages

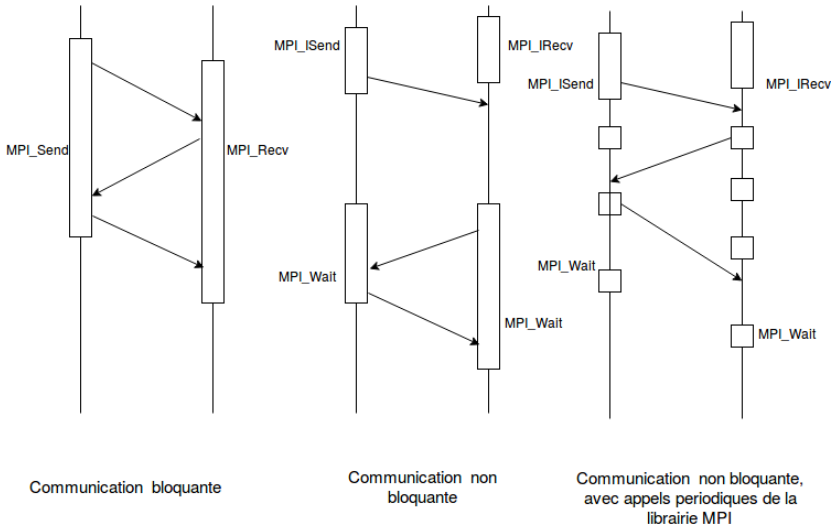

FIGURE 1 – Progression caractéristiques de nombreux protocoles MPI

reçus selon ces paramètres.

Selon la taille du message il peut être envoyé de deux manières :

— Eager pour les petits messages. Quand l'envoi est posté le message est directement envoyé au receveur. Le message y est stocké par le receveur jusqu'à ce qu'il post un receive qui copiera alors le message déjà reçu dans un buffer. Ce protocole réduit la latence mais utilise plus de mémoire et augmente le nombre de copie.

— Rendez-Vous pour les plus gros message. L'émetteur demande au receveur via eager si il est près a recevoir le message. Une fois le receive posté, une réponse est envoyé, puis l'émetteur envoie le message en une ou plusieurs parties selon sa taille.

MPI fournit également des fonctions non-bloquantes isend/ireceive, qui sont souvent utilisées afin de pouvoir recouvrir les communications par le calcul. Cependant le standard MPI ne garantit pas la progression des communications MPI non bloquantes hors des appels de la librairie. Selon l'implémentation de MPI utilisée, il peut donc être nécessaire de lui faire des appels périodiquement pour faire progresser les communications (Fig 2).

2.3 StarpuMPI

Le module StarpuMPI [4] permet d'utiliser StarPU sur plusieurs nœuds (en mémoire distribuée). Chaque nœuds exécute une instance de Starpu, les communications inter-nœuds sont effectuées par MPI et les communications intra-nœud sont inchangées. Il permet

de communiquer implicitement les données nécessaires a l'exécution des taches sur plusieurs nœuds, de faire des communications explicites a l'aide d'un interface send/receive propre a StarPU-MPI qui va planifier une communication dès que la donnée est disponible ainsi que de faire des MPI_Send/Receive directement sur des données.

Chaque donnée a un tag unique et distinct ainsi qu'un nœud propriétaire qui l'envoie aux nœuds la nécessitant pour exécuter une tache. Tous les nœuds connaissent le diagramme de tache et savent donc quand les autres nœuds ont besoin de quelles données. Les données ont un nœud propriétaire qui les envoie aux autres nœuds si ils ont besoin d'y accéder.

Les taches sont réparties de manière a minimiser les écritures sur des données qui ne sont pas propriétaires du nœud. Le communications se font avec des communication MPI non-bloquantes isend/ireceive, selon le protocole suivant :

— Lors d'une lecture, si la donnée n'a pas déjà été transmise au nœud exécutant la tache, elle lui est envoyé par son nœud propriétaire. Elle est conservée jusqu'à ce que le nœud receveur voit qu'un écriture doit avoir lieu sur cette donnée.

— Lors d'une écriture, la donnée est de même reçu si besoin par le nœud exécutant la tache puis renvoyée modifié au nœud propriétaire. Tous les autres nœuds voyant qu'une écriture est prévue dans leur diagramme de tache invalident la donnée.

Pour faire progresser les communications MPI, il est nécessaire de faire des appels fréquents a la librairie. Ces appels sont effectués par un thread de communication propre a chaque nœud.

Lorsqu'une tache souhaite débuter une communication, elle empile une requête StarpuMPI. Le thread de communication propre a chaque nœuds débute toutes les communications MPI empilé, puis teste si elles terminent de manière active. Cela garantit la progression et que MPI n'est appelé que depuis un seul thread (la mojorité des implémentations de MPI ne supportant pas, ou pas efficacement de faire de communications depuis plusieurs thread.).

Cette implémentation a deux problèmes majeurs :

— un cœur est effectivement sacrifié pour effectuer cette attente active, tant que des communications sont en attente.

— Le thread doit parcourir et tester toutes les requêtes en attente pour trouver qu'une communication s'est terminée. Cela cause une augmentation linéaire de la latence si un grand nombre de requêtes sont en attente.

Dans la dernière version de StarPU (1.2/1.3), des optimisations spécifiques à certaines implémentations MPI (en particulier OpenMPI) ont été ajoutées dans le module. Par exemple, les communications

MPI n'utilisent qu'un seul tag et le tag des données StarPU est représenté par un champ supplémentaire de la donnée MPI, certaines implémentation d'MPI ayant un surcoût pour un nombre élevé de tags.

2.4 NewMadeleine

NewMadeleine [5] est une librairie de communication développée à l'INRIA dans l'équipe TADaaM similaire mais plus bas niveau à MPI dont elle a une implémentation MadMPI.

NewMadeleine est capable de fusionner des communications de petite taille selon leur destination. Bien que pouvant augmenter la latence si les communications ne sont pas attendues, envoyer des communications de plus grosse taille permet d'atteindre un débit plus élevé. Elle maintient cependant une latence comparable a MPI.

Elle fournit, en plus d'une interface send/receive, une interface Remote Procedure Call. Le receveur enregistre un service sur un tag, un masque et une session (équivalent du communicateur MPI). Deux fonctions sont associées au service :

— Le handler, lit l'entête du message et alloue l'emplacement où recevoir le message (cette interface peut donc envoyer directement des messages de taille variable contrairement au send/-receive de MPI/nmad)
— Le finaliser, appelé quand le message a été reçu

Ces fonctions sont appelées lors de la réception des données par NewMadeleine et ne doivent pas faire d'appels coûteux ou bloquants, la progression des communications ne pouvant reprendre qu'après leur terminaison.

Elle fournit une implémentation de MPI, MadMPI, mais qui limitée par l'interface MPI ne permet pas d'utiliser toutes les fonctionnalités (RPC) de NewMadeleine. La fusion des messages courts ne nécessitant pas de changements particuliers de l'interface, elle reste possible.

Les communications progressent grâce à la librairie PIOMan [6] .. Trois mécanismes permettent de fait progresser les communications :

— idle : un thread a faible priorité pouvant être exécute fréquemment (appelé au plus tous les $\sim 5\mu s$). Ayant une priorité plus faible que le calcul il s'exécute sur les cœurs inactifs.
— timer : un thread a priorité normale garantit la progression (appelé tous les $\sim$ 2ms), mais pas suffisant pour avoir une latence acceptable pour de nombreuses applications.
— explicit polling : Lors d'un test/wait, le thread faisant l'appel fait progresser les communications.

3 Objectifs

3.1 Portage de StarPU sur NewMadeleine

Les librairies MPI classiques ne correspondent pas aux besoins spécifiques d'une application StarPU, un passage a NewMadeleine serrait intéressant pour les raisons suivantes :

— Les performances du module StarPUMPI chutent pour un grand nombre de requêtes en attente, d'un coté car certaines implémentation de MPI et en particulier OpenMPI, peinent pour un nombre important de requêtes en attente, de l'autre car le thread de progression a par nature une complexité linéaire au nombre de requête en attente pour tester les requêtes terminées. Une implémentation RPC utilisant NewMadeleine, qui est efficace même avec un grand nombre de requêtes en attente, ayant une complexité selon leur nombre constante au lieu de linéaire, et permettrait également d'éviter de maintenir une liste des requêtes en attente, réduirait le surcoût causé par les requêtes en attentes..

— Supprimer le thread de progression, la progression étant assurée par NewMadeleine. Cela pourrait améliorer sensiblement le recouvrement, ne faisant plus une attente active quand une communication est en attente.

Nos objectifs sont les suivants :

— Obtenir une meilleur scalabilité en nombre de requêtes.
— Avoir un meilleur recouvrement
— Un impact positifs causé par la fusion des petits messages
— Garder l'augmentation éventuelle de la latence sous un seuil raisonnable.

3.2 Travail effectué

— Ajout d'un module StarPUNMad basé sur la version 1.1 de StarPU-MPI et 1.3 de StarPU
— Compiler le module avec MadMPI, qui nous permettra une transition plus facile vers NewMadeleine.
— Port du module sous NewMadeleine, en utilisant la progression idle de PIOMan au lieu d'un thread de progression. L'utilisation par taches étant l'élément principal, mais on souhaite que l'utilisation de NewMadeleine soit transparente pour un utilisateur de StarPU.
— Étude de performance du port

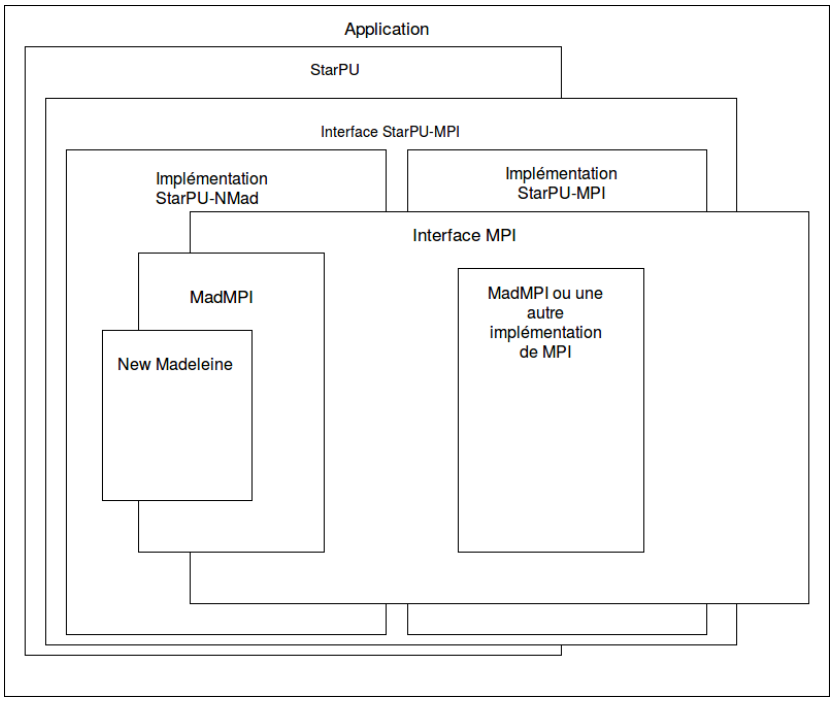

FIGURE 2 – Le nouveau module StarPU-NMad implémentant StarPU-MPI

4 Portage sur Newmadeleine

4.1 Ajout d'un nouveau module compilant avec MadMPI

StarPU est compilé avec Autotool. Afin de faciliter une intégration future, nous avons ajouté un dossier MadMPI dans StarPU et modifié les fichier d'autoconf/automake de manière a ce que le module ne soit compilé que si l'option –enable-nmad est passée a autoconf et la librairie MadMPI trouvée (passée via –with-mpexec=, –with-mpicc=, –with-mpiexec=). Elle désactive également la compilation du module mpi, cependant la librairie StarPU-MPI est compilé quand même mais a partir du dossier MadMPI au lieu de MPI.

Nous avons utilisé la version du module starpu-mpi 1.1 comme base de notre module MadMPI, n'intégrant pas encore des optimisations spécifiques à certaines implémentations de MPI qui serraient inutiles ou contre-productives avec la librairie NewMadeleine. Cependant l'interface de StarPU-MPI ayant changé entre 1.1 et 1.3,, nous avons du la porter a la nouvelle interface.

Nous avons du faire les changements suivants :

— Ajout de fonctions faisant des appels indirects a MPI pour Simgrid (une librairie permettant de simuler un cluster plus grand que celui disponible) : trivial a porter, le support de Sim-

grid n'étant pas encore intégré dans StarPU-MadMPI, nous faisons directement un appel MPI.
— Changement de paramètrestype de retours : trivial
— Ajout du sous modules starpu_mpi_datatype gérant les types MPI définit par l'utilisateur : intégré depuis StarPUMPI1.3.
— Ajout de diverses fonctions permettant d'initier un transfert de données : change seulement l'initialisation des requêtes StarPU-MPI.
— Port des tests de la version 1.3 de StarPU-MPI afin de tester la nouvelle interface, il a également fallu les rendre plus rigoureux au standard MPI, MadMPI étant moins tolérant que OpenMPI/IntelMPI. Ainsi, par exemple, il refuse de s'initialiser si pas appelé par mpirun/exec (le standard recommence qu'un communicateur contenant 1 seul nœuds soit créé).

4.2 Portage du module StarPU-nmad sur NewMadeleine

Garder MadMPI

Nous souhaitons passer de MadMPI a NewMadeleine afin de pouvoir faire des requêtes ne nécessitant pas d'être complétées par un thread en attente active. Nous souhaitons cependant respecter l'interface StarPU-MPI afin de minimiser les changements nécessaires pour une application passant de StarPU-MPI a StarPU-NMad.

L'interface StarPU-MPI prends en paramètre des types spécifiques de MPI, afin notamment de définir comment des données sont représentées en mémoire, ou de créer des communicateurs regroupant des nœuds différents. Elle permet de plus de mélanger des communications MPI écrites par l'application utilisateurs avec les communications gérées par StarPU.

Il est donc nécessaire de continuer a inclure MadMPI, l'utilisateur pouvant donc continuer a faire des requêtes MPI . Nous avons également utilisé certaines de ses fonctions internes afin de pouvoir convertir les types MPI en types nmad, et empaqueter les données selon la représentation qu'ils décrivent.

Enfin certaines fonctionnalités de MPI n'ont pas d'équivalent dans NewMadeleine, et ne seraient donc pas amélioré, voire dégradé suite la perte d'optimisations spécifiques, par un passage de MadMPI à une nouvelle implémentation basée sur NewMadelaine. Par exemple, starpu_mpi_barrier, reste implémenté par un appel de MPI_Barrier.

Nous voulons principalement remplacer les initialisation des requêtes Send/Receive, le thread de progression gérant leur complétion et donc également les Test/Wait, où l'utilisateur peut tester/attendre

la fin d'une requête.

Choix du protocole : RPC ou Send/Receive

StarPU nmad possède deux paradigmes pour communiquer : le send receive similaire a MPI, et le Remote Procedure Call. Nous comptions initialement utiliser des communications RPC de NewMadeleine pour remplacer le send/receive de MPI. Ainsi quand nous recevons une communication, le handler est appelé et nous n'avons donc pas besoin de tester chacune des requêtes en attente, pour trouver laquelle s'est terminée. Cependant que faire si on reçoit une donnée qui n'est pas encore attendue ? On ne peut pas remplacer directement la donnée par sa version modifiée que l'on vient de recevoir, en effet il est possible que notre StarPU local soit moins avancé dans l'exécution du diagramme de tache que l'émetteur et que nous ayons encore des taches accédant en lecture a cette donnée.

Il serrait donc nécessaire de maintenir une liste des transfert reçu mais pas encore attendu. Ce qui est déjà fait pour un Send/Receive de manière plus efficace (protocole eager, qui est inefficace pour les gros messages).

Il serrait certes également possible de définir un handler pour chaque requête prête a être reçu, mais ce serrait coûteux et pourrait de causer de effets de course.

Cependant NewMadeleine offre la possibilité d'utiliser les send/recieve de manière événementielle en définissant un callback sur les communications Send/Receive appelé a la terminaison de la requête. Il n'est alors pas nécessaire de tester la terminaison. Les requêtes utilisent un protocole similaire a MPI pour gérer les réceptions inattendues.

Débuter une requête

Pour faire un send/receive,l'utilisateur ou StarPU appelle notre starpu_mpi_(i)send/recieve(_detached), en nous passant un Communicateur, un tag, une donnée, un format MPI. Si la fonction est détachée il peut également nous passer une fonction de callback utilisateur, à appeler a la fin de la requête (et qui dans ce cas ne serra pas attendue/test par l'utilisateur). Il attends en retour un statut MPI, lui indiquant si la requête a réussi.

Ces paramètres combiné a une requête MPI étaient contenu dans un type de donnée requête StarPU-MPI, donné a utilisateur afin qu'il puis test/wait les requêtes. StarPU-MPI poussait tous ces requêtes StarPU-MPI dans une pile des requêtes a initialiser. Le thread de

communication faisait l'appel MPI asynchrone correspondant, puis ajoutait les paramètres et la requête en attente dans une liste des requêtes en attente. Cela permet de lancer toutes les requêtes MPI depuis un seul thread, ce qui pour certaines implémentations MPI est préférable.

Dans notre implémentation NewMadeleine nous n'avons pas cette préoccupation et pouvons lancer des requêtes NewMadeleine depuis n'importe quel thread. Une fois que MadMPI a converti le communicateur et le tag MPI en une session et tag NewMadeleine nous pouvons directement lancer un nmad_send/receive asynchrone. Nous lui ajoutons un callback NewMadeleine qu'il appellera lors de la complétion de la requête.

Dans les deux versions, si la donnée est de taille variable, nous envoyons la taille suivit de la donnée et nous recevons la taille puis dans le callback nous recevons la donnée, aillant besoin de sa taille.

Progression et complétion des requêtes

Dans l'implémentation StarPU-MPI, le thread de progression initialisait les requêtes dans sa pile des requêtes a initialiser et les mettait dans celle des requêtes en attente. Il testait ensuite une a une les requêtes en attente. Quand une finissait, il marquait les données comme reçues (ce qui permettait de débloquer une éventuelle tache en attente), marquait la requête comme terminée et réveillait les threads en attente de la fin de la requête (ayant fait un starpu_mpi_wait), via une variable conditionnelle. Les piles/files et la variable conditionnelle signalant la fin de la requête étaient protégées par un mutex. Le thread dormait si il n'y avait aucune requête en attente d'être initialisée ou pas encore complétée. Sinon il faisait une attente active, faisant progresser les communications grâce a ses test des requêtes en attentes.

Dans notre implémentation StarPU-NMad, la progression des requêtes est assurée par NewMadeleine. Le thread faisant la progression de NewMadeleine, appellera lors de la complétion d'une requête le callback que nous lui avons définit. Il ne peut pas assurer la progression quand il l'exécute, il est donc nécessaire de ne pas faire d'appels bloquant, ni de taches lourdes (comme envoyer des données sur un GPU) afin que la progression ne soit pas arrêté pendant une longe durée. Si il n'y a pas de callback voulu par l'utilisateur, nous marquons les données comme reçues (non bloquant, les threads internes de StarPU font une attente active dessus) ainsi que la requête comme terminée. Pour réveiller les threads attendant la fin d'une communication, nous utilisons une variable conditionnelle

non bloquante fournis par Pioman.

Callbacks Starpu-MPI

Dans l'implémentation StarPU-MPI, l'exécution des callbacks fournit par l'utilisateur est triviale : il suffisait de les faire exécuter par les thread de progression lorsqu'il marque la requête comme terminée.

Cependant,dans notre implémentation StarPU-NMad, il est inenvisageable de faire exécuter un callback demandé par l'utilisateur par notre callback NewMadelaine. Par exemple, il pourrait tenter d'y faire une communication synchrone, or pour pouvoir commencer la communication NewMadeleine va devoir attendre la fin du callback, ce serait donc un deadlock. Nous avons donc ajouté un thread de callback.

Lorsqu'une communication pour laquelle l'utilisateur veut un callback, nous empilons dans une pile non bloquante la requete StarPU-MPI contenant les arguments qu'il nous a passé ainsi que la requête NewMadeleine. Nous utilisons ensuite un sémaphore pour réveiller le thread de callback. Le thread de callback se réveille suite a la modification du sémaphore, et exécute les callbacks présents dans la pile. Quand un callback se termine il marque la requête comme terminé de la même manière que les requêtes sans callback.

Si il y a un callback le thread l'ayant exécuté libère la mémoire de la requête StarPU-MPI sinon elle sera libérée après le premier Wait/Test sur la requête.

Nous avons également envisagé de réduire le nombre de réveils du thread de callback en testant lors d'une réception si la donnée n'a pas déjà été reçu et dans ce cas appeler nous même le callback donné NewMadeleine au lieu de l'assigner a la requête pour que NewMadeleine l'appelle. Ainsi nous pourrions y faire des appels bloquant et donc appeler nous même le callback demandé par l'utilisateur. Cependant nous avons noté que les callbacks sont relativement rare, et avons décidé de ne pas l'optimiser. Par exemple si des données de taille variables ne sont pas envoyées, et qu'on utilise seulement des taches, c'est a dire dans le cas le plus fréquent, aucun callback n'a lieu et le thread de callback reste endormi.

5 Performances de StarPU-nmad

La librairie MPI utilisée dans les performances suivantes est Intel MPI [7]. Les communications ont lieu sur un réseaux InfiniBand. Les communications ont été demandées de manière implicite par

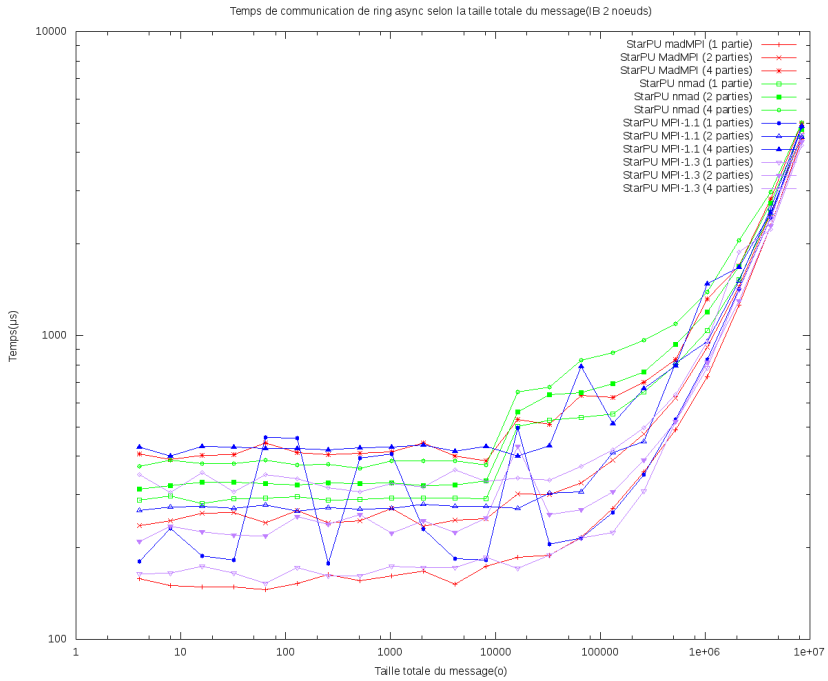

FIGURE 3 – Double latence selon la version de StarPU-MPI et le nombre de données transmises

transfert de paramètres suite a l'exécution d'une tache nécessitant des paramètres provenant d'un autre nœud, cette utilisation étant la plus commune.

5.1 Latence

La principale dégradation possible est la latence qui reste importante dans de nombreuses applications. Nous l'avons donc mesuré pour les différentes versions de StarPU-MPI selon la taille des messages(Fig 3). Nous créons des taches sur 2 nœuds dépendant des même données, qu'elles doivent s'échanger. Les données sont envoyées avec un ou plusieurs paramètres, afin de tester la capacité de NewMadeleine d'agréger des messages.

On remarque que pour envoi d'un seul message notre implémentation basée sur NewMadeleine a une latence nettement supérieure (deux fois plus élevée, pour des messages n'ayant pas une très grande taille) aux implémentations MPI, en particulier celle basée sur MadMPI. Certes une attente active peut être plus réactive et NewMadeleine la fusion des communication peut aussi la dégrader, mais l'écart reste surprenant.

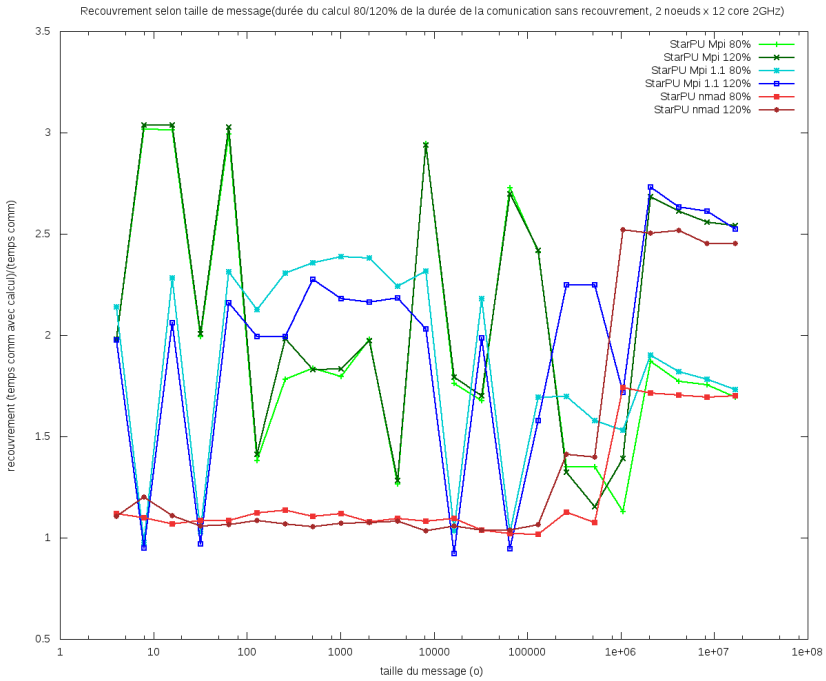

FIGURE 4 – Taux de recouvrement selon la version de StarPU-MPI et la taille des données transmises

En revanche envoyer une donnée en plusieurs messages a un surcoût largement inférieur, l'implémentation MadMPI ne semblant pas pouvoir les fusionner, de même que les autres implémentations MPI qui payent autant de fois la latence.

5.2 Recouvrement

Pouvoir recouvrir efficacement nos communications permettrait de réduire le coût de cette latence en en réduisant l'impact sur les calculs. Nous avons donc ajouté à l'application précédente des taches exécutant un calcul simultanément aux communications, d'une durée équivalente a celle de la communication (80/120% de la durée originale de la communication), et mesuré l'impact sur leur durée(Fig 4). Le recouvrement est calcul par le temps d'une exécution avec communication et calculs divisé par celui d'une exécution avec seulement les calculs. Un ratio de 1, représente un recouvrement parfait et de 2 équivaut a une exécution séquentielle.

Nous avons noté que l'implémentation StarPU-MPI tend a avoir des performance peu stables quand du calcul s'exécute simultanément aux communication, le thread de communication pouvant avoir des

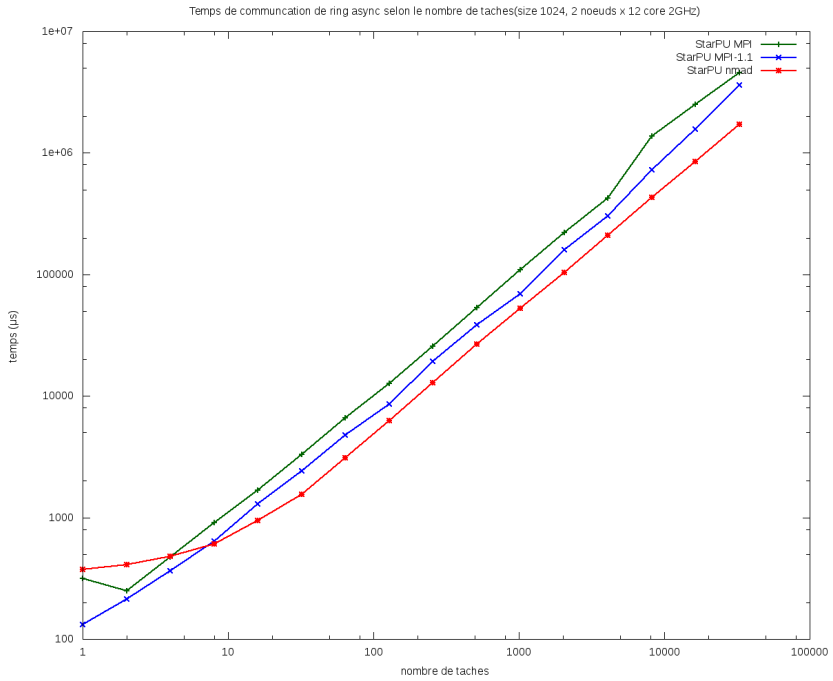

FIGURE 5 – Temps total de communication selon nombre de taches communicant

difficultés a la faire progresser, rendant fréquemment les communications et le calcul séquentiel. A l'inverse l'implémentation NewMadeleine a un excellent recouvrement, l'ajout de calcul ne dégradant que légèrement la durée des communications.

5.3 Scalabilité en nombre de requêtes

Notre objectif principal reste d'améliorer la scalabilité en nombre de requête. Ainsi au lieu d'échanger un message entre deux taches sur deux nœuds, nous avons tenté d'échanger N messages entre N tache sur les 2 nœuds, augmentant ainsi le nombre de requêtes en attente(Fig 5).

On note que l'implémentation utilisant NewMadeleine est rapidement plus efficace que les autres jusqu'à avoir un temps total en moyenne de 2 à 3 fois plus court.

Si on calcule le temps entre deux envois (Fig 6) de ces messages on observe que d'un coté pour un grands nombre de taches les implémentations StarPU-MPI perdent en efficacité soit notre problème original contrairement a notre implémentation, mais que la diminution du temps entre deux envois suite a l'envoi successifs de données par NewMadeleine est beaucoup plus prononcé, qui ne

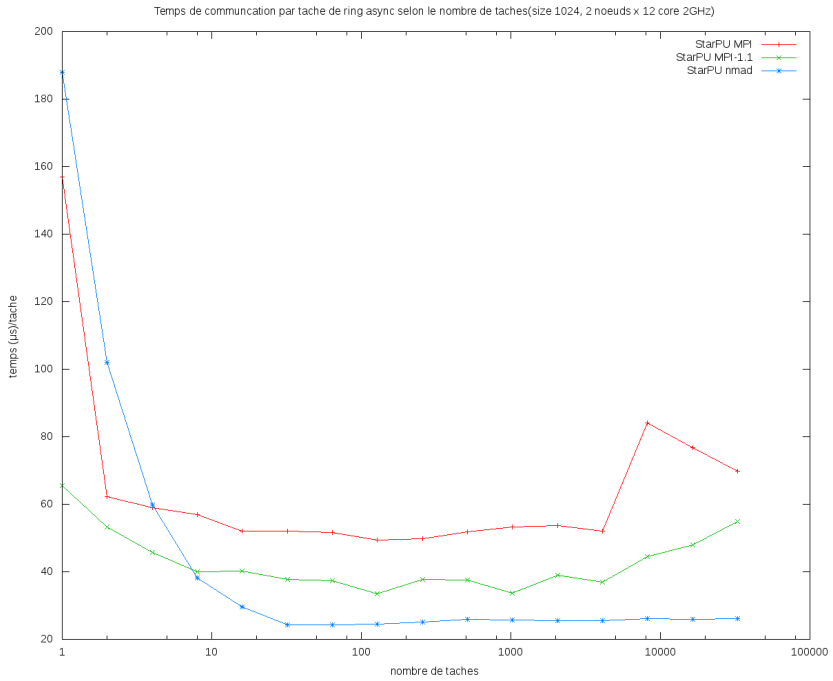

FIGURE 6 – Temps total de communication selon nombre de taches communicant

peut pas être seulement expliqué par la fusion des messages de petite taille par NewMadeleine.

5.4 Performance Cholesky

Afin de mesurer les performances de notre module dans le contexte d'une utilisation réelle par une application,nous avons mesuré les performances d'une factorisation cholesky distribuée (Fig 7) (factoriser une matrice de manière à trouver un facteur étant une matrice triangulaire inférieure) en faisant varier le nombre des blocs (plus la matrice est divisée en de nombreux blocs, plus fréquemment on doit les échanger entre nœuds, et moins les échanges sont de grande taille).

On note que pour la taille de bloc optimale, notre implémentation basée sur NewMadeleine a des performances 50% plus élevées que les autres. Ailleurs l'écart est inférieur voire négligeable. Nous supposons que pour le nombre optimal de blocs le recouvrement a un impact majeur sur les performances, alors que pour d'autres nombre de blocs, soit il n'y a pas assez de blocs, et donc lors d'une communication il n'y a pas de blocs déjà reçu permettant de la recouvrir, soit le temps passé à effectuer les communication devient trop im-

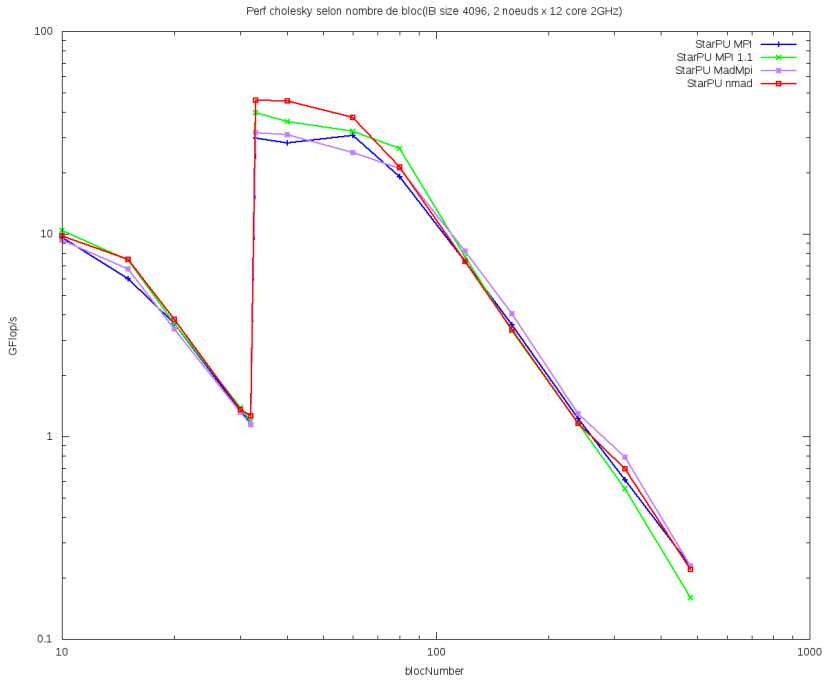

FIGURE 7 – Performance Cholesky selon la version de StarPU-MPI et la taille de bloc utilisée.

portant et donc le temps de calcul devient négligeable (et avec lui l'importance du recouvrement).

5.5 Un autre ordonnanceur StarPU : LWS

StarPU offre plusieurs ordonnanceurs dont :

— Eager : liste centrale des taches. Des qu'un thread est libre, il demande une tache.

— LWS : chaque support d'exécution a sa propre liste des taches. Les taches prêtes a être exécutée sont ajouté a la liste du support d'exécution qui l'a libérée. Si un support d'exécution n'as plus de taches prévue, il en vole aux autres.

Eager est l'ordonnanceur utilisé dans les autres performances. LWS pourrait être intéressant, en effet si une tache est libérée par une communication le serra généralement par le thread idle de NewMadeleine. Comme il s'exécute a une plus faible priorité que les calculs, si un cœur est inactif il est probable qu'il s'y exécute.

Nous avons donc tenté de changer l'ordonnancement de la factorisation précédente (Fig 8). Nous remarquons que pour un nombre extrêmement faible de blocs, LWS a des performances légèrement

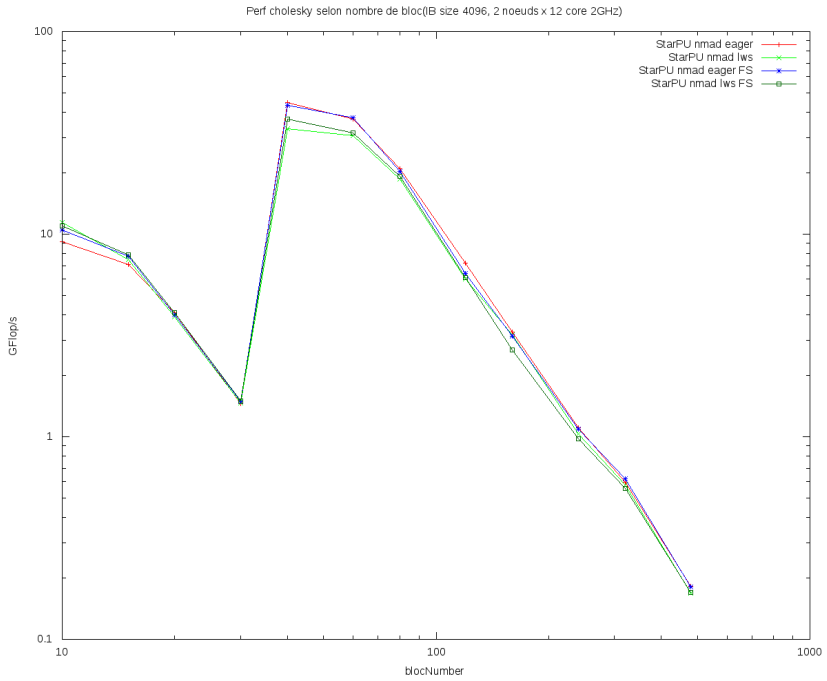

FIGURE 8 – Performance Cholesky en changeant ordonnancement StarPU et forant l'envoi des données par NewMadeleine.

meilleures (+10%) que eager, cependant pour un nombre optimal de blocs les performances sont nettement réduites (-25%) et les performances sont généralement amoindries. LWS est certes plus efficace si des threads sont inactifs par manque de tache, cependant ce n'est pas un cas souhaitable. Il est surtout plus coûteux dans la plupart des cas à cause des vols de tache.

5.6 Flush après envoi

Si l'efficacité du recouvrement semble nous permettre d'obtenir de bonnes performances, la latence de notre implémentation reste préoccupante. Forcer NewMadeleine a envoyer les messages dès que possible au lieu d'attendre pour le fusionner avec un autre pourrait la réduire. Nous avons donc tenter de flush les message après avoir posté un envoi (Fig 8).

Cela semble avoir un impact négligeable sur la latence des messages en une partie, et risque au contraire d'augmenter fortement la latence de ceux envoyés en plusieurs parties.

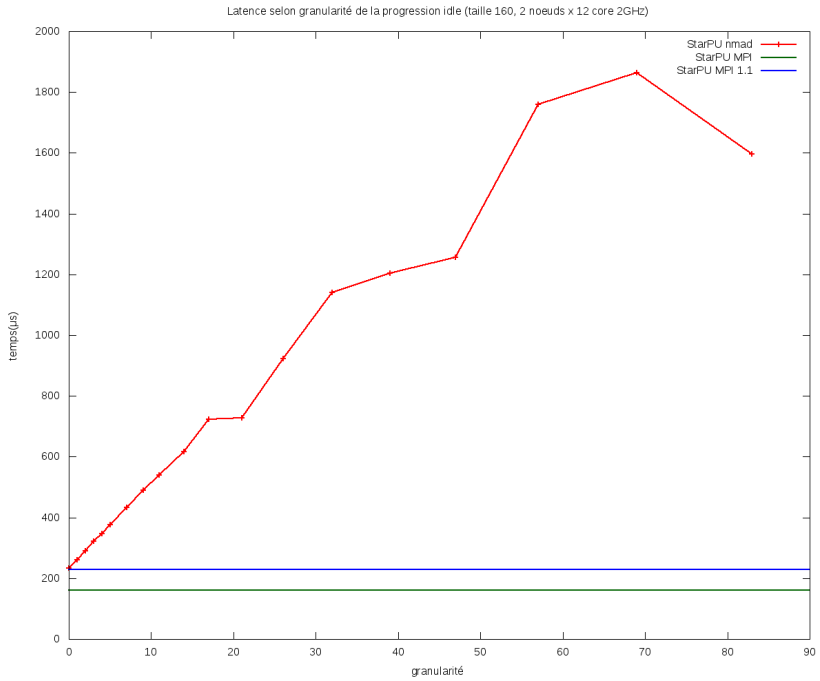

FIGURE 9 – Latence selon granularité de la progression idle par Pioman

5.7 Granularité

Effets de la granularité sur NewMadeleine

Notre module utilise principalement le thread idle de Pioman[6] pour assurer la progression des communications. La granularité représente l'intervalle minimal entre deux réveil de ce thread.

Il a une influence importante sur la latence, cependant sa granularité par default est censée être de $5 \sim 5\mu s$. Ainsi pour un envoi de message via le protocole eager, nous devons dans le pire cas payer deux fois cet intervalle. Cependant cela représenterait seulement un coût de 5-10 µs sur une latence de $\sim 300\mu s$..

Latence selon granularité

Nous avons donc mesuré l'impact réel de la granularité sur la latence entre deux nœuds (Fig 9). Il n'est pas surprenant que la latence soit croissante selon la granularité. Cependant le facteur de cette croissance est bien plus importante que prévue. Il semble que la granularité réelle soit de l'ordre de 30 fois celle demandée.

Suite à cela, Alexandre Denis a réalisé de nouveaux benchmarks permettant de tester l'effet sur la latence de NewMadeleine de se baser uniquement sur la progression idle de Pioman. Il semble que :

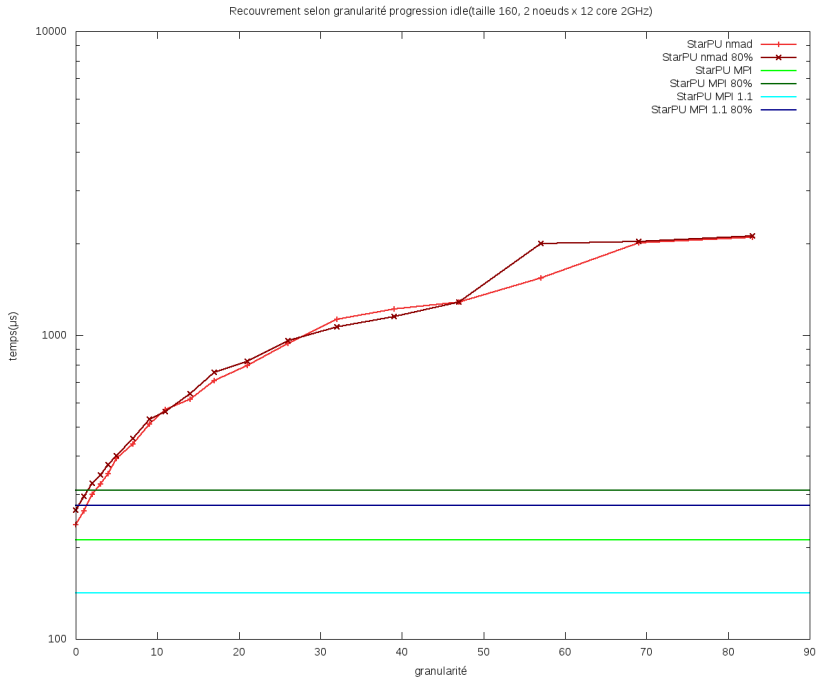

FIGURE 10 – Recouvrement selon la granularité

— Une fonctionnalité expérimentale augmentant la granularité du thread idle quand aucune communication n'as eu lieu récemment soit activée par default.
— La fonction réveillant le thread idle a un grain non négligeable qui augmente fortement la valeur réelle de la granularité.

Pour une très faible granularité la latence de notre module est comparable a celle du module utilisant IntelMPI.

Recouvrement selon granularité

Réduire la granularité permettrait certes d'améliorer la latence mais augmenter la fréquence des réveils du thread idle risque d'avoir un impact négatif sur le recouvrement. Nous avons donc fait une nouvelle mesure du recouvrement cette fois ci selon la granularité (Fig 10). Le rapport entre temps de calcul et de communication est certes légèrement meilleur pour un granularité comprise entre 5 et 10, qu'inférieure a 4, cependant la différence reste mineure. Étant donnée la différence de latence, il est probable que ce surcoût soit en pratique encore plus réduit, s'appliquant sur une période moins longue.

La granularité a certes un impact sur le recouvrement, il semble

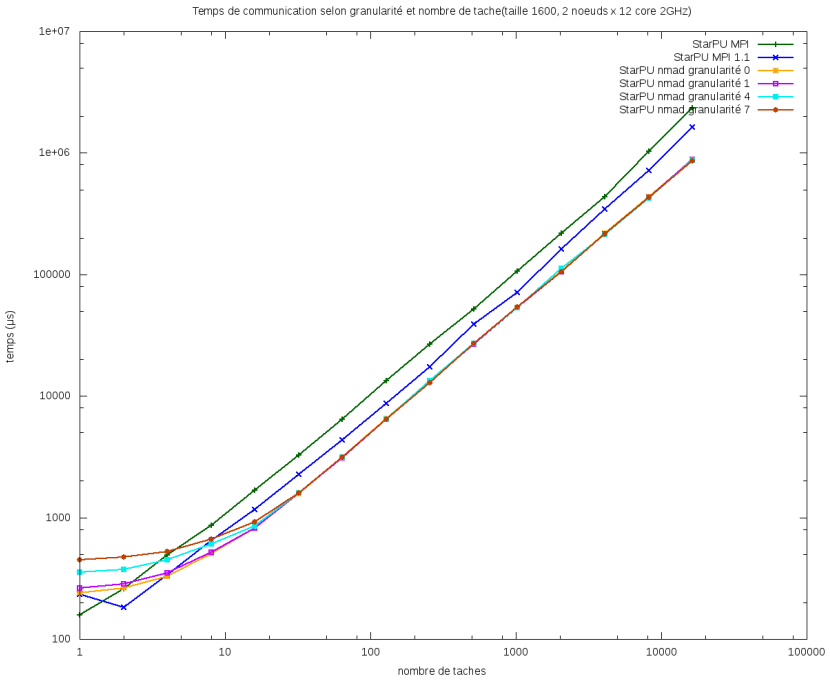

FIGURE 11 – Scalabilité en nb de requêtes selon la granularité

mineur et en particulier comparé à l'augmentation de la latence.

Scalabilité en nb requettes selon granularité

Un plus faible granularité semble donc préférable. Il convient néanmoins de tester l'impact sur la scalabilité en nombre de requettes(Fig 11), sa réduction augmentant la probabilité que l'ajout de nouvelles communications soit ralentit devant attendre que le thread idle s'endorme.

Pour un nombre important de communications simultanées, la différence de performance est négligeable, cependant un nombre faible de communication avantage une granularité extrêmement faible, la latence étant plus importante que l'intervalle d'envoi entre deux données. Un nombre important de requettes n'est donc pas problématique même avec une granularité extrêmement faible.

Cholesky et Granularité

Enfin nous souhaitons voir l'impact dans une utilisation réelle. Nous avons donc mesuré les performances de Cholesky selon la granularité(Fig 12).

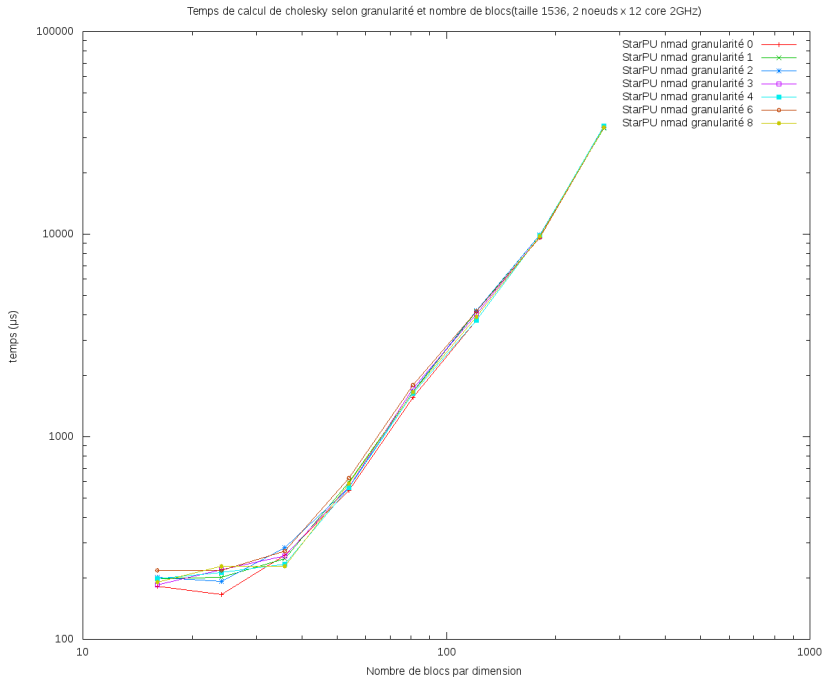

FIGURE 12 – Performances Cholesky selon la granularité.

Pour un grands nombre de blocs, les performances sont similaires. Cholesky effectue constamment des communications, un thread idle débutant plus de requêtes a chaque réveil. Pour un faible nombre de blocs (et leur nombre optimal), l'impact de la granularité sur la latence permet pour une granularité de réduire de jusqu'à 10% la durée du calcul par rapport a celle par default.

Une granularité extrêmement faible, en particulier 0 qui se contente de faire rendre la main au thread idle sans l'endormir, semble améliorer sensiblement les applications dépendantes de la latence(jusqu'à 10% pour la taille de blocs optimale). Le surcoût, en particulier causé par la contention avec le thread idle lors de la création de nouvelles requêtes, semble négligeable par rapport a celui de StarPU.

6 Conclusion

Notre implémentation StarPU-NMad remplie donc ses objectifs principaux, ayant une meilleure scalabilité pour un grands nombre de taches en attentes, le temps entre deux communications n'augmentant pas contrairement à l'implémentation basée sur StarPU-MPI et offrant un recouvrement jusqu'à 2 fois plus efficace. Elle offre également un bien meilleur recouvrement que les implémentations

basées sur MPI.

Sa latence, avec la granularité par default du thread idle de Pioman, est cependant notablement élevée tant que le nombre de communications en attente reste faible. En réduisant cette granularité, nous pouvons cependant atteindre une latence proche de celle de l'implémentation StarPU-MPI, sans autrement dégrader les performances de notre module de manière notable, StarPU causant déjà un surcoût important.

Des améliorations de note module sont envisageables comme rendre l'interface StarPU-MPI plus générique et la distinguer de son implémentation du même nom. Il serrait également utile d'évaluer l'impact sur les performances de réduire la granularité des réveils du thread idle sur d'autres applications que Cholesky, qui dépendraient moins de la latence.

Enfin il serrait intéressant de porter certaines fonctionnalités restant spécifiques au module StarPU-MPI comme intégrer SimGrid dans notre implémentation qui permet de simuler un cluster de grande taille.

Remerciements

Je tiens à remercier mon maître de stage Alexandre Denis de la confiance qu'il m'a accordé en me confiant ce travail.
Je remercie également Nathalie Furmento, Olivier Aumage, Emmanuel Agullo et Samuel Thibault qui ont aidé ma collaboration à StarPU.
Enfin, j'aimerais remercier les membres de TADaaM, STORM et HiePACS pour leur accueil.

Bibliographie

[1] Starpu : A unified runtime system for heterogeneous multicore architectures, Aout 2017. http ://starpu.gforge.inria.fr/.

[2] Chameleon : A dense linear algebra software for heterogeneous architectures, Aout 2017. https ://project.inria.fr/chameleon/.

[3] Cédric Augonet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu : A unified runtime system for heterogeneous multicore architectures, 2011.

[4] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Samuel Thibault, and Raymond Namyst. Starpu-mpi : Task programming over clusters of machines enhanced with accelerators. *Research Report RR-8538, INRIA*, Mai 2014.

[5] Alexandre Denis. Newmadeleine : An optimizing communication library for high-performance networks, Aout 2017. http ://pm2.gforge.inria.fr/newmadeleine/.

[6] Alexandre Denis. pioman : a pthread-based multithreaded communication engine. *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Mars 2015.

[7] Intel mpi library, Aout 2017. https ://software.intel.com/en-us/intel-mpi-library.