Évolution de MadMPI vers MPI-3 : Opérations Unilatérales

Clément Foyer

Bordeaux INP — ENSEIRB-MATMECA

Rapport de Projet de Fin d'Étude

Évolution de MadMPI vers MPI-3 : Opérations Unilatérales

Clément FOYER
Filière Informatique
Promo 2016

Tuteur : M. Alexandre DENIS
Chef d'équipe : M. Emmanuel JEANNOT

01 Février 2016 — 31 Juillet 2016

Remerciements

Je tiens tout particulièrement à remercier M. Alexandre DENIS, mon maître de stage, pour les échanges qui m'ont permis de progresser efficacement durant la réalisation de ce stage, pour son encadrement et sa confiance. Mes remerciements vont également à M. Emmanuel JEANNOT, chef de l'équipe-projet *TADaaM*, pour son accueil chaleureux, ainsi que l'INRIA Bordeaux – Sud-ouest pour m'avoir accepté en tant que stagiaire.

Je tiens aussi à remercier les membres des équipes *TADaaM* et *STORM* pour leurs conseils et explications qui m'ont permis d'avancer.

Je remercie aussi l'équipe pédagogique de l'ENSEIRB-MATECA pour son accompagnement durant les trois années passées au sein de cette école, et pour l'enseignement reçu qui m'a donné les connaissances nécessaires à la bonne réalisation de ce stage.

Table des matières

1 Introduction

Ce document présente le stage réalisé dans le cadre de mon projet de fin d'étude du cycle d'ingénieur. Cette mise en application consistait en l'enrichissement d'une bibliothèque de communications par l'ajout de fonctionnalités introduites dans la norme MPI-3.

Ce stage s'est déroulé dans le secteur de la recherche, du 1er février au 31 juillet 2016. Celui-ci m'a fait découvrir le métier d'ingénieur de recherche, et a confirmé ma volonté de travailler dans le domaine du calcul haute performance, ainsi que mon engouement pour le développement de bibliothèques logicielles.

Ce rapport cadrera tout d'abord mon travail. Celui-ci explicitera notamment le domaine d'application et exposera de façon concise le calcul haute performance ainsi que ses contraintes spécifiques. Puis, je présenterai les tâches réalisées et les résultats obtenus. Enfin, je résumerai les apports de cette étude et mettrai en exergue les nouvelles problématiques induites.

2 Cadre du stage : Entreprise et contexte

2.1 Présentation de l'entreprise

Le stage s'est déroulé au sein de l'équipe *TADaaM* (Topology-Aware System-Scale Data Management for High-Performance Computing) du laboratoire **INRIA Bordeaux – Sud-Ouest**.

Cet établissement public de recherche est dédié aux sciences du numérique. Son objectif est de fournir des solutions face à des problèmes informatiques et mathématiques dans des domaines tels que la santé, les transports, l'énergie, la robotique ou encore le calcul haute performance.

L'INRIA est organisé en "équipes-projets" qui rassemblent des chercheurs aux compétences complémentaires autour d'un projet scientifique défini, dans un temps donné. Je suis accueilli par l'équipe *TADaaM*, dirigé par Emmanuel. Elle se compose de quatre chercheurs, deux professeurs, sept doctorants, deux post-doctorants, deux ingénieurs et de deux assistantes administratives. Le projet de recherche ayant entraîné la création de cette équipe a pour but de créer une couche logicielle dynamique de service, étendue à l'ensemble du système, pour les environnements de calculs haute performance, afin d'optimiser les supports d'exécution selon les besoins de chaque application.

L'équipe *TADaaM* est née de la réorganisation de l'équipe Runtime, dont les chercheurs appartenaient à l'équipe SATANAS au sein du *LaBRI* (Laboratoire Bordelais de Recherche en Informatique). À la fin de son temps imparti, cette équipe fut scindée en deux, l'équipe *STORM* (STatic Optimizations, Runtime Methods), dont le domaine de recherche tourne autour des intergiciels et de l'abstraction du parallélisme appliqués au calcul haute performance, et l'équipe *TADaaM*.

2.2 Domaine d'application

2.2.1 Calcul haute performance

Le calcul haute performance (HPC) est devenu un outil essentiel de la recherche scientifique, technologique et industrielle. En effet, ce domaine de l'informatique permet de réaliser des simulations remplaçant les expériences qui ne peuvent être menées en laboratoire quand elles sont dangereuses (accidents), de longue durée (climatologie), inaccessibles (astrophysique), onéreuses (crash-tests d'avions) ou interdites (essais nucléaires). La simulation améliore également la productivité en procurant un gain de temps important.

Ces simulations mettent généralement en application de nombreuses équations physiques, lesquelles nécessitent des calculs intensifs pouvant demander d'importantes de ressources temporelles et mnémoniques. Aussi, ces applications numériques sont réalisées sur des supercalculateurs composés de nombreux ordinateurs mis en réseaux. Ainsi, les opérations sont effectuées en parallèle sur les différentes unités de calculs, permettant de répartir les besoins en mémoire pour chacune d'elle ; chaque ordinateur ne travaille que sur un fragment du problème. Par exemple, pour une simulation physique appliquée à un espace partitionné, abstrait

par un graphe, chaque unité ne calculera l'application numérique que sur un sous-graphe, appelé domaine de calcul.

2.2.2 Mémoire distribuée et mémoire partagée

La notion de parallélisme représente en informatique le fait d'avoir plusieurs fils d'exécution concurrents et indépendants [1]. Il existe en informatique deux principaux types de parallélisme : le parallélisme des instructions et le parallélisme des tâches. Le premier est une composante propre aux processeurs et représente la façon dont celui-ci va interpréter une séquence d'instructions. Le second constitue le fait de créer de multiples séquences d'instructions. Ce dernier nous intéressera au sein de ce rapport.

On peut rencontrer deux méthodes principales dans le parallélisme de tâches : la programmation parallèle à mémoire distribuée et la programmation parallèle à mémoire partagée. Bien que l'idée générale d'elles soit identique (avoir plusieurs fils d'exécution distincts), la mise en place est fondamentalement différente.

En effet, la première méthode peut-être implémentée très simplement. De fait, aujourd'hui les processeurs fondus présentent généralement plusieurs noyaux d'exécution, appelés *cœurs*, lesquels se divisent en fils d'exécution, matériels ou logiques, appelés *threads*. Il est ainsi possible d'avoir plusieurs *threads* concurrents, issus d'un même programme, s'exécutant en même temps sur un même processeur. Ils occupent le même espace d'adressage ; la mémoire est donc partagée. De plus, cette avancée s'apprécie en comparaison des précédentes générations de processeurs. Ceux-ci étaient constitués d'un seul cœur dans lequel l'exécution de multiples fils entraînait la nécessité de basculer de l'un à l'autre, induisant un fort surcoût dû aux sauvegardes d'états et aux changements de contextes. Outre la facilité de mise en place, un second avantage de la programmation *multithreadée* est de permettre de tirer profit des architectures multifilières des machines. La mémoire physique est ainsi partagée entre chaque *thread*, ce qui facilite l'échange de données.

La seconde méthode est constitue par un ensemble de programmes [2] s'exécutant sur différentes unités de calculs. Par ailleurs ces dernières ne nécessitent pas de se trouver au sein de la même machine ; les mémoires sont donc isolées : physiquement si les machines sont distinctes, logiquement si les exécutions s'effectuent sur une même machine. Cette méthode permet donc de réaliser des calculs sur un grand nombre de supports parallèles, et donc de multiplier par autant le nombre d'unités de calculs travaillant simultanément. Elle présente donc un grande capacité de mise à l'échelle. Il est ainsi possible d'effectuer des calculs sur des centaines de milliers de cœurs, voire sur des millions de cœurs, plus ou moins distants. Cette méthode nécessite donc la mise en place de structures résiliaires complexes qui permettent communications et synchronisations entre les différents processeurs effectuant les calculs. L'évolution permanente des technologies, qui offrent des réseaux toujours plus rapides, entraîne nécessairement un surcoût, qu'il s'agit d'atténuer.

Aujourd'hui, les solutions déployées dans le domaine du calcul parallèle sont généralement hybrides. Elles utilisent un grand nombre de machines, chacune permettant l'exécution de programmes multifilières. Les bibliothèques logicielles proposées doivent donc être adaptées pour profiter au maximum de l'ensemble des ressources mises à disposition. Celles qui prennent en charge les échanges de données via le réseau, doivent donc proposer des outils adaptés à ces configurations. Afin de ne pas séquentialiser les communications, et de ne pas perdre en performances, ces bibliothèques doivent être en mesure de gérer un multiplexage et démultiplexage des échanges. Ce peut-être, par exemple, une gestion multicanale des envois et réceptions de données [3], ou une mise en commun de ces services par un ensemble de processus légers dédiés aux communications. Par ailleurs, elles permettent de proposer des solutions coordonnées plutôt que compétitives aux utilisateurs. Ainsi, les bibliothèques sont généralement configurables, permettant une prise en charge de différents systèmes de communication, et s'adaptant aux exécutions multifilières.

2.3 *Message Passing Interface* et *Remote Memory Access*

Cependant, par la nature souvent itérative des calculs, deux problématiques se posent. Il s'agit d'abord de synchroniser ces opérations, afin que chaque pas d'itération soit réalisé au même moment pour assurer la

1. L'avancement d'un fil d'exécution ne modifie pas l'avancement des autres.
2. Les programmes peuvent être distincts ou identiques.
3. Échanges sur réseaux Ethernet, InfiniBand, Mirinet, segments de mémoire partagée, etc

correction des algorithmes. Ensuite, il faut pouvoir permettre aux différents cœurs de calculs d'échanger ou de communiquer leurs résultats, notamment aux frontières des domaines. C'est à ces problématiques que la norme *MPI*[11] (Message Passing Interface, interface d'échanges de messages) tente d'apporter des réponses.

Ce standard est très répandu pour la réalisation d'applications numériques sur des machines massivement parallèles, tant à mémoire partagée qu'à mémoire distribuée. La première version fut publiée en 1993, et fut issue de la participation de plus de quarante organisations. Depuis, la norme évolue, notamment par la participation en ligne via leur site internet[10].

L'objectif de ce standard est donc de définir des interfaces les plus simples et fonctionnelles possible, afin de faciliter la parallélisation des codes. *MPI* permet d'abstraire les communications réseau afin de faciliter la création de codes massivement parallèles. Cette interface permet, entre autre, une augmentation de l'autonomie des chercheurs non informaticiens quant à la réalisation de leurs calculs informatiques. La dernière version publiée de ce standard introduit aussi une interface d'abstraction de mémoire partagée, entre autres ajouts.

De plus, et de façon liée, dans la version 3 de ce standard [12] fut aussi introduite la notion de *RMA* (Remote Memory Access, accès mémoire distant [4]). Jusqu'à présent, les échanges d'informations entre les différents acteurs d'un calcul haute performance s'effectuaient par réception explicite. Ils nécessitaient de gérer une synchronisation de chacun des acteurs de la communication. Les cibles de ces opérations n'ont qu'à indiquer les origines potentielles, sans définir les séquences de communication. De même, les émetteurs n'ont plus nécessairement besoin de se synchroniser explicitement avec les récepteurs lors des phases d'échanges. Le support d'exécution *MPI* s'occupe de gérer les communications de façon transparente pour l'utilisateur et le programmeur. Les opérations *RMA* permettent donc de ne définir qu'une partie des acteurs.

Un programme *MPI* se présente sous la forme d'un code unique qui sera dupliqué pour chaque instance et exécuté sur différents nœuds. Chaque instance d'exécution du programme sera démarrée par un lanceur [5] qui s'occupe de les répartir sur les unités de calculs renseignées. Chacune peut-être différenciée par un identifiant unique, qui correspond à son *rang* [6] dans le communicateur global à l'application `MPI_COMM_WORLD`.

Parmi les structures définies dans le standard, il existe un certain nombre d'*objets opaques*, lesquels permettent une représentation au niveau utilisateur d'éléments constitutifs du cœur. Ainsi sont définis, parmi d'autres :

`MPI_Comm` qui représente une connexion entre nœuds

`MPI_Request` qui représente unilatéralement un échange de données [7]

`MPI_Datatype` qui représente un type de données

`MPI_Win` qui représente les interfaces de communications unilatérales [8]

`MPI_Info` qui permet une configuration plus fine des comportements des fonctions, définie à l'exécution.

Pour cela l'implémentation *MPI* doit être en mesure de détecter la réception d'un message inattendu, et d'y réagir. La réalisation de cette nouvelle fonctionnalité nécessite donc l'utilisation d'algorithmes plus proches de la programmation événementielle que de la programmation impérative stricte. Auparavant les précédentes fonctionnalités étaient plutôt régies par de la synchronisation entre les appels d'émissions et de réceptions. Bien qu'en pratique, ces synchronisations existent toujours, l'objectif est que celles-ci soient gérées de façon transparente pour le programmeur.

Le standard définit donc, de façon générale, les prototypes des fonctions accessibles, et décrit leur comportement attendu sachant les paramètres donnés. Ainsi sont aussi définis les erreurs et le caractère bloquant ou non des fonctions. Chaque version du standard prend généralement en compte une rétro-compatibilité, en laquelle il s'agit donc n'ajouter que de nouveaux symboles.

La notion de *fonction bloquante* est très importante dans le domaine du calcul haute performance, notamment à propos des communications. Ce caractère propre définit le fait qu'une fonction ne se termine et ne retourne qu'une fois son exécution achevée, et, dans le cas des communications, uniquement lors de l'échange de

4. À savoir lire (`get`) et écrire (`put`) dans la mémoire d'un processus distant à partir d'un autre processus
5. `mpiexec` [6]
6. une explicitation de la notion de rang sera faite dans la section 3.3
7. une communication est constituée de deux requêtes : une requête de réception et une requête d'émission.
8. En cela, elles peuvent être vues comme des encapsulations de `MPI_Comm`

données effectué. Les fonctions *non-bloquantes* effectuent le contrôle d'erreurs des paramètres, puis débutent la communication et enfin retournent sans attendre la complétion de celle-ci. En effet, il est souhaitable, et même préférable, de pouvoir effectuer des échanges en arrière-plan, alors que se déroulent des calculs. Bien que cela impacte les performances, les processeurs devant à la fois effectuer l'envoi (ou la réception) de données, en plus des opérations, le caractère *multithreadé* des exécutions peut permettre un impact minimal du mode non-bloquant sur les communications (voir figure 1).

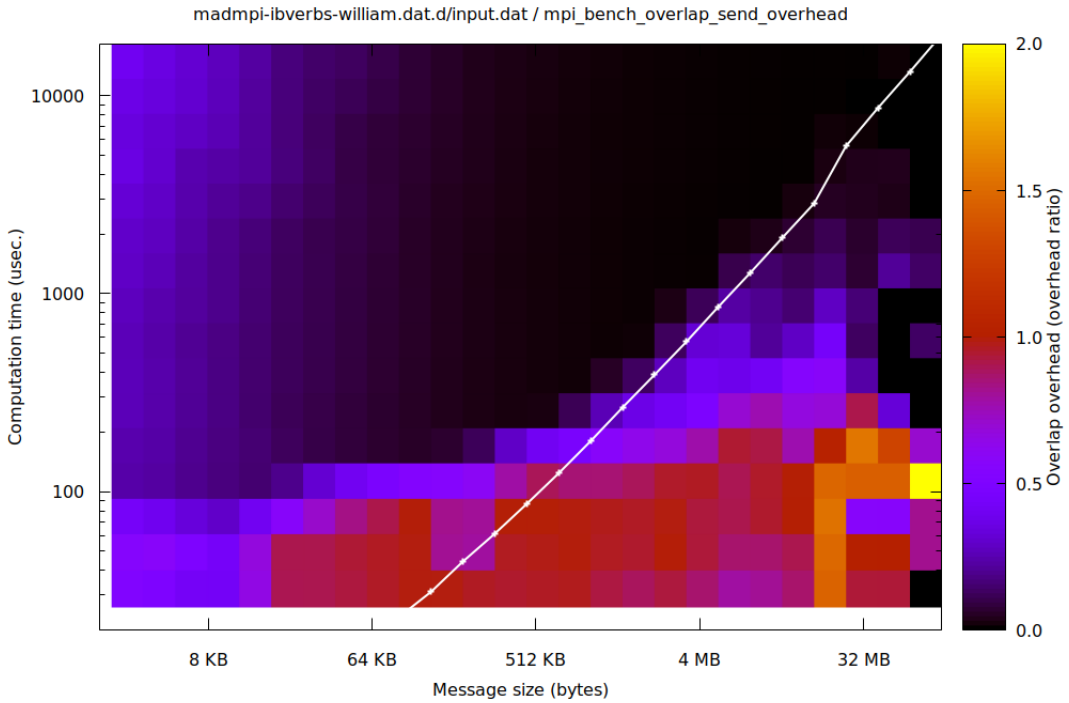


FIGURE 1 – Impact de la communication sur le temps de calcul[7]

Légende :
– 0 : Recouvrement parfait
– 1 : Pas de recouvrement : sérialisation des communications et des calculs
– 2 : Plus lent que la version sérialisée
La ligne blanche décrit la limite où le temps de calcul est égal au temps d'envoi des données.

2.4 Problématique

À partir de ces notions, il est donc possible de développer des bibliothèques respectant le standard *MPI*. Celles-ci peuvent néanmoins proposer des fonctions supplémentaires. Par ailleurs, la norme peut laisser un certain degré de liberté quant à certains comportements des fonctions, notamment dans des cas particuliers tels que le chevauchement de données dans les tampons de réception. Pour le standard *MPI*, les principales implémentations sont *MPICH*[8], *MVAPICH*[9] et *OpenMPI*[5]. Dans le cadre de ce stage, ma mission a été de participer au développement de la bibliothèque *NewMadeleine*[3], développée au sein de l'INRIA Bordeaux – Sud-ouest.

Ma tâche a donc été de développer, les différents mécanismes nécessaires aux opérations unilatérales en m'appuyant sur la norme définie dans le standard et sur les fonctions déjà présentes. La bibliothèque, codée en C, s'appuie sur différents systèmes de processus légers (*threads*) en utilisant pour cela *PIOMAN*[4, 2].

2.4.1 *NewMadeleine*

NewMadeleine est une bibliothèque de communications. Elle s'appuie sur *PIOMAN* qui est un *framework* de programmation générique. Il permet une progression asynchrone des communications en utilisant de façon

opportuniste les cœurs disponibles. Faisant usage les threads système, ce canevas est compatible avec tous les supports d'exécution multifilières.

Le code des stratégies d'optimisations de la bibliothèque est générique et portable. Il est paramétré à l'exécution par les capacités des pilotes réseau sous-jacents. La base de données des stratégies d'optimisations prédéfinies est facilement extensible. L'ordonnanceur est en outre capable de mixer de façon globalisée de multiples flux logiques sur une ou plusieurs cartes physiques, potentiellement de technologies différentes en multi-rail hétérogène[1].

Cette bibliothèque propose une implémentation du standard *MPI*, *Mad-MPI*. Celle-ci couvre l'ensemble de la version 1, la majorité de la version 2, et la couverture de la version 3 est en cours. Par ailleurs, elle supporte entièrement l'option `MPI_THREAD_MULTIPLE`, laquelle permet une intégration aisée des codes *multi-threadés*.

2.4.2 Asynchronisme

La développement de la bibliothèque s'appuyant sur un moteur asynchrone, il semblait donc tout à fait opportun d'ajouter les fonctionnalités RMA. En effet, ces progressions et interactions sont par nature asynchrones par le besoin d'abstraire au programmeur les mécanismes d'émission et de réception. En effet, si on se limitait à déclencher les échanges lors des appels de synchronisation, on perdrait toute l'utilité de ce mode de communication dans le cadre des calculs haute performance. Par ailleurs, les émissions de données ne pourraient être effectuées en utilisant le mode "passif" [9].

Par ailleurs, la bibliothèque présente nativement un mécanisme de gestion d'événements. Chaque communication entraîne l'appel d'une chaîne d'événements, auxquels il est possible d'enregistrer des fonctions de rappels. Il en existe deux types, et deux niveaux de surveillance/réaction de la part des *handlers* :

au niveau d'une session correspondant à l'ensemble des communications de l'application

au niveau d'un message permettant ainsi le suivi fin de l'évolution d'une communication unique

L'usage de ces deux types de *handlers* permet une gestion asynchrone des opérations. De plus, grâce à l'utilisation du moteur *PIOMan*, ces gestionnaires sont gérés de façon opportuniste sur les cœurs disponibles, ce qui permet de bonnes propriétés de recouvrement des communications par le calcul.

L'objectif est donc de pouvoir proposer une solution la plus portable possible, tout en respectant le standard. Il est cependant nécessaire de rester compatible avec d'autres solutions, notamment vis à vis des comportements attendus par les utilisateurs. Ainsi, la bibliothèque développée doit passer des suites de tests créées pour vérifier son fonctionnement, ainsi que celles fournies par l'implémentation *MVAPICH*, afin d'assurer une homogénéité dans les comportements attendus par les développeurs.

3 Travaux réalisés

La bibliothèque *NewMadeleine* est constituée de différentes interfaces, sur lesquelles s'appuyer afin d'ajouter de nouvelles fonctionnalités. Elles permettent d'abstraire le cœur applicatif, et une plus grande modularité. La bibliothèque *Mad-MPI* est ainsi implémentée par l'interface *mpi*. La principale utilisée lors de l'ajout des fonctionnalités liées aux opérations unilatérales fut l'interface *sendrecv*. Celle-ci propose une interface assez proche des fonctions d'envois et de réceptions telles que définies dans la norme MPI-1 bien que de plus bas niveau, les interactions avec le cœur de la bibliothèque étant directes. Cette interface définit aussi la gestion des événements, tant liés aux requêtes qu'aux sessions [10].

Le code développé au cours de ce stage se scinde en deux parties, chacune correspondant à un fichier source. Il a été recherché une grande indépendance entre la synchronisation et les opérations dans la programmation. Bien qu'il y ait une certaine intrication irréductible, les ponts entre ces deux fichiers sont minimaux.

9. Voir sous-section 3.3.3

10. Les requêtes correspondent à l'abstraction au niveau du cœur de la bibliothèque des `MPI_Request` et les sessions des `MPI_Comm`.

Dans cette section, j'exposerai mes contributions à la bibliothèque, en introduisant l'implémentation de l'objet d'interfaçage principal défini par le standard (`MPI_Win`), son initialisation et les structures connexes liées. Puis, je présenterai l'ajout qui a été nécessaire dans la gestion des types de données. Je décrirai ensuite les différents modes de synchronisation, leur fonctionnement et l'implémentation qui en a été faite. Enfin et pour conclure, je spécifierai les différentes opérations, notamment en les mettant en parallèle avec les réceptions explicites.

3.1 `MPI_Win`

Le standard définit donc la notion de *fenêtre*. Celle-ci se réfère à la fois à la zone mémoire accédée à distance, ainsi qu'à l'objet opaque d'interfaçage. Il s'agit d'un nouvel objet syntaxique qui permet de gérer les interactions entre les différentes unités de calculs. Celui-ci permet les échanges de données, en appui sur les `MPI_Comm`. Est également introduite la notion de *période*. Celle-ci représente l'ouverture de la fenêtre en *exposition* ou en *accès*. L'*exposition* correspond à la période durant laquelle des processus peuvent accéder à la mémoire d'un processus d'une fenêtre. L'*accès* est le laps de temps durant lequelle le processus possesseur d'une fenêtre donnée accède à la mémoire d'un processus distant. Il y a bien évidemment concordance entre période d'accès de l'*origine*[11] et période d'exposition de la *cible* d'une opération RMA. Puis sont définis deux modes de communication, à savoir le mode à "cible active" et le mode à "cible passive" (que je simplifierai en *mode actif* et *mode passif* dans la suite de ce rapport). La différence entre ces deux modes est le déclenchement de la période d'exposition ; dans le premier, celui-ci est explicite et synchrone, alors que dans le second il est implicite et géré de façon asynchrone par le moteur.

Quatre saveurs de `MPI_Win` sont définies : *MPI_WIN_FLAVOR_CREATE*, *MPI_WIN_FLAVOR_ALLOCATE*, *MPI_WIN_FLAVOR_DYNAMIC* et *MPI_WIN_FLAVOR_SHARED*. Les fenêtres sont donc des objets syntaxiques propres à chaque programme. Elles contiennent l'ensemble des informations nécessaires à la communication entre deux processus. Pour cela, elles échangent durant leur phase d'initialisation les adresses des zones de données, dimensions et unités de taille. Aussi est-il possible, à partir d'un processus donné, de calculer l'adresse mémoire distante à laquelle lire ou écrire. Ceci permet une gestion plus rapide des requêtes pour le processus en période d'exposition. Cela permet en outre de vérifier que les adresses accédées sont dans les limites de la fenêtre avant l'envoi des données. Les fenêtres possèdent également un ensemble de paramètres nécessaires à la vérification d'erreurs, à la synchronisation et à l'écoute d'événements. Par ailleurs, le comportement dans le cas où deux fenêtres partageraient une zone mémoire est non-spécifié.

Les différentes saveurs se distinguent de la façon suivante :

MPI_WIN_FLAVOR_CREATE est la fenêtre de base. Chaque information est fournie par l'utilisateur.

MPI_WIN_FLAVOR_ALLOCATE a me même conftionnement que la fenêtre de base. La seule différence est que la mémoire est allouée par le moteur. Cela permet de fournir de la mémoire issues de zones permettant un accès plus rapide au réseau : par exemple des segments pré-enregistrés pour une carte réseau.

MPI_WIN_FLAVOR_DYNAMIC est une fenêtre dont la mémoire peut être agrandie ou réduite au cours du temps. Elle permet une gestion dynamique de la mémoire attachée, similairement à l'utilisation du tas.

Ces opérations sont effectuées par les fonctions `MPI_Win_attach` et `MPI_Win_detach`. Cela permet, par exemple, la construction d'une liste chaînée distribuée sur l'ensemble des nœuds. Les zones mémoires attachées étant par nature dynamiques, ce type de fenêtre nécessite une gestion particulière des vérifications de dépassement mémoire. Cette gestion sera abordée dans la section 3.4.

MPI_WIN_FLAVOR_SHARED est une fenêtre dont la mémoire, allouée par le cœur, est un segment de mémoire partagée entre les différents nœuds du communicateur. Son initialisation nécessite le partage d'un nom unique de fichier permettant de créer le segment de mémoire partagée.

11. Pour les opérations RMA, on parle d'*origine* (ou de *source*) et de *cible* plutôt que d'émetteur et de récepteur, car ces échanges peuvent entraîner la nécessité de l'envoi de messages de la part de chacun des acteurs de ces opérations. Ainsi, l'initiateur d'une requête RMA est la source, et le destinataire la cible.

Dans un souci d'efficacité, ainsi que pour chaque type d'objet opaque, les fenêtres sont allouées par des allocateurs spécifiques typés définis dans la bibliothèque. Ce sont des *allocateurs en tranches*, ou *slab allocator*. C'est à dire que l'allocation se fait par l'attribution d'une zone mémoire de taille égale à plusieurs instances de l'objet. Ainsi, la première allocation, la plus coûteuse, alloue plusieurs objets en une fois. Ceci permet d'amortir le coût de l'appel système car les objets suivants n'ont qu'à trouver le prochain objet pré-alloué. Ainsi, le coût des objets suivants de même type ne sera que de quelques cycles. Chacun possède un identifiant unique tout au long de son cycle de vie, au sein de son instance d'exécution. Les objets ainsi alloués, une fois libérés, peuvent être réutilisés. Pour les fenêtres, ces identifiants sont échangés durant la phase d'initialisation.

3.2 Gestion des types de données

Lors des communications synchrones à réception explicite, les données échangées sont définies par les objets opaques *MPI_Datatype*. Ces structures permettent de décrire l'agencement mémoire des données. Elles servent tant au regroupement des octets lors de l'envoi qu'à leur répartition lors du déballage dans le tampon de réception.

NewMadeleine décrit ces données d'une manière proche de la programmation fonctionnelle. En effet, elle utilise des foncteurs itérants sur les blocs consécutifs de données afin de permettre un parcours de celles-ci. Ce mécanisme permet leur emballage, lors de l'envoi, qui ne nécessite pas de copie de l'ensemble des données dans un tampon intermédiaire. De même, lors de la réception d'un message, les types de données permettent de déballer directement les données par blocs de données consécutives sans avoir à passer par un tampon intermédiaire. Chaque *datatype*[12] connaît sa dimension et le décalage nécessaire pour accéder à la donnée suivante. Un parcours d'un type de données à partir d'une adresse spécifiée entraîne donc le parcours des données.

Cependant, dans le cadre des opérations unilatérales, seul le processus source connaît l'ensemble des données, parmi lesquelles le type à l'envoi, le nombre de données de ce type à l'envoi, le type à la réception et le nombre de données de ce type à la réception. N'ayant pas la possibilité de connaître si le type utilisé lors de la réception est défini chez la cible, il a été nécessaire d'établir un protocole permettant de communiquer ce *datatype* depuis l'origine vers la cible.

Pour cela, la définition d'une version sérialisée du type de données s'est imposée, accompagnée d'une mise en place d'un protocole de cohérence de cache. Il en découle le besoin de définir un attribut permettant une reconnaissance unique des types à travers l'ensemble de l'application. Pour cela, un *hash* du type de données est calculé. Il dépend du type de combinaison[13], du nombre d'éléments, de la taille et des paramètres spécifiques à chaque combinaison. Afin de pouvoir réaliser une correspondance entre *hash* et *datatype*, une table de hachage a donc été mise en place.

Bien que ces modifications aient été nécessaires dans le cadre des opérations unilatérales, ces mécanismes sont entièrement indépendants de la synchronisation ou de l'application d'opération RMA.

3.2.1 Sérialisation et désérialisation des types

Les types de données se construisent à partir d'un ensemble de types prédéfinis par le standard, et d'un ensemble de fonctions permettant de combiner ces types afin de décrire les données. En interne, les types sont construits de telle sorte qu'un type combiné à partir d'autre types (qui seront ici appelés sous-types) possède un pointeur vers ceux-ci. De plus les paramètres communiqués au *datatype* lors de l'appel de la fonction de combinaison sont aussi conservés. La taille et l'extension des données sont aussi calculées à la volée et conservées dans la structure du *datatype*.

Le parcours du *datatype*, dans le cadre du parcours des données, s'effectue donc par des fonctions itératrices, parcourant types et sous-types, en utilisant l'extension et les tailles de données.

De même, la sérialisation suit le même processus, à cela près que le parcours de ces types se fait en introspection. Chaque type, lors de sa création est donc sérialisé. Il s'agit, pour cela, de linéariser dans un tableau

12. Au cours de ce document, il sera fait indistinctement référence aux termes *datatype* et *type de données*
13. Les types prédéfinis possèdent le type de combinaison *MPI_COMBINER_NAMED*.

de données l'ensemble des paramètres communiqués à la fonction de combinaison. La sérialisation d'un type nécessite donc la sérialisation du sous-type correspondant. Pour cela, les sous-types ne sont sérialisés qu'une seule fois, et préfixent la version sérialisée des types de données qui en dépendent. Ainsi, lors de la désérialisation, le processus de recréation de ces types peut-être effectué en parcourant linéairement les données reçues, chaque type étant recréé chez la cible dans le même ordre que lors de leur création chez le processus source. Les types de données prédéfinis ne sont pas sérialisés. Ils sont identifiés par leur *hash* unique.

3.2.2 Protocole de cohérence de caches distribués

Les *datatypes* pouvant maintenant être échangés entre différents processus, il a donc été nécessaire de mettre en place un cache des types échangés. Celui-ci fut implémenté à l'aide d'une seconde table de hachage, indépendante de celle utilisée pour la correspondance *hash*/*datatype*. Les entrées de cette seconde table de hachage sont fonction d'un type et d'un processus. En effet, les processus peuvent être identifiés de façon unique dans la bibliothèque, la table est donc partagée entre l'ensemble des fenêtres.

Les entrées dans cette table peuvent être effectués de deux façons différentes. D'abord, lors de l'envoi d'un message (i.e. lors d'une opération RMA, voir section 3.4), mais aussi lors de la réception d'une demande d'opération. Ce second mode d'ajout à la bibliothèque permet de limiter le nombre d'échanges de types. Par ailleurs, chaque réception d'un nouveau type entraîne son ajout dans la table de hachage de correspondance *hash*/type. Si le *hash* est déjà présent dans la table, on considère alors que le type existe déjà du côté cible, et le type n'est pas remplacé. Cela permet de ne pas provoquer d'erreur pour un type en cours d'utilisation, ou en attente d'utilisation qui serait libéré trop tôt.

3.3 Communications

Les communications sont donc le cœur du fonctionnement de la bibliothèque. On les sépare en deux groupes distincts : les communications point-à-point, et les communications collectives. Les premières sont naturellement des communications entre deux nœuds, les secondes des communications auxquelles participent l'ensemble des nœuds d'un communicateur. Chaque nœud est identifié de façon unique dans un communicateur par son *rang*. Ainsi, toute communication point-à-point est définie comme étant effectuée entre deux nœuds, identifiés par leur rang. Le standard autorise, dans un souci de simplification syntaxique, les échanges entre un nœud et lui-même, auquel cas, le rang du nœud émetteur sera égal au rang du nœud récepteur. Un même processus peut appartenir à différents communicateurs ; il possédera un rang dans chacun. Les rangs au sein d'un communicateur étant toujours consécutifs et leur numérotation commençant toujours à zéro, un processus peut avoir des rangs différents selon que l'on considère son rang dans un communicateur ou dans un autre. Par ailleurs, chaque fenêtre étant définie sur un communicateur unique, le rang d'un processus dans ce communicateur sera aussi le rang de ce processus dans la fenêtre.

3.3.1 Réception explicite

Le cas général de communication est donc la communication à réception explicite. Le standard définit la clé d'identification unique d'un message au sein d'un communicateur comme étant facteur de quatre paramètres : le rang de l'émetteur, le rang du récepteur, la taille des données et le *tag* du message. La bibliothèque assure donc que pour deux messages concurrents [14], si la combinaison de ces paramètres est différente, alors les messages ne seront pas permutés. Le tag étant défini par l'utilisateur, il lui appartient de veiller à l'unicité de ces critères entre plusieurs messages concurrents. Par ailleurs, les tags sont définis dans le standard comme des entiers positifs codés sur 32 bits. Ils sont cependant encodés sur des entiers signés. Ceux-ci sont donc compris entre 0 et $2^{31} - 1$. La bibliothèque utilise en outre les 2^{31} valeurs négatives restantes afin d'identifier de façon unique les messages échangés dans le cadre d'appels de synchronisation, ou de communications collectives.

Dans ce mode de communication, le processus émetteur indique donc un *buffer* d'envoi, le communicateur utilisé pour cet échange, le rang du processus visé, le type des données envoyées, le nombre d'éléments de ce

14. Deux messages sont concurrents lorsque l'envoi du second message débute avant la complétion de l'envoi du premier.

type, ainsi qu'un tag. Le processus récepteur donne un *buffer* de réception, le même communicateur, le rang du processus émetteur, le type de données reçues, le nombre d'éléments de ce type, et le même tag que le processus émetteur. La quantité de données envoyées doit correspondre à la quantité de données reçues. Pour un processus P_1, envoyant n_1 données de type T_1 et pour un processus P_2, recevant n_2 données de type T_2, on doit avoir $n_1 \times \text{taille}(T_1) = n_2 \times \text{taille}(T_2)$.

3.3.2 RMA

Le cas des messages échangés dans le cadre d'opérations mémoire distante est donc celui qui devait être ajouté dans la bibliothèque à l'issue de ce stage. Celui-ci diffère du mode à réception explicite en deux points. Tout d'abord, son usage n'est possible que dans le cadre d'une période d'exposition. Ensuite, l'absence d'opération de la part du processus cible dans le cadre de cette communication implique pour le processus source de définir l'ensemble des paramètres de l'opération. Ces opérations sont uniquement des opérations point-à-point. Il n'y a pas d'opérations mémoire distante collectives de définies dans le standard.

Ainsi, et pour faire le parallèle avec la réception explicite, c'est le processus source qui doit définir le nombre de données envoyées, le type des données envoyées, le nombre de données reçues, le type des données reçues [15], le processus cible, et l'opération effectuée s'il y a lieu d'en avoir une. Le tag du message est, quant à lui, généré par la bibliothèque.

La génération du tag répond d'ailleurs à des règles spécifiques déterminantes dans le cadre qui nous intéresse ici. En effet, les moniteurs d'événements positionnés sur les sessions réagissent vis à vis des tags des messages. La construction de ceux-ci est présentée en figure 2, sachant que les tags tels que présentés dans la section 3.3.1, sont en réalité utilisés sur 64 bits au sein de la bibliothèque, grâce à une injection des tags 32 bits sur les tags 64 bits.

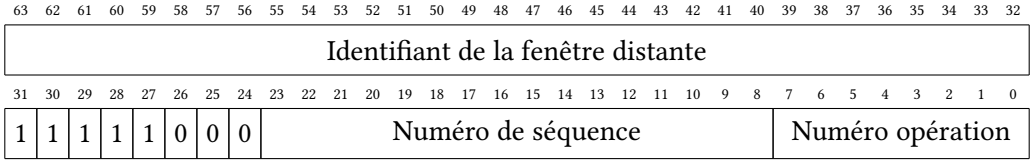


FIGURE 2 – Description champ par champ du tag pour les messages RMA

Les bits 28 à 31 définissent le fait que ce message a un tag généré par la bibliothèque, et donc qu'il s'agit ou bien d'une opération collective, d'une opération mémoire distante ou d'un message de synchronisation. Le bit 27 définit qu'il s'agit d'une opération mémoire. Le bit 26 définit qu'il s'agit d'un message de synchronisation. Dans le cadre des opérations RMA, ce bit est toujours à 0. Les bits 24 et 25 sont toujours à 0. Le numéro de séquence est un numéro de séquence localement unique [16] défini par fenêtre. Le numéro d'opération est défini en fonction du type d'opération mémoire effectuée, ainsi que du paramètre *MPI_Op* donné par l'utilisateur. Le détail des codes d'opération sera donné dans la section 3.4.

L'identifiant de fenêtre distante ainsi que les bits 31 à 26 du tag des messages permettent donc de définir de façon unique la fenêtre destinataire d'un message, ainsi que son objectif (opération ou synchronisation). Afin de faciliter la gestion de ces deux objectifs, ces deux types de messages sont gérés par des moniteurs d'événements différents. Chaque fenêtre définit donc deux moniteurs d'événements sur la session (correspondant au *MPI_Comm*) à laquelle elles appartiennent. Ces moniteurs d'événements sont donc à l'écoute de l'arrivée d'un message *inattendu* [17], ayant un tag dont les 32 bits de poids fort correspondent au numéro de fenêtre, puis les 6 bits suivants sont 111110 ou 111101.

15. Le nombre de données envoyées et reçues peuvent différer tant que la règle d'égalité des quantités de données envoyées et reçues est respectée.

16. Unique localement dans le temps. Ce numéro étant sur 16 bits, le compteur réutilise les numéros de séquence tous les 65536 opérations.

17. Les message sont dits *inattendus* lorsqu'il n'y a pas eu de réception explicite de postée.

3.3.3 Synchronisation

La seconde tâche de la fenêtre, outre la gestion du passage de messages, est donc la gestion de la synchronisation entre les processus. Cette gestion des accès distants à la mémoire s'effectue donc par le biais des périodes d'*exposition* et d'*accès*. Le processus de synchronisation diffère totalement selon que l'on soit durant un mode actif ou un mode passif. Cependant, que la période d'accès soit active ou passive, que l'accès soit exclusif ou partagé, les fonctions d'opérations se doivent de conserver un comportement unique. Par ailleurs, bien que cela ne soit pas recommandé par le standard, il est autorisé d'utiliser une même fenêtre pour créer des périodes tant actives que passives, dans la mesure ou il n'y a pas de recoupement. Une fenêtre ne peut être ouverte à la fois en mode actif et en mode passif. Le basculement d'un mode à l'autre ne peut s'effectuer qu'en passant par le mode "inutilisée" (voir figure 3).

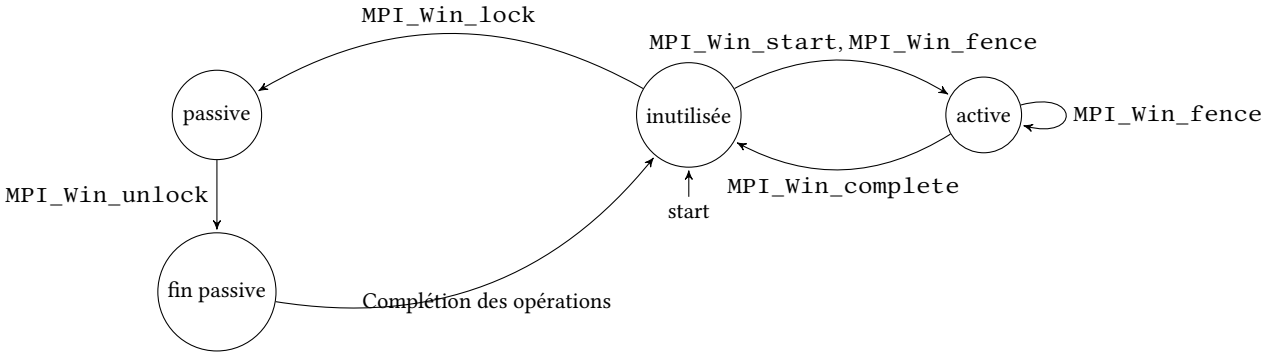


FIGURE 3 – États et transitions d'états d'une fenêtre en période d'exposition

Outre les messages de synchronisation demandant une fermeture de la fenêtre, une période d'exposition se termine lorsque l'ensemble des opérations effectuées par la source est terminé. Afin de savoir si l'ensemble des opérations a été effectué, notamment dans le cas d'une exécution multifilière, lors de l'envoi du message de fin d'accès, le processus source envoie le nombre de messages ayant dû être échangés. Selon les opérations, il peut s'agir d'un ou de deux messages par opérations [18]. La fenêtre conserve donc en interne un compteur du nombre de messages reçus, et un compteur du nombre de messages traités. Lorsque le nombre de messages traités est égal au nombre de messages reçus, et que le nombre de messages échangés est égal au nombre de messages requis par la source, alors la période peut-être close.

3.3.3.1 Mode Actif

Le mode actif est le mode le plus simple à définir, et le plus simple à faire fonctionner. En effet, dans ce mode, les périodes d'exposition et d'accès sont explicitées par les processus participant aux communications. Le cas général est celui impliquant les fonctions `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete` et `MPI_Win_wait`.

3.3.3.1.1 `MPI_Win_post`,`MPI_Win_start`,`MPI_Win_complete`,`MPI_Win_wait`

À l'inverse de `MPI_Win_fence`, ces fonctions, permettent donc de définir précisément pour un processus donné quels processus vont pouvoir venir lire (lors d'une période d'exposition), et à quels processus il pourra accéder (lors d'une période d'accès).

Les messages de synchronisation ne sont pas numérotés par les numéros de séquence. La gestion des périodes est telle qu'il ne soit pas possible d'avoir deux périodes successives concurrentes. En effet, une période d'accès ne peut débuter tant que la période d'exposition correspondante n'a pas débutée. De plus, il n'est pas possible de sortir d'une période d'accès tant que les opérations ne sont pas toutes terminées du côté de la source. Cela nécessite une synchronisation à base de messages échangés de pair à pair entre les processus impliqués dans

18. Dans le cas des fenêtre de type *MPI_WIN_FLAVOR_DYNAMIC*, le nombre de messages échangés peut monter jusqu'à trois à cause du message de validation (voir section 3.4)

ces communications et d'une attente de réception de l'ensemble des messages nécessaires. Le processus est décrit dans la figure 4.

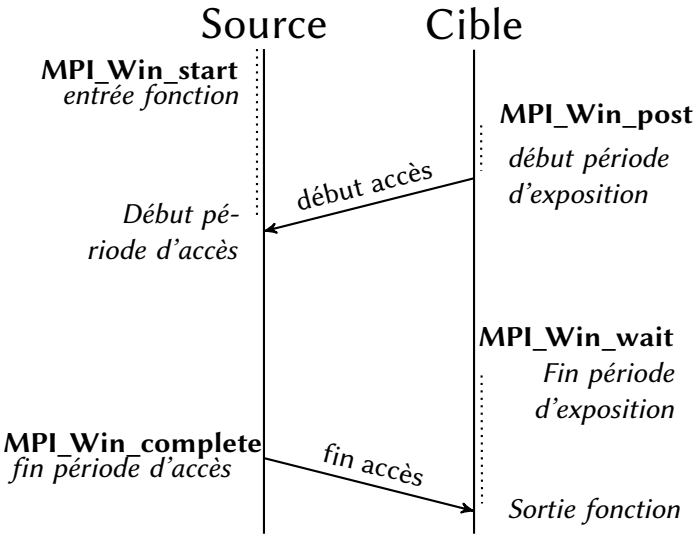


FIGURE 4 – Diagramme de séquence du cycle de vie d'une période avec synchronisation active

L'absence de nécessité d'accusé de réception lors de la fermeture de la fenêtre est due au fait que la période suivante entre ces deux processus ne pourra commencer tant que la cible ne sera pas prête, et donc tant qu'elle n'aura pas terminé la précédente période d'exposition.

3.3.3.1.2 `MPI_Win_fence`

Cette fonction permet d'ouvrir à la fois une fenêtre en accès et en exposition sur l'ensemble des processus de la fenêtre donnée. En outre, cette fonction peut-être utilisée aussi comme une barrière de synchronisation. Cette synchronisation effectuera d'ailleurs une synchronisation de la mémoire, tant au niveau du processeur, qu'au niveau des segments de mémoire partagée. Par ailleurs, cette fonction marque aussi une barrière en ce que chaque opération débutée lors de la période d'accès sera terminée et son résultat visible dans la mémoire de la cible à l'issue de l'appel à `MPI_Win_fence`.

Enfin, cette fonction accepte comme argument des assertions permettant d'optimiser son exécution dans des cas bien précis. Il est à noter que bien qu'il ne soit pas obligatoire pour une implémentation de les prendre en compte, c'est une erreur de donner des assertions fausses. Par ailleurs, ces assertions sont effectivement prises en compte dans la bibliothèque *NewMadeleine*.

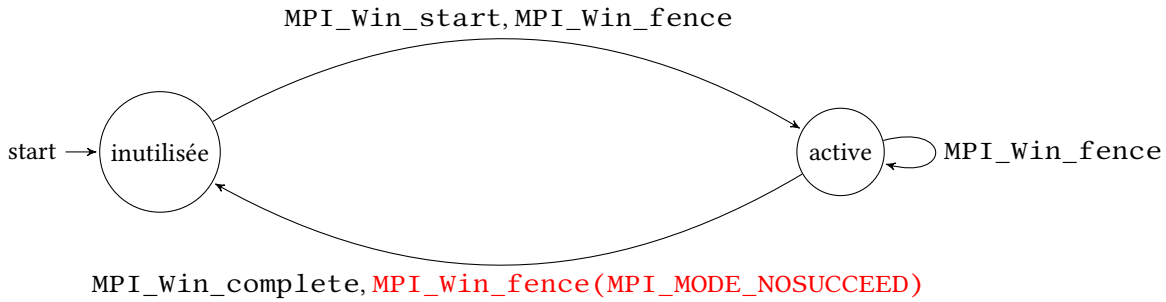


FIGURE 5 – Transitions d'états d'une fenêtre via `MPI_Win_fence`

En outre, l'assertion *MPI_MODE_NOSUCCEED* permet de terminer effectivement les périodes d'exposition et d'accès. L'automate d'état présenté à la figure 3 pourrait donc être précisé tel que montré dans la figure 5.

3.3.3.2 Mode Passif

Le mode passif est assez similaire au mode actif *post-start-complete-wait*, en ce qu'il y a toujours un échange de messages entre la source et la cible pour démarrer une période d'accès (voir figure 6) et par les conditions de terminaison qui sont identiques. Cependant, la période d'exposition est implicite car c'est la bibliothèque qui donne les droits d'accès aux processus distants. De plus, il est spécifié que la commande `MPI_Win_lock`, contrairement à son nom, est *non-bloquante*, exceptée lorsque l'appel est effectué d'un processus vers lui-même. De plus, les messages de synchronisation du mode passif nécessite naturellement d'être gérés par le moniteur idoine de la fenêtre cible. Pour cela, le tag, généré lui aussi, est formé tel que montré dans la figure 7.

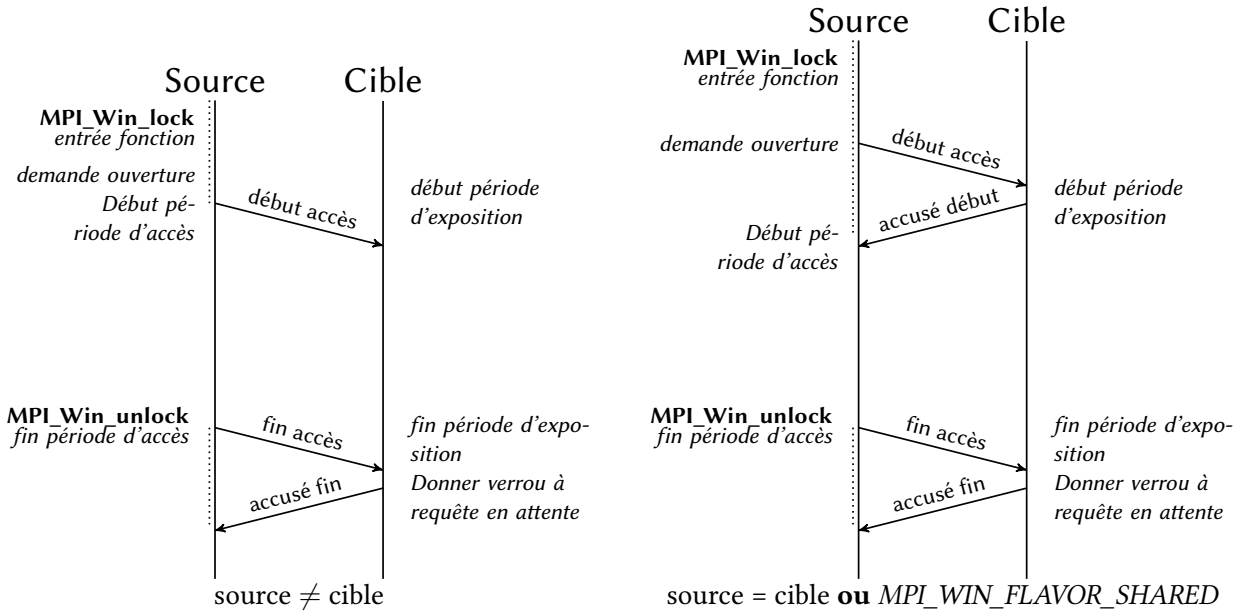


FIGURE 6 – Diagramme de séquence du cycle de vie d'une période avec synchronisation passive

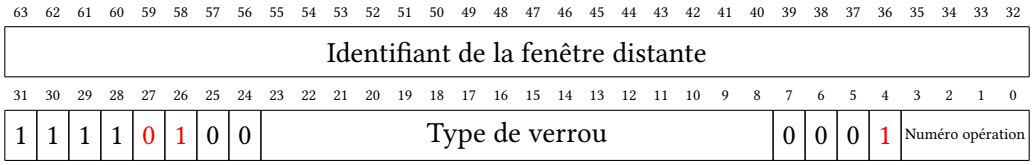


FIGURE 7 – Description champ par champ du tag pour les messages de synchronisation

On peut remarquer trois bits particuliers. Tout d'abord, les bits 26 et 27 qui échangent leur valeur (le moniteur d'opérations prenant les messages avec le bit 27 à 1, et le moniteur de synchronisations qui attend le bit 26 à 1). Ensuite, le bit 4 qui est nécessairement à 1. En effet, afin de séparer les messages de synchronisation et les messages d'opération, la signature des tags a été modifiée, entraînant l'exclusion mutuelle des bits 26 et 27. Cependant, suite à l'évolution du cœur de la bibliothèque, il a été nécessaire de définir un bit supplémentaire pour activer le moniteur de synchronisation. En effet, si un message était envoyé alors que la réception n'était pas postée, celui-ci était traité comme un message inattendu et appelait le moniteur. Or, si un moniteur était activé, la correspondance entre le message en attente et le postage correspondant ne pouvait plus être fait lorsque la réception était postée. Il a donc fallut modifier le tag de sorte que seule la demande de verrou appelle le moniteur. Enfin, le champ du numéro d'opération a diminué aussi, cependant le nombre de bits restait suffisant pour encoder les différents types de messages de synchronisation.

La gestion des périodes d'exposition a nécessité beaucoup d'attention, de précision et de délicatesse dans sa mise en place. En effet, il s'est agit de gérer les accès à la mémoire chez la cible de telle sorte que l'on puisse accéder à la ressource critique comme si elle était protégée par un sémaphore lecteur/écrivain. L'algorithme

mis en place est le celui présenté par l'algorithme 1.

```
   Entrée : requête
 1 verrou_source ← type_de_verrou(requête);
 2 if verrou_cible = LIBRE ou (verrou_cible = PARTAGÉ et verrou_source = PARTAGÉ et cmpt_exclusif
     = 0) then
 3 |   verrou_cible ← verrou_source ;
 4 |   cmpt_courant ← cmpt_courant + 1;
 5 |   Activer(requête);
 6 else
 7 |   Enfiler(file_attente,requête);
 8 |   if verrou_source = EXCLUSIF then
 9 |   |   cmpt_exclusif ← cmpt_exclusif + 1;
10 |   end
11 end
```

Algorithme 1 : Algorithme de gestion d'ouverture de période d'exposition

Par ailleurs, l'algorithme 2 présente le fonctionnement du verrou lors de la fermeture effective d'une période d'exposition. Cet algorithme est donc un sémaphore écrivain/lecteur, avec préférence pour le lecteur. Cependant, afin d'éviter les cas de famine, à partir du moment où un écrivain arrive, les verrous seront donnés par ordre d'arrivée.

```
 1 // La fonction Test_verrou_tete() retourne le type de verrou du
      premier élément de la file
 2 verrou_cible ← LIBRE ;
 3 cmpt_courant ← cmpt_courant- 1;
 4 if cmpt_courant = 0 et non(Est_vide(file_attente)) then
 5 |   requête ← Défiler(file_attente) ;
 6 |   verrou_source ← Type_de_verrou(requête);
 7 |   if verrou_source = EXCLUSIF then cmpt_exclusif ← cmpt_exclusif- 1;
 8 |   verrou_cible ← verrou_source ;
 9 |   cmpt_courant ← cmpt_courant + 1;
10 |   Activer(requête);
11 |   if verrou_cible = PARTAGÉ then
12 |   |   while Test_verrou_tete(file_attente) = PARTAGÉ do
13 |   |   |   requête ← Défiler(file_attente) ;
14 |   |   |   cmpt_courant ← cmpt_courant + 1;
15 |   |   |   Activer(requête);
16 |   |   end
17 |   end
18 end
```

Algorithme 2 : Algorithme de gestion du passage de verrou

L'ensemble des additions, soustractions et comparaisons des algorithmes 1 et 2 est réalisé grâce aux fonctions prédéfinies du compilateur, permettant une exécution atomique de ces opérations. Par ailleurs, les accès à la file sont aussi protégés par des barrières d'exclusion mutuelle.

L'appel à la procédure décrite dans l'algorithme 2 se produit donc lors de la terminaison d'une période d'exposition. Celle-ci peut-être déclenchée par l'arrivée du message envoyé par `MPI_Win_unlock`, ou, plus généralement, par les fonctions de rappel liées aux requêtes et à leur complétion (voir section 3.4).

Enfin, une gestion toute particulière des opérations a été nécessaire dans le cas des fenêtres à mémoire partagée. En effet, les opérations n'étant pas exécutées par la cible, mais uniquement par la source, cette spécificité nécessite de shunter le compteur du nombre de messages échangés. En outre, c'est uniquement la source qui se doit d'attendre la complétion des opérations.

Cependant, le cas des fenêtres de type *MPI_WIN_FLAVOR_SHARED* pose un second problème. En effet, le comportement visible des fonctions se doit d'être constant, peut importe le type de fenêtre. Ainsi, il fallait que la fonction `MPI_Win_lock` ne soit bloquante que lorsqu'elle est appelée sur soi-même, et que le caractère non-bloquant des opérations *RMA* soit conservé de la même façon. Pour cela, il a été nécessaire d'ajouter un nouvel état, côté source, à l'automate décrit à la figure 3. Par ailleurs, l'état "attente passive" peut être atteint directement depuis l'état "active", lorsque la période est vide de toute opération. Ce cas se présente si une période est débutée et terminée avec `MPI_Win_fence`, notamment lorsque, lors de la fermeture, le paramètre *MPI_MODE_NOSUCCEED* n'a pas été communiqué. L'automate obtenu est celui décrit à la figure 8.

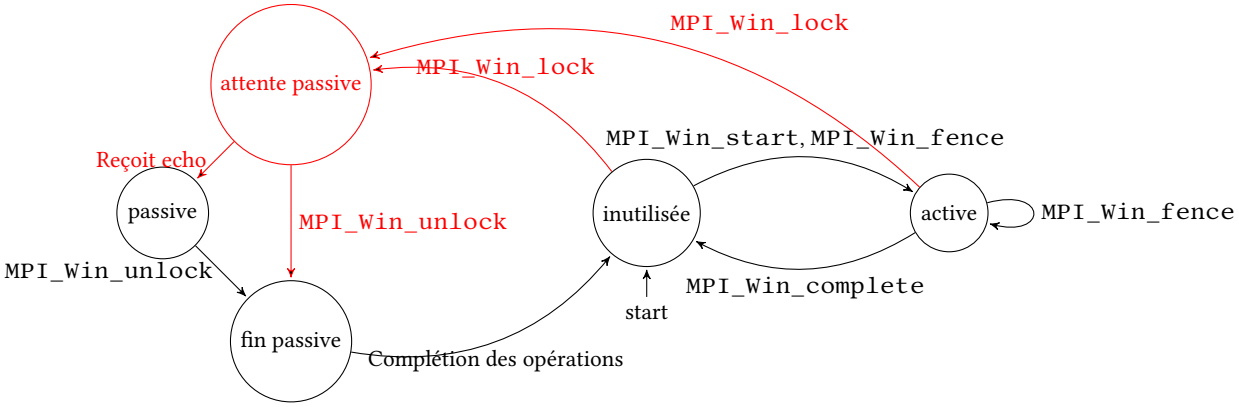


FIGURE 8 – États et transitions d'états d'une fenêtre en période d'accès passif

Bien que l'état "fin passive" soit techniquement un état actif d'opérations, il n'est pas nécessairement problématique de changer directement de l'état "attente passive" à "fin passive". En effet, lors de l'entrée dans la fonction, toutes les opérations ont été soumises. Même s'il peut arriver qu'aucune de ces fonctions n'ait effectivement eu lieu, le verrou n'ayant pas encore été donné à ce processus, aucune nouvelle opération ne peut être soumise.

3.4 Présentation des opérations

Dans un souci d'efficacité des opérations, et afin de pouvoir permettre un recouvrement des communications par le calcul, chacune de ces opérations est non-bloquante *sur l'opération*. Ces opérations s'effectuent en deux temps : envoi du type de données utile à la cible si celle-ci ne le connaît pas déjà, puis envoi des données. Afin de s'assurer que le *datatype* est bien reçu et désérialisé avant de commencer, l'envoi des données des données attend un accusé de réception de la part de la cible avant de démarrer. Cette partie ne dépend pas de l'obtention d'un verrou, ou du démarrage d'une période d'accès. C'est pourquoi cet échange peut-être réalisé immédiatement. De plus, le mécanisme permettant la réception asynchrone du *datatype* est géré extérieurement aux mécanismes mis en œuvre pour les opérations ou pour la synchronisation.

Les opérations, une fois reçues par la cible, entraînent l'exécution de la fonction de rappel asynchrone correspondante générant une *MPI_Request* pour la communication. Si le verrou est pris par l'origine, alors la requête est postée et la procédure spécifique à chaque opération est exécutée, sinon la requête est stockée dans la file d'attente correspondante à l'origine. Cette file sera vidée, et les requêtes postées, lorsque le verrou sera acquis par l'origine.

Pour les opérations exécutées sur une fenêtre de type *MPI_WIN_FLAVOR_SHARED*, si la cible n'a pas encore commencé la période d'exposition, alors les requêtes correspondantes aux opérations sont enfilées à la source. En effet, les opérations sur mémoire partagée ne nécessitant pas d'être exécutées par la cible, il est plus économique d'éviter les envois de messages inutiles.

Les opérations entraînent l'échange de un ou deux messages, selon que l'opération nécessite un retour. Une exception à cette règle étant les fenêtres de saveur *MPI_WIN_FLAVOR_DYNAMIC*. La vérification de l'adresse d'écriture des données ne pouvant être exécutée par le processus source, celle-ci nécessite donc une vérification distante ainsi qu'un message validant ou non l'adresse. Dans un souci de conserver le caractère non-bloquant des opérations, si l'opération d'écriture s'effectue en dehors de la mémoire attachée à la fenêtre,

l'opération n'est évidemment pas appliquée, mais de plus, si une réponse était nécessaire, celle-ci est annulée, chez la source comme chez la cible.

En outre, c'est le besoin d'un message retour correspondant à chaque opération, sans possibilité de mauvais appairage dans le cas d'exécution multifilière, qui nécessite l'usage d'un numéro de séquence pour chaque opération.

Pour l'ensemble des communications nécessaires au bon fonctionnement des opérations unilatérales et de la synchronisation, une grande attention à été portée sur la gestion des *MPI_Request*. Celles-ci sont allouées et libérées entièrement automatiquement grâce, une fois de plus, aux événements enregistrés sur les statuts de ces requêtes. Ces événements entraînent l'exécution de fonctions de rappel gérant les libérations des mémoires tampon si nécessaire, les terminaisons de la période ou la libération de la requête. Par ailleurs, la terminaison de la période entraîne justement l'application de l'algorithme 2, si la fenêtre est dans l'état "passive", et que l'ensemble des opérations est terminé.

3.4.1 Construction dynamique des messages

Par la définition des types de données, les messages sont générés à la volée lors du paquetage des messages par les pilotes réseaux. Ce rassemblement des données est réalisé par un foncteur sur les données. Cependant, dans le cadre des opérations unilatérales, ce parcours doit aussi pouvoir être effectué lors du déballage. Pour cela, le *datatype* a été transmis (cf. section 3.2.1), mais il reste nécessaire d'envoyer, avec les données, cette information. Par ailleurs, tant pour une question de performances que pour éviter les erreurs en cas d'exécution multifilière, cette information ne peut être envoyée dans un message séparé.

Il a donc été nécessaire d'ajouter à nos données linéarisées un entête, lequel contient le *hash* du type de données nécessaire à l'opération, le décalage dans la mémoire attachée à la fenêtre auquel commencent les données ainsi que le nombre d'éléments du type donné. Cet entête est donc dynamiquement ajouté avant les données lors du regroupement des données envoyées.

Cependant, les pilotes réseaux utilisés proposent deux modes de communication. Un premier mode, le mode *pressé*, envoie immédiatement les données dès qu'elles sont disponibles. Ce mode est utilisé pour l'envoi de données dont la taille est inférieure ou égale au seuil de rendez-vous. Le second mode, le mode *rendez-vous*, envoie une demande de rendez-vous à un nœud distant, et lorsque les deux nœuds sont prêts, l'envoi de données peut débuter.

Or, l'accession aux données, et leur dépaquetage ne peut être réalisé qu'une fois que l'entête a été lu, et si l'échange nécessite une prise de rendez-vous, aucune donnée n'est envoyée. Il a donc été ajouté, au cœur de la bibliothèque, la gestion de l'envoi de ces entêtes. Ainsi, celui-ci est ajouté au message de prise de rendez-vous. Cet ajout s'est accompagné de l'apparition d'une fonction de *peek*. Celle-ci permet une lecture de l'entête sans consommation des données de la part du pilote. Il est ainsi possible d'accéder aux informations nécessaires à la cible, et de poster la requête idoine. La quantité de données échangeable pour ce mode est donc limitée par le seuil de rendez-vous défini par les pilotes réseaux.

Enfin, lors de la réception d'une opération unilatérale, le type d'opération est donc encodé dans le tag du message reçu. Par ailleurs, les opérations étant au nombre de sept (six opérations auxquelles s'ajoutent un tag pour les données retournées), leur encodage peut donc être effectué sur quatre bits. Or, certaines opérations RMA nécessitent de communiquer une variable de type *MPI_Op*. Ces *MPI_Op* étant au nombre de quatorze, elles sont aussi encodées dans le tag. Cela donne la structure telle que présentée dans la figure 9.

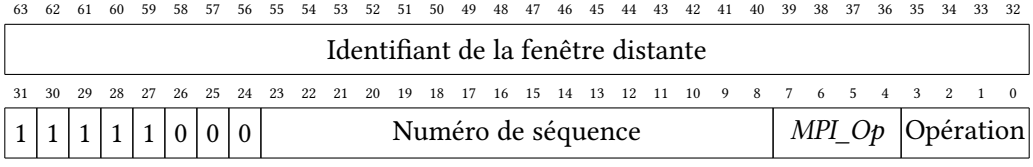


Figure 9 – Description champ par champ du tag pour les opérations RMA

3.4.2 `MPI_Put` et `MPI_Rput`

Ces opération sont les opérations de base de la bibliothèque. Elle consistent en une écriture dans la mémoire du processus distant. Lors de la réception d'une opération de type `put`, l'entête permet de générer la requête de réception immédiatement, et de déballer les données à la bonne position. L'utilisation du verrou de protection des données écrites n'est pas nécessaire, car la gestion du cas de chevauchement de données n'est pas spécifiée.

La version `MPI_Rput` diffère en ce qu'elle retourne à l'utilisateur une *MPI_Request* permettant de suivre l'avancement de la requête à l'origine. La terminaison de cette requête n'implique pas la terminaison de l'opération du côté cible. La complétion de la requête n'indique que la possibilité de modifier le tampon d'envoi de données.

3.4.3 `MPI_Get` et `MPI_Rget`

Ces opérations sont les secondes opérations de base de la bibliothèque. Elle consistent en une lecture dans la mémoire du processus distant. Le message envoyé consiste en un entête simple contenant un *hash* du *datatype* distant, le nombre d'entité du *datatype* requis ainsi qu'un décalage dans la zone mémoire distante. Lorsque la requête sera exécutée, les données envoyées permettront de générer la requête d'envoi de ces données, de la cible vers l'origine. Cette génération est bien entendu gérée automatiquement par la bibliothèque.

La version `MPI_Rget` diffère en ce qu'elle retourne à l'utilisateur une *MPI_Request* permettant de suivre l'avancement de la requête à l'origine. La terminaison de cette requête est la seule qui implique la terminaison de l'opération du côté cible. En effet, la terminaison de la requête retournée indique que les données sont accessibles dans le tampon d'origine.

3.4.4 `MPI_Accumulate` et `MPI_Raccumulate`

L'opération d'accumulation est une opération distante permettant d'appliquer des opérations, généralement commutatives, sur des données distantes. Les opérations possibles sont cependant restreintes aux seules opérations prédéfinies par le standard [19].

Lors de la réception d'une requête de ce type, un buffer de la taille des données envoyées est alloué, et la requête postée immédiatement. En effet, la possibilité de déballer les données en mémoire en appliquant immédiatement une opération sur ces dernières n'étant pas accessible, le passage par une copie dans un tampon intermédiaire est de toute façon nécessaire. Ce tampon ne pouvant entrer en interaction avec les données de la cible, il est donc possible d'y conserver les données envoyées. Ainsi, dans le cas d'envoi d'un grand nombre de données, l'envoi peut être effectué immédiatement, sans avoir à attendre l'ouverture de la période d'exposition. Une fois les données reçues et la période d'exposition débutée, l'application de l'opération s'effectue en utilisant le *datatype* communiqué dans l'entête.

La version `MPI_Raccumulate` diffère en ce qu'elle retourne à l'utilisateur une *MPI_Request* permettant de suivre l'avancement de la requête à l'origine. La terminaison de cette requête n'implique pas la terminaison de l'opération du côté cible. La complétion de la requête n'indique que la possibilité de modifier le tampon d'envoi de données.

3.4.5 `MPI_Get_accumulate` et `MPI_Rget_accumulate`

Cette fonction est l'équivalent en une seule opération de l'application d'une opération `get` suivie de l'application d'une opération `accumulate`. L'utilisateur récupère donc les données avant modification. Pour cela, lors de la réception par la cible d'une opération de ce type, de même que dans le cas d'une opération `accumulate`, de la mémoire est allouée afin de recevoir les données. Puis, lorsque la période d'exposition commence, le zone mémoire est verrouillée, les données copiées dans un tampon intermédiaire, puis l'opération est effectuée. La copie dans un tampon intermédiaire avant l'envoi permet de lancer l'envoi pendant

19. N'ayant pas de moyen de partager avec un processus distant une fonction, il est donc naturel que les seules opérations autorisées soient celles prédéfinies.

l'application des opérations. Sans cela, il y aurait un délai supplémentaire avant l'application de l'opération, et il serait nécessaire de chaîner des requêtes. La copie permet donc une plus grande flexibilité et une plus grande indépendance des requêtes.

La version `MPI_Rget_accumulate` diffère en ce qu'elle retourne à l'utilisateur une *MPI_Request* permettant de suivre l'avancement de la requête à l'origine. La terminaison de cette requête n'implique pas la terminaison de l'opération du côté cible. Cependant, sa terminaison, de même que pour l'opération `get`, indique que les données sont disponibles dans le tampon de réception.

3.4.6 `MPI_Fetch_and_op`

Cette fonction est l'équivalent de l'opération `get_accumulate`, avec la restriction du type. En effet, cette opération a pour but de permettre d'être effectuée plus rapidement, car ne s'appliquant que sur une donnée unique. Par ailleurs, le *datatype* fourni est nécessairement un type prédéfini dans la bibliothèque. Cette opération est exécutée de façon atomique, utilisant une barrière d'exclusion mutuelle pour cela.

3.4.7 `MPI_Compare_and_swap`

Cette opération est la base des opérations atomiques. Celle-ci compare donc une valeur envoyée par origine avec une valeur de la cible. Si celles-ci sont égales, la valeur chez la cible est remplacée par une seconde valeur envoyée par l'origine. La fonction renvoie à la source la valeur initiale de la cible. De même que pour les fonctions prédéfinies dans les compilateurs, l'application de cette opération est effectuée de façon atomique, utilisant une barrière d'exclusion mutuelle pour cela.

4 Résultats

Suite à la création de l'ensemble de ces fonctions, nous avons donc chercher à évaluer les résultats et la qualité du travail réalisé. Nous avons donc mesuré les performances de chaque fonction, en mesurant également le temps d'ouverture des fenêtres, tant en mode "actif" qu'en mode "passif". Par ailleurs, par le fonctionnement des bancs d'essais, nos mesures ne prennent pas en compte l'échange des types de données. Ils sont considérés comme étant déjà connus de chaque côté. Les bancs d'essais ne présentent pas les résultats pour les fonctions `MPI_Win_Fetch_and_op` et `MPI_Win_Compare_and_swap`, celles-ci n'envoyant un nombre de données sinon fixe, pour le moins majorable par la taille maximale des *datatypes* prédéfinis pour la première (64 bits) et deux fois cette taille pour la seconde (128 bits). Ces opérations sont cependant pleinement fonctionnelles.

4.1 Performances des opérations

Les résultats de performances sont mis en parallèle avec les performances obtenues pour un envoi simple de message. Les résultats peuvent être observés dans les figures 10 et 11, pour une synchronisation active et une synchronisation passive respectivement.

4.1.1 Synchronisation active

Cette première figure permet de mettre en lumière trois points. Tout d'abord, on peut observer le coût assez important de la synchronisation. En effet, dans le cas des envois de messages de quelques octets, on a une latence pour un envoi à réception explicite de l'ordre d'une microseconde et demie. Dans le cadre des RMA, cette latence est effectivement triplée voire quadruplée, ce qui est un résultat conforme vis à vis du protocole implémenté [20]. C'est pourquoi, pour un code dont l'objectif est d'échanger de nombreux messages de petite taille, il sera plus intéressant de n'ouvrir qu'une seule fois la fenêtre. En outre, l'utilisation d'opérations RMA n'entre pas en conflit avec l'utilisation des fonctions de synchronisation usuelles.

20. La synchronisation coûte l'échange de trois messages, parmi lesquels deux vides et un de 64 bits. (cf. section 3.3.3.1)

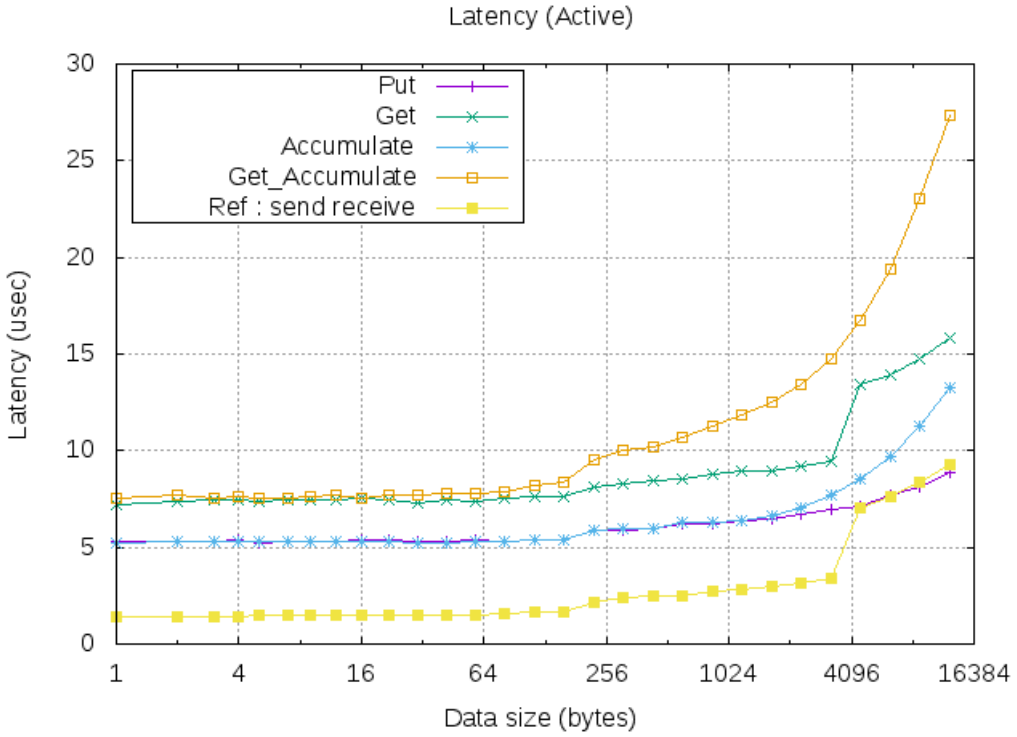


FIGURE 10 – Résultats bancs d'essais, synchronisation active

Deuxièmement, on remarque que la latence induite par la synchronisation n'est un réel surcoût que lorsque les données sont envoyées en mode *pressé*. La limite des quatre kilo-octets entraîne un pic pour l'envoi à réception explicite. Ce pic n'est pas présent pour les opérations `put`, `accumulate` et `get_accumulate` grâce aux stratégies d'agrégation des paquets. On observe cependant la présence de ce pic des quatre kilo-octets pour la fonction `get`. On peut d'ailleurs observer une excellente corrélation entre les résultats obtenus pour `sendreceive` et pour `get`. Ce résultat est dû au fait que outre l'envoi de la requête de données, les données effectivement retournées le sont par un envoi standard ; la cible connaît le type de données, le nombre de données de ce type à envoyer et leur position, l'origine sait ce qu'elle doit recevoir.

Enfin, on peut observer que la courbe de l'opération `get_accumulate` suit régulièrement la courbe du `accumulate`. La divergence finale vient du fait que l'échange supplémentaire de données ajoute du temps d'échange supplémentaire. De plus, il est nécessaire d'attendre la complétion de cet envoi avant de pouvoir effectuer les opérations. Le coût d'application des opérations est d'ailleurs double pour l'opération `get_accumulate`, car les données sont d'abord copiées dans un tampon d'envoi, puis les opérations sont réalisées lors d'un second parcours.

4.1.2 Synchronisation passive

Pour la synchronisation passive, on observe une fois encore le coût de la synchronisation qui explique le décalage initial entre nos opérations et un échange à réception explicite de données. Par ailleurs, on peut remarquer que la latence est supérieure pour les opérations ne nécessitant pas de message de retour, contrairement au mode actif. Cette différence s'explique par l'application de la stratégie d'agrégation de paquets. Dans le cas du mode actif, il y avait envoi d'un message d'ouverture de la cible vers l'origine, puis application des opérations, et enfin envoi de fin de période. Cela entraîne l'échange de trois messages pour les opérations `put` et `accumulate`, et de quatre messages pour les opérations `get` et `get_accumulate`. Dans le mode passif, il y a nécessité d'attendre un accusé de réception de fermeture. Ce message explique la latence supérieure des opérations ne nécessitant pas de réponse. Par ailleurs, pour les deux autres opérations, on observe une baisse de la latence, pouvant s'expliquer par l'application de la stratégie d'agrégation de paquets. Par exemple, si les trois envois (demande ouverture, envoi des données, demande de fermeture) sont agrégés en un seul envoi sur le réseau, la latence réseau n'est présente qu'une seule fois. On observe, en outre, ce coût supplémentaire

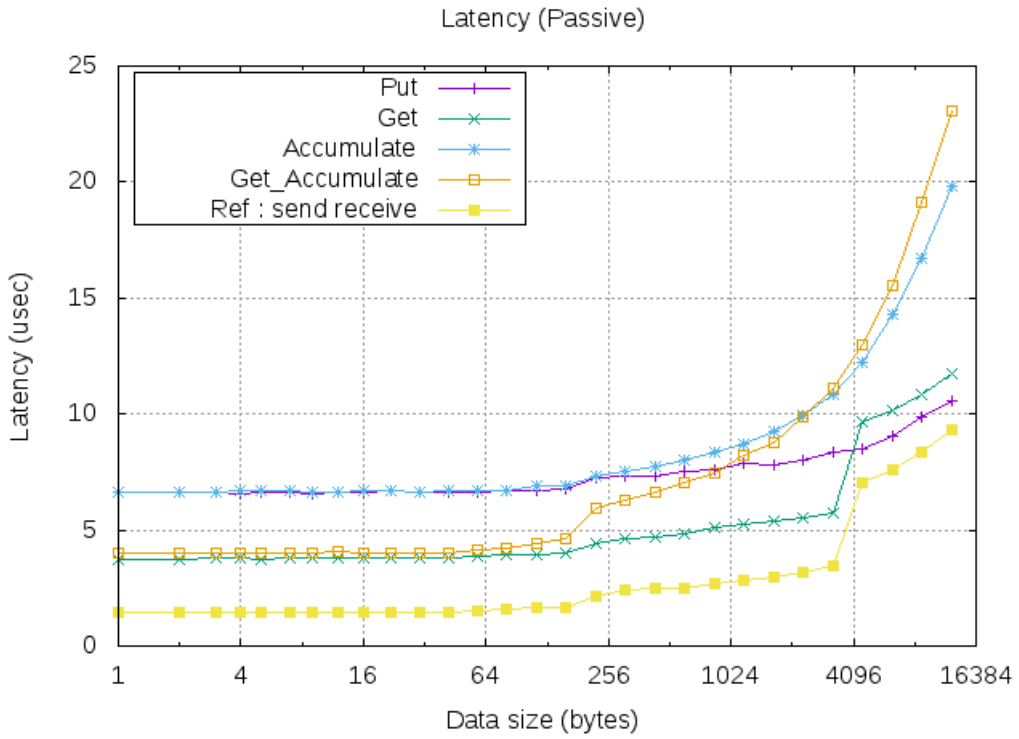


FIGURE 11 – Résultats bancs d'essais, synchronisation passive

fixe de 3 microsecondes entre l'opération `get` et la réception explicite, ainsi qu'un coût supplémentaire fixe de 1,5 microsecondes entre l'opération `put` et la réception explicite, passé la limite des quatre kilo-octets.

Enfin, on observe toujours la forte influence du coût des copies et de l'application des *MPI_Op* des opérations unilatérales `accumulate` et `get_accumulate`.

4.2 Comparaisons avec les implémentations *OpenMPI* et *MVAPICH*

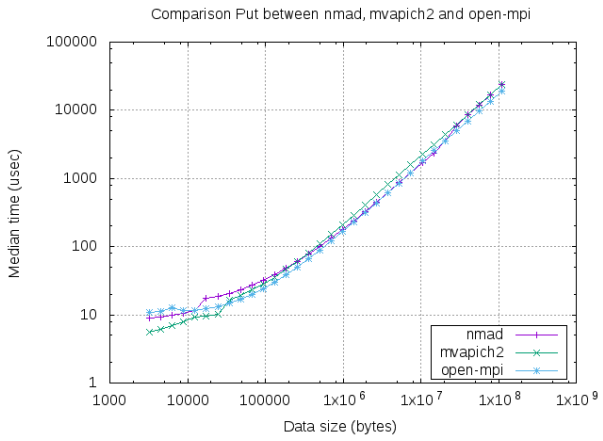
Cette section présente les résultats obtenus avec *NewMadeleine* comparés aux performances des bibliothèques *OpenMPI* et *MVAPICH*. Les graphiques des figures 12, 13 et 14 présentent ces comparatifs, dans le cadre d'absence de recouvrement de calculs par des communications. *NewMadeleine* est compilé avec *gcc*, en optimisation –O2, et utilise *PIOMAN* en collaboration avec *pthread*. Les figures 15, 16, 17, 18 et 19 présentent les résultats dans le cas de communications recouvertes par des calculs. Ces résultats ont été obtenus ultérieurement à la terminaison du stage. Les tests ont été exécutés et les résultats fournis par M. Alexandre Denis, et vous sont présentés ici avec son censentement.

4.2.1 Sans recouvrement

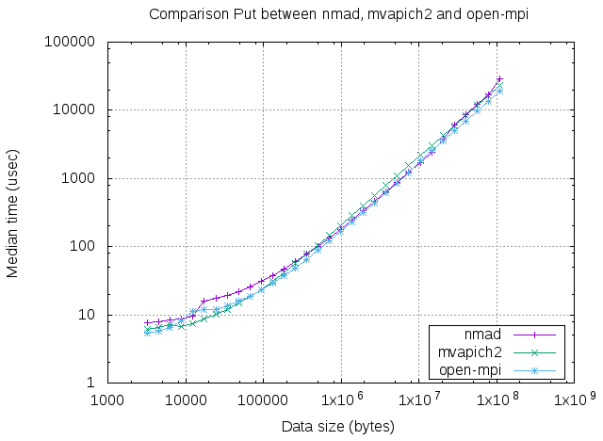
On observe une plus grande régularité dans les résultats obtenus pour la bibliothèque *NewMadeleine* que pour *MVAPICH* ou *OpenMPI*. Bien que les performances soient légèrement moindres, les ordres de grandeurs sont plutôt équivalents et les performances plutôt homogènes. Il est cependant à noter que ces tests ne prennent pas en compte l'influence de calculs exécutés au même moment. Plusieurs améliorations sont toujours possibles, notamment au niveau de la latence de l'envoi des messages. Les évolutions susceptibles d'améliorer ces résultats seront présentées et discutées dans la section 4.3.

4.2.2 Avec recouvrement

La lecture des figures idoines s'effectue avec la même grille que la figure 1. La ligne blanche explicite les paramètres pour lesquels le temps de calcul est égal au temps d'envoi des données. Le ratio de recouvrement est normalisé.

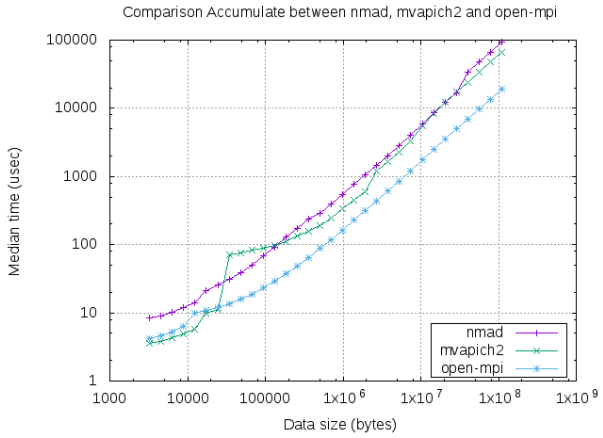


Synchronisation active

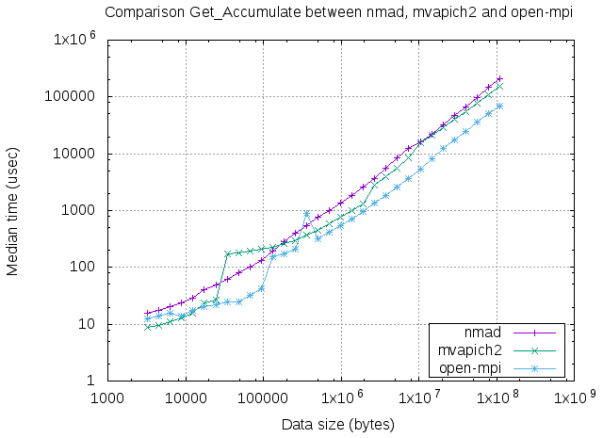


Synchronisation passive

FIGURE 12 – Comparaison `MPI_Put`

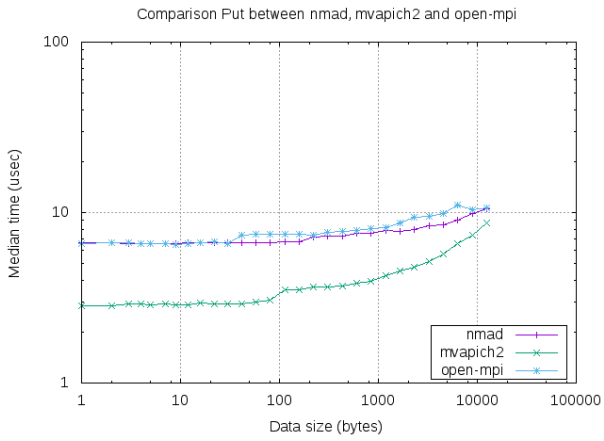


`MPI_Accumulate`

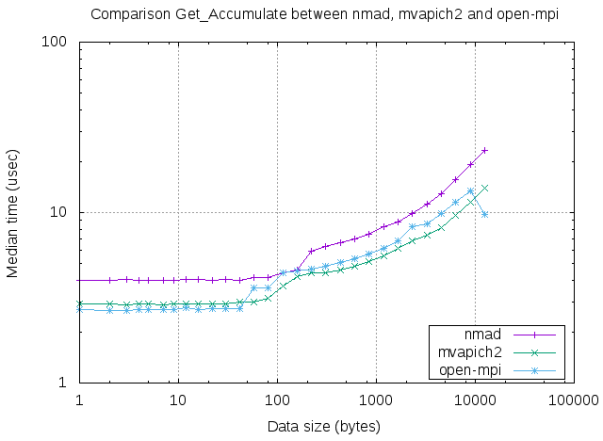


`MPI_Get_accumulate`

FIGURE 13 – Comparaison `MPI_Accumulate` et `MPI_Get_accumulate`, synchronisation active

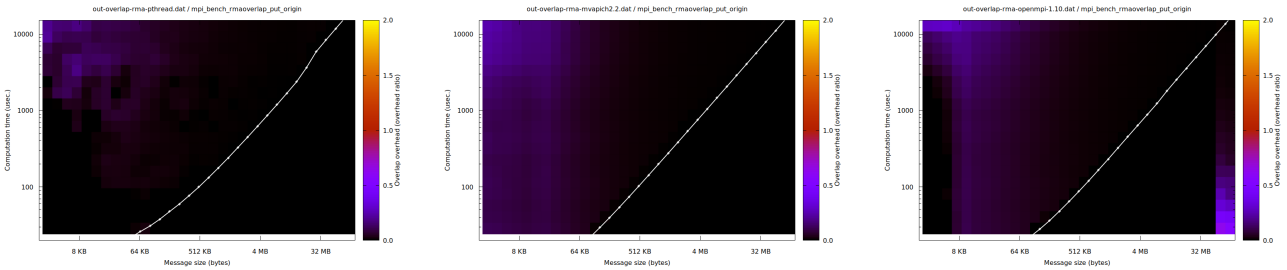


`MPI_Put`



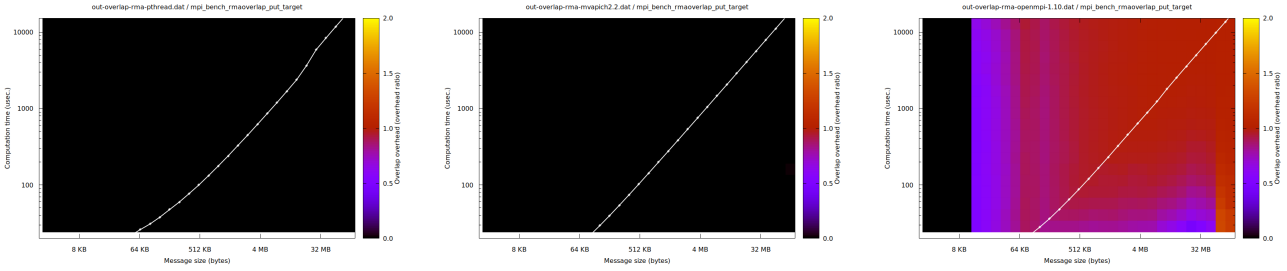
`MPI_Get_accumulate`

FIGURE 14 – Comparaison latences `MPI_Put` et `MPI_Get_accumulate`, synchronisation passive



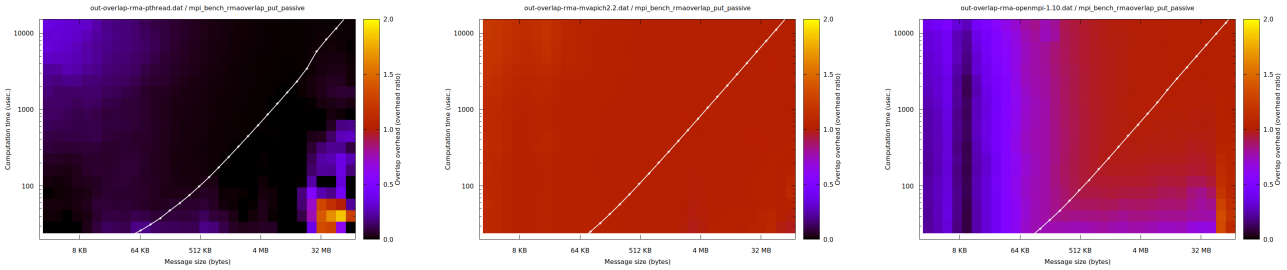
NewMadeleine *MVAPICH* *OpenMPI*

FIGURE 15 – Comparaison `MPI_Put`, calculs à l'origine, synchronisation active



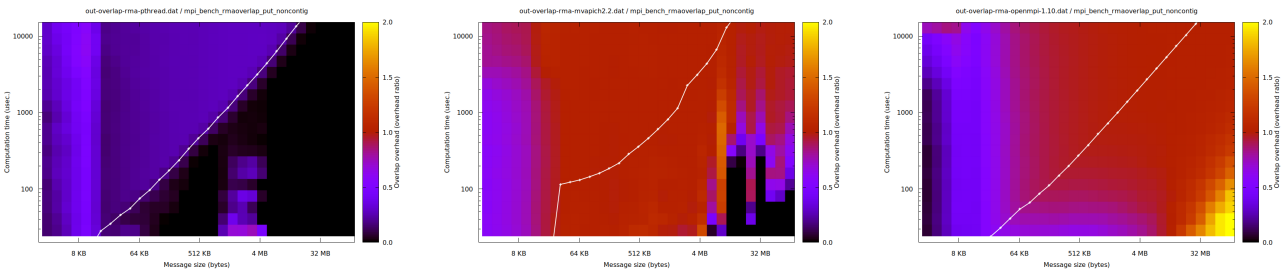
NewMadeleine *MVAPICH* *OpenMPI*

FIGURE 16 – Comparaison `MPI_Put`, calculs à la cible, synchronisation active



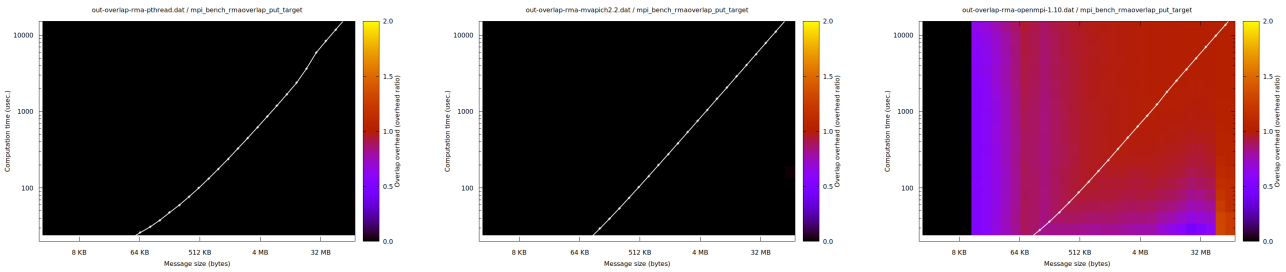
NewMadeleine *MVAPICH* *OpenMPI*

FIGURE 17 – Comparaison `MPI_Put`, calculs à l'origine et à la cible, synchronisation passive



NewMadeleine *MVAPICH* *OpenMPI*

FIGURE 18 – Comparaison `MPI_Put`, calculs à l'origine, synchronisation active, *datatype* non-contigü



NewMadeleine *MVAPICH* *OpenMPI*

FIGURE 19 – Comparaison `MPI_Get`, calculs à l'origine et à la cible, synchronisation active

On peut observer que le cas standard est optimisé pour chacune des implémentations, lors d'une synchronisation active. Cependant, le recouvrement des calculs du côté cible, malgré la synchronisation active, entraîne un premier surcoût pour *OpenMPI*. La figure 19, associée aux figures 15 et 16, montre que la localité des calculs n'influence pas les temps de communication, tant pour la bibliothèque *NewMadeleine* que pour la bibliothèque *MVAPICH*. En revanche, la figure 17 montre que le passage d'une synchronisation active vers une synchronisation passive entraîne un fort impact sur les performances des bibliothèques *MVAPICH* et *OpenMPI*, de même que le caractère non-continu des données.

4.3 Améliorations futures

Bien qu'il fut apporté un soin tout particulier aux performances, celles-ci ne sont pas encore optimales. Il apparaît cependant un certain nombre de pistes d'améliorations de ces résultats. Tout d'abord, on observe que le surcoût principal, pour des messages de petite taille et vis à vis des envois à réception explicite, vient de la latence des différents messages de synchronisation échangés. Une meilleure intégration de ces opérations vis à vis de la stratégie d'agrégation des paquets permettrait une nette amélioration. Par exemple, une opération `put` exécutée avec une synchronisation passive pourrait être effectuée en deux messages. Le premier message contiendrait la demande d'ouverture, les données de l'opération et la demande de fermeture. Le second contiendrait l'accusé de réception de fermeture. Pour une opération `get`, le message de fermeture pourrait aussi être accompagné par les données retournées. Par ailleurs, le changement du caractère bloquant de la fonction `MPI_Win_start` permettrait aussi une meilleure intégration à la stratégie.

Une seconde piste d'amélioration serait la gestion de la copie supplémentaire pour l'application des opérations, ainsi que celle nécessaire au renvoi des données. Actuellement, les pilotes réseaux ne permettent pas simplement d'utiliser d'opérations autres que la copie d'un buffer vers un autre buffer. La possibilité d'appliquer à la volée, lors du déballage, permettrait de ne parcourir les données qu'une seule fois du côté cible. En effet, il serait possible de déplacer dans le buffer d'envoi la donnée avant modification, puis d'appliquer l'opération *MPI_Op* demandée. Le buffer resterait cependant nécessaire, mais son coût à l'envoi serait moins important que celui d'un parcours supplémentaire des données.

En outre, les cas particuliers, indiqués à l'aide des assertions et des objets *MPI_Info*, ne sont pas tous pris en compte. Leur traitement permettrait encore une amélioration des performances dans certains cas. De même, une gestion plus fine des échanges de *datatypes* pourrait permettre une baisse du nombre de messages dans certains cas. En effet, lorsqu'un type de données est échangé, l'information est conservée pour le type, mais non pour les sous-types desquels il dérive. Il peut donc potentiellement advenir de l'envoi d'un *datatype* et de son sous-type lors de deux échanges de données, là où l'envoi du *datatype* principal suffirait.

Enfin, quelques parties du code pourraient être modifiées pour l'impact de la synchronisation dans certains cas. Par exemple, lorsqu'un processus essaie de se verrouiller, il pourrait être envisageable de gérer le cas par sémaphores système plutôt que par l'envoi d'un message du processus vers lui-même. De même, dans le cas des fenêtres de type *MPI_WIN_FLAVOR_SHARED*, la gestion de la synchronisation pourrait être effectuée par sémaphores partagés plutôt que par l'échange de messages. Enfin, la gestion des files d'attente de requêtes pourrait être améliorée en utilisant des listes chaînées rapides plutôt que les listes chaînées du système.

5 Conclusion

Bien qu'une très grande autonomie m'ait été accordée, les échanges réguliers avec mon maître de stage m'ont permis de mieux situer les éléments les plus critiques de cette réalisation et de rapidement trouver des solutions lors du développement ; par exemple lors de la mise en place des protocoles de synchronisation. Ces interactions ont contribué à une rapide avancée du projet, m'évitant ainsi un certain immobilisme pouvant être frustrant.

Outre me passionner et confirmer mon appréciation grandissante pour ce domaine d'application de l'informatique, ce stage m'a permis d'appronfondir et de comprendre les modalités de développement des projets de taille conséquente.

De plus, la découverte d'une nouvelle vision du domaine de l'informatique haute performance a été très enrichissante par une plus grande connaissance des problématiques abordées et des solutions apportées. De surcroît, l'évolution de la bibliothèque au fur et à mesure des ajouts était très valorisante. En outre, l'étude et l'interprétation du standard ont représenté des nouveaux problèmes auxquels je n'avais jamais été confronté. J'ai ressenti une grande satisfaction à être parvenu à en couvrir les spécifications.

Les performances obtenues étant d'un niveau assez proche des bibliothèques "standards" permettent de bons espoirs pour l'évolution de *NewMadeleine*, notamment avec les pistes précédemment évoquées. De plus, il semblerait que la norme *MPI-4*, actuellement en discussion, tendrait à modifier la synchronisation des opérations. Cette évolution a donc été anticipée lors de la création de l'architecture du projet, et pourrait être mise en place assez aisément.

Enfin, le développement actuel des applications dans le domaine du HPC semble se diriger vers une utilisation plus importante des cartes de calculs dites *Many-Core*, telles que les *Xeon Phi* développés par Intel®. Ces cartes possédent un système d'exploitation et un environnement d'exécution. Aussi, peut-on s'attendre à voir les applications utiliser un grand nombre de cœurs, entraînant un usage massif de la mémoire partagée.

6 Références

[1] Elisabeth Brunet, Olivier Aumage et Raymond Namyst : NewMadeleine : ordonnancement et optimisation de schemas de communication haute performance. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, vol. 27(3-4/2008), 2008.

[2] Alexandre Denis : pioman : a Generic Framework for Asynchronous Progression and Multithreaded Communications. *In IEEE International Conference on Cluster Computing (IEEE Cluster)*, Madrid, Spain, septembre 2014.

[3] Alexandre Denis, Nathalie Furmento et Raymond Namyst : NewMadeleine, An Optimizing Communication Library for High-Performance Networks. `http://pm2.gforge.inria.fr/newmadeleine/`.

[4] Alexandre Denis et François Trahay : PIOMan, A generic I/O Manager. `http://pm2.gforge.inria.fr/pioman/`.

[5] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham et Timothy S. Woodall : Open MPI : Goals, concept, and design of a next generation MPI implementation. *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[6] mpiexec man page. `http://linux.die.net/man/1/mpiexec`.

[7] MadMPI benchmark : MPI overlap benchmark. `http://pm2.gforge.inria.fr/mpibenchmark/`.

[8] MPICH. `https://www.mpich.org/`.

[9] MVAPICH. `http://mvapich.cse.ohio-state.edu/`.

[10] The MPI Forum : Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/`, June 4th 2015.

[11] The MPI Forum. *MPI : A Message-Passing Interface Standard. Version 3.1*, June 4th 2015.

[12] The MPI Forum. *MPI Section 11 : One-Sided Communications*, June 4th 2015.