Analysis and optimizations for partitioned global address space based HPC applications

Célia Tassadit Ait Kaci

▶ **To cite this version:**

Célia Tassadit Ait Kaci. Analysis and optimizations for partitioned global address space based HPC applications. Other [cs.OH]. Université de Bordeaux, 2022. English. NNT : 2022BORD0464 . tel-04291830

HAL Id: tel-04291830

https://theses.hal.science/tel-04291830

Submitted on 17 Nov 2023

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

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

THÈSE PRÉSENTÉE POUR

OBTENIR LE GRADE DE

DOCTEUR

DE L'UNIVERSITÉ DE BORDEAUX

Ecole Doctorale de Mathématiques et Informatique Bordeaux

Spécialité : **Informatique**

Par **Célia Tassadit AIT KACI**

Sous la direction de : **Denis Barthou**

Analyse et optimisations pour les applications HPC à mémoire distribuée et adressable globalement

Soutenance le 28 Novembre 2022 devant la commisission d'examen composée de :

M. William Jalby	Professeur, Université de Paris Saclay - Versailles	Rapporteur
M. Laurent Colombet	Directeur de Recherche, CEA	Rapporteur
M. Denis Barthou	Professeur, Bordeaux INP	Directeur de thèse
M. Brice Goglin	Directeur de Recherche, Inria	Examinateur
Mme. Corinne Ancourt	Directrice de Recherche, Ecole des Mines de Paris	Examinatrice
Mme. Emmanuelle Saillard	Chargée de Recherche, Inria	Examinatrice (Encadrante)
M. Marc Sergent	Ingénieur de Recherche, Atos	Examinateur (Encadrant)
M. Scott Baden	Professeur Émérite , Université de San Diego Californie	Invité

Résumé

Le monde du Calcul Haute Performance ne cesse d'évoluer, en proposant des modèles de programmation parallèles vastes et variés. Pour s'adapter aux exigences de l'exascale en terme de puissance de calcul et de traitement rapide. Le modèle de programma- tion parallèle le plus utilisé est le Message Passing Interface (MPI). Le standard MPI a été utilisé dans le HPC pendant des décennies. Le protocole de communication point à point send/receive, appelé two-sided MPI, est largement utilisé et privilégié dans les applications. Une alternative à ce modèle de communication point à point est les com- munications one-sided, principalement implémenté dans les langages et bibliothèques PGAS (Partitioned Global Address Space). Le standard MPI-3, qui a été présenté en Septembre 2012, a inclus une mise à jour importante des communications one-sided dans MPI, aussi appelées les RMA (Remote Memory Access) et adoptées par le standard de- puis MPI-2, pour fournir plus de performance et introduire de nouveaux modes d'accès aux données. Cependant, les performances des communications one-sided restent loin d'être celles attendues.

Développer un programme parallèle est souvent plus difficile mais plus performant dans de la mémoire partagée plutôt que d'utiliser des transferts send/receiv pour échan- ger des données, les processus peuvent implicitement communiquer dans la mémoire partagée avec une utilisation de certains mécanismes de synchronisation (verrous, séma- phores..) pour garantir un accés sans concurrence à la partie mémoire souhaitée.

Dans ces travaux de recherche nous allons principalement nous intéresser au modèles de programmation à mémoire distribuée et globalement adressable MPI-RMA dont le principe est de designer un espace mémoire virtuel global et partagé dans les systèmes à mémoire distribuée, ce qui permet aux processus de communiquer à travers cette mémoire. Bien que le modèle PGAS existe depuis très longtemps, et promet plus d'asyn- chronisme. Ce modèle reste peu utilisé par la communauté du HPC à cause de plusieurs raisons notamment les modes de synchronisation requis pour sécuriser le programme. Comme le modèle PGAS propose un concept de programmation en mémoire partagée dans de la mémoire distribuée. Les problèmes de concurrence d'accès s y appliquent. Pour cette raison la programmation PGAS peut se révéler très difficile, car l'utilisateur a la responsabilité de gérer explicitement tous les accès mémoire pour garantir la co- hérence du programme. Il est donc intéressant pour les programmeurs d'applications, d'avoir des outils qui leur permettent de faciliter la programmation. De développer des codes correctes et efficaces.

Dans le cadre de ces travaux de recherche l'objectif principal est de développer une analyse dynamique à l'exécution, et une analyse statique à la compilation, pour vérifier les codes des applications PGAS. Cette analyse mixte permet d'exploiter les avantages des deux approches. une approche dynamique qui repose sur des exécutions concrètes qui dépendent d'un seul jeu d'entrée, et donc se limiter à détecter que les erreurs présentes

dans l'exécution analysée. Une approche statique qui ne dépend pas du jeu d'entrée et offre une vue globale du code, et considère tous les chemins d'exécution possibles.

Durant cette thèse nous avons developpé un outil qui regroupe les deux analyses statique et dynamique appelé RMA-Analyzer. Il a été propsé dans but d'aider à la programmation de codes MPI-RMA, notamment en proposant une aide avancée à l'utilisateur dans la détection d'erreurs de concurrence connus comme accès illégaux liés aux applications MPI-RMA. Dans le but de faciliter la programmation aux utilisateurs, avec une aide dynamique sur la réalité et l'origine des éventuels blocages en MPI-RMA.

Mots clès : Programmation parallèle, MPI-RMA, Analyse statique et dynmaique.

Contexte de la recherche

La simulation numérique demande des codes qui ont un fort besoin en puissance de calcul et de traitement rapide, pour résoudre des problèmes à très grande taille. Cela a conduit à la conception de machines très puissantes et massivement parallèles et qui visent le milliard de milliards d'opérations par seconde (exaflops). Pour utiliser ces ma- chines efficacement, nous avons besoin de modèles de programmation parallèles simples et de supports d'exécution efficaces. Dans ce contexte, nous allons nous intéresser au modèle de programmation parallèle basé sur des communications unilatérales appelé MPI-RMA.

Développer un programme parallèle en utilisant MPI-RMA est souvent plus difficile, mais plus performant grace aux accès directs à la mémoire partagée entre les processus, plutôt que d'utiliser des transferts send/receive pour échanger des données, les processus peuvent implicitement communiquer directement en utilisant les communications unila- térales. Dans le cadre de l'utilisation de ces communications, et comme la mémoire et globalement accessible par tous les processus d'un communicateur, des erreurs d'accès à la mémoire partagée peuvent avoir lieu. La détection de ces erreurs peut se révéler difficile et cela peut avoir des impacts négatifs et réduit l'utilisation de ce modèle de programmation.

Très peu d'outils ont été developpés dans le cadre de la détection d'accès concurrents dans des programmes MPI-RMA. Les outils qui existent privilégient des solutions à base de fichiers de traces qui proposent des retours à l'utilisateur à la fin du programme. Les developpeurs sont donc contraints d'attendre la fin de leurs programmes pour retrouver l'erreur avec très peu d'aide. En plus, la taille des fichiers de traces peut vite devenir très grande. Ces solutions ne sont pas adaptées à des applications à très grande échelles.

Durant cette thèse, nous nous sommes intéressés à la détection des erreurs d'accès concurrents à la mémoire partagée entre processus dans les applications MPI-RMA. Nous avons développé un outil de vérification appelé le RMA-Analyzer qui effectue une analyse

statique et une analyse dynamique afin de détecter les erreurs d'accès concurrents dans des applications MPI-RMA. Le RMA-Analayzer vérifie à la compilation la présence de potentielles erreurs locales de concurrence pour alerter l'utilisateur avant le lancement du programme. Ensuite, le RMA-Analyzer vérifie tous les accès concurrents à l'exécution pour arrêter le programme à la première erreur trouvée.

Démarche adoptée

Dans le cadre de cette thèse, notre but principal est d'apporter une aide efficace au programmeur pour lui faciliter la programmation en MPI-RMA. Programmer en MPI-RMA requiert une connaissance particulière de comment programmer en mémoire distribué et globalement adressable pour avoir une partie de la mémoire qui sera par- tagée entre processus. Un programmeur doit d'abord ouvrir une fenêtre mémoire pour que les processus s'échangent les accès, et à la fin du programme, cette fenêtre mémoire est libérée. Durant la vie de cette fenêtre mémoire des communications entre processus peuvent avoir lieu dans des époques de synchronisation bien précises pour préciser avec quel processus et quel type d'opération vont avoir lieu sur une zone mémoire ciblée. Avec ce principe de programmation, des zones mémoires sont partagées entre processus et des opérations de lecture écriture sur une même zone mémoire peuvent vite devenir problématiques. Des accès concurrents peuvent avoir lieu sur la même zone mémoire.

Notre outil repère les erreurs de concurrence d'accès à la mémoire et se charge de les montrer à l'utilisateur. Pour cela, pendant l'exécution du programme, note outil instru- mente le code pour vérifier les erreurs. Nous traçons tous les accès mémoire et nous les enregistrons dans une structure de données appelée arbre binaire de recherche et nous analysons tous les accès mémoire entre les processus d'un même communicateur loca- lement et globalement. Les processus se partagent chacun les accès enregistrés à leurs mémoires respectives aussi en s'envoyant les informations grâce à un protocole de Send/- Recv classique. Si deux accès concurrents sont détectés et au moins un des accès est une opération d'écriture, on arrête immédiatement le programme. Une analyse statique a été ajoutée comme un support à cette analyse dynamique dans le but d'alerter le program- meur pendant la compilation des potentielles erreurs liées aux accès concurrents. Toutes les erreurs locales à un processus sont ainsi détectable au préalable de l'exécution. Ce- pendant, comme le jeu d'entrée n'est pas connu au moment de la compilation, nous ne pouvons pas affirmer la présence des erreurs.

Notre première contribution consiste à identifier les d'afférents types d'erreurs qui peuvent avoir lieu dans un programme MPI-RMA. Le premier type d'erreur peut avoir lieu au sein d'un même processus et donc, des accès natifs load et store peuvent se chevaucher avec des accès distants (RMA) ou que des accès RMA entre eux (nous ne détectons pas les erreurs de concurrence d'accès purement natives). Le deuxième type d'erreur c'est les erreurs d'accès concurrents qui peuvent impliquer plusieurs processus d'un même communicateur. Ces accès peuvent avoir lieu au sein d'une mémoire d'un

processus donné avec des accès provenant de différents processus. Il peut aussi y avoir un chevauchement entre les accès load/store et les opérations RMA de plusieurs proces- sus visant la même adresse mémoire. En identifiant les types d'erreurs, nous avons créé une matrice avec croisement des accès locaux et distants dans le but de l'utiliser durant le process de la vérification d'erreur.

Une fois les types d'erreurs sont connus et identifiés, Cette matrice sera utile à notre algorithme de détection dynamique pourlever une erreur de concurrence si elle existe. À l'aide de la platforme de profiling MPI (PMPI). Nous enregistrons tous les accès d'un jeu de données durant le temps d'exécution du programme. Nous construisons des intervalles qui représentent la zone mémoire sur laquelle un des processus lit ou écrit selon l'opéra- tion de communication. La borne inférieure de l'intervalle représente le début de la zone mémoire choisie et la borne inférieure représente la fin de la zone mémoire. Ces inter- valles sont tous enregistrés avec leurs types d'accès (lecture, écriture) durant le temps d'exécution. Selon le type d'accès et la zone mémoire, s'il y a un chevauchement entre les intervalles (intersection d'intervalle), ça veut dire qu'une zone mémoire est simultané- ment accédée avec au moins un accès en écriture. Nous précisons ici que Le programme s'arrête dès la première erreur trouvée, car d'autres erreurs de concurrences dans le pro- gramme peuvent être des séquelles de la première, d'où notre intérêt de traiter le plus tôt possible ce type d'erreurs. Cette démarche implique aussi une instrumentation des accès load /store grâce à une pass LLVM pour enregistrer tous les accès locaux aussi et les comparer avec les accès RMA. Ceci peut engendrer un temps d'exécution beaucoup plus élevé au moment on nous exceutons un programme avec notre outil. C'est pourquoi nous avons réfléchi à notre deuxième contribution qui est de faire une analyse statique qui sera un support à la première analyse.

Une analyse statique a pour but de trouver où se placer dans la chaîne de compilation pour analyser le flot de contrôle du programme et de regarder tous les chemins possibles du code. Nous allons ensuite appliquer une recherche en largeur et en profondeur du Graphe de Flot de Contrôle (CFG) pour rechercher des accès concurrents à la mémoire dans les différentes branches du CFG et être sûr de tout vérifier. Comme le CFG dépend fortement du rang de chaque processus, il est important de souligner que certaines er- reurs de concurrence entre plusieurs processus ne peuvent pas être visibles et donc, pas détectables avec une simple analyse statique. Cela dit, toutes les erreurs potentielles et locales à un seul processus peuvent être examinées et détectées avec cetteanalyse.

Résultats obtenus

Nous avons implémenté le RMA-Analyzer comme étant un outil indépendant qui peut être utilisé pour la vérification des programmes MPI-RMA écrits en C et Fortran. Car, un support pour les codes Fortran a été ajouté.
Nous avons utilisé pour nos tests la machine *Pise* qui appartient à l'équipe MPI et IA distribuée de la R&D de l'entreprise Atos située à Échirolles.

À fin de nous rendre compte du bon fonctionnement du RMA-Analyzer, nous avons développé une suite de tests qui contient toutes les erreurs possibles qui peuvent avoir lieu à cause de différents accès à la même zone mémoire et qui couvre tous les croisements possibles de la matrice d'erreur précédemment décrite. Cette matrice a été réduite pour le cas d'erreurs locales détectables statiquement. Notre outil détecte toutes ces erreurs dynamiquement et statiquement aussi, mais uniquement pour les tests qui contiennent des erreurs locales. L'outil renvoie un retour à l'utilisateur en soulignant le fichier, la ligne et le type d'erreur. De plus, nous avons fait tourner notre outil sur deux grandes applications appelées CFD-Proxy et Nemo.

Nous avons utilisé CFD-Proxy pour l'instrumentation partielle avec les accès MPI- RMA seulement sans les accès locaux load/store, pour mesurer le cout engendré par l'utilisation de l'outil en faisant une comparaison entre les temps d'exécution avec et sans le RMA-Analyzer. Le coût engendré par le RMA-Analyzer sur CFD-Proxy est re- lativement haut, mais raisonnable avec un taux à 40% au mieux sur l'une des variantes de cette application.

Nous avons ensuite testé le RMA-Anlayzer sur Nemo pour une instrumentation glo- bale en prenant en compte cette fois-ci les accès locaux, load/store. Nous avons effectué une comparaison des temps d'exécution sur deux variantes de l'application avec et sans le RMA-Analyzer. Le coût engendré sur l'application Nemo avec une instrumentation générale augmente considérablement à cause des enregistrements des accès load/store.

Nous avons utilisé notre analyse statique sur un code expérimental d'environ 3500 lignes de codes, écrits en C++. Le code est basé sur le benchmark Global Update Ran- domAccess (AKA GUPS). Le code que nous avons utilisé est une version MPI-RMA d'un code existant écrit en UPC++. Notre analyse arrive à détecter les erreurs locales trouvées dans ce code.

Conclusion

Les programmeurs des applications MPI ont besoin d'outils efficaces et fiables à fin de pouvoir détecter des erreurs de programmation. Dans le cadre de cette thèse, nous proposons un outil appelé le RMA-Analyzer qui a pour but d'aider au mieux les pro- grammeurs des applications MPI-RMA à détecter les erreurs liées aux accès concurrents à la mémoire. Le RMA-Analyzer propose une analyse dynamique accompagnée d'une analyse statique pour vérifier les erreurs de concurrence dans des programmes MPI- RMA. La particularité de cet outil est qu'il propose une analyse complète qui détecte toutes les erreurs durant le temps d'exécution du programme et arrête immédiatement le programme dès la première erreur trouvée. Cet outil est le premier à pouvoir rajouter une analyse statique pour détecter statiquement les potentielles erreurs de concurrence locales. L'outil alerte l'utilisateur de la présence des erreurs avec un retour précis de l'endroit et du type de l'erreur.

Comme travaux futurs, nous prévoyons d’améliorer l’outil sur plusieurs axes :

- —Améliorer le coût engendré par l’outil en améliorant l’analyse statique pour pou- voir détecter des erreurs entre processus, et ensuite combiner les deux analyses.
- —Notre outil ne prend pas en compte les synchronisations unilatérales, mais uni- quement les synchronisations globales entre les époques. De ce fait, l’outil peut être amélioré pour prendre les synchronisations inter époques.
- —Nous prévoyons d’intégrer les travaux de cette thèse dans PARCOACH : PARallel COntrol flow Anomaly CHecker. Cet outil est utilisé dans la détection d’erreur liées aux blocages de fonctions collectives dans des programmes MPI,OpenMP, MPI+OpenMP.
- —Nous prévoyaons aussi d’étendre la vérification d’erreurs à d’autres applications PGAS. Comme le modèle de communication unilatéral et le modèle de mémoire basé sur PGAS de MPI-RMA est compatible avec d’autres runtimes et implé- mentations PGAS. Étendre l’outil dans cette direction va aider d’autres pro- grammeurs.

Abstract

Almost all high performance computing applications are written in MPI, which will continue to be the case for at least the next several years. MPI offers one-sided communications which is a well known distributed programming paradigm for high performance computers, as its properties allows for a greater asynchronism and computation/communication overlap than classical message passing mechanisms. In this work, we focus on the Remote Memory Access interface of MPI (MPI-RMA), in which each process explicitly exposes an area of its local memory as accessible to other processes to pro- vide asynchronous one-sided reads, writes and updates. While MPI-RMA is expected to greatly enhance the performance and permit efficient implementations on multiple platforms, it also comes with several challenges with respect to memory consistency. This programming model imposes restrictions with respect to performing asynchronous accesses to shared data. Developers must handle complex memory consistency models and complex programming semantics.

This thesis focuses on detecting memory consistency errors in MPI-RMA programs. We developed an hybrid approach verification tool for MPI-RMA programs that can provide solutions in detecting errors over the space of memory consistency (also known as data races) in MPI-RMA programs.

Our method combines two analyses. First, we perform an on-the-fly analysis to stop the program in case of a consistency violation during runtime. Second, a static analysis support is added to detect local concurrency errors and warn the user of the presence of potential memory consistency errors at compile time.

The experimental results demonstrate that our method is validated on a collection of codes containing errors and on two real applications. Our experiments show that our approach is scalable when running on MPI one-sided applications with an overhead of 40% at best on one of our experiments. We also show on several tests and an MPI-RMA variant of the GUPS benchmark that the static analysis allows to detect such errors on user codes. The error codes have been validated for an integration in the MPIBugs Initiative open-source test suite (MBI).

Key words : High Performance Computing, Hybrid analysis, MPI-RMA

Acknowledgement

Avant de tourner la page de cette magnifique aventure qui est ma thèse et partir pour une nouvelle aventure, je tiens à remercier mes encadrants Emmanuelle et Marc de m'avoir permise de réaliser cette thèse. Je n'oublierai jamais nos moments ensemble.. Biarritz, Bordeaux, Grenoble, Denver, Zurich ou encore l'Ile d'Oléron.. J'ai pris beau- coup de plaisir à travailler avec vous. Merci pour votre suivi, vos conseils avisés, votre précieuse aide et votre soutien.
Un Merci tout particulier à mon directeur de thèse Denis d'avoir su me comprendre et d'avoir été là pour orienter et assurer le bon déroulement de ma thèse. Merci de m'avoir formé à la recherche durant cette thèse, avec beaucoup de patience il faut le dire ! Merci d'avoir su constituer une équipe dans laquelle il fait plus que bon vivre ! J'ai appris tant humainement que professionnellement à tes côtés. Je ne pouvais demander mieux comme directeur de thèse j'ai été très chanceuse.

J'adresse également mes remerciements à mes rapporteurs William Jalby et Laurent Colombet pour la lecture de mon manuscrit et leurs retours constructifs. Merci à Brice Coglin et Corrine Ancourt d'avoir accepté de se joindre à mon jury en tant que président de jury et examinatrice.

Merci à mon comité de suivi de thèse Laurent Simon et Serge Chaumette d'avoir challengé mes résultats lors de nos réunions et aider ainsi à développer mon esprit scien- tifique et répondre mieux aux questions posées lors de mes présentations.

Scott Baden thank you for always teaching me new things and telling me lot of stories from your sabbatical year. Thank you as well for being part of my thesis Jury as guest of honor I'm the one honored by having you. I can't forget all the amazing moments we have had together your pictures are the best ever ..

Merci à toute mon équipe STORM Amina, Nathalie, Marie Christine, Olivier, Sa- muel, Raymond et Pierre André pour votre accueil, les pauses chocolats, nos moments d'échange au restaurant les vendredis ou à la cafette et les nombreuses présentations très enrichissantes. J'adorais parler avec chacun d'entre vous. Raymond tu m'as beaucoup fait rire avec les anecdotes CEA lors de nos nombreuses pauses dans l'openspace.

Merci à tous les doctorants stagiaires et ingénieurs STORM, quelle joie de venir vous raconter mes galères de parking le matin à mon arrivée et de partager des pauses chocolat ou encore des pauses sans intérêt scientifique juste pour faire de la politique avec des débats interminables. Merci à toi Baptiste d'avoir su me distraire tous les ven- dredis pour rédiger zéro phrase car impossible de résister aux pauses avec toi et devoir rédiger le week-end pour rattraper. Merci Chiheb de ton aide pour l'installation logiciel sur plafrim et pour nos nombreuses discussions sur tes entretiens et ton ressenti sur le HPC de tes nombreux conseils pour une carrière réussie après la thèse. Merci Vincent de nous avoir toujours fait rire avec ton projet de ventilation dans l'open space et de nous

avoir fait de belles présentations de ce projet brillantissime qui va surement devenir une belle idée de startup merci de t'être incrusté à pratiquement toutes les pauses STORM ou ailleurs pour nous donner le max d'infos. Merci Gwenolé d'avoir toujours été sage d'avoir toujours défendu avec moi les doctorants contre les nouveaux stagiaires qui nous envahissaient. Merci aussi de m'avoir aidé à ajuster mes figures durant ma rédaction, c'était un plaisir de te voir tous les jours et de t'entendre rigoler très souvent avec nous et surtout avec Maxime. Merci à Maxime d'avoir souvent proposé des plans sorties ou j'ai souvent été absente car j'avais souvent une sortie de prévue, et d'avoir été Mon- sieur chemise du lundi. On s'impatientait d'entendre tes histoires avec tes étudiants et les galères techniques de tes cours. Romain notre voyage à Rome après notre école d'été restera toujours et à jamais dans ma tête. Merci d'avoir été le guide par excellence pour la visite de cette magnifique ville. Radja merci de nous avoir fait rire pendant ton stage avec tes beaux ongles et tes blagues parisiennes merci de m'avoir écouté sur les avantages de la thèse et pas que ... et te voilà doctorant. Je te souhaite bon courage pour la suite. Bastien, merci pour nos échanges passionnants sur comment gagner plus d'argent après la thèse d'avoir toujours été présent pour une pause café à n'importe quelle heure de la journée. Merci Van Man de m'avoir écouté me plaindre sur les inscriptions chères de la thèse durant toutes ses dernières années et merci pour ton résumé de thèse pour mon related work :p Pierre Antoine merci de m'avoir écouté quand je me plaignais des RMA et de m'avoir souvent posé des questions pertinentes sur mon sujet de thèse. Beaucoup de belles choses à toi pour la suite. Merci Thomas pour les discussions intéressantes sur ta formation et tes stages. Gwenolé, Maxime, Bastien, Chiheb, Radja, Romain, Bap- tiste, Thomas, VanMan et Pierre Antoine vous étiez plus que des collègues pour moi vous êtes de vrais bons amis. Je n'oublierai jamais votre soutien pendant ma rédaction de thèse (à part Maxime qui n'était pas dutout rassurant :D) . Je ne pouvais pas oublier de remercier les anciens STORM aussi Adrien et Pierre merci de m'avoir aidé à bien commencer cette aventure en 2019 vous m'aviez dit Célia t'inquiète pas tu vas y arriver à notre toute première pause déjeuner au haut carrée.

Un merci tout particulier à Alice, Diane, et Pélagie pour nos nombreuses discussions rigolades et pauses thée girly. Alice merci d'avoir toujours eu les mots justes pour me dire à quel point je paraissais apaisée et relaxée pendant ma rédaction et que ça te donnait limite envie de rédiger en première année;) tu as toujours su trouver la phrase parfaite au bon moment pour moi. Je vous souhaite à toutes les deux (Alice et Diane) de vivre vos meilleures thèses. Je sais que vous allez faire du bon boulot. Bon courage Pélagie avec ton nouveau travail chez Atos.

Merci à Kun et Mariem mes meilleures amies et collègues d'avoir toujours été là pour moi. Merci de m'avoir encouragé et donné la force pour continuer cette aventure. Merci Mariem pour tes délicieux plats tunisiens et gâteaux, merci de m'avoir fait goûter à tes merveilles. Kun t'avoir à côté de moi dans l'open space quand je m'arrachais les cheveux durant la rédaction me réconfortait tellement. Nos très nombreuses pauses, fous rires sur la terrasse de l'Inria et nos apéros chez moi demeureront sans doutes dans ma tête à jamais merci pour ta gentillesse sans limite. Mariem et Kun je vous aime énor- mément.

À mes très chers parents :
Source de vie, d'amour, de joie et d'affection. Dont le mérite, les sacrifices, les qualités humaines m'ont permis de vivre ce jour. Ce travail et ces longues années d'études c'est aussi grâce à vous deux. MERCI de m'avoir tout donné et de m'avoir tout appris et de m'avoir transmis les bonnes valeurs que je transmettrai à mon tour à mes enfants plus tard. Merci maman pour tes nombreuses nuits blanches pour rester avec moi durant mes révisions de bac scientifique, merci de m'avoir préparé mes plats préférés et de m'avoir toujours encouragé à finir mes études et d'aller le plus loin possible. Merci à toi papa de m'avoir toujours attendu devant notre portail après mes cours au lycée pour faire quelques pas ensemble avant de rentrer à la maison parler d'une nouvelle politique, de ce que j'ai appris de ma journée ou juste rigoler pensemble avant de retrouver les bons plats de maman au déjeuner. Tu m'as tout donné et tu as tout fait pour que je puisse poursuivre mes études en France. Merci de nous avoir aidé à accomplir nos meilleures carrières mes frères, soeurs et moi. Ce travail aujourd'hui je vous le dédie entièrement à vous. C'est moi qui suis fière de vous avoir et pas l'inverse.

À mon tendre et cher mari :
Merci d'avoir été à mes côtés dès le début de cette aventure, de m'avoir si fortement soutenu et de m'avoir donné tant d'amour et de courage pour affronter mes doutes. Tu me donnais tous les jours la joie et la confiance dont j'avais besoin. A notre plus beau jour le jour de notre mariage le 12 juin 2021, à notre inoubliable voyage de noces en République Dominicaine à nos nombreux voyages à deux et celui des Maldives cet été ou j'avais complètement oublié que j'avais une thèse à rédiger tellement on s'amusait comme des petits sur la plage, à nos restaurants et dégustation de plats inconnus, à nos sorties et escapades en amoureux, à chacun des instants vécus à tes côtés à nos innombrables discussions et débats politiques à nos passions la découverte le sport et les voyages ce travail t'est dédié. La vie avec toi m'a tout simplement rendue heureuse et m'a permise d'avancer même quand je pensais que c'était impossible. Merci pour tout. Il me tarde de vivre l'après thèse avec nos futurs projets ensemble (Très nombreux) <3.

À mes deux chères soeurs adorées merci d'avoir été ma source d'inspiration et merci d'avoir été toujours présentes aux plus beaux moments de ma vie et de m'avoir soutenu au moments durs merci de m'avoir donné deux merveilleux neveux et une très belle nièce. La vies est tellement belle avec des soeurs.

À ma famille et belle famille :
Merci de m'avoir fait confiance et de m'avoir toujours encouragé et pour toutes les belles occasions célébrées avec vous.

À mes supers amies Belinda, Alicia, Louiza, Kenza et plus particulièrement à toi Lynda je n'oublierai jamais notre voyage en côte d'Azur et les merveilleux moments passés avec toi .. les fous rires, les jeux de cartes, nos balades dans Paris les week-ends

d’anniversaires .. un pur bonheur merci pour absolument tout.

"Living your dreams takes courage, persistence, and lots of trial and error"

–By me!

CONTENTS

1 Introduction27

1.1 Plan of Study ... 29
1.2 Terms Used ... 30
1.3 Thesis Contributions ... 31

2 Background and Motivation32

2.1 Introduction ... 32
2.2 Partitioned Global Address Space (PGAS) ... 33
2.2.1 PGAS Languages ... 34
2.2.2 PGAS Libraries ... 35
2.2.3 PGAS Data Distribution Model ... 35
2.2.4 PGAS Data Access Model ... 35
2.3 The Message Passing Interface (MPI) ... 36
2.3.1 Point to Point Communication ... 37
2.3.2 Collective Communication ... 39
2.3.3 MPI One-sided Communication with Remote Memory Access (RMA)41
2.4 MPI-RMA programming Overview ... 43
2.4.1 Key Benefits of MPI-RMA Programming ... 45
2.4.2 Comparing MPI-RMA and MPI Two-sided Programming Models.45
2.4.3 Implementation of MPI-RMA Operations ... 46
2.4.4 MPI-RMA Window Creation ... 47
2.4.5 MPI-RMA Data Movement Operations ... 47
2.4.6 Accumulates Purpose ... 49
2.4.7 Synchronization in MPI-RMA ... 50
2.4.8 MPI-RMA Memory Model ... 52
2.4.9 MPI-RMA Operation Ordering ... 55
2.4.10 MPI-RMA Request Based operations ... 56
2.5 MPI-RMA Proprieties and Programming Challenges ... 57

2.6 Conclusion 59

3 Dynamic Data Race Errors Detection in MPI-RMA Programs60

3.1 Introduction 60
3.2 Memory Consistency in MPI-RMA 61
3.2.1 Happens-before Rules 61
3.2.2 Consistency Order Rules 62
3.2.3 Memory Consistency Errors in MPI-RMA Programs (Data Race.62
3.2.3.1 Non-determinism in MPI-RMA 64
3.2.4 Motivating Examples of MPI-RMA Programs 65
3.2.5 Extending MPI-RMA Operations Compatibility 69
3.3 Dynamic Data Race Detection Algorithm 71
3.4 Design and Implementations of the RMA-Analyzer 74
3.4.1 The MPI profiling Interface 74
3.4.2 LLVM Pass 75
3.4.3 Parallel Control Flow Anomaly Checker (PARCOACH) 77
3.4.4 The RMA-Analyzer Framework Overview 80
3.5 Implementation concerns 81
3.6 Experimental Results 82
3.6.1 Experimental Setup 82
3.6.2 Targeted Applications 82
3.6.3 Microbenchmarks 83
3.6.4 Runtime Overhead Impacts of Dynamic verification on CFD-Proxy84
3.6.5 Runtime Overhead Impacts of Dynamic verification on NEMO 87
3.6.6 Memory Consumption Impacts 88
3.7 Conclusion 89

4 Static Data Race Detection in MPI-RMA Programs90

4.1 Introduction 90
4.2 Motivating Examples 91
4.3 Static Detection of Local Concurrency Errors 95
4.4 Local Concurrency Errors Detection Algorithm 95
4.4.1 Example of a Control Flow Graph 97
4.4.2 Proof of the Algorithm 98
4.5 Experimental Results 99
4.6 Conclusion 101

5 Related Work103

5.1 Introduction 103
5.2 Debugging and Correctness Checking Tools for MPI Programs 103
5.2.1 Debugging tools 104
5.2.2 Correctness Checking Tools for MPI Programs 104
5.2.3 Discussion 109

5.3 Data race errors detection Tools for MPI-RMA and Shared Memory Pro- grams 109
5.3.1 Data Race Detection Tools forMPI-RMA Programs................................109
5.3.2 Data Race Detection tools for SharedMemory Programs.................... 111
5.3.3 Discussion...112
5.4 Summary.. 113

6 Conclusion115

LIST OF FIGURES

1.1 MPI One-sided communication bug. 28

2.1 Overview of the Partitioned Global Address Space (PGAS) model. 33
2.2 Comparing the syntax used in CAF, X10, and UPC to access remote data. 36
2.3 Example of asynchronous point-to-point communication using the rendez- vous protocol. 39
2.4 Comparing the syntax of MPI_Put with MPI_Send. 42
2.5 One-sided communication example. Process 1 is not actively involved in the communication 44
2.6 Comparing MPI one-sided and MPI two sided programming Models 46
2.7 MPI-RMA window initialisation operations. 48
2.8 MPI-3 memory window creation variants. 48
2.9 Overview of communication options in the MPI-3 specification from [49] 49
2.10 Examples of Active Target modes where synchronizations are made through Fence functions (on the left) or with Post-Start-Complete-Wait (on the right). 51
2.11 Examples of Passive Target modes where synchronizations are made through Lock/Unlock functions targeting a specific process (on the left) or Lock_all Unlock_all (on the right). 51
2.12 MPI RMA memory model (separate windows). 53
2.13 MPI RMA memory model (unified windows) 53
2.14 Ordered accumulate operations. 55
2.15 Ordering by synchronizations. 56
2.16 An example illustrating an unknown completion of MPI-RMA operations inside an epoch 58
2.17 An example illustrating the lack of ordering when using MPI-RMA oper- ations. 58
2.18 An example illustrating the non atomicity when using MPI-RMA opera- tions. .. 59

3.1 Conflicting two operations in the same program. 63
3.2 Synchronized operations in an MPI-RMA program. 63
3.3 Example of a non-deterministic behavior in RMA programs. 64
3.4 Memory consistency error within an epoch at a single process. 66
3.5 Memory consistency error occurring in several synchronization epochs. 67
3.6 Consistency error between two processes. 68
3.7 Consistency error between three processes. 68
3.8 Consistency error between native Store and MPI_Put across two pro- cesses 69
3.9 Example of memory consistency errors. Dashed edges represent WRITE operations while plain edges represent READ operations on the colored boxes. *O* and *T* respectively indicate the origin and target processes 70
3.10 A simple example of PMPI wrapper counting the number of MPI_Put. 75
3.11 Static compilation using the llvm toolchain. 76
3.12 Overview of the PARCOACH Framework. 77
3.13 Examples of MPI codes containing collectives. 78
3.14 MPI Code 2 functions CFG (left) and the corresponding PPCFG (right). 79
3.15 Error output returned by PARCAOCH used for the code in figure 3.13(a). 79
3.16 Overview of the RMA-Analyzer Framework. 80
3.17 Error output returned by the RMA-Analyzer tool used for the code of Figure 3.7 84
3.18 Runtime overhead of the RMA-Analyzer on CFD-Proxy passive target with tree approaches. CF = Comm Free, PT = Passive Target, NPT = Notified Passive Target. "+ A" means execution time with the RMA- Analyzer. 85
3.19 Runtime overhead of the RMA-Analyzer on CFD-Proxy active target with tree approaches. CF = Comm Free, mpi_sync = Bulk fence synchroniza- tion version, mpi_async = MPI fence with early receives in asynchronous manner. "+ A" means execution time with the RMA-Analyzer. 86
3.20 Runtime overhead of the RMA-Analyzer on NEMO tra_adv kernel. "+ A" means execution time with the RMA-Analyzer. 87

4.1 An example of local memory concurrency errors at origin process *P0* in passive target "Lock/Unlock_all" mode. Bold statements are conflicting memory accesses(Get, Get) at origin side. X = Window memory location of *P0* 91
4.2 An example of local memory concurrency errors at origin process *P1* in passive target "Lock/Unlock_all" mode. Bold statements are conflicting memory accesses (Put,Get) at origin side. Y = Window memory location of *P1* 92

4.3 An example of local memory concurrency errors at origin process *P2* in passive target "Lock/Unlock_all" mode. Bold statements are conflicting memory accesses (Get with Store on **buf**) at origin side. Z = Window memory location of *P2* 92
4.4 An example of local memory concurrency errors at origin *P0* in active target "Fence" mode. Bold statements are conflicting memory accesses (Get with Put) at origin side. X = Window memory location of *P0* 93
4.5 An example of local memory concurrency errors at origin *P1* in active target "Fence" mode. Bold statements are conflicting memory accesses (Get with Load on **buf**) at origin side. Y = Window memory location of *P1* 93
4.6 An example of local memory concurrency errors at origin *P2* in active target "Fence" mode. Bold statements are conflicting memory accesses (Put with Store on **buf**) at origin side. Z = Window memory location Of *P2* 94
4.7 Correction of code4.1 94
4.8 Correction of code4.6 94
4.9 CFG from a benchmark computing a binary tree broadcast algorithm. 98
4.10 Output returned by the analysis on thecode figure4.1 99
4.11 Code snippet from an MPI-RMA versionof GUPS. 101

LIST OF TABLES

2.1 MPI-RMA operation compatibility table when two or more processes ac- cess a window at the same target concurrently in the unified memory model. OVL: Overlapping operations permitted. NOVL: Non-overlapping operations permitted. From [31]. ... 54
2.2 MPI-RMA operation compatibility table when two or more processes ac- cess a window at the same target concurrently in the separate memory model. OVL: Overlapping operations permitted. NOVL: Non-overlapping operations permitted. X: The combination of operations is erroneous. From [31] ... 54

3.1 Compatibility of RMA operations and local load/store accesses on the same address space. O=ORIGIN, T=TARGET, ✓= overlapping is per- mitted,x=undefined behavior, overlapping isnot permitted. ... 69
3.2 Transition table for access types. Given an address with an access type (first row) and a new operation to this address (first column), the table defines the new access type after the operation. We assume there is no data race within a (multithreaded) process. ... 73
3.3 RMA-Analyzer statistics on BST for each application. Memory sizes are in KB... 88

4.1 Compatibility of RMA operations and local load/store accesses on the same address space. ✓= overlapping is permitted,x= undefined behavior, overlapping is not permitted. R = READ, W = WRITE. ... 95
4.2 Results on our microbenchmark suite. ... 100

5.1 List of MPI correctness checking tools ... 109

5.2 List of MPI-RMA memory consistency checking tools 111

LISTINGS

2.1 A Simple MPI program with send receive pattern 37
2.2 An MPI Program Calculating π by Using Collectives 41
2.3 Example of One-sided Communication 43
2.4 Pseudo Code Using Request-based Operations 57
images/scottscode.c 101

LIST OF ALGORITHMS

1 Data Race Errors Detection 72
2 Local Concurrency Errors Detection 96
3 Breadth-first Search 96
4 Analysis of a basic block 97

CHAPTER 1

INTRODUCTION

High-Performance Computing (HPC) has become an important area of progress across a wide range of scientific and engineering disciplines like Computational Fluid Dynamics (CFD) engineering, climate simulation, weather prediction, computational chemistry, bio-informatics and nuclear reactors. High performance computing is the ability to process data and perform complex calculations at high speeds which predicted the increase in frequency of a computer's Central Processing Unit (CPU). Today su- percomputers are becoming faster, cheaper, and more popular, the future growth in computing power will have to come from both the hardware side and the software side. Programmers who are used to think and code sequential software now have to turn to parallel software to achieve the desired performance. Parallel programs can be written by using different programming paradigms. Among them, the Message Passing Inter- face (MPI) [42], and the Partitioned Global Address Space (PGAS) [104] model that are largely used in HPC systems. While MPI and PGAS are often referred as two different programming paradigms MPI is the industry standard communication library for HPC. MPI is the most widely used programming API for writing parallel programs that run on large clusters. The design goals of MPI are flexibility, performance and portability.

MPI accomplishes those goals by providing a very rich semantics that incorporate the features of synchronous and asynchronous systems with several communication modes. Synchronous communications are easy to use and understand in contrast to asynchronous communications which are more complex. MPI introduces a model for asynchronous and remote memory access called MPI-RMA first presented in the MPI-2 specification and then improved and updated in MPI-3. this programming model is quite similar to PGAS, as it is also based on one-sided communications of data, and global access of partitioned memory. Unlike MPI two-sided, where the sender and the receiver explicitly call the send and receive functions, one-sided communications decouple data movement from synchronization and offer asynchronous reads, writes, and updates without involving the target process. MPI-RMA allows efficient data movement between processes with

less synchronizations but the performance and flexibility of MPI-RMA come with several debugging challenges. MPI programs, especially under the presence of non-determinism, are notoriously hard to debug. It thus poses programming challenges to use as few synchronizations as possible, while preventing data race and unsafe accesses without tampering with the performance.

```
MPI_Win_lock_all(win)
MPI_Get(&buf, 1, MPI_INT, 0, 0, 1, MPI_INT, win)
if(buf%2 == 0) /*bug:load/store access of buf */
  buf ++;
...
MPI_Win_unlock_all(win)
```

Figure 1.1 – MPI One-sided communication bug.

To highlight how difficult debugging can get with MPI, we consider a simple MPI- RMA program shown in figure1.1which contains a subtle bug where the one-sided MPI_Get operation is asynchronous, it retrieves the data from the target process and as a result the data may not be ready before the MPI_Win_unlock. Because of this situation the load access of **buf** can retrieve an old value and the store access on **buf** can be overwritten by a value retrieved from MPI_Get. This example illustrates the need for more powerful verification techniques tools than ordinary random testing on a cluster. These tools can help maintaining data consistency in the presence of asynchronous data accesses from multiple processes.
Even though there are many techniques and tools that help developers discover MPI non-determinism errors, they basically fall into one of these three categories: static methods, dynamic methods, and model checking. Static methods have the advantages of being input-independent since they verify the program at the source code level. However, they tend to provide too many false alarms, especially for a large code base, due to the lack of runtime knowledge. Model checking methods are very powerful for small programs in terms of verification coverage but they quickly become impractical for large software due to the infeasibility of building models for such software. Dynamic methods such as testing or dynamic verification are the most applicable methods for large MPI programs since they produce no false alarms and also require little work from the tool users.
Unfortunately, the state of the art in MPI-RMA debugging remains a fundamental chal- lenge and to the best of our knowledge, very few works exist to detect concurrency bugs in MPI-RMA and most of them rely on post-mortem analyses.

This thesis was set up as part of a collaboration between Inria Bordeuax Sud-Ouest and Atos Echirolles. The main objective was to develop a tool that can help the pro- grammer to easily code MPI-RMA applications. This tool should provide solutions to detect memory consistency errors in MPI-RMA applications. We will introduce the RMA-Analyzer, a framework that identifies memory consistency errors in MPI-RMA programs written in C and Fortran. Our framework uses a static and a dynamic on-the- fly analyses and focuses on MPI-3 features. The advantages of our approach are twofold. First, contrary to state-of-the-art solutions, this analysis can detect potential local con- currency errors at compile time. Second, it detects all consistency errors that can happen during the execution of an MPI-RMA program for a given set of input data. When an error occurs, the analysis can directly stop the program, warn the user about the error, and provide detailed information on the conflicting accesses. Indeed, since a silent race condition can provoke errors later in the program, it is mandatory to detect the first race condition and immediately warn the user that an error has happened, instead of waiting until the end of the program to do so. We argue that these two properties are of tantamount importance for helping users porting large-scale code bases on MPI-RMA.

The RMA-Analyzer will be fully integrated into the *PARallel COntrol flow Anomaly CHecker* (PARCOACH) [85]. PARCOACH was developed to provide a combination of static and dynamic analyses to enable an early verification of hybrid HPC applications. This thesis proposes to extend the PARCOACH framework to help and guide the pro- grammer in the development of PGAS programs, in particular by detecting data race errors with a dynamic help on the reality of these issues with precise feedback to the user. It can also provide an optimization of an MPI program into a PGAS or MPI one-sided program.

1.1 Plan of Study

This thesis focuses on data race detection for MPI-RMA programs. We talk about two main pieces. The first part, focuses on dynamic analysis of MPI-RMA programs to detect all the errors that can occur during the execution time of the program. The second part, focuses on a static analysis of MPI-RMA programs that comes as support to the dynamic analysis with the aim of detecting more relevant local concurrency errors before executing the program.

The rest of this thesis is organized as follows: Chapter 2 presents the related background and motivation of this work. Chapter 3 and chapter 4 describe both the corresponding contributions. Chapter 5 discuss the related work to survey some debugging and correctness tools of MPI programs. Chapter 6 concludes the work and discuss the future research directions.

1.2 Terms Used

A **cluster** consists of hundreds or thousands of compute servers that are networked together. Each server is called a **node**which in each cluster work in parallel with each other, boosting processing speed to deliver high performance computing.
A **process** is an entity that encapsulates a private local memory space, i.e., memory that by default is only accessible from within that process. A process may expose parts of its local memory space to other processes, either through shared memory to allow access to other processes on the same node, or by contributing it to a global memory space, in which memory is available to processes executing on different nodes. Inside a process, several threads of execution (or threads) may execute concurrently with shared access to the memory in the local memory space and shared node-local and global memory that the process has access to.
A **thread** typically executes in a local execution context, i.e., local variables and a pri- vate stack to call functions, but has access to global variables shared by all threads in the process. The concurrent execution may be real parallel execution on distinct CPU cores or seemingly parallel through time-slicing performed by the operating system or a user-level runtime library. Thus, threads (in the meaning of the term used in this work) are preemptable, i.e., the operating system scheduler may force the thread to yield the core and schedule another thread or process to execute on it. A **task** typically consists of an action to be executed, i.e., a function call, as well as a set of inputs and outputs. The task's action is executed either directly on the stack of the executing thread or in the context of a user-level thread (ULT), which then contains the execution state throughout the course of execution of the action. The difference between a thread and task is that tasks are typically non-preemptable and a cooperative scheduler relies on the task to eventually complete its execution. A task is a self-contained work-package that transforms a set of input data into a set of output data.
The term **parallel** will be used to describe actions that actually happen at the same time, e.g., two processes executing on two different nodes run in parallel.
A **communicator** : a group of processes. An MPI applications starts always by in- cluding all processes in a default communicator called MPI_COMM_WORLD. New communicators can be created from the MPI_COMM_WORLD. MPI ensures that each communicator is unique thus a communicator can be seen as a system defined tag. Col- lective communications rely on communicators.
A **window** is composed of a group of processes, specified at window creation time by a communicator and a contiguous region of memory at each process and this memory region may differ in size and address.
An **epoch** is the execution span occurring between calls to MPI synchronization func- tions.
The term **concurrent** will be used to describe actions that may happen at the same time, e.g., two independent operations may be executed at the same time. The term **concurrency** on the other hand, describes the maximally possible degree of parallelism, i.e., the set of actions that could be executed at the same time in the absence of resource

restrictions.
The term **runtime** system (or runtime) refers to an entity that is part of the soft- ware stack and typically resides between the application and the operating system or hardware, coordinating operations and resources requested by the application. Task schedulers and MPI implementations are two examples of runtime systems.

1.3 Thesis Contributions

This thesis makes the following contributions :
We present the extended compatibility table to show all the errors that can possibly occur in MPI-RMA programs. Errors are grouped into two categories : local concur- rency errors and remote concurrency errors depending on the process that performs the communication operation (origin or target). We bring forward our data race detec- tion algorithms to discuss our technique that comes with a new hybrid approach. Our method presents two main steps to detect memory consistency errors in MPI-RMA pro- grams. Firstly, at execution time we perform an on-the-fly analysis to collect relevant MPI-RMA operations and load/store accesses. If a data race error is found we stop the program and report a consistency violation. Secondly, we present a static analysis that has been added in order to automate the dynamic analysis. It detects all common local concurrency errors before dynamic analysis takes place.

CHAPTER 2

BACKGROUND AND MOTIVATION

Since the work described in this thesis is largely relying on MPI communication paradigms, with an emphasis on the semantics of MPI one-sided communication, in this chapter we provide related background on MPI communication paradigms so that the reader can better understand the semantics provided by the MPIstandard.

2.1 Introduction

Parallel programs can be written by using different programming paradigms. Among them, the Message Passing Interface (MPI), and the Parallel Global Address Space (PGAS). These paradigms are largely used in HPC systems. While MPI and PGAS are often referred as two different programming paradigms, MPI allows a model for remote memory access called MPI-RMA also called MPI one-sided communication. First in- troduced in the MPI-2 specification, and has updated and seen a major review in the MPI-3 by including several routines in order to handle additional window allocations and add new synchronization modes. The main goal of these modifications is to fully take advantage of the network's low-level remote direct memory access (RDMA) ca- pabilities[31]. This programming model is quite similar to PGAS, as it is also based on one-sided communications of data, and global access of partitioned memory. Unlike MPI two-sided, where the sender and the receiver explicitly call the send and receive functions, one-sided communication decouples data movement from synchronization and offer asynchronous reads, writes, and updates without involving the target process and allow a better overlap of computation with communication. While MPI-RMA allows efficient data movement between processes with less synchronizations, its programming is error-prone as it is the user responsibility to ensure memory consistency. It thus poses programming challenges to use as few synchronizations as possible, while preventing data race and unsafe accesses without tampering with the performance. In this chapter we give the background according to PGAS and both MPI two-sided and one-sided com-

munications. We first, introduce the partitioned global address space and its languages and system-level libraries. Second, we discuss both MPI point-to-point communication and collectives. Finally, we introduce MPI one-sided by giving its overview, and its programming challenges.

2.2 Partitioned Global Address Space (PGAS)

PGAS is a parallel programming paradigm that aims to improve programmer productivity while at the same time targeting for high performance. The main premise of PGAS is that a globally shared address space improves productivity, and that distinction between local and remote data accesses allow performance optimizations and support scalability on large-scale parallel architectures. To this end, PGAS preserves the global address space while embracing awareness of non-uniform communicationcosts.
In PGAS programs, each process exposes a part of its local memory to other processes as shown in figure2.1. This way, the memory of other processes can be directly ad- dressed from a sender, thus allowing to perform one-sided communications (e.g. Put, Get). This communication model is known to significantly improve the asynchronism and the overlap of communications with computations, which is why it is expected to gain focus in the next years for the Exascale era and beyond [7]. Compute Express Link CXL [99] which is a cache-coherent interconnect for processors, memory expansion, and accelerators implements advanced PGAS inter-process communication mechanism. It maintains a unified coherent memory space between the CPU and any memory on the attached CXL device. It is designed to address the growing needs of high-performance computational workloads by supporting heterogeneous processing and memory systems.

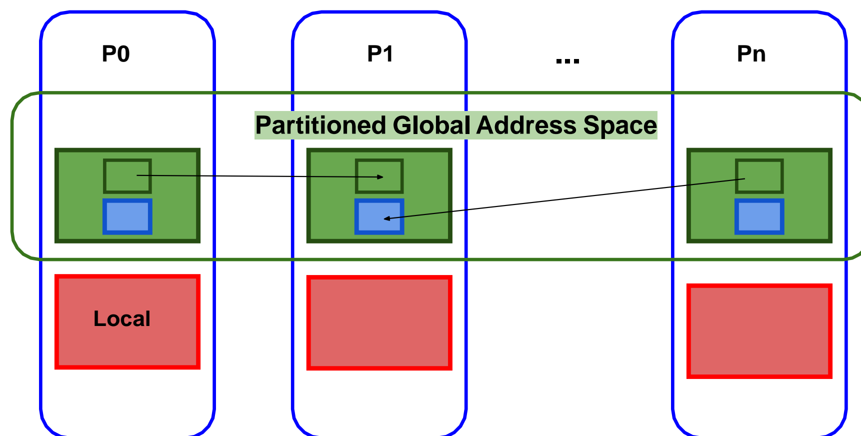


Figure 2.1 – Overview of the Partitioned Global Address Space (PGAS) model.

2.2.1 PGAS Languages

Today, about a dozen languages exist that adhere to the PGAS model here we men- tion:
Co-ArrayFortran (CAF) [73] is a very old PGAS language, however, it remains one of the most known. It is a parallel extension of Fortran 95 which adds the Co-Array as a feature to the language. CAF aims to manage and explicit the cost of accessing remote data asynchronously. CAF fosters the programmer to use the local proprieties of the code. Remote accesses are different from the classic Fortran code.
Titanium[105] is a Java language designed for high-performance parallel scientific com- puting. It was conceived at UC Berkeley and provides implementations for symmetric multiprocessing (SMP) as well as for distributed systems. The language is designed to enable explicit asynchronous parallel programming, while facilitating compiler optimiza- tions for optimal performance. It provides the notion of local and remote references and uses explicit asynchronous communication primitives to exchange data.
Unified Parallel C (UPC) [39] is a PGAS extension of the C language. It integrates features from three proposals: PCP [13], Split C [26], and AC. The specification of the UPC language is provided by the UPC consortium, which consists of academic and government institutions as well as companies. Well-known implementations of the UPC language include Berkeley UPC, GNU GCC UPC, and HP UPC. UPC programs can make use of shared data objects, which is the main PGAS facility of the language. Data values that reside in shared memory are hosted by one of multiple threads but can be accessed asynchronously in a syntactically transparent way from different threads, even though a 'remote access' normally comes at a communication cost.
Chapel[17] is a parallel programming language developed by Cray as part of the Cray Cascade project [16]. It identifies the global view of computation and give the support for both task and data-driven parallelism, besides, the separation of the algorithm and data structure details is the main programmability concepts of the language. Chapel provides concepts for multi-threaded and locality-aware parallel programming. The language also supports many concepts from object-oriented languages and generic programming.
X10 [20] is a programming language developed by IBM Research. The name X10 refers to times 10, the aim of the language to achieve 10 times more productivity in HPC soft- ware development. X10 is described as a modern object-oriented programming language providing an asynchronous PGAS programming model with the goal of enabling scalable parallel programming for high-end computers. X10 extends the PGAS model with asyn- chronicity by supporting lightweight asynchronous activities and enforcing asynchronous access to non-local state. Its explicit fork/join programming abstractions and a sophis- ticated type system are meant to guide the programmer to write highly parallel and scalable code, while making the cost of communication explicit. The task parallelism is implemented on top of a work-stealing scheduler.
Fortress [94] is a programming language designed for high-performance computing, originally developed by Sun Micro-systems. The expressive type system facilitates static analysis, while efficient scheduling of implicitly parallel computations is guaranteed by the work-stealing algorithm. Another characteristic of Fortress is its mathematical syn-

tax. The use of uni-code for instance for the sum operator and Σ the idea of "typeset- ting" code give the language a mathematical look-and-feel. This language is no longer under development.

2.2.2 PGAS Libraries

PGAS terminology is also used to design a system level in the context of a num- ber of communication libraries such as MPI-2 [42], GASNet [9], ARMCI [72] ,GPI [45] and OpenSHMEM [19]. These libraries are used for SPMD programs to store memory segments for remote memory access through one-sided operations such as put, get and accumulate. MPI-3 [38] proposes to fix the MPI-2 RMA API as it does not respond to the needs of application programmers. GASNet is used by Berkely UPC and other PGAS languages, while ARMCI is used by Global Arrays. Theses libraries are not meant to be used by application developers.
UPC++ [6] is also a C++ library that supports PGAS programming, and is designed to interoperate with MPI, OpenMP [18], CUDA [25], ROCm HIP [59] and other HPC frameworks. It leverages GASNet-EX [8] to deliver low-overhead, fine-grained commu- nication, including Remote Memory Access and Remote Procedure Call (RPC) [69].

In the context of this thesis, we will only focus on MPI library with a particular interest in MPI-3 specification.

2.2.3 PGAS Data Distribution Model

What we mean by the data distribution model is how data are distributed in the program. There are two distribution models: in languages where the programmer does not control data distribution thus, the language has implicit model. However, languages where the programmer can specify the data distribution are said to have an explicit model. In addition to this two distribution models, if a PGAS language allows the data to be distributed by remote pointers we say the language supports irregular data distri- bution. However, if a PGAS language allows the distribution of packed data structures, such as arrays or matrices, we say the language supports regular data distribution. Note that for the different types of data distributions (e.g. regular and irregular data) implicit and explicit data distribution may coexist.

2.2.4 PGAS Data Access Model

Data access model describes how data can be distributed, represented, declared and accessed. The data access model is part of data distribution model. The main aspects for data accesses are how a distributed data is accessed and how access to remote data is performed. The remote data access to shared data can be either implicit or explicit. Remote data access is explicit when accessing remote data requires devoted syntax.
The syntax of the remote data accesses can either imply how the data is retrieved like in CAF (the co-array syntax is shown in figure2.2(a)) or a migration of the computation

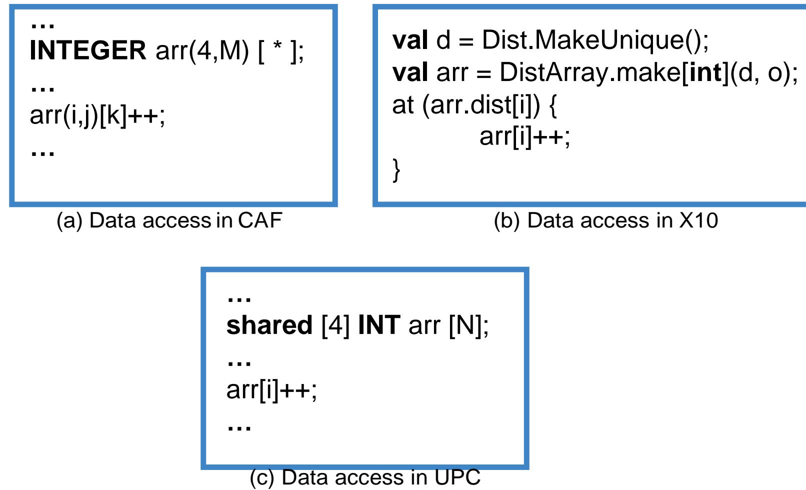


Figure 2.2 – Comparing the syntax used in CAF, X10, and UPC to access remote data.

to the place where the data is, like in X10 (as shown in figure2.2(b)). Otherwise if remote data access is syntactically transparent like in UPC (like in figure2.2(c)), the PGAS language supports implicit access of remote data.

2.3 The Message Passing Interface (MPI)

The message passing interface MPI [44] is a library interface. MPI is designed to help programmers write high performance parallel message passing programs that are scalable and portable. MPI is the defacto standard for writing parallel programs run- ning on large clusters in order to achieve higher performance by providing a very rich semantics that incorporate the features of both asynchronous and synchronous systems. A description of an MPI program can be found in the MPI standard [44]. MPI programs are written in C, C++ and Fortran. There are many different implementations [67,76, 43] that follow the specific instructions given by the standard.
Note that MPI supports several modes such as non blocking communication, non deterministic receives and collective calls. These modes are largely used in order to provide higher performance and portability.

2.3.1 Point to Point Communication

The point to point communication is the basic process of sending and receiving mes- sages between different MPI processes. By using MPI point-to point communication one process performs a send operation MPI_Send while the other performs a matching re- ceive operation MPI_Recv involving a synchronization between peers as shown in listing 1. With this kind of communication MPI guarantees that every message will be received without errors. Errors as deadlocks often occur when the send and receive operations do not match. Deadlock means that neither the sending process nor receiving process can perform an operation until the other completes its action. Note that MPI_Send and MPI_Recv use a blocking methods for transferring messages between two MPI pro- cesses. The term Blocking means that the sending process waits until the whole message has been correctly sent to the receiving process, and the receiving process waits until it correctly receives the complete message. More complex communications methods can be built upon these two methods we define synchronous point-to-point communication and non-blocking point-to-point communication.

```
1   #include<stdio.h>
2   #include<mpi.h>3
4    int main ( int argc , char ** argv ) {
5           int my_rank, cluster_size , message_item ;
6           MPI_Init(&argc , &argv ) ;
7           MPI_Comm_size(MPI_COMM_WORLD, &cluster_size ) ;
8           MPI_Comm_rank(MPI_COMM_WORLD,&my_rank) ; 9
10          if(my_rank == 0){
11                 message_item = 4 2;
12                 MPI_Send(&message_item , 1 , MPI_INT, 1 , 1,
13                               MPI_COMM_WORLD);
14          printf("Message  Sent : %d\n" , message_item ) ; 15        }
16
17          else if ( my_rank ==1 ) {
18                 MPI_Recv(&message_item , 1 , MPI_INT, 0 , 1,
19                               MPI_COMM_WORLD,MPI_STATUS_IGNORE);
20          printf("Message  Received:%d\n" , message_item ) ; 21    }
22
23          MPI_Finalize() ;
24   return 0 ;25
     }
```

Listing 2.1 – A Simple MPI program with send receive pattern

Synchronous Point-to-Point Communication is the synchronous blocking send form of MPI point-to-point communication. While MPI_Send has the same blocking behavior as MPI_Ssend, according to the MPI standard, its behavior is asynchronous because the call can return before a matching receive is posted. Synchronous point-to- point communication as its name indicates needs the implementation of some rendez-vous

protocols for synchronization between peers where the receiver blocks until it starts to receive data from a matching sender. In the case of a synchronous send, the sender should block until it receives an acknowledgement from the receiver that has started the receiving process.
This type of communication are used because it presents several advantages: it is easier to use and understand compared to non-blocking communication(detailed as well in this section), which allows higher productivity and the programmer can easily provide correct programs by using this type of communication. It also helps prevent memory exhaus- tion by not requiring the MPI runtime to provide message buffer. However, synchronous communication usually comes with performance penalty due to the synchronization, especially for applications that use large messages. In order to address this problem, MPI offers two alternatives: buffered communication and asynchronous communication. Buffered communication allows the process to issue a sending request and continue pro- cessing without waiting for the acknowledgement from the receiver. MPI programmers can take advantage of buffered communication through one of these two methods: first, by allocating an explicit buffer and provide it to the MPI_Bsend call through the use of MPI_Buffer_attach. Note that the user can only attach one buffer per process and the buffer can be used for more than one message. MPI_Buffer_detach can be called later to force the reception of all messages that are in the buffer. Second, it takes advantage of the MPI runtime's buffer through the use of MPI_Send. The user buffer is available immediately after the call since the data has been copied into the runtime's buffer. It is generally not recommended to rely on the runtime to provide such a buffer because MPI standard does not mandate that the implementation should provide any buffer. If the runtime runs out of buffer space due to excessive pending communication, MPI_Send will block until more buffer space is available, or until the data has been transmitted on the receiver's buffer.
For this type of communication MPI offers the matching MPI_Ssend which is quite similar to the MPI_Send. There is a small but important difference between the two. MPI_Ssend does not return until the message has been received at the receiver side even for small messages.

Non-blocking Point-to-point Communication unlike synchronouspoint-to-point communication that offers robustness and predictability of message delivery at the ex- pense of program flexibility and performance. Many applications show their interest in performing communication-computation overlap. The overlap of computation with communication would allow having the ability to issue some communication requests, continue with local processing, and process the results of those requests when the com- putation phase is done, with the hope that the MPI runtime has sent/delivered the message during the computation phase. This could be very beneficial for applications to gain in computation time. For these reasons MPI offers non-blocking point-to-point communication through the use of calls such as MPI_Isend and MPI_Irecv. As shown in figure2.3the process would provide a buffer, issue the call, obtain a request handle from the runtime, and wait for the communication request to finish later using either

MPI_Wait or MPI_Test or their variants MPI_Waitall or MPI_Waitany. Before MPI-2 the process cannot access the buffer while the requests are still pending. In MPI-2.2 and higher this has changed to only read sending requests that are pending with no-access for the receiving requests that are pending.
Similarly to MPI_Send, the MPI runtime use the buffering technique for the messages sent by MPI_Isend and thus, the call to MPI_Wait simply states that the data have been copied to the runtime's buffer and the process can now reuse the buffer associated with the MPI_Isend. Those applications that require a rendezvous semantics for such situations will have to use MPI_Issend where the corresponding MPI_Wait will block until the receiver has started to receive the data.

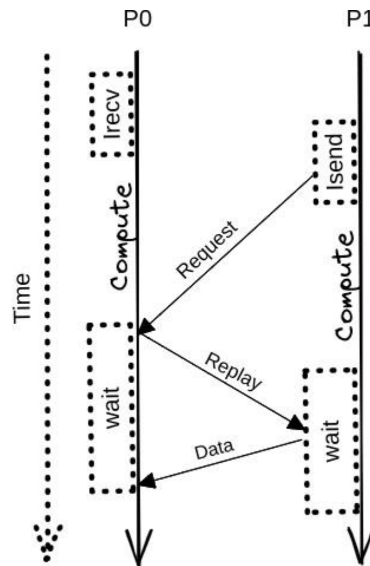


Figure 2.3 – Example of asynchronous point-to-point communication using the rendez- vous protocol.

2.3.2 Collective Communication

As the name suggests, collective communication refers to MPI functions that require the participation of all processes collectively within a defined communicator. We can see the collective communications as a set of point-to-point operations. We give the exam- ple of MPI_Bcast call which can be decomposed into multiple MPI_Send calls from the root to all other processes in the communicator and multiple MPI_Recv calls from the other processes to receive the data from the root. In general, collective operations are heavily optimized by most implementations depending on the size of the messages and the network structure. For example, the MPI_Bcast call can use a tree-based algorithm to broadcast the message efficiently.
While it is intuitive for developers to consider collective operations as having synchroniz-

ing behavior, the implementation is often not required to provide such semantics. There are only a few collective calls that have synchronization semantics such as MPI_Barrier while the rest are only required to block until they have fulfilled their roles in the col- lective operation. For example, for an MPI_Reduce call, after a process has sent out its data to the reducing root, it can proceed locally without having to wait for the root to receive all messages from other processes. However, the MPI standard does require that all processes in the communicator execute the collective.
Although, the collective operations are the most used, they also have concerns related to synchronization semantics, but these are not the same issues that we deal with in the context of thesis. An example of code evoking collective communication is given in listing 2 that shows how to calculate π by using MPI_Bcast and MPI_Reduce.

```
1 #include  "mpi.h"
2 #include  <stdio .h> 3
#include <math .h>
4
5 int main ( int argc ,  char ∗ argv [ ] ) 6 {
7      int n , myid , numprocs , i ;
8      double PI25DT = 3 .141592653589793238462643 ;
9      double mypi , pi , h , sum , x ;
10     MPI_Init(&argc ,&argv ) ;
11     MPI_Comm_size(MPI_COMM_WORLD,&numprocs ) ;
12     MPI_Comm_rank(MPI_COMM_WORLD,&myid ) ;
13     while ( 1 ) {
14         i f ( myid ==0 ) {
15             printf ("Enter  the  number of intervals : (0  quits )  ") ; 16          scanf
("%d",&n) ;
17         }
18         MPI_Bcast(&n , 1 , MPI_INT,  0 , MPI_COMM_WORLD); 19        i f ( n ==
0 )
20             break;
21         else  {
22             h      =  1.0 / (double) n;
23             sum = 0 . 0 ;
24             for  ( i = myid + 1 ;  i <= n ;  i  += numprocs )  {  25
               x = h∗ ( ( double ) i − 0 . 5 ) ;
26                 sum  += (4.0 / (1.0  +  x∗x)) ;
27             }
28             mypi = h ∗ sum;
29             MPI_Reduce(&mypi , &pi , 1 , MPI_DOUBLE, MPI_SUM, 0 ,
30                         MPI_COMM_WORLD);
31             i f ( myid ==0 )
32                 printf ("pi  is  approximately   %.16f ,  Error  is  %.16f\n" ,
33         pi ,  fabs(pi  −  PI25DT)) ; 34 }
35     }
36     MPI_Finalize () ;
37     return 0 ;38
}
```

Listing 2.2 – An MPI Program Calculating π by Using Collectives

2.3.3 MPI One-sided Communication with Remote Memory Access (RMA)

MPI standard offers in addition to two-sided synchronous and non-blocking point- to-point communication the one-sided fashion also called MPI remote memory access (MPI-RMA). MPI-RMA performs one-sided, asynchronous and direct accesses to the processes memory location that are part of a given communicator. It permits to one- sidedly send data to a memory location of another MPI process without any buffering technique. It uses MPI_Put which can be seen as executing a send by the origin process and a matching receive by the target process. The difference with the send in two-

sided communication is that all arguments are provided by the origin process in a single routine as shown in figure2.4. It also uses an MPI_Get which is the read equivalent of MPI_Put.
In MPI-RMA the synchronization is decoupled from the communication but one still needs global or local synchronizations to perform the communication within epochs. As we can see in the example of listing 3, where the MPI_Put is performed between two synchronization calls by using MPI_Fence.

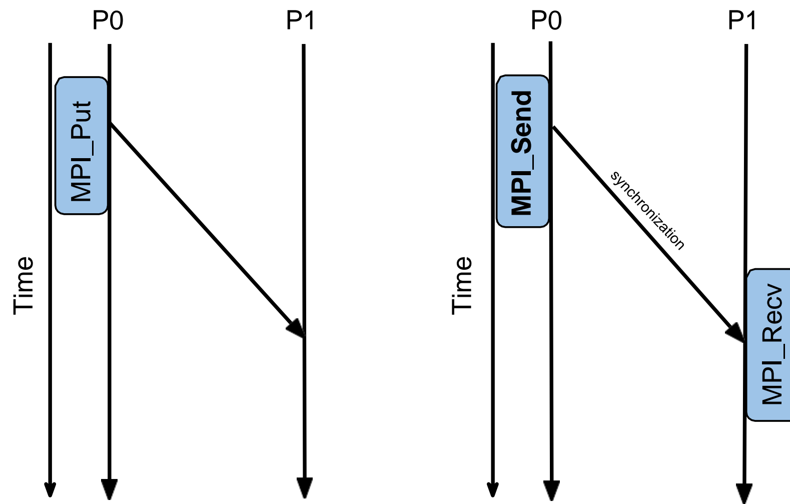


Figure 2.4 – Comparing the syntax of MPI_Put with MPI_Send.

```
1        #include <stdio .h>
2        #include <stdlib .h>
3        #include <mpi.h>
4         int main ( int argc , char ∗ argv [ ]) 5    {
6                 MPI_Init(&argc , &argv) ;
7                 int comm_size ;
8                 MPI_Comm_size(MPI_COMM_WORLD,&comm_size )
9                 /∗ This application needs two MPI processes ∗/
10                if ( comm_size != 2 )
11                {
12                        printf ("2 MPI processes , not %d.\n" ,comm_size) ;
13                MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE) ; 14            }
15                int my_rank ;
16                MPI_Comm_rank(MPI_COMM_WORLD,&my_rank) ;
17                int window_buffer = 0;
18                MPI_Win window ;
19                MPI_Win_create(&window_buffer , sizeof(int) ,sizeof(int) ,
                               MPI_INFO_NULL, MPI_COMM_WORLD, &window ) ;
20                MPI_Win_fence(0 , window) ;
21                if(my_rank == 0)
22                {
23                        int my_value = 12345;
24                        MPI_Put(&my_value ,1 ,MPI_INT,1 ,0 , 1, MPI_INT, window) ;
25                        printf (" [MPI process 0] I put data %d in MPI process 1 window via MPI_Put.\n" ,
                                   my_value) ;
26                }
27                MPI_Win_fence(0 , window) ;
28                MPI_Win_free(&window) ;
29                MPI_Finalize () ;
30        return EXIT_SUCCESS; 31          }
32
33
```

Listing 2.3 – Example of One-sided Communication

Since the focus of this thesis is directly related to MPI one-sided communication, the following sections of this chapter will be dedicated to MPI-RMA with an overview and more detailed definitions.

2.4 MPI-RMA programming Overview

The MPI-3 standard, introduced in September 2012 includes a significant update to the one-sided communication interface. In particular, the interface has been extended to better support popular one-sided and global-address-space parallel programming mod- els, to provide better access to hardware performance features, and to enable new data access modes. By using MPI-RMA the target process is not actively involved in the communication as shown in the example of figure2.5. Unlike non-blocking two-sided point to point communication here the communication operations are issued without

using additional buffers.

MPI disassociates the memory synchronization or consistency and the process synchro-

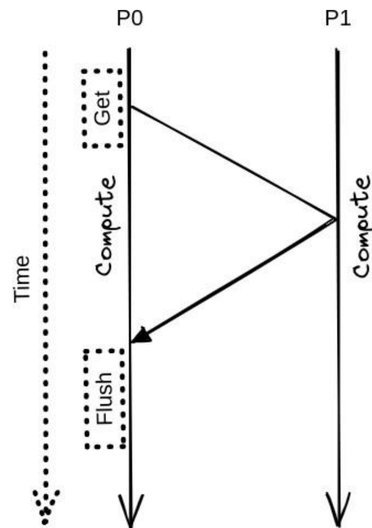


Figure 2.5 – One-sided communication example. Process 1 is not actively involved in the communication.

nization. Furthermore, such synchronization can be non-blocking.

Consistency, completion, and synchronization are the key points in MPI-RMA program- ming. MPI provides them as separate concepts and allow the user to reason about them separately. MPI-RMA programming is thus slightly more complex because of complex interactions of synchronization and communication operations. In MPI-RMA program- ming model all communication operations are non-blocking. It means that the commu- nication functions may return before the completion of operations and bulk synchro- nization functions are used to complete before the initiated operations. Like most RMA programming models, it allows the programmer to initiate operations asynchronously and complete them (locally or remotely) later in order to allow the communication- computation overlap.

All these features make using MPI-RMA programming model more beneficial and it permits to enhance the performance of application that use one-sided communication pattern. All at once, these features also make reasoning about MPI-RMA program semantics much more complex and it comes with complex programming environment which is often hard and not suitable for domain-scientists programmer. It the responsi- bility of high-level libraries writers to provide domain-specific extensions that can hide the complexity.

2.4.1 Key Benefits of MPI-RMA Programming

As MPI-RMA programming model enables direct access to remote memory through the network interface. It allows a better overlap of computation with communication by providing asynchronous direct reads, writes and updates. It provides a remote memory access and bypasses the operating system and the CPU, enabling low latency andhigh bandwidth. Since the hardware implementation in the network card is a simple set of queues at the lowest hardware level, RMA networks are programmed through user- level libraries such as MPI, ARMCI and GASNet that directly communicate with the hardware. The rule of these libraries is to provide calls to read and write or update the remote memory location and give several modes of synchronization that a given program can use to handle the communication through peers. The RMA programming systems are similar to shared memory systems however, RMA systems do not offer atomicity by default and the global address space is partitioned such that each network endpoint owns a fixed address range. Furthermore, a study [31] shows that MPI-3 RMA enhances the performance and the usability of MPI-RMA. This study also provides a complete implementation of MPI-3 RMA and its performance optimization opportunities. In addition, As RMA technology is widely supported, it is available for InfiniBand [84], Blue Gene/P [2], Blue Gene/Q [22], IBM PERCS [5], and Cray's Gemini and Aries networks [4,35]. RMA technology is largely used by several programming systems like UPC, Co-array Fortran and MPI-3 which is our main focus in thisthesis.

2.4.2 Comparing MPI-RMA and MPI Two-sided Programming Mod- els

One-sided communication offers several advantages to the programmer. One-sided communication reduces synchronization while two-sided communication implies some degree of synchronization. A receive operation cannot complete before the respective send has started. Further, because a sender cannot usually write in the receiver's address space. This type of communication is based on buffering technique, it uses buffers to transfer data from the sender's side to the receiver's side. If a buffer is small compared to the data volume, additional synchronization should occur and the sender must wait for the receiver to free up the buffer. With one-sided communication there is no need to specify a synchronization at the same time as issuing communication operations, it uses a single synchronization over many data movements and thus, it decouples data movement from synchronization.
As shown in figure2.6, one-sided communication offers computation-communication overlap and reduces data movement, in comparison with two-sided communication that introduces intermediate buffering and rendez-vous protocols. Two-sided communication incurs extra data movement compared to the case where processes write directly to another process's address space. Furthermore, one-sided communication can simplify programming because the information about a data transfer must be known and specified on only the sender side instead of both sender and receiver.

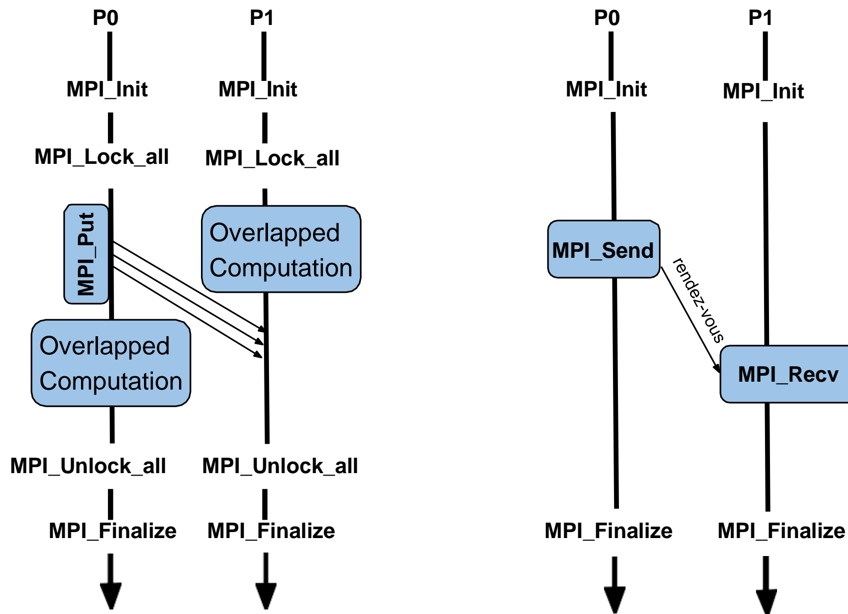


Figure 2.6 – Comparing MPI one-sided and MPI two sided programming Models.

2.4.3 Implementation of MPI-RMA Operations

There are different implementation choices for MPI-RMA operations. Two exiting open-source MPI implementations support MPI-RMA: MPICH [43] and Open MPI [37].

HW-based operation Implementation is supposed to provide higher performance, but, it comes with some restrictions because this implementation only offers simple RDMA communication such as writing and reading contiguous data. It does not sup- port complex communication like accumulating non contiguous datasegments
(e.g. MPI_Accumulate).

AM-Based operation Implementation this is different from the previous one be- cause when an origin process preforms a one-sided operation by using MPI_Isend the target process receives a set of messages with a receiving progress.
As HW-based operation comes with some limitations, MPI implementations combine both HW-based and AM-based implementations. We mention that the remote mem- ory access between processes on the same node is implemented as HW-based operations because it uses direct memory accesses to a shared-memory region.

2.4.4 MPI-RMA Window Creation

All MPI-RMA communication operations occur in a window. A window in MPI- RMA is represented by a group of processes, stated at window creation time by a com- municator and a contiguous region of memory at each process. The memory region at each process can be different in address and size for each process. MPI window creation is a collective and synchronous operation over the given communicator that occurs by calling MPI_Win_create or other window creation routines that will be described here. In addition to a communicator, buffer pointer, and buffer size parameter, there is a 'dis- placement unit' and an optional set of MPI_Info hints which can be specified to enable potential optimizations by the MPI implementation.

One-sided communications can only be used in the exposed memory in the window. MPI-3 specifies three new window creations: we cite here the allocated windows, dy- namic windows, and shared-memory windows. MPI allocated windows are created by using MPI_Win_allocate in comparison with MPI-2 windows creation where the user provides the window buffer, here MPI allocates the buffer for this window. Windows creation are done collectively in the communicator, each process associates only one memory region with the window. This is an issue for the applications that need a dynamic allocation and deallocation of memory. This is why MPI-3 provides a new dy- namicwindowcreationasshownin figure2.7whichcreatesawindowcollectivelywithout initiating the associated memory. Memory can thus be asynchronously attached to or detached from this window by a given process by using MPI_Win_create_dynamic, MPI_Win_attach, and MPI_Win_detach.

MPIalso providesthe MPI_Win_allocate_shared routine which is used to create a new shared memory window, it allows processes to allocate a shared memory segment and then be used for the shared memory programming. It allows programmers to do hybrid programming by combining MPI with OpenMP or threading libraries. Additionally, MPI_Win_sync is used to synchronize load/store operations to minimize the overhead compared to a full window synchronization. The shared memory windows supply pro- grammersacompleteandportable inter-processsharedmemoryprogrammingsystemby using MPI-RMA synchronization and atomic operations. Figure2.8shows an overview of three different versions of windows creation.

2.4.5 MPI-RMA Data Movement Operations

MPI-RMA offers the basic data-movement operations put and get and additional atomic operations called accumulates. MPI provides six types of data movement opera- tions. MPI_Put, MPI_Get, MPI_Accumulate, MPI_Get_accumulate, MPI_Fetch_and_op, and MPI_Compare_and_swap.

MPI_Put transfers the data from the origin process to the target process. MPI_Get transfers the data from the target process to the origin process.

MPI_Accumulate add the data from the origin process to the data on the target process. MPI_Get_accumulate also does the same as MPI_Accumulate but it re- turns the original value of the target data. MPI_fetch_and_op is a custom case of

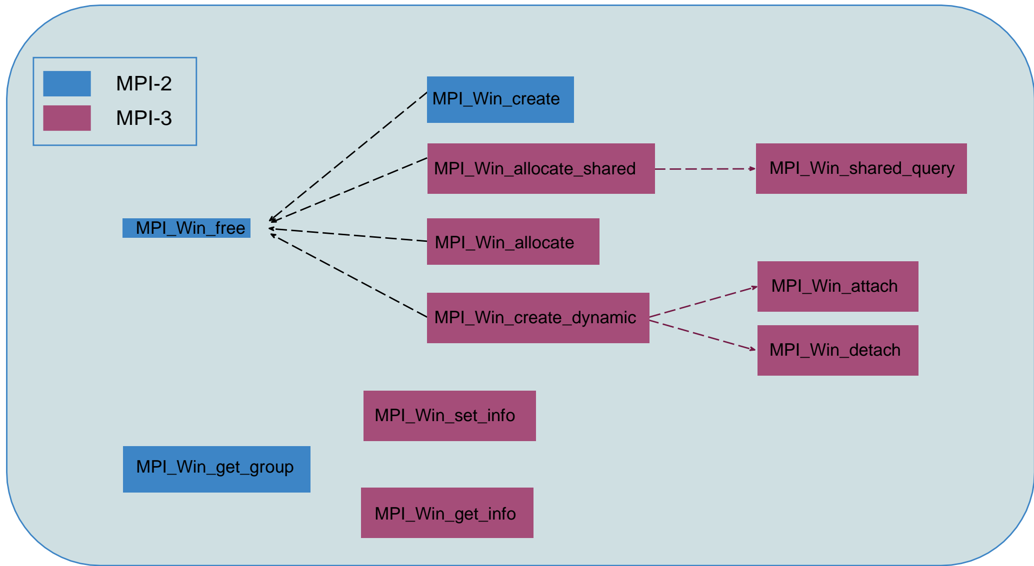


Figure 2.7 – MPI-RMA window initialisation operations.

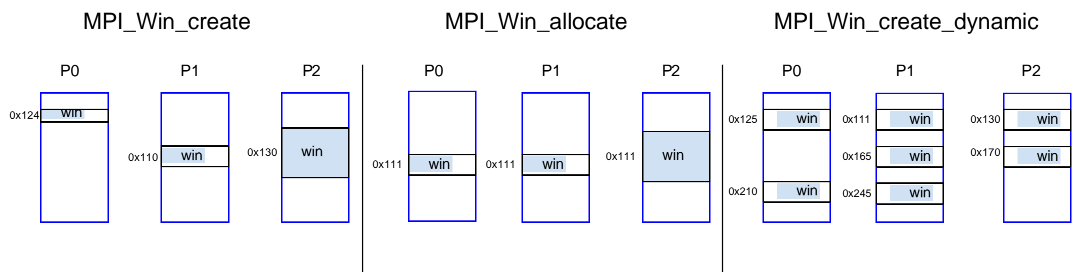


Figure 2.8 – MPI-3 memory window creation variants.

MPI_Get_accumulate that accumulates only one basic element from the origin process to the target process. Those MPI_Accumulate-like operations do computation to accu- mulate both the origin and the target data. MPI_Compare_and_swap transfers one basic element from the origin process to the target process by transferring two elements the origin element and the compare element. It first compares the "compare" element with the target element, if the two elements are equal then, it replaces the target element with the origin element and returns the original target value.
The three data movement operations MPI_Get_accumulate, MPI_Fetch_and_op and MPI_Compare_and_swap perform the "read-modify-write" type of one sided operation between the origin process and the target process.

In the context of this thesis we focused only on MPI_Put, MPI_Get and MPI_Accumulate. All these data movement operations are non-blocking. We see on the right of figure2.9 an overview of communication options in the MPI specification.

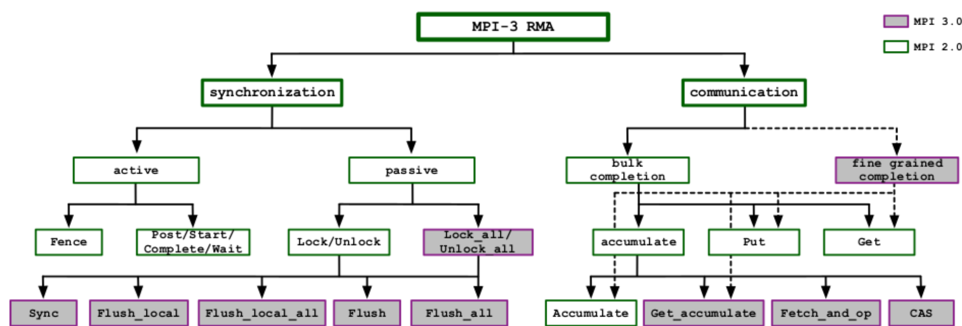


Figure 2.9 – Overview of communication options in the MPI-3 specification from [49].

2.4.6 Accumulates Purpose

MPI-RMA provides accumulate operations in addition to the basic data transfer op- erations. Accumulate-like operations are designed to take full advantage of the hardware support. By using accumulate-like operations the overlapping conflicting accesses are allowed only if the basic types are identical. MPI requires a specification of ordering and by providing these operations, it offers strict ordering by default, which is most conve- nient and easy to use for programmers. However, one should know that accumulate-like operations come with a cost and can worsen the performance because atomic updates are much harder to achieve than basic data transfer with put and get. A programmer can write its program with no ordering. Accumulates can also be adopted as atomic put or get if overlapping accesses are mandatory. The MPI_Get_accumulate with the operation "no op" acts like an atomic read, and MPI_Accumulate with the operation "re- place" acts like an atomic write. However, according to [49] the atomicity is guaranteed only at the level of each basic datatype. In other words, if two processes use "replace" to

perform two simultaneous accumulates of the same set of two integers (either specified as a count or as a datatype), the result may be that one integer has the value from the first process and the second integer has the value from the second process and thus, the result may be false.

2.4.7 Synchronization in MPI-RMA

In MPI-RMA all communication operations are asynchronous and performed in epochs. An epoch is defined by synchronization operations and forms a unit of com- munication. By the end of an epoch all the initiating communication operations are completed locally at the origin process and remotely at the target process. Epochs can be also seen as access and exposure epoch. In an exposure epochs the target process exposes its local window memory in order to be accessed remotely by other processes in the communicator, however, an access epoch is used to access the remote memory of another process and thus performed by origin processes. Note here that a process can be simultaneously in access and exposure epochs.
MPI offers two main synchronization modes based on the involvement of the target pro- cess: active target synchronization and passive target synchronization. In active target synchronization, the target processes expose their memory in exposure epochs and thus participate in process synchronization. In passive target synchronization, the target pro- cesses are always in an exposure epoch and do not participate in synchronization with the accessing processes. Each mode is targeted at different use cases. Active target synchronization supports bulk-synchronous applications with a relatively static commu- nication pattern, while passive target synchronization is best suited for random accesses with quickly changing target processes.

Active Target Synchronization MPI offers two modes of active target synchroniza- tion: fence and general post-start-complete-wait. In the fence synchronization mode, all processes associated with the window call fence and advance from one epoch to the next. Fence epochs are always both exposure and access epochs. This type of epoch is best suited for bulk synchronous parallel applications that have quickly-changing access patterns, such as many graph-search problems [21]. In Post-start-complete-wait mode, active target synchronization, processes can choose to which other processes theyopen an access epoch and for which other processes they open an exposure epoch. Access and exposure epochs may overlap. This method is more scalable than fence synchronization when communication is with a subset of the processes in the window, since it does not involve synchronization among all processes. Exposure epochs are started with a call to post (which exposes the window memory to a selected group) and completed witha call to test or wait (which tests or waits for the access group to finish their accesses). Access epochs begin with a call to start (which may wait until all target processes in the exposure group exposed their memory) and finish with a call to complete. The groups of start and post and complete and wait must match; that is, each group has to specify

the complete set of access or target processes. This type of access is best for compu- tations that have relatively static communication patterns, such as many stencil access applications [27]. Figure2.10shows an example of both active target modes.

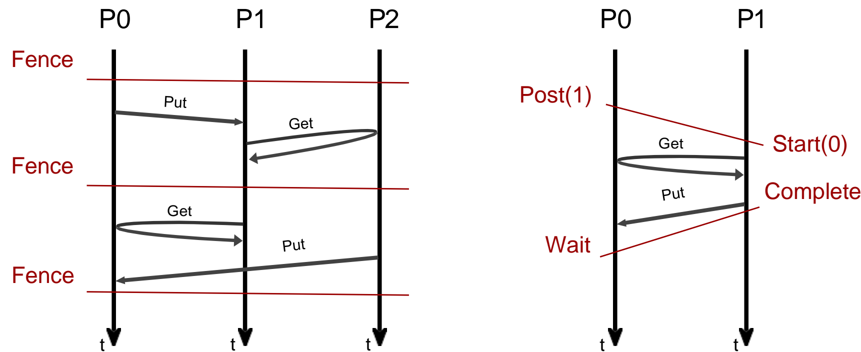


Figure 2.10 – Examples of Active Target modes where synchronizations are made through Fence functions (on the left) or with Post-Start-Complete-Wait (on the right).

Passive Target Synchronization The concept of exposure and access epochs is not relevant in passive mode, since all processes always expose their memory and can ac- cess other memory locations. This feature leads to arbitrary access to different memory locations but also potentially to improved performance. Passive mode can be used in two ways: single-process lock/unlock as in MPI-2 and global shared lock accesses since MPI-3 as we can see in on the left of figure2.9that shows the main changes in MPI-3.

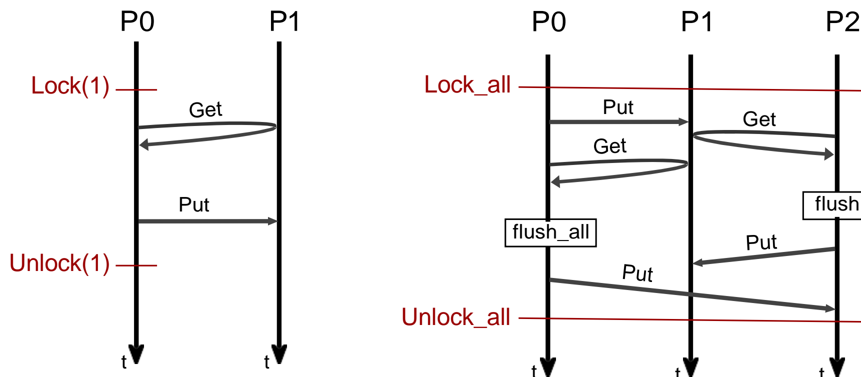


Figure 2.11 – Examples of Passive Target modes where synchronizations are made through Lock/Unlock functions targeting a specific process (on the left) or Lock_all Unlock_all (on the right).

In the single process lock/unlock model, a process locks the target process before ac- cessing it remotely. There are two types of locks defined in the passive target mode: (a)

MPI_LOCK_SHARED: multiple process can acquire a lock on the same target con- currently; (b) MPI_LOCK_EXCLUSIVE: only one process can acquire a lock on the target process in order to avoid conflicts with local accesses, a process may lock its local window exclusively. Exclusive remote window locks may be used to protect conflicting accesses and avoid data-race errors. The lock function itself is a non-blocking function it does not need a wait for the lock to be acquired. In the "lock_all unlock_all" model, each process starts a lock_all epoch (it is by definition shared) to all other processes. Processes then communicate via MPI-RMA operations to read write and update data by using MPI-RMA communication operations or synchronization operations. Figure2.11 shows an example of a lock_all epoch with several communications and flushes. MPI also allows mixing both models freely.

2.4.8 MPI-RMA Memory Model

MPI provides two memory models for the remote memory access the separate mem- ory model and since MPI-3 the unified memory model. These memory models are used to better support different applications and non-coherent systems and offer more pro- gramming flexibility. The MPI standard separates the window into a private and public copies. Many systems offer either a public memory region that is globally addressable by all processes or private buffers that are local to each process which is used for local load and store accesses. Local buffers are used primarily to store the copied data from the main memory. Besides, this local buffers are either coherent where all updates to main memory are reflected in all private copies consistently, or non-coherent, where conflicting accesses to main memory need to be synchronized and updated in all the private copies. Consistent systems allow for direct remote memory updates without any remote partic- ipation. On the other hand, non-coherent systems must call RMA functions to reflect public window updates into their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non- coherent case. MPI thus differentiates between two memory models called RMA unified, if public and private window are logically identical, and RMA separate, otherwise.

Furthermore, local load and store accesses always access the local window copy while the remote put, get and accumulates access the public window copy. Figures2.12and 2.13show the two memory models.

In the separate memory model in order to handle memory consistency the the remote updates are handled by the public copy whereas, load and store accesses are managed by the private copy. Note here that the lock/unlock and sync synchronize the contents of the two copies for a local window. As described in [49] the semantics don't say that windows have to be separated, just that they can be separated. In other words, remote updates may also update a private copy. However, the rules in the separate memory model ensure that a correct program will always observe the memory in a coherent manner. Those rules force the programmer to perform separate synchronizations. In contrast, the unified memory model relies on hardware in order to manage memory co- herency. According to Hoefler et al, the unified memory model assumes that the private and public copies are identical and the hardware automatically spreads updates from

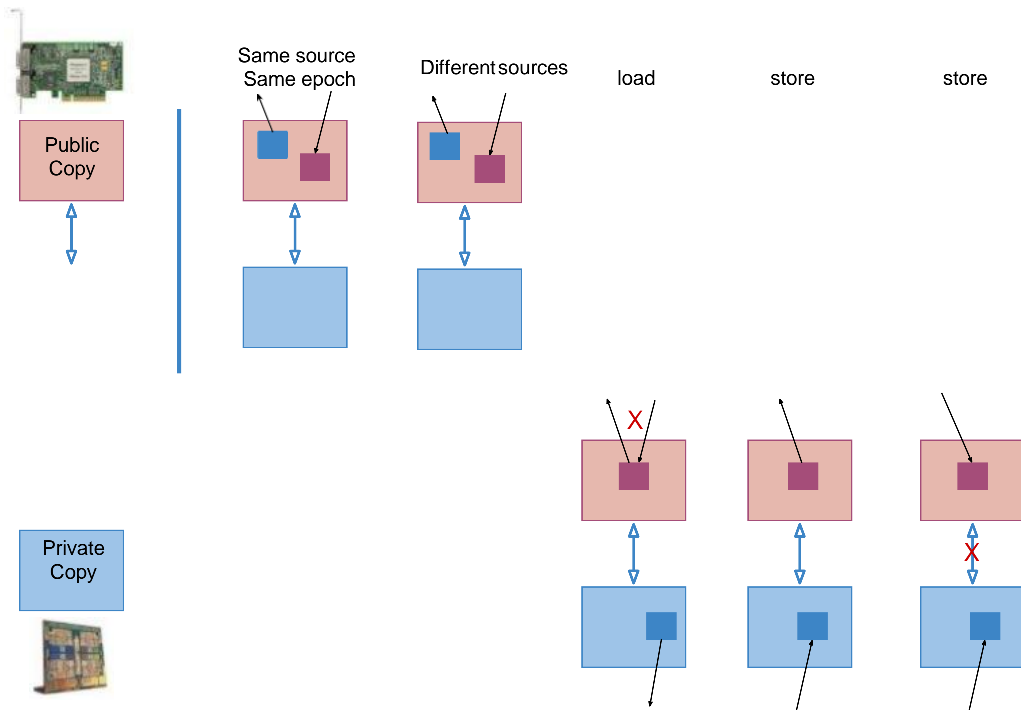


Figure 2.12 – MPI RMA memory model (separate windows).

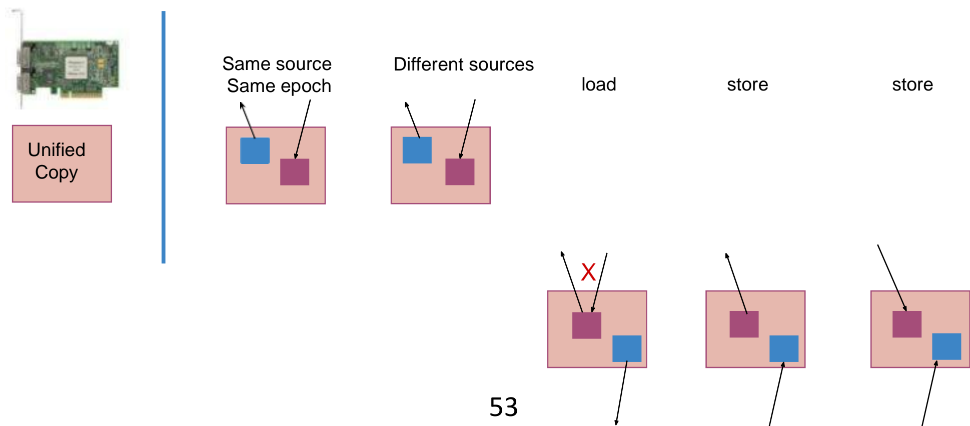


Figure 2.13 – MPI RMA memory model (unified windows)

one to the other. This model is close to today's existing RDMA networks [81] where such a transmission is always performed. It allows one to exploit the whole performance potential from architectures in which both the processor and network provide strong ordering guarantees. Note that programs that are correct in the separate model are always also correct in the unified model and programming in the separate model is more portable but may require additional synchronization calls.

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	NOVL	NOVL
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	NOVL	NOVL	NOVL	NOVL
Acc	NOVL	NOVL	NOVL	NOVL	OVL+NOVL

Table 2.1 – MPI-RMA operation compatibility table when two or more processes access a window at the same target concurrently in the unified memorymodel.
OVL: Overlapping operations permitted. NOVL:
Non-overlapping operations permitted. From [31].

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	X	X
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	X	NOVL	NOVL	NOVL
Acc	NOVL	X	NOVL	NOVL	OVL+NOVL

Table 2.2 – MPI-RMA operation compatibility table when two or more processes access a window at the same target concurrently in the separate memorymodel.
OVL: Overlapping operations permitted. NOVL:
Non-overlapping operations permitted. X: The
combination of operations is erroneous. From [31]

The compatibility of MPI-RMA and native load/store operations changes fromthe two memory models when two or more processes access a window at the same target concurrently. For the separate memory model the non-overlapping of store with put or accumulates leads to an error whereas in the separate model the non-overlapping between those operations is permitted. the two tables2.1,2.2show two different compatibility matrix of MPI-RMA and native load/store on both separate and unified memory model respectively. As in the unified memory model concurrent local load store and RMA

accesses are allowed and according to the standard they are not invalid (the outcome is not defined by MPI but defined by the hardware). Thus, our focus in this thesis will be on the unified memory model model.

2.4.9 MPI-RMA Operation Ordering

The MPI standard in its current form does not offer any ordering mechanisms for the classic put get operations (non atomic operations) or non-overlapping operations. MPI offers two solutions to order the MPI-RMA operations from the same origin to the same target: atomic operations and additional synchronizations by using flush, fence or lock that wait for the operation to complete on the target side.

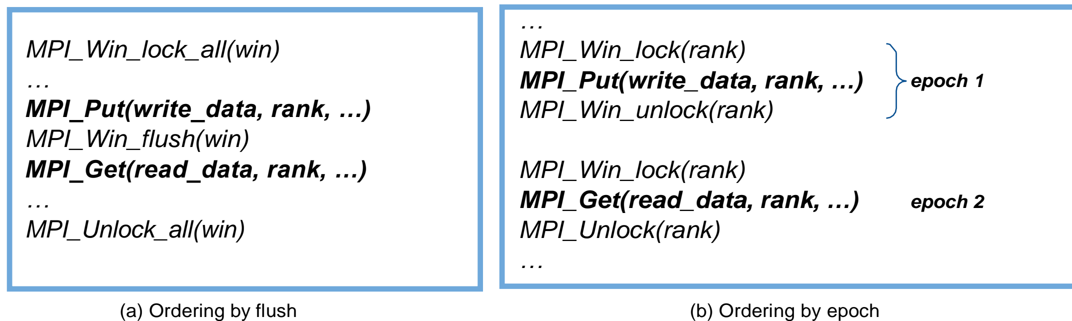
Atomic Operations from the origin to the same target memory location these oper- ations are ordered for the program execution by default. Atomic operations have limited applicability because of their complex use and expensive in term of implementation. An example is shown in figure2.14.

```
MPI_Win_lock_all(win)
…
MPI_Accumulate(write_data, rank, REPLACE)
MPI_Get_accumulate(read_data, rank, NO_OP)
MPI_Win_flush(rank)
…
MPI_Win_unlock_all(win)
```

Figure 2.14 – Ordered accumulate operations.

Flush Operations or Separate Epochs Flush operations become necessary in the passive target mode. A flush waits until all messages that were sent before the flush call are completed and their effects are visible to other processes. Flush synchronization can be called during any passive-target access epoch as shown in figure2.15(a) and ensures either local (flush-local, flush-local-all) or both local and remote (flush, flush- all) completion of all preceding MPI-RMA operations to the target.

This provides a way to synchronize without the overhead associated with re-obtaining a lock. A flush incurs an expensive network round trip and thus a significant overhead by forcing the application to wait for the completion of the operations even if the immediate completion is not required. It is highly recommended to minimize the use of a flush synchronizations to improve the performance.
Separate epochs as shown in figure2.15(b) can also be used to guarantee the ordering of operations. However it requires more synchronizations in the program.



(a) Ordering by flush (b) Ordering by epoch

Figure 2.15 – Ordering by synchronizations.

2.4.10 MPI-RMA Request Based operations

MPI-RMA since MPI-3 provides some variants of put, get, accumulate, and get_accumulate called "R" version (MPI_Rput, MPI_Rget, MPI_Raccumulate, and MPI_Rget_accumulate) in order to control the completion of specific messages and the associated local buffer resources.

These operations return an MPI_Request that can be used to determine a completion of a particular MPI-RMA operation. The request object can be similar to non-blocking point-to-point communication that wait or test the completion by using MPI_Test and MPI_Wait or their equivalent.

The same can be used in MPI-RMA request-based operations. It allows the user to dedicate a request handler with the MPI-RMA operations and test or wait for the local completion of these requests using the functions MPI_Wait and MPI_Test. Neverthe- less, completion refers only to local completion. For MPI_Rput and MPI_Raccumulate operations, local completion means that the local buffer is willing to be accessed. For MPI_Rget and MPI_Rget_accumulate operations, local completion means that the re- mote data has been landed in the local buffer. This operations are useful in cases where the application issues an enormous set of out-standing RMA operations and waits for the completion of them before it can start its computation. A common case would be for the application to issue data fetch opera- tions from a number of remote locations by using MPI_Rget and process them out of order as each one finishes (see listing 2.4). The Request-based operations present some drawbacks, these request-generating operations may be used only in passive target syn- chronization epochs (i.e., with lock/unlock and lock_all/unlock_all). The completion of a request returned by MPI_Rput, MPI_Raccumulate, and MPI_Rget_accumulate only signals local completion, thus the remote completion requires flush synchroniza- tion to signal the completion to the target. This flush may wait for the completion of unrelated operations, potentially from other processes. Additionally, users should be aware that the associated request manipulation can also cause additional overhead in

the MPI implementation because it uses a finer-grained of individual RMA operations. For those reasons the request-based operations fall out of the scope of this work we did not consider them.

```
1 int main ( int argc ,  char ** argv ) 2 {
3       /* MPI initialization and  window   creation   */
4
5       for ( i = 0 ; i < 1 0 0 ; i ++)
6             MPI_Rget( buf [ i ] , 1000 , MPI DOUBLE,. . . , &req [ i ] ) ;
7       while(1) {
8             MPI_Waitany ( 100 , req , &idx , MPI STATUS IGNORE);
9       process data (buf [ idx ]) ; 10  }
11
12      /* Window free and  MPI finalization */
13      return 0 ;14
}
```

Listing 2.4 – Pseudo Code Using Request-based Operations

2.5 MPI-RMA Proprieties and Programming Challenges

The main challenges of RMA programming revolve around the semantics of op- eration completion, operation ordering, and memory consistency. Most programming systems offer some kind of weak or relaxed consistency because sequential consistency is too expensive to implement. Thus, separating the communication (remote access) from synchronization leads to complications for remote memory access programming. Furthermore, the MPI interface divides synchronization into memory synchronization or memory consistency and process synchronization. On top of that, the asynchronous nature of MPI-RMA operations makes programming in MPI-RMA even more challeng- ing. As results we cite here three main programming priorities that are mandatory to achieve efficient overlap of communication with computation.

The first property relates to completion. To allow overlap of communications with com- putations, one-sided communications are asynchronous by nature, i.e. there is no need of progress for these communications to complete. This also means that, when initiated, the programmer has no way to know when the communication has been completed, be- fore the end of the current epoch. Figure2.16shows an example of this propriety. The Put from process *0* can complete at any time during the MPI epoch between the two Fence calls. This means that MPI_Put call can either return *1* or *0* depending on when the operation completes. A synchronization call like an end of epoch as Fence or an ex- plicit synchronization call such as MPI_Win_flush is needed to ensure the completion of MPI-RMA communications.

The second property revolves around the ordering of communications. The ordering of MPI-RMA operations during an epoch is not defined. Communications can happen in any order and it is often fixed by over-synchronizing the program, causing a severe

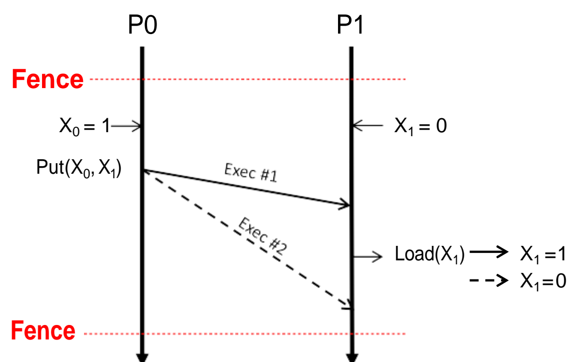


Figure 2.16 – An example illustrating an unknown completion of MPI-RMA operations inside an epoch.

performance loss due to lack of overlapping possibilities. As shown in Figure2.17de- pending on the execution order of the two MPI_Put operations, the result of the read on the value they write on can be either *2* or *1* depending on the execution order of the operations.

Finally, the third property is about atomicity. Except for Accumulate routines ensuring

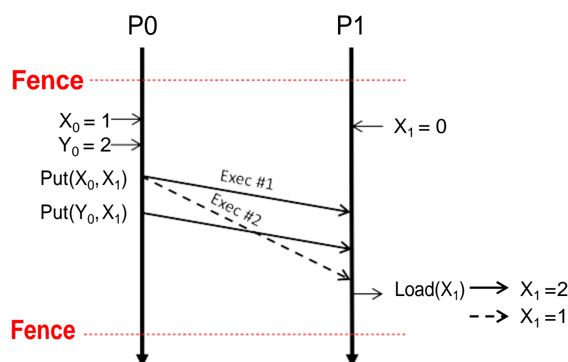


Figure 2.17 – An example illustrating the lack of ordering when using MPI-RMA oper- ations.

the atomicity of accesses, regular MPI-RMA one-sided communications (e.g. MPI_Put, MPI_Get) are not atomic. This means that concurrent accesses to the same memory location result in an undefined behavior. An example of the type of issues that can happen is shown on Figure2.18. If a buffer composed of several elements of a specific MPI datatype here, two integers are written concurrently through MPI_Put to the same destination, the result can be either one of them (depending on the ordering), or a com- bination of the two where the unit of atomicity is the MPI dataype, here an integer.

The MPI standard does not categorize this case as an error, but specifies that this is implementation dependent. In this work, we report such scenarios as errors.

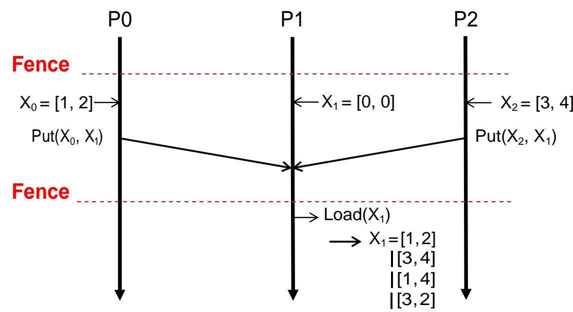


Figure 2.18 – An example illustrating the non atomicity when using MPI-RMA opera- tions.

While these three properties completion, ordering and atomicity are the root of the performance of MPI-RMA, the associated non-determinism makes it difficult to implement such programs without errors.
Our goal is to tackle these issues and help the programmer find such errors in MPI-RMA programs.

2.6 Conclusion

In this chapter, we first introduced PGAS and its characteristics, we then presented MPI synchronous and asynchronous two-sided point-to-point communication and MPI collectives before talking about MPI one-sided point-to-point communication and its semantics. We showed that the main difference between the two MPI point-to-point patterns (one-sided and two-sided) is that the RMA interface separates communication and synchronization and offers different collective and non-collective synchronization modes. Besides, it allows the programmer to choose between implicit notification in active target mode and explicit notification in passive target mode in addition to several ways of creating windows. This large variety of options allows users to create complex programs but, current MPI-RMA implementations come with several limitations and challenges with respect to operation ordering, operation completion andatomicity.
These challenges make reasoning about MPI-RMA more complex and lead to memory consistency errors. On top of that, These limitations generate non-deterministic behav- iors and thus result on undefined execution states.

Our goal is to address these limitations by providing a solution to tackle memory consistency issues. In the next chapter, we will introduce memory consistency and show some motivating examples of memory consistency errors. We will then, exhibit our main contributions for the purpose of detecting data race errors in MPI-RMA programs. First, by dynamically analyzing the program, we detect data race errors at runtime. Second, we detect local concurrency errors at compile time by using a static analysis based on pointer aliasing.

CHAPTER 3

DYNAMIC DATA RACE ERRORS DETECTION IN MPI-RMA PROGRAMS

The work that will be shown in this chapter has been published at EuroMPI in 2021 [1]. The content is reproduced in this chapter.

3.1 Introduction

Several MPI programs are written in Fortran/C/C++ and run on clusters with thou- sands to hundreds of thousands of cores. These programs can have not only the common C/C++/Fortran bugs such as memory leaks or buffer overflow, but also bugs specific to MPI such as deadlocks, illegal buffer reuse or hazard accesses to a memory region. Ear- lier, we presented MPI-RMA programming challenges involving data race errors, which is troublesome for developers to debug because those bugs appear somewhere in the pro- gram. There are many tools for MPI errors detection that are unreliable for such bugs because they do not provide any coverage guarantee over the space ofnondeterminism or data race errors in MPI programs.

The MPI standard offers a rich set of features such as non-blocking primitives and nondeterministic constructs that help developers write better high performance applica- tions. These features, however, complicate the task of large-scale debugging, especially over the space of nondeterminism, which requires causality tracking. It also creates debugging challenges that are even more difficult than the scenarios presented in the previous section. Traditional causality tracking algorithms, such as Lamport clocks and vector clocks, are usually not sufficient to handle such complex semantics especially when it comes to data race errors detection. In addition to MPI-related bugs, data race errors in distributed memory programming are recognized as challenging bugs. The difficulty of these bugs lies in the lack of ensuring the same order of processes interaction and the absence of completion awareness.

In this thesis, we investigate the insufficiency of the studies that have been done for this kind of errors such as trace-based approaches and Lamport happens-before order. We propose a new approach to detect data race errors in MPI-RMA programs. To detect this kind of errors in MPI-RMA programs for a given set of data, a dynamic verification is more suitable. By using a dynamic verification we can detect the very first data race error that occurs in the program and thus, stop the program immediately as other data races can be a side-effect of the first one.

While there are a lot of existing dynamic verification tools for other types of parallel software, to the best of our knowledge there is no similar tool that support MPI-RMA programs. We provide a main analysis that can realize and maintain a full coverage of the data race errors detection. It relies on analysing the program at execution time for purposes of collecting all memory accesses including native load and store ones. This approach gives a very good error coverage but comes with overhead costs. A static anal- ysis of the program could improve the overhead cost by better filtering and collecting load and store accesses.

While in dynamic verification, searching data race errors in a program count on a given set of data, it is still restricted by the information obtained during runtime. Static analysis might automate this process and can warn the user of data race errors before performing the dynamic analysis.

Both protocols are implemented in our tool called the RMA-Analyzer. The RMA- Analyzer implements the data race errors algorithm and provide a complete data race errors analysis to detect all possible errors in MPI-RMA programs.

3.2 Memory Consistency in MPI-RMA

A memory consistency is the term used to determine if the memory accesses state is well defined. A memory consistency state is generally used to verify if concurrent accesses or non-deterministic behaviors occur during a given program. If a memory vi- olation is found it can cause memory consistency errors.

To explain how a memory consistency error can occur in MPI-RMA programs. We introduce some memory consistency properties respectively called "Happens-before" and "consistency order" based on some formulas used by a paper written by the MPI-RMA working group [49].

3.2.1 Happens-before Rules

Based on Lamport clocks [80] and vector clocks [93] the relation $\xrightarrow{hb}$ [60] of a set of events of a system depends on the following conditions:

—if *a* and *b* are events in the same process, and *a* comes before *b*, then $a \xrightarrow{hb} b$.

—if *a* is the event that sends a message by one process and *b* is the event that receives the same message by another process then $a \xrightarrow{hb} b$.

—if $a \xrightarrow{hb} b$ and $b \xrightarrow{hb} c$ then $a \xrightarrow{hb} c$.

Furthermore the $\xrightarrow{hb}$ order can be necessary to know when the execution effects of an operation are completed. Retrieve this information for MPI events can be a difficult task because an MPI event can be in different states from the invocation of the MPI call by a given process to the end of the event locally. To order the MPI events correctly the completion state does not suffice thus a correct synchronization requires a definition of the $\xrightarrow{hb}$ relation between conflicting accesses. For example let's consider two operations *Op1* at time *t1* and *Op2* at time *t2* accessing the same memory location *M* with an intended ordering of $t1 < t2$. The effect of *Op1* on *M* (if any) must be visible for *t2* and no effect of *Op2* on *M* shall be visible before. Concerning the conflicting accesses read and write operations that access the same memory location so-called data hazards impose a strict ordering among operations: write-after-write, read-after-write, and write- after-read. The ordering restrictions of such operations are imposed by the application, e.g., by the sequential order [40] of operations in the application's source code. Compilers may typically reorder operations as long as these ordering constraints are not violated, which otherwise would lead to erroneous results.

3.2.2 Consistency Order Rules

The Consistency order [49] is the partial order of actions that happen in memory. If $a \xrightarrow{co} b$ that means that the memory effects of action *a* are visible before those of *b*. This order is mandatory for some synchronization actions like the flush ones to order the memory accesses without any synchronization between processes.

The consistency order does not introduce any happens before relations, if $a \xrightarrow{co} b$ then the effects are guaranteed to be visible only if $a \xrightarrow{hb} b$. Otherwise, *b* could happen in real time before *a* and thus *a*'s affects are not visible to *b* even if $a \xrightarrow{co} b$ and holds for a particular execution. It is guaranteed however, that $a \xrightarrow{co} b$ implies that operations that happen later than *b* will eventually observe the effects of *a*. This guarantee is needed for polling and does not require $\xrightarrow{hb}$ ordering. On the other hand, an operation that synchronizes processes and is thus, part of the $\xrightarrow{hb}$ may not synchronize memory accesses, that is if $a \xrightarrow{hb} b$ it does not imply the consistency order between the two operations. Thus, a $\xrightarrow{hb}$ does not imply any $\xrightarrow{co}$.

3.2.3 Memory Consistency Errors in MPI-RMA Programs (Data Race

Memory consistency errors also called data race errors in MPI-RMA programs were clearly specified in the work of [23].

If there are two concurrent events accessing the same memory area and there is at least one of them that is an update (write) operation (local or remote), there exists a memory

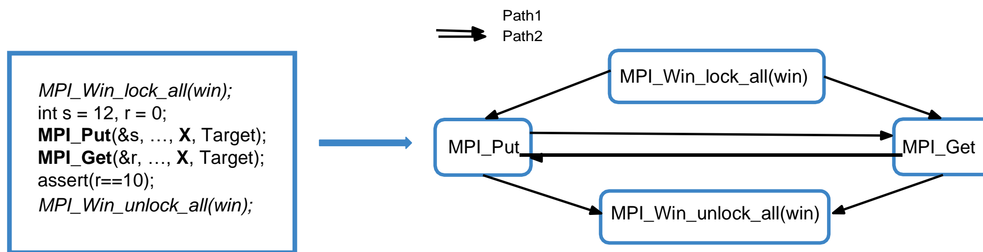


Figure 3.1 – Conflicting two operations in the same program.

consistency error in an MPI-RMA program execution. An exception to this definition is made for accumulate operations that use the same operation and basic datatype.

Two events *a* and *b* are concurrent ($/\!/_{cohb}$) when they are not ordered by consistency

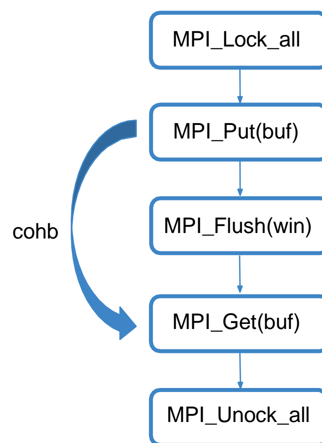


Figure 3.2 – Synchronized operations in an MPI-RMA program.

happens-before order $\xrightarrow{coh\,b}$. In other words, a data race error may occur in the program if $a \xrightarrow{cohb} b$, and if they are directed towards overlapping memory locations at the same process and either:

—one of the two operations is a put rp (remote put).

—if one of the operations is an accumulate RA (remote accumulate).

—if the first operation is a get rg (remote get) and the second one a local write (w).

$$a \parallel_{cohb} b \Leftrightarrow a \overset{cohb}{\nrightarrow} b \wedge b \overset{cohb}{\nrightarrow} a$$

Figure3.1shows an example of two conflicting operations (eg; put and get) in the same program.

By contrast a program is called data-races free if all the conflicting accesses are ordered by the $\xrightarrow{coh\,b}$ as depicted infigure3.2.

$$a \xrightarrow{coh\,b} b \Leftrightarrow a \xrightarrow{h\,b} b \wedge a \xrightarrow{co} b$$

Only programs where all executions are data-races free have well-defined memory semantics. If a program has well-defined semantics, then a read action r will always return the last written value (last as defined by the consistent happens-before order).

3.2.3.1 Non-determinism in MPI-RMA

The MPI standard allows MPI calls to have nondeterministic behaviour to give more flexibility to the programmers and reduce code complexity. In MPI-RMA programs we can face some nondeterministic behaviours due to the lack of memory consistency. To better understand the non-determinism issue in MPI-RMA programs, we provide an intuitive example to explain the MPI-RMA semantics, we illustrate possible MPI-RMA programs behavior using example shown in figure3.3. The figure considers the MPI-

X = 0	**Y** = 1
P1 :	P2 :
X = get(Y, P2)	**Y** = get(X, P1)
a = **X**	b = **Y**

Figure 3.3 – Example of a non-deterministic behavior in RMA programs.

RMA program with two processes *P1* and *P2*. The process *P1* has shared variable *X* with initial value *0* and local register **a**. Process *P2* has shared variable *Y* with initial value *1* and local register **b**. We assume here that the programs synchronize after setting initial values for their shared variables. In this program we see RMA's remote reads and writes. The first process reads remotely the value of *Y* and stores it in *X*. The process *P2* reads remotely the value of *X* and stores it it in *Y*. The CPU handles the remote accesses which are enqueued onto its network interface card (NIC), the NIC then executes required remote communication and memory accesses without involving the CPU. After initiating the remote accesses, each process reads locally the variables *X* and *Y*, respectively, and stores the results in registers **a** and **b**. To understand this program under sequential consistency, one needs to consider the inter-leavings of the actions making up the get statements in each process. Possible execution results include **a** = *0*, **b** = *0* and **a** = *1*, **b** = *1*, and, with non-atomic get statements, **a** = *1*, **b** = *0*. Nevertheless, RMA has additional behaviors, because the local reads are not guaranteed

to run after the get statements. We have a possible execution result under a non-sequentially-consistent memory model which is **a** = *0*, **b** = *1*.

Discussion

We have provided the notion of happens-before consistency order, which allows us to study the causality between events in an MPI execution. We have also defined the con- cept of concurrent events in a program, which is the effect of not verifying the happens- before consistency order between MPI-RMA events. This is in contrast with most proto- cols based on the traditional Lamport clocks and vector clocks, which consider an *a* event to happen-before the *b* event as sufficient to determine overlapping accesses. We will introduce some motivating examples and show some data race errors that can happen in MPI-RMA programs when the MPI-RMA events are not ordered by happens-before consistency order. Since MPI-RMA communication operations are asynchronous the happens-before relation between operation is by default not verified. We will also show two different kinds of data race errors and later in the chapter introduce the MPI-RMA operation compatibility table which is a matrix that presents all data race errors that can be caused between local and remote accesses.

3.2.4 Motivating Examples of MPI-RMA Programs

The main characteristic of MPI-RMA programming is the ability to decouple data movement from synchronization, it is the strength of one-sided communication, but, it is also hard to use and understand. Furthermore, programmers must face complex memory model and insert the synchronization operations needed to maintain data consistency in the presence of asynchronous data accesses from multiple processes. It thus, presents a complex synchronization model and it is difficult to maintain memory consistency between possibly conflicting asynchronous data accesses. Bugs related to memory con- sistency may lead to an erroneous state manifested during one execution that may not be triggered during another execution due to the underlying MPI library and network interconnect facilities. Often, errors remain unnoticed for a long period of time and only occur in large-scale scenarios or after porting the application to a different HPC platform. This complexity can expose applications to synchronization defects. We will introduce some memory consistency related bugs that are divided in two types of errors: at a single process in the same epoch and between several processes in different epochs. We will also show some motivating examples and highlight the most common errorsin a one-sided communication program.
In MPI-RMA programs we can have two kinds of memory consistency errors. One oc- curs within an epoch involving one process, the other happens across several epochs and between two ore more than two processes.

Consistency errors within the same epoch in MPI-RMA programs

A data race can happen at the same process within an epoch. Several data race errors can occur between the MPI-RMA operations and the native load store operations. In a single program, conflict accesses to the same memory region are allowed. Figure3.4 shows a typical example of a data race error within an epoch at a single process in a given MPI-RMA program. MPI_Put sends a data in **buf** from process *P0* to *P1*. In this Example the MPI_Put has the program order with the store operation on **buf** occurring right after. However, the MPI_Put does not have the consistency order with the store operation and as the MPI_Put is an asynchronous operation the data may or may not be sent. The MPI_Put and the store operation do not have the "consistency order happens- before" relation since the two operations access the variable **buf** and happen on the same process and the data may be corrupted. Such errors are very common in MPI-RMA programs and even in applications that use one-sided communication. The same problem was found in the Asynchronous Dynamic Load Balancing (ADLB) [65] library, which is used in the Green's function Monte Carlo (GFMC) [79]. Since this application transfers data by using MPI_Put and it does not wait for the completion. This application worked correctly on multiple machines. But, when the code was transferred to the IBM Blue Gene/Q, the MPI implementation has shown a buggy state. Because the function was out of local buffers and had to transfer the data when the buffer were freed. In such cases, the function stack was overwritten by other functions, resulting in concurrent accesses.
Several scenarios can be the origin of this kind of error for example between an MPI_Put and a store or MPI_Get with a load or store happening right after or even between only MPI-RMA operations from the same origin towards the same window memory location.

P0 (Origin)	P1 (Target) ***Window location Y***
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
MPI_Put(buf, 1, Y …)	
buf = …	
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 3.4 – Memory consistency error within an epoch at a single process.

Consistency Errors among several epoch in MPI-RMA programs

A data race also occurs across several processes in different epochs (eg, between two synchronization modes) as shown in figure3.5. The MPI_Put from process *P2* that is issued inside a lock/unlock synchronization epoch in group *B* towards process *P1* of group *A*, and the MPI_Get that is issued in a Post-Wait epoch in group *A* are con- flicting accesses. As a Get is a local write in process *P1* and the Put is a remote write in process *P1* from process *P2* even by using different synchronization modes, those communication operations are considered as concurrent accesses because they target the same memory location independently of epochs in which the communication operations are issued.

Figures3.6,3.7and3.8illustrate errors occurring among multiple processes and be-

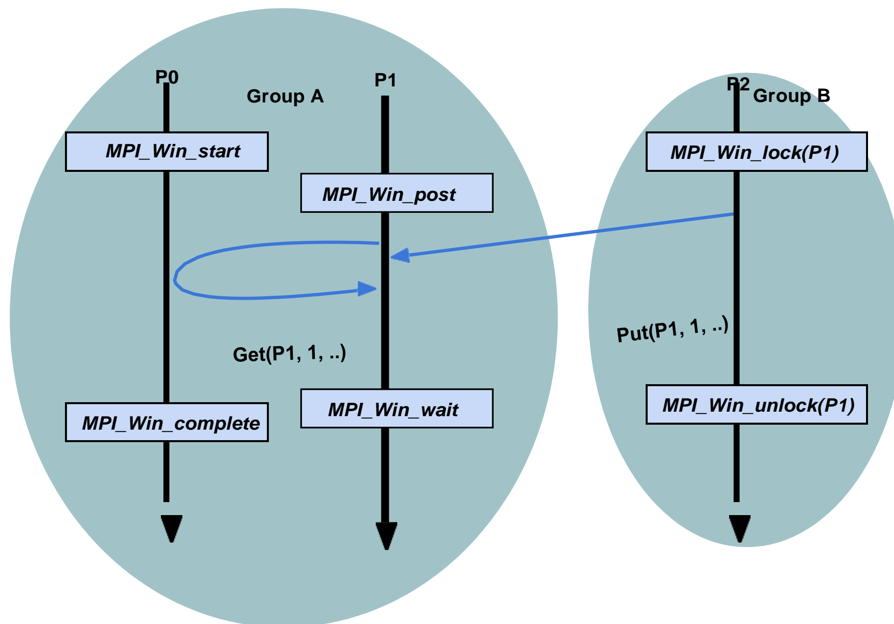


Figure 3.5 – Memory consistency error occurring in several synchronization epochs.

tween different epochs. In figure3.6, an MPI_Put on *P0* and an MPI_Get on *P1* both access the same window buffer *Y* located on *P1*. The two operations do not have happens-before consistency order relation between them, since they could happen in any order. This means they access the same memory location concurrently and thus lead to a data corruption or undefined results during the execution time of the program. Here we can see that memory consistency errors also exist when interleaving MPI-RMA operations between processes.

Secondly, in figure3.7, processes *P0* and *P2* both perform an MPI_Put at the same

P0 (Origin)	P1 (Target) *Window location Y*
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
MPI_Put(_, 1, Y, …)	**MPI_Get(Y, 0, _, ...)**
…	**…**
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 3.6 – Consistency error between two processes.

window location *Y* on process *P1*. For the same reason as before, a consistency issue also exists in this case, where different processes access a same memory location concurrently on the same peer.

Finally, figure3.8depicts an example of a memory consistency across processes where

P0 (Origin)	P1 (Target) *Window location Y*	P3 (Origin)
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
MPI_Put(_, 1, Y, …)		**MPI_Put(_, 1, Y, …)**
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 3.7 – Consistency error between three processes.

both MPI_Put on process *P0* and the store on process *P1* access the same window buffer *Y* belonging to process *P1*. Furthermore, the two operations are concurrent and happen between two different processes. Thus, there exists a data race error between MPI-RMA operations and native load store operations among several processes.

These errors that are demonstrated here are common and happen frequently in an MPI-RMA program because of the possibility of having several implementation choices. One can freely use communication operations with less synchronization calls. The lack of synchronization calls in the program can be a justified choice to deal with performance but it can also be the reason of the presence of data race errors in the program.

```
P0 (Origin)                          P1 (Target)
                                      Window location Y
MPI_Win_lock_all(win)                MPI_Win_lock_all(win)
MPI_Put(_, 1, Y, ...)                Store Y
...                                  ...
MPI_Win_unlock_all(win)              MPI_Win_unlock_all(win)
```

Figure 3.8 – Consistency error between native Store and MPI_Put across two processes

.

3.2.5 Extending MPI-RMA Operations Compatibility

Since MPI-3, all MPI-RMA operations are allowed inside an epoch but some situ- ations can lead to an undefined behavior. MPI-RMA operations compatibility table is presented in table3.1. In this table, all operations are supposed to read or write on the same memory location. The cross means that consistency order between the two operations is not guaranteed. This table is different from the one in [23] as we separate operations from/to the origin and target processes. Indeed, MPI-RMA operations can be considered as READ or WRITE operations depending on the process thatperforms them (Origin or Target). As an example, a Put operation is a READ for the origin pro- cess and a WRITE for the target process. We differentiate local and remote statements as follows:

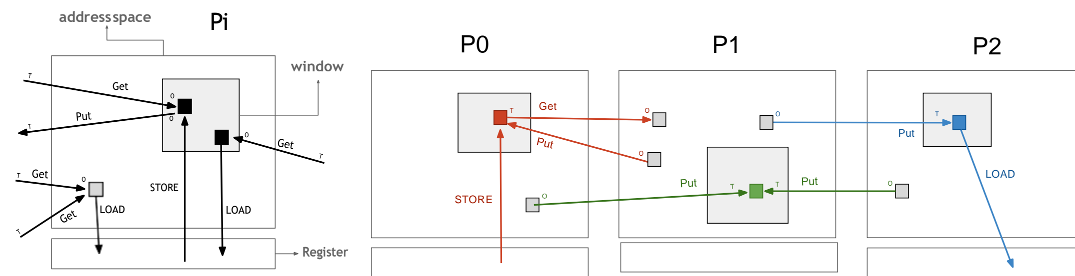
		ORIGIN		TARGET		LOAD	STORE
		GET	**PUT**	**GET**	**PUT**		
O	**GET**	x	x	x	x	x	x
	PUT	x	√	√	x	√	x
T	**GET**	x	√	√	x	√	x
	PUT	x	x	x	x	x	x
LOAD		x	√	√	x	-	-
STORE		x	x	x	x	-	-

Table 3.1 – Compatibility of RMA operations and local load/store accesses on the same address space. O=ORIGIN, T=TARGET, √= overlapping is permitted,x=undefined behavior, overlapping is not permitted.

Local Statements (Local Accesses) A local statement also called local access can only read or write variables that are at the origin process, the origin process executes

the statement. By using a local statement, a given process can access variables in its memory. The term local is used because of the process does not access the memory of other processes. Accessing a variable can be either atomic or non-atomic.
Local put Process P reads its local memory location and writes it to a remote memory location.
Local get Process P reads from a remote memory location and writes it to its local memory location.

Remote Statements (Remote Accesses) A remote statement also called a remote access can read or write to and from any memory location. By using a remote access, a given process can access any variable in memory. Remote accesses are performed asyn- chronously. When a process performs a remote access, it requests the network interface card to perform the needed read or write operations.
Remote put A remote process RP reads from a remote memory location of P and writes it to its local memory location.
Remote get A remote process RP reads its local memory location and writes it to a remote memory location of P.
We show an example of this local and remote accesses in figure3.9aand3.9b. As the completion of RMAs can occur any time during an epoch, and it is not dependent on the time when the RMA call is issued, so if two accesses are performed to the same address during the same epoch, and one of them is a write, then there is a memory consistency issue.
Several scenarios involving different processes can account for this situation: figure3.9



(a) Concurrency errors inside a process.

(b) Concurrency errors across processes.

Figure 3.9 – Example of memory consistency errors. Dashed edges represent WRITE operations while plain edges represent READ operations on the colored boxes. *O* and *T* respectively indicate the origin and target processes.

presents conflicting accesses at (a) origin and (b) target of RMA operations. Conflicts can also arise if one memory location is used as origin (i.e., as local buffer in Get/Put) and as target window. First, in figure3.9a, one process can issue a remote Get to write a local variable in its private address space and use this variable with a load. As both accesses are unordered, this is memory consistency issue. The same situation arises when

the access address is in a window: a local store for instance and a Put to a distant vari- able, reading this same variable is also a memory consistency issue. Second, figure3.9b illustrates different cases involving three processes. For instance, a remote Put by one process and a remote Get by another process, to the same address in the same window leads to non coherent result (red edges in the figure).

All this memory consistency errors should be detected and tracked during a program compile time and execution time in order to ensure a correct program and a correct execution state. By defining all possible errors and separating them into two groups: local errors and remote errors depending on the process causing them. On this basis, we can verify MPI-RMA programs and tackle coherency issues by providing a complete program analysis.

Discussion

As we introduced some examples that show how a data race error can occur inside a single epoch with one process and among multiple processes in several epochs.
We also extended the compatibility table that summarizes all the possible scenarios of concurrent accesses. In contrast to the state of the art compatibility matrix we distin- guish between local and remote accesses depending on the origin process that initiates the communication operation in order to cover all the possible data race errors. This table allows us to clearly identify the type of memory access that can lead to a data race error. In reference to this table we can easily pinpoint data race errors in a given program and tackle them at compile time and execution time.

3.3 Dynamic Data Race Detection Algorithm

To detect errors presented in table3.1, the runtime has to maintain a precise state of the distributed memory by keeping track, for each process, of all the accesses performed to memory addresses it owns and shares with other processes through RMA. Accesses can be either local loads and stores, or MPI-RMA operations, from the process itself or from remote processes. In order to reduce the cost in term of time and space of such bookkeeping, memory regions modified are stored as union of disjoint intervals in a binary search tree (BST). Each node of the BST contains a memory address interval (itv), the access type (access) and possibly empty left (Left) and right (Right) sub-tree containing intervals lower and higher resp. to the parent interval. All BST modifications are protected by a lock.
For each window created, each process creates a BST associated to the addresses it owns in this window. In order to be notified by distant processes of accesses to the addresses it owns in a window, a new thread is created and keeps calling MPI_Recv from any source to receive access notifications from other processes. A BST is also created for variables that are not in any window, at MPI_Init. For each beginning of epoch, associated to the

window is emptied. When a GET is executed, the BST of the origin process is updated with the local variable written, either the BST of a window or the BST of local variables. The distant variable has to be added to the BST of the target process as a read. The GET function is instrumented to send a notification (with an MPI_Send) of the address read to the target process, and the process increases the number of notification sent to the target. Likewise, a PUT sends a notification for a write to the target process and adds a read to the local BST. In both cases the target process receives the notifications and all updates to the BST are performed according to Algorithm1. It checks if the new memory access will lead to a concurrency error or not. If the memory access is safe, the memory interval is inserted in the BST (line 5 in the algorithm). If the memory access overlaps a memory interval stored in the BST and one of them is a write, the program is stopped and an error message is returned to the developer (line 3 in the algorithm). The same applies for PUT accesses.

When an epoch terminates, each process counts the number of notification the other

Algorithm 1 Data Race ErrorsDetection

Require: Binary search tree T, Memory interval I, Accesstype A ▷ A isREAD, WRITE, local READ or local WRITE

Ensure: Updated T. An error message is issued in case of a data race

1: **procedure** BSTUPDATE(T, I, A)
2: **if** $I \cap T.itv \neq \emptyset$ **then**
3: **if** ACCESS(A, $T.access$) == $ERROR$ **then**
4: Raise an error and stop the program
5: **else**
6: $T \leftarrow$ splitInterval($T, I \cap T.itv, A$)
7: $I \leftarrow I - T.itv$
8: **if** isLeaf(T) **then**
9: $T \leftarrow$ newNode(I,A)
10: **if** $I \cup T.itv$ is an interval **and** A == T.access **then**
11: $T.itv \leftarrow I \cup T.itv$
12: mergeNeighboringIntervals(T)
13: **else**
14: **if** $I < T.itv$ **then**
15: BSTupdate(T.Left, I, A)
16: **else**
17: BSTupdate(T.Right, I, A)

process have sent to it (with an MPI_Reduce) and terminates the receiver thread when all messages have been received. This behavior is made possible by the fact that, we focus on MPI-RMA programs that use the epoch synchronization calls on all the processes of the window, i.e. as a collective call. Generalizing this approach by exchanging this number of notifications through Send/Recv calls instead would be feasible, but is out of the scope of this thesis.

Finally, all load and store accesses are also instrumented during an epoch and theses accesses are registered in the BST for local variables or the BST associated to a window, depending on the range of addresses.
The intervals of the BST do not approximate regions: all addresses in the interval have been accessed with the access type registered. There is no over-approximation of the accessed regions and therefore no false positive.
table3.2defines the types of access to the memory (ACCESS function on line 2 in the

ACCESS		**local read**	**local write**	**read**	**write**
O	**GET**	write	write	ERROR	ERROR
	PUT	read	read	read	ERROR
T	**GET**	read	ERROR	read	ERROR
	PUT	ERROR	ERROR	ERROR	ERROR
LOAD		local read	local write	read	ERROR
STORE		local write	local write	ERROR	ERROR

Table 3.2 – Transition table for access types. Given an address with an access type (first row) and a new operation to this address (first column), the table defines the new access type after the operation. We assume there is no data race within a (multithreaded) process.

algorithm). Local access types correspond to coherent accesses within the process. No coherency issues can arise with only local accesses. A local store access to an address previously locally read modifies its access type without error. To compare two statements for memory accesses, the first access is to be read by column, and the second one by row. For example, in the figure3.4, the first access at origin side is a PUT, thus a read in column. The second access is a STORE in row. This combinaison leads to an error. If the two statements were inverted, a STORE would have been associated to a local write in column and the PUT at origin in row, which would have resulted in a legal read access. Line 6 splits the interval if the access A is different from the interval access. If accesses are the same, the splitInterval does nothing. Lines 11-12 correspond to the case where the new interval is disjoint and next to the interval T. Both intervals are then fused, and possibly with the rightmost interval of the left sub-tree or the leftmost interval of the right sub-tree of the modified node. This is what the function mergeNeighboringIntervals does.

Algorithm Proof we now prove that Algorithm1detects all memory consistency er- rors, and only memory consistency errors when there is no aliasing between co-existing windows.

All memory consistency errors are detected by the RMA-Analyzer : An error happens when two accesses are performed to the same address of the same process, one

of them is a write and they are unordered. Let us assume first that this address is inside a window, and only belongs to one window (aliasing between windows is not considered). It is owned by one process and this process has created a BST at window creation. One of the accesses is either a PUT or GET distant or local, and the other access is either a PUT, GET, load or store. Load and store accesses are assumed to be coherent ac- cesses, even if multi-threaded accesses are performed. The RMA calls a MPI_Send to the process owning the address and since all notifications are counted before closing an epoch, this notification reaches the target process and updates the BST of this window (unique). A load or store access updates the same BST of the window accordingly and since the BST is protected with a lock, modifications are serialized and the Algorithm 1 will raise an error at the second modification.
Since the BST is emptied only at the beginning of an epoch (epoch creation is a collec- tive) the BST will find that two accesses are performed to the same address with one of them is a write. If the address is not in a window, the BST in charge of local addresses is used. The local address of the GET or PUT is accessed and as these functions insert their accesses to the local BST, the coherency issue is detected.

The only errors detected by the RMA-Analyzer are memory consistency er- rors : In order to detect the error, the two accesses have to appear in the same epoch (BST are reset between epochs), and in the same window (there is only one BST per window) or in the same private memory (local variables of a process). According to table 3.2, the error can only be raised when two accesses are performed to the same address with one of them distant and one of them a write.

3.4 Design and Implementations of the RMA-Analyzer

Our data race errors detection algorithm passes by two main steps. As first step, we use the MPI profiling interface (PMPI) in order to intercept all RMA calls and retrieve all the needed information to our analysis. As second step, for the local load store accesses we use an llvm pass to collect them from the first MPI-RMA window opening in the program and check for data race errors (if any).
We first, present PMPI and the LLVM toolchain. Second, we introduce the PARCOACH framework overview prior to discussing the RMA-Analyzer framework overview.

3.4.1 The MPI profiling Interface

MPI defines a profiling interface called PMPI to help the user to perform various analyses. PMPI thus, can be used for the performance measurement and data tracking. The profiling interface provides wrappers for all MPI calls in order to help the user to take advantage of the profiling either individual MPI calls (e.g., PMPI_Put) or the whole library. This functionality is provided by providing two APIs for each MPI routine MPI_ and PMPI_. The wrapper invokes the MPI calls from the runtime by issuing the

corresponding PMPI calls.
Figure3.10shows a simple example of the use of PMPI wrapper that counts the number of MPI_Put that are found in the program. Furthermore, this interface may be used to change the behaviour of the MPI routines without code source modification. The flip side of the profiling interface is that there can be only one active wrapper linked with the program.

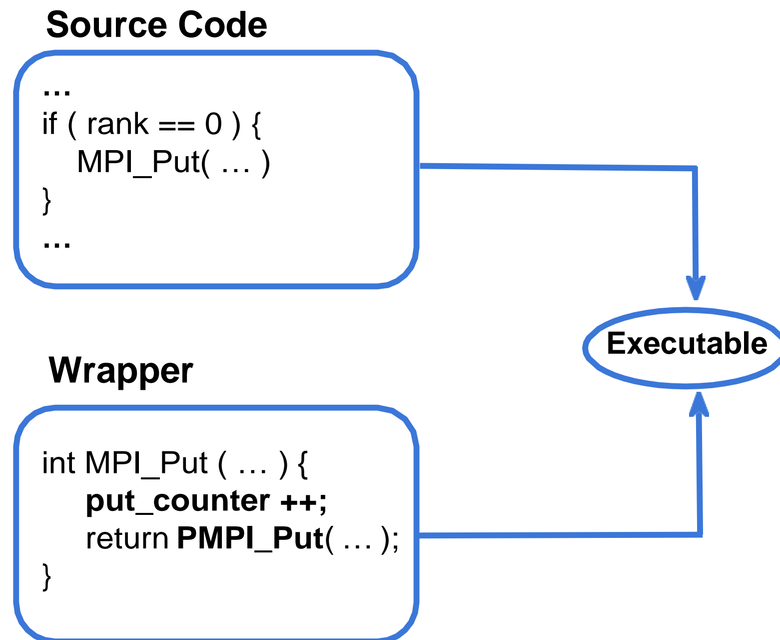


Figure 3.10 – A simple example of PMPI wrapper counting the number of MPI_Put.

3.4.2 LLVM Pass

"LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy." -LLVM Language Reference Manual [62].
The Low Level Virtual Machine (LLVM) is a compiler framework which is built around the LLVM Intermediate Representation (LLVM-IR) and comes with a large variety of analysis and transformation called LLVM passes. The LLVM framework was especially designed to optimize a program throughout its lifetime, which means at compile time, link time and even run time. Furthermore, LLVM includes back ends for static and

just-in-time code generation for architectures like x86-32, x86-64, ARM or PowerPC. LLVM has been developed as a set of libraries which implement various parts of a compiler. They can either be embedded into existing compilers to incorporate LLVM functionality or a set of LLVM command line tools can be used to access library features directly. All libraries provided by LLVM work on a common intermediate representation called LLVM-IR. It is possible to export the LLVM-IR at different stages of the compi- lation, to move it between tools or to modify it manually.
To get an idea of how LLVM works, figure3.11shows a classical static compilation process that uses LLVM tools. The example shows a small program consisting of three source files that is compiled into a single executable. The source files contain code writ- ten in c, c++, and Fortran. There exists a language specific frontend to LLVM-IR for each source code. We cite here the frontends clang or llvm-gcc for C/C++ code as well as flang or llvm-gfortran for FORTRAN code. According to [36] flang will be replaced by Fortran18 (F18). F18 is the he New Fortran Front-end and is becoming the standard Fortran compiler for LLVM. To optimize the translation unit of each source code the tool chain uses llvm-opt tool that is run on each LLVM-IR file. The optimizers can be run as set of standard optimizations passes or as individually selectedpasses.

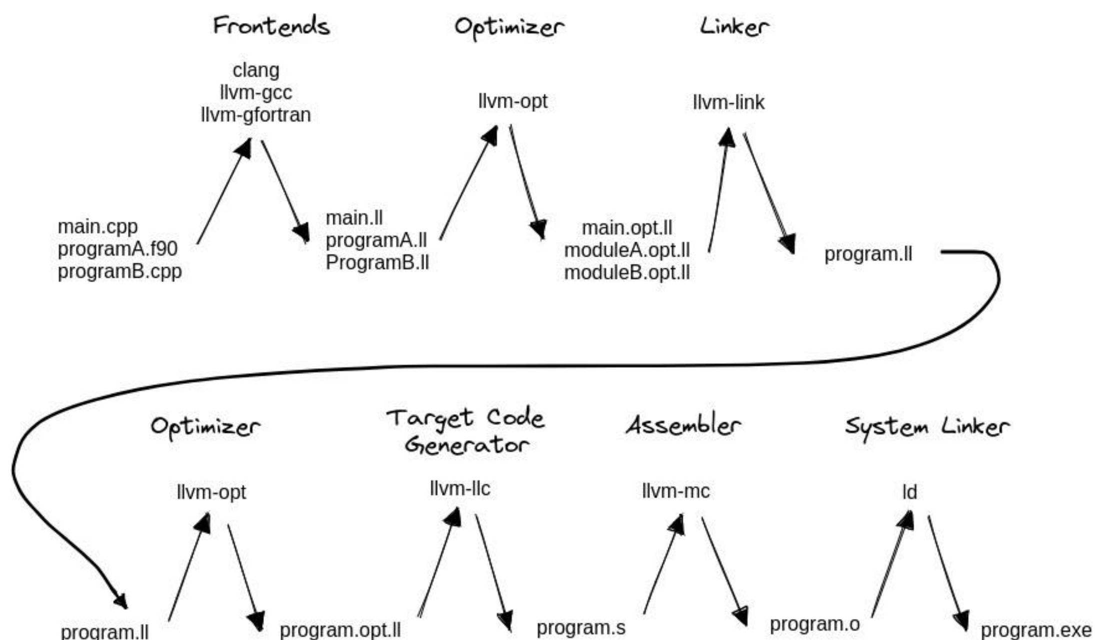


Figure 3.11 – Static compilation using the llvm toolchain.

Overall, LLVM provides a consistent infrastructure for the whole compilation process that is used in many important compilers. As all compilers target LLVM-IR as common intermediate representation LLVM is a great platform to write programming language

and target independent optimizations.

3.4.3 Parallel Control Flow Anomaly Checker (PARCOACH)

PARCOACH [85,50] is a framework that detects deadlocks caused by collectives in MPI, OpenMP, UPC and CUDA.
It detects errors in two steps. First, an interprocedural static analysis builds a parallel program control flow graph (PPCFG) in order to get interprocedural information. In the PPCFG, each function is replaced by its CFG. This analysis is performed in order to study the control flow of all the functions of a program to find statically the incorrect ones: functions containing potential deadlocks. In this step warnings are triggered with all conditionals that potentially can be the cause of a deadlock. If a potential deadlock is detected, all collectives that are found inside incorrect functions are instrumented in order to verify the potential deadlocks at execution time. Check functions are thus, inserted before all collectives and return statement of the program. In case of an actual deadlock situation at runtime, the program is stopped. A message displaying the error is shown to the user with compilation information.
An overview of the PARCOACH framework is given in figure3.12. The static analysis

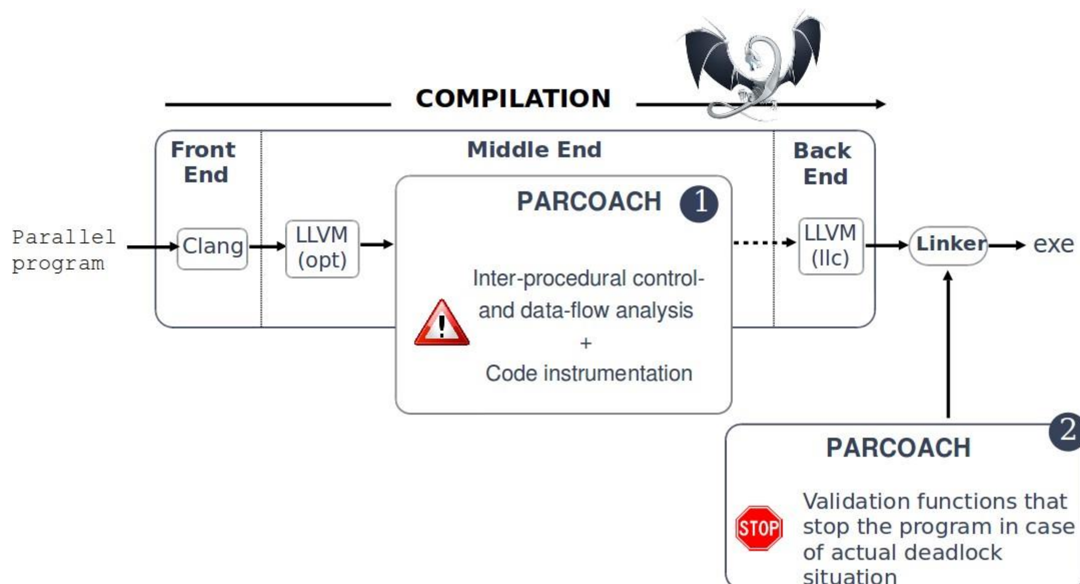


Figure 3.12 – Overview of the PARCOACH Framework.

is performed in the compilation chain (middle end). Check functions are implemented in a library.

Examples of potential incorrect situations are given in figure3.13.

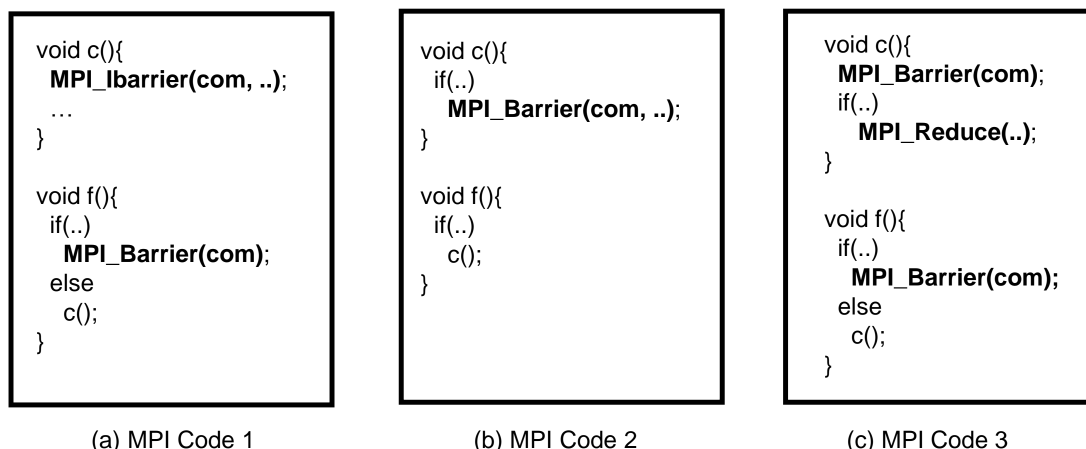


Figure 3.13 – Examples of MPI codes containing collectives.

All codes have two functions: f and c, collectives are written in bold. For the MPI code 1, PARCOACH finds a potential deadlock in function f because of the conditional (if) and triggers a warning about the MPI_ Barrier found the line right after (if). Indeed, depending on the conditional in f. The same error is reported for MPI_Ibarrier.

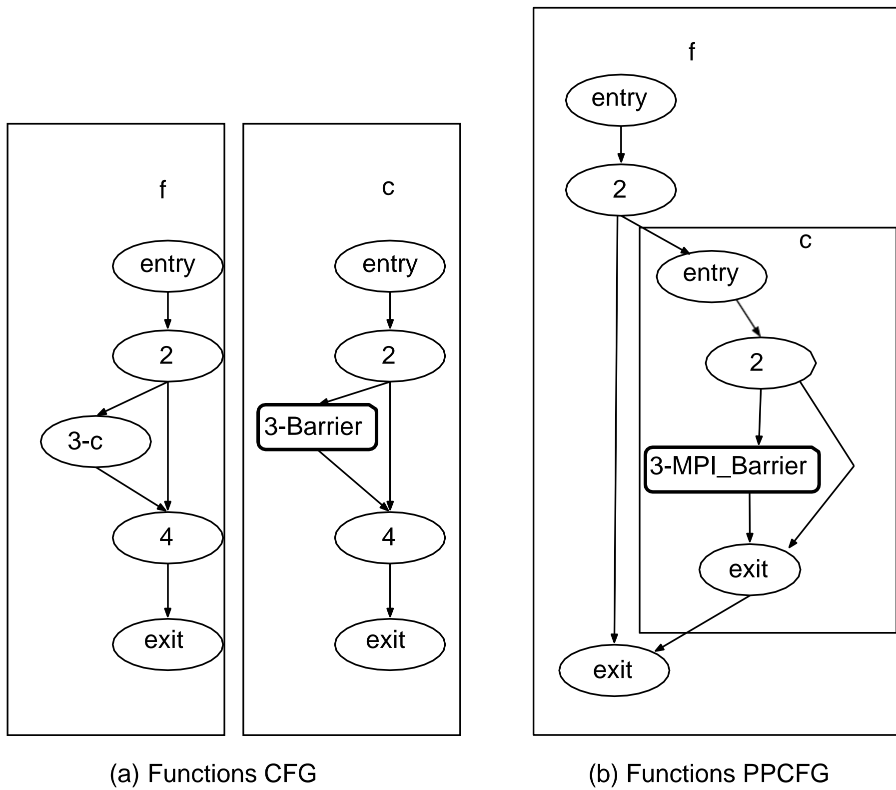
For the MPI code 2 PARCOACH identifies the conditional (if) in c as a potential deadlock, but not the conditional (if) in f. Parcoach finds the same collective error and will not report any problem in f either. Nevertheless, as there is no valid sequence of collectives, the summary of c is empty. And yet, the conditional (if) in f is also respon- sible for a potential deadlock.

In MPI code 3, PARCOACH will report an error for MPI_Reduce in c and MPI_Barrier in f. A potential error is detected for MPI_Reduce only because of the conditional (if) in c. PARCOACH returns conditionals (if) in c and f as potential deadlocks.

When a deadlock is about to occur, PARCOACH stops the execution and reports an error message with compilation information.

Figure3.14illustrates the PPCFG of the example MPI code 2 presented in figure3.13. The PPCFG is built based on the initial functions CFG presented in figure3.14(a). Thick nodes are collective nodes, boxes represent functions.

For the code (1) presented figure3.13, PARCOACH reports the message presented in figure3.15. This feedback helps fixing the deadlock. For example the non-blocking barrier can be easily replaced by a blocking one.



(a) Functions CFG (b) Functions PPCFG

Figure 3.14 – MPI Code 2 functions CFG (left) and the corresponding PPCFG (right).

```
$ PARCOACH : Error detected on rank 0
Abort is invoking line 8 before calling MPI_Barrier in MPIcode1.c
See Warning (s) : MPI_Barrier line 8 possibly not called by all
processes because of conditional (s) line(s) 7,
MPI_Ibarrier line 2 possibly not called by all processes because of
conditional (s) line (s) 7.
```

Figure 3.15 – Error output returned by PARCAOCH used for the code in figure3.13 (a).

The RMA-Analyzer like PARCOACH combines the two analyses to detect memory consistency errors in MPI-RMA programs. However, the algorithms are not the same. The RMA-Analyzer adopts the synchronization semantics imposed by the MPI standard.

3.4.4 The RMA-Analyzer Framework Overview

An overview of the whole RMA-Analyzer framework is given in figure3.16.

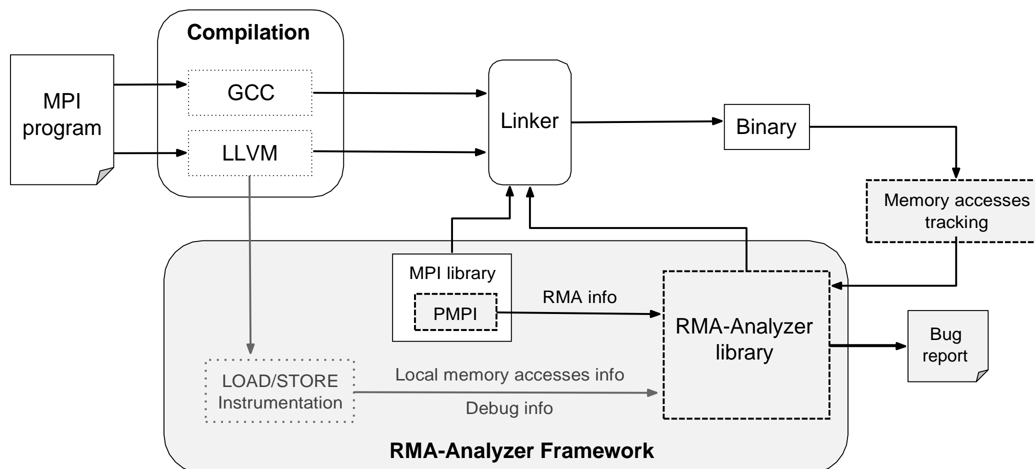


Figure 3.16 – Overview of the RMA-Analyzer Framework.

Collecting Memory Accesses The first step of our analysis is about collecting all memory accesses of a program. To do so, we use two methods, depending on which memory accesses are recorded.
To collect MPI-RMA routines information needed by our data race analysis, we use the PMPI interface. From the MPI_Win_create call, we get the size and the base pointer of the memory region exposed to other processes, and we start tracking it. From the MPI_Put and MPI_Get calls, we get the size and the offset of the remote access, the pointer of the local access, and their respective access types (read or write). We also instrument the beginning and the end of MPI epochs (i.e. MPI_Win_lock_all and MPI_Win_unlock_all in our case) to trigger and stop the recording of memory events respectively, and purge them at each end of epochs. Finally, we instrument the MPI_Win_free call to assess when to stop tracking the memory accesses on the associated MPI window. These pieces of information are then registered by the core of the RMA-Analyzer. While this solution is quite simple to implement and use in practice it only needs to be pre-loaded at runtime through the LD_PRELOAD environment variable, it does not consider local memory accesses (i.e., not from an MPI routine) and cannot detect all the errors listed in table3.1.

In addition to MPI-RMA routines information, we have developed an LLVM pass to instrument relevant load and store instructions. This enables us to detect all the errors listed in table3.1. Moreover, we implemented in this solution a compatibility with C and Fortran programs, which makes it desirable for analyzing production-quality codes that are often written in Fortran. However, it requires the user to rebuild its code to execute the LLVM pass, which can hinder its applicability for large-scale code bases.
A developer may use the GCC compiler instead of LLVM but will be limited to the detection of memory consistency errors between MPI-RMA operations only. Our LLVM pass needs to be adapted to instrument load/store instructions inGCC.

RMA-Analyzer Core The core of our RMA-Analyzer takes as input the memory accesses gathered by the instrumentation presented previously, and implements the data race detection algorithm. For those purposes, we implemented the memory interval itv as a structure containing the lower and upper bounds of the memory region, and its ac- cess type. Then, we implemented a BST of memory intervals, where we aggregate both local accesses (i.e. load, store and local data accesses due to MPI_Put and MPI_Get) and remote accesses due to remote MPI-RMA calls for each MPI window. This allows the tool to compare local accesses with remote accesses, enabling it to properly cover all error cases shown in table3.1.
Local access registration is quite simple. For load and store accesses, the routine that implements the data race detection is simply called for each MPI window that is tracked by the RMA-Analyzer at the time of the access. For MPI-RMA communications, the data race detection is simply called on the local access made by the communication. Remote access registration, however, is more complex. For each MPI window, we im- plement a routine handled by a dedicated POSIX thread that is spawned at window creation. It is tasked to poll for all incoming communications on a specific tag range dedicated to this window by the RMA-Analyzer, and within this tag range on a specific tag identifying the MPI epoch that is currently active. This tag-based implementation is made possible by the semantics of the MPI-RMA window creation and synchronization routines, that must be performed by all the involved MPI processes for the program to be correct. This weak synchronicity allows us to implement a tag-based communica- tion recognition system to identify all the control messages pertaining to a specific MPI epoch. Then, each time an MPI-RMA call is performed by the application program, the RMA-Analyzer adds a two-sided control message beside it (e.g. MPI_Send, MPI_Irecv) to send to the target the memory interval structure related to this memory access, with the appropriate MPI tag. Finally, when a control message is received, the thread that matches it calls the routine that implements the data race detection algorithm on its associated MPI window.

3.5 Implementation concerns

While this implementation successfully detects all memory consistency errors, we need to pay attention to the overhead of such analysis on the execution time of the

application. The first issue we tackle here is about instrumenting all load and store calls, that can become very costly at scale.
We implement a twofold solution in the RMA-Analyzer to filter the registering of these calls. The tool only registers memory accesses performed when an MPI epoch is opened on the tracked MPI window, i.e. only when MPI-RMA calls are legal on this window. The second issue is about the active polling of communications, made by the threads dedicated to receive incoming control messages from the RMA-Analyzer. While it greatly improves the reactivity of our tool, it can also severely hinders the performance of the application. In our implementation, we yield the thread every 100 calls to MPI_Test on the request associated to the MPI_Irecv call to release the thread as soon as possible and reduce the pressure on computational resources. It is also possible to reserve additional CPU resources so that the RMA-Analyzer threads can work on it without disturbing the application threads. Finally, it is noteworthy that while the Passive Target mode with MPI_Lock/Unlock_all is our main focus in this thesis, the RMA-Analyzer also supports the Fence model of the Active Target synchronization mode to ease the development of other MPI-RMA programs.

3.6 Experimental Results

3.6.1 Experimental Setup

Our experiments were performed on the Pise cluster that belongs to the Atos R&D department, located at Echirolles, France. Each node is composed of two AMD EPYC 7402 @2.8GHz 24-core processors with 128 GB of RAM. The nodes we used are linked with InfiniBand Mellanox 200 GB/sec (4X HDR) network cards. For the software con- figuration, we use a RHEL8.1 environment. Our software stack is built with LLVM 9.0.0. We use the OpenMPI implementation of MPI, built in its 4.0.5 version. The OpenMPI components are setup as follows:

—OMPI_MCA_pml=ucx

—OMPI_MCA_osc=ucx

—OMPI_MCA_btl= vader,openib,uct

3.6.2 Targeted Applications

We performed our experimental results on two applications : CFD-Proxy and Nemo.

CFD-Proxy [78] is a proxy-application for computational fluid dynamics. It imple- ments and evaluates the overlap efficiency for halo exchanges in unstructured meshes that requires indirect read/write access via the edges of the mesh to the actual mesh data.
For an efficient overlap of computation with communication, the application aims to trigger the communication as early as possible. When preprocessing the mesh, it an- notates with color all finalized points that belong to the mesh halo. The thread which completes the final update for a specific communication partner peer on these halo

points then triggers the corresponding communication either via MPI_Isend, MPI_Put or gaspi_write_notify. While this method allows for a maximal overlap of commu- nication and computation, it either requires a full MPI_THREAD_MULTIPLE or a MPI_THREAD_SERIALIZED of MPI version. For the latter version the developers have encapsulated the actual MPI_Isend and MPI_Put in an OpenMP critical section. The actual CFD kernel implements and evaluates the overlap efficiency of the two one- sided Active Target models and the two-sided model of MPI, and a GASPI version of the kernel. For our overhead evaluation, as we focus on this work on the Passive Target mode of MPI-RMA, we retrieved the Passive Target flavors implemented by Sergent et al. in [89] to test with the RMA-Analyzer. This application is of interest for overhead evaluation for MPI-RMA, as it is implemented in a full MPI_THREAD_MULTIPLE + OpenMP model, with all OpenMP threads performing communications in parallel. This means that any overhead introduced will strongly impact the performance, which makes it a perfect candidate to evaluate our RMA-Analyzer tool in the context of strongly optimized MPI-RMA applications.

Nucleolus for European Modeling of the Ocean (NEMO) [70] is a state-of-the art modelling platform for oceanographic research, operational oceanography, seasonal forecasts and climate studies. NEMO includes three major components; the blue ocean (dynamics), the white ocean (sea-ice) and the green ocean (ocean biogeochemistry). It also allows coupling through interfaces with atmosphere (through OASIS software), waves, ice-shelves, so as nesting through the adaptive mesh refinement software AGRIF [28]. Some reference configurations and test cases allowing to explore, to set-up and to validate the applications, and a set of tools to use the platform are also available to the community. The whole numerical ocean platform and its documentation are available under free licence. The evolution and reliability of NEMO are organised and controlled by a European Consortium between CMCC (Italy), CNRS (France), NOC (UK), UKMO (UK).Consortium members agree on long term strategy and yearly plans, sharing exper- tise and efforts within the NEMO System Team: the core team of NEMO developers in order to ensure the successful and sustainable development of the NEMO System as a well-organised, state-of-the-art ocean model code system suitable for both research and operational work.
We target this application because it is a representative candidate to tackle the appli- cability of our contribution for production-oriented applications. The specific kernel we target in this work is the tra_adv kernel (Tracer Advection). We retrieved a flavor of this kernel developed internally by Atos that uses MPI-RMA communications instead of two-sided ones for performance study purposes, and run our RMA-Analyzer with it.

3.6.3 Microbenchmarks

To highlight the functionality of the RMA-Analyzer, we created a micro-benchmark suite containing programs with correct and incorrect uses of MPI one-sided operations. This suite covers all error cases depicted in table3.1.
An example of error output returned by the RMA-Analyzer when run on the code of

figure3.7is shown on figure3.17. To help the user identifying the cause of the issue in its code, the file name and file lines of the accesses causing the error, the type of these accesses, and the MPI process on which the conflict has been detected are displayed to the user before aborting the program.

```
$ mpirun -np 3 ./rr_put_put
[RMA-Analyzer Process 1] Error when inserting memory access of type
RMA_WRITE from file remote_remote/rr_out_put.c at line 35 with
already inserted access of type RMA_WRITE from file
remote_remote/rr_out_put.c at line 35.
The program will be exiting now with MPI_Abort.
```

Figure 3.17 – Error output returned by the RMA-Analyzer tool used for the code of Figure3.7.

3.6.4 Runtime Overhead Impacts of Dynamic verification on CFD- Proxy

In order to show the overhead impacts of the RMA-Analyzer when using only MPI- RMA calls instrumentation (without load and store impact), we present in figure3.18a comparison of CFD-Proxy run with and without the RMA-Analyzer, on 2 nodes of the Pise cluster. For this first application, we only use the PMPI instrumentation of our RMA-Analyzer. We compare three distributions between MPI processes and OpenMP threads on this configuration, from 12 MPI processes and 8 OpenMP threads to 48 MPI processes and 2 OpenMP threads. We compare three flavors of CFD-Proxy in these experiments. The Comm Free (CF) flavor corresponds to a run of CFD-Proxy applica- tion where all the communications are assumed to be instantaneous and with negligible overhead. When comparing the different communication schemes, the CF accords to the practical maximum attainable performance. This provides the best execution time to compare with both implementations, as it takes into account the increasing number of MPI processes, which increase the computations due the halo exchanges. The two other flavors stands for Passive Target (PT) and Notified Passive Target (NPT). The PT flavor uses the MPI_Win_flush call to synchronize the communications. The NPT one adds a flag at the end of each sent data buffer, so that each process cancheck this flag to ensure the completeness of the operation. It emulates a notification system for communications, thus its name.

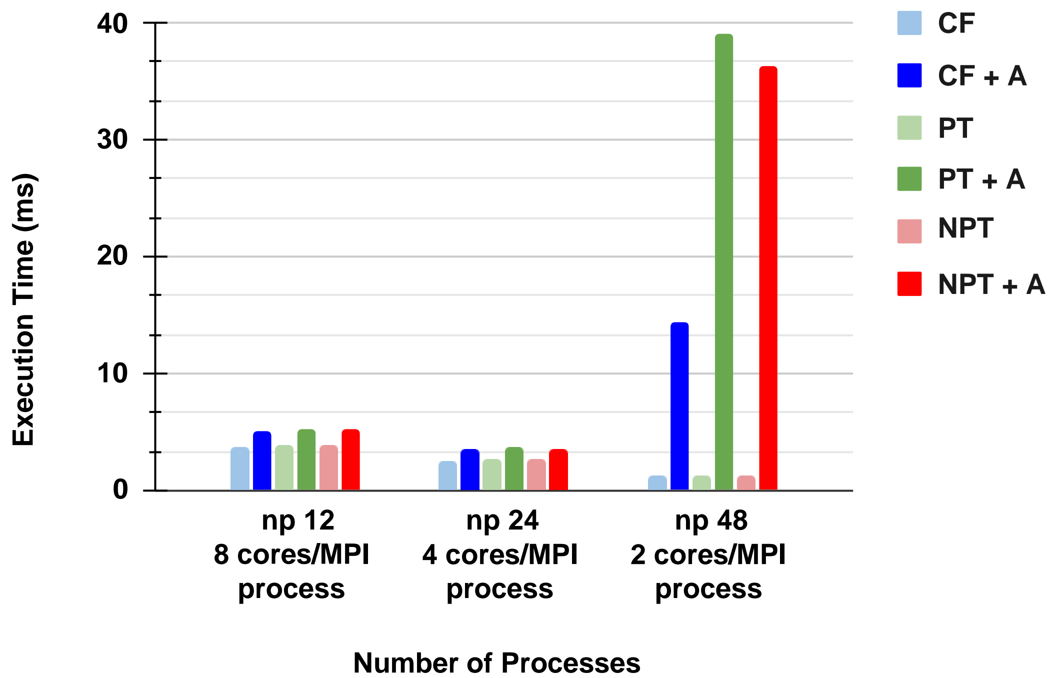


Figure 3.18 – Runtime overhead of the RMA-Analyzer on CFD-Proxy passive target with tree approaches. CF = Comm Free, PT = Passive Target, NPT = Notified Passive Target. "+ A" means execution time with the RMA-Analyzer.

We observe two specific behaviors in this figure. For 12 and 24 MPI processes, the

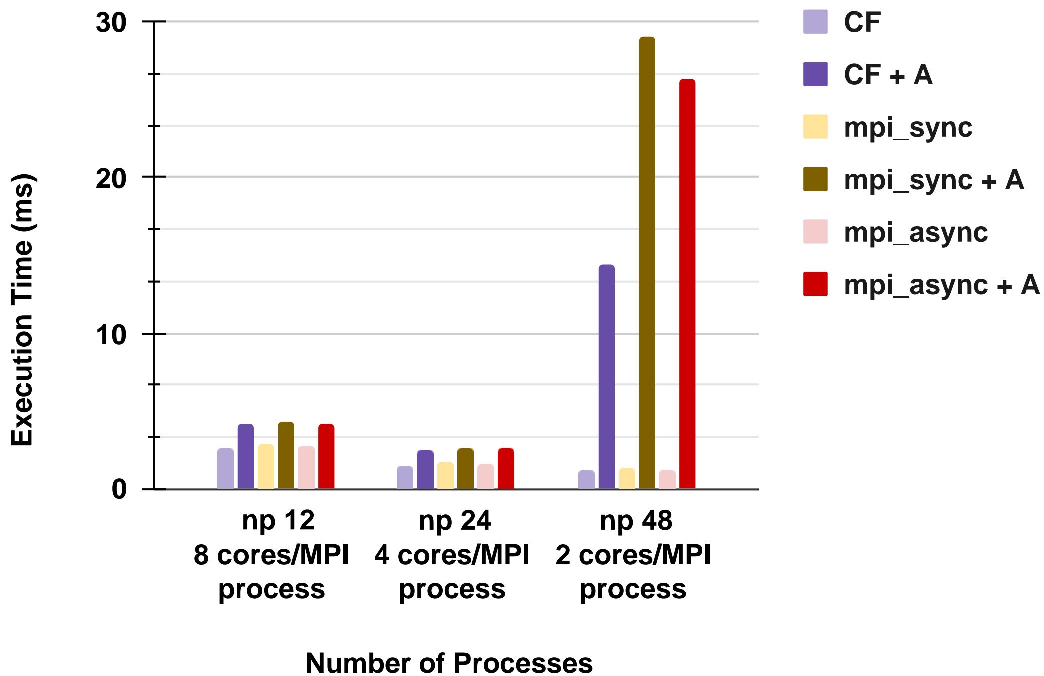


Figure 3.19 – Runtime overhead of the RMA-Analyzer on CFD-Proxy active target with tree approaches. CF = Comm Free, mpi_sync = Bulk fence synchronization version, mpi_async = MPI fence with early receives in asynchronous manner. "+ A" means execution time with the RMA-Analyzer.

overhead incurred by the RMA-Analyzer stabilizes around 40%. On these distributions, the threads of the RMA-Analyzer here 4 as CFD-Proxy uses 4 MPI windows in par- allel can use the hyperthreads of the cores used by the OpenMP threads to poll the communications, thus minimizing the impact on application threads. However, for 48 MPI processes, we see a severe degradation of the performance, with an execution time multiplied by 40 in the worst case. This is due to a lack of spare cores to use for the RMA-Analyzer threads (only 2 for 4 RMA-Analyzer threads), that will then compete with the application threads for the CPU resources. This also means that, if application developers can use spare cores during the design phase of their application, the cost of using the RMA-Analyzer remains reasonable. It is also noteworthy to recall that the overhead is not an issue for detecting errors, as the analysis is not dependent of the execution order: it detects all errors pertaining to a specific entry data set.
We also tested the active target flavors of CFD-Proxy with our RMA-Analyzer, with similar results as shown in figure3.19.

3.6.5 Runtime Overhead Impacts of Dynamic verification on NEMO

In order to show the overhead impacts of the RMA-Analyzer with its full instrumen- tation, i.e. the PMPI instrumentation plus the LLVM pass which instruments the load and store calls. We run NEMO with and without the RMA-Analyzer.
We present an excution time comparison of the tra_adv kernel with the RMA-Analyzer in figure3.20, on 4 nodes of the Pise cluster with two distributions. In the right one, we allocate 5 cores per MPI process, so that the 4 threads allocated by the RMA-Analyzer. As this kernel also uses 4 MPI windows for its computations the kernel can use spare

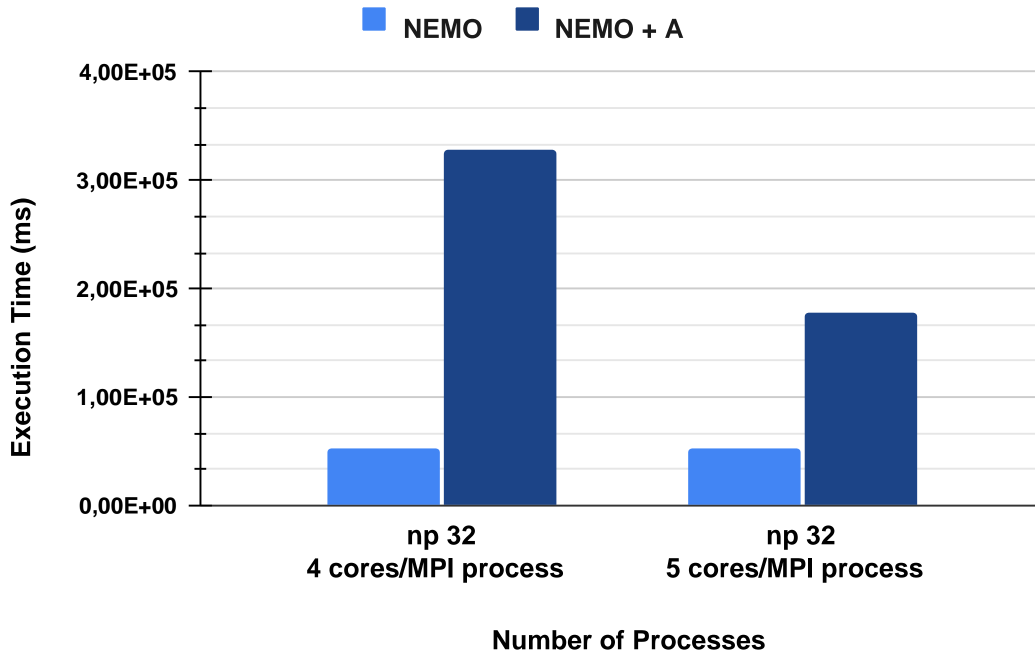


Figure 3.20 – Runtime overhead of the RMA-Analyzer on NEMO tra_adv kernel. "+ A" means execution time with the RMA-Analyzer.

cores and that won't disturb the main application's thread. We observe that, contrary to the CFD-Proxy experiments, the overhead is already quite high, with a $\tilde{3}$50% overhead. This is mainly due to the registering of the load and the store calls, which are performed by the main application's thread and heavily slows down the execution. For this code, the LLVM pass detects around 5.300 load and 2.300 store calls. In the second (left) distribution, where we remove one core per MPI process, we observe that the overhead grows to $\tilde{6}$00% of the original execution time. Similarly with the CFD-Proxy application, this is explained by the RMA-Analyzer's threads that competes for CPU resources with the main application's thread, thus disturbing it heavily.

3.6.6 Memory Consumption Impacts

In order to see the impacts of all previous collecting of memory accesses, the RMA-Analyzer has to store all transitions (i.e., the MPI-RMA and load/store calls) for each process. This consumes a considerable amount of memory. The problem was not very apparent when we tested the RMA-Analyzer with the benchmark suite, which made fewer than a dozen MPI calls in our testing. However when we tested the RMA-Analyzer on CFD-Proxy and NEMO; the runs exceeded all available memoryallocations.
The problem was attributed to the storage taken by RMA-Analyzer's Node structure which maintains the interval-tree data structure of transitions for each process. This structure grew quadratically.
In order to take a look at the overhead in terms of memory and complexity. table 3.3shows different statistics of our RMA-Analyzer tool for the three applications we presented in this section, from the validation test to the NEMO kernel. The third and fourth columns shows the memory fooprint of the RMA-Analyzer's BST and the user's MPI window respectively. This allows to compare the size in memory of the BST compared to the user memory region it tracks.
We observe that, while the BST size seems reasonable for the CFD-Proxy application,

Benchmark Appli	Language	BST memory size	User window memory size	Nodes in BST	BST Max depth
Validation test	C	0.08	0.39	2	2
CFD-Proxy	C	60	479	1500	61
Nemo	Fortran	5700	6490	142183	64893

Table 3.3 – RMA-Analyzer statistics on BST for each application. Memory sizes are in KB.

it becomes of the same order of magnitude than the user memory region for the NEMO kernel. This is explained by the additional tracking of the load and store calls, which adds a lot of information to register in the tree. We can also observe this inflation in the fifth and sixth columns, which display the number of nodes in the BST, and its balance (i.e. the maximum depth of the longest branch of the tree). While the number of nodes was around 1500 in CFD-Proxy, it goes up to 142183 nodes with NEMO. Moreover, the balance of the BST is worse with NEMO again, with the longest branch containing almost half of the nodes of the tree. This means that the BST may need to be rotated to ensure a good balance and reduce the complexity of inserting new nodes in it, and thus the overhead.

3.7 Conclusion

In this chapter, we introduced a method to address memory consistency and non-determinism challenges in MPI one-sided communication. We implemented these contributions in a tool called RMA-Analyzer, which will be integrated in the PARCOACH software.

Existing MPI-RMA verification tools have several limitations with respect to memory consistency. These limitations prevent the MPI runtime system from successfully scaling applications onto large-scale machines, because metadata for processes and outstanding operations uses up all internal resources. Our contribution provides new algorithms and strategies to manage different kinds of metadata in MPI one-sided communication by using a dynamic analysis. This method detects all possible errors that can occur in an MPI-RMA programs that we have shown in table3.1. Nevertheless, we have also shown that it comes with a significant overhead cost. That is why an additional static analysis is needed to better filter load and store accesses and reduces the overhead generated by the RMA-Analyzer due to the dynamic registering of load and store accesses. We detect all possible data race errors at runtime but some of them can be detected at compile time without executing the program. This is the main focus of the next chapter. We will discuss our static analysis that we have developed in order to detect only local concurrency errors at compile-time.

CHAPTER 4

STATIC DATA RACE DETECTION IN MPI-RMA PROGRAMS

The work that will be presented in this chapter has been published at the correctness workshop 2022 (TO APPEAR) [87]. The content is reproduced in this chapter.

4.1 Introduction

One-sided communications are often praised to be efficient to overlap communications with computations, but challenging to program. The use of one-sided communications is becoming increasingly popular. Applications often use them through abstractions, such as PGAS languages, that hides the complexity of the handling of one-sided com- munications in terms of memory consistency and performance optimizations. However, even expert runtime programmers can face issues providing a support for such languages when using runtime systems that implement one-sided communications support.

While being complex at program and use, tools that have tackled the issue of detect- ing memory consistency errors at compile time in MPI-RMA programs are no-existent (referred as local and global concurrency in [61]). In this chapter, we introduce a novel static analysis that enables programmers to detect local memory consistency errors di- rectly at compile time. While the dynamic analysis is exhaustive in terms of error detection, the static analysis will be more restrictive. Nevertheless, since the analysis is performed at compile time, it scales on large code bases and can be of use for produc- tion quality codes. The detection is based on a novel local concurrency errors detection algorithm that tracks accesses through BFS searches on the Control Flow Graphs of a program.

4.2 Motivating Examples

In order to see how a local concurrency error occurs at an origin processes either between only MPI-RMA accesses or mixing MPI-RMA and load/store accesses. Here we give some example in both active target and passive target synchronization epochs. Examples of these concurrency situations are shown : in example4.1, the first and sec- ond accesses are both MPI-RMA Get accesses which are write accesses in local window memory *X*, these two succeeded accesses in local memory conduct to a data raceerror. As these two communication operations are asynchronous the second Get that writes a value retrieved from *P1* can issue before the first Get and thus, update the value of **buf** before the first write access with a value retrieved from *P2*. The same is described in figure4.2. In this example the two MPI-RMA Put and Get accesses are considered as a data race error (Get as a write Put as a read) in the same memory location *Y* for the same reasons of example4.1. Another example is shown in figure4.3, a data race error occurs at the origin process *P2*. The MPI-RMA Get access conflicts with the store on **buf**. As MPI-RMA communication operations are asynchronous the value on **buf** can be updated before the Get returns which is an error in thiscase.

P0 (Origin)	**P1(Target)**	**P2(Target)**
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
MPI_Get(buf, 2, X, …)		
MPI_Get(buf, 1, X, …)		
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 4.1 – An example of local memory concurrency errors at origin process *P0* in passive target "Lock/Unlock_all" mode. Bold statements are conflicting memory ac- cesses(Get, Get) at origin side. X = Window memory location of *P0*.

Examples in active target "Fence" mode are also shown in order to prove that these local concurrency errors also occur even in a collective synchronization epoch.
Example4.4the MPI-RMA asynchronous Get and Put operations (respectively a write after a read on **buf** at the window memory location *X*) lead to a data race error. Since the completion of the first operation (Get) is not known the Put operation may read an old value of **buf** before sending it to the target.
Still in this active target mode example4.5shows another data race error between the MPI-RMA Get operation which is a write with the load on **buf** at the origin process *P1*. As explained previously the write operation of Get can happen at any time during this epoch. Thus the load on **buf** can finish before the issuing of the Get operation.
The last example in figure4.6also shows a data race errors between the MPI-RMA Put

P0 (Target)	P1 (Origin)	P2 (Target)
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
	MPI_Put(buf, 2, Y, …)	
	MPI_Get(buf, 0, Y, …)	
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 4.2 – An example of local memory concurrency errors at origin process *P1* in pas- sive target "Lock/Unlock_all" mode. Bold statements are conflicting memory accesses (Put,Get) at origin side. Y = Window memory location of *P1*.

P0 (Target)	P1 (Target)	P2 (Origin)
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
		MPI_Get(buf, 1, Z, …)
		buf = …
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 4.3 – An example of local memory concurrency errors at origin process *P2* in pas- sive target "Lock/Unlock_all" mode. Bold statements are conflicting memory accesses (Get with Store on **buf**) at origin side. Z = Window memory location of *P2*.

and the store on **buf**. The store can finish before the Put and thus, update wrongly the value of **buf** transferred by the MPI-Put.

Note that if the accesses were reversed i.e. the store before the Put or any native access before an MPI-RMA one there would be no errors, since the native load or store would have completed before the beginning of the Put.

P0 (origin)	**P1 (Target)**	**P2 (Target)**
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)
MPI_Get(buf, 1, X, …)		
MPI_Put(buf, 2, X, …)		
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)

Figure 4.4 – An example of local memory concurrency errors at origin *P0* in active target "Fence" mode. Bold statements are conflicting memory accesses (Get with Put) at origin side. X = Window memory location of *P0*.

P0 (Target)	**P1 (Origin)**	**P2 (Target)**
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)
	MPI_Get(buf, 2, Y, …)	
	… = buf	
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)

Figure 4.5 – An example of local memory concurrency errors at origin *P1* in active target "Fence" mode. Bold statements are conflicting memory accesses (Get with Load on **buf**) at origin side. Y = Window memory location of *P1*.

Correct solutionof codes one can correct their codes either by adding an "in-epoch" synchronizationcalllikeMPI_Win_flushinthecaseofapassivetargetlock_all/unlock_all only like in examples in4.1,4.2, and4.3or by ending the epoch and starting a new one to ensure all communications have ended in the case of active target fence mode4.4,4.5, and4.6. Figure4.7and4.8show the correct version of codes4.1and4.6.

On both of these codes, the static analysis successfully identifies the codes as correct after such changes.

P0 (Target)	**P1 (Target)**	**P2 (Origin)**
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)
		MPI_Put(buf, 0, Z, …)
		buf = …
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)

Figure 4.6 – An example of local memory concurrency errors at origin *P2* in active target "Fence" mode. Bold statements are conflicting memory accesses (Put with Store on **buf**) at origin side. Z = Window memory location Of *P2*.

P0 (Origin)	**P1(Target)**	**P2(Target)**
MPI_Win_lock_all(win)	MPI_Win_lock_all(win)	MPI_Win_lock_all(win)
MPI_Get(buf, 2, X, …)		
MPI_Win_flush(2, win)		
MPI_Get(buf, 1, X, …)		
MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)	MPI_Win_unlock_all(win)

Figure 4.7 – Correction of code4.1.

P0 (Target)	**P1 (Target)**	**P2 (Origin)**
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)
		MPI_Put(buf, 0, Z, …)
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)
		buf = …
MPI_Win_fence(win)	MPI_Win_fence(win)	MPI_Win_fence(win)

Figure 4.8 – Correction of code4.6.

4.3 Static Detection of Local Concurrency Errors

Local concurrency errors occur at one and single process locally and cause conflicts between native load/store and MPI-RMA accesses as shown in the previous examples. The goal of adding static analysis is to detect all errors that can be identified at compile time. In this thesis, we focus on all errors happening at the origin side of communications, where a program analysis coupled with an alias analysis is sufficient to find errors between overlapping accesses. Detecting errors at target side would be possible if the offset of the remote communication is completely known at compile time (in particular, if this offset is not dependent of the MPI rank), but this is out of the scope of this work.
To clearly state the errors that our static analysis can detect, we reuse the table3.1 and keep only the accesses at origin side, which results in table4.1. In this table, we suppose that memory accesses are performed on overlapping memory regions, and we show the compatibility of local load/store operations with local buffers of MPI-RMA communications. The table should be read as follows: the first access is read in row, the second in column.

		READ		WRITE	
		PUT	**LOAD**	**GET**	**STORE**
R	**LOAD**	✓	-	✓	-
	PUT	✓	✓	x	x
W	**GET**	x	x	x	x
	STORE	✓	-	✓	-

Table 4.1 – Compatibility of RMA operations and local load/store accesses on the same address space. ✓= overlapping is permitted,x= undefined behavior, overlapping is not permitted. R = READ, W = WRITE.

4.4 Local Concurrency Errors Detection Algorithm

To detect the errors presented in table4.1we devised a local concurrency detection algorithm that takes place in the middle of the compilation chain. It consists in an analysis of the Control Flow Graph (CFG). The CFG is defined as a directed graph. Nodes are basic blocks that represent maximal sequence of linear code and contains a list of instructions. Edges represent the flow of control between the nodes.
Based on this structure, Algorithm2details the steps to detect local concurrency errors in a function. First, a breadth-first search (BFS) is done on each loop of a function from the inner loops to the outer loops. Once a loop has been checked, the back edges are removed to avoid infinite loop during the BFS on an outer loop and the CFG. Finally a BFS is done on the entire CFG of a function. During the BFS, the memory accesses

are spread between basic blocks (from a basic block to its successors). This is madeby keeping a ValueMap containing (Value, Instruction) pairs. A *value* represents a memory access. Also, a basic block cannot be analyzed if all its predecessors have not been seen (function mustWait line 9 in Alg.3). The BFS is described in Algorithm3. The algorithm shows the BFS on a function but the principle is the same on loops. A coloring system is used during the graph traversal to identify unvisitednodes.

Algorithm 2 Local Concurrency Errors Detection

Require: CFG of function F

```
1: procedure DETECTION(Function F)
2:     for each Loop L in F do
3:         BFS(L)                                   ▷ (see Alg.3)
4:         Remove back edges in L
5:     BFS(F)                                       ▷ (see Alg.3)
```

Algorithm 3 Breadth-first Search

Require: CFG of function F, ValueMap MemInBB (contains {Value, Instruction} pairs)

```
1: procedure BFS(Function F)
2:     Set each BasicBlock BB in F at WHITE
3:     deque < BasicBlock > Unvisited
4:     Unvisited.push_back(F → getEntryBlock())
5:     while Unvisited.size() > 0 do
6:         header ← Unvisited.begin()
7:         Unvisited.pop_front()
8:         if header.color = BLACK then continue
9:         if mustWait(header) then
10:            Unvisited.push_back(header); continue
11:        UpdateMemAccesses(header) (see Alg.4)
12:        header.color = GREY
13:        for all successors Succ of header do
14:            if Succ.color = WHITE then
15:                Succ.MemInBB = header.MemInBB
16:                Unvisited.push_back(Succ)
17:                Succ.color = GREY
18:            else
19:                Succ.MemInBB.insert(header.MemInBB)
20:        header.color = BLACK
```

Algorithm4gives the analysis of a basic block. The algorithm iterates over the instructions of a basic block to find memory instructions (lines 2-3). A memory instruction is either a MPI_Put, MPI_Get, a LOADor a STORE. When a memory instruction is found,

we retrieve the memory location (line 4) and check if another access on this memory location has been recorded (line 5). Function find line 5 returns the previous memory access on mem or a memory access that alias with it. If there is no previous access, we register it in MemInBB but only if it is a one-sided operation (lines 6-7). We don't need to keep load/store instructions. If there is a previous access, two options arise. If the previous access or the new one is a write, we raise an error (line 9-10). Other- wise, we update the last instruction accessing mem if it is a Put (we don't register load instructions).

Finally, if a synchronization instruction (i.e., end of an epoch like MPI_Win_fence, MPI_Win_Flush, MPI_Win_Flush_all) is found, MemInBB is reset.

Algorithm 4 Analysis of a basic block

Require: ValueMap MemInBB (contains {Value, Instruction} pairs)

```
 1: procedure UPDATEMEMACCESSES(BasicBlock BB)
 2:     for each Instruction I in BB do
 3:         if I is a memory instruction then
 4:             mem ← getLocalMemAccess(I)
 5:             PrevAccess ← find(BB.MemInBB, mem)
 6:             if !PrevAccess AND I is an MPI-RMA then
 7:                 add (mem, I) in BB.MemInBB
 8:             else
 9:                 if isWrite(PrevAccess) OR isWrite(I) then
10:                     Raise an error
11:                 else
12:                     if I = MPI_Put then
13:                         PrevAccess ← I
14:         if I is a synchronization instruction then
15:             BB.MemInBB.clear()
```

4.4.1 Example of a Control Flow Graph

Figure4.9presents a CFG extracted from a benchmark computing a binary tree broadcast algorithm. This example contains a MPI_Get in a loop (node if.then9). Only relevant instructions are shown in the basic blocks. The algorithm first analyzes the loop {while.cond, while.body,if.then9, if.end11}. The BFS updates the memory accesses from the header of the loop (while.cond). MemInBBin the header is empty at the beginning of the BFS. The load instruction thus does not conflict with any other memory access. Nothing is registered when analyzing while.body. MemInBBis updated for if.then9 with the MPI_Get instruction and 9 which is the local memory location associated. When we encounter while.cond the second time, the load instruction is reported as conflicting with MPI_Get. MemInBBis updated for while.condwith the Get instruction. The algorithm then removes the back edge if.end11 → while.condfrom

the loop. A new BFS is performed on the entire CFG without the back edge. The loop is analyzed again to report concurrent accesses with the beginning of the graph if any. The Get function can conflict with itself at the second iteration of the loop. An error is then reported for that. No other local concurrency is detected.

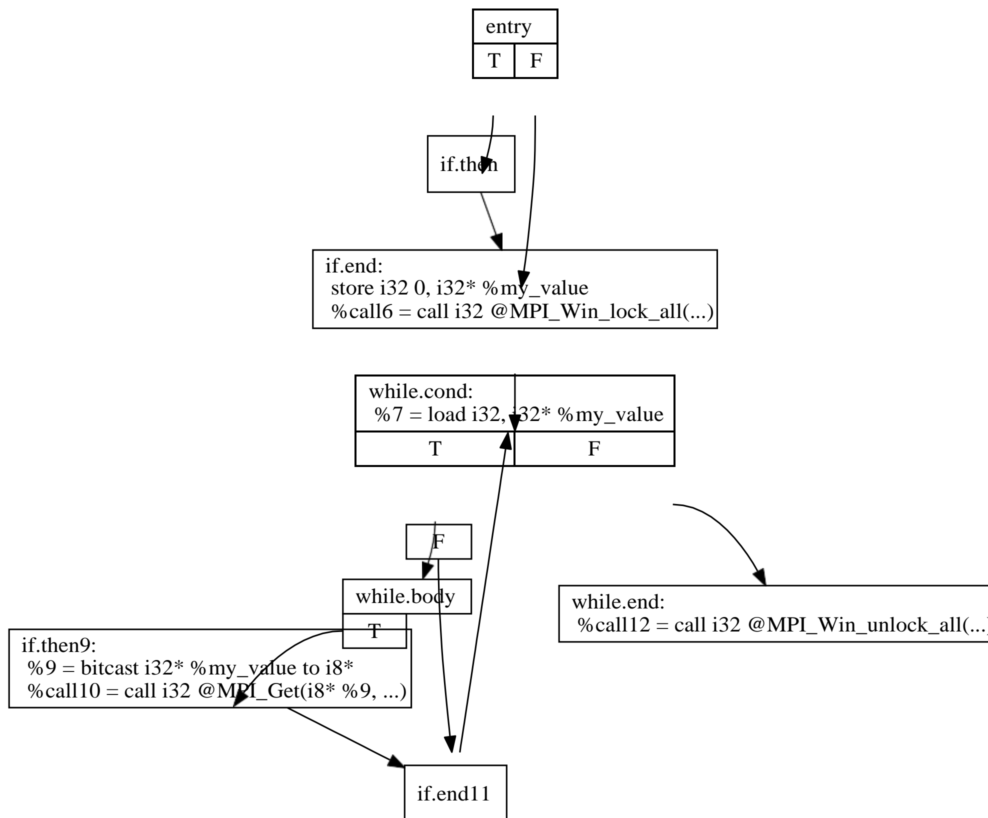


Figure 4.9 – CFG from a benchmark computing a binary tree broadcast algorithm.

4.4.2 Proof of the Algorithm

Algorithm2is correct if all local concurrency situations are captured. The main constraint we have is that our analysis is intra-procedural and is thus limited to the scope of a function. An inter-procedural analysis is left for future work. We then assume that a function contains all memory information needed to detect a data race. As our goal is to detect local concurrency errors only with the static analysis, we just

need to record memory accesses in a single epoch.
The first BFS on all loops ensures the detection of concurrency errors inside them and set the memory accesses in the headers of the loops. Then, the BFS on the entire CFG takes into account another iteration in loops and finishes because back edges are removed. All basic blocks are encountered and thus we register all memory accesses in a function.
The precision of the analysis is related to the alias analysis we use to detect aliasing. The alias analysis of LLVM is conservative and false positives are possible.

4.5 Experimental Results

Our analysis is implemented as a pass in the LLVM framework 9.0 and use Flang based on this version. We use the basic alias analysis of LLVM to detect aliasing on memory accesses. The pass detects errors in C, C++ and Fortran codes.
To highlight the functionality of our analysis, we created a microbenchmark suite con- taining small programs with correct and incorrect use of MPI-RMA written in C and Fortran. The suite contains 34 codes in total. table4.2shows a subset of the codes, the language in which they are written, if they contain a local concurrency and if our analysis was able to detect the error. The names of the codes correspond to the order of the operations in the code. The first 12 codes represent all scenarios intable4.1.
The second part of the table are more complicated codes, such as codes with loops, and the correct codes of figures4.7and4.8. Our static analysis was able to detect all concurrency errors and does not report any false positives on the microbenchmark suite. An example of the feedback reported by our analysis is presented in figure4.10.

Note that our microbenchmark suite has been approved to be integrated in the MBI project [61], a collection of correct and incorrect MPI codes.

```
$ mpicc -c -g -emit-llvm ll_get_get.c
$ opt -basicaa -load analysis.so ll_get_get.bc

[STATIC ANALYSIS] LocalConcurrency detected:
conflict with MPI_Get line 38 in ll_get_get.c and MPI_Get line 35 in
ll_get_get.c.
```

Figure 4.10 – Output returned by the analysis on the code figure4.1.

Program Name	Language	Local concurrency	Detect?
ll_get_get	C	yes	yes
ll_get_store	C/Fortran	yes	yes
ll_get_load	C/Fortran	yes	yes
ll_get_put	C	yes	yes
ll_put_get	C	yes	yes
ll_put_store	C/Fortran C	yes	yes
ll_put_put	C/Fortran C	no	no
ll_put_load	C	no	no
ll_load_get	C	no	no
ll_load_put	C/Fortran	no	no
ll_store_get		no	no
ll_store_put		no	no
ll_load_get_loop	Fortran	yes	yes
ll_get_get_ok	C	no	no
ll_put_store_ok	C	no	no

Table 4.2 – Results on our microbenchmark suite.

We also used our analysis on an experimental code of around 3500 lines of codes, written in C++. The code is based on the Global Update RandomAccess benchmark (AKA GUPS) [83], which updates memory at random locations according to a sequence of random numbers. The code we used is an MPI-RMA version of an existing code written in UPC++ [6], obtained from the UPC++ website atupcxx.lbl.govand was modified to (1) aggregate data to avoid overheads costs, (2) relax the look-ahead con- straint and (3) to run deterministically. A snippet of the code is presented in figure 4.11.

The code iterates over a set of batches of random numbers. Each number is routed to its corresponding target, but aggregated.

Then aggregated random numbers are put to the memory of a remote rank. The code has been simplified to show only relevant instructions and the communication scheme. Our analysis reports a local concurrency between MPI_Putline 21 and the store instructions *pa = A - 1and T[target * A + counter[targ]] = ranslines 19 and
12. This error happens because of the missing else statement in the conditional line 25. No flush is encountered if the conditional is false which lead to a concurrency at the second iteration of the for loop.

```
1     Window buffer upd
2   /* Create a 2D t a b l e T o f s i z e ( nranks x A) ; */
3   /* Create a t a b l e counter o f s i z e nranks ; */
4
5   for each batch {
6     for each x i n batch { 7
8       /* . . . */
9
10       if (target != myrank) {
11         //Insert rans in T at the right target:
12         T[ target * A + counter [targ ]] = rans ; 13
14         /* . . . */
15
16         if (counter [target] == A) {
17           uint 64 *pa = T + o f f s e t ; 18/* .
. . */
19           *pa = A − 1;
20           // Send the row in out buff table concerning target to target:
21           MPI_Put(pa, A, MPI_UINT64_T, target , my_koff + n_buffered , A, MPI_UINT64_T, upd);
22           // local_flush:
23           if (isFlushMode ())
24             MPI_Win_flush(target , upd);
25           else if (isFlushLocalMode () || isSrcComplete ())
26             MPI_Win_flush_local(target , upd); 27
28         }
29       }else{
30         /* . . . */
31       }
32     }
33     /* Sending the remaining updates ; */
34   }
```

Figure 4.11 – Code snippet from an MPI-RMA version of GUPS.

4.6 Conclusion

In this Chapter we have discussed our static analysis of MPI-RMA programs. We presented a novel static analysis that helps programmers debugging and understanding their MPI-RMA programs. While one-sided communications have some traction in the community, especially through PGAS languages it remains difficult for programmers to use directly runtime-level libraries that implements one-sided communications due to complex and error-prone memory semantics, that prevent them from obtaining the wanted performance gain.

This static analysis is based on a local concurrency errors detection algorithm that performs a double BFS search on the CFG of the program to tag all conflicting local

memory accesses (loads, stores and local buffers of MPI-RMA operations) and raises a compiling error when an issue is identified.
We implemented it in an LLVM pass and showed on small tests and on an MPI-RMA version of the GUPS benchmark that our analysis can detect local concurrency errors on such codes.

In the next and last Chapter, we will discuss the related work and emphasize those that are more closely related work to memory consistency errors detection.

CHAPTER 5

RELATED WORK

"The only good bug is a dead bug"
– Starship Troopers (1997)

5.1 Introduction

There have been a lot of recent interest in analysing and debugging MPI programs. A number of debugging and correctness tools have been developed and proposed over the years, some of which have been used to debug MPI programs while others to analyze MPI programs. Discovering bugs and analyzing MPI programs is a very challenging task especially for programs running on large scale and use a large degree of complexity such as real-life parallel applications. Luckily, there has been an enormous amount of research works that have been done on developing tools used in debugging and analyzing these type of applications, thanks to the large use and popularity of large clusters. In this chapter we survey some of those tools and their functionalities and highlight those that are more closely related work to ours. We discuss two kinds of related work. First, we discuss the work on debugging and correctness checking tools for MPI programs. Second, we group data race detection tools into two major categories: data race detection tools for MPI-RMA programs, and data race detection tools for shared memory programs.

5.2 Debugging and Correctness Checking Tools for MPI Programs

In this section we provide some MPI debuggers and correctness checking tools. We further split this category up into debuggers and correctness checking tools. The differ- ence between debuggers and correctness checking tools is that debuggers only provide a debugging interface and do not offer any error checking capabilities contrary to most of

correctness checking tools. However, correctness checking tools do not allow the devel- opers to interact with the MPI processes while they are running.

5.2.1 Debugging tools

The GNU Project Debugger(gdb)[96] is a well known debugger and the most commonly used tool among programmers. It was developed by Richard Stallman who played a key role in the the development of Linux. It is possible to use gdp as a support for MPI by attaching gdb to each MPI process of an application. We can use mpigdb
[75] in order to run the MPI processes with gdb, but the number of MPI processes is limited.
Valgrind[71] is a dynamic binary instrumentation (DBI) framework designed for build- ing heavyweight dynamic binary analysis (DBA) tools. Valgrind is a bug detector and can be used as gdp for debugging MPI applications. It provides a support for shadow values-a for DBA techniques which is difficult to implement. This tool executes pro- grams by using dynamic binary translation without requiring source code, and without the need for recompilation or relinking prior to execution. The major drawback of Val- grind is that it runs very slowly comparatively to other DBI framworks like Pin[64] and DynamoRIO[14] because of the support provided for several crucial design features. Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with the other DBI frameworks.
Tools like **DDT** [3] and **Totalview**[97,41] are often regarded as the gdb for MPI pro- grams. these tools provides usual functionality of debuggers. In fact, DDT attaches gdb instances to running MPI processes to provide debugging capabilities. These tools allow the users to step through MPI programs as they would with a normal C/C++ program. The users are provided with a host of useful debugging tools such as breakpoints inser- tion, procedures stepping, viewing the values of a variable across multiple processes, and obtaining stack traces. However, like gdb, they do not provide any correctness checking and only serve as debugging IDEs. As these tools do not require the recompilation or relinking of the source code, they are usually the only choices available if the user does not have access to the source. Furthermore, TotalView can be used to debug both se- rial and parallel programs. It supports several HPC platforms and can handle multiple types of HPC parallel coding (MPI, OpenMP, UPC and GA, OpenACC and CUDA). TotalView provides key features like a graphical data visualization and a memory leaks and malloc errors debugging.

5.2.2 Correctness Checking Tools for MPI Programs

MPI correctness checking tools are used to check if there are runtime errors by run- ning the MPI program or check for potential errors at compile-time in order to verify the correctness of the program. What we call by runtime errors is errors that occur during a given run. This is usually accomplished by recompiling and relinking the program.
We group the correctness checking tools for MPI programs into four categories: (i) online dynamic analyses, (ii) static analyses, (iii) trace based dynamic analyses and (iv) hybrid

approaches.

Online Dynamic Tools :

These tools checks for runtime errors that happen during the execution time. The ability of detecting MPI errors by these tools depend greatly on errors that actually occur in a given run only. Unfortunately, these tools do not take into account the al- ternative outcome due to the nondeterminism of MPI programs. That means that the nondeterminism induced bugs is still a challenge for these tools. A proposed solution is to run the program with the same test harness as many times as the computing re- sources permit. Unluckily, by doing so the performance endures severe degradation as it has been shown in some recent studies [103]. In this category of tools we present: **Umpire** [100] is a dynamic software testing tool developed at the Lawrence Liver- more National Laboratory (LLNL) by Jeffrey Vetter and Bronis de Supinski, Umpire is one of the first correctness checking tools for MPI. Despite not being actively main- tained, Umpire remains a useful tool for many MPI programmers. Umpire does not require recompilation of the MPI programs being checked, but it does require relink- ing the MPI program with its MPI profiling interface. At runtime, each MPI program launches several threads that communicate with the Umpire manager thread about the processes' MPI activities. The communication between the manager and the error check- ing threads rely on MPI itself, which means Umpire requires the MPI runtime to support MPI_THREAD_MULTIPLE. Umpire separates MPI error checks into local checks and global checks in which local checks include unfinished communication requests, unfreed communicators, uncommitted types, and bad arguments while global checks include deadlocks and type mismatch. In the most widely available Umpire version, deadlock checking is done through a simple dependency graph mechanism. A new deadlock de- tection mechanism that is based on Wait-For-Graphs and provides better scalability has been implemented as an experimental project for Umpire.
Marmot [57,56] is also an MPI dynamic checker which is similar to Umpire. This tool uses the MPI profiling interface to capture the MPI calls at runtime in order to analyze them. The error checking consists of local checks and global checks, similarly to those of Umpire. Each processes handles the local checks such as resource leaks and passes along the data to a debug server, which is a separate MPI process (Marmot requires one extra process to run the debug server), for global error checking such as deadlocks. In contrast with the previous tools, Marmot uses a simple timeout-based deadlock detec- tion scheme that has low overhead but can potentially produce false alarms. Marmot has extensive integration capabilities with other GUI tools to help the user visualize the checking results. Marmot has also common work with the following tools: Cube and Microsoft Visual Studio [58], DDT [55], Eclipse [33], and Vampir[15].
Must [48] is the successor of Marmot that combines the functionalities of the two tools (Marmot Umpire Support Tool). Must is a runtime MPI correctness checking. It was designed to overcome the limitations of scalability and extensibility of Umpire and Mar- mot and their hard coded trace communication with a centralized manager. Mustuses

a fine-grained module-based design that uses PMPI, it allows the offloading of checks to extra processes and threads. Further, by using a flexible communication system that promises an efficient transfer of trace records between different processes or threads.

Scalasca [47] is a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks in particular those concerning communication and synchronization and offers guidance in exploring their causes by providing useful information in order to diagnose the root of the issue.

MPIDD [46] has been developed for dynamically detecting deadlocks in parallel pro- grams that are written using C++ and MPI. MPIDD only offers deadlock detection capabilities. It uses a centralized approach in which a separate MPI process acts as a manager and communicates with other processes through TCP socket calls and builds a dependency graph based on the data that it receives from the processes. The tool uses a standard Depth-First-Search cycle detection algorithm to detect deadlock during runtime.

ISP [98,102] is a dynamic formal verification tool for MPI. ISP uses PMPI to intercept MPI operations and to enforce particular outcomes for non-deterministic operations. In particular, the ISP scheduler employs an MPI-semantics aware algorithm that reorders or rewrites MPI operations before sending them into the MPI runtime. ISP uses cen- tralized scheduling algorithm that is non-scalable and applying it to significantly larger process counts is infeasible. ISP must delay non-deterministic outcomes even at small scales, which leads to long testing times. In effect, its scheduler poorly exploits the par- allelism offered by the cluster on which the MPI program is being dynamically verified. **DAMPI** [101] is the successor of ISP, it is a dynamic analyzer for MPI programs that overcomes the performance limitations of ISP through a decentralized and scalable algo- rithm based on Lamport-clocks(vector clocks focused on call order). DAMPI computes alternative non-deterministic matches and enforces them in subsequent program replays.

Static Tools :

Static tools are mainly based on model checking and require symbolic execution of the program. These tools check the correctness of the program at compile-time. Static analysis is the automated testing of source code without executing it. It considers all possible paths of a program. It has the advantage of being input-independent. However, static analysis tends to provide false positives. We cite :

MPI-SPIN [91] is a model checker for verifying and debugging MPI-based parallel pro- grams. It detects deadlock related bugs. MPI-SPIN can show that the program does not deadlock by exploring all possible executions of an MPI program.

TASS [92] is the successor of MPI-SPIN that uses model checking and symbolic exe- cution. TASS in addition to deadlock errors verifies various safety properties as well as comparing two programs for functional equivalence and also if a propriety is violated such as an incorrect order of collective calls. TASS explores reachable states of the model that was built. The tool also gives a trace of the program in order to show the values of variables at each state of the given model.

MPI-Checker [32] is a static analysis checker that verifies the correct usage of the MPI API written in C and C++. It principally uses the Clang's Static Analyzer. MPI- Checker works with both path-sensitive (e.g., double request usage of non-blocking calls, missing wait) and non-path-sensitive analysis by using the information provided by the abstract syntax tree representation of the source code. MPI-Checker enables AST-based checks to verify correct type usage in MPI functions, and unmatched point-to-point calls, unreachable calls, type mismatch, invalid argument type, collective call in rank branch. **Nguyen et al.** [86] presents an extension of PARCOACH (PARCOACH will be pre- sented in the category that discusses hybrid approaches). This work presents a static analysis to detect misuse of nonblocking and persistent operations in MPI programs including MPI persistent collectives. This work proposes two algorithms based on the notion of generalized dominators and post-dominators to add new error detection. This analysis is built on top of LLVM **10** and it is based on data-flow analysis which is fully automatic and it is implemented as an LLVM pass. This static analysis can automat- ically find five type of errors: wrong management of the operation arguments, missing wait, unmatched wait, request overwriting and buffer data race.
Greg Bronevetsky [12] presents an approach that focuses on send-receive matching and dataflow analysis. This work provides framework from the compiler analysis of the program. It can extend traditional dataflow analyses to message passing applications. It works on an extended control-flow graph that includes all possible inter-process in- teractions of any numbers of processes. It provides a dataflow analyses built on top of this framework to gather information about the application's parallel behavior and communication topology. This framework is used for the communication model with an unlimited number of processes that communicate principally via send and deterministic receive operations. To analyze the parallel applications This analysis defines a parallel control flow graph (pCFG) and dataflow equations. A pCFG is an extension of CFG, the main difference is that a pCFG shows all possible control-flow states and transitions that may be performed by multiple sets of processes.

Trace-based Dynamic tools :

Trace-based tools rely on postmortem analysis. They provide a trace file to the developer that contains a run history information about the error that was generated during the program.
Dealing with trace based analysis we can mention:
Intel Message Checker (IMC) [29,90] is an MPI checker that provides postmortem analysis of MPI-related errors detected during program execution in a trace files. IMC has three main components: the Trace Collector that intercepts MPI calls using PMPI to collect information such as input parameters and message buffer checksum. the Analyzer Engine is used to read the trace files from the Trace Collector and analyzes them to check for MPI errors. The Visualizer that interprets the output from the Analyzer and allows the user to find the errors. IMC can detect common MPI errors such as deadlock, unsafe buffer access (i.e., accessing the buffer of a pending communication request), and type mismatches. IMC has limited scalability due to the trace size generated by the

program analysis. Furthermore, if a critical MPI error occurs and the program crashes, the behavior of the Trace Collector remains undefined, which means the user might not get the trace files.
Intel Trace Analyzer and Collector (ITAC) [74,51] is the successor of IMC, the two tools are part of Intel Cluster Tools (ICT) [95]. ITAC is designed for Intel MPI. Unlike IMC, TAC does not rely on postmortem analysis. Similarly to Marmot and Umpire, TAC distinguishes between local checks and global checks. The local checks return not only the line number in the source code but also provide a full stack trace. In contrast with how most tools handle global checks, TAC handles global checks in a distributed fashion and does not rely on a centralized approach. Instead, each process creates different TCP-based communication channels with all other processes and communicates with them through a predefined API. This mechanism allows TAC to detect deadlock as well as type mismatches. However, this independent communication layer potentiallylimits the scalability of the tool.

Hybrid Approaches :

Tools combining both methods have also been developed to detect errors in MPI programs. These methods leverage information about the control flow, in order to dynamically adjust the analysis. This leads to greater flexibility when balancing accuracy and performance, as well as enabling long-term and large-scale analyses that might not be possible with other techniques. We can cite :
PARCOACH [85] is a tool that combines static and dynamic analyses by mainly per- forming two major steps to detect incorrect collective patterns in MPI programs. For each function Parcoach first uses an llvm pass to statically identify the code fragments calling collectives that may deadlock at compile time and analyses the control-flow parts that may lead to this deadlock. If a potential deadlock is found Parcoach sends warn- ings to the user. Second, Parcoach transforms the identified code fragments in order to dynamically capture these situations before they arise. This work has been extended to an inter-procedural analysis [50] in order to detect collective errors in MPI , OpenMP and MPI+OpenMP applications and pinpoint their root causes.
Note here that the work of this thesis will fully be integrated in the Parcoach frame- work.
MPI-CHECK [63] is a tool developed to aid the programmer in the debugging of MPI programs that are written in free or fixed format Fortran 90 and Fortran 77. MPI-CHECK provides automatic compile-time and run-time checking of MPI programs. An experimental C/C++ version exists but is no longer in active development. MPI- CHECK instruments the source code of the program to replace MPI calls with MPI- CHECK's own versions. During the parsing of the source code, MPI-CHECK also checks the program for usage errors (e.g., using a negative number for the destination field in MPI Send). During execution, the MPI processes sends information of the execution to a centralized manager through the use of TCP sockets. MPI-CHECK can detect common errors such as deadlock, type mismatches between sends and receives, and under-allocated message buffer).

We show a survey of these correctness checking tools in table5.1.

MPI programs Verification tools			
Dynamic tools	Static tools	Trace Based On-line tools	Hybrid Approaches
Umpire Marmot Must Scalasca MPIDD ISP DAMPI	MPI-SPIN TASS MPI-Checker Nguyen et al. Greg Bronevetsky	IMC ITAC	PARCOACH MPI-CHECK

Table 5.1 – List of MPI correctness checking tools

5.2.3 Discussion

Some of these tools stated above are applied for MPI-RMA programs verification. Must and Marmot can be used for semantic parameter checking in order to detect errors which are caused by an erroneous sequence of MPI-RMA calls for example mismatched lock/unlock calls. Scalasca also can be used to detect inefficient wait states to highlight performance bottlenecks in MPI-RMA applications. However these tools can not uncover data race errors due to the nature of the problem tracked which is with respect to memory consistency and synchronization semantics of an MPI-RMA program. This is why we will discuss data race error detection tools separately.

5.3 Data race errors detection Tools for MPI-RMA and Shared Memory Programs

Here we present some tools that focus on data race errors detection for both MPI- RMA and shared memory programming models.

5.3.1 Data Race Detection Tools for MPI-RMA Programs

To help the programmer write correct MPI-RMA programs and help them to reason about memory consistency and data hazard accesses very few tools were proposed. In this part of the thesis we discuss related research focused on data race detection for MPI-RMA programs. We refer to:

MC-Checker[23] is a trace-based approach tool that focuses on detecting memory con- sistency errors in MPI-RMA programs. MC-Checker analyses the trace files to build a directed acyclic graph (DAG) based on the happens-before relation. MC-Checker detects memory consistency errors by profiling both MPI RMA and native load store accesses by using three main steps ST-Analyzer, Profiler and DN-Analyzer. Based on the MPI-2

semantics, it finds potential data races between different MPI processes which concur- rently access overlapping target memory. This analysis does not scale well because it takes much time when traversing the DAG and uses an incomplete utilization of the happened-before relation. Besides, MC-Checker only covers the MPI-2 standard which follows different synchronization semantics compared to MPI-3 and it focuses only on C programs. In addition, the tool provides many false positives and it is considered as its major limitation. Unlike MC-Checker our RMA-Analyzer performs an on-the-fly dynamic analysis to detect memory consistency errors between local and remote accesses and covers new functionalities introduced in MPI-3. Moreover, the RMA-Analyzer cov- ers C and Fortran programs.
MC-Cchecker[30] is also a trace based approach that is based on vector clocks in order to detect memory consistency errors in MPI-RMA programs. It works as an advanced iteration in order to enhance the MC-Checker analysis by taking full advantage of the encoded vector clock to replace the DAG with the aim of fully preserving the happened- before relation by making use of an encoded vector clock. MC-Cchecker inherits its main features from MC-Checker by reusing ST-Analyzer and Profiler while focusing mainly on the optimization of DN-Analyzer. It eliminates the potential source of false positives but still does not scale well because it reuses the same static analysis as MC-Checker. This tool is also based on a trace file approach which is different from our approach. Besides, this tool does not cover MPI-3 functionalities and Fortran programs.
NastyMPI [54,53] is another tool developed to address synchronization errors in MPI- 3 one-sided applications. Nasty-MPI also relies on program profiling. It dynamically intercepts RMA calls and reschedules them into pessimistic executions which are valid in terms of the MPI-3 standard. It mainly detects latent synchronization bugs. Based on the semantic flexibility of the MPI-3 specification the tool dynamically modifies execu- tions of improperly synchronized MPI remote memory accesses to force a manifestation of an error. Despite the fact that this tool is considered as a memory consistency error detection tool this approach forces synchronization errors rather than detecting them which is far different from our method.
Mi-Young et al. [77] presents an effort to uncover memory consistency errors dynam- ically by leveraging mirror memory. This approach also uses PMPI to create a mirror window whenever a process creates a window memory for one-sided communication, and uses it to monitor and detect race conditions. If any one-sided operation accesses a window memory, the corresponding mirror window will be marked and checked to see if the window memory is unsafely accessed by any concurrent operation. The number of entries in the mirror memory and the window memory is the same. However, the mir- ror memory takes less memory compared with the window memory. The justification for using the mirror memory is to know the current state of the corresponding window memory being ‘read’, ‘write’ or ‘no_op’, thereby, detecting memory consistency errors precisely. However, the on-the-fly approach is unfeasible in practice since each time it remotely writes (MPI_Put) or reads (MPI_Get) into the window memory of the target process, it needs to check the mirror memory first. For that reason, it is implemented by putting PMPI_Get inside both MPI_Put and MPI_Get via the MPI profiling inter-

face, which leads to performance depletion and even conflicts implementation principles of MPI one-sided communication. This tool can not be used for large MPI programs and unlike the RMA-Analyzer this tool doesn't handle the concurrent accesses between native load store and RMA operations because the scope of this research has not consid- ered local memory accesses. Hence, it cannot detect the conflicting operations between MPI communication calls and the local memory accesses.

To our best knowledge there is no tool that detects data race errors for MPI-RMA programs at compile time. The method that we presented in this thesis is the first to tackle memory consistency errors at compile time and presents a relevant support for the dynamic analysis of MPI-RMA programs.
table5.2summarizes memory consistency checking tools for MPI-RMA programs.

MPI-RMA Memory Consistency Checking Tools		
Dynamic tools	Static tools	Trace Based tools
RMA-Analyzer Nasty-MPI Mi-Young et al.	RMA-Analyzer	MC-Checker MC-Cchecker

Table 5.2 – List of MPI-RMA memory consistency checking tools

5.3.2 Data Race Detection tools for Shared Memory Programs

Several studies have been conducted to deal with data race errors in shared memory programs by performing static analyses, dynamic analyses or combining both static and dynamic analyses. Some tools are stated here:
RacerX [34] is a static tool that uses flow-sensitive, inter-procedural analysis to detect both race conditions and deadlocks in multi-threaded programs. RacerX is explicitly designed to find errors in complex and large multi-threaded systems. It checks several information such as operations protected by locks, it also checks code contexts if they are multi-threaded or not and the tool also verifies shared data accesses. It uses techniques to counter the impact of analysis mistakes by tracking a set of code features in order to sort errors from most to least severe.
LOCKSMITH [82] was developed to statically detect data races in C programs by look- ing for memory accesses violations. The tool looks for the relationship between locks and the locations they protect and the consistent correlation between them, this technique is a constraint-based analysis that infers consistent correlation context-sensitively, using the results to check that locations are properly guarded by locks. The tool uses several techniques to improve the precision and performance of the analysis, including a sharing analysis for inferring thread locality; existential quantification for modeling locks in data structures; and heuristics for modeling unsafe features of C such as type casts.
ERASER [66] is a data race instrumentation tool that uses several levels of static pro-

gram analysis ranging from little analysis to aggressive inter-procedural and dependence analyses. It uses this aggressive program analysis in order to prune the number of references to be monitored. The tool principally aims to reduce data race instrumentation overhead at compile time. It identifies variable references that not have the need to be monitored at runtime because these references cannot be involved in a data race.
Chord [68] is a static data race detection in shared memory programs in presence of locks. It is based on must not alias analysis. Chord reformulates the data race detection with a dual not alias analysis instead of the classical must alias analysis.
Kahlon et al. [52] propose a race warning generation system for data race errors which is based on a must alias analysis of data accesses in a shared program. It presents a dataflow algorithm for shared variable detection. The algorithm focuses on the precision of the pointer analysis used to compute aliases for lock pointers. It formulates a new context sensitive alias analysis that effectively combines a divide and conquer strategy with function summarization to provide a new technique in resolving data race errors. **LLOV** [11] is a static data race checker for shared memory programs based on the LLVM compiler framework. LLOV developed on top of the LLVM compiler framework and operates at LLVM-IR level, it can support a multitude of programming languages supported by the LLVM infrastructure.
PACER [10] is a dynamic tool that detects data race errors at runtime by finding races whose first occurs during a global sampling period. During runtime PACER tracks all accesses using the dynamically sound and precise FastTrack algorithm. At non-sampling periods, PACER skips sampled access information that cannot be part of a reported race thus, PACER simplifies tracking of the happens-before relationship to effectively report races found in the program.
Eraser data race detector [88] aimed to improve the static analysis provided by ERASER and provide a dynamic analysis. Eraser data race detector dynamically de- tects data races in lock-based multi-threaded programs. Eraser uses binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behavior is observed.
Jong-Deok Cho et al. [24] also presented an approach to dynamically detect data-race errors in multi-threaded object-oriented programs. This approach combines a static data race analysis, optimized instrumentation, runtime access caching and runtime detection phases.

5.3.3 Discussion

The aforementioned methods and tools are used to detect data race errors in shared memory programs by using a static or a dynamic analysis to find the data race error.

Several key differences can be distinguished between shared memory data race detection and detecting memory consistency errors in the MPI one-sided memory model. In the MPI RMA model, only the origin process can perform direct load and store ac- cesses. All other accesses are performed through library calls. In addition, in the MPI

RMA model, only buffers that are involved in MPI-RMA calls can be subject to mem- ory consistency errors, whereas, in shared-memory programs data race errors occur in any memory location. Because of theses differences, the bug detection algorithms are different. Since in MPI-RMA the data race detection algorithm is strongly related to the synchronization semantics that orders the memory accesses, our work focuses on tackling the MPI-RMA specific synchronization modes to devise a detection algorithm adapted to the constraints of these semantics.

5.4 Summary

The difficulty of debugging programs under the presence of data race errors has trig- gered many research efforts for multi-threaded programs as shown in this chapter but only few works have been done for MPI-RMA programs.
Non-deterministic behaviors and data race errors, when found in the program, make difficult the programming process. To this end a data race error tool is needed because in addition to analyzing the program, a data race detection tool has many other usages including performance analysis and relevant feed-backs of the data raceerror.

For MPI-RMA programs, finding data race errors is quiet complex because a data race error could be local to one process's memory location or present remotely on another process's memory location. In addition, in distributed memory programming the error detection becomes a lot more complicated because an MPI process has to locally and remotely keep the state of memory accesses and the information of the current run. This bookkeeping process requires tracking read and write accesses to all shared variables as well as enforcing the same order of accesses by the thread that preforms the permanent receive during this process. Since tracking every access to shared data is expensive, works that have been done in data race errors detection for MPI-RMA programs rely on post mortem analysis that uses a trace file based approach. The major drawback of this approach is that the trace data could be very large, especially for programs that exchange huge amounts of data. Thus, performing a complete analysis to detect data race errors by a post mortem analysis is very costly. These approaches do not scale well. For theses reasons our data race detection approach performs an on the fly dynamic analysis with a static analysis support to detect data race errors between local and remote accesses and covers new functionalities introduced in MPI-3 (lock_/unlock_all synchronization mode).

We have also presented some data race error detection tools for shared memory programming and the main difference resides in the handling of MPI-RMA specific syn- chronization semantics. Indeed, the data race condition problem in shared memory has been heavily addressed in the literature, but its handling is strongly linked to the syn- chronization model available to enforce the consistency between memory accesses. Since the data race detection algorithm is strongly related to the synchronization semantics that orders the memory accesses, our work differs from this approaches by tackling the

MPI-RMA specific synchronization modes to provide an adapted a algorithm to satisfy the specifications of these semantics.

CHAPTER 6

CONCLUSION

The MPI standard offers a rich set of features such as remote memory access primi- tives and nondeterministic constructs that help developers write better high performance applications.

Synchronization and communication are at the heart of one-sided programming paradigm, since processes are able to read from and write to any location in the lo- cal and global memory space during their execution without relying on communication activity of other processes. In general, this leads to concurrent accesses to shared data. Thus, data race errors are present in local and global memory locations.

This work presents a novel approach towards the detection of memory consistency errors in MPI-RMA programs. We have presented local and global data race errors that occur within a single process and among several processes in MPI-RMA programs. To tackle these issues, we provide a dynamic analysis technique for MPI-RMA programs in which we cover all possible data race errors. We have also developed a static analysis support to deal with local concurrency errors at compile time.

We implemented our data race detection approach in a tool called RMA-Analyzer. The RMA-Analyzer provides a global instrumentation of local load store accesses that detects conflicts between MPI-RMA operations and native accesses. We have measured the overhead costs of our tool on two real-world applications CFD-Proxy and NEMO. On CFD-Proxy we have shown the overhead cost that was reasonable with partial in- strumentation without taking into account the native load/store accesses. Nevertheless, the overhead cost increases drastically on NEMO because of the full instrumentation including load and store accesses. We successfully applied our method on two micro- benchmark suites that were developed in order to cover all possible data race errors including local concurrency errors for the static detection. In addition, our static anal- ysis reported local concurrency errors on an experimental code which is based on the

Global Update RandomAccess benchmark (GUPS).

As far as we know, this is the first time that MPI-RMA programs have been verified based on static and dynamic analyses. We believe that this work makes a step towards hybrid techniques on detecting memory consistency errors in MPI-RMA programs.

Finally, this thesis is part of a collaborative effort between industry and academia. Our contributions will be incorporated in the PARCOACH framework. PARCOACH actually detects deadlocks in MPI, openMP, MPI+OpenMP, UPC and CUDA programs. Our goal is to expand PARCOACH to detect memory consistency errors within MPI- RMA programs. To this end, a wide-ranging partnership for collaboration growth was discussed and negotiated between Inria and Atos. Furthermore, Atos is involved in the European DEEP-SEA project in which Atos contributes in the data race errors detection for MPI-RMA programs.

Future Research Directions

As future work, we plan to investigate some other lines of research directions :

First of all we plan to improve the overhead of the tool. As discussed in section3 the overhead of the load and store instrumentation can limit the scaling of the RMA- Analyzer. To deal with the overhead cost, we think of filling the BST with these calls before the execution with a static analysis. Once the static analysis is extended to detect other errors in addition to those that occur locally at the origin's memory location. For examples concurrent accesses from more than one process (as target) in the same mem- ory location. As our approach is based on a CFG traversal and it depends on the process rank, the offset of the remote communication is not known at compile time. We can miss some concurrent accesses among processes in different branches, this area needs further investigation. We can then, provide a complete hybrid method which can significantly reduce the overhead of the dynamic analysis. We can filter all the load store accesses at compile time. By detecting all the potential errors at compile time, the user will have the choice to perform only the static analysis without dealing with the overhead costs begotten at execution time. Another solution would be to delegate the registering of these calls to the RMA-Analyzer threads. Currently, the LLVM pass instrumentation of load and store calls is performed by the main thread of the application. Delegating this work to the RMA-Analyzer threads (or another dedicated thread) through a queue of requests to register memory access would be beneficial to reduce the overhead.

We can also investigate how to handle in-epochs synchronization (e.g. MPI_Win_flush) and atomic like operations (MPI_Accumulate and the put/get variations). Introducing the synchronizations is key to ensure the memory consistency of calls inside an MPI epoch without multiplying the epochs – thus the global synchronizations – inside the program. However, implementing the support of such synchronizations is not trivial.

The semantic of the MPI_Win_flush and MPI_Win_flush_local implies only the local completion (at the origin side). The origin knows that communications have been com- pleted which is not the case of the target process. This means that we need to introduce specific control messages to warn the target that communications from a specific origin has been completed (in the case of an MPI_Win_flush call). Moreover, with this han- dling, we may also give advice to users about where to introduce those synchronizations inside a bogus program to ensure memory consistency. For atomic operations, while we believe that their access rights can be described with the taxonomy explained in this thesis, the interactions between atomic and classical RMA operations can prove to be tedious to study and describe.

We can provide better feedback on the data race error by adding a solution that may be suitable for the programmer. We can improve the bug report and suggest simple and effective solutions such as recommending the use of additional synchronizations such as MPI_Fence in the case of active target synchronization or MPI_Flush (and variants) in the case of passive target synchronization.

We can also investigate how to transform programs written in MPI two-sided into MPI one-sided. We can provide options in how to rewrite these programs and ease the alteration two-sided to one-sided. With the aim of facilitating MPI-RMA programming, programmers will have support on their one-side programs and can easily adopt MPI- RMA semantics.

We have already done some work on synchronization optimization through previous work [89]. During my internship at Atos we have implemented notified synchronizations to replace passive target synchronization modes and warn the target as soon as any data has landed locally in its memory (communication and a notification sent at the same time). We can investigate how to extend our RMA-Analyzer to detect data race errors when using notified communications.

MPI-RMA comes with the promise of enhancing applications performance due to the overlapping of communication with computation by decoupling the data movement from synchronization. However, programming with MPI-RMA comes with several challenges as it has been shown in this thesis. The programmer needs to handle complex synchro- nization semantics and know how to use them in order to provide correct programs. I believe that the lock_all/unlock_all synchronization mode should definitely replace the other modes. This mode can be used as a global call (not collective). It can free the communications between peers and thus, processes can compute more. In addition, this mode is easy to use and can be seen as a global window opening at the beginning of the program, and globally close the window before freeing the window.

Publications :

Thesis Related Publications :

Dynamic Data Race Detection for MPI-RMA Programs [1]. Published at *EuroMPI in 2021*.
Static Local Concurrency Errors Detection in MPI-RMA Programs [87]. Published at *the Correctness Workshop 2022* (TO APPEAR).

Other Publications :

Efficient notifications for MPI one-sided applications [89]. Published at EuroMPI in 2019.

BIBLIOGRAPHY

[1] Tassadit Célia Aitkaci et al. “Dynamic Data Race Detection for MPI-RMA Pro- grams”. In: *EuroMPI 2021-European MPI Users’s Group Meeting*. 2021.

[2] Frances Allen et al. “Blue Gene: A vision for protein science using a petaflop supercomputer”. In: *IBM systems journal* 40.2 (2001), pp. 310–327.

[3] *Allinea DDT*. http://www.allinea.com/products/ddt/.

[4] Robert Alverson, Duncan Roweth, and Larry Kaplan. “The gemini system inter- connect”. In: *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE. 2010, pp. 83–87.

[5] Baba Arimilli et al. “The PERCS high-performance interconnect”. In: *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE. 2010, pp. 75–82.

[6] John Bachan et al. “UPC++: A High-Performance Communication Framework for Asynchronous Computation”. In: *2019 IEEE International Parallel and Dis- tributed Processing Symposium (IPDPS)*. 2019, pp. 963–973. DOI: 10.1109/IPDPS. 2019.00104.

[7] David E Bernholdt et al. “A survey of MPI usage in the US exascale computing project”. In: *Concurrency and Computation: Practice and Experience* 32.3 (2020), e4851.

[8] Dan Bonachea and Paul H Hargrove. “GASNet-EX: A high-performance, portable communication library for exascale”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2018, pp. 138–158.

[9] Dan Bonachea and Jaein Jeong. “Gasnet: A portable high-performance commu- nication layer for global address-space languages”. In: *CS258 Parallel Computer Architecture Project, Spring* 31 (2002).

[10] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. “Pacer: Propor- tional detection of data races”. In: *ACM Sigplan Notices* 45.6 (2010), pp. 255– 268.

[11] Utpal Bora et al. “Llov: A fast static data-race checker for openmp programs”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17.4 (2020), pp. 1–26.

[12] Greg Bronevetsky. “Communication-sensitive static dataflow for parallel message passing applications”. In: *2009 International Symposium on Code Generation and Optimization*. IEEE. 2009, pp. 1–12.

[13] Eugene D Brooks III. *PCP: A parallel extension of C that is 99% fat free*. Tech. rep. Lawrence Livermore National Lab., CA (USA), 1988.

[14] Derek Bruening, Qin Zhao, and Saman Amarasinghe. “Transparent dynamic instrumentation”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 2012, pp. 133–144.

[15] Holger Brunst et al. “Tools for scalable parallel program analysis: Vampir NG, MARMOT, and DeWiz”. In: *International Journal of Computational Science and Engineering* 4.3 (2009), pp. 149–161.

[16] David Callahan, Bradford L Chamberlain, and Hans P Zima. “The cascade high productivity language”. In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE. 2004, pp. 52–60.

[17] Bradford L Chamberlain, David Callahan, and Hans P Zima. “Parallel pro- grammability and the chapel language”. In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[18] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[19] Barbara Chapman et al. “Introducing OpenSHMEM: SHMEM for the PGAS community”. In: *Proceedings of the Fourth Conference on Partitioned Global Ad- dress Space Programming Model*. 2010, pp. 1–3.

[20] Philippe Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Acm Sigplan Notices* 40.10 (2005), pp. 519–538.

[21] Fabio Checconi et al. “Breaking the speed and scalability barriers for graph ex- ploration on distributed-memory machines”. In: *SC’12: Proceedings of the Inter- national Conference on High Performance Computing, Networking, Storageand Analysis*. IEEE. 2012, pp. 1–12.

[22] Dong Chen et al. “The IBM Blue Gene/Q interconnection network and message unit”. In: *SC’11: Proceedings of 2011 International Conference for High Perfor- mance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–10.

[23] Zhezhe Chen et al. “Mc-checker: Detecting memory consistency errors in mpi one-sided applications”. In: *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 499–510.

[24] Jong-Deok Choi et al. “Efficient and precise datarace detection for multithreaded object-oriented programs”. In: *Proceedings of the ACM SIGPLAN 2002 Confer- ence on Programming language design andimplementation*. 2002, pp. 258–269.

[25] Shane Cook. *CUDA programming: a developer’s guide to parallel computing with GPUs*. Newnes, 2012.

[26] David E Culler et al. “Parallel programming in Split-C”. In: *Supercomputing’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE.1993, pp. 262–273.

[27] Kaushik Datta et al. “Stencil computation optimization and auto-tuning on state- of-the-art multicore architectures”. In: *SC’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE. 2008, pp. 1–12.

[28] Laurent Debreu, Christophe Vouland, and Eric Blayo. “AGRIF: Adaptive grid refinement in Fortran”. In: *Computers & Geosciences* 34.1 (2008), pp. 8–13.

[29] Jayant DeSouza et al. “Automated, scalable debugging of MPI programs with Intel® Message Checker”. In: *Proceedings of the second international workshop on software engineering for high performance computing system applications*. 2005, pp. 78–82.

[30] Thanh-Dang Diep, Karl Fürlinger, and Nam Thoai. “MC-CChecker: A clock- based approach to detect memory consistency errors in MPI one-sided applica- tions”. In: *Proceedings of the 25th European MPI Users’ Group Meeting*. 2018, pp. 1–11.

[31] James Dinan et al. “An implementation and evaluation of the MPI 3.0 one- sided communication interface”. In: *Concurrency and Computation: Practice and Experience* 28.17 (2016), pp. 4385–4404.

[32] Alexander Droste, Michael Kuhn, and Thomas Ludwig. “MPI-checker: static anal- ysis for MPI”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–10.

[33] *Eclipse*.http://www.eclipse.org/.

[34] Dawson Engler and Ken Ashcraft. “RacerX: Effective, static detection of race con- ditions and deadlocks”. In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 237–252.

[35] Greg Faanes et al. “Cray cascade: a scalable HPC system based on a Dragonfly network”. In: *SC’12: Proceedings of the International Conference on High Per- formance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–9.

[36] *Flang and F18: Home Page*.https://https://github.com/flang-compiler/flang/ wiki/.

[37] Edgar Gabriel et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2004, pp. 97–104.

[38] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. “Enabling highly-scalable remote memory access programming with MPI-3 one sided”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.

[39] Tarek El-Ghazawi and Lauren Smith. “UPC: unified parallel C”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 2006, 27–es.

[40] James R Goodman. *Cache consistency and sequential consistency*. Tech. rep. Uni- versity of Wisconsin-Madison Department of Computer Sciences, 1991.

[41] Chris Gottbrath. “Eliminating parallel application memory bugs with totalview”. In: *Proceedings of the 2006 ACM/IEEE conference on supercomputing*. 2006, 210– es.

[42] William Gropp, Steven Huss-Lederman, and Marc Snir. *MPI: the complete ref- erence. The MPI-2 extensions*. Vol. 2. Mit Press, 1998.

[43] William Gropp and Ewing Lusk. *User’s Guide for mpich, a Portable Implemen- tation of MPI*. 1996.

[44] William Gropp et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789– 828.

[45] Daniel Grünewald and Christian Simmendinger. “The GASPI API specification and its implementation GPI 2.0”. In: *7th International Conference on PGAS Programming Models*. Vol. 243. 2013, p. 52.

[46] Waqar Haque. “Concurrent deadlock detection in parallel programs”. In: *Inter- national Journal of Computers and Applications* 28.1 (2006), pp.19–25.

[47] Marc-André Hermanns et al. “Understanding the formation of wait states in applications with one-sided communication”. In: *Proceedings of the 20th European MPI Users’ Group Meeting*. 2013, pp. 73–78.

[48] Tobias Hilbrich et al. “MUST: A scalable approach to runtime error detection in MPI programs”. In: *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[49] Torsten Hoefler et al. “Remote memory access programming in MPI-3”. In: *ACM Transactions on Parallel Computing (TOPC)* 2.2 (2015), pp. 1–26.

[50] Pierre Huchant et al. “Parcoach extension for a full-interprocedural collectives verification”. In: *2018 IEEE/ACM 2nd International Workshop on Software Cor- rectness for HPC Applications (Correctness)*. IEEE. 2018, pp. 69–76.

[51] *Intel Trace Analyzer*. http://software.intel.com/en-us/intel-trace-analyzer. pages28.

[52] Vineet Kahlon et al. “Fast and accurate static data-race detection for concur- rent programs”. In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 226–239.

[53] Roger Kowalewski and Karl Fürlinger. "Debugging Latent Synchronization Errors in MPI-3 One-Sided Communication". In: *Tools for High Performance Computing 2016*. Springer, 2017, pp. 83–96.

[54] Roger Kowalewski and Karl Fürlinger. "Nasty-MPI: Debugging synchronization errors in MPI-3 one-sided applications". In: *European Conference on Parallel Processing*. Springer. 2016, pp. 51–62.

[55] Bettina Krammer, Valentin Himmler, David Lecomber, et al. "Coupling DDT and Marmot for Debugging of MPI Applications." In: *PARCO*. Vol. 7. Citeseer. 2007, pp. 653–660.

[56] Bettina Krammer and Michael M Resch. "Correctness checking of MPI one-sided communication using Marmot". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2006, pp. 105–114.

[57] Bettina Krammer et al. "MARMOT: An MPI analysis and checking tool". In: *Advances in Parallel Computing*. Vol. 13. Elsevier, 2004, pp. 493–500.

[58] Bettina Krammer et al. "MPI correctness checking with marmot". In: *Tools for High Performance Computing*. Springer, 2008, pp. 61–78.

[59] Evgeny Kuznetsov and Vladimir Stegailov. "Porting CUDA-based molecular dy- namics algorithms to AMD ROCm platform using hip framework: performance analysis". In: *Russian Supercomputing Days*. Springer. 2019, pp. 121–130.

[60] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

[61] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. "The MPI Bugs Ini- tiative: a Framework for MPI Verification Tools Evaluation". In: *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Cor- rectness)*.

[62] *LLVM Language Reference Manual. 2016*. http://llvm.org/docs/LangRef.html/.

[63] Glenn Luecke et al. "MPI-CHECK: a tool for checking Fortran 90 MPI programs". In: *Concurrency and Computation: Practice and Experience* 15.2 (2003), pp. 93– 100.

[64] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dy- namic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[65] Ewing L Lusk, Steve C Pieper, Ralph M Butler, et al. "More scalability, less pain: A simple programming model and its implementation for extreme computing". In: *SciDAC Review* 17.1 (2010), pp. 30–37.

[66] John Mellor-Crummey. "Compile-time support for efficient data race detection in shared-memory parallel programs". In: *ACM SIGPLAN Notices* 28.12 (1993), pp. 129– 139.

[67] *MPI-2 over InfiniBand*. http://mvapich.cse.ohio-state.edu/.

[68] Mayur Naik and Alex Aiken. “Conditional must not aliasing for static race de- tection”. In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 327–338.

[69] Bruce Jay Nelson. *Remote procedure call*. Carnegie Mellon University, 1981.

[70] *NEMO : Nucleus for European Modelling of the Ocean*. https://www.cmcc.it/models/nemo/.

[71] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89– 100.

[72] Jarek Nieplocha and Bryan Carpenter. “ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems”. In: *International Parallel Processing Symposium*. Springer. 1999, pp.533–546.

[73] Robert W Numrich and John Reid. “Co-Array Fortran for parallel programming”. In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM New York, NY, USA. 1998, pp. 1–31.

[74] Patrick Ohly and Werner Krotz-Vogel. “Automated MPI Correctness Checking What if there was a magic option?” In: *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*. 2007, pp. 19–25.

[75] Masao Okita, Fumihiko Ino, and Kenichi Hagihara. “Debugging Tool for Localiz- ing Faulty Processes in Message Passing Programs”. In: *arXiv preprint cs/0310015* (2003).

[76] *OpenMPI:Open Source High Performance Computing*. http://www.openmpi. org/.

[77] Mi-Young Park and Sang-Hwa Chung. “Detecting race conditions in one-sided communication of MPI programs”. In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. IEEE. 2009, pp. 867–872.

[78] *PGAS-community-benchmarks*. http://github.com/PGAS-community-benchmarks/ CFD-Proxy/.

[79] Steven C Pieper and Robert B Wiringa. “Quantum Monte Carlo calculations of light nuclei”. In: *Annual Review of Nuclear and Particle Science* 51.1 (2001), pp. 53–90.

[80] Manoj Plakal et al. “Lamport clocks: verifying a directory cache-coherence proto- col”. In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. 1998, pp. 67–76.

[81] Marius Poke and Torsten Hoefler. “Dare: High-performance state machine repli- cation on rdma networks”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 2015, pp. 107–118.

[82] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. “Locksmith: context- sensitive correlation analysis for race detection”. In: *Acm Sigplan Notices* 41.6 (2006), pp. 320–331.

[83] *RandomAccess, HPC ChallengeBenchmarks*.http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[84] Yufei Ren et al. “Design and testbed evaluation of RDMA-based middleware for high-performance data transfer applications”. In: *Journal of Systems and Soft- ware* 86.7 (2013), pp. 1850–1863.

[85] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. “PARCOACH: Combining static and dynamic validation of MPI collective communications”. In: *The International Journal of High Performance Computing Applications* 28.4 (2014), pp. 425–434.

[86] Emmanuelle Saillard et al. “PARCOACH Extension for Static MPI Nonblocking and Persistent Communication Validation”. In: *2020 IEEE/ACM 4th Interna- tional Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE. 2020, pp. 31–39.

[87] Emmanuelle Saillard et al. “Static Local Concurrency Errors Detection in MPI- RMA Programs”. In: *2022 IEEE/ACM 6th International Workshop on Software Correctness for HPC Applications (Correctness)*.

[88] Stefan Savage et al. “Eraser: A dynamic data race detector for multithreaded programs”. In: *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997), pp. 391–411.

[89] Marc Sergent et al. “Efficient notifications for MPI one-sided applications”. In: *Proceedings of the 26th European MPI Users’ GroupMeeting*. 2019, pp. 1–10.

[90] Subodh V Sharma, Ganesh Gopalakrishnan, and Robert M Kirby. “A survey of MPI related debuggers and tools”. In: *Researchgate-Article* (2007).

[91] Stephen F Siegel. “Verifying parallel programs with MPI-Spin”. In: *European Par- allel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2007, pp. 13–14.

[92] Stephen F Siegel and Timothy K Zirkel. “Automatic formal verification of MPI- based parallel programs”. In: *ACM Sigplan Notices* 46.8 (2011), pp. 309–310.

[93] Mukesh Singhal and Ajay Kshemkalyani. “An efficient implementation of vector clocks”. In: *Information Processing Letters* 43.1 (1992), pp. 47–52.

[94] Guy L Steele Jr. “Parallel programming and code selection in Fortress”. In: *Pro- ceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 1–1.

[95] Alexander Supalov et al. *Optimizing HPC applications with intel cluster tools*. Springer Nature, 2014.

[96] *The GNU Project Debugger*.http://www.gnu.org/software/gdb/gdb.html/.

[97] *TotalView Software*.http://www.roguewave.com/products/totalview/.

[98] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M Kirby. “Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings”. In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 66–79.

[99] Stephen Van Doren. “HOTI 2019: compute express link”. In: *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2019, pp. 18–18.

[100] Jeffrey S Vetter and Bronis R De Supinski. “Dynamic software testing of MPI applications with Umpire”. In: *SC’00: Proceedings of the 2000 ACM/IEEE Con- ference on Supercomputing*. IEEE. 2000, pp. 51–51.

[101] Anh Vo et al. “A scalable and distributed dynamic formal verifier for MPI pro- grams”. In: *SC’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–10.

[102] Anh Vo et al. “Sound and efficient dynamic verification of MPI programs with probe non-determinism”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2009, pp. 271–281.

[103] Ruini Xue et al. “MPIWiz: Subgroup reproducible replay of MPI applications”. In: *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2009, pp. 251–260.

[104] Katherine Yelick et al. “Productivity and performance using partitioned global address space languages”. In: *Proceedings of the 2007 international workshop on Parallel symbolic computation*. 2007, pp. 24–32.

[105] Kathy Yelick et al. “Titanium: a high-performance Java dialect”. In: *Concurrency and Computation: Practice and Experience* 10.11-13 (1998), pp. 825–836.