UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS
HUAWEI PARIS RESEARCH CENTER

THÈSE présentée par :

Filip Arvid JAKOBSSON

soutenue le : 28 juin 2019

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline : **Informatique**

Static Analysis for BSPlib Programs

THÈSE DIRIGÉE PAR :

Frédéric LOULERGUE	Professeur, University Northern Arizona et Université d'Orléans

RAPPORTEURS :

Denis BARTHOU	Professeur, Bordeaux INP
Herbert KUCHEN	Professeur, WWU Münster

JURY :

Emmanuel CHAILLOUX	Professeur, Sorbonne Université, Président
Gaétan HAINS	Ingénieur-Chercheur, Huawei Technologies, Encadrant
Wijnand SUIJLEN	Ingénieur-Chercheur, Huawei Technologies, Encadrant
Wadoud BOUSDIRA	Maître de conference, Université d'Orléans, Encadrante
Frédéric DABROWSKI	Maître de conference, Université d'Orléans, Encadrant

Acknowledgments

Firstly, I am grateful to the reviewers for taking the time to read this document, and their insightful and helpful remarks that have greatly helped its quality.

I also thank my team of supervisors for guiding me through this PhD. Gaétan for his imagination and deep theoretical knowledge — but firstly, for his big heart. Wijnand for his impressive knowledge of parallel computing, his tireless proofreading, and for keeping my innate laziness in check. Wadoud and Frédéric D. for their rich insight in formal methods, and for our long discussions in front of the whiteboard. To Frédéric L., for directing this thesis with endless optimism, for overseeing the team of supervisors, and for his ever reliable advice. I hope our collaboration remains as fruitful in the future.

I am also grateful to my colleagues at Huawei France Research Center. Anthony, Thibaut and Jan-Willem, for sharing the delights and pains of the doctorate since we first came to Huawei. To Pierre and Filip that joined later on, but who merit no less mention. I wish you the best for the completion of your theses. To Antoine, Arnaud, Alain, Mathias and Louise, and to all other colleagues that I have failed to mention.

I also have debt of gratitude to Nikolai Kosmatov and Julien Signoles for supervising my master's internship at CEA. They initiated me to research, and gave me confidence to continue its pursuit.

I thank Linus, Henrik, Per and Jonas, for their friendship. As well as all the great friends I have made during my stay in France: Hai, Santiago, Michel, Lawrence, Leah, Anaiz, Lucile and Yu. To my family and parents-in-law, for being there, for supporting and for encouraging me. But finally, and foremost, to Laura. She made all this possible through her unwavering love and support, without which I would have long since abandoned.

Résumé étendu en français

Introduction

Les ordinateurs sont utilisés pour automatiser des calculs volumineux qui seraient hors de portée d'un humain. La recherche et le développement en informatique augmentent progressivement leur capacité à effectuer des calculs de plus en plus grands, sans épuiser la patience de l'utilisateur. Ce processus permet l'analyse mathématique, assistée par ordinateur, d'un ensemble toujours croissant de phénomènes naturels complexes. Pour citer un exemple parmi tant d'autres, les chercheurs ont utilisé l'informatique pour obtenir une meilleure compréhension de l'origine de l'univers et de la nature de la matière [97].

Dans certains cas, l'augmentation de la capacité de calcul permet de remplacer ou même de surpasser du matériel spécialisé. C'est le cas de la radio logicielle[1], composante essentielle des réseaux 5G [107], qui remplace le matériel de télécommunication personnalisé et qui permet d'atténuer le problème de rareté du spectre [177].

L'informatique parallèle[2] est une méthode importante pour obtenir de grandes capacités de calcul. Elle consiste à connecter plusieurs processeurs informatique via un réseau électronique, et à les programmer pour collaborer à la résolution d'une tâche commune. La plupart des ordinateurs modernes exploitent le parallélisme. Cela inclut les smartphones, comme le Huawei P30[3], qui comporte 8 processeurs. Mais aussi, des *superordinateurs*, comme le Summit [14], qui contient 2,4 millions de processeurs en réseau. Le Summit, entre autres utilisations, sert à effectuer des simulations du système terrestre qui produisent des prévisions pour le climat du futur.

Comme dans tous projets, lorsque d'avantage de ressources sont mobilisées, on s'attend à une efficacité plus grande. Ceci est également vrai dans le contexte

[1] En anglais, *Software-defined radio*

[2] Dans cette thèse, nous distinguons *le calcul parallèle* du *calcul concurrent*. Dans le premier, le parallélisme est utilisé de manière plus grossière pour exécuter simultanément des calculs reliés. Dans le dernier, le parallélisme est utilisé de manière fine pour exécuter simultanément des calculs non-reliés.

[3] `https://en.wikipedia.org/wiki/Huawei_P30`

de l'informatique parallèle. Quand on connecte plus de processeurs, on s'attend à une augmentation de la capacité de calcul proportionnelle aux ressources ajoutées. Hélas, ce ne sera pas nécessairement le cas. Une analogie peut être faite avec une organisation sociale. L'addition de ressources humaines ne résulte pas immédiatement en une organisation capable de faire plus de travail dans la même unité de temps. Cela est dû à l'effort induit par la distribution et la coordination du travail. Si fait maladroitement, l'addition de plus de travailleurs pourraient même diminuer l'efficacité de l'organisation. Il en va de même dans le calcul parallèle, où le travail doit aussi être distribué et coordonné entre les processeurs participants. Par conséquent, l'une des tâches fondamentales du calcul parallèle est de concevoir des architectures et des programmes adaptés afin qu'ils *passe bien à l'échelle*, c'est-à-dire que l'ajout de ressources provoque l'augmentation souhaitée de la puissance de calcul. Ceci est le sujet du *parallélisme évolutif*[4].

Cette tâche est réputée difficile. Dans cette thèse nous allons attaquer cette difficulté dans le contexte de BSPlib, une bibliothèque de programmation pour Bulk Synchronous Parallelism (BSP). BSP est un modèle de parallélisme avec des caractéristiques désirable en terme de structure, de sécurité et de performance. Nos armes de prédilection sont des outils de vérification automatiques appelés *analyses statiques*. Ces outils, spécifiés et éprouvés mathématiquement, appartiennent aux *méthodes formelles*. Dans la suite de ce résumé, nous illustrerons les difficultés de la programmation parallèle évolutive. Nous introduirons le modèle BSP, les méthodes formelles et l'analyse statique. Puis, nous énoncerons notre thèse et résumerons nos contributions qui argumente notre thèse, avant de conclure.

Défis de la programmation parallèle évolutive

Appliquer le parallélisme évolutif à un problème de calcul présente trois difficultés principale : concevoir l'algorithme, le mettre en œuvre correctement et mesurer sa capacité de passer à l'échelle.

L'algorithme est la séquence d'étapes nécessaires pour résoudre le problème. La conception d'un algorithme qui exploite le calcul parallèle, nécessite d'analyser le problème et de découvrir si, et comment, sa résolution peut être découpée et distribuée. Cela nécessite une idée créative qui est très spécifique à chaque problème. Cependant, dans cette thèse, nous nous concentrons sur les

[4]En anglais, *scalable parallelism*

deux difficultés restantes, c'est-à-dire la vérification de sa mise en œuvre et de sa performance.

L'implémentation correcte de l'algorithme est rendu difficile par les erreurs subtiles auxquelles la programmation parallèle est sujette. En plus des erreurs possibles en programmation classique, dite *séquentielle*, comme la division par zéro ou la déréférence d'un pointeur `NULL`, le parallélisme introduit une multitude de nouvelles erreurs. Celles-ci sont dues à de mauvaises l'interaction entre les processus ou à une coordination fautive.

Nous illustrons cela avec deux erreurs communes en calcul parallèle. Un *interblocage* est un type d'erreur impliquant au moins deux processus, A et B. Les deux processus exigent des informations l'un de l'autre pour procéder. Mais ce que A doit fournir à B dépend de ce que B doit fournir à A, et vice versa. La progression est bloquée et le calcul ne se termine jamais.

Une *data race* se produit lorsque deux processus tentent d'accéder, par lecture ou par écriture, à la même ressource, dont au moins un des accès est une écriture, et lorsque l'ordre avec lequel les accès se produisent n'est pas fixé. Dans le cas où un processus lit et un autre écrit la ressource, la valeur lue dépend de l'ordre des accès. Dans le cas où les deux processus écrivent la ressource, la valeur finale écrite dépend également de l'ordre. Ces situations sont indésirables et peuvent conduire à des erreurs de calcul subtiles, difficilement détectables et résolubles.

Les interblocages et les data races sont causés par des *entrelacements* imprévus d'exécutions parallèles. Quand plusieurs processus exécutent un flux d'instructions en parallèle, le nombre de possibles entrelacements de ces flux augmente de façon exponentielle. Le programmeur doit s'assurer que son programme est correct sous chaque entrelacement possible. On peut comparer cela à un jeu où le joueur (un processus) doit prévoir chaque coup possible des adversaires (les autres processus) et planifier sa réponse en conséquence. Cela devient rapidement impossible au-delà de quelques tours, et plus difficile encore de façon exponentielle en fonction du nombre d'adversaires.

Pour compliquer les choses, l'entrelacement de chaque exécution est *non déterministe*. Cela signifie que des exécutions différentes du même programme avec la même entrée peuvent engendrer des entrelacements différents : certains qui mettent en lumière des erreurs, d'autres non. La difficulté de reproduire les entrelacements erronés se traduit par une recherche de bug et une réparation difficile.

Troisièmement, après avoir conçu et développé correctement un algorithme parallèle, reste la tâche d'évaluer son efficacité par rapport à une solution sé-

quentielle. L'approche du benchmarking, c'est-à-dire exécuter et mesurer la durée de l'exécution du programme, ne donne que des indications pour une architecture parallèle et une instance du problème. Pour obtenir des résultats plus globaux, qui prédisent la performance pour toutes les instance lors de l'ajout de processeurs ou lorsque l'on passe à une architecture parallèle différente, il faut modéliser à la fois l'algorithme et l'architecture. Cette modélisation est difficile : le but est d'inclure uniquement les aspects essentiels pour la performance pour obtenir un modèle suffisamment simple, apte à l'analyse mais qui reste réaliste.

Ajouter plus de processeurs pour obtenir une capacité de calcul plus élevée donne donc, au mieux, une augmentation linéaire en efficacité avec chaque processeur. Mais, à la lumière de ces trois difficultés, elle vient au prix d'une augmentation exponentielle à la fois de la complexité conceptuelle et de la mise en œuvre.

Le modèle BSP

Bulk Synchronous Parallel[5] (BSP) [191] est un modèle pour la programmation parallèle évolutive qui aide à atténuer les problèmes discutés ci-dessus. De plus, ce modèle a été implémenté à la fois dans des librairies de programmation [100] et dans des langages dédiés [18].

Le calcul parallèle d'un programme BSP suit notamment une structure qui exclut à la fois les interblocages et les data races, grâce à des restrictions sur la synchronisation et la communication. En effet, dans BSP, le calcul est divisé en grands pas, appellés « supersteps ». A son tour, chaque superstep est divisé dans une phase de calcul locale, une phase de communication et une phase de synchronisation. Les interblocages sont évités puisque tous les processus se synchronisent en même temps, évitant la dépendance circulaire d'un interblocage. Les data races sont empêchées car les communications dans un calcul BSP (telles que les accès à une ressource commune) sont exécutées en vrac et en ordre fixe. Malgré ces restrictions, BSP permet l'expression d'une grande variété d'algorithmes parallèles [186].

Le modèle BSP facilite également le parallélisme évolutif en fournissant des prévisions de performance pour des programmes parallèles grâce à son modèle de coût. Ce modèle est simple mais réaliste. La performance d'une architecture parallèle est caractérisée par quatre paramètres :

[5]Parfois traduit en *parallélisme isochrone* ou *parallélisme quasi-synchrone* en français. Dans ce document nous écrierons « BSP ».

p le nombre de processus

r la caractérisation du coût local

g la caractérisation du coût de communication

l la caractérisation du coût de synchronisation

Le temps d'exécution d'un programme parallèle est caractérisé par une formule de coût. La formule est une fonction des paramètres de l'architecture qui décrit les ressources consommées par l'exécution du programme. La durée d'exécution estimée d'un programme est ainsi obtenue en appliquant sa fonction de coût aux paramètres de l'architecture où il sera exécuté. Ainsi, les formules de coût BSP sont portables : elles sont valables pour toutes architectures parallèles dans le modèle BSP.

BSP aide à atténuer certaines des difficultés de l'application du parallélisme. Mais ce n'est pas non plus une solution miracle. Dans cette thèse, nous porterons notre attention sur BSPlib, une bibliothèque de programmation pour la mise en œuvre des programmes BSP dans le langage général C. Nous aborderons en particulier certaines erreurs et problèmes courants affectant les programmes utilisants BSPlib.

Comme nous le verrons, ces problèmes résultent en partie de l'adjonction de parallélisme sur un langage généraliste sous la forme d'une bibliothèque de programmation. L'avantage de telles bibliothèques de programmation est qu'elles facilitent l'application de parallélisme dans des programmes séquentiels existants et, inversement, qu'elles permettent la réutilisation de code séquentiel dans des programmes parallèles. Cependant, la généralité de ces langages permet l'expression d'exécutions qui ne sont pas valides dans le modèle parallèle sous-jacent et qui sont donc erronées. Ceci peut être opposé aux langages spécifiques au domaine du parallélisme où de telles exécutions peuvent être restreintes. Nous proposons d'utiliser l'analyse statique, un type de méthode formelle, pour combler le fossé entre ces deux approches.

Méthodes formelles et analyse statique

Les méthodes formelles sont des techniques qui possèdent des fondements mathématiques rigoureux servant à la modélisation, la spécification, le développement et la vérification de programmes et de matériel informatique. *L'analyse statique* est un type de méthode formelle. Les analyses statiques sont elles-mêmes des

programmes informatiques qui ont pour but de découvrir des propriétés qui sont valables pour chaque exécution dans le programme analysé. Le mot *statique* fait référence au fait que les analyses statiques *n'exécutent pas* le programme. Ceci s'oppose aux approches *dynamiques*, comme celles basées sur des tests.

Notre thèse est que l'analyse statique peut et doit être utilisée pour vérifier l'absence d'erreurs dans les programmes BSPlib. En détectant des programmes écrits dans un langage général transgressant un modèle parallèle exploité à travers une bibliothèque de programmation, nous pouvons combiner les avantages des langages parallèles dédiés et des bibliothèques de parallélisme.

De plus, nous allons montrer qu'une majorité des programmes BSPlib est *structurée* de manière à garantir l'absence d'erreurs de synchronisation. Cette structure peut également être découverte par analyse statique et ensuite exploitée pour vérifier d'autres propriétés au-delà de la bonne synchronisation, en particulier de sûreté et de performance. Enfin, nos analyses statiques découvrent des propriétés qui sont valables dans un programme indépendamment du nombre de processus qui l'exécutent : cela garantit que les analyses elles-mêmes s'appliquent aux programmes développés pour des architectures futures.

Alignement syntaxique

L'élaboration de ces analyses est une tâche ardue : en effet, il ne suffit pas de vérifier la quantité exponentielle d'entrelacements pour un certain nombre fixe de processus. Au contraire, il faut supposer un nombre quelconque de processus, et vérifier tous les entrelacements possibles sous cette hypothèse.

Cependant, nous avons découvert que les programmes parallèles évolutifs et réalistes sont en général structurés. Cette structure limite les divergence entre la façon dont des processus différents exécutent les structures de contrôle du programme. De plus, cette structure réduit le nombre d'entrelacements dont l'interaction doit être vérifiée. Nous supposons que les programmeurs diligents ont un œil prudent sur des patrons de programmation qui augmentent la difficulté à mentalement exécuter leurs programmes, et donc naturellement écrivent des programmes qui sont structurés[6].

L'alignement syntaxique est un moyen de structurer des programmes parallèles autour d'*actions collectives*. Ce sont des actions qui nécessitent la participation de tous les processus. La synchronisation en barrière en BSP est un exemple d'une action collective. Des programmes syntaxiquement alignés sont écrits de façon à

[6]Cela fait écho à l'idée que la paresse est une vertu du programmeur [41].

ce que chaque action collective résulte de l'exécution d'une instruction au même point de programme par chaque processus.

La structure d'alignement syntaxique simplifie le raisonnement statique sur les programmes parallèles pour deux raisons. Premièrement, l'alignement syntaxique assure que chaque processus exécute la même séquence d'actions collectives. Par conséquent la vérification de l'utilisation correcte des actions collectives se réduit à vérifier que chaque séquence possible est correcte quand répliqué par tous les processus. Deuxièmement, elle limite les entrelacements aux séquences d'instructions qui séparent chaque paire d'actions collectives.

Dans les travaux précédents, l'alignement syntaxique a été utilisé pour vérifier la synchronisation [204] et pour améliorer la précision d'une analyse *may-happen-in-parallel* [121]. Notre thèse est que l'alignement syntaxique est également une base utile pour l'analyse statique des programmes BSPlib.

Nous détaillons ci-dessous nos trois contributions qui argumentent cette thèse. Premièrement, nous décrierons une analyse statique pour l'inférence de l'alignement syntaxique dans des programmes BSPlib, qu'on applique à la vérification de leur bonne synchronisation. Deuxièmement, nous expliquerons comment nous avons exploité cette même structure dans le développement d'une analyse statique de la performance des programmes BSPlib. Finalement, nous parlerons de nos travaux sur l'enregistrement en BSPlib. *L'enregistrement* est un composant important dans le système de communication en BSPlib. Notre contribution finale est une condition suffisante pour l'utilisation correcte de l'enregistrement basée sur l'alignement syntaxique.

Synchronisation répliquée

Nous abordons la thèse par le développement d'une analyse statique sous-approchée pour la détection des points syntaxiquement alignés dans un programme BSPlib. L'idée derrière cette analyse est de suivre les valeurs, variables et expressions qui sont dépendants du *pid*, une expression symbolique en BSPlib qui identifie uniquement chaque processus. Dans notre formalisation, c'est uniquement cette expression qui peut être évaluée différemment dans la mémoire initiale. En suivant les dépendances de cette expression, on peut détecter les structures de contrôle dans le programme qui peuvent engendrer des divergences dans le flux de contrôle entre chaque processus, mais aussi les points de programme qui ne sont pas impactés par ces divergences. Ces derniers sont aussi ceux qui sont syntaxiquement alignés.

Cette analyse est spécifiée grâce à une formalisation de BSPlib minimaliste :

un langage séquentiel de type `WHILE`, étendu avec une sémantique BSP pour permettre l'exécution parallèle et une primitive de synchronisation. Ainsi, un programme de ce langage peut engendrer des erreurs dynamiques par l'utilisation non-collective de la synchronisation. Un exemple simple est donné par ce programme court : `if` `pid` $= 0$ `then` `sync` `else` `skip` `end`.

Dans les contributions suivantes, nous étendrons cette formalisation pour modéliser un sous-ensemble plus large de BSPlib. Cependant, cette version suffit pour démontrer que l'alignement syntaxique de chaque point de programme qui correspond à une primitive de synchronisation est un garant pour la bonne synchronisation. Nous avons démontré cette propriété dans l'assistant de preuve Coq.

Enfin, pour vérifier l'applicabilité de cette analyse, nous l'avons implémentée en Frama-C, une plate-forme d'analyse des programmes C. Cette implémentation étend la formalisation de l'analyse en traitant des fonctions. Nous avons également implémenté un traitement des pointeurs et de la communication. Sur un échantillon de 20 programmes BSPlib, nous avons pu vérifier la bonne synchronisation de 17 grâce à cette analyse, et trouver des erreurs de synchronisation dans les 3 restants.

Analyse de coût

Nous nous intéressons ensuite à la question de la prévisibilité de performance. Le modèle de coût de BSP donne un cadre pour raisonner sur la performance des programmes. Bien entendu, ce modèle s'applique également aux programmes BSPlib. Cependant, son utilisation nécessite une analyse manuelle des programmes pour inférer leurs formules de coût.

Dans cette contribution, nous avons automatisé cette inférence. D'abord, nous étendons notre formalisation de BSPlib pour inclure les communications et nous caractérisons le coût BSP dans cette formalisation. Nous avons développé une transformation des programmes impératifs BSP en programmes séquentiels de façon à ce qu'ils simulent de manière non déterministe un processus différent à chaque superstep. Cette transformation exige et exploite le fait que les programmes analysés ont une synchronisation syntaxiquement alignée et elle nous permet d'utiliser une analyse de coût classique adaptée pour des programmes séquentiels de façon à faire ressortir le coût de calcul BSP pour des programmes parallèles.

Néanmoins, le coût de communication nécessite une analyse plus fine, car, dans le modèle BSP, ce coût dépend du comportement de l'ensemble des pro-

cessus. En se basant uniquement sur la transformation décrite ci-dessus, nous sommes obligé de faire des assomptions conservatrices punitives. Pour contourner cette problématique nous utilisons le modèle polyédrique pour analyser la communication, et nous insérons ensuite ces coûts dans le programme transformé sous forme d'annotations.

Cette analyse a été implémentée dans un prototype que nous avons utilisé pour obtenir la formule de coût de 8 algorithmes BSP classiques. Nous avons testé empiriquement que ces formules donnent bien des bornes supérieures au coût en comparant leurs prévisions avec le coût réel, spécifié par notre sémantique. C'est effectivement le cas, sauf pour l'un des programmes, dont le motif de communications dépend des données qui sont communiquées.

Après avoir vérifié que nos formules de coût BSP étaient valides, nous avons vérifié leur utilité pour prévoir la durée des calculs dans un cadre réaliste. Nous avons comparé le temps prévu avec celui mesuré dans deux architectures parallèles : un ordinateur de bureau multi-cœur et une grappe d'ordinateurs. Dans des configurations qui remplissent l'hypothèse du modèle BSP sur le réseau de communication, nous avons confirmé la précision de ces prévisions avec une erreur inférieur à 50%.

Enregistrement sûr

Enfin, nous retournons à la vérification d'absence d'erreurs dans les programmes BSPlib. Logiquement, ces programmes exécutent toujours en mémoire distribuée, et communiquent en réalisant des écritures et lectures à distance. L'enregistrement est une procédure collective préalable aux accès, qui permet de relier des objets de mémoire (variables, allocations dynamique etc.) existant dans des processus différents. Malheureusement, ce mécanisme comporte plusieurs écueils, et son utilisation nécessite une programmation prudente pour les éviter.

C'est pour cette raison que nous proposons une condition suffisante pour l'enregistrement sûr. Une telle condition forme la base formelle d'une analyse statique. Pour résumer, notre condition stipule que, sous l'hypothèse que les enregistrements et les désenregistrements soient syntaxiquement alignés, et que chaque enregistrement concerne le même objet dans chaque processus, la correction locale implique aussi la correction globale.

Nous spécifions cette condition grâce à une nouvelle formalisation de BSPlib, cette fois étendue avec allocation dynamique et pointeurs ainsi qu'enregistrement et communication. Nous caractérisons les exécutions qui sont correctes par

rapport l'API de BSPlib, et nous prouvons formellement que notre condition suffisante est un garant de cette correction.

Conclusion

Contexte

Le calcul parallèle est un élément important pour obtenir de hautes capacités de calcul. Il existe de nombreux domaines d'application qui profitent de l'application du parallélisme, notamment dans le domaine des sciences naturelles où une précision croissante dans les simulations permet une compréhension plus précise de sujets aussi divers que l'origine de l'univers et la composition de la matière [97], le fonctionnement du système terrestre [137] — avec des implications importantes pour la science du climat, la formation des galaxies [64], ou encore le cerveau humain [136].

Cependant, encore plus que le calcul séquentiel, le calcul parallèle est chargé d'erreurs. Les difficultés bien connues du développement correct des programmes séquentiels, et les effets désastreux quand il est traité à la légère (les exemples spectaculaires abondent [65]), sont exacerbés par le nombre exponentiel d'interactions entre processus dans le calcul parallèle. De plus, l'augmentation espérée de la puissance de calcul lors de l'application de parallélisme a un prix. Sauf dans les cas basiques, une augmentation de la performance nécessite une stratégie bien pensée pour paralléliser la résolution d'un problème. Il est également difficile de garantir *a priori* que la parallélisation passe bien à l'échelle et qu'elle est portable.

Le modèle BSP, et la bibliothèque BSPlib qui le met en œuvre, répond à certaines de ces préoccupations en fournissant une structure de calcul parallèle qui exclut plusieurs classes d'erreurs. Il permet également une performance fiable, portable et prévisible. Cependant, il faut toujours prendre soin d'éviter des erreurs dans le développement de programmes BSPlib, et une analyse manuelle des programmes est nécessaire pour profiter du modèle de performance BSP.

Les méthodes formelles apportent un cadre pour le développement de logiciels qui sont garantis mathématiquement sûrs et efficaces. Des méthodes automatiques, telles que l'analyse statique, sont particulièrement prometteuses car elles ne nécessitent pas l'intervention d'experts en méthodes formelles. C'est d'ailleurs le manque de méthodes formelles adaptées à BSPlib qui a motivé cette

thèse et nous a poussé à concevoir des outils automatisés pour aider au développement de programmes BSPlib qui soient à la fois corrects et efficaces.

Thèse

Notre thèse est que la majorité des programmes BSPlib se conforment à une structure appelée *alignement syntaxique*. Nous avons fait valoir que l'alignement syntaxique devait être imposé dans des programmes parallèles évolutifs et que les analyses statiques devaient exploiter cette propriété. Cette approche atténue avec élégance l'un des principaux problèmes d'analyse des programmes parallèles, à savoir le grand nombre d'interactions entre processus.

Contributions

Pour argumenter l'importance et l'utilité de l'alignement syntaxique dans les programmes BSPlib, nous avons premièrement conçu une analyse statique pour vérifier l'alignement syntaxique de la synchronisation. Nous avons montré comment cette propriété garantit une synchronisation correcte et nous avons formalisé, certifié en Coq, mis en œuvre dans Frama-C et évalué cette analyse. Deuxièmement, nous avons conçu, mis en œuvre comme prototype et évalué une analyse statique des coûts pour les programmes BSPlib qui exploitent l'alignement syntaxique. Troisièmement, nous avons conçu une condition suffisante, basée également sur l'alignement syntaxique. Nous avons prouvé que cette condition garantit un enregistrement sûr dans des programmes BSPlib. Enfin, ces développements reposent sur une série de formalisations progressivement plus complexes des fonctionnalités de BSPlib, de la synchronisation à la communication et l'enregistrement.

Perspectives

Nous concluons en discutant de pistes de recherche prometteuses. La précision de l'analyse statique de l'alignement syntaxique peut être améliorée. C'est en particulier l'hypothèse conservatrice sur la communication qui nécessite une révision. Une analyse fine des motifs de communication dans BSPlib (éventuellement basée sur des techniques polyédriques) pour étendre la reconnaissance des expressions dépendantes sur *pid* permettrait de réduire le nombre d'annotations actuellement requises.

Le prototype actuel de l'analyse des coûts devrait être étendu à un outil complet pour des programmes BSPlib réalistes, et évalué afin de valider son applicabilité. Son analyse du coût de la communication est précise, mais seulement pour les motifs de communication « data-oblivious ». Une recherche plus profonde est nécessaire pour concevoir des analyses de coût de communications dépendantes des données.

La condition suffisante pour l'enregistrement sûr devrait être la cible d'une analyse statique. Cette analyse doit être conçue de manière à ce qu'elle approche statiquement la condition suffisante et ensuit être mise en œuvre pour évaluer son applicabilité.

Nous prévoyons également d'autres cas d'utilisation pour l'alignement syntaxique dans l'analyse des programmes BSPlib, par exemple, pour détecter les écritures concurrentes, une erreur en BSPlib qui ressemble aux data races. Celles-ci se produisent lorsque deux processus utilisent DRMA pour écrire à la même zone de mémoire dans le même superstep. Le résultat final est spécifique à chaque implémentation de BSPlib et présente donc une source possible d'erreurs. Une analyse statique pour la détection de ces écritures pourrait se baser sur l'alignement syntaxique.

L'alignement syntaxique pourrait également être exploité en dehors de l'analyse statique. Des auteurs ont précédemment entamé des travaux en vue de la vérification déductive des programmes parallèles évolutifs, notamment par l'introduction des invariants sur tous les processus [175]. Ces invariants sont attachés en tant qu'assertions aux primitives de synchronisation. Nous croyons que ces assertions peuvent être attachées à tous les points de programme syntaxiquement alignés, et servir de base à un nouveau système de preuve compositionnelle pour des programmes parallèles comme BSPlib.

Dans cette thèse, nous nous concentrons sur BSPlib. BSPlib peut être considéré comme un modèle du sous-ensemble BSP dans d'autres bibliothèques parallèles comme MPI. Enfin, nous proposons des recherches sur l'exploitation de l'alignement syntaxique pour les méthodes formelles pour ces bibliothèques.

Table des matières

Table des matières xvii

Table des figures xx

Liste des tableaux xxiii

1 Introduction 1
- 1.1 Challenges of Scalable Parallel Programming . . . 3
- 1.2 The BSP model . . . 4
- 1.3 Formal Methods and Static Analysis . . . 5
- 1.4 Textual Alignment . . . 6
- 1.5 Contributions . . . 6
- 1.6 List of Publications . . . 7
- 1.7 Outline of Thesis . . . 8

2 Preliminaries 11
- 2.1 Notation . . . 12
- 2.2 The BSP Model . . . 12
  - 2.2.1 The BSP Computer . . . 14
  - 2.2.2 The BSP Execution Model . . . 15
  - 2.2.3 Example of a BSP Algorithm: *reduce* . . . 16
  - 2.2.4 The BSP Cost Model . . . 18
- 2.3 BSPlib . . . 23
  - 2.3.1 SPMD: Single Program, Multiple Data . . . 24
  - 2.3.2 Memory Model and Communication . . . 26
  - 2.3.3 BSPlib Program Structure . . . 26
  - 2.3.4 BSPlib by Example . . . 27
  - 2.3.5 The BSPlib API . . . 30
  - 2.3.6 BSPlib Implementations . . . 39
  - 2.3.7 BSPlib Limitations . . . 39
  - 2.3.8 Relationship to MPI . . . 42

2.4 The Data-Flow Approach to Static Analysis . . . 43
2.4.1 The Sequential Language **Seq** . . . 44
2.4.2 Control Flow Graph . . . 46
2.4.3 Data-Flow Analysis . . . 47
2.4.4 Abstract Domain . . . 49
2.4.5 Transfer Functions . . . 50
2.4.6 Calculating Solution Through Fixpoint Iteration . . . 51
2.5 Frama-C . . . 52

3 State of the Art 55
3.1 Parallel Models . . . 56
3.1.1 Other than BSP . . . 56
3.1.2 BSP Extensions . . . 57
3.2 Parallel Programming . . . 60
3.2.1 Other than BSP . . . 60
3.2.2 BSP . . . 63
3.3 Formal Methods for Scalable Parallel Programming . . . 64
3.3.1 Deductive Verification . . . 65
3.3.2 Model Checking . . . 68
3.3.3 Static Analysis . . . 70
3.3.4 Other Formal Methods . . . 79
3.4 Discussion . . . 79

4 Replicated Synchronization 81
4.1 Synchronization Errors in BSPlib Programs . . . 83
4.1.1 Textual Alignment and Replicated Synchronization . . . 84
4.2 The **BSPlite** Language . . . 85
4.2.1 Operational Semantics . . . 86
4.2.2 Denotational Semantics . . . 87
4.3 Static Approximation of Textual Alignment . . . 92
4.3.1 Pid-Independence Data-Flow Analysis . . . 93
4.3.2 Replicated Synchronization Analysis . . . 101
4.4 Implementation . . . 102
4.4.1 Adapting the Analysis to Frama-C . . . 103
4.4.2 Edge-by-Edge Flow Fact Updates . . . 103
4.4.3 Frama-C Control Flow Graph . . . 105
4.4.4 Implementing Interprocedural Analysis Using Small Assumption Sets . . . 111

4.5 Evaluation . . . 114
4.6 Related Work . . . 116
4.7 Concluding Remarks . . . 117

5 Automatic Cost Analysis **119**
5.1 **Seq** With Cost Annotations . . . 121
5.1.1 Syntax . . . 122
5.1.2 Semantics . . . 123
5.1.3 Sequential Cost . . . 125
5.1.4 Sequential Cost Analysis . . . 125
5.2 **BSPlite** With Cost Annotations and Communication . . . 126
5.2.1 Syntax . . . 127
5.2.2 Semantics . . . 127
5.2.3 Parallel Cost . . . 131
5.3 Cost Analysis . . . 134
5.3.1 Sequential Simulator . . . 135
5.3.2 Analyzing Communication Costs . . . 140
5.3.3 Analyzing Synchronization Costs . . . 145
5.3.4 Time Complexity of Analysis . . . 146
5.4 Implementation and Evaluation . . . 147
5.4.1 Benchmarks . . . 148
5.4.2 Symbolic Evaluation . . . 148
5.4.3 Concrete Evaluation . . . 148
5.4.4 Conclusion of Evaluation . . . 152
5.5 Related Work . . . 152
5.6 Concluding Remarks . . . 154

6 Safe Registration in BSPlib **157**
6.1 BSPlib Registration and its Pitfalls . . . 158
6.2 **BSPlite** with Registration . . . 161
6.2.1 Local Semantics . . . 162
6.2.2 Global Semantics . . . 167
6.3 Instrumented Semantics . . . 170
6.3.1 Instrumented Global Semantics . . . 176
6.4 Correct Registration . . . 179
6.4.1 Correctness . . . 179
6.5 Sufficient Condition for Correct Registration . . . 182
6.6 Related Work . . . 183

6.7 Concluding Remarks . . . 184

7 Conclusion and Future Work 185
7.1 Context . . . 185
7.2 Thesis . . . 186
7.3 Contributions . . . 186
7.4 Perspectives . . . 187

A Proofs for Replicated Synchronization 191
A.1 Operational Semantics Simulates Denotational . . . 191
A.1.1 Stable State Transformers . . . 192
A.1.2 Simulation . . . 194
A.2 Correctness of PI . . . 203
A.2.1 Domain . . . 204
A.2.2 Parameterized Constraint System . . . 204
A.2.3 Constraint System Facts . . . 205
A.2.4 Marked Path Abstractions and *pid*-independent Variables . . . 205
A.2.5 Correctness of the Analysis . . . 209
A.3 Correctness of *RS* . . . 216
A.3.1 Safe State Transformers . . . 216

B Proof Sketches for Safe Registration in BSPlib 221
B.1 Proof Sketch For Lemma 1 . . . 221
B.2 Proof Sketch For Theorem 4 . . . 222
B.3 Proof Sketch For Theorem 5 . . . 224

Bibliography 225

Table des figures

2.1 A BSP computer and an execution with $\mathbf{p} = 4$ . . . 14
2.2 The algorithm *reduce* . . . 17
2.3 BSP computer characterization . . . 19
2.4 The alternative algorithm *reduce'* . . . 22
2.5 Snapshot of a Single Program, Multiple Data execution with $\mathbf{p} = 3$ 24

2.6 BSPlib program structure . . . 26
2.7 Implementing *reduce* in BSPlib . . . 27
2.8 Schematic view of the DRMA operations in the BSPlib implementation of *reduce* . . . 29
2.9 Schema of the `bsp_put` remote memory write . . . 35
2.10 Schema of the `bsp_get` remote memory read . . . 37
2.11 A BSPlib program with a potential registration error . . . 40
2.12 Over-approximations of program behaviors. The set of program behaviors has been classified into safe and unsafe. The feasible behaviors of the program is represented by a blob, nested in an octagon representing their static over-approximation by a static analysis. In the case (a), program can be safe. The analysis cannot distinguish the cases (b) and (c), and thus cannot show that the program in (b) is actually safe. . . . 44
2.13 The **Seq** program s_{div} . . . 45
2.14 Semantics of expressions in **Seq** . . . 46
2.15 Operational big-step semantics of **Seq** programs . . . 47
2.16 The control flow graph of s_{div} . . . 48
2.17 Frama-C architecture . . . 52

3.1 An OpenMP example . . . 61
3.2 An example of a data race in a OpenMP program . . . 75
3.3 An example of a concurrent write in a BSPlib program . . . 75

4.1 Running examples for Replicated Synchronization Analysis . . . 84
4.2 Semantics of arithmetic expressions . . . 86
4.3 Semantics of boolean expressions . . . 86
4.4 **BSPlite** local operational semantics . . . 88
4.5 **BSPlite** global operational semantics . . . 89
4.6 Update, mask and combine operations . . . 90
4.7 Denotational semantics of textually aligned **BSPlite** programs . . . 91
4.8 Example program s_{nok} . . . 93
4.9 Control flow graph of s_{nok} . . . 93
4.10 Control flow graph and edge functions . . . 94
4.11 Example of path abstraction ordering . . . 95
4.12 Examples of the functions *exprs* and *free* . . . 97
4.13 The predicates ϕ^d and ϕ^c and the functions *cdep* and *vdep* . . . 97
4.14 Equation system $PI(s_{nok})$ and its solution . . . 100

4.15 Replicated synchronization analysis . . . 101
4.16 Simplified signature of a Frama-C forward data-flow analysis . . . 104
4.17 A simple interprocedural BSPlib program. A naive interprocedural analysis cannot verify the synchronization of this program. . . . 111

5.1 Big-step semantics of **Seq** extended with cost annotations . . . 124
5.2 The work-annotated program s_{fact} . . . 126
5.3 Local big-step semantics of **BSPlite** with cost annotations and communication . . . 128
5.4 Global big-step semantics of **BSPlite** with cost annotations and communication . . . 129
5.5 The program s_{scan} implementing parallel prefix calculation . . . 133
5.6 An execution of the program s_{scan} . . . 133
5.7 Parallel Cost Analysis pipeline . . . 135
5.8 Sequential simulator $S^{\mathtt{w}}(s_{\text{scan}})$ . . . 139
5.9 The program s_{scan}, recalled . . . 141
5.10 Polyhedral analysis of common communication patterns . . . 144
5.11 Sequential simulator $S^{1}(s_{\text{scan}})$, with annotations for communication bounds and synchronization costs . . . 146
5.12 `BcastLog` on Cluster, $\mathbf{p} = 8$ . . . 151
5.13 `BspFold` on Desktop, $\mathbf{p} = 8$ . . . 151
5.14 `Bcast1` on Cluster, $\mathbf{p} = 128$ . . . 151

6.1 Running examples for Safe Registration . . . 160
6.2 Syntax of **BSPlite** with registration . . . 162
6.3 **BSPlite** arithmetic, pointer and boolean expression semantics . . . 164
6.4 Configurations of the local semantics . . . 164
6.5 Local semantics of commands in **BSPlite** with registration . . . 166
6.6 Local multi-step semantics of **BSPlite** commands . . . 167
6.7 The function $\mathcal{R}$ formalizing updates of the registration sequence . . . 167
6.8 Communication in **BSPlite** programs . . . 168
6.9 Global big-step semantics of **BSPlite** programs and the reachability relation . . . 169
6.10 Illustration of paths by unfolding loops . . . 171
6.11 Operators and functions on paths and nesting stack . . . 173
6.12 Source of expressions . . . 173
6.13 Local, instrumented, semantics of **BSPlite** commands . . . 174
6.14 Local, instrumented, multi-step semantics of **BSPlite** commands . . . 175

6.15 Instrumented global big-step semantics of **BSPlite** programs and the reachability relation 176
6.16 Trace vectors from running examples with $\mathbf{p} = 2$ 177
6.17 Local correctness of an action trace 180
6.18 Global correctness of a trace vector 181

Liste des tableaux

2.1 The BSPlib API 31
2.2 An approximative and incomplete Rosetta Stone translating between MPI and BSPlib 43
2.3 Reaching Definitions in the program s_{div} 49

4.1 Evaluation results for Replicated Synchronization analysis 115

5.1 Statically obtained upper bounds of benchmarks for cost analysis . 149
5.2 Maximal error in predictions on benchmarks by cost analysis . . . 151

Introduction

1

Contents

1.1 Challenges of Scalable Parallel Programming 3
1.2 The BSP model 4
1.3 Formal Methods and Static Analysis 5
1.4 Textual Alignment 6
1.5 Contributions 6
1.6 List of Publications 7
1.7 Outline of Thesis 8

Computers are used to automate large calculations that would be prohibitively time consuming or intractable for humans. Research and development in computer science is progressively increasing their capacity to perform larger and larger computations without exhausting the patience of the user. This process enables computer-aided mathematical analysis of an ever-expanding set of complex natural phenomena. To cite one of many examples, researchers have used computers to obtain a finer understanding of the origin of the universe and the nature of matter [97].

In some cases, increased computational capacity enables replacing and or even surpassing special-purpose hardware. This is the case of Software-defined Radio, an essential component of 5G networks [107], which replaces custom telecommunications hardware and alleviates the spectrum scarcity problem [177].

Parallel computing[1] is an important method for obtaining large amounts of computational capacity. It consists of connecting multiple computers via an electronic network, and programming them to collaborate on solving a common

[1] In this thesis we distinguish scalable parallel computing from concurrent computing, wherein parallelism is employed in a fine-grained manner to execute unrelated computations in simultaneously.

task. Most modern computers exploit parallelism. This includes smartphones, like the Huawei P30, which has 8 cores[2]. But also, *supercomputers*, like Summit [14], which contains 2.4 million networked compute cores. Amongst other uses, Summit performs Earth System simulations that produce forecasts for the climate of the future.

As in any project, when you put in more resources, you expect higher efficiency. This is also true in the world of parallel computing. As you connect more computers, you expect an increase in computing power proportional to the added resources. However, this will not necessarily be the case. An analogy can be made with a social organization. Adding more human resources does not immediately translate to an organization capable of doing more work in the same time unit. This is due to the overhead involved in distributing and coordinating work. If done clumsily, adding more workers might even decrease the organization's power. The same is true in parallel computing where work must also be distributed and coordinated between the participating computers. Hence, one of the fundamental tasks of parallel computing is to devise parallel architectures and programs adapted for these architectures so that they *scale well*. In other words, so that adding resources gives the desired increase in computing power. This is the subject of *scalable parallel programming*.

This is a notoriously difficult task. In this thesis we will attack this difficulty in the context of BSPlib. This is a programming library for Bulk Synchronous Parallelism, a model of parallelism with salient features for structure, safety and performance. Our weapons of choice are automated verification tools called *static analyses*. Being mathematically specified and proven, these tools pertain to *formal methods*. The applications of this work are wide-reaching due to the tight relationship between BSPlib and the Bulk Synchronous Parallel subset of MPI [89, p. 55], a popular library for distributed-memory parallel programming [178].

In the remainder of this introduction we will illustrate the difficulties of scalable parallel programming. We introduce the Bulk Synchronous Parallel model, formal methods and static analysis. We then state our thesis before concluding

[2] `https://en.wikipedia.org/wiki/Huawei_P30`

with the list of our contributions, publications and the outline of the following chapters.

1.1 Challenges of Scalable Parallel Programming

There are three main difficulties in writing scalable parallel programs: devising the algorithm, implementing it correctly and gauging its scalability. The algorithm is the set of steps necessary to solve the problem at hand. Devising an algorithm that exploits parallel computing requires analyzing the problem and discovering if, and how, the work required to solve it can be distributed. This requires creativity, and is highly problem-specific. However, in this thesis, we focus on the remaining two difficulties.

Implementing the algorithm correctly is difficult since parallel programming is error-prone. In addition to the errors that are possible in normal, *sequential* computing, such as attempting to divide by zero or dereferencing a `NULL` pointer, parallelism enables a new set of errors. These are due to the interaction and coordination of processors.

We illustrate this with two common errors in parallel computing. A *deadlock* is a type of error that involves at least two processors, A and B. Both processors require some information from the other in order to proceed. But, what A needs to give B depends on what B needs to give to A, and vice versa. Progress is stalled, and the computation never terminates.

A *data race* occurs when two processes attempt to access (read or write) to the same resource, at least one of the accesses is a write, and the order with which the accesses occur are not fixed. In the case where one process reads and another writes the resource, the value read depends on the order of the access. In the case where both processes write the resource, the final value written to the resource depends on the order. Typically, neither situation is desired, and can lead to subtle miscalculations.

Both deadlocks and data races are caused by unforeseen *interleavings* of parallel executions. When multiple processors executes a stream of instructions in parallel, then the number of possible interleavings of these streams grows exponentially. The programmer must ensure that their program is correct under any feasible interleaving. This is similar to the situation in a game, where the player must predict each possible future move by the opponents and plan their response accordingly. This quickly becomes intractable further than a few moves, and grows exponentially more difficult with the number of opponents.

To complicate matters, the interleaving of each execution is *indeterministic*. This means that different executions of the same program with the same input may exhibit different interleavings: some of them erroneous and some of them not. The difficulty of reproducing erroneous interleavings translates to difficult debugging and repair.

Thirdly, having devised a parallel algorithm and implemented it correctly, there is still the issue of gauging its efficiency compared to a sequential solution. The approach of benchmarking, that is, executing and measuring the run time of the program, only gives indications for a specific parallel architecture and problem instance. To obtain more general results, predicting performance when adding more processors or when moving to a different parallel architecture, one needs to model both algorithm and architecture. This modeling is difficult, since it requires a judicious choice of what aspects are essential to performance and what aspects can be ignored to obtain a model simple enough to be analyzable.

Adding more processors to obtain higher computational capacity gives, at best, a linear increase in efficiency with each processor. But, in the light of these three difficulties, it comes at the price of an exponential increase in conceptual and implementation complexity.

1.2 The BSP model

Bulk Synchronous Parallel (BSP) [191] is a model for scalable parallel programming, with practical implementations, that help to alleviate the problems discussed above.

Notably, the parallel computation of a BSP program follows a structure that precludes both deadlocks and data races, by restrictions on synchronization and communication. Deadlocks are prevented since all processors synchronize at the same time, preventing the circular dependency of a deadlock. Data races are prevented since communications in a BSP computation (such as accessing a common resource) are executed in bulk. Notwithstanding these restrictions, BSP allows the expression of a large variety of parallel algorithms [186].

BSP also helps by providing portable performance predictions for parallel programs, by its simple but realistic cost model. The performance of parallel architectures are characterized by four parameters, and the run time of a parallel program as a cost formula: a function of these parameters that describes the program's resource consumption. The estimated run time of a program is obtained

by applying its cost function to the parameters of the architecture where it will be executed.

BSP helps to alleviate some of the difficulties of parallel programming. But it is no panacea to all its headaches. In this thesis we will direct our attention to BSPlib, a programming library for implementing BSP programs in the general purpose language C. In particular, we will address some common errors and issues that afflict BSPlib program.

As we will see, these issues result from grafting parallelism onto a general purpose language in the form of a programming library. The advantage of such programming libraries is that they ease the application of parallelism inside existing sequential programs and conversely, that they allow the reuse of sequential code in parallel programs. However, the generality of those languages permits the expression of executions that are not acceptable in the underlying parallel model and hence erroneous. This can be opposed to domain specific languages for parallelism where such executions can be restricted. We propose to use static analysis, a type of formal method, to bridge the gap between the two approaches.

1.3 Formal Methods and Static Analysis

Formal methods are techniques with rigorous, mathematical foundations for modeling, specifying, developing and verifying computer programs and hardware. *Static analysis* is a type of formal method. Static analyses are themselves computer programs that discover properties that hold for any execution in the programs they analyze. The word static refers to the fact that static analyses do not *execute* the program. This is opposed to dynamic approaches for discovering properties of the executions of a program, such as testing.

Our thesis is that static analysis can and should be used to verify the absence of errors in BSPlib programs. By detecting general purpose programs that transgress a parallel model exploited through a library, we can combine the benefits of dedicated parallel languages and library embeddings of parallelism.

Furthermore, we show that a majority of BSPlib programs are *structured* in a way that ensures the absence of synchronization errors, and that this structure can be discovered and exploited by static analyses to verify other safety and performance properties. Finally, our static analyses discover properties that hold in a program independently on the number of processes executing it. This ensures that the analyses themselves apply for future architectures.

1.4 Textual Alignment

Developing these analyses is a daunting task, as it does not suffice to verify the exponential number of interleavings for some fixed number of processors. Instead, all possible interleavings for any number of processors must be verified.

However, our intuition of realistic scalable parallel programs is that they tend to be structured. This structure limits the divergence of parallel control flow (that is, the differences between how different processes execute control structures of the program). Additionally, this structure reduces the number of interleavings whose interaction must be verified. We speculate that prudent parallel programmers have a wary eye towards programming patterns that increase the difficulty of mentally executing the program, and so naturally write programs that are structured[3].

Textual alignment is a way of structuring parallel programs around *collective actions*. These are actions that require the participation of all processes. Synchronization in BSP is an example of a collective action. The incorrect usage of collective actions is a common cause of errors in parallel programming. Textually aligned programs are written so that each collective action results from executing an instruction at the same program point in each process.

The textual alignment structure simplifies static reasoning on parallel programs for two reasons. First, it ensures that each process executes the same sequence of collective actions. It follows that the task of verifying correct usage of collectives is reduced to verifying that each feasible sequence of collective actions is correct when executed in replication by all processes. Second, it limits interleavings to the sequence of instructions that separates each pair of collective instructions.

1.5 Contributions

In previous work, textual alignment has been used to enforce correct synchronization [204] and to improve the precision of May-Happen-in-Parallel analysis [121]. Our thesis is that textual alignment also serves as a useful basis for static analysis of BSPlib programs. To argue our case, we statically infer textual alignment of BSPlib programs. We then use it to verify synchronization and obtain static cost predictions. Registration is an important component of BSPlib

[3] Echoing the idea that laziness is a virtue in programmers [41].

that enables communication. Our final contribution is a sufficient condition exploiting textual alignment that forms the basis of a future static analyses of safe usage of registration in BSPlib.

More specifically, our contributions are the following:

- A static analysis for verifying textual alignment and its application to verifying synchronization:
  - A formalization of BSPlib
  - A formalization of the analysis and its soundness proof, verified in the proof assistant Coq
  - An implementation for the C analysis framework Frama-C
  - An evaluation on a set of 20 BSPlib programs
- A static cost analysis:
  - A formalization of BSPlib with cost model
  - A prototype implementation of the analysis
  - An evaluation of the obtained cost formulas
- A sufficient condition for safe registration in BSPlib:
  - A formalization of BSPlib with registration
  - A sufficient condition based on textual alignment that ensures safe registration
  - A formal proof that this condition is sufficient for safe registration

1.6 List of Publications

The contributions detailed in this thesis have been the subject of the following publications:

- A. Jakobsson, F. Dabrowski, W. Bousdira, F. Loulergue, and G. Hains. Replicated Synchronization for Imperative BSP Programs. In *International Conference on Computational Science (ICCS), Procedia Computer Science*, 108:535–544, Jan. 2017., Zürich, Switzerland, 2017. Elsevier. doi: 10.1016/j.procs.2017.05.123.

- A. Jakobsson. Automatic Cost Analysis for Imperative BSP Programs. *International Journal of Parallel Programming*, 47(2):184–212, Apr. 2019. ISSN 0885-7458, 1573-7640. doi: 10.1007/s10766-018-0562-1.
- A. Jakobsson, F. Dabrowski, and W. Bousdira. Safe Usage of Registers in BSPlib. In *Proceedings of the 34th Annual ACM Symposium on Applied Computing*, SAC '19, Limassol, Cyprus, Apr. 2019. ACM. ISBN 978-1-4503-5933-7. doi: 10.1145/3297280.3297421.

1.7 Outline of Thesis

The rest of this thesis proceeds as follows:

- In Chapter 2, we give the preliminary notions necessary for reading the main contributions:
  - The notation used;
  - The BSP Model;
  - The BSPlib programming library;
  - Data-Flow Analysis;
  - The source-code analysis platform Frama-C;
  - and the formalization of a small sequential language **Seq** that will be used as a basis for the following formalizations.
- In Chapter 3, we review the state of the art in formal methods for scalable parallel programming in general, with a focus on static analysis in particular.
- In Chapter 4, we define a static analysis for verifying textual alignment and use it to verify synchronization of BSPlib programs. We also introduce **BSPlite**, our formalization of BSPlib, and prove the analysis sound with respect to this formalization.
- In Chapter 5, we extend **BSPlite** to include communication and then define its cost model. We then develop a static cost analysis, based on sequentialization of textually aligned programs and a communication volume analysis based on the polyhedral model. We implement and evaluate this analysis.

- In Chapter 6, we extend **BSPlite** further, adding pointers and primitives to model BSPlib registrations. We then define a sufficient condition that we prove ensures safe registration.
- Finally, in Chapter 7, we conclude this thesis, and give perspectives on future research in the context of analysis on scalable parallel programming exploiting textual alignment.

Preliminaries

2

Contents

2.1 Notation . . . 12
2.2 The BSP Model . . . 12
2.2.1 The BSP Computer . . . 14
2.2.2 The BSP Execution Model . . . 15
2.2.3 Example of a BSP Algorithm: *reduce* . . . 16
2.2.4 The BSP Cost Model . . . 18
2.3 BSPlib . . . 23
2.3.1 SPMD: Single Program, Multiple Data . . . 24
2.3.2 Memory Model and Communication . . . 26
2.3.3 BSPlib Program Structure . . . 26
2.3.4 BSPlib by Example . . . 27
2.3.5 The BSPlib API . . . 30
2.3.6 BSPlib Implementations . . . 39
2.3.7 BSPlib Limitations . . . 39
2.3.8 Relationship to MPI . . . 42
2.4 The Data-Flow Approach to Static Analysis . . . 43
2.4.1 The Sequential Language **Seq** . . . 44
2.4.2 Control Flow Graph . . . 46
2.4.3 Data-Flow Analysis . . . 47
2.4.4 Abstract Domain . . . 49
2.4.5 Transfer Functions . . . 50
2.4.6 Calculating Solution Through Fixpoint Iteration . . . 51
2.5 Frama-C . . . 52

In this section we give the preliminary notions necessary for reading the following chapters that describe the main contributions of this thesis.

We begin by giving the notations used throughout the thesis. This is followed by a presentation the Bulk Synchronous Parallel (BSP) model, its cost model and the programming library BSPlib. We then present static program analysis and in particular, data-flow analysis. We introduce the modern program analysis framework Frama-C, used to implement the synchronization analysis.

2.1 Notation

We write $A \hookrightarrow B$ (respectively $A \to B$) for the type of a partial (respectively total) function from A to B. The domain of a function $f : A \hookrightarrow B$ is given by $\mathbf{Dom}(f) = \{x \mid \forall x \in A,\ f(x) \neq \text{undef}\}$, where undef signifies lack of definition. The composition of two functions f and g is given by $g \circ f$.

We write A^* to denote the type of a sequence of elements from the set A, and for a sequence $as \in A^*$, we write $|as|$ to obtain its length. The symbol ϵ denotes the empty list of any type. The element $a \in A$ concatenated to the list $as \in A^*$ is written $a : as$, and the concatenation of two sequences as_1 and as_2 is given by $as_1 +\!\!+ as_2$. A literal sequence is written $[a_1, a_2, a_3, \ldots]$. To obtain the ith element of a sequence as, we write $as[i]$.

We write $A \times B$ for the Cartesian product of the sets A and B, and denote an element thereof (a, b). For such an element, $\pi^1(a, b) = a$ and $\pi^2(a, b) = b$.

We write $\mathcal{P}(A)$ for the power set of A. We write A^p to denote a vector of dimension p, and often refer to such vectors as p-vectors. A literal vector is written $\langle a_1, a_2, a_3, \ldots \rangle$. We also write $\langle a_i \rangle_{i \in p}$ to denote the p-vector where the ith element is a_i. When the size of the vector is given by the context, we abbreviate this to $\langle a_i \rangle_i$. To obtain the ith element of a vector V, we write $V[i]$.

We write $\#(A)$ for the cardinality of the set A. We will use the sets **Nat** to refer to the natural numbers, **Int** to the set of integers, and **Bool** for the set of booleans, whose literals we denote `tt` and `ff`.

2.2 The BSP Model

BSP is a *bridging model* for parallel computation: it abstracts away implementation details leaving only those needed to realistically reason on the properties of parallel computers, parallel programs and their executions.

Bridging models are important for the success of any mode of computation, but when Valiant proposed BSP in 1994, there was no realistic and widely used model for parallel computation. Parallel programs would be implemented and optimized for specific parallel architectures and their idiosyncrasies.

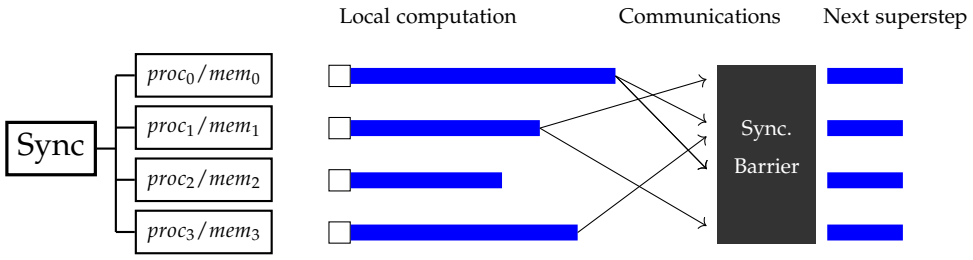
A bridging model is an ideal model for computation, with guarantees that programs written for the model can be executed on real computers and with a behavior as predicted by the model. The von Neumann architecture plays this role for sequential computation and has been important in the success of computing in general. This model gives a minimum set of components: a processing unit for arithmetic, instruction register and program counter, short-term and long-term memory and input-output devices. The bridging model can be seen as a contract between the programmer and the hardware designer: The programmer promises to program with these components in mind, and in return, their program can be executed on any computer that implements the model. The hardware designer promises to provide this set of components, and in return has access to all programs written with the model in mind.

BSP provides the same contract to programmers and hardware designers of parallel architectures. The parallel programmer designs their algorithm with the abstract BSP machine in mind, and the model guarantees that it will run on any realistic parallel machine. The hardware designer implements the necessary components of the BSP model, and in return, their machine can run any BSP algorithms.

In addition to portability, the hallmark of any bridging model, BSP was designed with three main design goals: **predictability**, **safety** and **structure**.

The model should be predictable, so that programmers can foresee the performance of their algorithms on any BSP computer, and conversely, the hardware designers foresee the performance of their architecture for any BSP algorithm. The model should be safe, so that bugs such as deadlocks and data races can be avoided. The model should be structured, as to simplify algorithm design and comprehension.

A clarification before we present the BSP model and explain how it fulfills these design goals: BSP was not designed to reason on concurrent programs. It is not the model appropriate to reason about the concurrency exhibited, for instance, by a modern web browser where many processes cooperate at disparate tasks such as to rendering web pages, handling input output, etc. Rather, BSP is deployed in data parallelism, where processes cooperate to solve a common



(a) A BSP computer (b) A BSP execution

Figure 2.1 – *A BSP computer and an execution with* $\mathbf{p} = 4$

goal by dividing the work. Typical examples are linear algebra computations or scientific simulations.

In the following sections we present the constituents of the BSP model: the BSP computer and how it executes BSP programs. We illustrate BSP using the classic "reduce" algorithm. We then give the BSP cost model, which is key to obtaining predictable performances, and use it to analyze the "reduce" algorithm.

For elementary introductions to BSP algorithms and their implementation we refer to [94] respectively [23], for a functional approach (based on the OCaml programming library BSMLlib) respectively imperative approach (based on the C programming library BSPlib).

2.2.1 The BSP Computer

The **BSP computer** is the BSP model's abstract view on the underlying parallel architecture. A BSP computer (Figure 2.1(a)) is composed of $\mathbf{p}$ homogeneous processor-memory units. Each unit has immediate access to its own memory, but communication is necessary to access the memory of other units. For this purpose, the BSP machine provides a communication network, connecting each pair of units with homogeneous bandwidth. The system is governed by a synchronization unit.

Any reasonable parallel architecture can be seen as a BSP computer. By setting $\mathbf{p} = 1$, a sequential computer is obtained. But typically, a modern computer has many cores, and so a higher $\mathbf{p}$ is taken. The network in this case is the socket interconnect. A cluster of compute nodes connected by an Ethernet network is another example. In all these examples, the synchronization unit is actually implemented in software using primitives such as locks.

2.2.2 The BSP Execution Model

The BSP machine executes parallel programs in a series of computational steps called of *supersteps*.

Each superstep (see Figure 2.1(b)) is composed of three phases corresponding to the three components of the BSP machine: (1) asynchronous local computation, (2) communication and (3) barrier synchronization.

Let us examine each phase. In the first phase, asynchronous local computation, each process executes (sequentially) and without interference from the other processes. The computation phase is followed by a communication phase. At this point, the processes may read or write into the memory of the other processes via the communication network. Finally, the processes synchronize. After synchronization, global computation either continues with another superstep, or terminates globally. This phase is often referred to as a *synchronization barrier*, since each process must reach it before computation continues.

Some immediate implications of this execution model are the following: (1) Logically, computation and communication do not overlap By removing such interactions between processes in the same superstep, parallel algorithms become simpler to understand. On the other hand, as long as this is opaque to the programmer, a BSP implementation can interleave computation and communication under the hood. However, even in the ideal case where computation and communication time of each process is the same, at most a factor of 2 speedup can be obtained this way. (2) The algorithm designer must make sure that any data a process requires in each superstep has been communicated to it during the previous. (3) All processes participate in synchronization. This differs from other models where processes can form groups that synchronize (called *subset synchronization*). Again, this restriction simplifies program comprehension and greatly facilitates the cost model presented below [93]. (4) Execution is deterministic, modulo concurrent writes and environmental factors.

This execution model is key to obtaining **safe**, **structured** and **predictable** parallelism. BSP programs are *safe* since a range of synchronization issues are ruled out. For instance, a classic deadlock where process A waits for B, and process B waits for A cannot occur in this model. Data races, in the form of concurrent writes, may occur, but they do so *deterministically*: the execution model ensures that if a concurrent write happens, then it will happen on each execution. If the model is implemented to resolve such write-write conflicts deterministically, then full determinism is obtained.

BSP programs are *structured*. The model forces the parallel algorithm designer

to think in terms of supersteps and phases. The resulting programs are easier to understand and analyze.

BSP programs are *predictable*. First, in terms of behavior, as ensured by the determinism, lack of data races and dead locks. Intuitively, the lack of interfering interleavings and communications reduces the set of possible outcomes of the BSP execution that the algorithm designer must consider. Second, in terms of performance. The structured execution model of BSP enables its cost model, which in turn permits the algorithm designer to predict the scalability of BSP algorithms.

2.2.3 Example of a BSP Algorithm: *reduce*

In this section we illustrate the BSP model by implementing a classic parallel algorithm: *reduce*.

Description of the Problem

The goal of *reduce* is to merge the elements of an array using an operator $\oplus$. We require that $\oplus$ is associative, and that it has a neutral element $\mathbf{0}$. This operation is also called *fold*, due to its similarity to the function available in many sequential, functional languages that can be summarized thus:

$$\mathit{fold}([X_1, \ldots, X_n], \oplus, \mathbf{0}) = X_1 \oplus X_2 \oplus \ldots \oplus X_N \oplus \mathbf{0}$$

By instantiating $\oplus$ with addition (whose neutral element $\mathbf{0} = 0$), we obtain the sum of all elements in X.

We wish to parallelize this operation in the algorithm *reduce*. We assume that the input array X's size is divisible by $\mathbf{p}$, and that X is *block distributed*. This means that the kth element of X is stored at process $\lfloor k/\mathbf{p} \rfloor$. In the source text, each process has a local view of X. They see only their block through the local variable `X`. An access to $\mathtt{X}[k]$ in this local view in process i corresponds to accessing $X[i * (n/\mathbf{p}) + k]$ in the global view. To simplify, we assume that the associative operator is $+$, but updating the algorithm to change operator is trivial.

The Algorithm *reduce*

We now give the algorithm of *reduce* in Figure 2.2. The idea is that the associativity of the operator allow us to first sum up each block into a partial sum

Algorithm *reduce*

Input The block distributed array X of whose length `n`, is divided by `p`, the number of processes. The integer `pid`, containing the process identifier of the executing process.

Variables The array `|Part|` of length **p** for storing exchanged local reduction. The variable `|local|` for storing local reduction.

Output `|S|` $= \sum_{i=0}^{n-1} X[i]$

```
1  (1) // compute local reductions
2      local = 0
3      for i in 0 ... n/p - 1:
4        local = local + X[i]
5
6      // exchange local reductions
7      for i in 0 ... p - 1:
8        put local in Part[pid] at process i
9
10     synchronize;
11
12 (2) // global reduction
13     S = 0
14     for i in 0 ... p - 1:
15       S = S + Part[i]
16
17     synchronize;
```

Figure 2.2 – *The algorithm reduce*

and then obtain the global sum by summing these. Each process executes this program text, and all variables are private to that process.

The algorithm consists of two supersteps numbered (1) and (2) terminated by a synchronization barriers. In the computation phase of the first superstep, each process reduces their block of `X` into `local` (Lines 2 to 4). Each process then schedules **p** remote write requests (called *puts*) so that the variable `Part`$[i]$ of each process contains the local reduction of each process i.

In the beginning of the second superstep, the transfer of all local reductions is completed and available in `Part` at each process, as the BSP model guarantees. The global reduction is now obtained by summing the local reductions, which

terminates the algorithm. At this point,

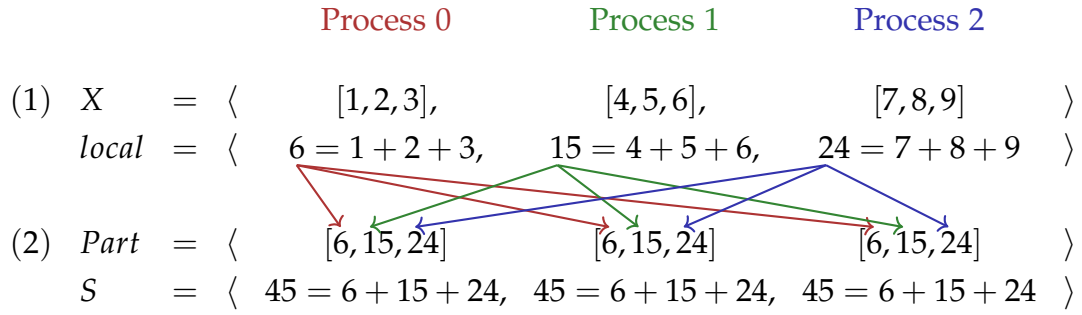
$$S = \sum_{i=0}^{\mathbf{p}-1} Part[i] = \sum_{i=0}^{\mathbf{p}-1} \sum_{k=0}^{\frac{n}{\mathbf{p}}-1} X[i * (n/\mathbf{p}) + k] = \sum_{i=0}^{n-1} X[i]$$

as desired by the specification.

Note that the identical result S will be available in all processes. This is natural since the *reduce* might be part of a larger computation where each process requires the result of the sum for the next step.

Executing *reduce* With 3 Processes

Consider an execution of *reduce* with 3 processes (i.e. $\mathbf{p} = 3$). Let $X = [1,2,3,4,5,6,7,8,9]$ and so $n = 9$. We will use $\langle y_0, y_1, y_2 \rangle$ to denote compactly the contents of the local variable y for each process, where, y_i is the value of y at process i. The contents of the memory of each process during the execution of the algorithm and the communication between supersteps are illustrated by the following schema:



As the algorithm starts, X is distributed by blocks in each process's local memory. After executing Lines 2 to 4, the variable `local` contains the partial reduction of each process. Each process then transfers their value of `local` to all other processes, with each remote write illustrated by one arrow. Finally, the contents of `Part` is summed up and stored in `S`.

2.2.4 The BSP Cost Model

The **cost model** of BSP ensures the predictability in performance. By the BSP model, not only can a BSP algorithm be executed on any BSP machine, but we can also predict at which *cost*, i.e. with what run time, it will execute.

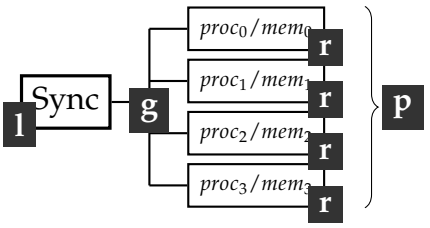


Figure 2.3 – *BSP computer characterization*

The cost model consists of two parts: (1) a characterization of BSP computers, which sums up the performance of a BSP machine with 4 parameters and (2) a method for attributing costs to executions.

Characterizing the BSP Machine

The cost model characterizes the performance of a BSP machines by four parameters (see Figure 2.3):

p the number of processes

r a measure on local computation cost

g a measure on communication cost

l a measure on synchronization cost

We have already seen **p**, so let us study the latter three more closely. The parameter **r** is the cost of taking one step of local computation. The notion of *computation step* depends on the context. For instance, in scientific computation, the steps of interest are floating-point operations (flops) and **r** is the time taken to perform one such operation. Sometimes **r** is removed and we express the other parameters in flops.

To understand **g** and **l**, we first define the concept of h-relations. An h-relation is a communication phase where each process sends or receives at most h words. Furthermore, there is at least one process which receives or sends h words. The BSP model assumes that it is not the overall communication volume of the system that determines the communication cost of a superstep, but rather, the *contention*. This corresponds to the maximum number of words received or sent by any process. Furthermore, BSP assumes separate reception and emission channels, thus h is the largest of the number of sent or received words at any processor and not their sum.

The underlying assumption of the communication cost model is that the time taken for the BSP computer to communicate and synchronize an h-relation is

described by

$$T_{\text{comm}}(h) = h\mathbf{g} + \mathbf{l}$$

for the appropriate choice of $\mathbf{g}$ and $\mathbf{l}$ specific to that computer. We then define $\mathbf{g}$ as $\lim_{h\to\infty} T_{\text{comm}}(h)/h$. Intuitively, $\mathbf{g}$ is the cost of sending one word under the assumption of asymptotic network communications and the parameter $\mathbf{l}$ is the overhead of starting up communication and synchronizing processes.

Except for $\mathbf{p}$, the BSP parameters of a parallel architecture are measured by benchmarking. As $\mathbf{r}$ is application dependent, so is its benchmark. For the communication parameters $\mathbf{g}$ and $\mathbf{l}$, the benchmark typically measures the actual communication time for a range of values of h, representative for the expected communication values of the application, and computes the $\mathbf{g}$ and $\mathbf{l}$ that minimize the difference between the communication times predicted by $T_{\text{comm}}(h)$ and the measured time.

Attributing a Cost to a BSP Execution

In short, the cost of a BSP execution is the sum of the cost of its supersteps and the cost of a superstep is the sum of the cost of its phases.

The cost of a computation phase is the length of the longest local computation. If $w_{i,k}$ is the number of local steps taken by processor i in superstep k, then the cost of local computation in this superstep is $w_k\mathbf{r}$ where $w_k = \max_{i=0}^{\mathbf{p}-1} w_{i,k}$, the longest local computation.

As discussed above, the cost of communication is determined by a measure of the communication pattern of the superstep, namely the h-relation. If $h^-_{i,k}$ respectively $h^+_{i,k}$ are the number of words received respectively sent by processor i in superstep k then the communication in superstep k is an h_k-relation with

$$h_k = \max_{i=0}^{\mathbf{p}-1}(\max(h^-_{i,k}, h^+_{i,k}))$$

and the cost of the communication phase is $h_k\mathbf{g}$. Finally, the cost of synchronization is $\mathbf{l}$.

By summing these, we obtain the cost of superstep k:

$$w_k\mathbf{r} + h_k\mathbf{g} + \mathbf{l}$$

If we assume an execution in S supersteps, then its total cost is the summation:

$$W\mathbf{r} + H\mathbf{g} + S\mathbf{l}$$

with

$$W = \sum_{k=0}^{S-1} w_k \quad H = \sum_{k=0}^{S-1} h_k$$

Execution Cost of *reduce* With 3 Processes

To illustrate the cost model, consider the execution of the *reduce* algorithm given in Section 2.2.3. We now calculate the cost of this execution, considering each addition as one step of local computation.

In the first superstep, each process performs 2 additions to obtain local sums, and so $w_1 = 2$. Then each process sends their local reductions to all 3 processors[1], and each processor receives 3 local reductions, so $h^-_{i,1} = h^-_{i,1}$ for all processes i, hence $h_1 = 3$, We add the synchronization cost and obtain $2\mathbf{r} + 3\mathbf{g} + \mathbf{l}$ for the first superstep.

In the second superstep, each process sums up the $\mathbf{p}$ local reductions in $\mathbf{p} - 1$ steps, hence $w_2 = 2$. There is no communication. Adding the synchronization cost, we obtain $2\mathbf{r} + \mathbf{l}$ for the second superstep. The full cost of this execution is then

$$(2 + 2)\mathbf{r} + 3\mathbf{g} + 2\mathbf{l}$$

This result can be generalized to any BSP machine with $\mathbf{p}$ processors and input array of size n by considering that the first superstep consists of $n/\mathbf{p} - 1$ additions and $\mathbf{p}$ sends and receives per process. The second superstep entails $\mathbf{p} - 1$ additions. We can thus express the cost of reduce for any $\mathbf{p}$ or n by the formula

$$(n/\mathbf{p} + \mathbf{p} - 2)\mathbf{r} + \mathbf{pg} + 2\mathbf{l}$$

Such a formula, parameterized by BSP parameters and the size of the problem, is referred to as the BSP algorithm's *cost formula*.

As we noticed earlier, *reduce* has all processes calculate the global sum in the last superstep. In addition to simplifying the algorithm, we can now argue formally that it comes at no additional performance penalty: whether all or only one processes calculate the global sum, the last superstep still has a local computation cost of $\mathbf{p} - 1$.

We could even consider an implementation where process 0 receives all local reductions, computes the global reduction and sends it to the others (see Figure 2.4). Even though the total number of transmitted words is smaller, the communication cost of the first superstep is still $\mathbf{p}$, since process 0 receives $\mathbf{p}$ reductions, and so the cost of this superstep is still $(n/\mathbf{p} - 1)\mathbf{r} + \mathbf{pg} + \mathbf{l}$. In the

[1]For simplicity, we count each process's put to itself.

Algorithm *reduce'*

```
1  (1) // compute local reductions
2      local = 0
3      for i in 0 ... n/p - 1:
4        local = local + X[i]
5
6      // exchange local reductions
7      put local in Part[pid] at process 0
8
9      synchronize;
10
11 (2) // global reduction
12     if pid = 0:
13       S = 0
14       for i in 0 ... n:
15         S = S + Part[i]
16
17       // broadcast global reduction
18       for i in 0 ... p - 1:
19         put S in S at process i
20
21     synchronize;
```

Figure 2.4 – *The alternative algorithm reduce'*

second superstep in this implementation, only process 0 performs the $\mathbf{p}$ additions to obtain the global sum, but it must also perform $\mathbf{p}$ puts to transmit it to the other processes. In total, the cost of the second superstep is $(\mathbf{p}-1)\mathbf{r}+\mathbf{pg}+\mathbf{l}$. The full cost of *reduce'* is:

$$(n/\mathbf{p}+\mathbf{p}-2)\mathbf{r}+2\mathbf{pg}+2\mathbf{l}$$

In conclusion, the alternative implementation *reduce'* is more complicated and has a higher communication cost than *reduce'*. This example demonstrate how to the BSP cost model can be used to guide algorithm design.

Implications of the Cost Model

After analyzing a BSP algorithm and obtaining its cost formula, we can predict its performance on any BSP machine by measuring its BSP parameters and plugging them into the cost formula.

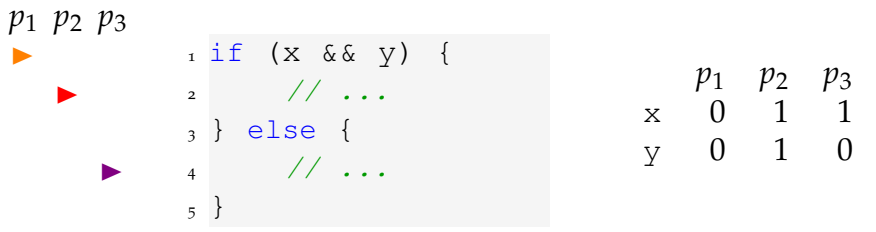
Conversely, for a given BSP machine, we can predict how different algorithms will perform on it. A problem may have different algorithms that are more or less efficient depending for BSP computers with different parameters. This opens up the possibility of *performance portable* programs that dynamically choose which implementation to use as a function of execution architecture. The idea of *immortal* algorithms are based on this idea.

An immortal algorithm has provably optimal BSP cost *regardless of the BSP machine that executes it*. Just as it is known that no sorting algorithm can perform better than $O(n*\log n)$, as long as the BSP model remains a realistic model of parallel architectures, immortal algorithms are guaranteed to stay optimal regardless of future architectural developments.

With the development of the BSP model, several *programming models* for BSP appeared. A push for standardization led to the elaboration of BSPlib [100], a programming library and API-interface to answer the question "How to implement BSP algorithms?". In the next section we present BSPlib and demonstrate it by implementing *reduce*.

2.3 BSPlib

BSPlib [100] is a library and standard API for imperative BSP programming. BSPlib resulted from the standardization effort and combination of two preced-



(a) Program counter (b) Program (c) Memory contents

Figure 2.5 – *Snapshot of a Single Program, Multiple Data execution with* $\mathbf{p} = 3$

ing libraries: Oxford BSP Library [141], the first portable BSP library, and Green BSP [88], which introduced Bulk Synchronous Message Passing.

The design goals of BSPlib can be resumed by minimalism and portability. It provides a small but highly composable set of 20 primitives callable from C and Fortran. Consequently, the implementation effort of porting BSPlib to new platforms is reduced. In terms of size, BSPlib can be compared to the Message Passing Interface (MPI) [139], a widely used communication library, that contains well over 200 different primitives. In terms for functionality, BSPlib can be seen as a nimble model of the Bulk Synchronous Parallel subset of MPI. Modern interconnects implement Remote Direct Memory Access operations directly, and the parallel programmer accesses them through this subset. Hence, using this part of MPI, and doing so correctly, is becoming essential for performance in many applications [103, 132, 83]. We discuss this relationship more closely in Section 2.3.8.

2.3.1 SPMD: Single Program, Multiple Data

BSPlib programs are written in *Single Program, Multiple Data* [59] (SPMD) style. In this mode of programming, each process executes the same program, but execute in separate memories and with their own program counter. Informally, this can be understood as the parallel composition of the same program c, parameterized by a unique process identifier:

$$c(0) \parallel \ldots \parallel c(\mathbf{p} - 1)$$

Figure 2.5 contains a SPMD [59] program under execution by 3 processes. Each triangle represents the program counter of one process, and the memory contents of each process are given by the table on the right. The program text is the same, but the memory contents of each process admits a different evaluation

of the `if`-construct, allowing different processes to execute different parts of the program.

The SPMD model enables parallelization of an initially sequential programs, by decorating it with appropriate parallel primitives to distribute data and computation and to insert communication and synchronization where necessary.

While the overlap is considerable, SPMD can be thought of as an alternative to *fork-join* or *master-slave* parallelism. In the fork-join parallelism, process creation and destruction is explicit and controlled by primitives in the program. For instance, new processes can be forked off to recursively solve sub-problems in divide-and-conquer algorithms, or to treat parallel iterations of loops. In master-slave parallelism, a master process controls the creation of slave processes and distributes work between them.

SPMD can also be positioned with respect to Flynn's taxonomy [71] that classifies computer architectures based on their number of instruction streams (Single or Multiple) and their number of data streams (Single or Multiple), giving rise to the following classifications:

Single instruction stream, Single data stream (SISD)
: A sequential computer with no parallel execution.

Single instruction stream, Multiple data streams (SIMD)
: A computer that executes a single stream of instructions in lockstep over multiple data streams.

Multiple instruction streams, Single data stream (MISD)
: Multiple processes treat the same data. This architecture can be used to implement is pipeline parallelism, where each process executes on stage of a series of transformations on the input data. Or, to implement fault tolerance, where disparate results between processes is indicative of error [176].

Multiple instruction streams, Multiple data stream (MIMD)
: An architecture with multiple processes executing different instruction streams on different data streams. This corresponds to modern multi-core processors or multi-node clusters where each node has its own memory.

In this context, SPMD can be viewed as a form of MIMD, but where the different instruction streams arise from (potentially) different source code locations the same source program.

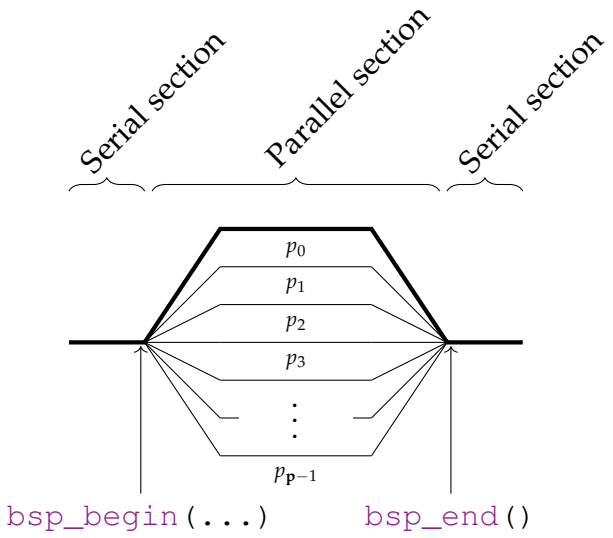


Figure 2.6 – *A BSPlib program is a sequence of an optional serial section, a parallel section and a final serial section.*

2.3.2 Memory Model and Communication

The BSP model prescribes that processes execute in distributed memory. Depending on the BSPlib implementation, this might or not be the case. However, BSPlib programs should be programmed as if each process has a private, local memory that is not accessible from the other processes by default.

Instead, process communication is enabled either by *Bulk-Synchronous Message Passing* (BSMP) or *Direct Remote Memory Access* (DRMA), both of which are explained in more detail in the following sections. Both communication types are guaranteed to be executed before the start of the next superstep. The BSPlib runtime handles a queue for each process that is used for BSMP. This allows processes to receive message without the need to allocate space before receiving messages, and is suitable for applications with sparse, dynamic communication patterns. With DRMA, processes can perform safe and buffered (or high-performance but unbuffered) reads and writes into the memory of other processes, after a preliminary registration.

2.3.3 BSPlib Program Structure

A BSPlib program is an optional serial section, followed by a parallel section and a final serial section (See Figure 2.6). The serial section is executed by one process, which becomes process 0 (i.e. the process with process identifier 0) in the parallel section.

```c
#include <bsp.h>
#include <stdio.h>
#include <stdlib.h>

const int N = 1000;

double reduce(double *X, int m, int pid, int p);

int main(void) {
    double *X;
    int i, m;
    int pid, p;
    double res;

    // start parallel section
    bsp_begin(bsp_nprocs());

    // obtain total number of processes and process id
    p = bsp_nprocs();
    pid = bsp_pid();

    // initialize block distributed X with some example
    // data
    m = N/p;
    X = malloc(m * sizeof(double));
    for (i = 0; i < m; i++) {
        X[i] = pid*m + i;
    }

    // reduce X and store results in res
    res = reduce(X, m, pid, p);

    // display the results
    if (pid == 0) {
        printf("The sum is: %f\n", res);
    }

    // clean up and end parallel section
    free(X);
    bsp_end();
}

double reduce(double *X, int m, int pid, int p) {
    double sum, local;
    double *Part;
    int i;
    size_t sz = sizeof(double);

// superstep (0): setup storage for partial results
    // create array for storing partial results
    Part = malloc(p * sz);
    // register the arrays Part of each process
    bsp_push_reg(Part, p * sz);
    bsp_sync();

// superstep (1): compute & broadcast partial results
    // compute partial results
    local = X[0];
    for (i = 1; i < m; i++) {
        local = local + X[i];
    }

    // broadcast the partial result of this process s
    // into the sth cell of Part in each process
    for (i = 0; i < p; i++) {
        bsp_put(i, &local, Part, pid * sz, sz);
    }
    bsp_sync();

// superstep (2): sum partial in to global results
    sum = Part[0];
    for (i = 1; i < bsp_nprocs(); i++) {
        sum = sum + Part[i];
    }

    // destroy registration of Part
    free(Part);
    bsp_pop_reg(Part);
    bsp_sync();

    return sum;
}
```

Figure 2.7 – *Implementing reduce in BSPlib*

The parallel section is a function that is bracketed by calls to `bsp_begin` and `bsp_end`. That is, the first respectively last statement of the function is a call to `bsp_begin` respectively `bsp_end`. This section is executed in parallel by **p** processes, where **p** is determined by the number of requested and available processes. Only process 0 has access to memory that was allocated in the optional preceding serial section, including all global variables. Furthermore, except for printing to standard output or standard error, only process 0 can perform input/output.

2.3.4 BSPlib by Example

An Example Program: Reduce in BSPlib

To demonstrate how to program BSP algorithms with BSPlib and to give a gentle introduction to the most commonly used primitives, we implement the *reduce* algorithm from Figure 2.2 using BSPlib in C. This example is then followed by a description of each of the 20 BSPlib primitives.

The source code of *reduce* in BSPlib is given in Figure 2.7. The program consists of a `main` function that sets up BSPlib and the necessary data structures, and the `reduce` function that performs the actual reduction. The program is

a direct translation of the pseudocode of Figure 2.2 into C, with the additional bookkeeping needed by BSPlib to set up parallel computation and communication. All BSPlib primitives are prefixed `bsp_` and, in this document, colored (e.g. `bsp_pid`).

Starting the Parallel Section: `bsp_begin` and `bsp_nprocs`

The first statement in `main` is a call to `bsp_begin`. The call `bsp_begin(P)` denotes the start of the parallel section of the BSPlib program, and the argument `P` the number of requested processes. The function `bsp_nprocs` has two uses: before calling `bsp_begin` it returns the number of available processors. After calling `bsp_begin`, it returns the number of processes **p** that participate in the parallel section. Line 16 thus starts a parallel section with the maximum number of available processes. From this point on, execution splits into **p** processes, each with its own memory.

Queries: `bsp_nprocs` and `bsp_pid`

On Line 19, `bsp_nprocs` is called again, this time to obtain the number of processes allocated to the computation. We then call `bsp_pid`, which returns its process identifier (pid). On Lines 24 to 28 we allocate and initialize the globally distributed array `X`: from the local point of view, `X` is a handle to each process's block of the global distributed variable `X`. We initialize `X` with some example data.

On Line 31, each process calls the `reduce` function with a pointer to its block of `X`, the local block length, its pid and the total number of processes as arguments.

Setting up Communication: Registering with `bsp_push_reg`

The function `reduce` implements the *reduce* algorithm of Figure 2.2.

Before the actual algorithmics start, some setup is required. We implement the communication of partial sums as DRMA, which enables processes to read and write specific parts of each others memory. To do so BSPlib provides a mechanism for addressing remote memory called "registrations". On Line 50, each process allocates memory that will contain the partial sums of all processes, pointed to by `Part`. We then call `bsp_push_reg(Part, p * sz)`. This creates an association, called a registration, between each process's memory area for partial sums, the extending `p * sz` bytes.

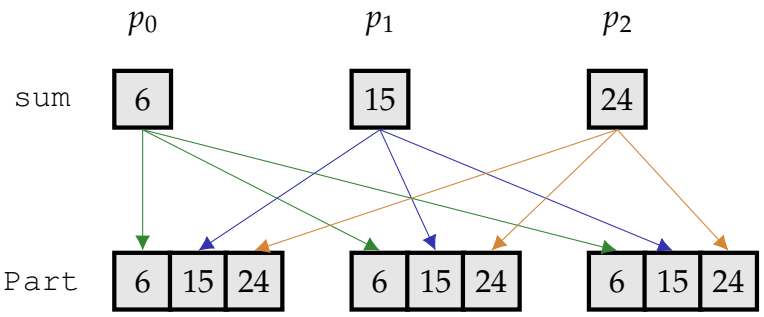


Figure 2.8 – *Schematic view of the DRMA operations in the second superstep of the BSPlib implementation of reduce with* $\mathbf{p} = 3$ *and* $\mathtt{x} = [1,2,3,4,5,6,7,8,9]$

One Line 53, we terminate the first superstep by calling `bsp_sync`: synchronization is needed for registration to take effect in BSPlib. Consequently this implementation has one preliminary superstep before calculating partial sums, contrary to the algorithm in Figure 2.7. Calling `bsp_sync` stops local computation and does not return control to local processes until all the registration requests (of which there is one for `Part`) and communication requests (of which there is none) have been handled by the BSPlib runtime.

Point-to-Point Communication: `bsp_put`

We now engage in the actual algorithm. On Lines 57 to 60 we calculate the partial sum in the variable `local`. Since `Part` was registered in the previous superstep, we can now let each process `pid` communicate its partial sum `local` into `Part[s]` on all processes. The call to `bsp_put` on Line 65 schedules this communication. It is read: "request the transfer of `sz` bytes starting from `Part` into the memory that process `i` has associated with `Part`, offset by `pid * sz`".

By default, communication in BSPlib is *buffered* until the next superstep. When `bsp_put` is called, BSPlib stores the data to be transferred in an internal buffer and queues it to be transferred at the next synchronization barrier, which occurs when `bsp_sync` is called on Line 67. If we assume the same input data as in the example execution in Section 2.2.3, then the resulting communication phase is as illustrated in Figure 2.8.

Each process schedules a write to itself, which might seem like an unnecessary use of the network. However, BSPlib implements such inter-process communications as memory copies. Thus, there is no need to make a special case for `i == pid`, keeping the code lighter.

Ending the Parallel Section: `bsp_pop_reg` and `bsp_end`

Each process will have the received the partial sums of all other processes in `Part` after synchronization barrier. There are summed up in `sum` on Lines 70 to 73, concluding *reduce*'s implementation.

Before returning `sum` to `main`, we deallocate and destroy the registration of `Part` using the `bsp_pop_reg` primitive on Lines 76 to 78. Like for their creation, the destruction of registrations takes effect at the next synchronization. To ensure the destruction of the `Part`'s registration we synchronize one last time in `reduce` on Line 78.

Back in the `main` function, process 0 prints the results of the reduction. Then `bsp_end` is called to terminate the parallel section.

2.3.5 The BSPlib API

The full API of BSPlib is summarized in Table 2.1. The rest of this section contains an description of each primitive. The full description of BSPlib can be had in [100] and in the BSPlib manual, available online[2]. For a full guide to programming in BSPlib, the reader is referred to the textbook [23].

Initialization

```
void bsp_init(void (*startproc)(void), int argc, char **argv)
```

There are two ways of initializing BSPlib: either by having the main function start the parallel section, or, when some serial processing is needed before the parallel section starts, having a dedicated SPMD function.

In the latter case, the first statement in the `main` function of the program must be a call to `bsp_init`, with a function pointer to the dedicated SPMD function and the program arguments. In this mode, serial execution starts in the `main` function and a call to the SPMD function initiates the parallel section.

```
void bsp_begin(int maxprocs)
```

A call to `bsp_begin` is used to initiate the parallel section in the SPMD function of a BSPlib program, and must be the first statement of that function.

The argument `maxprocs` denotes the desired number of processes. BSPlib might spawn fewer than the desired number (for instance, if the requested number of processes are fewer than the number of available processors).

[2] `http://www.bsp-worldwide.org/implmnts/oxtool/man/`

Category	Function		Description
Initialization	`void`	`bsp_init(void(*startproc)(void), int argc, char **argv)`	Initialize the BSPlib system.
	`void`	`bsp_begin(int maxprocs)`	Spawn a number of BSP processes.
	`void`	`bsp_end()`	Terminate BSP processes.
Halt	`void`	`bsp_abort(char *format, ...)`	Stop a BSP computation.
Inquiry	`int`	`bsp_pid()`	Determine the process identifier of a BSP process.
	`int`	`bsp_nprocs()`	Determine the total number of BSP processes.
	`double`	`bsp_time()`	High-precision real-time clock.
Synchronization	`void`	`bsp_sync()`	End a superstep.
DRMA	`void`	`bsp_push_reg(const void *ident, int size)`	Register a data-structure as available for direct remote memory access.
	`void`	`bsp_pop_reg(const void *ident)`	Remove the visibility of a previously registered data-structure.
	`void`	`bsp_put(int pid, const void *src, void *dst, int offset, int nbytes)`	Write data into a remote process's memory.
	`void`	`bsp_get(int pid, const void *src, int offset, void *dst, int nbytes)`	Read data from a remote process's memory.
BSMP	`void`	`bsp_set_tagsize(int *tag_bytes)`	Set tag size of a BSMP message.
	`void`	`bsp_send(int pid,const void *tag, const void *payload,int payload_bytes)`	Transmit a BSMP message to a remote process.
	`void`	`bsp_qsize(int *packets, int *accum_nbytes)`	Check how many BSMP messages arrived.
	`void`	`bsp_get_tag(int *status, void *tag)`	Retrieve the tag on a BSMP message.
	`void`	`bsp_move(void *payload, int reception_bytes)`	Move a BSMP message from the queue.
High Performance	`void`	`bsp_hpput(int pid, const void *src, void *dst, int offset, int nbytes)`	Unbuffered write data into a remote process's memory.
	`void`	`bsp_hpget(int pid,const void *src, int offset, void *dst, int nbytes)`	Unbuffered read data from a remote process's memory.
	`int`	`bsp_hpmove(void **tag_ptr_buf, void **payload_ptr_buf)`	A lean method for moving a BSMP message from the queue.

Table 2.1 – *The BSPlib API. Primitives are hyper-linked to their corresponding online BSPlib manual page.*

No BSPlib primitives, except bsp_nprocs and bsp_init, can be called before bsp_begin has been called.

```
void bsp_end()
```

A call to bsp_end must terminate the SPMD function of a BSPlib program. It ends the parallel section and the last superstep. Note that outstanding communications and registration requests are not delivered by bsp_end. All processes except process 0 are terminated.

No BSPlib primitives, except bsp_nprocs, can be called after bsp_end has been called (see Section 2.3.3).

Halt

```
void bsp_abort(char *format, ...)
```

This function is used to terminate the BSP computation and signal an erroneous situation. Any process can call bsp_abort, and the BSPlib runtime handles the termination of the other processes with no need for the user to synchronize.

The first argument to bsp_abort specifies a C-style format message, remaining arguments are interpreted as for printf. The formatted message is printed before the termination of the program.

Inquiry

```
int bsp_pid()
```

The bsp_pid function returns the unique process identifier of the calling process, which is between 0 and $\mathbf{p} - 1$, where $\mathbf{p}$ is the number of processes in the parallel section.

```
int bsp_nprocs()
```

Outside the parallel section of a BSPlib program, before bsp_begin has been called, bsp_nprocs returns the number of available processors. Inside the parallel section, bsp_nprocs returns the number of processes participating in the BSP computation

```
double bsp_time()
```

Returns the contents of a implementation-specific, high-precision clock. The value of bsp_time is local to each process.

Synchronization

`void bsp_sync()`

The function `bsp_sync` is called collectively to end a superstep. Its call marks the end of the local computation phase. The calling process is frozen until all other processes have called `bsp_sync`. At that point, all registration and communication requests are executed, control is returned to each process and the next superstep starts. Calling `bsp_sync` also ensures the delivery of all outstanding *high-performance communication requests* (see functions `bsp_hpput` and `bsp_hpget` below).

The BSPlib program's behavior is undefined if some process does not participate in the call to `bsp_sync`. This can happen if it terminates the BSP computation by reaching `bsp_end`, if it diverges, or if it terminates due to a local error (such as division by zero). In this case, depending on the implementation, the BSP program will typically hang or terminate with a dynamic error.

Direct Remote Memory Access (DRMA)

BSPlib enables two communication types: Direct Remote Memory Access (DRMA)[3] and Bulk Synchronous Message Passing (BSMP). The former is more commonly used, whereas the latter is useful for parallel computations with sparse data communication patterns. This section describes the DRMA primitives and the next deals with those for BSMP.

DRMA terminology Throughout this thesis, we will use the terms source, destination, origin and source in accordance to how it is used by the MPI standard [139, p. 404].

The *origin* is the process that is calling the DRMA primitive, and the *target* is the process whose memory is being remotely accessed. The *source* is the process where the transferred data is located, and the *destination* is the process who will receive the transferred data. Hence, in put-requests, the origin and the source is the same process, and the target and destination is also the same process. In get-requests, the origin is the destination and the target is the source.

The Registration Sequence The BSPlib **registration sequence** is an internal data structure used by BSPlib to create associations between **p** memory areas:

[3]The equivalent terms "Remoted Direct Memory Access" (RDMA), and "Remote Memory Access" (RMA) are less commonly employed in the BSPlib community.

one per process in the parallel computation. These associations are called a **registrations** and can be understood as **p**-vectors of memory areas. A process uses its local address in the registration as a handle to refer to the memory areas the other processes registered for DRMA operations. Multiple registrations can be created: the registration sequence is the list of these registrations.

The same address can be registered multiple times, but only the last registration of an address in the registration sequence is **active** and can be used for referring to remote addresses. Hence the last registration *shadows* previous registrations of the same address. The motivation is modularity: to allow addresses to be reused for communication in different parts of the code, possibly unbeknownst to each other.

A collective call to `bsp_push_reg` requests the creation of a registration (a **push**-request, or push for short) for the next superstep, and a collective call to `bsp_pop_reg` request the destruction (a **pop**-request, or pop for short), for the next superstep. These functions are detailed below.

Registration requests must be **compatible**: the order of all pushes must be the same on all processes, and for the pops likewise. However, it does not matter how requests are interleaved within one superstep.

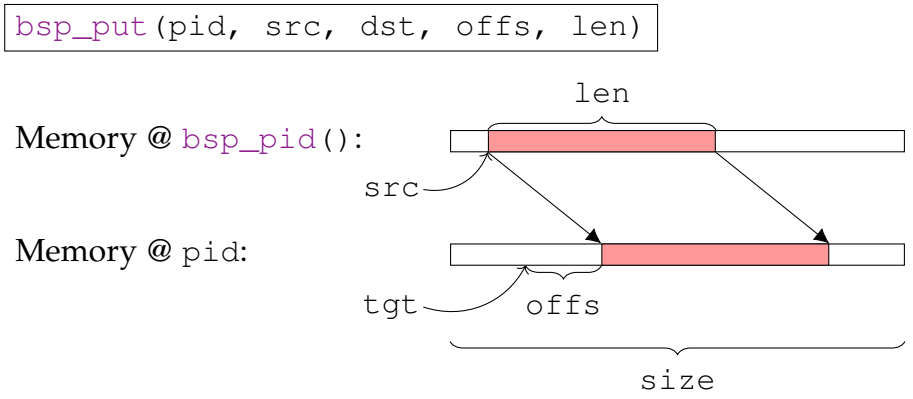
```
void bsp_push_reg(const void *ident, int size)
```

The function `bsp_push_reg` requests the creation of a registration in the next superstep ("pushing" the registration).

Calls to `bsp_push_reg` must be made collectively. When called collectively with the arguments $\langle (l_0, s_0), \ldots, (l_{\mathbf{p}-1}, s_{\mathbf{p}-1}) \rangle$, (i.e., each process i calls `bsp_push_reg(`l_i`,` s_i`)`) then the registration $\langle (l_i, s_i) \rangle_i$ will be added to the registration sequence at the next synchronization barrier. The intuitive effect is that the address l_i at process i is associated with the address l_j in process j from the next superstep, and that the memory area in process i starting at l_i and extending s_i bytes is exposed for DRMA operations.

If some of the processes should not to expose any memory in the registration, then they can provide `NULL` for `ident`. However, they will be unable to access the memory exposed by the other processes.

As detailed above (Section 2.3.5), the sequence of pushes and pops in the same superstep must be compatible.



where $\langle \ldots, (\texttt{dst}, \_), \ldots, (\texttt{tgt}, \texttt{size}), \ldots \rangle$ is the active registration for `dst`

Figure 2.9 – *Schema of the `bsp_put` remote memory write*

`void bsp_pop_reg(const void *ident)`

The function `bsp_pop_reg` requests the creation of a registration in the next superstep ("popping" the registration).

Calls to `bsp_pop_reg` must be made collectively. When called collectively with the arguments $\langle l_0, \ldots, l_{p-1} \rangle$ (i.e., each process i calls `bsp_pop_reg(`l_i`)`) then the registration $\langle (l_i, s_i) \rangle_i$ is removed from the registration sequence at the next synchronization barrier.

The popped registration $\langle (l_i, s_i) \rangle_i$ must have been pushed in a previous superstep. Furthermore, each l_i must be in the same registration, and may not have been pushed again in a more recent registration. If this is not the case, a dynamic error may occur, as defined by the implementation.

Formally, if rs is the original registration sequence and

$$rs = rs_1 +\!\!+ \langle (l_i, s_i) \rangle_i +\!\!+ rs_2$$

where $rs_2^T[i]$ does not contain l_i for each i, then rs' is the registration sequence after the pop is applied during synchronization

$$rs' = rs_1 +\!\!+ rs_2$$

where $+\!\!+$ is list concatenation.

The intuitive effect is that the address l_i at process i is no longer associated with the address l_j in process j in the next superstep. If some process i has pushed l_i in some older registration, then this registration becomes active for l_i.

As detailed above (Section 2.3.5), the sequence of pushes and pops in the same superstep must be compatible.

```
void bsp_put(int pid, const void *src, void *dst,
             int offset, int nbytes)
```

The function `bsp_put` requests the transfer of the memory area starting at `src` in the origin and extending `nbytes` bytes, into the memory area at the target process `pid` that is in an active registration with `dst` at offset `offset` (See schema in Figure 2.9).

If the write goes outside the memory area that the target registered with `dst`, that is if `offset + nbytes > si`, where `si` is the extent that the target specified for the registration, then this call is illegal. If the calling process have registered `NULL` for `dst`, then this call is illegal. Putting to oneself is implemented as memory copy.

The function `bsp_put` is *buffered on source, buffered on destination*. This means that the origin process can safely access (read or write) the memory containing the data to be sent immediately after the call to `bsp_put`, since the data to be transferred is copied by the BSPlib runtime into a buffer and transferred at the next call to `bsp_sync`. Similarly, the target process can safely access the memory that is the target of the put with no risk of disturbing the communication.

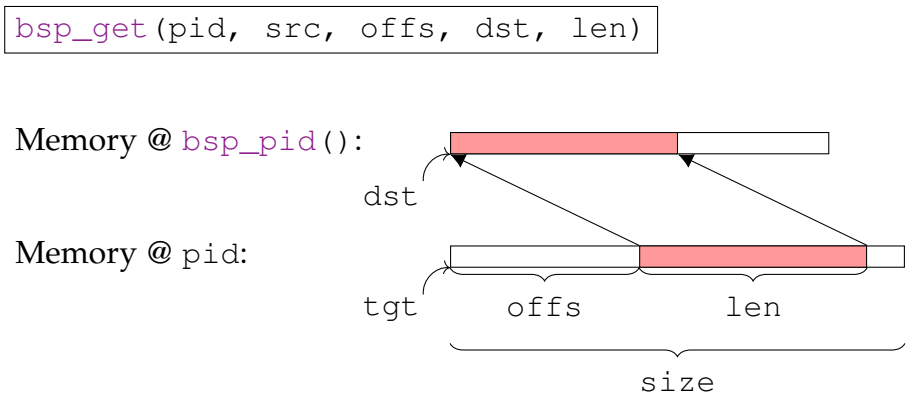
The high-performance variant of this function, `bsp_hpput`, is described below.

```
void bsp_hpput(int pid, const void *src, void *dst,
               int offset, int nbytes)
```

The function `bsp_hpput` differs with `bsp_put` with respect to buffering. It is *unbuffered on source, unbuffered on destination*. The transfer can take any place after the issuing of `bsp_hpput` and the next synchronization. Due to the lack of buffering, until the next synchronization, the origin process cannot write into the source memory after issuing `bsp_hpput` without potentially modifying the data that will be sent. Conversely, the target process cannot write the target memory, without potentially overwriting the received data. Furthermore, the same memory area in the same process cannot safely be the subject of several high-performance communications.

```
void bsp_get(int pid, const void *src, int offset, void *dst,
             int nbytes)
```

The function `bsp_get` requests the transfer of the memory area at the target `pid` that is in an active registration with `src` at offset `offset` and extending



where $\langle \ldots, (\texttt{src}, \_), \ldots, (\texttt{tgt}, \texttt{size}), \ldots \rangle$ is the active registration for `src`

Figure 2.10 – *Schema of the `bsp_get` remote memory read*

`nbytes` bytes, into the memory area of the origin process at `dst` (see schema in Figure 2.10).

If the write goes outside the memory area that the target process registered with `src`, that is if `offset + nbytes > si` where `si` is the extent that the specified for the registration, then this call is illegal. Like for `bsp_put`, if the origin process have registered `NULL` for `src`, then this call is illegal. Similarly, getting from oneself is implemented as a memory copy.

Like `bsp_put`, read requests issued by `bsp_get` are *buffered on source, buffered on destination* and delivered at the next synchronization. The high-performance variant of this function that forgoes such buffering, `bsp_hpget`, is described below.

```
void bsp_hpget(int pid, const void *src, int offset,
               void *dst, int nbytes)
```

The function `bsp_hpget` differs with `bsp_get` with respect to buffering. The function `bsp_hpget` is *unbuffered on source, unbuffered on destination*. When using `bsp_hpget`, the same safety precautions apply as when using `bsp_hpput`.

Bulk Synchronous Message Passing (BSMP)

Message passing is the second way of communicating between processes in BSPlib, but arguably less used. However, it is convenient in applications with irregular and data-dependent communication patterns.

The idea is that each process has a queue handled by the BSPlib runtime. During the computation phase of a superstep, processes can send messages into the queue of other processes. As for DRMA communications, messages are guar-

anteed to be delivered after synchronization and the receiving process can then read the messages from their queue. Messages are removed after being read, and any unread messages are removed at the end of the superstep. Unlike DRMA, the receiving process is not required pre-allocate memory, but can instead allocate memory according to received messages.

Each message consists of a *tag* and a *payload*, both with user-defined contents. The tag is fixed, user-defined size (0 bytes by default), where as the payload is variable length.

```
void bsp_set_tagsize(int *tag_bytes)
```
Sets the size of the tag in bytes. This function must be called collectively and with the same argument by each process. The change takes effect in the next superstep. The previous tag size is stored in `tag_bytes` on return.

```
void bsp_send(int pid, const void *tag, const void *payload,
              int payload_bytes)
```
Used to send tag and payload to the process specified by `pid`. Both payload and tag are buffered. Messages are delivered in the next superstep, but there are no guarantees on the delivery order. The tag size must conform to what has been set by `bsp_set_tagsize` in previous superstep.

```
void bsp_qsize(int *packets, int *accum_nbytes)
```
Returns the number of messages that has been received (i.e. sent by to the calling process in the previous superstep) and not yet read by `bsp_move`. Also writes the total size of all received, not yet read, messages in the location pointed to by `accum_nbytes`.

```
void bsp_get_tag(int *status, void *tag)
```
Reads the tag part of the next message in the queue. The location pointed to by `status` is set to `-1` if there are no more messages, otherwise it is set to the size of the payload in the message. If there is a tag, then it is copied to the address referred by `tag`.

```
void bsp_move(void *payload, int reception_bytes)
```
Reads and removes the next message in the queue. The payload of the message is copied to the address referred to by `payload`. The argument

`reception_bytes` is used to specify an upper limit on the number of bytes that can be written in to the memory referenced by payload.

Since `bsp_move` is used both to read and remove the message, it follows that (1) `bsp_move` must be called even if only the tag is used (i.e. all messages are fixed length), to be able to retrieve the next message; (2) `bsp_get_tag` must be called before `bsp_move`.

```
int bsp_hpmove(void **tag_ptr_buf, void **payload_ptr_buf)
```

The high-performance primitive `bsp_hpmove` reads both tag and payload of the next message without any copying, and removes the message from the queue. Whereas `bsp_move` respectively `bsp_get_tag` *copies* the tag respectively payload into specified memory areas, `bsp_hpmove` points its argument to the location of these two in the BSPlib queue. In returns `-1` if there is no next message, and otherwise the length of the payload.

2.3.6 BSPlib Implementations

Thanks to the small size of the API, BSPlib has become the *de facto* standard interface for imperative BSP programming, with several implementations.

Implementations for multi-node architectures include Oxford BSP Toolset [141], The Paderborn University BSP library (PUB) [27] and BSPonMPI [182]. Implementations for multi-core architectures include MulticoreBSP for Java [207] and C [208], Zefiros BSPlib [193], Bulk [34] and Epiphany BSP [33]. Lightweight Parallel Foundations has a BSPlib compatible interface and implementations for multi-node, multi-core and hybrid architectures [183].

2.3.7 BSPlib Limitations

Using `int` For Indices and Sizes

BSPlib uses the C type `int` for memory object sizes and indices (for instance, in the arguments for `bsp_push_reg`, `bsp_put`, `bsp_qsize` in Table 2.1), instead of the unsigned integer type `size_t`, which would be more appropriate for C. This is also the case in the API of MPI, and a possible reason is to ensure that primitives are callable from Fortran, which does not have unsigned integers. However, this hinders working with objects whose size exceeds `INT_MAX` bytes, the largest integer that can be represented by an `int`. For instance, the full extent of such objects cannot be registered.

```
1         int *p = malloc(sizeof(int));
2         int *q = malloc(sizeof(int));
3         bsp_push_reg(p, sizeof(int));
4         bsp_push_reg(q, sizeof(int));
5         bsp_sync();
6         bsp_pop_reg(p);
7         bsp_sync();
```

Figure 2.11 – *A BSPlib program with a potential registration error. If both calls to* `malloc` *fails and returns* `NULL` *in process* 0*, but succeed in other processes and return two distinct objects, then the call to* `bsp_pop_reg` *fails since process* 0 *attempts to remove the second registration in the registration sequence, but the others attempts to remove the first.*

Some BSPlib implementations [208] give an *alternate API* where the `size_t` type is used instead of int, along with a conformant BSPlib API.

Inflexible Buffer Size

The communication functions of BSPlib are buffered (except `bsp_hpget` and `bsp_hpput`). The size of these buffers are outside the user's control. Depending on the BSPlib implementation, they are either fixed size, with buffer overruns a potential problem, or they adapt size dynamically. The latter solution may lead to unpredictable performance, going against the BSPlib design goals of portability and predictability. Modern BSP libraries forgo buffering [183] to avoid these problems.

Registration

The registration mechanism used by BSPlib to create associations between memory objects in different processes is error prone. The example in Figure 2.11 demonstrate how the interaction of memory allocations and registrations can cause subtle heisenbugs. In Chapter 6, we lay the formal groundwork for a static verification of registrations.

Other parallel programming models propose less error-prone schemes to expose memory areas for DRMA. A common idea is to use a special data-type, either to encapsulate the exposed memory area, or to act as a handle for it. In the BSP paradigm, Yzelman et al. [207] give an example of the former approach. They use a communication container class to turn regular objects into distributed data structures. MPI [139] uses the latter approach, calling their handles *windows*. Like BSPlib registrations, windows act as handles and are created and removed collectively. Unlike registrations, windows can be removed in any order.

Lightweight Parallel Foundation's [183] *memory slots* are similar to windows. An idea that forgoes dedicated data-types is proposed in OpenSHMEM [36] where DRMA operations are restricted to "symmetric" objects that the runtime system ensures have the same relative address in each process.

Lack of High-Level Collectives

One drawback of the minimalism of BSPlib is that commonly used parallel programming building blocks and high-level collective must be implemented by the user. Notably, BSPlib contains no primitives for common communication patterns such as broadcast, scatter, all-to-all, gather and reduction. A "Level 1" BSPlib API with such high-level primitives has been proposed [100], but to the best of our knowledge, no modern BSPlib libraries implement this API.

Lack of Composability and Fault Tolerance

In many situations, it would be desirable to be able to include BSPlib to speed up computations in a larger application. However, three design flaws of BSPlib render such use cases difficult, namely the restriction to one parallel section (see Section 2.3.3), the lack of any provision to return data from this section to the rest of the application and the lack of fault tolerance.

In a BSPlib program, either the parallel section is the `main` function, immediately precluding having it be a part of a larger application, as C programs have at most one entry point. Or, a SPMD function containing the parallel section is designated by a call to `bsp_init`. However, the function pointer that should be passed to the `bsp_init` function has the `void` return type and no parameters. Hence, the only way to return data from the parallel section to the following serial section in this case is by global variables, whose usage is considered contrary to good software engineering practices.

While the function `bsp_abort` can be used to terminate completely a program using BSPlib in case of errors, there is no mechanism in BSPlib to recover *gracefully* from faults. Should for instance memory allocation fail in some process, then there is no simple way to propagate this error, terminate the parallel computation and return to the serial section to clean up any other eventual resources that have been allocated.

Loss of High-Level Structure

A critique that can be leveraged to any SPMD language implemented in imperative language is the loss of the high-level structure of synchronization.

As noted by [179], a BSP program can either be seen as a sequence of supersteps, each containing the parallel computation, or as the parallel composition of sequential **p** instruction streams, equipped with a synchronization primitive. BSPlib implements the latter approach, by giving users access to the `bsp_sync` primitive.

The drawback of this approach is that there is no restriction from expressing programs that have no sense in the BSP model, such as the following:

```
1 if (bsp_pid() == 0) { bsp_sync(); } else { bsp_end(); }
```

which will cause a synchronization error when executed by at least 2 processes.

The first model can be implemented more easily in functional languages, where computations (in the guise of functions) are first-class values. This is indeed similar to how programs are expressed in BSML [18]. There, programs are divided into a global part which sequences a series of local parts, expressed as functions and corresponding to supersteps. Errors such as the synchronization error above are ruled out by forcing all collective operations to be performed in the global part.

2.3.8 Relationship to MPI

In spirit, BSPlib can be seen as a model of the Bulk Synchronous subset of the popular parallel programming library MPI [139]. Both BSPlib and MPI are SPMD libraries for distributed memory programming. The unbuffered, high-performance RDMA operations of BSPlib (`bsp_hpput` and `bsp_hpget`) corresponds to the RMA operations of MPI [89, p. 55] (`MPI_Put` and `MPI_Get`) and the synchronization primitive `bsp_sync` of BSPlib to the `MPI_Barrier` of MPI; BSPlib registers corresponds to MPI windows; the BSPlib process identifier corresponds to ranks in the MPI. We summarize this approximate correspondence in Table 2.2.

The reader should be aware that this correspondence should not be read as a semantic equivalence as there are many subtle differences. To name two, MPI implements *communicators* that group sets of processes, a feature with no equivalent in BSPlib. Secondly, the registration sequence that enables DRMA in BSPlib follows a stack-like semantics, as detailed above in Section 2.3.5. This is

Function	BSPlib	MPI
Inquiry	`bsp_pid`	`MPI_Comm_rank`
	`bsp_nprocs`	`MPI_Comm_size`
Synchronization	`bsp_sync`	`MPI_Barrier`
DRMA Setup	`bsp_push_reg`	`MPI_Win_Create`
	`bsp_pop_reg`	`MPI_Win_Destroy`
DRMA	`bsp_hpput`	`MPI_Put`
	`bsp_hpget`	`MPI_Get`

Table 2.2 – *An approximative and incomplete Rosetta Stone translating between MPI and BSPlib*

not the case in MPI, where windows are added and removed independently from each other.

Nonetheless, the correspondence is close enough that some authors have developed implementations of parallel algorithms that run under both MPI and BSPlib by wrapping corresponding primitives in macros [82].

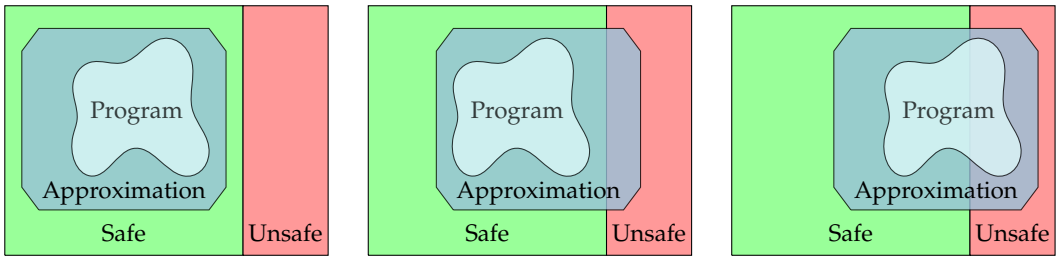
2.4 The Data-Flow Approach to Static Analysis

The goal of program analysis [151] is to develop algorithms to decide whether a program has a specific semantic property or not. Uses for such algorithms are not hard to find: we could aim to discover whether a certain program terminates, whether it executes without error, where a variable is no longer used, and so forth. Unfortunately, the consequence of Rice's Theorem [162] is that it is not possible to design such decision algorithms for Turing complete programming languages.

However, it is possible to make semi-algorithms for such problems. A general algorithm for a decision problem answers *yes* or *no*. The semi-algorithms instead answers *yes* or *I don't know*.

Static analyses are program analyses that do not execute the analyzed program. There are many approaches to static analysis: Type and Effect Systems, Control and Data-Flow Analysis, Symbolic Execution are some examples. However, all these techniques have a deeper connection through the general theory of *abstract interpretation* [48].

Indeed, the use of *abstraction* is a unifying theme behind these techniques. Abstraction is used to "compress" and reason about large, intractable set of possible executions. Abstraction is applied to *program states* (e.g. the contents of the memory), to compactly represent many possible states. Abstraction is applied to *operations* (e.g. updating the contents of the memory). It is important that this



(a) Safe program proven safe (b) Spurious warning (c) True warning

Figure 2.12 – *Over-approximations of program behaviors. The set of program behaviors has been classified into safe and unsafe. The feasible behaviors of the program is represented by a blob, nested in an octagon representing their static over-approximation by a static analysis. In the case (a), program can be safe. The analysis cannot distinguish the cases (b) and (c), and thus cannot show that the program in (b) is actually safe.*

abstraction is done *safely*. Safety follows by ensuring that abstract operations correctly over-approximate concrete operations.

The price payed for abstraction is the inclusion of spurious behaviors that do not exist in the concrete program. This situation is illustrated by Figure 2.12. By showing that an over-approximation of the program's behaviors lies in a safe subset of behaviors the program analysis guarantees the program's safety, and answers *yes* (Figure 2.12(a)). But if the over-approximation is not fully contained by the safe subset, then the actual program behavior might be safe, but it might also not be, and the analysis must answer *I don't know*. These last two situations are illustrated Figures 2.12(b) and 2.12(c).

In this section we give the required notions to go from the intuition of abstractions to implementable static analyses in the form of *data-flow analyses*, the technique employed in this thesis to analyze BSPlib programs. The presentation in this section is heavily indebted to [151].

2.4.1 The Sequential Language Seq

We demonstrate the concepts of data-flow analysis on the small imperative language **Seq**. This language forms the sequential core of the formalization of BSPlib that will be used throughout the thesis. The expressions and instructions of **Seq** are defined by the following grammar:

$$
\begin{array}{llll}
\textbf{AExp} & \ni e & ::= & x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \\
\textbf{BExp} & \ni b & ::= & \texttt{true} \mid \texttt{false} \mid e_1 < e_2 \mid e_1 = e_2 \mid b_1 \textbf{ or } b_2 \mid b_1 \textbf{ and } b_2 \mid !b \\
\textbf{Seq} & \ni s & ::= & [x{:=}e]^\ell \mid [\texttt{skip}]^\ell \mid s_1; s_2 \mid \texttt{if } [b]^\ell \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \\
 & & \mid & \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}
\end{array}
$$

$[y{:=}57]^1$;
$[d{:=}8]^2$;
$[x{:=}0]^3$;
`while` $[d < y$ **or** $d = y]^4$ `do`
  $[x{:=}x + 1]^5$;
  $[y{:=}y - d]^6$
`end`

Figure 2.13 – *The* **Seq** *program* s_{div}

where $x \in$ **Var**, the set of program variables, and $n \in$ **Nat**, the set of natural numbers. Instructions are labeled with labels $\ell \in$ **Lab**. We shall assume that all programs are consistently labeled, i.e. that each label appears at most once.

The language contains arithmetic and boolean expressions, as defined by the syntactic groups **AExp** and **BExp**. The instructions are formed by the syntactic group **Seq**, and consists of assignments, the skip instruction that does nothing, sequences of instructions, conditionals and loops. An example of a **Seq** program is given in Figure 2.13. This program s_{div} calculates the euclidean division of 57 by 8. At the end its execution x contains the quotient and y the remainder, so that $8x + y = 57$.

The Semantics of Seq

The precise meaning of **Seq** is defined by its formal semantics that describes how the execution of programs transforms memory states. Memory states are represented by mappings from variables to natural integers. This is a standard, big-step operational semantics [203]. The big-step semantics is a relation $\rightarrow$ between initial and final configurations of non-diverging and non-erroneous program executions. Initial configurations $\langle s, \sigma \rangle$, consists of a program $s \in$ **Seq** to execute and an initial memory state $\sigma \in$ **State**. Final configurations are the memory state at the end of execution. In sum, we have:

$$\begin{array}{lcl} \textbf{State} & = & \textbf{Var} \rightarrow \textbf{Nat} \\ \rightarrow & : & \textbf{Seq} \times \textbf{State} \times \textbf{State} \end{array}$$

The informal idea of a variable x containing the value n is formalized by $\sigma(x) = n$. The updated memory state where x is set to contain m is written $\sigma[x \leftarrow m]$ and defined by

$$\sigma[x \leftarrow m] = \lambda y. \begin{cases} m & \text{if } x = y \\ \sigma(y) & \text{otherwise} \end{cases}$$

Before defining $\rightarrow$, we define the semantics of arithmetic respectively boolean

$$\begin{cases} \mathcal{A}[\![\cdot]\!] & : \quad \textbf{AExp} \to (\textbf{State} \to \textbf{Nat}) \\ \mathcal{A}[\![x]\!]\,\sigma & = \quad \sigma(x) \\ \mathcal{A}[\![n]\!]\,\sigma & = \quad n \\ \mathcal{A}[\![e_1 + e_2]\!]\,\sigma & = \quad \mathcal{A}[\![e_1]\!]\sigma + \mathcal{A}[\![e_2]\!]\sigma \\ \mathcal{A}[\![e_1 - e_2]\!]\,\sigma & = \quad \mathcal{A}[\![e_1]\!]\sigma - \mathcal{A}[\![e_2]\!]\sigma \\ \mathcal{A}[\![e_1 \times e_2]\!]\,\sigma & = \quad \mathcal{A}[\![e_1]\!]\sigma \times \mathcal{A}[\![e_2]\!]\sigma \end{cases}$$

(a) Semantics of arithmetic expressions

$$\begin{cases} \mathcal{B}[\![\cdot]\!] & : \quad \textbf{BExp} \to (\textbf{State} \to \textbf{Bool}) \\ \mathcal{B}[\![\texttt{true}]\!]\,\sigma & = \quad \texttt{tt} \\ \mathcal{B}[\![\texttt{false}]\!]\,\sigma & = \quad \texttt{ff} \\ \mathcal{B}[\![e_1 < e_2]\!]\,\sigma & = \quad \texttt{tt} \text{ if } \mathcal{A}[\![e_1]\!]\,\sigma < \mathcal{A}[\![e_2]\!]\,\sigma, \texttt{ ff oth.} \\ \mathcal{B}[\![e_1 = e_2]\!]\,\sigma & = \quad \texttt{tt} \text{ if } \mathcal{A}[\![e_1]\!]\,\sigma = \mathcal{A}[\![e_2]\!]\,\sigma, \texttt{ ff oth.} \\ \mathcal{B}[\![b_1 \textbf{ or } b_2]\!]\,\sigma & = \quad \texttt{tt} \text{ if } \mathcal{B}[\![b_1]\!]\,\sigma \textbf{ or } \mathcal{B}[\![b_2]\!]\,\sigma, \texttt{ ff oth.} \\ \mathcal{B}[\![b_1 \textbf{ and } b_2]\!]\,\sigma & = \quad \texttt{tt} \text{ if } \mathcal{B}[\![b_1]\!]\,\sigma \textbf{ and } \mathcal{B}[\![b_2]\!]\,\sigma, \texttt{ ff oth.} \end{cases}$$

(b) Semantics of boolean expressions

Figure 2.14 – *Semantics of expressions in* **Seq**

expressions by the functions $\mathcal{A}[\![\cdot]\!]$ respectively $\mathcal{B}[\![\cdot]\!]$ in Figure 2.14. The function $\mathcal{A}[\![\cdot]\!]$ maps an arithmetic expression to a function from a memory state to a natural number. Similarly, the function $\mathcal{B}[\![\cdot]\!]$ maps a boolean expression to a function from a memory state to a member of **Bool**. Note that by abuse of notation, the operators appearing on the left-hand side in the equations defining $\mathcal{A}[\![\cdot]\!]$ and $\mathcal{B}[\![\cdot]\!]$ are symbol of the syntax in **Seq**, whereas operators on the right-hand side are the corresponding the mathematical operators.

The semantics of instructions is given by the big-step rules in Figure 2.15. The rules are read as implications: when the premises in the numerator holds, the conclusion in the denominator defines a member of the relation. These rules are standard and we do not expound on their meaning, and instead refer to a standard textbook on semantics such as [203].

In the following chapters, we will extend **Seq** with parallel primitives to formalize BSPlib, but this sequential fragment suffices for the purpose of illustrating data-flow analysis.

2.4.2 Control Flow Graph

Instead of operating directly on the syntax of the program, data-flow analysis operates on the "control flow graph" of the program. This directed graph is

$$\frac{}{\langle [\texttt{skip}]^\ell, \sigma\rangle \to \sigma} \textsc{skip}$$

$$\frac{\langle s_1, \sigma\rangle \to \sigma'' \quad \langle s_2, \sigma''\rangle \to \sigma'}{\langle s_1; s_2, \sigma\rangle \to \sigma'} \textsc{seq}$$

$$\frac{}{\langle [x{:=}e]^\ell, \sigma\rangle \to \sigma[x \leftarrow \mathcal{A}[\![e]\!]\,\sigma]} \textsc{assign}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{tt} \quad \langle s_1, \sigma\rangle \to \sigma'}{\langle \texttt{if } [b]^\ell \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}, \sigma\rangle \to \sigma'} \textsc{if\_true}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{ff} \quad \langle s_2, \sigma\rangle \to \sigma'}{\langle \texttt{if } [b]^\ell \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}, \sigma\rangle \to \sigma'} \textsc{if\_false}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{tt} \quad \langle s, \sigma\rangle \to \sigma'' \quad \langle \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma''\rangle \to \sigma'}{\langle \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma'} \textsc{while\_true}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{ff}}{\langle \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma} \textsc{while\_false}$$

Figure 2.15 – *Operational big-step semantics of* **Seq** *programs*

constructed from a program by adding one node per program point, as identified by the labels occurring in the program, and then adding edges between two nodes if control can flow from the former to the latter.

For a program s, we also distinguish one initial node $\mathit{init}(s)$ and a set of final nodes $\mathit{final}(s)$. Intuitively, the former corresponds to the point of the program where execution starts, and the latter, where execution may end. The set of all labels in s, is given by $\mathit{labelss}$.

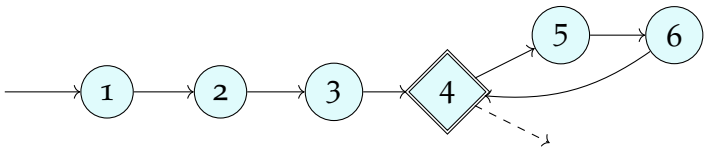
Formally,

$$\begin{cases} \mathit{init} & : \ \mathbf{Seq} \to \mathbf{Lab} \\ \mathit{final} & : \ \mathbf{Seq} \to \mathcal{P}(\mathbf{Lab}) \\ \mathit{flow} & : \ \mathbf{Seq} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab}) \\ \mathit{labels} & : \ \mathbf{Seq} \to \mathcal{P}(\mathbf{Lab}) \end{cases}$$

These constructs are standard. The CFG and the value of these functions for s_{div} are given in Figure 2.16. However, we omit the definition of these functions and refer to the standard text book [151] for more details.

2.4.3 Data-Flow Analysis

Data-flow analysis derives semantic properties associated to program points, represented by the nodes of the program's CFG. These *program properties*, that in the context of data-flow analysis are sometimes called *data-flow facts* or *ab-*



$$
\begin{array}{lcl}
\mathit{flow}(s_{\text{div}}) & = & \{(1,2),(2,3),(3,4),(4,5),(5,6),(6,4)\} \\
\mathit{init}(s_{\text{div}}) & = & 1 \\
\mathit{final}(s_{\text{div}}) & = & \{4\} \\
\mathit{labels}(s_{\text{div}}) & = & \{1,2,3,4,5,6\}
\end{array}
$$

Figure 2.16 – *The control flow graph of s_{div}*

stract states, move along the edges of the program's CFG. Each program point has an incoming, respectively outgoing, abstract state that represents what is always true in all executions before, respectively after, executing that node. The incoming abstract state is an over-approximation of all incoming abstract states, and the outgoing abstract state is calculated by applying a *transfer function* to the input state.

The analysis is classified depending on the direction the data-flow facts run. In a "forward analysis", data-flow facts are propagated in the direction of control flow. In a "backward analysis", they are propagated against the control flow.

The former is used when the property to be computed is a function of all possible past actions, whereas the latter is used when the all future actions are of interest. For instance, a forward analysis can be used to over-approximate the set of variables that may have been defined up to a certain point (see Reaching Definitions below), whereas a backward analysis can be used to over-approximate the set of variables that may be read at some later point (Live Variables analysis).

In addition to establishing safety properties as in this thesis, data-flow analysis is commonly used in compilation. To give an example, "Reaching Definitions" is a forward data-flow analysis useful for compilation. In an execution, a **definition** is a variable and a program point at which it was assigned. A definition reaches another program point if there are no other assignments to that variable between the definition and the program point.

Consider the assignment to x labeled 3 in s_{div}. This definition $(x,3)$ may reach program point 4 (in the first iteration of the loop) but so can the definition $(x,5)$ (for the remaining iterations). By the same reasoning $(x,3)$ and $(x,5)$ reaches point 5. However, in all executions that reaches the assignment labeled 6, x must have been redefined at 5. Thus $(x,5)$ reaches this point, but not $(x,3)$.

Line	Reaching Definitions
1	$\{(y,?),(d,?),(x,?)\}$
2	$\{(y,1),(d,?),(x,?)\}$
3	$\{(y,1),(d,2),(x,?)\}$
4	$\{(y,1),(d,2),(x,3),(x,5),(y,6)\}$
5	$\{(y,1),(d,2),(x,3),(x,5),(y,6)\}$
6	$\{(y,1),(d,2),(x,5),(y,6)\}$

Table 2.3 – *Reaching Definitions in the program* s_{div}

All the reaching definitions of s_{div} are given in Table 2.3. The definition $(x,?)$ at a program point signifies that execution may reach that point with x uninitialized.

In the following sections, we show how the Reaching Definitions analysis statically over-approximates the set of reaching definitions for all program points.

2.4.4 Abstract Domain

The abstract domain L of a data-flow analysis is a structure containing the semantic properties to be associated to program points. The choice of domain depends the set of facts one would like to derive.

However, the data-flow analysis framework imposes some structure on L. It should be partially ordered by some order $\sqsubseteq$ (i.e. a poset). This order formalizes the notion that some properties are more precise than others. The framework also requires that L is a complete lattice. That is, each set of properties $X \subseteq L$ has a *least upper bound*, written $\bigsqcup X$, and a *greatest lower bound*, written $\bigsqcap X$. An upper (respectively lower) bound on X is an element $l \in L$ such that $\forall l' \in X : l' \sqsubseteq l$ (resp. $\forall l' \in X : l \sqsubseteq l'$). Then for any such upper (respectively lower) bound l, we have $\bigsqcup X \sqsubseteq l$ (respectively $l \sqsubseteq \bigsqcap X$). It is often convenient to define binary versions of these bounds: $l_1 \sqcup l_2$ is defined by $\bigsqcup\{l_1, l_2\}$ and $l_1 \sqcap l_2$ by $\bigsqcap\{l_1, l_2\}$. We distinguish the elements $\bigsqcup X$ and $\bigsqcap X$ that are written $\top$ and $\bot$ and pronounced *top* and *bottom*.

We also require that L satisfies the Ascending Chain Condition. That is, for any infinite sequence of properties $(l_i)_i$ in L such that $l_0 \sqsubseteq l_1 \sqsubseteq \cdots$, there is n such that $l_n = l_{n+1} = \cdots$. As we will see below, this along with the monotonicity of the transfer functions ensures termination of the analysis.

For the sake of the analysis, we also distinguish an *extremal* abstract state ι that represents the initial state of the program.

In the Reaching Definitions analysis, we associate sets of definitions with program points, and take $L = \mathcal{P}(\mathbf{Var}_s \times \mathbf{Lab}_s)$, where $\mathbf{Var}_s$, respectively $\mathbf{Lab}_s$,

are the finite sets of variables, respectively labels, that appear in the analyzed program s. The analysis being an over-approximation, we order the properties by subset inclusion $\subseteq$. This concurs with the intuition of order as a measure on precision: we can add more definitions to each program point and retain a sound over-approximation (all concrete behaviors are still in the abstraction), but we lose in precision.

The least upper bound of $X \subseteq L$ now translates to the "smallest set containing all sets in L" which is exactly $\bigcup L$. Similarly, the greatest lower bound is $\bigcap L$. It can be shown that all finite posets fulfill the ascending chain condition, and so this also this L. Finally, since all variables are uninitialized in the initial state of the program, the extremal abstract state is given by $\{(x, ?) \mid \forall x \in \mathbf{Var}_s\}$.

2.4.5 Transfer Functions

Each program point of the analyzed program is associated with an incoming abstract state and an outgoing abstract state.

The incoming abstract state is computed from the abstract state of preceding points in the control flow graph. The analysis cannot assume how execution arrived there, and so takes the least upper bound to obtain an over-approximation of properties of preceding program points.

In Reaching Definitions, this corresponds intuitively to saying that the set of definitions that can reach a program point is the union of the definitions flowing from the preceding points.

The outgoing abstract state at point ℓ is obtained by applying its *transfer function* to the incoming abstract state. The aim of this transfer function is to translate the concrete operation of program point ℓ into the abstract domain

$$f_\ell : L \to L$$

The transfer function f_ℓ must be monotone: if $l_1 \sqsubseteq l_2$ then $f_\ell(l_1) \sqsubseteq f_\ell(l_2)$. Intuitively, this corresponds to requiring that when the input of the transfer function is less precise, then so is the output.

Consider the transfer function of Reaching Definitions. Assume x is assigned at label ℓ. Intuitively, the set of reaching definitions immediately after executing this assignment must contain (x, ℓ). Furthermore, all previous reaching definitions of x must be removed. Formally, we write this:

$$f_\ell(X) = (X \setminus \{(x, \ell) \mid \forall (x, \ell) \in X\}) \cup \{(x, \ell)\}$$

No other instructions in the language modifies definitions by assigning variables and so their transfer function is the identity function. We can easily verify the monotonicity of the identity function and the transfer function for assignments.

2.4.6 Calculating Solution Through Fixpoint Iteration

By assembling the elements defined above, we characterize a (forward) data-flow analysis by a 6-tuple $(L, \sqsubseteq, \bigsqcup, \bot, \iota, f)$ where

- L is the abstract domain;
- $\sqsubseteq \subseteq \mathcal{P}(L \times L)$, orders properties by decreasing precision;
- $\bigsqcup : \mathcal{P}(L) \to L$, the least upper bound, combines properties;
- $\iota \in L$, the extremal abstract state, is the property representing the initial concrete state
- $f_{(\cdot)} : \mathbf{Lab} \to (L \to L)$, maps program points to transfer functions;

and the additional requirements that $(L, \sqsubseteq)$ is a complete lattice that satisfies the ascending chain condition and that f_ℓ is monotone for all ℓ.

From this 6-tuple, we construct an equation system $\mathit{Analysis}(s) = (\mathit{Analysis}_\circ, \mathit{Analysis}_\bullet)$ for the program s, where $\mathit{Analysis}_\circ(\ell)$ respectively $\mathit{Analysis}_\bullet(\ell)$ is the incoming respectively outgoing abstract state associated with ℓ. They are defined in the following way:

$$\begin{cases} \mathit{Analysis}_\circ, \mathit{Analysis}_\bullet & : \quad \mathbf{Lab} \hookrightarrow L \\ \mathit{Analysis}_\circ(\ell) & = \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in \mathit{flow}(s)\} \sqcup \iota^\ell \\ & \qquad \text{where } \iota^\ell = \begin{cases} \iota & \text{if } \ell = \mathit{init}(s) \\ \bot & \text{otherwise} \end{cases} \\ \mathit{Analysis}_\bullet(\ell) & = f_\ell(\mathit{Analysis}_\circ(\ell)) \end{cases}$$

A solution sol is a pair of functions $(\mathit{sol}_\circ, \mathit{sol}_\bullet)$ with $\mathit{sol}_\circ, \mathit{sol}_\bullet : \mathbf{Lab} \hookrightarrow L$ that satisfies the equation system above. We then write $\mathit{sol} \models \mathit{Analysis}(s)$. Such a solution can be found by a standard fix-point iteration [151] whose termination is guaranteed by the monotonicity of the transfer functions and the Ascending Chain Condition of L.

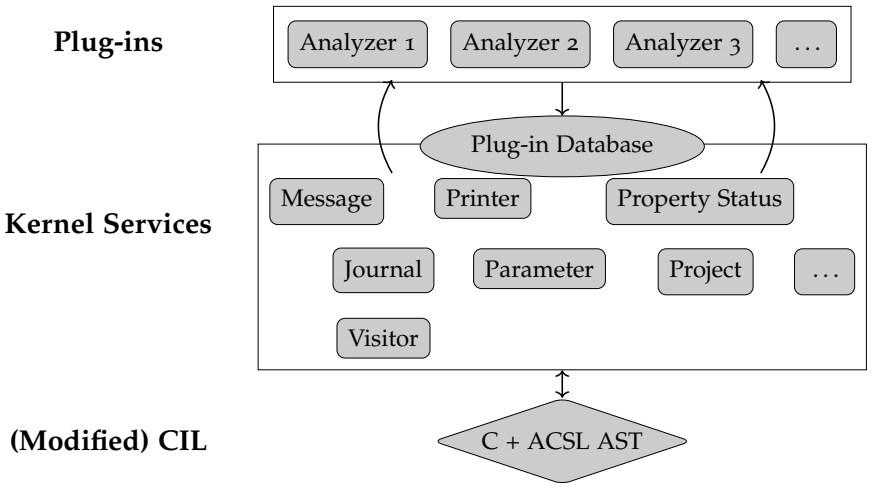


Figure 2.17 – *Frama-C architecture*

2.5 Frama-C

This section is extracted from the author's master's thesis [113].

Frama-C [54] is an analysis platform for C code implemented in OCaml. It consists of a kernel and a set of plug-ins (see Figure 2.17). The Frama-C kernel parses input programs, and offers an abstract view of C programs to the plug-ins that implement different analyses, or program transformations. The kernel also provides several services that aid the implementation of new plug-ins, such as several facilities for implementing data-flow analyses, and it ensures the consistency of plug-ins.

One benefit of regrouping several tools in the same framework is that each individual tool does not need to deal with the technical aspects of parsing C code and can thus concentrate on implementing one specific analysis. In addition to this, the kernel of Frama-C simplifies the input program and offers a slightly abstracted view to the plug-in. Another positive aspect of the plug-in architecture is that one plug-in can share its results with other plug-ins. For example, the RTE plug-in identifies source code locations that can cause a run-time error. The verification engineer can then use the deductive verification plug-in WP to rule out the possibility of actual errors in these locations.

Since verification engineers often take a multi-pronged approach to verifying a program, using both automatic, semi-automatic and manual techniques, it is convenient to gather tools using different kinds of techniques under the same umbrella.

Frama-C has a rich set of plug-ins. Some important ones are:

- WP that enables program proving by weakest precondition calculus [63];
- EVA [25] that over-approximates the set of possible values of any program variables at any program point using abstract interpretation [48];
- PathCrawler [202] that generates test cases for a rigorous all-path (or *k*-path) coverage criterion, using a combination of concrete and symbolic execution; and
- E-ACSL [61] that enables dynamic verification of ACSL specifications.

Notable uses of Frama-C include the verification of the Open Source cryptographic library PolarSSL[4], where Frama-C has been used to prove the absence of memory errors of the type which caused the infamous Heartbleed bug[5] in OpenSSL. Frama-C has also been used to check the absence of run-time errors in instrumentation and control (I&C) nuclear code, using a combination of Value and WP [52]. Additionally WP is used at Airbus SAS for verifying control software for Airbus aircraft.

The Frama-C kernel reads the input C program, that can include annotations in the ACSL specification language. The input program along with the specification is parsed into an abstract syntax tree (AST), using the CIL library [147].

CIL translates the source program into a simplified intermediate language. Some of the simplifications include:

- turning all expressions side-effect free,
- normalizing loop constructs (all loops are translated into `while`-loops),
- using single `return` in each function,
- expanding compound array-initializations,
- replacing ternary operator (`-?-:-`) with `if-then-else`.

These simplifications greatly decrease the amount of cases that need to be handled by a program analysis or transformation, since several cases can be handled in one. For instance, instead of dealing separately with `while`, `do-while` and `for`-loops, a plug-in only has to handle the `while`-loop.

Frama-C then passes on the simplifies AST of the program and applies the analyses and transformations requested by the user.

[4] `http://trust-in-soft.com/polarssl-verification-kit/`
[5] `http://en.wikipedia.org/wiki/Heartbleed`

State of the Art

3

Contents

3.1 Parallel Models . . . 56
3.1.1 Other than BSP . . . 56
3.1.2 BSP Extensions . . . 57
3.2 Parallel Programming . . . 60
3.2.1 Other than BSP . . . 60
3.2.2 BSP . . . 63
3.3 Formal Methods for Scalable Parallel Programming . . . 64
3.3.1 Deductive Verification . . . 65
3.3.2 Model Checking . . . 68
3.3.3 Static Analysis . . . 70
3.3.4 Other Formal Methods . . . 79
3.4 Discussion . . . 79

In this chapter we review the state of the art in formal methods for *scalable parallel programming* to give the context of the three main contributions of this thesis: static analysis of parallel structure and synchronization, static cost analysis and a sufficient condition for safe registration in BSPlib programs.

We first review models of scalable parallel programming and how they relate to BSP in Section 3.1. We focus in particular on models for data-parallelism. Such models are implemented in libraries and programming languages, which we review in the Section 3.2.

Our review then surveys the major families of formal methods and how they have been adapted for scalable parallelism, both to BSP and other models in Section 3.3. We take special interest in *static analysis*, and devote Section 3.3.3 to static analyses for scalable parallelism in the BSP model and other models.

We close the chapter in Section 3.4 by positioning the main contributions of this thesis with respect to the findings of the literature review.

3.1 Parallel Models

As the importance of parallelism to scalable computing became apparent, several *models* of parallel computing have been proposed. As with any kind of model, it is desirable that the model abstracts away enough irrelevant details to allow simple reasoning on high-level properties such as correctness and performance, while remaining realistic enough to be practical. In other words, the high-level properties of the model should, at least to some degree, translate to the concrete computation that is modeled.

An analogy can be made with static analysis: a key challenge when applying formal methods to programs is finding appropriate abstractions that are abstract enough to be computable, but concrete enough to give pertinent information to the user.

Our study targets programs written in BSPlib for the BSP [191] model of parallel computation. BSP is formulated as a bridging model for parallel computing, analogous to the von Neumann model of sequential computing. It allows for reasoning on parallel computation in a sufficiently abstract setting.

To situate BSP, we briefly review in the following sections other parallel models that have been proposed. This includes extensions of BSP (Multi-BSP [192] and Scatter-Gather Language model [128]) but also other models (PRAM [75] and LogP [51]). For a more extensive review of parallel models, we direct the reader to [10].

3.1.1 Other than BSP

PRAM

The main purpose of the PRAM model [75], Parallel Random Access Machine, is to model how parallel algorithms share work. A Parallel Random Access Machine consists of a fixed number p of processors that execute in lockstep with equal (random) access to a shared memory. All communication go through this shared memory. There is no synchronization mechanism.

The PRAM model allows reasoning on the amount of parallelism in programs. That is, to which degree increasing p speeds up the computation. However, PRAM does not allow to reason on the cost of communication, which is often the bottleneck of parallel computations. The lockstep model of PRAM is

also ill-adapted to modeling realistic architectures where, on the contrary, computation nodes progress independently of each other.

LogP

The LogP model [51] aims to provide a more realistic model of parallel computation than PRAM by accounting for communication. LogP models distributed architectures where processes execute asynchronously. Memory is distributed and processes communicate by point-to-point message passing.

The model is named after the 4 parameters L, o, g, P used to characterize the performance of the message passing network in the underlying parallel architecture. The parameter L quantifies the upper bound on communication latency; the overhead o the "incompressible send or receive" cost; g the minimum "gap" between each message, that is, the inverse of the bandwidth; P the number of processes. By plugging the parameters of a specific parallel architecture into the communication pattern of an algorithm, its designer can predict its performance in that architecture.

Two main differences distinguish the LogP and the BSP model. First, while both models account for communication costs, LogP does not consider contention [76]. A communication pattern is modeled with the same cost whether balanced or not. For instance, LogP does not distinguish the cost of the "all-to-one" and the "shift" communication patterns. In the former, one process receives one word from each process. In the latter, each process sends a word to a distinct process. Shift is balanced, since each process sends and receives one word. All-to-one is unbalanced, since one process receives P words and sends none, while the others send one word and receive none. In LogP, only total communication volume counts, and it is the same in both patterns. In reality, the contention at the receiver gives all-to-one longer run time.

Second, LogP imposes less structure on the parallel computation. As result, LogP is more flexible, but gives less insight into how computation and synchronization affect the total cost. The more rigid superstep structure of BSP also serves to guide algorithm design: no such guidance is given by the LogP model.

3.1.2 BSP Extensions

Several BSP extensions have been proposed to compensate for perceived shortcomings in its original formulation. A common critique is that it does not con-

sider the *hierarchality* and *heterogeneity* that is common in modern parallel architectures.

BSP, like PRAM and LogP, models *flat* and *homogeneous* parallel computers, where each computation node is identical in terms of computation capacities and the bandwidth between each pair of nodes is the same. This view is often at odds with concrete parallel computers that consist of heterogeneous computers situated in hierarchical networks. A common example is multi-core nodes arranged in a tree-like network. Typically, inter-core communication is faster than inter-node communication.

We here consider two extensions of BSP, Scatter-Gather Language model [128] and Multi-BSP [192], that are tailored for this kind of heterogeneous and/or hierarchical parallel computers, while retaining the portability and predictability of BSP.

Scatter-Gather Language model

The Scatter-Gather Language (SGL) model [128] considers multi-level, heterogeneous parallel architectures. A SGL machine is a recursively defined tree structure, with a *root-master* at the top. The child nodes are either themselves SGL machines or "leaf-workers". As in BSP, parallel computation is arranged in supersteps. These are composed of four phases: (1) the master broadcasting data to its child nodes (scatter); (2) local computation in the child nodes; (3) each child node transfers data to its master (gather); (4) local computation in the master. The local computation step of child nodes that are themselves SGL machines is recursively composed of a sequence of supersteps.

The SGL cost model is parameterized by the scatter bandwidth $g_{\downarrow}$, the gather bandwidth $g_{\uparrow}$ and the synchronization cost l of each SGL machine in the network. The cost of a SGL execution is the sum of the cost of each superstep. The cost of one SGL superstep is the sum of the cost of each phase: $(h_{\downarrow}g_{\downarrow} + l) + w_c + (h_{\uparrow}g_{\uparrow} + l) + w_m$ where $h_{\downarrow}$ respectively $h_{\uparrow}$ is the number of words scattered respectively gathered, w_c is the cost of the longest local computation of any child node and w_m is the cost of local computation performed by the master. The local computation cost of child nodes that are themselves SGL machines is recursively defined in the same way.

The SGL model is simple yet expressive enough for several fundamental parallel algorithms. Elegantly, its hierarchical structure generalizes a restricted form of flat BSP communication. Namely, a SGL computer with two levels (one root node and its leaf-workers) can be seen as a BSP computer that alternates between

scatter and gather supersteps. However, SGL does not as of yet have any implementations. In the next section, we discuss the Multi-BSP model [192] that can be considered a generalization of SGL model and which has been implemented in programming libraries and languages [208, 27, 11].

Multi-BSP

Multi-BSP generalizes BSP to a balanced tree of networked compute nodes with homogeneous processing power but heterogeneous network speeds. Internal nodes have memory, and leaf nodes have memory and compute power.

Like in Scatter & Gather, the Multi-BSP execution proceeds in nested supersteps organized by levels. For a given node, the superstep is arranged in three phases similar to those of BSP: (1) the children of the node perform local computation in parallel; (2) the children exchange messages with the parent node[1]; (3) the children synchronize with each other. As in the Scatter & Gather model, the children might themselves be internal nodes. In this case, their local computation step is composed of a sequence of supersteps on the lower level.

The cost model is that of BSP applied recursively. Assume a Multi-BSP computer in L levels. Let the bandwidth and latency parameters of level i be given by parameters g_i and l_i. Consider an execution where N_i is the number of supersteps at the ith level, $h_{k,i}$ is the maximum of all h-relations on the ith level in superstep k and $w_{k,i}$ is the maximum local computation on the ith level in superstep k. The cost of this execution is given by $\sum_{k=0}^{L-1} (\sum_{i=0}^{N_k-1} w_{k,i} + h_{k,i}g_k + l_k)$.

Discussion

Hierarchical models such as Scatter & Gather and Multi-BSP have received comparatively little attention compared to flat models such as BSP. One potential reason is that many important parallel algorithms require all-to-all communication (e.g. Tiskin–McColl sort [186]). The performance bottleneck of the all-to-all is the slowest link in the network of the parallel computer. Considering such algorithms in a hierarchical model with more parameters than a flat model does not give better performance predictions: the slowest network parameter can be taken as a pragmatic upper bound on all network parameters, simplifying the model while obtaining similar predictions.

[1]Compute nodes on the same level can exchange messages through the parent in two supersteps. Multi-BSP model is often extended to allow this type of communication directly.

3.2 Parallel Programming

BSP and the other parallel models reviewed in the last section provide tools for reasoning abstractly over parallel computers and computations. To actually implement parallel programs, we need programming libraries and languages with primitives for controlling the parallel execution. This thesis presents methods for analyzing programs using BSPlib, which is such a library for BSP programming.

It is more rule than exception that formal methods for scalable parallel computing applies directly on a specific language or library rather than on the underlying abstract model. Hence, to better understand the context of BSPlib and the formal methods presented in Section 3.3, we first review some notable languages and libraries for scalable parallel programming, without limiting us to those based on the BSP model. We include domain specific languages for parallel programming, as well as libraries extending general purpose languages.

3.2.1 Other than BSP

MPI

The Message Passing Interface (MPI), is a widely used library standard for writing scalable and portable message passing programs in distributed memory. Implementations exists for C, C++ and Fortran, with bindings to other languages. Notable implementations include MPICH and Open MPI.

As in BSPlib, MPI programs are written in SPMD-style. Unlike BSPlib's spartan interface and rigid synchronization structure, MPI offers a wide range of different primitives supporting point-to-point communication as well as commonly used collective communication patterns such as broadcasts and reductions. Primitives typically come in different flavors, accommodating the type of synchronization and buffering required by the application. MPI supports several flexible primitives for synchronization: from point-to-point synchronization and synchronization involving dynamically defined process sub-groups to barrier synchronization. Like BSPlib, MPI also supports Direct Remote Memory Access, allowing processes to access memory of remote processes.

While MPI is not explicitly based on any specific parallel model, it can be thought of as an implementation of a message passing processes calculus such as Communicating Sequential Processes (CSP) [102] or Calculus of Communicating Systems [142]. As an aside, the language occam [119] was conceived to explicitly

```
#pragma omp parallel for reduction (+:psum)
for (i = 0; i < N; i++) {
        psum = psum + a[i];
}
```

Figure 3.1 – *An example where OpenMP annotations are used to parallelize a program that adds up all elements of the array* `a` *in the variable* `psum`*. The annotation specifies to execute this* `for` *loop in parallel. Loop iterations are split up between threads. The* `reduction (+:sum)` *annotation specifies that each thread shall have a private copy of the* `psum` *variable, and that the value of* `psum` *after the threads have joined at end of the loop should be the sum of each thread's copy of* `psum`*.*

implement CSP. MPI can also be used to write BSP programs by restricting the type of communication and synchronization used.

The portability and flexibility of MPI is key to its success. It is commonly used for implementing high-level parallel programming libraries and languages. This is the case for BSPonMPI[2], Multi-ML [11], BSML [18] and many others. However, this same flexibility and the size of the interface coupled with informal semantics (with subtle corner cases) renders MPI programs error-prone [129].

OpenMP

OpenMP is a library and annotation language for multi-threaded, shared memory parallelism. Compared to MPI or BSPlib, OpenMP offers a more fine-grained, implicit and incremental approach to parallelism. Whereas a MPI or BSPlib program can be seen as the parallel composition of the same program, the execution of an OpenMP program is more similar to a sequential program, that occasionally splits up in several threads, does parallel computing and then joins and continues sequential computing. This forking and joining is controlled by annotations. OpenMP can be seen as more structured than BSPlib in the sense that the parallel sections are more clearly delineated. However, whereas the superstep structure of BSP forbids one process from influencing another during the local computation phase, concurrent accesses to shared memory in OpenMP can lead to a variety of different behaviors, impeding program comprehension.

To execute the loop with an OpenMP annotation in Figure 3.1, execution forks into a number of threads. Each thread executes a subset of the loop's iterations. When all iterations have been executed, execution joins into one thread and sequential computation continues.

[2] `http://github.com/wijnand-suijlen/bsponmpi`

In theory, the OpenMP approach allows a programmer to parallelize an existing sequential application incrementally and without having to think about how to share work and distribute data between processes. In practice, the speedup obtained depends both on the architecture and the OpenMP implementation and is thus of limited portability. And without carefully considering the dependencies between loop iterations, subtle data races may be introduced such that the semantics of the parallelized programs is no longer the same as the semantics of the sequential program.

One specific use case of OpenMP is in conjunction with MPI: MPI handles multi-node parallelism, whereas OpenMP handles multi-core parallelism [183].

Parallel Skeletons

Compared to MPI or OpenMP, parallel skeletons [87] offer high-level building blocks for constructing parallel programs.

Parallel skeletons were introduced by Cole [46], to *simplify* implementation of parallel programs and to enable *portable performance*. Inspired by the observation that many scalable parallel programs repeat common patterns, Cole proposed to extract these patterns in the form of higher-order functions, referred to as skeletons. The parallel programmer can then construct their program by composing such skeletons, simplifying implementation by reducing the amount of boilerplate that must be written and verified. By giving skeletons optimized implementations for different parallel architectures, performance portability is ensured.

Two main categories of parallel skeletons are data- and task-parallel skeletons. Close to the Bird–Meertens formalism, data-parallel skeletons implement data transformations and are close to higher-order functions familiar from functional languages, such as *map*, *scan* and *fold*. Task-parallel skeletons implement coordination patterns common in task-based parallelism. For instance, the *pipeline* skeleton can be used to implement computations that are a composition of operations $t_1 \circ \ldots \circ t_n$ such that each t_i is a task (or process) that executes the corresponding operation.

A large variety of skeletons programming models exists, both in the form of dedicated languages and as libraries. We highlight some examples, but refer the reader to a survey [85] for more complete information. An early example of a dedicated language is P3L [155] that features parallel skeletons as first-class language constructs which are combined by sequential skeletons for iteration and composition. Another two-layer approach is taken by the C++ pro-

gramming library Muesli [42]. Using Muesli, a program is constructed by nesting data-parallel skeletons in task-parallel skeletons. Muesli supports heterogeneous architectures by a hybrid MPI/OpenMP implementation. Finally, we mention Skepu 2 [67], another C++ library implementing data-parallel skeletons. It also supports heterogeneous architectures, but is specialized for multi-core and multi-GPU architectures.

3.2.2 BSP

BSP Implementations and Libraries for Imperative Languages

We here discuss libraries for imperative BSP programming preceding BSPlib, libraries that have extended it as well as libraries outside the BSPlib family.

The direct ancestors of BSPlib are Green BSP [88] and Oxford BSP Library [141]. The former introduced Bulk Synchronous Message Passing and the latter DRMA. The first BSPlib implementation was the Oxford BSP Toolset [100]. In addition to ports to a variety of platforms (reviewed in Section 2.3.6), later libraries have extended BSPlib with new functionality.

The Paderborn University BSP library (PUB) for C [27] implements the BSPlib interface, but with several extensions. First, PUB includes high-level collective communications. PUB also provides an optimized barrier synchronization called "oblivious synchronization" that can be used when the number of messages that will be received in that superstep is known by each process. Finally, PUB implements subset synchronization, simplifying the implementation of divide-and-conquer algorithms, and improving performance in some cases.

MulticoreBSP [208] also extends the BSPlib interface with primitives for Multi-BSP programming. Its Java implementation provides container classes for distributed data objects that provides a similar function as BSPlib registrations, but with a simpler API.

Lightweight Parallel Foundations [183] provides a BSPlib compatible interface, in addition to a more general interface. Many of the limitations of BSPlib mentioned in Section 2.3.7 are overcome by this library, as described in that section.

Outside BSPlib descendants, a recent addition to the BSP family is Pregel [135]: a C++ library for writing graph computations following the BSP model. A Pregel program takes as input a directed graph, and the program defines a computation that is carried out in parallel for each vertex of the graph. Computation synchronizes with barriers, and vertices communicate amongst

each other by message passing. A vertex becomes inactive when it locally votes to halt computation. It can become re-activated by an incoming message. The global computation is terminated when all vertices are inactive. Pregel is a specialization of BSP, where the network topology is an input parameter of the programs. The focus is on vertices, which are implicitly distributed over processes.

BSP Implementations and Libraries for Functional Languages

BSMLlib [96] is a OCaml library that partially implements the BSML language based on the BSλ-formalism [133].

Programs are constructed as an explicit composition of local computation operating over parallel vectors and communication phases, with the latter implicitly inducing barrier synchronizations. The parallel structure of the program is thus explicit, which is not the case in SPMD programs using BSPlib or similar libraries. BSλ does not allow the nesting of parallel values, effectively ruling out dynamic process creation. BSMLlib, does not statically enforce this restriction. This model-implementation mismatch has been remedied by later authors using static analysis (see Section 3.3.3).

BSP Skeletons

Orléans Skeleton Library (OSL) [114] extends the idea of providing commonly used programming parallel primitives, as PUB does, by implementing a set bulk-synchronous algorithmic skeletons. OSL is implemented using C++ and MPI [139] and lets programmers encode data-distributions using higher-order functions, and defines common operations from functional programming such as *map*, *reduce* and *zip* over the distributed data, in addition to commonly used communication patterns.

3.3 Formal Methods for Scalable Parallel Programming

Formal methods is the application of rigorous methods based on mathematical tools for the specification, development and verification of computer software and hardware. In this main section of this chapter, we review such research into methods for scalable parallel computer programs in particular.

The review is first split in two parts: a general overview of available methods focusing on deductive verification and model checking for scalable parallel programs. This part is followed by a focus on formal methods based on static analysis for verification of scalable parallel programs. In both parts, we consider how the methods have been adapted to scalable programming models in general and BSP in particular.

3.3.1 Deductive Verification

Verifying that a program fulfills its specification by manually providing a mathematical proof to this effect has been an important occupation of computer scientists and programmers since the inception of computing [188]. This is often referred to as *deductive verification*.

Such proofs can be carried out directly on the formal semantics of a program [203], but is onerous. Hoare logic [101] offers an approach where the task is decomposed by the structure of the program. Proving the program is reduced to annotating the program with the appropriate specification in terms of a pre-condition and post-condition and *invariants* to looping constructs: properties that must hold at the beginning and end of each loop iteration.

A verification condition generator [68, 53] turns the thus annotated program in to a set of sub-properties to be verified. Tools for automated reasoning, such as SAT [144] and SMT [60] solvers, greatly reduce the charge of the verifier by automatically proving some sub-properties. Nevertheless, verification condition that are out of reach for automatic tools must be discharged by hand or with proof assistants such as Coq [21].

With the advent of multi-processing, various authors proposed extensions to the above procedure adapted for concurrent programs [153, 15, 70, 123, 125, 45]. These systems treat models with fine-grained interleavings concurrency and shared memory. On the other hand, scalable parallel programs, as we are concerned with in this study, are characterized by coarse-grained parallelism and distributed memory.

One would think that these characteristics of scalable parallel programs would simplify deductive verification and encourage the development of compositional proof systems with tool support for these languages. The above proof systems often decompose the task of verifying a concurrent program into establishing an invariant on how processes may *interfere* with each other, and then prove the desired specification under that interference. Naturally, coarse-grained

distributed memory parallelism limits interference and simplifies the first task. But as we shall see in the following two sections, there have been comparatively little work in this direction. Indeed, as far as we have seen, there are currently no tools that permits the deductive verification of applications written in MPI or BSPlib, with some exceptions discussed below [74].

In the first of the following two sub-sections we review deductive verification for scalable parallel programs in general, followed by a focus on the deductive verification of Bulk Synchronous Parallel programs.

Deductive Verification Outside of BSP

To the best of our knowledge, there are no existing tools that allow deductive verification of *unstructured* programming models for scalable parallelism such as C or Fortran with MPI in the same way that tools like Frama-C [53] permit the verification of sequential C and VCC [45] permits for concurrent C.

However, some foundations have been laid. The voluminous (and notoriously subtle) MPI semantics has been partially formalized in the specification language TLA+ [129]. A front end allows the extraction of C snippets using MPI primitives to TLA+, which can be deductive verified [38]. In parallel, the Remote Memory Access API of MPI-3 is being formalized [103].

Simpler, more compositional approaches have been proposed for more structured parallel programming models (where the parallel program is close to a sequential program solving the same problem) than MPI such as subsets of OpenMP [58]. In this framework, proving the sequential code correct suffices to prove that the parallelization is correct. Another approach is given by Couturier et al. [50]. They transform a sequential to a parallel program such that the parallel program preserves the semantics of the sequential program. This is done by specifying and proving the sequential code and a *glue*-code, that combines the result of sequential computations from each process and thus proves the complete parallel program.

We now consider the application of deductive verification to proving specific properties instead of verifying arbitrary specifications. One approach is based on treating the program like a sequential one, by axiomatizing, simulating or over-approximating the influence of other processes. For instance, verifying that the program does not write any resource that may be accessed by the other processes proves race freedom. This approach has seen promising results for deductively verifying data race absence and correct synchronization in GPU kernels [22, 19].

Work in progress aims to extend this method to MPI programs using Frama-C [214].

Finally, there is work for verifying deductively that MPI programs correctly implement *protocols* [166]. Protocols give the overall communication pattern of the application, and are specified using multi-party session types [150]. Such programs are deadlock-free and enjoy communication safety, but give no other guarantees on functional correctness.

Deductive Verification for BSP

Undoubtedly thanks to its highly structured nature, BSP has a relatively rich connections with formal methods compared to other scalable parallel programming models and methods. We first discuss the various formal semantics that have been developed for BSP, then the existing work on the deductive verification of BSP programs that has been built upon these theoretical underpinnings.

Semantics of BSP Programs A number of formal semantics for functional and imperative BSP languages has been proposed: Loulergue formalizes BSML [133], Tesson et al. formalize BSPlib [184] and Gava et al. formalize Paderborn's BSPlib [80].

A number of authors proposes axiomatic and algebraic semantics of BSP programs. The first ones are Jifeng et al. [117]. They also give algebraic laws for transformation and derivation of BSP programs. Chen et al. [40] present LOGS, an algebraic framework to reason about BSP programs and refinements [117, 39]. Stewart et al. [179] expose the symmetry of a BSP computation by giving it two semantics: either as a sequential composition of supersteps, or as a parallel composition of sequential processes. This duality was named "`SEQ` of `PAR`" versus "`PAR` of `SEQ`" by Bougé [29], who explored it in the context of data-parallel programs.

Deductive Verification of BSP Programs With the exception of BSP-Why [74] discussed below, there are no tools for the deductive verification of imperative BSP programs. On the other hand, the structured and composable nature of the functional BSML language has enabled a number of works on correct-by-construction BSML programs [66], proof-and-extract methodology [79], and verified BSP skeletons [134, 31, 84, 185].

Fortin et al. [74] extends the WhyML-language [68], with BSP constructs to obtain a verification condition generator for imperative BSP programs. The BSP

constructs are translated to regular WhyML by *sequentialization* of the parallel program, a semantics-preserving transformation certified in Coq. It is unclear whether proving BSP programs is simpler with BSP-Why than with his earlier Coq embedding [80]. The author provides no comparison between the two approaches.

A large number of proof obligations are generated by the sequentialization. Many of them can be dismissed by automated SMT solvers. But, some big and unwieldy proof obligations due to the translation must be dismissed by the user. Because of the translation step, the link is lost between the input program and the final proof obligations, and it might be difficult for the user to understand what they are proving and why it is necessary.

In conclusion, whereas tools are widely available and being applied to deductively verify sequential and even concurrent programs, the same cannot be said for scalable parallel applications, with functional BSP programs as a notable exception. While a rich theoretical groundwork in the form of formal semantics has been laid, and while some initial work exists towards the practical deductive verification of BSP and MPI programs [74, 166], it is not yet mature for use outside academic contexts. This is surprising, since highly structured and coarse-grained nature of these applications should simplify their verification compared to concurrent applications, as argued earlier.

Consequently, we propose as a future research direction the development of new methods enabling the verification of scalable parallel programs that combine the wealth of work on verifying sequential programs and the theoretical ground work for scalable parallelism, to exploit the characteristics of scalable parallel language to enable their verification.

3.3.2 Model Checking

Model checking consists of creating a logical model of the system being tested using some formalism, usually Kripke structures, as well as a specification, usually using temporal logic [156], and then verifying that each reachable state in the model conforms to the specification [43].

Model Checking for Scalable Parallelism

Naively model checking the parallel execution of a system with p processes can be done by modeling the set of reachable states by the cartesian product of the p process models and exploring all possible interleaving executions.

The drawbacks of this approach are immediate. First, the model checking will be bound in the number of processes: we can check the model of a program for a given number of processes, but do not know how to generalize when scaling the program to another number of processes.

Second, the number of states and interleavings to explore grows exponentially with the number of processes p: an unacceptable limit when realistic scalable parallel applications use hundreds or even thousands of processes. This is known as the "state explosion" problem.

Despite these drawbacks, model checking promises a "push-button solution" to verification: unlike deductive verification, the verifier does not need to do any work besides specifying the desired behavior of the program. Many authors have applied model checking to scalable parallel languages, targeting MPI in particular. In addition to adapting existing model checkers for sequential code to parallel code [172, 91], specialized model checkers have been developed [91, 215, 211, 174, 129]

Authors have attempted to tackle the state explosion by applying clever reductions [173, 171, 170], by considering limited subsets of scalable parallel programs [108], or by combining dynamic methods with model checking. In this hybrid approach, an instrumented program is executed, its execution trace is collected and then all possible variations of that trace are verified [146, 198, 190, 189, 168].

In addition to verifying assertions and general specifications, model checking has been used to verify specific properties. Examples include detecting zero buffer incompatibilities for single path MPI programs [72], deadlocks [198] and irrelevant barriers [168]. Finally, model checking has been used to verify the (bounded) equivalence of sequential and parallel programs [174] and to verify whether one program calculates the derivative of another program [109].

Model Checking for BSP To the best of our knowledge, there are no model checkers for BSP programs. However, the many model checkers that exist for MPI could be used to check programs using the BSP subset of MPI. The downside of this approach is that the superstep structure of BSP programs is not used to avoid exploring equivalent interleavings.

Naively model checking a bulk synchronous MPI program would explore all possible interleavings of the local computing phases. To no avail, since BSP semantics specifies that the processes do not influence each other in this phase, and thus checking a single interleaving suffices.

As our review above shows, one trend in model checking parallel programs is special purpose checkers, that concentrate on detecting specific errors or properties (data race, deadlocks or irrelevant barriers) using problem-specific reductions. Another trend is exploiting structural properties of special classes of parallel programs that reduce the number of interleavings that must be explored.

As in the case of deductive verification, it is surprising that no authors have yet to exploit the structural simplicity of BSP programs to implement efficient model checkers. Along with intelligent problem specific reductions, this presents a promising venue for future advances in the state of model checking realistic scalable parallel applications.

Since interleavings pose no problem to verifying BSP programs, effort can be redirected to reducing the state space of the model. State compression, symbolical state representations [215] and the use of static analysis to detect single-valued variables [5] are promising directions.

3.3.3 Static Analysis

The goal in deductive verification and model checking is often establishing full correctness. Deductive verification requires significant manual intervention of the verifier and model checking is inherently bounded by the state explosion problem, which is exacerbated by parallelism. On the other hand, the aim of *static analysis* [151], introduced in Section 2.4, is to automatically discover program *invariants*: properties that hold for any execution. Furthermore, to do so with a run time that scales tractably with the size of the analyzed program. The typical approach of static analysis is finding appropriate abstractions of both states and instructions. Besides verifying safety properties such as the absence of data races, static analysis is also used for optimization [17], compilation [4] and security [16].

The main goal of this section is review static analyses for the verification of scalable parallel programs. We will also point to static analyses of such programs with other aims than verification when relevant. As many analyses for more fine-grained parallelism could be transferred to scalable parallel programs, we shall also on occasion point to such work.

This section proceeds as follows. We first review general frameworks for static analysis: such frameworks do not target specific properties but instead aim for generality and extensibility. We then proceed by considering specialized analyses based on the type of properties they target. First, analyses that aim to un-

derstand the parallel *structure* of programs are reviewed. Next, we review data races analyses that often rely upon such structural analyses.

Whereas structural and data race analyses aim to answer how parallel programs execute and whether they do so correctly, the goal of cost analyses is to deduce at what cost (in run time or other resources) a parallel program executes. The section continues by reviewing such analyses.

Finally, we review static analyses that are specific for BSP programs.

Static Analysis Frameworks for Parallel Programs

Data-flow analysis and more generally abstract interpretation are general frameworks for static analysis. A natural idea is to extend such frameworks to incorporate parallelism in such a way that more specialized analyses (such as the sign analysis mentioned above) become parallelism-aware instantiations of the framework.

One idea for data-flow analysis is constructing parallel control flow graphs. The parallel control flow graph is formed by decorating the sequential control flow graph of a program with edges from communication sources to communication destinations, so that data-flow facts pertaining to a communicated value can be transferred from one program point to another [169, 180, 32]. This approach requires an initial analysis to discover the nodes that should be connected by such communication edges. This type of parallel data-flow analysis has been used for constant propagation [180] and the detection of communication patterns [32].

Data-flow analysis is an instance of the general framework of abstract interpretation. Some authors propose parallelism-aware abstract interpretation frameworks, such as Botbol's transducer-based approach for MPI [28]. Botbol's approach is bounded in the number of processes but otherwise makes few assumptions on the parallel structure of the analyzed program.

A possible future direction of research could be general analysis frameworks that are process-identifier-sensitive: either by applying the independent attribute method to combine an abstraction on the process identifier with an application-specific abstraction, or by relational method so that the application-specific abstraction depends on the process identifier abstraction. This former approach has been explored for MPI [32] and GPU programs [13].

Thread-modular verification[69], which has been successfully applied to fine-grained forms of parallelism [143], is an approach where each process is analyzed in isolation and an over-approximation of the interference of this process

on the other processes is calculated. Then the next process is analyzed with the interference of the first process taken into account. The analysis continues in this way until a fixpoint is obtained. This approach has been applied to synchronous message passing programs [140] and its extension to scalable parallel programming models could be a future research direction.

Structural Analysis

This section regroups a set of different analyses whose aim is to infer facts regarding the *parallel structure* of a parallel program. More specifically, these analyses aim to answer questions such as:

- Is the program well-synchronized?
- Are there any deadlocks?
- Which instructions of the program may be executed in parallel (i.e. "at the same time")?

We also consider analyses that aim to find relationships between state of all processes. For instance, are there any variables that always have the same value in all processes? Lastly, there are a set of analyses that aim to infer the *communication topology* of the analyzed program. That is, which processes will communicate with each other, how much and at which point? In MPI, which are the send and receive-instructions that match? These analyses have a wide range of applicability, from program comprehension and optimization to verification. Furthermore, as we shall see in the section on data race analysis (Section 3.3.3), they are often used as a building block for other analyses.

Synchronization Analysis A first, fundamental issue is to understand the interaction of synchronization constructs in a parallel program. Examples of such constructs are barrier synchronization in BSPlib, MPI and OpenMP, but also point-to-point synchronization such as send and receive in MPI. A number of analyses have been proposed that match calls to `send` with calls to `recv` [169, 180, 32].

Barrier synchronization is a coarse-grained form of synchronization that require the participation of all processes. All processes must execute the *same number* of barriers to ensure correct synchronization. Jeremiahs et al. [116] present an initial work for verifying correct barrier synchronization for SPMD programs, based on named barriers and control flow graph reachability. Aiken and Gay

present the seminal Barrier Inference analysis [5] for verifying barrier synchronization. It is formulated as a type and effect system based on the notions of *structurally correct* programs and *single-valued* variables. Intuitively, a single-valued variable has the same value in all processes at the same time. If the choice to synchronize depends exclusively on such variables, then the same choice is made by all processes. Intuitively, a program is structurally correct when each sub-program (e.g. branches in conditionals) executes the same number of barriers.

The work of Aiken and Gay is adapted by Zhang et al. [209] to verify barrier synchronization in MPI programs by analyzing their program dependency graphs. Additionally, they compute the barriers that match, i.e. that will be reached at the same time. Barrier inference has also been implemented for OpenSHMEM [158] and Titanium [121]. Hybrid analyses where barriers that cannot be verified statically are instrumented and verified dynamically have also been proposed [122, 165].

Throughout this thesis, we study textually aligned programs, and static analysis under the assumption of textual alignment. These programs form a subset of structurally correct programs. Intuitively, in a program with textually aligned synchronization, each barrier synchronization result from the same textual instance of the synchronization primitive. Furthermore, should that instance be in a loop, then all processes must be in the same iteration. This set of programs has been formally defined by Dabrowski [56], by characterizing their execution paths with an instrumented semantics. In a later work, he presents a denotational semantics for such programs [55].

The idea of single-valued variables, which has its roots in the binding-time analysis used for partial evaluation [17], has found other applications in the analysis of parallel programs. The two-process reduction used in GPUVerify [22] simulates the execution of two processes by, amongst other things, duplicating all local variables. Single-value analysis is used to reduce the number of duplicated variables. Laguna et al. [124] detect variables that depend on the process identifier (i.e., that are not single-valued) as potential causes of "scale-dependent" bugs. In [3], single-valued variables serve as process-independent iteration counters that are used to produce debugging information. A complementary objective to single-value analysis is to detect exactly how non single-valued variables depend on the process identifier [13, 32].

Later authors have considered less coarse synchronization mechanisms. Clocks in X10 [167] generalize barriers in that the processes that adhere to each

clock, and must participate in its synchronization, can be modified dynamically. An analysis based on the polyhedral model for verifying the correct usage of X10 clocks to avoid data races is given by Yuki et al. [205, 206].

Both barriers and X10 clocks are generalized by "phasers", implemented in Habanero-Java [35]. Ganjei et al. [78] contribute a reachability analysis for such programs.

May-Happen-in-Parallel Analysis A second, fundamental task is may-happen-in-parallel analysis (MHP). Which are the statements that may execute in parallel, and for which statements are there synchronization mechanisms precluding this possibility?

A general approach for SPMD programs with barrier synchronization is given by Jeremiassen et al. [116]. Similar approaches have been proposed for OpenMP [130, 210, 37] and X10 [2, 126, 205].

Typically, MHP analysis is based on identifying synchronization structures in the control flow graph of the analyzed program, and by establishing a partial execution order between statements [116, 130, 210, 2]. In models of fine-grained parallelism such as OpenMP and X10, a strain of recent work relies on the polyhedral model to obtain a finer MHP-relation [205, 37].

An important client of MHP analyses is data race analyses. Static data race analyses are reviewed more in detail in the next section. When verifying data race freedom deductively, an initial MHP analysis can be used to reduce proof obligations [187].

Communication Analysis Inferring the communication topology of a parallel application is a third important aspect with use cases ranging from verification and optimization to program comprehension.

One type of communication analysis aims to reconstruct high-level communication patterns (such as *all-to-one*, *shift*, etc.) from point-to-point interactions. Di Martino detects communication patterns using the polyhedral model [62], but data-flow based approaches have also been suggested [32, 118]. Use cases include optimization, by replacing "hand-written" communication patterns with optimized library routines [160], by optimizing process placement [138] or optimizing checkpointing [199].

The above analyses detect high-level collective patterns that are common in scalable parallel programming. Analyses based on *session types* have been proposed to verify intricate protocols. An example might include a program where

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
        psum = psum + a[i];
}
```

Figure 3.2 – *An example of a data race in a OpenMP program: each iteration of the loop accesses the shared variable* `psum` *but nothing guarantees the atomicity of this assignment. It is possible that process A writes* `psum` *just after another process B had read it but before B writes the sum back to* `psum`*. The write of B cancels the write of A, and the final result is invalid.*

```
bsp_put(0, &x, &y, 0, sizeof(y));
```

Figure 3.3 – *An example of a concurrent write in a BSPlib program: if at least two processes execute this instruction and then both processes will write to the same memory area in process* 0*. The value that is ultimately written is implementation dependent.*

a token is passed between processes in ring configuration. Session based type systems have been proposed for C [149] and for Java [148].

New Research Directions Future work on structural analysis of scalable parallel programs could involve handling more advanced forms of synchronization such as subset synchronization [26]. However, extending synchronization analysis to programs that are not structurally correct (in the sense of Aiken and Gay) seems unnecessary: Zhang [209] finds the vast majority of the MPI programs they evaluate to be textually aligned, a subset of the structurally correct programs. Inspired by the finer analysis of branching conditions in MHP analysis of OpenMP and X10 programs [2] we imagine that MHP can be combined with analyses for active set analysis [13] to obtain finer MHP relations for SPMD programs.

Data Race Analysis

A data race occurs when two processes access the same memory area in a conflictual manner (at least one of the accesses is a write) and there is no synchronization between the two accesses. Data races occur in both distributed and shared memory parallelism.

The OpenMP program in Figure 3.2 contains a data race. The program is identical to the one in Figure 3.1, but the annotation `reduction (+:psum)` is erroneously omitted. Now each process will read and write concurrently from the same shared memory variable `psum`, and the final result in this variable will depend on the order of their interleavings.

The BSPlib program in Figure 3.3 contains a concurrent write if executed by at

least two processes in the same superstep.[3] Concurrent writes are similar to data races in that they are caused by two processes writing the same resource. They differ in that if present, they occur independently of interleavings. In theory, non-determinism can be avoided by imposing an ordering on the execution of communication request (e.g. communications requests are executed in order of increasing process identifier of issuer.) In practice, BSPlib imposes no such order. The final result written thus depends on the BSPlib implementation.

Verifying the absence of data races and concurrent writes is important since data races are often indicative of programming errors, but more importantly they may be a source of *non-determinism*. A program is non-deterministic if the same input may give rise to different outputs. As in the Figure 3.2 above, this may happen due to different interleavings of the same program exposing a data race. Unintended non-determinism is undesirable if some of the possible executions are erroneous. Non-determinism may also be hard to track down, giving rise to so-called *heisenbugs* that are visible in some executions but not in others.

Hence a large body of work on various methods for verifying the absence of data races. Detecting data races can be decomposed into two tasks: (1) deducing which instructions may happen in parallel (may-happen-in-parallel analysis) and (2) deducing which memory areas will be accessed by those instructions. If all instructions that may happen in parallel only access distinct memory areas or only access the same memory area non-conflictually (i.e. read-read) then the program is data race free. Hence data race analyses are important clients of MHP analysis [205, 206, 37].

More generally, data race analysis can be seen as a specific form of communication analyses. However, it is less interested in the overall communication pattern, and more interested in ruling out a specific kind potentially erroneous behavior.

Unlike concurrent programs, for which type and effect systems have successfully been used to rule out data race errors [145], scalable parallel programs have not yet seen many static data race analyses.

An exception can be made for finer grained models such as OpenMP [37] and X10 programs [205, 2, 206] for which the polyhedral model has been deployed to model the access pattern of the programs and then rule out data races. Key to permitting this kind of analysis is the highly structured nature of these languages compared to other programming models.

[3] Additionally, the address of x in the two processes must be associated with the same memory area in process 0 by a BSPlib registration, as explained in Section 2.3.5.

We note the absence of static tools for detecting data races in coarse grained parallel models. Presumably, this is due to the prevalence of send/receive message passing and the use of collectives in MPI, which precludes the existence of data races. However, later versions of MPI include one-sided RMA operations whose usage may cause races [103]. To the best of our knowledge, no static analysis has been proposed for their detection. For instance, future work for detecting races for scalable parallel programs might include combining existing structural analyses with the type of numerical analyses that has successfully analyzed OpenMP and X10 programs. BSPlib, where remote memory access is frequently used, would be a prime target for this kind of analysis. The structured nature of BSPlib would simplify MHP analysis that could be coupled to numerical analyses for the detection of conflicting accesses.

Rinard [163] suggested that the data race analysis would use methods originally developed for automatic parallelization, and this has been proven true by the widespread use of the polyhedral model. More inspiration might be found from the community of parallelizing compilation. High level patterns that are guaranteed to be data race free can be detected by detecting higher level communication patterns [62, 1], or by reversing methods for automatic parallelization.

Cost Analysis

The goal of cost analysis is to deduce, for a given program and input, a statistic on the *cost* of executing that program. The exact *cost* of interest depends on the context, but typically one is interested in measures on resources such as run time, memory consumption or number of communicated bytes. Typically one is interested in the upper bound of this cost, to allow provisioning for worst-case scenarios, but deducing lower bounds and averages also have uses.

Cost analysis is often separated into Cost Analysis and Worst-Case Execution Time (WCET). This division is quite arbitrary: the difference lies rather in community and domain of application than the actual methods used. The former aims to find asymptotic bounds in a more abstract setting, while the latter has a more concrete view of hardware, taking into account architecture-specific features such as caches and pipelines.

Cost analysis for sequential programs is a well-studied domain [201] with extensions for concurrent programs [154, 92]. In this section we will give some pointers on cost analysis adapted to scalable parallelism.

To the best of our knowledge, there are few published works contributing cost analysis for coarse-grained parallel programs outside the world of BSP (with

some recent exceptions [77]). A possible reason is the emphasis placed in the BSP community on cost and the fact BSP has a cost model, a pre-requisite for formal reasoning on cost.

Outside the world of scalable parallelism, we find cost analyses extended to parallel, functional programs: Zimmermann [213] uses classic cost analysis for treating functional programs with parallelism restricted to divide-and-conquer algorithms. Resource Aware ML [104] implements a type-based approach to amortized cost analysis for ML with parallel extensions [105]. We also find extensions of classic cost analysis to task-based distributed programs with dynamic spawning [8, 9].

More applicable to scalable parallelism, we also see methods for analyzing communication loads: the polyhedral model has been used to automatically evaluate the communication volumes produced by loops and evaluating their different transformations by this measure [30, 44].

In summary, as algorithmic complexity analysis of sequential programs is still quite immature, the same is even more true of parallel programs. The issues that render cost analysis of sequential programs difficult are exacerbated by parallelism. However, we still foresee the adaption of techniques that have been applied in sequential contexts, such as amortized cost analysis [104], to parallel languages. We also think that the highly structured nature of SPMD programs and BSP programs in particular could alleviate this difficulty by reducing the amount of parallelism that must be considered [131].

Static Analysis of BSP Programs

Amongst the types of static analyses that are reviewed above, we have found little work that applies explicitly to BSP, and in particular, no work at all that targets BSPlib programs. We discuss the exceptions in this section on static analyses of other types of BSP programs.

In terms of structural analysis, a type system has been developed for BSML [81] that precludes invalid nesting of parallel values. Recall from Section 3.2.2 that the core data structure of BSML is the parallel vector, a parametric type. Nested parallel vectors has no meaning (unless each node is itself a parallel machine, a use case that is not considered) and should not be formed. However, BSML is a OCaml library, and the type system of OCaml does not enable restrictions on type parameters. Hence implementing the restriction on a library level is not feasible, motivating their extended type system. Multi-ML, the Multi-BSP (Section 3.1.2) extension of BSML, imposes the same restriction on parallel

vectors. Furthermore, the architecture programmed by Multi-ML imposes additional restrictions, such as forbidding the reference of data defined on a lower level, that are enforced by Multi-ML's type system [10].

Hayashi proposes a cost analysis for *shapely skeletal* BSP programs [98]. Shapely programs are written so that the size of data structures is always known statically. Skeletons are ready-made parallel constructs that the programmer uses as building blocks for their program. The cost function of each skeleton and the input data size are *a priori* known and so the matter of computing the cost function for a program is obtained by statically composing the cost functions of each skeleton. Recently, a cost semantics of Multi-ML has been proposed [12] as an initial step towards automatic cost analysis of Multi-ML programs.

3.3.4 Other Formal Methods

To reduce the scope of this survey to a tractable size, we have not surveyed runtime and dynamic analysis [197], hybrid analysis [165], *a posteriori* trace analysis [159], program refinement [212], symbolic execution [211], nor testing [181] for scalable parallel programs.

3.4 Discussion

Our first conclusion of this review is that compared to the state of sequential or concurrent programs, there is a lack of formal methods tools for developing scalable parallel programs. In particular, we note the lack of automatic tools to this effect. We argue that such tools are necessary to successfully develop software for forthcoming large scale parallel computers since the programmers will most probably not be verification experts [86].

We have observed that much of scalable computing performed in imperative languages that permit a wide range of synchronization patterns, and that consequently, a big effort of existing tools is spent in dealing with unstructured and chaotic parallelism. However much of scalable parallel code is actually written in a highly structured fragment of their implementation language: as we have seen in the section on static analysis of parallel structure (Section 3.3.3), most SPMD programs are *structurally correct* as per Aiken and Gay [5], or even *textually aligned* [56, 209]. We have also seen that more structured parallelism frameworks are more amenable to formal methods. Consider for instance the state of deductive verification of MPI vs. OpenMP vs. BSML programs. The more structured

the language, the more formal methods have been developed. This can also be observed for static race or static cost analysis. This hidden structure in scalable parallel programs is a property that can and should be exploited to simplify and promote the development of formal methods for such programs.

Specifically for BSP and BSPlib, we note that there is no formal connection between the high-level properties promised by the BSP model (correctness and predictable performance) and BSPlib programs. We argue that formal methods can serve as a bridge between general purpose languages with libraries, such as BSPlib, and the high-level model the libraries implement, such as BSP. Formal methods can and should be used to ensure that the low-level program fulfills the promises of the high-level model.

Replicated Synchronization 4

Contents

4.1 Synchronization Errors in BSPlib Programs . . . 83
4.1.1 Textual Alignment and Replicated Synchronization . . . 84
4.2 The **BSPlite** Language . . . 85
4.2.1 Operational Semantics . . . 86
4.2.2 Denotational Semantics . . . 87
4.3 Static Approximation of Textual Alignment . . . 92
4.3.1 Pid-Independence Data-Flow Analysis . . . 93
4.3.2 Replicated Synchronization Analysis . . . 101
4.4 Implementation . . . 102
4.4.1 Adapting the Analysis to Frama-C . . . 103
4.4.2 Edge-by-Edge Flow Fact Updates . . . 103
4.4.3 Frama-C Control Flow Graph . . . 105
4.4.4 Implementing Interprocedural Analysis Using Small Assumption Sets . . . 111
4.5 Evaluation . . . 114
4.6 Related Work . . . 116
4.7 Concluding Remarks . . . 117

This chapter is extracted from the author's article [111].

Synchronization is a potential source of errors in imperative BSP programs. BSPlib programs interleave the code that handles local computation and code that handles synchronization. As a consequence, incorrect programs are easy to write but hard to debug.

It has already been noted that quality parallel code has strict synchronization patterns that ensure correct synchronization, and there are static analyses that enforce these patterns [5, 209]. However, our review of realistic BSPlib programs suggests that the stricter convention of *textually aligned barriers* is sufficient to capture a large majority of correct programs. In programs with textually aligned barriers, each barrier is the result of a synchronization request from the same source code location in all processes. In addition to ensuring correct synchronization, such programs are conceptually simpler which help steer program design toward correctness, and ease other validation steps such as code review.

In this chapter we present a static analysis that verifies whether a program has *replicated synchronization*: a static approximation of textually aligned barriers. Therefore, the analysis reconstitutes the backbone of a BSP algorithm, which is its synchronization.

This static analysis poses higher requirements on the analyzed source code than existing analyses such as Barrier Inference [5]. For this reason, it is defined directly on the syntax of our language, assumes structured control flow and allows fewer synchronization patterns. This is intentional, since our purpose is to carve out a stricter subset of the language that complies with best practices and that ensures correctness.

The main contributions of this chapter are:

- a formalization of C with BSPlib, building upon the language **Seq** introduced in Section 2.4.1;
- a reformulation of the Barrier Inference static analysis [5] to identify programs with textually aligned barriers, which has been certified in the Coq proof assistant;
- an implementation of this analysis as a Frama-C plug-in and its evaluation on set of representative BSPlib programs.

Our formalization of C with BSPlib extends the language **Seq** presented in Section 2.4.1 with parallel primitives. Hence, the formalization does not feature functions, contains only scalar variables, and does not model pointers and nor communication. We argue these simplifications by the hypothesis that synchronization in BSPlib programs rarely depends on such features. This allows us to retain simplicity in the model.

Nonetheless, to be usable for real BSPlib programs, our implementation of the analysis must handle these language features, as they are present in C. We

do this using a mix of limitations, conservative assumptions and extensions. Notably, we reject unstructured programs. The same for programs whose synchronization depends on the contents of arrays, structures or on objects that may be modified by pointer manipulations or communication. Finally, we extend the formalization interprocedurally. We implement a standard extension based on small assumption sets [151]. We detail these extension in Section 4.4. We evaluate this implementation in Section 4.5 and confirm that these implementation decisions are not overly punitive, and thus also confirming the choice of a minimal formalization.

This chapter is organized as follows. First, we discuss synchronization errors and textual alignment in Section 4.1. We then present **BSPlite**, our modelization of C with BSP, in Section 4.2 and give its operational semantics and use the denotational semantics by Dabrowski [55] to formalize textual alignment. Section 4.3 is devoted to the replicated synchronization analysis. The implementation of the analysis and its evaluation is discussed in Section 4.4 and Section 4.5 respectively. We position our work with respect to previous work in Section 4.6. We close this chapter by discussing limitations of the analysis and future research directions in Section 4.7.

The properties of the semantics and the static analysis in this section have been verified in the Coq proof assistant [21]. A natural language version of these proofs is found in Appendix A.

4.1 Synchronization Errors in BSPlib Programs

The synchronization phase in the execution of a BSPlib program occurs when *all* the processes call `bsp_sync`. Under correct dynamic use, `bsp_sync` generates the sequence of supersteps that is the structure of a BSP computation.

Yet, it is quite easy to write an BSPlib program with incorrect synchronization. For example, consider Example 1 in Figure 4.1. This program is not well-synchronized, because only half of the processes are calling the synchronization barrier. In contrast, Examples 3 and 4 are well-synchronized, since all processes produce the same number of barriers. Even Example 2 is well-synchronized, as there is no requirement in BSPlib that all processes call `bsp_sync()` from the same program point.

BSPlib Program		WS	TAB	RS
Example (1)	`if (2 * bsp_pid() < bsp_nprocs())` `  bsp_sync();`	✗	✗	✗
Example (2)	`if (bsp_pid()) { bsp_sync(); }` `else { bsp_sync(); }`	✓	✗	✗
Example (3)	`if (bsp_pid() >= 0)` `  bsp_sync();`	✓	✓	✗
Example (4)	`for (i = 0; i < 100; i++)` `  bsp_sync();`	✓	✓	✓

Figure 4.1 – *Running examples for the Replicated Synchronization Analysis. The right part of the table notes if the program is well-synchronized (**WS**), if it has textually aligned barriers (**TAB**), and if it has replicated synchronization (**RS**).*

4.1.1 Textual Alignment and Replicated Synchronization

A sufficient condition for the absence of synchronization errors in a program is having *textually aligned barriers*. Intuitively, a program has textually aligned barriers when all processes agree on the evaluation of all guard expressions in the program that lead up to any `bsp_sync` primitive. We say that these guard expressions are *uniform*. As a consequence, either all processes synchronize or none of them do. If they synchronize, all requests to synchronize come from the same occurrence of a `bsp_sync` primitive: same occurrence both in terms of source code location, and in terms of iteration count for `bsp_sync` primitives in loops. For a formal definition of textual alignment, we refer to [56].

Examples 3 and 4 of Figure 4.1 are textually aligned. In Example 3, the condition will evaluate to `tt` for all processes so they all synchronize. All barriers in Example 4 result from the same iteration and the same program point. Note that some programs, like Example 2, do not have textually aligned barriers, but are still well-synchronized. However, from a Software Engineering best practices point of view, this type of synchronization pattern is discouraged, since the divergence of control flow between processes impedes program comprehension.

The proposed analysis of this chapter statically verifies whether a program is a member of a static under-approximation of the set of textually aligned programs, and thus is error-free. *Replicated synchronization* is a sufficient condition for textual alignment. In programs with replicated synchronization, all `bsp_sync` primitives are guarded by *pid-independent conditions*. A condition is *pid*-independent if none of the variables in the expression are data or control

dependent on the `pid` expression. By extension, we say that such variables and expression are *pid*-independent. Being *pid*-independent is sufficient for a guard condition to be uniform: except for the `pid` primitive and the variables that depend on it, all expressions evaluate to the same value over all processes. Example 4 has replicated synchronization, since the guard of the `for` loop does not depend on `pid`.

4.2 The **BSPlite** Language

As a target for the Replicated Synchronization Analysis, we formalize a small subset of C with BSPlib. This formalization, **BSPlite**, is an extension of **Seq** with concurrency primitives. Communication are not yet modeled, since the focus of this chapter is synchronization, nor are local errors such as division by zero, modeled. Expressions and instructions are defined by the following grammar:

$$
\begin{array}{lcll}
\mathbf{AExp}_p & \ni & e & ::= x \mid n \mid e + e \mid e - e \mid e \times e \\
 & & & \mid \texttt{nprocs} \mid \texttt{pid} \\
\mathbf{BExp}_p & \ni & b & ::= \texttt{true} \mid \texttt{false} \mid e < e \mid e = e \mid b \ \mathbf{or}\ b \mid b \ \mathbf{and}\ b \mid !b \\
\mathbf{Par} & \ni & s & ::= [x{:=}e]^{\ell} \mid [\texttt{skip}]^{\ell} \mid s; s \mid \texttt{if}\ [b]^{\ell}\ \texttt{then}\ s\ \texttt{else}\ s\ [\texttt{end}]^{\ell^m} \\
 & & & \mid \texttt{while}\ [b]^{\ell}\ \texttt{do}\ s\ \texttt{end} \\
 & & & \mid [\texttt{sync}]^{\ell}
\end{array}
$$

Note that the $\mathbf{AExp}_p$ extends **AExp** with the new symbolic expressions `nprocs` and `pid`. These denote the number of processes in the BSP computation and the process identifier respectively, modeling BSPlib's `bsp_nprocs` and `bsp_pid`, respectively. The set of boolean expressions contain the same constructions as in the sequential language, but we denote them $\mathbf{BExp}_p$ to remind the user that the underlying arithmetic expressions may contain the new parallel primitives.

The new instruction $[\texttt{sync}]^{\ell}$ generates a barrier synchronization when called collectively, and models BSPlib's `bsp_sync`. Variables $x \in \mathbf{Var}$ and numerals $n \in \mathbf{Nat}$ are as before.

Statements are labeled, but now with an additional label on the end of conditionals. This ensures that all statements have unique exits, simplifying the presentation of the static analysis in Section 4.3.

$$\begin{cases} \mathcal{A}[\![\cdot]\!]^i & : & \mathbf{AExp}_p \to (\mathbf{State} \to \mathbf{Nat}) \qquad i \in \mathbf{Pid} \\ \mathcal{A}[\![x]\!]^i\,\sigma & = & \sigma(x) \\ \mathcal{A}[\![n]\!]^i\,\sigma & = & n \\ \mathcal{A}[\![e_1 + e_2]\!]^i\,\sigma & = & \mathcal{A}[\![e_1]\!]^i\,\sigma + \mathcal{A}[\![e_2]\!]^i\,\sigma \\ & \vdots & \\ \mathcal{A}[\![\texttt{pid}]\!]^i\,\sigma & = & i \\ \mathcal{A}[\![\texttt{nprocs}]\!]^i\,\sigma & = & \mathbf{p} \end{cases}$$

Figure 4.2 – *Semantics of arithmetic expressions*

$$\begin{cases} \mathcal{B}[\![\cdot]\!]^i & : & \mathbf{BExp}_p \to (\mathbf{State} \to \mathbf{Bool}) \qquad i \in \mathbf{Pid} \\ \mathcal{B}[\![\texttt{true}]\!]^i\,\sigma & = & \texttt{tt} \\ \mathcal{B}[\![\texttt{false}]\!]^i\,\sigma & = & \texttt{ff} \\ \mathcal{B}[\![e_1 < e_2]\!]^i\,\sigma & = & \texttt{tt} \text{ if } \mathcal{A}[\![e_1]\!]^i\,\sigma < \mathcal{A}[\![e_2]\!]^i\,\sigma,\ \texttt{ff} \text{ oth.} \\ & \vdots & \end{cases}$$

Figure 4.3 – *Semantics of boolean expressions*

4.2.1 Operational Semantics

The operational semantics of **BSPlite** models BSPlib, but is simplified due to the exclusion of communication and local errors. The semantics is parameterized by the number $\mathbf{p} > 0$ of processors of the underlying BSP machine. The set of process identifiers is $\mathbf{Pid} = \{0, \ldots, \mathbf{p} - 1\}$.

The semantics of numerical and boolean expressions, parameterized by the local process number i, is given by $\mathcal{A}[\![\cdot]\!]^i$ and $\mathcal{B}[\![\cdot]\!]^i$, respectively. These functions extend $\mathcal{A}[\![\cdot]\!]$ and $\mathcal{B}[\![\cdot]\!]$ with the semantics of the primitives `pid` and `nprocs`, and are partially given in Figures 4.2 and 4.3. Remaining cases can be derived in the natural way.

The operational semantics for **BSPlite** programs is divided into local and global rules operating on state $\mathbf{p}$-vectors. Intuitively, the local rules, an extension of the semantics of **Seq**, compute the new state of one component in the state vector, corresponding to one processor. An optional continuation describes the next step of local computation. The global rules compute the new state of a complete state vector by applying the local rules to each component and, after synchronization, possibly performing any remaining computations by re-applying the global rules.

The semantics of local computation is given by the evaluation relation $\to^i$,

indexed by $i \in \mathbf{Pid}$:

$$\begin{array}{lcll} \rightarrow^i & : & \mathbf{Par} \times \mathbf{State} \times \mathbf{Term} \times \mathbf{State} & i \in \mathbf{Pid} \\ \mathbf{Term} & = & \{Ok\} \cup \{Wait(s) \mid s \in \mathbf{Par}\} & \\ \mathbf{State} & = & \mathbf{Var} \rightarrow \mathbf{Nat} & \end{array}$$

where **Term** is the set of termination states, with Ok denoting end of computation and $Wait(s)$ a remaining computation to execute. As in **Seq**, memory states in **State** are mappings from variables to values. We write $\langle s, \sigma \rangle \rightarrow^i \langle t, \sigma' \rangle$ for $(s, \sigma, t, \sigma) \in \rightarrow^i$. The inference rules defining this relation are given in Figure 4.4.

Now the global semantics of **BSPlite** programs is given by the global evaluation relation

$$\longrightarrow \quad : \quad \mathbf{Par^p} \times \mathbf{State^p} \times (\mathbf{State^p} \cup \{\Omega_S\})$$

that relates initial **p**-vectors of programs and states (one per process) with a final **p**-vector of states, or a synchronization error (Ω_S). The inference rules defining this relation are given in Figure 4.5, where we write $\langle S, \Sigma \rangle \longrightarrow t$ for $(S, \Sigma, t) \in \longrightarrow$. The third rule specifies that a synchronization error occurs when all processes terminate, and the termination states of at least two processes are incoherent. The goal of the static analysis in Section 4.3 is to rule out this error for a given program.

SPMD Execution

BSPlite programs are executed in SPMD-fashion: one initial program s is replicated over all processes. Furthermore, we require that the initial state is the same on all processes in the first superstep. The semantics of a **BSPlite** program s with the initial state σ is thus t if and only if

$$\langle \langle s \rangle_i, \langle \sigma \rangle_i \rangle \longrightarrow t$$

4.2.2 Denotational Semantics

We rely on the denotational semantics of Dabrowski [55] as the formal definition of textual alignment. In this section, we present this semantics that unlike the operational semantics, only gives meaning to programs with textually aligned barriers. This presentation is heavily indebted to [55]. We hide program labels in this section to improve legibility.

$$\frac{}{\langle [\texttt{skip}]^{\ell}, \sigma\rangle \rightarrow^{i} \langle Ok, \sigma\rangle} \quad [\textsc{skip}]$$

$$\frac{}{\langle [\texttt{sync}]^{\ell}, \sigma\rangle \rightarrow^{i} \langle Wait([\texttt{skip}]^{\ell}), \sigma\rangle} \quad [\textsc{sync}]$$

$$\frac{}{\langle [x{:=}e]^{\ell}, \sigma\rangle \rightarrow^{i} \langle Ok, \sigma[x \leftarrow \mathcal{A}[\![e]\!]^{i}\,\sigma]\rangle} \quad [\textsc{assign}]$$

$$\frac{\langle s_1, \sigma\rangle \rightarrow^{i} \langle Ok, \sigma''\rangle \qquad \langle s_2, \sigma''\rangle \rightarrow^{i} t}{\langle s_1; s_2, \sigma\rangle \rightarrow^{i} t} \quad [\textsc{seq-ok}]$$

$$\frac{\langle s_1, \sigma\rangle \rightarrow^{i} \langle Wait(s_1'), \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \rightarrow^{i} \langle Wait(s_1'; s_2), \sigma'\rangle} \quad [\textsc{seq-wait}]$$

$$\frac{\mathcal{B}[\![b]\!]^{i}\,\sigma = \texttt{tt} \qquad \langle s_1, \sigma\rangle \rightarrow^{i} t}{\langle \texttt{if } [b]^{\ell} \texttt{ then } s_1 \texttt{ else } s_2 \ [\texttt{end}]^{\ell^m}, \sigma\rangle \rightarrow^{i} t} \quad [\textsc{if-tt}]$$

$$\frac{\mathcal{B}[\![b]\!]^{i}\,\sigma = \texttt{ff} \qquad \langle s_2, \sigma\rangle \rightarrow^{i} t}{\langle \texttt{if } [b]^{\ell} \texttt{ then } s_1 \texttt{ else } s_2 \ [\texttt{end}]^{\ell^m}, \sigma\rangle \rightarrow^{i} t} \quad [\textsc{if-ff}]$$

$$\frac{\mathcal{B}[\![b]\!]^{i}\,\sigma = \texttt{tt} \qquad \langle s, \sigma\rangle \rightarrow^{i} \langle Ok, \sigma'\rangle \quad \langle \texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}, \sigma'\rangle \rightarrow^{i} t}{\langle \texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^{i} t} \quad [\textsc{wh-tt-ok}]$$

$$\frac{\mathcal{B}[\![b]\!]^{i}\,\sigma = \texttt{tt} \qquad \langle s, \sigma\rangle \rightarrow^{i} \langle Wait(s'), \sigma'\rangle}{\langle \texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^{i} \langle Wait(s'; \texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}), \sigma'\rangle} \quad [\textsc{wh-tt-wait}]$$

$$\frac{\mathcal{B}[\![b]\!]^{i}\,\sigma = \texttt{ff}}{\langle \texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^{i} \langle Ok, \sigma\rangle} \quad [\textsc{wh-ff}]$$

Figure 4.4 – **BSPlite** *local operational semantics*

$$\frac{\forall i \in \mathbf{Pid}, \langle S[i], \Sigma[i]\rangle \to^i \langle Ok, \Sigma'[i]\rangle}{\langle S, \Sigma\rangle \longrightarrow \Sigma'} \quad [\textsc{all-ok}]$$

$$\frac{\forall i \in \mathbf{Pid}, \langle S[i], \Sigma[i]\rangle \to^i \langle Wait(S'[i]), \Sigma'[i]\rangle \qquad \langle S', \Sigma'\rangle \longrightarrow t}{\langle S, \Sigma\rangle \longrightarrow t} \quad [\textsc{all-wait}]$$

$$\frac{\begin{array}{c}\forall i \in \mathbf{Pid}, \langle S[i], \Sigma[i]\rangle \to^i \langle T[i], \Sigma'[i]\rangle \\ \exists i \in \mathbf{Pid}, T[i] = Ok \qquad \exists j \in \mathbf{Pid}, T[j] = Wait(s)\end{array}}{\langle S, \Sigma\rangle \longrightarrow \Omega_S} \quad [\textsc{glb-err}]$$

Figure 4.5 – **BSPlite** *global operational semantics*

Semantic domain The denotational semantics of **BSPlite** programs is given by functions ranging over **p**-vectors of environments, with optionally hidden components. Vectors are the elements of D defined by

$$D = (\mathbf{State_0})^{\mathbf{p}} \cup \{\bot, \Omega_S\}$$

where $\mathbf{State_0}$ stands for $\mathbf{State} \cup \{\mathbf{0}\}$ and $\mathbf{0}$ is a special constant denoting an hidden component. As usual, $\bot$ denotes non-termination, and as in the operational semantics, Ω_S denotes a synchronization error. For $I \subseteq \mathbf{Pid}$, we note $D_I = \{\theta \in D \setminus \{\bot, \Omega_S\} \mid \theta\, i = \mathbf{0} \iff i \notin I\}$. We now define three operations over vectors: *update*, *mask* and *combine*. These are formally defined in Figure 4.6. Intuitively, they can be understood thus:

- The *update* function updates environments pointwise where visible.
- The *mask* function hides components at which the condition does not hold.
- The *combine* function combines two vectors which must not have common visible components.

Semantic functions The semantic functions of **BSPlite** statements are given in Figure 4.7.

- Functions $[\![x := e]\!]$, $[\![\texttt{skip}]\!]$ and $[\![s_1; s_2]\!]$ operate pointwise in the standard way.
- The function $[\![\texttt{sync}]\!]$ verifies that either all components or none are involved in the computation. Otherwise, an error is returned: thus mimicking the behavior of a global synchronization barrier.

$$[x \leftarrow e] : D \to D \qquad x \in \mathbf{Var}, e \in \mathbf{AExp}_p \qquad \text{(update)}$$

$$\begin{array}{lcl} [x \leftarrow e]\,\bot & = & \bot \\ [x \leftarrow e]\,\Omega_S & = & \Omega_S \\ [x \leftarrow e]\,\theta & = & \lambda i. \begin{cases} \theta[i][x \leftarrow \mathcal{A}[\![e]\!]^i\,\theta[i]] & \text{if } \theta[i] \neq \mathbf{0} \\ \mathbf{0} & \text{if } \theta[i] = \mathbf{0} \end{cases} \end{array}$$

$$\partial_b : D \to D \qquad b \in \mathbf{BExp}_p \qquad \text{(mask)}$$

$$\partial_b = \lambda\theta. \begin{cases} \lambda i. \begin{cases} \theta[i] & \text{if } \mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt} \\ \mathbf{0} & \text{otherwise} \end{cases} & \theta \notin \{\bot, \Omega_S\} \\ \theta & \text{otherwise} \end{cases}$$

$$\| : D \times D \to D \qquad \text{(combine)}$$

$\|$ is the least partial commutative operator s.t.

$$\begin{array}{lcll} \bot & \| \; \theta & = & \bot \\ \Omega_S & \| \; \theta & = & \Omega_S \qquad \text{if } \theta \neq \bot \\ \theta & \| \; \theta' & = & \lambda i.\theta[i] + \theta'[i] \qquad \text{if } \theta \notin \{\bot, \Omega_S\} \end{array}$$

where $+$ is defined by $X + \mathbf{0} = \mathbf{0} + X = X$

Figure 4.6 – *Update, mask and combine operations*

$$
\begin{array}{lcl}
\multicolumn{3}{c}{[\![\cdot]\!] : \mathbf{Par} \to (D \to D)} \\
[\![x := e]\!] & = & [x \leftarrow e] \\
[\![\texttt{skip}]\!] & = & \lambda\theta.\theta \\
[\![\texttt{sync}]\!] & = & \lambda\theta. \begin{cases} \theta & \text{if } \theta \in D_\emptyset \cup D_{\mathbf{Pid}} \\ \Omega_S & \text{otherwise} \end{cases} \\
[\![s_1; s_2]\!] & = & [\![s_2]\!] \circ [\![s_1]\!] \\
[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}]\!] & = & \lambda\theta.[\![s_1]\!]\,(\partial_b\,\theta) \parallel [\![s_2]\!]\,(\partial_{!b}\,\theta) \\
[\![\texttt{while } b \texttt{ do } s \texttt{ end}]\!] & = & \mathit{fix}\, F \\
\multicolumn{3}{l}{F = \lambda f.\lambda\theta. \begin{cases} ((f \circ [\![s]\!])\,(\partial_b\,\theta)) \parallel (\partial_{!b}\,\theta) & \text{if } \partial_b\,\theta \notin D_\emptyset \cup \{\bot, \Omega_S\} \\ \theta & \text{otherwise} \end{cases}}
\end{array}
$$

Figure 4.7 – *Denotational semantics of textually aligned* **BSPlite** *programs*

- The function $[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}]\!]$ applies $[\![s_1]\!]$ or $[\![s_2]\!]$ at each visible component depending on the evaluation of b.
- The function $[\![\texttt{while } b \texttt{ do } s \texttt{ end}]\!]$ uses the same mechanism to define the semantics of loops by fixpoint.

Textual aligned barriers We now define programs with textually aligned barriers as those that have an error-free execution in the denotational semantics:

Definition 1. *A program s has textually aligned barriers for some set of environments D', if for all $\theta \in D'$ we have $[\![s]\!]\,\theta \neq \Omega_S$.*

Relationship to Operational Semantics

As expected, the denotational semantics coincides with the operational semantics for programs that are textually aligned. This intuition is formalized by the theorem below where it is shown that both semantics assign the same meaning to such programs.

Theorem 1. *Let $\theta \in D_{\mathbf{Pid}}$ be an unmasked environment vector. If $[\![s]\!]\,\theta = \theta' \notin \{\bot, \Omega_S\}$, then $\langle\langle s\rangle_i, \theta\rangle \longrightarrow \theta'$.*

Proof: See Appendix A.1. □

It follows immediately that if a program that terminates without error under the denotational semantics, then it also does so in the operational semantics:

Corollary 1. *If* $\forall \theta \in \mathbf{State}^{\mathbf{p}}, [\![s]\!]\,\theta \notin \{\bot, \Omega_S\}$ *then* $\forall \theta \in \mathbf{State}^{\mathbf{p}}, \langle \langle s \rangle_i, \theta \rangle \not\longrightarrow \Omega_S$.

As noted in Section 4.1.1, a program may be well-synchronized, although non-textually aligned. The relationship between the operational and denotational semantics reflect this fact. Consider the Example 2, given in Figure 4.1. The denotational semantics returns an error for this program in any environment with at least two processes, as the `sync` primitives are executed with masked environments. The operational semantics, on the other hand, does not return an error since each process executes exactly one `sync`. The denotational semantics forbids such partial synchronization, and anything that happens after it is considered undefined behavior. The static analysis presented in the next section rules out programs where such non-textually aligned synchronizations occur.

4.3 Static Approximation of Textual Alignment

Replicated synchronization, the static approximation of textual alignment, is verified by a restricted version of the Type and Effect System of Aiken and Gay for Barrier Inference [5]. We reformulate Barrier Inference as a data-flow analysis, restricted to identify programs with replicated synchronization, and prove it correct with respect to the semantics of **BSPlite**.

The analysis is divided into two phases: (1) a data-flow analysis which under-approximates the set of "*pid*-independent" variables at each program point — variables which do not have a data or control dependency on the special `pid` primitive; (2) a structural analysis that uses the result of the data-flow analysis to verify that each branching statement has a *pid*-independent guard expression, or that its sub-statements are syntactically synchronization-free.

The program s_{nok} in Figure 4.8 is used as running example. This program is erroneous, since the `sync` labeled 6 will only be executed by one process if $\mathbf{p} \geq 4$. This happens since the condition labeled 3 is not *pid*-independent, due to the data dependence of x on `pid` from the assignment labeled 2. As we will see in the rest of this section, the analysis rejects this program for same reason.

$$
\begin{aligned}
s_{nok} \;=\; & [i{:=}10]^1; \\
& [x{:=}\texttt{pid}]^2; \\
& \texttt{while } [0 < i]^3 \texttt{ do} \\
& \quad [\texttt{sync}]^4; \\
& \quad \texttt{if } [x = 3]^5 \texttt{ then} \\
& \quad\quad [\texttt{sync}]^6 \\
& \quad \texttt{else} \\
& \quad\quad [i{:=}0]^7 \\
& \quad [\texttt{end}]^{5^m}; \\
& \quad [i{:=}i-1]^8 \\
& \texttt{end}
\end{aligned}
$$

Figure 4.8 – *Example program* s_{nok}

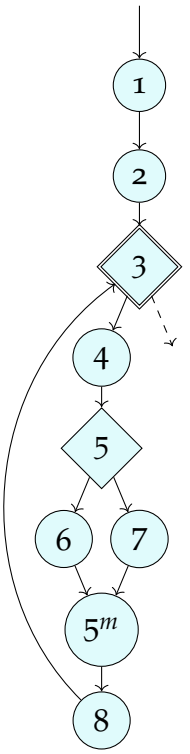


Figure 4.9 – *Control flow graph of* s_{nok}

4.3.1 Pid-Independence Data-Flow Analysis

Control Flow Graph and Merge Nodes

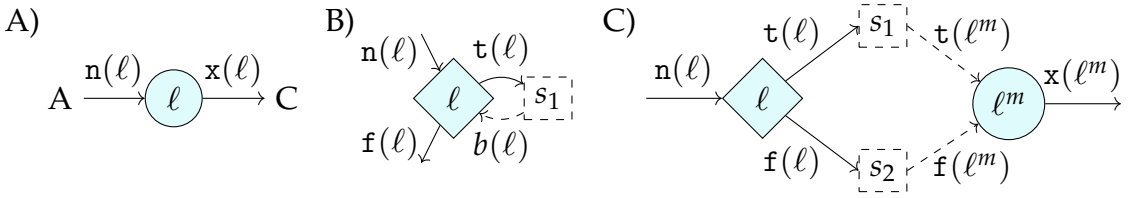
Conditionals in **BSPlite** have an additional label that corresponds to the merge of control flows from its two branches. Consequently, the program's control flow graph, as given by *flow*, has an additional node corresponding to this label. This node is called the **merge node**. See Figure 4.9, which contains the control flow graph of s_{nok}, for an illustration. In this graph, 5^m is the merge node corresponding to the conditional labeled 5.

To refer to the different incoming and outgoing edges of nodes, we define the functions $\texttt{n}, \texttt{x}, \texttt{f}, \texttt{t}, \texttt{b} : \mathbf{Lab} \hookrightarrow \mathbf{Lab} \times \mathbf{Lab}$, so that:

`n` returns the incoming edge from the immediately dominating node.

`x` returns the outgoing edge of nodes with one single outgoing edge.

`t` **(**respectively `f`**)** for nodes of conditional and `while` instructions, returns the outgoing edge corresponding to a truthy (falsy) evaluation of the condition. For merge nodes, returns the incoming edge from the `then` (`else`) branch of the conditional.



A) $s \in \{[\texttt{skip}]^\ell, [y{:=}b]^\ell, [\texttt{sync}]^\ell\}$ B) $s = \texttt{while}\ [b]^\ell\ \texttt{do}\ s_1\ \texttt{end}$

C) $s = \texttt{if}\ [b]^\ell\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ [\texttt{end}]^{\ell^m}$

Figure 4.10 – *Control flow graph and edge functions for instruction, loop and conditional statements*

`b` for the node of a `while` instruction, returns the incoming edge from its loop body.

Figure 4.10 illustrates these functions.

Analysis Domain

The data-flow analysis calculates an abstract state for each node and edge in the control flow graph of the program. An abstract state $l = (V, p)$ is a tuple consisting of: (1) an under-approximation of the set of variables considered *pid*-independent V; (2) a path abstraction p. To refer to the first and second component, $\pi^1(l) = V$ and $\pi^2(l) = p$ are used. The set of abstract states is L.

A **path** in $\mathbf{Lab}^*$ is associated with each program point. The path of a program point is the sequence of labels of each branching statement in the abstract syntax tree from the root-node to that program point. We write $p : \ell$ for the label ℓ appended to the path p[1]. By abuse of notation, we write $\ell_1 : \ell_2 : \cdots : l_n$ for $(((\epsilon : \ell_1) : \ell_2) : \cdots) : l_n$.

Let $\overline{\mathbf{Lab}}$ be the set of *marked labels*. The **semantic path** in $(\mathbf{Lab} \cup \overline{\mathbf{Lab}})^*$ at some program point and step of an execution is the path of the program point, where each label is marked if the guard expression was not *pid*-independent when evaluated last in the execution. Marking is an idempotent operation, written $\bar{}$ for both labels and sequences:

$$\begin{array}{lcllcl} \bar{} & : & (\mathbf{Lab} \cup \overline{\mathbf{Lab}}) \to \overline{\mathbf{Lab}} & \bar{} & : & (\mathbf{Lab} \cup \overline{\mathbf{Lab}})^* \to \overline{\mathbf{Lab}}^* \\ \bar{\ell} & = & \bar{\ell} & \bar{\epsilon} & = & \epsilon \\ \bar{\bar{\ell}} & = & \bar{\ell} & \overline{p:\ell} & = & \bar{p} : \bar{\ell} \end{array}$$

The **path abstraction** in $\mathbf{Path}^\sharp = (\mathbf{Lab} \cup \overline{\mathbf{Lab}})^* \cup \{\bot, \top\}$ for a program point

[1]Contrary to other sequences in this thesis, we grow paths to the right and the notation mirrors this. This gives a more natural reading since program nesting also indents to the right.

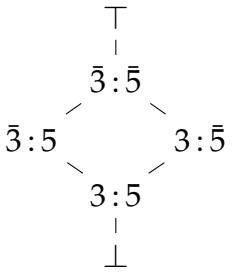


Figure 4.11 – *Example of path abstraction ordering*

is a conservative over-approximation of all the semantic paths of that program point, at any step and execution. If a label is marked in any of those semantic paths, then it must be marked in the path abstraction. The elements $\bot$ and $\top$ are added so that $\mathbf{Path}^\sharp$ forms a complete lattice:

$$\mathbf{Path}^\sharp = (\mathbf{Lab} \cup \overline{\mathbf{Lab}})^* \cup \{\bot, \top\} \qquad L = \mathcal{P}(\mathbf{Var}) \times \mathbf{Path}^\sharp$$

Consider the program s_{nok} in Figure 4.8. The path at Label 6 is $3:5$. A process reaching this label in the first iteration of the loop will have the semantic path $3:\bar{5}$, since the evaluation of the guard expression at 5 depends on *pid*. Thus a path abstraction at program point 6 must have label 5 marked. Path abstractions reflect the program nesting of the program and the *pid*-independence of guards.

The order on path abstractions $\preceq$ is given by the following rules:

$$\frac{}{\bot \preceq p} \qquad \frac{}{\epsilon \preceq \epsilon} \qquad \frac{p \preceq p'}{p:a \preceq p':\bar{a}} \qquad \frac{p \preceq p'}{p:a \preceq p':a} \qquad \frac{p \preceq p'}{p:\bar{a} \preceq p':\bar{a}} \qquad \frac{}{p \preceq \top}$$

Intuitively, if $p \preceq p'$ then both p and p' are path abstractions of the same program point, but p' can be a more conservative over-approximation than p, i.e., if a label is marked in p then it must also be in p' (or p' is $\top$). The least upper bound, written $\sqcup$, of two path abstractions referring to the same program point is the path abstraction where each label is marked when the corresponding label is marked in any of the arguments. If the arguments do not refer to the same program point, $\top$ is their least upper bound.

Example 1. *Taking as an example program point 6 in the program s_{nok}, the possible path abstractions are $\{\bot, 3:5, \bar{3}:5, 3:\bar{5}, \bar{3}:\bar{5}, \top\}$, and the ordering between them are (omitting transitive and reflexive orderings) $\bot \preceq 3:5, 3:5 \preceq \bar{3}:5, 3:5 \preceq 3:\bar{5}, \bar{3}:5 \preceq \bar{3}:\bar{5}, 3:\bar{5} \preceq \bar{3}:\bar{5}, \bar{3}:\bar{5} \preceq \top$, illustrated by the Hasse diagram in Figure 4.11.*

Now L forms a complete lattice ordered by $\sqsubseteq$, defined by

$$(V, p) \sqsubseteq (V', p') \iff V \supseteq V' \wedge p \preceq p'$$

As data-flow analysis in our formulation aims to find the smallest abstract state for each program point, this order rhymes with our intuition. Namely, that we want find the largest set of *pid*-independent variables at each program point and a path abstraction with the fewest marked guard expressions.

Path abstractions are modified by concatenating labels to existing path abstractions using a concatenation operator (.), and by merging path abstractions using the merge operator ($\triangledown$), defined below. Merging removes the last label from the two path abstractions of the same program point and returns the least upper bound of the remaining path abstractions.

$$
\begin{array}{lcl}
(.) & : & \mathbf{Path}^\sharp \times (\mathbf{Lab} \cup \overline{\mathbf{Lab}}) \to \mathbf{Path}^\sharp \qquad \text{(concatenation)} \\
p.\ell & = & \begin{cases} p:\ell & \text{if } \; p \in (\mathbf{Lab} \cup \overline{\mathbf{Lab}})^* \\ p & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\triangledown & : & \mathbf{Path}^\sharp \times \mathbf{Path}^\sharp \to \mathbf{Path}^\sharp \qquad \text{(merge)} \\
\multicolumn{3}{l}{\triangledown \text{ is the least commutative operator such that}} \\
p.\ell \;\triangledown\; p'.\ell' & = & p \sqcup p' \quad \text{if } \overline{p.\ell} = \overline{p'.\ell'} \\
\bot \;\triangledown\; p.\ell & = & p \\
\bot \;\triangledown\; \bot & = & \bot \\
p \;\triangledown\; p' & = & \top \qquad \text{otherwise}
\end{array}
$$

When the nesting of programs deepens, as in the body of `while` and `if` statements, the concatenation operator is used to form the path abstraction of the nested program points. As nesting becomes more shallow, such as when leaving the body of `while` and `if` statements, the merge operator is used to form the path abstraction of the program points following the branching statement.

Data-Flow Equations

The analysis computes an abstract state from L in each node in the control flow graph of a program. The abstract state of each node of a program s is defined by an equation system is given by $PI(s)$. This system defines two functions, which are defined in terms of each other. First, $PI_\circ : \mathbf{Lab} \to L$ that gives abstract state stored in each node (the incoming state). This abstract state in each node is defined in terms of the outgoing state on each label. The second function, $PI_\bullet : \mathbf{Lab} \times \mathbf{Lab} \to L$, gives the outgoing state. It is obtained by applying a transfer function that updates the abstract state of the source node according to the nature of the edge.

Contrary to the presentation in Section 2.4, this formulation allows nodes to

$$exprs((x = 3)) = \{(x = 3), (x), (3)\} \qquad free((x = 3)) = \{x\}$$

Figure 4.12 – *Examples of the functions exprs and free*

$$\begin{array}{lcl} \phi^d(e, V) & = & \texttt{pid} \notin exprs(e) \wedge free(e) \subseteq V \\ \phi^c(p) & = & p \in \mathbf{Lab}^* \cup \{\bot\} \end{array}$$

$$cdep(\ell, e, V) = \begin{cases} \ell & \text{if } \phi^d(e, V) \\ \bar{\ell} & \text{otherwise} \end{cases}$$

$$vdep((V, p), e, x) = \begin{cases} V \cup \{x\} & \text{if } \phi^d(e, V) \wedge \phi^c(p) \\ V \setminus \{x\} & \text{otherwise} \end{cases}$$

Figure 4.13 – *The predicates ϕ^d and ϕ^c and the functions cdep and vdep*

have distinct outgoing abstract states on different outgoing edges. Furthermore, the incoming state will not be the least upper bound on the outgoing abstract states of the predecessors, but a function on incoming abstract states that joins incoming states in a node-dependent manner.

Before defining $PI(s)$, we define the predicates ϕ^d (*pid* data-independent) and ϕ^c (*pid* control-independent) and the functions *cdep* and *vdep* (Figure 4.13), where $exprs(e)$ gives the sub-expressions of an arithmetic or boolean expression e and $free(e)$ is the set of free variables (see Figure 4.12 for illustrations of the *exprs* and *free* functions).

The function $cdep(\ell, e, V)$ is used at branching statements and marks the label ℓ when the guard expression e is not data-independent from *pid*, as determined by the predicate $\phi^d(e, V)$. This predicate holds if the expression e does not contain the primitive `pid` nor any potentially *pid*-dependent variables. The function $vdep((V, p), e, x)$ is used at assignments to conditionally add (respectively remove) the assigned variable x to the set of *pid*-independent variables V, when the assigned expression e is (respectively may not be) data and control independent on *pid*, as determined by the predicates $\phi^d(e, V)$ and $\phi^c(p)$. The latter holds for a non-$\top$ path abstraction that contains no marked labels.

For each node labeled ℓ in $flow(s)$, depending on the type of command s' that ℓ belongs to, the equation system $PI(s)$ defines the functions $PI_\circ$ and $PI_\bullet$ in

terms of each other by the following scheme:

$PI_\circ(\ell) = (\mathbf{Var}_s, \epsilon)$	if $init(s) = \ell$
$PI_\circ(\ell) = PI_\bullet(\mathtt{n}(\ell))$ $PI_\bullet(\mathtt{x}(\ell)) = PI_\circ(\ell)$	if $s' = [\mathtt{skip}]^\ell$ or $s' = [\mathtt{sync}]^\ell$
$PI_\circ(\ell) = PI_\bullet(\mathtt{n}(\ell))$ $PI_\bullet(\mathtt{x}(\ell)) = (vdep((V,p),e,y),p)$ where $(V,p) = PI_\circ(\ell)$	if $s' = [y{:=}e]^\ell$
$PI_\circ(\ell) = PI_\bullet(\mathtt{n}(\ell))$ $PI_\bullet(\mathtt{t}(\ell)) = (V, p.cdep(\ell,b,V))$ $PI_\bullet(\mathtt{f}(\ell)) = (V, p.cdep(\ell,b,V))$ where $(V,p) = PI_\circ(\ell)$	if $s' = \mathtt{if}\ [b]^\ell\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ [\mathtt{end}]$
$PI_\circ(\ell^m) = (V_\mathtt{t} \cap V_\mathtt{f}, p_\mathtt{t} \triangledown p_\mathtt{f})$ where $(V_\mathtt{t}, p_\mathtt{t}) = PI_\bullet(\mathtt{t}(\ell^m))$ $(V_\mathtt{f}, p_\mathtt{f}) = PI_\bullet(\mathtt{f}(\ell^m))$ $PI_\bullet(\mathtt{x}(\ell^m)) = PI_\circ(\ell^m)$	if $s' = \mathtt{if}\ [b]\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ [\mathtt{end}]^{\ell^m}$
$PI_\circ(\ell) = (V_\mathtt{n} \cap V_\mathtt{b}, p_\mathtt{n}.\ell \triangledown p_\mathtt{b})$ where $(V_\mathtt{n}, p_\mathtt{n}) = PI_\bullet(\mathtt{n}(\ell))$ $(V_\mathtt{b}, p_\mathtt{b}) = PI_\bullet(\mathtt{b}(\ell))$ $PI_\bullet(\mathtt{t}(\ell)) = (V, p.cdep(\ell,b,V))$ where $(V,p) = PI_\circ(\ell)$ $PI_\bullet(\mathtt{f}(\ell)) = PI_\circ(\ell)$	if $s' = \mathtt{while}\ [b]^\ell\ \mathtt{do}\ c_1\ \mathtt{end}$

Each data-flow equation defines the set of *pid*-independent variables and the path abstraction for the corresponding node or edge. In the initial node, we consider all variables of s (as given by $\mathbf{Var}_s$) *pid*-independent, and the path abstraction is empty. The set of variables at a node is the intersection of the *pid*-independent variables on all incoming edges, which ensures that the outgoing edges contain the variables which must be independent on all incoming paths. Additionally, at assignments, we add (or remove) the assigned variable when the assigned expression is *pid*-independent (or may not be *pid*-independent) in the abstract state on the incoming edge, using the *vdep*-function.

The data-flow equations define the path abstraction of each node so that it follows the nesting structure of the program. Accordingly, `skip`, `sync` and assign-

ment nodes do not change the path abstractions. At conditionals, we concatenate it's label to outgoing edges. The label is marked unless the guard expression is *pid*-independent, as handled by the *cdep*-function. At the corresponding merge node we restore the path abstraction at the entry of the conditional by merging the two incoming path abstractions. At `while` statements we merge the path abstraction of the incoming edge (with the statement's label concatenated) with that of the back edge, and concatenate the label of the node (possibly marked by *cdep*) on the outgoing value for the true edge. The equations for branching statements ensure that the path abstraction at their exit is the same as at their entry.

Figure 4.14 illustrates the generated equation system for the program s_{nok}, as well as the solution found by fix-point iteration. Note that since x is assigned `pid` at label 2, it is immediately removed from the set of *pid*-independent variables. As a result, the label 5 is marked when pushed on to edges $(5,6)$ and $(5,7)$ since this guard expression contains x. The path abstraction is now marked when the assignment at label 7 is made so x is removed as well. This variable is used in the guard expression of the outer while loop labeled 3, so its label will now be marked when pushed on to the outgoing edge to the body. Iteration will stop as soon as this change has propagated throughout the system, converging on the solution in the third column of the table.

Correctness

A solution to this equation system pi is a pair of functions $(pi_\circ, pi_\bullet)$ with $pi_\circ : \mathbf{Lab} \to L$ mapping nodes to incoming abstract states and $pi_\bullet : \mathbf{Lab} \times \mathbf{Lab} \to L$ mapping edges to outgoing abstract states. We write $pi \models PI(s)$ when pi solves $PI(s)$.

Definition 2. *We write $\sim_V \theta$ for $V \subseteq \mathbf{Var}$ when the visible members of the environment vector $\theta \in D$ agree on the variables in V:*

$$\begin{aligned} \sim_V \theta \iff \theta \notin \{\bot, \Omega_S\} \wedge \forall i, j \in \mathbf{Pid}, \theta[i] \neq \mathbf{0} \wedge \theta[j] \neq \mathbf{0} \\ \implies \forall x \in V, \theta[i](x) = \theta[j](x) \end{aligned}$$

The correctness of the data-flow analysis is established by the following theorem, which states that if execution of a program s starts with an environment vector agreeing on the values of the *pid*-independent variables at the initial node,

Node ℓ or Edge (ℓ, ℓ')	$PI_\circ(\ell)$ or $PI_\bullet((\ell, \ell'))$	Solution
1	$(\{i, x\}, \epsilon)$	$(\{i, x\}, \epsilon)$
$(1, 2)$	$(vdep(PI_\circ(1), (10), i), \pi^2(PI_\circ(1)))$	$(\{i, x\}, \epsilon)$
2	$PI_\bullet(\mathtt{n}(2))$	$(\{i, x\}, \epsilon)$
$(2, 3)$	$(vdep(PI_\circ(2), (\mathtt{pid}), x), \pi^2(PI_\circ(2)))$	$(\{i\}, \epsilon)$
3	$(V_n \cap V_\mathtt{b}, p_\mathtt{n}.l' \triangledown p_\mathtt{b})$ where $(V_\mathtt{n}, p_\mathtt{n}) = PI_\bullet(\mathtt{n}(3))$, $(V_\mathtt{b}, p_\mathtt{b}) = PI_\bullet(\mathtt{b}(3))$, $l' = cdep(3, (i), V_\mathtt{n} \cap V_\mathtt{b})$	$(\emptyset, \epsilon)$
$(3, 4)$	$(\pi^1(PI_\circ(3)), \pi^2(PI_\circ(3)).cdep(3, (i), \pi^1(PI_\circ(3))))$	$(\emptyset, \bar{3})$
4	$PI_\bullet(\mathtt{n}(4))$	$(\emptyset, \bar{3})$
$(4, 5)$	$PI_\circ(4)$	$(\emptyset, \bar{3})$
5	$PI_\bullet(\mathtt{n}(5))$	$(\emptyset, \bar{3})$
$(5, 6)$	$(\pi^1(PI_\circ(5)), \pi^2(PI_\circ(5)).cdep(5, (x = 3), \pi^1(PI_\circ(5))))$	$(\emptyset, \bar{3}.\bar{5})$
6	$PI_\bullet(\mathtt{n}(6))$	$(\emptyset, \bar{3}.\bar{5})$
$(5, 7)$	$(\pi^1(PI_\circ(5)), \pi^2(PI_\circ(5)).cdep(5, (x = 3), \pi^1(PI_\circ(5))))$	$(\emptyset, \bar{3}.\bar{5})$
7	$PI_\bullet(\mathtt{n}(7))$	$(\emptyset, \bar{3}.\bar{5})$
$(6, 5^m)$	$PI_\circ(6)$	$(\emptyset, \bar{3}.\bar{5})$
$(7, 5^m)$	$(vdep(PI_\circ(7), (0), i), \pi^2(PI_\circ(7)))$	$(\emptyset, \bar{3}.\bar{5})$
5^m	$(V_\mathtt{t} \cap V_\mathtt{f}, p_\mathtt{t} \triangledown p_\mathtt{f})$ where $(V_\mathtt{t}, p_\mathtt{t}) = PI_\bullet(\mathtt{t}(5^m))$, $(V_\mathtt{f}, p_\mathtt{f}) = PI_\bullet(\mathtt{f}(5^m))$	$(\emptyset, \bar{3})$
$(5^m, 8)$	$PI_\circ(5^m)$	$(\emptyset, \bar{3})$
8	$PI_\bullet(\mathtt{n}(8))$	$(\emptyset, \bar{3})$
$(8, 3)$	$(vdep(PI_\circ(8), (i - 1), i), \pi^2(PI_\circ(8)))$	$(\emptyset, \bar{3})$

Figure 4.14 – *Equation system $PI(s_{nok})$ and its solution*

$$\frac{}{RS([\texttt{skip}]^{\ell}, pi)} \qquad \frac{}{RS([\texttt{sync}]^{\ell}, pi)} \qquad \frac{}{RS([x{:=}e]^{\ell}, pi)}$$

$$\frac{RS(s_1, pi) \qquad RS(s_2, pi)}{RS(s_1; s_2, pi)}$$

$$\frac{(V, p) = \pi^1(pi)(\ell) \qquad \neg\phi^d(b, V) \implies sf^\sharp(s_1) \wedge sf^\sharp(s_2) \qquad RS(s_1, pi) \qquad RS(s_2, pi)}{RS(\texttt{if } [b]^{\ell} \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ [end]}^{\ell^m}, pi)}$$

$$\frac{(V, p) = \pi^1(pi)(\ell) \qquad \neg\phi^d(b, V) \implies sf^\sharp(s) \qquad RS(s, pi)}{RS(\texttt{while } [b]^{\ell} \texttt{ do } s \texttt{ end}, pi)}$$

$$\text{where } sf^\sharp(s) = \begin{cases} \texttt{ff} & \text{if } s \text{ syntactically contains a } \texttt{sync} \text{ primitive} \\ \texttt{tt} & \text{otherwise} \end{cases}$$

Figure 4.15 – *Replicated synchronization analysis*

then it finishes with an environment vector agreeing on the values of the *pid*-independent variables at the outgoing edge of the final node (given by $final_e(s)$).

Theorem 2. *Let s be a program, $\theta, \theta' \in D_{\mathbf{Pid}}$ two environment vectors and $(pi_\circ, pi_\bullet) \models PI(s)$. Let $V = \pi^1(pi_\circ(init(s)))$ and $V' = \pi^1(pi_\bullet(final_e(s)))$. If $\sim_V \theta$ and $[\![s]\!]\,\theta = \theta'$, then $\sim_{V'} \theta'$.*

Proof: See Appendix A.2. □

4.3.2 Replicated Synchronization Analysis

We now turn to the second phase of the analysis. Replicated synchronization (written RS), the static over-approximation of textual alignment, is defined by the inference system in Figure 4.15.

A program s is accepted by this analysis when we can derive the property $RS(s, pi)$, where pi is the result of the data-flow analysis on s. An accepted program has textually aligned barriers, and we can show the existence of an error-free execution under the denotational semantics for any initially replicated environment.

Theorem 3. *If $(pi_\circ, pi_\bullet) \vDash PI(s)$, $RS(s, pi)$, and $D^V_{\mathbf{Pid}} = \{\theta \in D_{\mathbf{Pid}} \mid \sim_V \theta\}$ where $V = \pi^1(pi_\circ(init(s)))$, then s is textually aligned for any environment in $D^V_{\mathbf{Pid}}$.*

Proof: See Appendix A.3. □

As a consequence of Corollary 1 in Section 4.2.2, the execution of s is also error free in the operational semantics if run under any environment from $D^V_{\mathbf{Pid}}$.

We now see that the property *RS* cannot be derived for the unsafe program s_{nok}. The rule for `while` statements requires that either the guard expression is *pid*-independent, $\phi^d(b, V)$, which is not the case since the set of *pid*-independent at program point 3 is $\emptyset$ (see the solution to $PI(s_{nok})$ in Figure 4.14), or that the body is syntactically synchronization free, which is clearly not the case for this loop due to the `sync` primitives labeled 4 and 6. Furthermore, the `if` statement labeled 5 is rejected for the same reason.

Consider also the running examples of Figure 4.1. The examples accepted by *RS* (if translated to **BSPlite** in the obvious way) are indicated by the column **RS**. Clearly, the guards of the conditionals in Examples 1 to 3 are not *pid*-independent, since they contain the `bsp_pid`. Therefore, and since these conditionals contain the `bsp_sync` primitive, *RS* can not be derived for these examples. As expected, the Example 1, which is not well-synchronized, is rejected. On the other hand, the program in Example 2 is well-synchronized. But, it falls outside the model of textually aligned barriers, and so cannot be accepted by the analysis. Nor is Example 3 accepted, even though it has textually aligned barriers. It is a victim of the static under-approximation of replicated synchronization. The program Example 4 is textually aligned, and hence well-synchronized. It is included in the static approximation of replicated synchronization, and hence accepted by *RS*.

4.4 Implementation

The analysis has been implemented as a Frama-C plug-in in ∼1200 lines of OCaml. It verifies the synchronization of BSPlib programs and programs using Lightweight Parallel Foundations [183], a forthcoming BSP library developed at Huawei.

Structures and arrays are handled conservatively by always assuming they may depend on *pid*. We apply the same approximation to reads from the pointers, as well as reads from local variables that may be affected by communication. The set of such variables is conservatively assumed to be those who have had their address taken, since this is a prerequisite for DRMA communication in BSPlib. We have also extended the analysis interprocedurally using small assumption sets [151].

In the remainder of this section we deal with the issues encountered when implementing the analysis in Frama-C.

4.4.1 Adapting the Analysis to Frama-C

When implementing the replicated synchronization analysis as a Frama-C plug-in, we had to work around two difficulties due to the data-flow analysis API in Frama-C and the internal representation of C programs in Frama-C:

1. Edge-by-edge flow fact updates: The way data-flow analyses are implemented in Frama-C is incompatible with the way the replicated synchronization analysis is specified. At the confluence of control flow branches, our formalization uses the flow facts from all predecessors at the same time. However, in Frama-C the flow facts from each predecessor is provided one by one. The workaround consists of changing the domain of the analysis so that each node stores the data on all predecessors, and delay the calculation of the merge until the data from all predecessors is available.

2. Frama-C loop normalization and the lack of merge nodes: The control flow graph of Frama-C differs from the one we specify the analysis on, and so the analysis had to be adapted by adding a structural pre-analysis phase that gives the order in which the path abstractions of predecessors must be merged.

The following sections illustrate these problems and our solution in more detail. Neither of these implementation issues concerns the treatment of the set of *pid*-independent variables: only the path abstractions are concerned. Hence, in order to improve legibility, we hide the treatment of variables in what follows.

4.4.2 Edge-by-Edge Flow Fact Updates

The path abstraction at the merge node corresponding to an `if` statement is given by the following data-flow equation:

$$PI_{\circ}(\ell^m) = PI_{\bullet}(\mathtt{t}(\ell^m)) \triangledown PI_{\bullet}(\mathtt{f}(\ell^m)) \tag{4.1}$$

A forward data-flow analysis in Frama-C is an OCaml module implementing the signature presented[2] in Figure 4.16, where L is the abstract domain of the analysis.

The function `combinePredecessors` needs to implement the Equation (4.1) for `if` statements. However, Frama-C only gives access to one predecessor path

[2]The full interface on-line: `http://arvidj.eu/frama-c/frama-c-Aluminium-20160501_api/html/Dataflow2.ForwardsTransfer.html`

`doInstr` : **Lab** $\to L \to L$
Implements the (forward) transfer function of an instruction.

`combinePredecessors` : **Lab** $\to L \to L \to L$
Receives the label of a node ℓ, the previously stored data in this node (corresponding to $PI_\circ(\ell)$ in our presentation), and the outgoing data from some predecessor (corresponding to $PI_\bullet(\ell', \ell)$ for some unspecified predecessor ℓ'). This function then returns the combination of the two flow facts, which will be stored in the node.

`doEdge` : **Lab** $\to$ **Lab** $\to L \to L$
Allows the data-flow analysis to implement a specific transfer function for each outgoing edge. Can default to identity if no special handling is required.

Figure 4.16 – *Simplified presentation of the signature of a module implementing a Frama-C forward data-flow analysis*

abstraction at a time. But, the Equation (4.1) depends on the path abstraction from both incoming edges. Implementing the data-flow equation for loops is hindered by the same problem. There, we need the path abstraction on the edge of the immediate dominator node, and the path abstraction on the back edge of the loop.

To overcome this problem, we change the analysis's abstract domain of our analysis to store the path abstractions on all incoming paths, as well as their provenance, in each node. The implementation collects these values in `combinePredecessors`, and then performs the merge in the transfer function `doEdge`.

The modified abstract domain, ignoring variables, is given by path abstraction maps:

$$L' = \mathbf{Lab} \to \mathbf{Path}^\sharp$$

The interpretation of the abstract state $l \in L'$ at some program point ℓ in the program s is that if $l\,\ell' = p$ then $PI_\bullet(\ell', \ell) = p$.

These total functions are implemented by finite maps from labels to path abstractions with $\bot$ taken as the default value for unmapped labels.

Path abstractions maps are partially ordered pointwise and the least upper bound is pointwise:

$$\begin{array}{lcl} l \sqsubseteq l' & \iff & \forall \ell \in \mathbf{Lab}, l\,\ell \preceq l'\,\ell \\ (l \sqcup l')\,\ell & = & (l\,\ell) \sqcup_{\mathbf{Path}^\sharp} (l'\,\ell) \end{array}$$

where $\sqcup_{\mathbf{Path}^\sharp}$ is the least upper bound on path abstractions.

The modified data-flow equation for incoming abstract states is now the least upper bound of the outgoing path abstractions of all predecessors:

$$PI_\circ(\ell) = \bigsqcup\{PI_\bullet(\ell', \ell) \mid (\ell', \ell) \in \mathit{flow}(s)\}$$

and it is the transfer function that merges the path abstractions. As in the original equation system, we have one single path abstraction per edge that we now represent by a singleton map, where the node's label maps to the path abstractions and all others labels map to $\bot$.

Implementation

Since the least upper bound operator is commutative, we can now implement the new data-flow equation directly in the `combinePredecessors` function. In pseudocode, we write:

```
combinePredecessors pred incoming old = incoming ⊔ old
```

4.4.3 Frama-C Control Flow Graph

The Frama-C CFG differs from that on which we specify the analysis in two crucial ways. First, Frama-C normalizes all loops (`while`, `do-while` and `for` loops) to a single `loop` construct and transforms the guard expression to an `if` statement guarding a `break`, so that guard expressions are no longer evaluated at the loop head:

```
while (guard) {
  // loop-body ...
}
```

is transformed into

```
loop {
  if (!guard)
    break;
  // loop-body ...
}
```

Second, there is no special node corresponding to the confluence of control flow after the execution of conditionals, such as the merge nodes that the analysis exploits. Consequently, any nodes may have several predecessors, and the correct order in which to merge the path abstractions from predecessor edges depends on the structure of the program.

We work around this by (1) extending the merge operator to sets of path abstractions and (2) implementing a structural analysis of the program, that associates a "merge tree" with each node. A merge tree specifies the correct order in which merge the path abstractions at predecessor nodes should take place at a specific node.

Consider the following examples:

$$
\begin{array}{lll}
s_1 = & \texttt{if } [b_1]^1 \texttt{ \{} & \\
& \quad \texttt{if } [b_2]^2 \texttt{ \{} & \\
& \quad\quad [\texttt{skip}]^3 & \\
& \quad \texttt{\} else \{} & \\
& \quad\quad [\texttt{skip}]^4 & \\
& \quad \texttt{\}} & \\
& \texttt{\} else \{} & \\
& \quad [\texttt{skip}]^5 & \\
& \texttt{\};} & \\
& [\texttt{skip}]^6 &
\end{array}
\qquad
\begin{array}{lll}
s_2 = & [\texttt{loop}]^1 \texttt{ \{} & \\
& \quad \texttt{if } [b_1]^2 \texttt{ \{ } [\texttt{break}]^3 \texttt{ \}} & \\
& \quad \texttt{if } [b_2]^4 \texttt{ \{ } [\texttt{break}]^5 \texttt{ \}} & \\
& \quad \texttt{if } [b_3]^6 \texttt{ \{} & \\
& \quad\quad [\texttt{skip}]^7 & \\
& \quad \texttt{\} else \{} & \\
& \quad\quad [\texttt{skip}]^8 & \\
& \quad \texttt{\}} & \\
& \quad [\texttt{skip}]^9 & \\
& \texttt{\};} & \\
& [\texttt{skip}]^{10} &
\end{array}
$$

To calculate the path abstraction for statement 6 in s_1, we need first merge the path abstractions on the two edges $(3,6)$ and $(4,6)$ coming from the inner `if` statement, before merging with the edge $(5,6)$ coming from the outer `else` branch. Assuming that the path abstractions from the branches of the inner `if` statement are p_3 and p_4 and that the one from the outer `else` branch is p_5, then the path abstraction for the last statement is given by the expression $(p_3 \triangledown p_4) \triangledown p_5$.

To obtain the correct path abstraction for the statement labeled 10 in s_2, we need to merge the path abstractions incoming from both `break` statements, but before doing so we must remove the label corresponding to the `if` branches on both path abstractions. Assuming these path abstractions are given by p_3 and p_5, respectively, and that we extend the $\triangledown$ operator to single arguments such that $\triangledown(p : \ell) = p$, then the path abstraction for the last statement is given by the expression $(\triangledown p_3) \triangledown (\triangledown p_5)$.

"Merge trees", defined below, correspond to the abstract syntax trees of such expressions, with "holes" for incoming path abstractions, provided by the data-flow analysis. Note that this approach assumes a structured control flow: a solution for arbitrary control flow graphs would require a more sophisticated way of tracking control dependencies [17].

Merge Trees

First, we extend the merge operator $\triangledown$ to sets of path abstractions with the following definition:

$$\begin{cases} \bigtriangledown ps & : & \mathcal{P}(\mathbf{Path}^\sharp) \to \mathbf{Path}^\sharp \\ \bigtriangledown ps & = & \bigsqcup_{p \in ps} \begin{cases} p' & p = p' : \ell \\ \bot & \text{otherwise} \end{cases} \end{cases}$$

We note that $\bigtriangledown\{p_1, p_2\}$ equals $p_1 \triangledown p_2$ as before, and that $\bigtriangledown\{p_1 : \ell\}$ equals p_1. We will write $p_1 \triangledown p_2$ as notation for the former and $\triangledown(p_1 : \ell)$ for the latter.

A merge tree corresponds to the abstract syntax tree of an expression over path abstractions. For instance, consider the expression:

$$p = (p_1 \triangledown p_2) \triangledown p_3$$

The corresponding merge tree needs to tell us that p_1 and p_2 must be merged, before we can merge that result with p_3. Merge trees are defined by the inductive data-type M:

$$\begin{aligned} M \ni m ::= I \mid T\, ms \mid J\, ms \mid L \mid N\, \ell \\ \text{where } ms \in \mathcal{P}(M) \text{ and } \ell \in \mathbf{Lab} \end{aligned}$$

The meaning of a merge tree is best explained by its interpretation, $\mathcal{M}[\![\cdot]\!]$, which is given by evaluating the tree in an abstract state from L':

$$\begin{cases} \mathcal{M}[\![\cdot]\!] & : & M \to L' \to \mathbf{Path}^\sharp \\ \mathcal{M}[\![I]\!]\, l & = & \epsilon \\ \mathcal{M}[\![T\, ms]\!]\, l & = & \bigtriangledown\{\mathcal{M}[\![m]\!]\, l \mid m \in ms\} \\ \mathcal{M}[\![J\, ms]\!]\, l & = & \bigsqcup\{\mathcal{M}[\![m]\!]\, l \mid m \in ms\} \\ \mathcal{M}[\![N\, \ell]\!]\, l & = & l\, \ell \\ \mathcal{M}[\![L]\!]\, l & = & \bot \end{cases}$$

We assign the merge tree I (initial) to the initial node of the program, so its interpretation is ϵ. When a node has several predecessors whose merge trees should be merged, we use the constructor T (tree). The constructor J (join) is used in the handling of loops. Its interpretation is the least-upper bound of its arguments. The interpretation of the constructor N (node) is given by the ab-

stract state. We use the constructor L (leaf) for statements whose path abstraction should not be considered, and its interpretation is bottom.

The merge trees corresponding to the last statement of the two previous examples are given by m_1 and m_2:

$$\begin{array}{rcl} m_1 & = & T\,\{T\,\{N\,3, N\,4\}, N\,5\} \\ m_2 & = & T\,\{T\,\{N\,3\}, T\,\{N\,5\}\} \end{array}$$

So given the incoming abstract states corresponding to program points 6 in example s_1 and program points 10 in example s_2:

$$\begin{array}{rcl} l_1 & = & \{3 \mapsto 1:2, 4 \mapsto 1:2, (5,1)\} \\ l_2 & = & \{3 \mapsto 1:2, 5 \mapsto 1:4\} \end{array}$$

we obtain

$$\begin{array}{rcl} \mathcal{M}[\![m_1]\!]\, l_1 & = & (1:2 \triangledown 1:2) \triangledown 1 = \epsilon \\ \mathcal{M}[\![m_2]\!]\, l_2 & = & (\triangledown 1:2) \triangledown (\triangledown 1:4) = 1 \triangledown 1 = \epsilon \end{array}$$

Calculating Merge Trees

We construct the merge trees of each program point by structural recursion on the abstract syntax tree of programs. We first define the auxiliary function *get_break* which returns the merge tree corresponding to successors of loops: they require special treatment since it is the `break` statements of the loops that indicate their exit. Here the syntactic group **FCStmt** refers to a simplified version of the abstract syntax tree on which Frama-C operates, with `break` and `loop` statements:

$$\begin{cases} \mathit{get\_break} & : \quad \textbf{FCStmt} \to M \\ \mathit{get\_break}(s) & = \quad T\,\{\mathit{get\_break}'(s)\} \end{cases}$$

$$\begin{cases} \mathit{get\_break}' & : \quad \textbf{FCStmt} \to M \\ \mathit{get\_break}'(\texttt{if}\ [e]^\ell\ \{s_1\}\ \texttt{else}\ \{s_2\}) & = \quad T\,\{\mathit{get\_break}'(s_1), \mathit{get\_break}'(s_2)\} \\ \mathit{get\_break}'([\texttt{loop}]^\ell\ \{s\}) & = \quad L \\ \mathit{get\_break}'([\texttt{break}]^\ell) & = \quad N\,\ell \\ \mathit{get\_break}'(s_1; s_2) & = \quad J\,\{\mathit{get\_break}'(s_1), \mathit{get\_break}'(s_2)\} \\ \mathit{get\_break}'(s) & = \quad L \end{cases}$$

For example, for the program

$$[\texttt{loop}]^1\ \{\texttt{if}\ [b]^2\ \{[\texttt{break}]^3\}\ \texttt{else}\ \{[\texttt{break}]^4\}; [\texttt{break}]^5;\}$$
$$[\texttt{skip}]^6;$$

the merge tree for label 6 is given by

$$\begin{array}{rcl} m_1 & = & \mathit{get\_break}(\texttt{if}\ [b]^2\ \{[\texttt{break}]^3\}\ \texttt{else}\ \{[\texttt{break}]^4\}; [\texttt{break}]^5) \\ & = & T\ \{J\ \{T\ \{N\ 3, N\ 4\}, N\ 5\}\} \end{array}$$

and assuming the environment

$$l_1 = \{3 \mapsto 1:2, 4 \mapsto 1:2, 5 \mapsto 1\}$$

we have $\mathcal{M}[\![m_1]\!]\, l_1 = \triangledown\{(1:2 \triangledown 1:2) \sqcup 1\} = \epsilon$.

The merge tree for programs preceded by the other statements is now given by *get_merge_tree*:

$$\left\{\begin{array}{l} \mathit{get\_merge\_tree} : M \to \mathbf{FCStmt} \to (M \times (\mathbf{Lab} \hookrightarrow M)) \\ \mathit{get\_merge\_tree}(m_{in}, \texttt{if}\ [e]^\ell\ \{s_1\}\ \texttt{else}\ \{s_2\}) = \\ \quad \text{let}\ (m_1, ms_1) = \mathit{get\_merge\_tree}(N\ \ell, s_1)\ \text{in} \\ \quad \text{let}\ (m_2, ms_2) = \mathit{get\_merge\_tree}(N\ \ell, s_2)\ \text{in} \\ \quad (T\ [m_1, m_2], (ms_1 \cup ms_2)[\ell \leftarrow m_{in}]) \\ \mathit{get\_merge\_tree}(m_{in}, [\texttt{loop}]^\ell\ \{s_1\}) = \\ \quad \text{let}\ (\_, ms_1) = \mathit{get\_merge\_tree}(N\ \ell, s_1)\ \text{in} \\ \quad (\mathit{get\_break}\, s_1, ms_1[\ell \leftarrow m_{in}]) \\ \mathit{get\_merge\_tree}(m_{in}, s_1; s_2) = \\ \quad \text{let}\ (m_1, ms_1) = \mathit{get\_merge\_tree}(m_{in}, s_1)\ \text{in} \\ \quad \text{let}\ (m_2, ms_2) = \mathit{get\_merge\_tree}(m_1, s_2)\ \text{in} \\ \quad (m_2, ms_1 \cup ms_2) \\ \mathit{get\_merge\_tree}(m_{in}, [\texttt{break}]^\ell) = (L, \{\ell \mapsto m_{in}\}) \\ \mathit{get\_merge\_tree}(m_{in}, s) = (N\ \mathit{init}(s), \{\mathit{init}(s) \mapsto m_{in}\}) \end{array}\right.$$

This function takes an initial tree, a program and returns a pair where the first component is the merge tree that should be associated to the successor of

that program, and the second a mapping associating a merge tree to each point in the program.

As we apply this function to example programs s_1 and s_2, we get the following merge trees for their respective final statements:

$$\begin{array}{lcl}(\pi^2(\mathit{get\_merge\_tree}\ I\ s_1))\ 6 & = & T\ \{T\ \{N\ 3, N\ 4\}, N\ 6\} \\ (\pi^2(\mathit{get\_merge\_tree}\ I\ s_2))\ 10 & = & T\ \{J\ \{T\ \{N\ 3\}, T\ \{N\ 5\}, T\ \{L, L\}, L\}\}\end{array}$$

where we note that the last merge has the same interpretation as the desired merge tree $T\ \{T\ \{N\ 3\}, T\ \{N\ 5\}\}$.

Applying Merge Trees

We let *ms* be the mapping returned in the second component of *get_merge_tree*, giving the merge tree of each program point in the analyzed program. We use it in the new transfer function $PI'_\bullet$ as follows:

$$\begin{array}{lcl}PI'_\bullet(\ell, \ell') & = & \text{let } p = \mathcal{M}[\![ms\ \ell]\!]\ PI_\circ(\ell) \\ & & \{\ell \mapsto f_{(\ell,\ell')}(p)\}\end{array}$$

where $f_{(\ell,\ell')}(p)$ implements the original transfer function, so that $PI_\bullet(\ell, \ell') = f_{(\ell,\ell')}(PI_\circ(\ell))$.

In other words, the modified transfer function calculates the path abstraction for that edge by interpreting the associated merge tree in the environment which is the incoming set of path abstractions of predecessors.

Handling Loop Guard Expressions

An additional problem caused by the representation of loops in Frama-C is that the guard expression that determines whether to leave or to re-execute a loop is no longer directly associated with the loop head. We solve this by another pre-phase which scans the body of each loop, finds all conditionals that guard `break` statements, and maps them to the corresponding `loop` statement.

In the transfer function for those conditional statements, in addition to marking the label of the conditional itself if the guard expression may be `pid`-dependent, we also mark the label of the corresponding loop.

```
1  int succ(int a) {
2    return a + 1;
3  }
4
5
6
7
8
9
10
11
12
```

```
13 void shift() {
14   int s = bsp_pid();
15   int p = bsp_nprocs();
16   int neighb = succ(s) % p;
17   int i = 0;
18   while (i < p) {
19     // communicate to neighbor
20     bsp_put(neighb, /* ... */);
21     bsp_sync();
22     i = succ(i);
23   }
24 }
```

Figure 4.17 – *A simple interprocedural BSPlib program. A naive interprocedural analysis cannot verify the synchronization of this program.*

4.4.4 Implementing Interprocedural Analysis Using Small Assumption Sets

The language **BSPlite** does not contains functions. Consequently, nor does the formalization of the analysis handle functions. However, realistic BSPlib programs are typically composed by a set of functions, and interprocedural analysis is needed to verify them.

In this section we detail our interprocedural extension of the analysis as formalized. First, we describe how we extend the pid-independence data-flow analysis using the standard "small assumption set" approach [151], and the adaptations necessary to implement this approach in Frama-C. Second, we describe how we adapt the second, replicated synchronization, phase of the analysis to an interprocedural setting.

Theoretical Background

A naive implementation of interprocedural data-flow analysis can treat function calls and returns as jumps, and input and output parameters as assignments. The drawback of this approach is that the program points of any given function have the same abstract state regardless of the *context* in which the function was called.

To illustrate this problem in our setting, consider a trivial function `succ` that takes an integer parameter `a` and returns its successor `a+1` (See Figure 4.17). If `succ` is invoked *once* with a pid-dependent argument, then the formal parameter `a` will be considered pid-dependent in all invocations, and so also its return value.

Now consider the function `shift` in Figure 4.17. Here, the `succ` function is used both at Line 16 compute pid-dependent identifier of each neighbor process `neighb`, and at Line 22 to compute the next state of the loop index `i`. Due to the first call, the result value of `succ` is always considered pid-dependent. Consequently, the variable `i` and the guard condition of the `while` loop is considered potentially pid-dependent and the synchronization in the loop is rejected.

A standard solution this problem is to introduce a *context* $c \in \Delta$ that allows the data-flow analysis to distinguish different calls to the same function. The idea is that the intraprocedural abstract domain L is extended to an interprocedural abstract domain

$$\hat{L} = P(\Delta \times L)$$

so that in each program point, we may have many different contexts c and their corresponding abstract states.

Call paths and assumption sets are two ways of encoding such contexts. The first represents control dependencies, and the latter data dependencies. Our assumption is that pid dependency is mainly due to data dependency and opted for the latter approach in our implementation.

Our intuition is that the context should contain the minimal set of information that is important for pid-dependency, namely the pid-dependence of arguments and the pid-dependency of the control point from which the call comes. To increase precision further, we also distinguish calls from different callees by adding the the function label from whence the call occurred. We use this label to encode pid-dependency of the control point of the call by marking it when this is the case.

$$\Delta = \mathcal{P}(\mathbf{Var}) \times (\mathbf{Lab} \cup \overline{\mathbf{Lab}} \cup \{\ell_?\})$$

The artificial label $\ell_?$ is used for the initial context in the entry point of the analyzed program, since, logically, the entry point function `main` has no callee. As the initial abstract state of the intraprocedural analysis of a program s is $(\mathbf{Var}_s, \epsilon)$ so the initial interprocedural abstract state of a program is $\{((\mathbf{Var}_{\texttt{main}}, \ell_?), (\mathbf{Var}_{\texttt{main}}, \epsilon))\}$ where $\mathbf{Var}_{\texttt{main}}$ is the set of local variables of the entry point function.

Now, it remains to adapt the transfer functions to take into account the context, and to formulate the transfer functions for function call and return to update the contexts. The transfer function for function calls must install the new context by transferring data-flow facts from the arguments to the formal parameters. Conversely, the transfer function for function returns must transfer

data-flow facts from the return value to the recipient of the return value in the corresponding context of the function call. The extensions of transfer functions to as well as the transfer functions for function call and return are standard. We do not detail them further and instead refer to a standard textbook [151, p. 82].

Implementation in Frama-C

Small assumption sets as described above are not directly implementable in Frama-C since the data-flow functor of Frama-C operates on one function in isolation. We circumvent this limitation by establishing a worklist algorithm. It operators over two data-structures:

1. A worklist of function-context pairs to analyze.
2. A global abstract state, consisting of the set of analyzed contexts per function and the resulting abstract states.

The implementation analyzes each function in the worklist using the corresponding context in isolation (intraprocedurally), and the result is then stored in the global abstract state.

For each function call that is encountered during the intraprocedural analysis, the corresponding context in the called function is computed. When this context is already present in the global abstract state, meaning that it has been previously analyzed, then analysis continues. Otherwise, the function and the corresponding context is added to the worklist.

After analyzing a function-context pair from the worklist, we verify whether its final abstract state has been updated. This indicates that the return value must be updated in all the callees of that context. When this is the case, we add all callees to the worklist.

The algorithm continues thus until exhaustion of the worklist, at which point all reachable functions have been analyzed.

A natural question is whether this procedures is guaranteed to terminate, particularly in the presence of recursive function calls. While we have not formally proved it, this should be assured by the fact that a function-context pair is only analyzed if it has not been previously analyzed or if its the callee of an function whose return state has been updated. As there is a finite number of function calls and function returns, termination is argued by the absence of infinite chains in the domain of contexts and the abstract states.

The second natural question is whether the precision afforded by this implementation is *sufficient* to treat realistic BSPlib program. And conversely, is this

precision *necessary*? After all, situations as in the motivational example of this section in Figure 4.17 might be rare in practice. We return to these questions in the Section 4.5, where the implementation is evaluated.

Interprocedural Replicated Synchronization

We now turn to the replicated synchronization phase of the analysis.

First, we define an instruction as being *potentially synchronizing* if it is a call to a function that is associated to `bsp_sync` in the transitive, reflexive transitive call graph of the analyzed program. In other words, a potentially synchronizing instruction is either a direct call to `bsp_sync`, or a call to a function that contain `bsp_sync`, etc. We then modify definition of *RS* in Section 4.3.2, to reject potentially synchronizing instructions that are guarded by pid-dependent guard conditions.

Finally, we apply this modified *RS* to all functions in the reflexive transitive closure of the call-graph from the entry function of the analyzed program, i.e. to all reachable functions.

4.5 Evaluation

We have evaluated the analysis by applying it to a set of BSPlib programs and verifying whether they have textually aligned barriers.

The set of analyzed programs are those distributed with BSPedupack [23] (`bspbench.c`, `bspfft.c`, `bspinprod.c`, `bsplu.c`, `bspmv.c`); a Huawei-developed BSP solution to the fixed-time constrained routing problem [95] (`SDN_BSP.c`); a set of programs developed by Alexandros V. Gerbessiotis, including a BSP parameter assessment program (`assess.c`), a comparison of different broadcast implementations (`brdmain.c` and `ppfmain.c`), an implementation of matrix multiplication (`mulmain.c`) and parallel radix sort (`prmain.c`); the set of example programs distributed with the Oxford BSP Toolset [141] (`array_get.c`, `array_put.c`, `helloworld.c` and `helloworld_init.c`, `helloworld_seq.c`, `reverse.c`, `sparse.c` and `sum.c`); and a branch-and-bound BSP algorithm for the $0-1$ Knapsack problem (`knapsack.c`).

The implementation does not handle `switch` statements and it requires that functions have unique return statements. We have manually rewritten `switch` to `if` statements, and functions with multiple returns to have one unique return in the analyzed programs.

Program	TAB	PIA	LOC	Analysis time (s)
`BSPedupack/bspbench.c`	✓	0	271	0.27
`BSPedupack/bspfft.c`	✓	1	600	0.28
`BSPedupack/bspinprod.c`	✓	0	293	0.24
`BSPedupack/bsplu.c`	✓	1	438	0.30
`BSPedupack/bspmv.c`	✓	0	900	0.31
`Huawei/SDN_BSP.c`	✓	0	1584	0.65
`AlexG/as02a/assess.c`	✓	3	675	0.37
`AlexG/bp03v2/brdmain.c`	✗	8	865	0.39
`AlexG/bp03v2/ppfmain.c`	✗	6	1259	0.41
`AlexG/mult03v6/mulmain.c`	✓	4	1261	0.43
`AlexG/prdx14v06/prmain.c`	✗	1	556	0.34
`OxfBSPlib/array_get.c`	✓	0	88	0.27
`OxfBSPlib/array_put.c`	✓	0	85	0.23
`OxfBSPlib/helloworld.c`	✓	0	10	0.18
`OxfBSPlib/helloworld_init.c`	✓	0	25	0.18
`OxfBSPlib/helloworld_seq.c`	✓	0	15	0.18
`OxfBSPlib/reverse.c`	✓	0	55	0.20
`OxfBSPlib/sparse.c`	✓	0	109	0.23
`OxfBSPlib/sum.c`	✓	0	74	0.26
`knapsack.c`	✓	1	280	0.30

Table 4.1 – *Evaluation results for Replicated Synchronization analysis. For each program, we indicate whether it is textually aligned (**TAB**), the number of pid-independence annotations added (**PIA**), and its size in number of lines of code (**LOC**). The two programs* `brdmain.c` *and* `prmain.c` *are not textually aligned due to a dependency on a non-initialized global variable.*

Some of the programs surveyed would not have been accepted as textually aligned by the implementation without further modification. Typically, their synchronization depends on configuration parameters that are broadcast from process 0 in the first superstep. As communicated variables, these are conservatively assumed to be *pid*-dependent by the analysis. We add *pid*-independence annotations to these variables that force the analysis to treat them as *pid*-independent.

The results of our evaluation is given in Table 4.1. For each program we mark whether the program was accepted as textually aligned, indicate the number of *pid*-independent annotations that were added. For each benchmark we also give its size and analysis time.

All programs we have surveyed but three are textually aligned. However, these three, `brdmain.c`, `ppfmain.c` and `prmain.c` are written with the intent of being textually aligned. But, they read global variables that are not initialized by all processes in the parallel section. By the BSPlib API, only process 0 is allowed to read such variables – other processes doing so results in undefined behavior, and hence nothing can be said about synchronization thereafter. In

a typical BSPlib implementation however, execution continues but each process may read different values from the uninitialized variables. Since synchronization in these programs depends on these variables, this undefined behavior may lead to synchronization errors. For these programs, we have indicated in Table 4.1 the number of *pid*-independent annotations necessary to show the program textually aligned if these uninitialized variables are repaired.

In sum, the results of our evaluation is in line with what previous authors have noted: scalable parallel programs adhere to highly structured synchronization patterns [5, 209] and moreover, they are typically textually aligned [209]. Our evaluation also shows that in order to fully automatize synchronization analysis and remove the need for annotations, more work is needed in the recognition of *pid*-independent communication patterns such as broadcasts and reductions.

We also conclude that the precision enabled by the interprocedural analysis is sufficient to analyze realistic programs, as none of the annotations introduced is due to imprecision of interprocedural analysis. On the other hand, our evaluation does not demonstrate that this precision is *necessary*. A naive solution might also suffice to verify realistic programs, whereas our solution has an additional cost in terms of implementation complexity and increased computation time. Hence, more research in order to evaluate other precision trade-offs. Our contribution provides an initial result by providing an upper bound on the precision necessary for verifying realistic programs.

4.6 Related Work

Synchronization analysis of the parallel programs has been extensively studied for the purpose of deadlock and data-race detection as well as optimization. We provide an overview of this work in Section 3.3.3. Notably, Aiken and Gay [5] propose Barrier Inference: a system to verify the synchronization of SPMD programs based on *structural correctness* and single-valued expressions (another term for our *pid*-independent expressions). This analysis has been extended in [209] for MPI, also handling inter-procedurality and indicating how non-textually aligned barriers match. Our contribution is inspired by [5], but differs in that we consider *textually aligned* programs, a subset of structurally correct programs with formally defined underpinnings [56, 55]. Our intent is to enforce textual alignment, with the intuition that the simplicity of this model will facilitate further analysis of other aspects of BSPlib programs. Similar ideas where

explored in the Titanium project, a Java dialect for high-performance computing. In [121], the author exploits that Titanium programs are statically guaranteed to be structurally correct to construct a May-Happen-in-Parallel relation and uses it for data race detection.

Several works propose operational semantics for BSPlib. Tesson et al. formalize BSPlib [184], and the semantics in this chapter can be seen as a subset of their semantics. Gava et al. formalize Paderborn's BSPlib [80], and later extend their formalization to consider subset synchronization [73]. The semantics we propose in this chapter differs from previous ones as explicitly designed to model only the features of BSPlib relevant to synchronization.

4.7 Concluding Remarks

In this chapter, we have presented our first contribution: the replicated synchronization analysis. This static analysis identifies programs with textually aligned barriers, a sufficient condition for correct synchronization. We have implemented this analysis in Frama-C, extended it to handle interprocedural programs and evaluated it on a set of 20 BSPlib programs. We have also proved the soundness of the analysis in the Coq proof assistant.

We identify the conservative treatment of communication as its main limitation. A natural direction for future research into static structural analysis of scalable parallel programs is thus applying communication analysis, in the goal of recognizing *pid*-independent communication patterns.

Using the replicated synchronization analysis, we statically reconstruct the synchronization pattern of the BSPlib program that constitutes the high-level structure of BSP algorithms. This structural property of BSPlib programs will be exploited in the following chapters to analyze the performance of BSPlib programs and to analyze registration in BSPlib programs.

Automatic Cost Analysis 5

Contents

5.1 **Seq** With Cost Annotations . . . 121
5.1.1 Syntax . . . 122
5.1.2 Semantics . . . 123
5.1.3 Sequential Cost . . . 125
5.1.4 Sequential Cost Analysis . . . 125
5.2 **BSPlite** With Cost Annotations and Communication . . . 126
5.2.1 Syntax . . . 127
5.2.2 Semantics . . . 127
5.2.3 Parallel Cost . . . 131
5.3 Cost Analysis . . . 134
5.3.1 Sequential Simulator . . . 135
5.3.2 Analyzing Communication Costs . . . 140
5.3.3 Analyzing Synchronization Costs . . . 145
5.3.4 Time Complexity of Analysis . . . 146
5.4 Implementation and Evaluation . . . 147
5.4.1 Benchmarks . . . 148
5.4.2 Symbolic Evaluation . . . 148
5.4.3 Concrete Evaluation . . . 148
5.4.4 Conclusion of Evaluation . . . 152
5.5 Related Work . . . 152
5.6 Concluding Remarks . . . 154

This chapter is extracted from the author's article [110].

The BSP model ensures *predictable scalability*. However, manual performance analysis of BSP programs, as demonstrated in Section 2.2.4, quickly becomes tedious and even intractable with growing program sizes. Furthermore, use cases such as on-line scheduling, algorithm prototyping and evaluation motivate *automatic* performance prediction of programs as another desirable quality of any parallel model.

To our knowledge, there are no methods for automatic cost analysis of imperative BSPlib programs: the BSPlib programmer is charged with manually deriving the cost of her program. In this chapter, we exploit textual alignment to develop an automatic parametric cost analysis for BSPlib programs. We rely on the static analysis of Chapter 4 to ensure this property.

Specifically, our contributions are:

- An adaptation of cost analysis for sequential programs [200, 6] to imperative Bulk Synchronous Parallel programs by program transformation.
- The application of the polyhedral model to the estimation of communication costs of imperative BSP programs.
- A prototype implementation combining these two ideas into a tool for static cost analysis of imperative BSP programs.
- Two evaluations, one symbolic and one concrete, of the implementation on 8 benchmarks.

The obtained cost formulas are *parameterized* by the input variables of the analyzed program. These variables can represent things such as the size of the problem instance size (e.g. the dimensions of a matrix for a matrix multiplication) and other arguments. They can also include the special variable `nprocs`, which corresponds to the BSP parameter $\mathbf{p}$. Thus the obtained cost formula bound the cost of running the program with any number of processes and with any input.

While not an inherent limitation of this work, the current implementation requires that analyzed programs are structured, in addition to the textual alignment constraint. In practice, we have yet to see a BSPlib program that does not fulfill these criteria.

We obtain a tight bound on communication cost when the input program has textually aligned, polyhedral communications that are *data-oblivious*. In other words, where the communication pattern does not depend on the contents of the communicated data. When this is not the case, we still obtain safe upper bounds.

However, progress in the applicability of the polyhedral model [20] leads us to believe that the communication of most real-world BSP programs can be represented in this model.

The chapter proceeds as follows. In Section 5.1 we add *cost annotations* to the sequential language first introduced in Section 2.4.1 with which we formalize the notion of sequential cost and sequential cost analysis. In the Section 5.2 we bring these extensions to our BSPlib formalization of Section 4.2 to formalize the notion of *BSP cost*. Since our cost analysis handles communication, we also extend our BSPlib formalization with DRMA primitives modeling those of BSPlib. In Section 5.3, we present the main contribution: the cost analysis for imperative BSPlib programs. In Section 5.4, we describes the prototype implementation and its evaluation. In Section 5.5 we position our contributions with respect to the state of the art. We conclude this chapter in Section 5.6 by discussing the limitations of the presented method and proposing future research directions for automatic cost analysis of BSPlib programs.

5.1 **Seq** With Cost Annotations

We begin by formalizing an extension of the sequential language **Seq** introduced in Section 2.4.1 allowing us to reason on the cost of executing programs. The semantics of the extended language is instrumented to return a trace of resource usage. With this, we compute the *sequential cost* of each execution. The cost is a measure on what abstract computational resources are needed to complete that execution. *Units* are used as arbitrary labels for different kinds of resources (arithmetic operations, floating point operations, I/O, etc.). We assume that the instructions of the input program are annotated with their individual cost and unit. Such annotations could also be added by an automatic pre-analysis. This scheme abstracts away from the specificities of different computer architectures and allows for the segmentation of costs.

We assume the existence of a Sequential Cost Analysis, which is a static analysis giving a safe upper bound on the cost of any execution (as determined by the annotation of each evaluated instruction in that execution) of a given program. The computed worst-case cost is parameterized by the input variables of the program. The description of such analyses can be found in literature [200, 6].

5.1.1 Syntax

We extend **Seq** with annotations as defined by the following grammar:

$$
\begin{array}{lcll}
\textbf{AExp} & \ni & e & ::= n \mid x \mid \textbf{log}\, e \\
\textbf{Seq} & \ni & s & ::= [x{:=}e]^{\ell} \mid [\texttt{skip}]^{\ell} \mid s;s \mid \texttt{if}\ [b]^{\ell}\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{end} \\
& & & \mid \texttt{while}\ [b]^{\ell}\ \texttt{do}\ s\ \texttt{end} \\
& & & \mid [x{:=}\texttt{any}]^{\ell} \mid [x{:=}[e_1 \mathinner{..} e_2]]^{\ell} \mid \{e\ u\}\ s
\end{array}
$$

Arithmetic expressions have been extended to include the integer binary logarithm. Boolean expressions are unchanged and we refer to Section 2.4.1 for their definition. In addition to those seen earlier, the set of instructions now include non-deterministic updates, range-constricted non-deterministic updates and work-annotated statements.

In addition to arithmetic expressions, variables can be assigned a non-deterministic value (`any`). This value can be optionally constrained to a range given by two arithmetic expressions ($[e_1..e_2]$), where e_1, respectively e_2, gives a lower, respectively upper, limit of the assigned value. Bot types of non-deterministic updates are used in Section 5.3 to sequentialize parallel programs.

Work annotations $\{e\ u\}$ can be added to any statement, and consists of an arithmetic expression e and a cost unit $u \in \textbf{Unit} = \{\texttt{a}, \texttt{b}, \ldots\}$. The expression gives the cost of the annotated statement and the unit gives the group of costs in which it should be counted. For instance, let $\texttt{a} \in \textbf{Unit}$ denote the cost of arithmetic operations. Then the annotated assignment $\{1\ \texttt{a}\}\ [x{:=}x+1]$ signifies that the cost of the assignment is 1 and that when executed this cost should be added to the total cost of arithmetic operations. Statements can have multiple annotations, thereby enabling modeling of statements with costs in different units. The cost of a program is given solely by these annotations: statements without annotations do not contribute to the cost of an execution. This annotation-based approach to specifying costly operations in a program is common in the cost analysis community [106, 57].

5.1.2 Semantics

The semantics of arithmetic and boolean expressions remain unchanged, with exception of a new clause for integer binary logarithm:

$$\begin{cases} \mathcal{A}[\![\cdot]\!] & : \quad \mathbf{AExp} \to (\mathbf{State} \to \mathbf{Nat}) \\ & \vdots \\ \mathcal{A}[\![\mathbf{log}\, e]\!]\, \sigma & = \quad \lfloor \log_2 \mathcal{A}[\![e]\!]\, \sigma \rfloor \end{cases}$$

As before, the semantics of **Seq** operates on mappings from variables to numerical values, but is now instrumented to collect *work traces*. A work trace $w \in \mathbf{WT} = (\mathbf{Nat} \times \mathbf{Unit})^*$ is a sequence that contains the value and cost unit of each evaluated work annotation in an execution.

The extended operational big-step semantics of **Seq** is given by the relation $\to$:

$$\to : (\mathbf{Seq} \times \mathbf{State}) \times (\mathbf{State} \times \mathbf{WT})$$

The rules defining this relation are given in Figure 5.1. Rule [**s−work**] defines how the evaluation of a work annotation adds an element to the work trace, by evaluating the expression of the annotation and adding it to the trace with the unit of the annotation. The effects of multiple annotations to the same underlying statement are accumulated. A sequence of statements (Rule [**s−seq**]) concatenates the traces from the execution of each subprogram, by the concatenation operator $+\!\!+$.

The semantics of non-deterministic assignments is given by the Rules [**s−havoc**] and [**s−havocr**] and assigns a non-deterministic value (from a restricted range for the latter) to the variable on the left-hand side, i.e. *havocking* it. This renders the language non-deterministic. The rules for conditional statements (Rules [**s−ift**] and [**s−iff**]) and loops (Rules [**s−whf**] and [**s−wht**]) are unchanged.

This semantics does not assign meaning to non-terminating programs. We restrict our focus to terminating programs, as typical BSP programs are algorithms that are intended to finish in finite time. Indeed, the BSP model does not assign cost for programs that do not finish. Some programs, such as reactive programs, repeat infinitely a finite calculation. These can be treated by identifying manually and analyzing separately their finite part, typically the body of an event loop.

$$\frac{}{\langle [\texttt{skip}]^{\ell}, \sigma\rangle \to \langle \sigma, \epsilon\rangle} \quad \textbf{(s-skip)}$$

$$\frac{}{\langle [x{:=}e]^{\ell}, \sigma\rangle \to \langle \sigma[x \leftarrow \mathcal{A}[\![e]\!]\,\sigma], \epsilon\rangle} \quad \textbf{(s-assign)}$$

$$\frac{\langle c, \sigma\rangle \to \langle \sigma', w\rangle}{\langle \{e\ u\}\ c, \sigma\rangle \to \langle \sigma', [\langle \mathcal{A}[\![e]\!]\,\sigma, u\rangle] +\!\!+ w\rangle} \quad \textbf{(s-work)}$$

$$\frac{\langle c_1, \sigma\rangle \to \langle \sigma'', w_1\rangle \qquad \langle c_2, \sigma''\rangle \to \langle \sigma', w_2\rangle}{\langle c_1; c_2, \sigma\rangle \to \langle \sigma', w_1 +\!\!+ w_2\rangle} \quad \textbf{(s-seq)}$$

$$\frac{n \in \mathbf{Nat}}{\langle [x{:=}\texttt{any}]^{\ell}, \sigma\rangle \to \langle \sigma[x \leftarrow n], \epsilon\rangle} \quad \textbf{(s-havoc)}$$

$$\frac{n_1 = \mathcal{A}[\![e_1]\!]\,\sigma \qquad n_2 = \mathcal{A}[\![e_2]\!]\,\sigma \qquad n \in [n_1 \ldots n_2]}{\langle [x{:=}[e_1 \,..\, e_2]]^{\ell}, \sigma\rangle \to \langle \sigma[x \leftarrow n], \epsilon\rangle} \quad \textbf{(s-havocr)}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{tt} \qquad \langle c_1, \sigma\rangle \to t}{\langle \texttt{if}\ [b]^{\ell}\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}, \sigma\rangle \to t} \quad \textbf{(s-ift)}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{ff} \qquad \langle c_2, \sigma\rangle \to t}{\langle \texttt{if}\ [b]^{\ell}\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}, \sigma\rangle \to t} \quad \textbf{(s-iff)}$$

$$\frac{\begin{array}{c}\mathcal{B}[\![b]\!]\,\sigma = \texttt{tt} \qquad \langle s, \sigma\rangle \to \langle \sigma'', w_1\rangle \\ \langle \texttt{while}\ [b]^{\ell}\ \texttt{do}\ s\ \texttt{end}, \sigma''\rangle \to \langle \sigma', w_2\rangle\end{array}}{\langle \texttt{while}\ [b]^{\ell}\ \texttt{do}\ c\ \texttt{end}, \sigma\rangle \to \langle \sigma', w_1 +\!\!+ w_2\rangle} \quad \textbf{(s-wht)}$$

$$\frac{\mathcal{B}[\![b]\!]\,\sigma = \texttt{ff}}{\langle \texttt{while}\ [b]^{\ell}\ \texttt{do}\ c\ \texttt{end}, \sigma\rangle \to \langle \sigma, \epsilon\rangle} \quad \textbf{(s-whf)}$$

Figure 5.1 – *Big-step semantics of* **Seq** *extended with cost annotations*

5.1.3 Sequential Cost

Given the work trace of an execution, we can obtain its cost. The cost is a mapping from units to numerical values:

$$\begin{cases} \mathit{Cost}_{\text{SEQ}} & : \quad \mathbf{WT} \to (\mathbf{Unit} \to \mathbf{Nat}) \\ \mathit{Cost}_{\text{SEQ}}(w) & = \quad \lambda u. \sum_{i=0}^{|w|-1} \begin{cases} n_i & \text{if } \; w[i] = \langle n_i, \mathrm{v}_i \rangle \text{ and } \mathrm{v}_i = u \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

5.1.4 Sequential Cost Analysis

There are sound static analyses automatically deriving conservative upper bounds on the cost of executing sequential, imperative programs [6]. Their basic principle is that of synthesizing *ranking functions* [47, 157]. These functions relate the semantics relation of the program to a well-ordered set, such that each execution step of the semantics relates the terminating state to a smaller element in this set. Intuitively, this can be understood as relating the program state to a counter and ensuring that each step of the program decrements this counter. This ensures termination, as the counter cannot decrement indefinitely. In addition to termination, the ranking functions also permit to obtain a measure on the number of steps necessary to terminate, i.e. the desired upper bounds. We refer to [161] for an elementary introduction to cost analysis, and to [47] for an elementary introduction to the related field of termination analysis.

We let SCA denote a sound sequential cost analysis for **Seq**. Given a program it returns an *upper bound* on the cost of executing that program. The bound is given as a cost expression from **CExp** that is parametric in the program's input parameters. Cost expressions **CExp** are arithmetic expressions extended with the symbol ω denoting unbounded cost.

$$\begin{array}{lcl} \text{SCA} & : & \mathbf{Seq} \to (\mathbf{Unit} \to \mathbf{CExp}) \\ \mathbf{CExp} & = & \mathbf{AExp} \cup \{\omega\} \end{array}$$

The semantics of cost expressions is given by the function $\mathcal{C}[\![\cdot]\!] : \mathbf{CExp} \times \mathbf{State} \to \mathbf{Nat}_\omega$ which is a natural extension of $\mathcal{A}[\![\cdot]\!]$ with $\mathbf{Nat}_\omega = \mathbf{Nat} \cup \{\omega\}$.

Since the halting problem is undecidable in general, SCA returns *conservative* upper bounds. Consequently, it might return ω for a program that actually terminates with any initial environment. However, since SCA is sound, we have the following for any unit u and program s:

$s_{\text{fact}} =$
$[f{:=}1]^1;$
`while` $[n > 0]^2$ `do`
$\{\mathbf{log}\, n\ \mathtt{a}\}\ [f{:=}f * n]^3;$
$[n{:=}n - 1]^4$
`end`

Figure 5.2 – *The program s_{fact} computes the factorial of the initial value of n. The work annotation at* **??** 2 *signifies that the assignment has a cost equal to the integer part of the binary logarithm of n, i.e. $\lfloor \log_2 n \rfloor$, when executed. The unit in this annotation is* a*, signifying additions.*

- If s terminates in any initial environment and the cost of its execution in unit u is at most n, then $\textsc{sca}(s)(u) = n'$ and $n \leq n'$ or $\textsc{sca}(s)(u) = \omega$.
- If s is non-terminating in some initial environment, then $\textsc{sca}(s)(u) = \omega$.

The intuitive understanding is that SCA may sometimes overestimate the cost needed to execute some program, but never underestimates it. And critically, it never reports that executing a program has a finite cost when, in reality, it may consume an unbounded amount of resources by running indefinitely.

Example 2. *The sequential program s_{fact} in Figure 5.2 computes the factorial of the parameter n and stores it in the variable f. For the sake of providing a non-trivial example, assume that n is of arbitrary precision so that multiplication by n consists of $\log n$ additions, and that we are interested in the number of such additions performed in any execution. We add a work annotation to the assignment at label 2 of value $\log n$ with unit* a *(for addition). With our implementation of* SCA*, based on [6], we have*

$$\textsc{sca}(s_{fact}) = \lambda u. \begin{cases} n \log n & \text{if } u = \mathtt{a} \\ 0 & \text{otherwise} \end{cases}$$

This is an upper bound on the cost of executing s_{fact}, parameterized by the input variable n, expressing that it performs at most $n \log n$ additions when calculating the factorial of n.

5.2 **BSPlite** With Cost Annotations and Communication

We now extend the BSPlib formalization **BSPlite** to allow us to reason on the parallel cost of programs. We add cost annotations and instrument the semantics

to return the information needed to obtain the *parallel cost* of each execution. We also add the communication primitives `put` and `get` to reason on the cost of communication.

5.2.1 Syntax

BSPlite with work annotations and communication is defined by the following grammar:

$$\begin{array}{rcll} \mathbf{Par} \ni s & ::= & [\texttt{skip}]^\ell \mid [x{:=}e]^\ell \mid \texttt{if } [b]^\ell \texttt{ then } s \texttt{ else } s \texttt{ end} \\ & \mid & \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end} \mid s;s \mid \{e\ u\}\ s \\ & \mid & [\texttt{sync}]^\ell \mid \texttt{put}^\ell(e,e,x) \mid \texttt{get}^\ell(e,y,x) \end{array}$$

where $e \in \mathbf{AExp}_p$, $b \in \mathbf{BExp}_p$, $x, y \in \mathbf{Var}$ and $u \in \mathbf{Unit}$. The set of arithmetic and boolean expressions $\mathbf{AExp}_p$ and $\mathbf{BExp}_p$ extend those of the previous section with the expressions `pid` and `nprocs`, with their usual meaning.

5.2.2 Semantics

As in the previous chapter, the semantics of **BSPlite** is defined by a set of local and global rules. Like the sequential semantics of the previous section, the local semantics is instrumented to collect work traces. But, it now also collects communication request traces. These traces are used to perform communication, but also in the calculation of communication costs.

The global semantics is extended to execute the communication request traces, as detailed below. It is also instrumented to collect the work traces and communication request traces of each process in two matrices, which are used to calculate the parallel cost of the execution. In these matrices, each column corresponds to the traces of one superstep and each row corresponds to the traces of one process. In this way, for an execution in S supersteps, two $(\mathbf{p} \times S)$-matrices are collected where element (i, j) corresponds to the work trace respectively communication trace generated by process i in superstep j.

The instrumented semantics of local computation is given by the relation $\rightarrow^i$, indexed by the $i \in \mathbf{Pid}$ and defined by the rules in Figure 5.3:

$$\begin{array}{rcll} \rightarrow^i & : & (\mathbf{Par} \times \mathbf{State}) \times (\mathbf{Term} \times \mathbf{State} \times \mathbf{WT} \times \mathbf{CReq}^*) & i \in \mathbf{Pid} \\ \mathbf{CReq} \ni c & ::= & \langle n@i \xrightarrow{\text{put}} x@i' \rangle \mid \langle x@i' \xleftarrow{\text{get}} y@i \rangle & \\ & & \text{with } n \in \mathbf{Nat}, i, i' \in \mathbf{Pid}, x, y \in \mathbf{Var} & \end{array}$$

$$\frac{}{\langle [\texttt{skip}]^{\ell}, \sigma\rangle \to^{i} \langle Ok, \sigma, \epsilon, \epsilon\rangle} \quad \textbf{(p-skip)}$$

$$\frac{}{\langle [x{:=}e]^{\ell}, \sigma\rangle \to^{i} \langle Ok, \sigma[x \leftarrow \mathcal{A}[\![e]\!]^{i}\,\sigma], \epsilon, \epsilon\rangle} \quad \textbf{(p-assign)}$$

$$\frac{\langle s, \sigma\rangle \to \langle t, \sigma', w, r\rangle}{\langle \{e\ u\}\ s, \sigma\rangle \to \langle t, \sigma', [\langle \mathcal{A}[\![e]\!]^{i}\,\sigma, u\rangle] +\!\!+ w, r\rangle} \quad \textbf{(p-work)}$$

$$\frac{}{\langle [\texttt{sync}]^{\ell}, \sigma\rangle \to^{i} \langle Wait([\texttt{skip}]^{\ell}), \sigma, \epsilon, \epsilon\rangle} \quad \textbf{(p-sync)}$$

$$\frac{\langle s_1, \sigma\rangle \to^{i} \langle Ok, \sigma'', w_1, r_1\rangle \qquad \langle s_2, \sigma''\rangle \to^{i} \langle t, \sigma', w_2, r_2\rangle}{\langle s_1; s_2, \sigma\rangle \to^{i} \langle t, \sigma', w_1 +\!\!+ w_2, r_1 +\!\!+ r_2\rangle} \quad \textbf{(p-seq-ok)}$$

$$\frac{\langle s_1, \sigma\rangle \to^{i} \langle Wait(s_1'), \sigma', w, r\rangle}{\langle s_1; s_2, \sigma\rangle \to^{i} \langle Wait(s_1'; s_2), \sigma', w, r\rangle} \quad \textbf{(p-seq-wait)}$$

$$\frac{n \in \mathbf{Nat}}{\langle [x{:=}\texttt{any}]^{\ell}, \sigma\rangle \to \langle Ok, \sigma[x \leftarrow n], \epsilon, \epsilon\rangle} \quad \textbf{(p-havoc)}$$

$$\frac{n_1 = \mathcal{A}[\![e_1]\!]^{i}\,\sigma \qquad n_2 = \mathcal{A}[\![e_2]\!]^{i}\,\sigma \qquad n \in [n_1 \ldots n_2]}{\langle [x{:=}[e_1\,..\,e_2]]^{\ell}, \sigma\rangle \to \langle Ok, \sigma[x \leftarrow n], \epsilon, \epsilon\rangle} \quad \textbf{(p-havocr)}$$

$$\frac{i' = \mathcal{A}[\![e_1]\!]^{i}\,\sigma \qquad n = \mathcal{A}[\![e_2]\!]^{i}\,\sigma}{\langle \texttt{put}^{\ell}(e_1, e_2, x), \sigma\rangle \to^{i} \langle Ok, \sigma, \epsilon, [\langle n@i \xrightarrow{\text{put}} x@i'\rangle]\rangle} \quad \textbf{(p-put)}$$

$$\frac{i' = \mathcal{A}[\![e]\!]^{i}\,\sigma}{\langle \texttt{get}^{\ell}(e, y, x), \sigma\rangle \to^{i} \langle Ok, \sigma, \epsilon, [\langle x@i \xleftarrow{\text{get}} y@i'\rangle]\rangle} \quad \textbf{(p-get)}$$

(5.1)

Figure 5.3 – *Local big-step semantics of* **BSPlite** *with cost annotations and communication*

$$\frac{\mathcal{B}[\![b]\!]^i\,\sigma = \texttt{tt} \qquad \langle s_1, \sigma\rangle \rightarrow^i t}{\langle \texttt{if } [b]^\ell \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}, \sigma\rangle \rightarrow^i t} \quad \textbf{(p−ift)}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\sigma = \texttt{ff} \qquad \langle s_2, \sigma\rangle \rightarrow^i t}{\langle \texttt{if } [b]^\ell \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}, \sigma\rangle \rightarrow^i t} \quad \textbf{(p−iff)}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\sigma = \texttt{tt} \qquad \langle s; \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^i t}{\langle \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^i t} \quad \textbf{(p−wht)}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\sigma = \texttt{ff}}{\langle \texttt{while } [b]^\ell \texttt{ do } s \texttt{ end}, \sigma\rangle \rightarrow^i \langle Ok, \sigma, \epsilon, \epsilon\rangle} \quad \textbf{(p−whf)}$$

Figure 5.3 – *Local big-step semantics of* **BSPlite** *with cost annotations and communication, continued*

$$\frac{\begin{array}{c}\forall i \in \textbf{Pid}, \langle C[i], E[i]\rangle \rightarrow^i \langle Wait(C'[i]), E'[i], W[i], R[i]\rangle \\ Comm(E', R, E'') \qquad \langle C', E''\rangle \longrightarrow^S \langle E''', W', R'\rangle\end{array}}{\langle C, E\rangle \longrightarrow^{S+1} \langle E''', W : W', R : R'\rangle} \quad \textbf{(p−glb−all−wait)}$$

$$\frac{\forall i \in \textbf{Pid}, \langle C[i], E[i]\rangle \rightarrow^i \langle Ok, E'[i], W[i], R[i]\rangle \qquad Comm(E', R, E'')}{\langle C, E\rangle \longrightarrow^1 \langle E'', W, E\rangle} \quad \textbf{(p−glb−all−ok)}$$

Figure 5.4 – *Global big-step semantics of* **BSPlite** *with cost annotations and communication*

where **Term** is the termination state as before (see Section 4.2.1). Work traces have the same meaning as in the sequential language. Communication requests in **CReq** are generated by the `put` and `get` primitives. Rule [**p−put**] appends the form $\langle n@i \xrightarrow{\text{put}} x@i'\rangle$ to the communication request trace, signifying process i requesting that the value n be put into variable x at process i'. Rule [**p−get**] appends the form $\langle x@i' \xleftarrow{\text{get}} y@i\rangle$ to the communication request trace, signifying process i' requesting that the contents of variable y at process i be retrieved into its variable x. Following the convention in this thesis, for both forms we say that the **source** is i and the **destination** is i'. The process initiating the call is called the **origin** (i for Rule [**p−put**] and i' for Rule [**p−get**]) and the remote process is called the **target** (i' for Rule [**p−put**] and i for Rule [**p−get**]).

The global level of the operational semantics of **BSPlite** programs is given by

the relation $\longrightarrow^S$, indexed by the number $S > 1$ of supersteps in the execution:

$$\longrightarrow^S : (\mathbf{Par}^{\mathbf{p}} \times \mathbf{State}^{\mathbf{p}}) \times (\mathbf{State}^{\mathbf{p}} \times \mathbf{WT}^{\mathbf{p}\times S} \times \mathbf{CReq}^{\mathbf{p}\times S}) \qquad S \in \mathbf{Nat}$$

where $A^{\mathbf{p}}$ denotes a column-vector of height $\mathbf{p}$ and $A^{\mathbf{p}\times S}$ denotes a $(\mathbf{p} \times S)$-matrix. This relation is defined by the rules in Figure 5.4. There are two differences with respect to the global rules of the **BSPlite** in Section 4.2: the handling of work traces and communication and the lack of explicit synchronization errors.

In Rule [**p−glb−all−ok**], all processes terminate. The work trace and the communication request trace calculated by each process are used to form $(\mathbf{p} \times 1)$-matrices of work respectively communication request traces. The communication requests are executed by the *Comm* relation detailed below, obtaining a new environment per process.

In Rule [**p−glb−all−wait**], all processes request synchronization. As before, they all calculate a continuation and a new environment. Just as in the terminating rule, communication is used to obtain a new environment per process. Global computation then continues recursively with the continuation and new environment of each process. Thus, a final environment vector is obtained, along with the matrices from the execution of the remaining supersteps.

The final matrices are then obtained by concatenating the trace vectors of the first superstep to these matrices. Concatenation of vectors to matrices is given by the operator $(:) : A^{\mathbf{p}} \times A^{\mathbf{p}\times S} \to A^{\mathbf{p}\times(S+1)}$.

The $Comm$: $\mathbf{State}^{\mathbf{p}} \times \mathbf{CReq}^* \times \mathbf{State}^{\mathbf{p}}$ -relation defines communication by executing the communication requests traces As BSPlib leaves several details of communication up to the implementation, we underspecify *Comm*. We only require that it executes all communication requests. Thus, different BSPlib implementations can be modeled precisely by varying *Comm*. Consider concurrent writes, occurring when the trace contains two `put` requests to the same variable on the same process. *Comm* can handle this by taking either value of the requests non-deterministically or deterministically by imposing a priority on origin processes, by combining the values, or disabling such writes completely. We provide no more precise definition of *Comm* in this chapter, since it has no bearing on the cost. For the interested reader, a more detailed definition is given in Chapter 6.

In this chapter, we consider programs that are assumed to synchronize correctly. For this reason, we do not model synchronization errors explicitly as in Chapter 4. Instead, the semantics of executions with incoherent local termination states is undefined.

SPMD Execution

The semantics of a **BSPlite** program s with the initial environment σ executed in SPMD fashion is obtained by replicating the programs and the initial environment:

$$\langle \langle s \rangle_{i \in \mathbf{Pid}}, \langle \sigma \rangle_{i \in \mathbf{Pid}} \rangle \longrightarrow^S \langle E, W, R \rangle$$

5.2.3 Parallel Cost

The parallel cost of a **BSPlite** program follows the BSP model, and so is given in terms of local computation, communication and synchronization. The units $\mathtt{g}, \mathtt{l} \in \mathbf{Unit}$, assumed not to appear in user-provided work annotations, denote communication and synchronization costs respectively. Remaining units denote local computation and are normalized by the function w.

The BSP cost of an execution is normally given as a sum of computation, communication and synchronization costs, but we shall give it in the form of a function $f : \mathbf{Unit} \to \mathbf{Nat}$. The classic BSP cost expressed by f is given by $\sum_{u \notin \{\mathtt{g},\mathtt{l}\}} f(u)w(u)\mathbf{r} + f(\mathtt{g})\mathbf{g} + f(\mathtt{l})\mathbf{l}$.

To define the cost of local computation we introduce the concept of *global work traces*. A global work trace is a vector of traces corresponding to the selection of one trace from each column of the work trace matrix of one execution. The set of global work traces of a work trace matrix is defined:

$$\begin{cases} G & : \quad \mathbf{WT}^{\mathbf{p} \times S} \to \mathcal{P}(\mathbf{WT}^S) \\ G(W) & = \quad \{[W[i_0, 0], W[i_1, 1], \ldots, W[i_{S-1}, S-1]] \mid [i_0, \ldots, i_{S-1}] \in \mathbf{Pid}^S\} \end{cases}$$

The cost of communication is defined in terms of h-relations. The h-relation of a superstep is defined as the maximum fan-in or fan-out of any process in that superstep, and can be calculated from the communication request traces of all processes in that superstep.

We define the $\mathcal{H}_i^+, \mathcal{H}_i^- : \mathbf{CReq}^* \to \mathbf{Nat}$ functions, for $i \in \mathbf{Pid}$, giving the fan-out respectively fan-in of process i resulting from the execution of a communication request trace. Using these, we define $\mathcal{H} : \mathbf{CReq}^* \to \mathbf{Nat}$ to give the maximum fan-out or fan-in of any process for a given communication request

trace.

$$\begin{aligned}
\mathcal{H}_i^+(r) &= \sum_{k=0}^{|r|-1} \begin{cases} 1 & \text{if the source of } r[k] \text{ is } i \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{H}_i^-(r) &= \sum_{k=0}^{|r|-1} \begin{cases} 1 & \text{if the destination of } r[k] \text{ is } i \\ 0 & \text{otherwise} \end{cases} \\
\mathcal{H}(r) &= \max_{i \in \mathbf{Pid}}(\max(\mathcal{H}_i^+(r), \mathcal{H}_i^-(r)))
\end{aligned}$$

The communication relation *Comm* that parameterizes the global semantics affects expressibility. Given a problem, different choices of *Comm* may permit solutions of different costs, but the program text of each solution would be different. The cost analysis (Section 5.3), being defined on the program text, would reflect the new cost. A program might generate a request whose effect is not defined by some choice of *Comm*, such as a concurrent write. However, our analysis returns an upper bound on the communication cost under the assumption that all communication requests traces are executed, and is therefore independent of *Comm*.

Using the $\mathcal{H}$-function and the concept of global traces, we define the parallel cost of an execution from the generated work trace matrix and communication request matrix:

$$\begin{aligned}
Cost_{\text{PAR}} &: \mathbf{WT}^{\mathbf{p} \times S} \times \mathbf{CReq}^{\mathbf{p} \times S} \to (\mathbf{Unit} \to \mathbf{Nat}) \\
Cost_{\text{PAR}}(W, R) &= \lambda u. \begin{cases} \max\{Cost_{\text{SEQ}}(+\!\!+ T, u) \mid T \in G(W)\} & \text{if } u \notin \{\mathtt{g}, \mathtt{l}\} \\ \sum_{0 \le k < S} \mathcal{H}(+\!\!+ R[*, k]) & \text{if } u = \mathtt{g} \\ S & \text{if } u = \mathtt{l} \end{cases}
\end{aligned}$$

where $+\!\!+$ gives the concatenation of each trace in a vector and $R[*, k]$ is the kth column of R. The parallel cost of local computation for some unit $u \notin \{\mathtt{g}, \mathtt{l}\}$ is equal to the cost of the global work trace with the highest sequential cost in that unit. The cost of communication ($u = \mathtt{g}$) is the sum of the h-relation of each column in the communication request matrix. The cost of synchronization ($u = \mathtt{l}$) is equal to the number of supersteps S in the execution.

Example 3. *The program s_{scan} (adopted from [184]) for calculating prefix sum is given in Figure 5.5. The input of the program is a* **p***-vector where the ith component is stored in the variable x at process i. The assignment at Label 7 is annotated with a work annotation* 1 *of unit* `w`.

The execution of this program over 4 *processes consists of* 3 *supersteps, and is illustrated in Figure 5.6. We write σ^y to denote $\sigma[x \leftarrow y]$. In this example, the initial value*

$s_{\text{scan}} =$
$[i{:=}1]^1;$
`while` $[i < \texttt{nprocs}]^2$ `do`
`if` $[\texttt{pid} \geq i]^3$ `then`
$\texttt{get}^4(\texttt{pid} - i, x, x_{in})$
`end;`
$[\texttt{sync}]^5;$
`if` $[\texttt{pid} \geq i]^6$ `then`
$\{1\ \texttt{w}\}\ [x{:=}x + x_{in}]^7$
`end;`
$[i{:=}2 * i]^8$
`end`

Figure 5.5 – *The program s_{scan} implementing parallel prefix calculation*

$$\langle\langle s_{\text{scan}}\rangle_i, \langle\sigma^1\rangle_i\rangle \longrightarrow^3$$

$$\left\langle \begin{pmatrix}\sigma^1\\ \sigma^2\\ \sigma^3\\ \sigma^4\end{pmatrix}, \begin{pmatrix}\epsilon & \epsilon & \epsilon\\ \epsilon & [\langle 1,\texttt{w}\rangle] & \epsilon\\ \epsilon & [\langle 1,\texttt{w}\rangle] & [\langle 1,\texttt{w}\rangle]\\ \epsilon & [\langle 1,\texttt{w}\rangle] & [\langle 1,\texttt{w}\rangle]\end{pmatrix}, \begin{pmatrix}\epsilon & \epsilon & \epsilon\\ [\langle x_{in}@1 \xleftarrow{\text{get}} x@0\rangle] & \epsilon & \epsilon\\ [\langle x_{in}@2 \xleftarrow{\text{get}} x@1\rangle] & [\langle x_{in}@2 \xleftarrow{\text{get}} x@0\rangle] & \epsilon\\ [\langle x_{in}@3 \xleftarrow{\text{get}} x@2\rangle] & [\langle x_{in}@3 \xleftarrow{\text{get}} x@1\rangle] & \epsilon\end{pmatrix} \right\rangle$$

Figure 5.6 – *The resulting state vector, work trace and communication request matrix from the execution of s_{scan} with 4 processes in 3 supersteps. In both matrices, rows correspond to processes, and columns to supersteps.*

of x in all processes is 1. The values of the other variables are omitted for brevity. The cost of this execution is

$$\lambda u. \begin{cases} 0+1+1=2 & \text{if } u = \mathtt{w} \\ 1+1+0=2 & \text{if } u = \mathtt{g} \\ 1+1+1=3 & \text{if } u = \mathtt{l} \end{cases}$$

where the local computation cost is given by the global work trace $[\epsilon, [\langle 1, \mathtt{w}\rangle], [\langle 1, \mathtt{w}\rangle]]$ and communication cost is given by the fact that the h-relation is 1 in each superstep but the last, where it is 0.

The cost of s_{scan} as a function of the number of processes can be obtained manually by rewriting the program as a recurrence relation. This relation is then solved to remove the recurrence. When executed with $\mathbf{p}$ processes, the loop is executed $\lceil \log_2 \mathbf{p} \rceil$ times, resulting in $\lceil \log_2 \mathbf{p} \rceil + 1$ supersteps. The largest local computation is performed by the process $\mathbf{p} - 1$, which performs the work $1\ \mathtt{w}$ in each superstep. The h-relation of each superstep but the last (which has no communication) is 1, since each process receives at most one value and sends at most one value. Thus, the cost of s_{scan} is given by the function

$$\lambda u. \begin{cases} \lceil \log_2 \mathbf{p} \rceil & \text{if } u = \mathtt{w} \\ \lceil \log_2 \mathbf{p} \rceil & \text{if } u = \mathtt{g} \\ \lceil \log_2 \mathbf{p} \rceil + 1 & \text{if } u = \mathtt{l} \end{cases}$$

which is parametric in the number of processes.

The next section describes our method for *automatically* obtaining bounds on the parallel cost of programs like s_{scan}.

5.3 Cost Analysis

This section describes the main contribution of this chapter: a method for transforming a parallel program to a sequential program amenable to the pre-existing sequential cost analysis. The transformation ensures that the worst-case parallel cost of the original program is retained. The transformation, summarized graphically in Figure 5.7, consists of 3 steps:

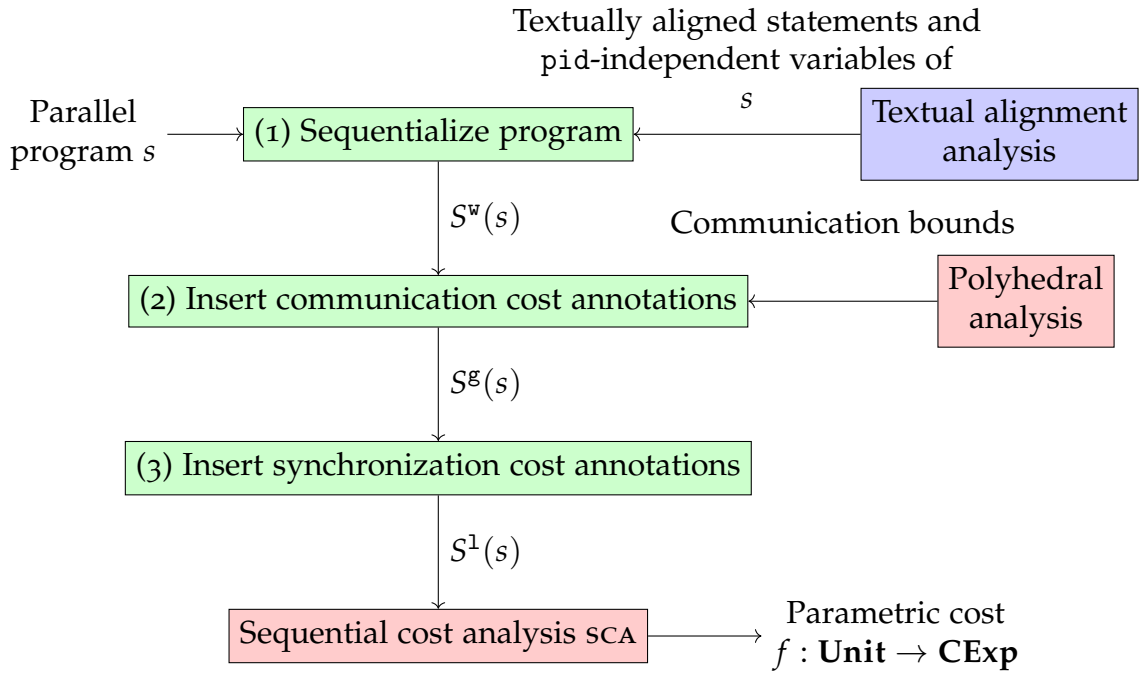


Figure 5.7 – *Parallel Cost Analysis pipeline. Green boxes are new contributions, blue is our previous work described in Chapter 4 and red are external dependencies.*

1. First, we verify that the input program s is textually aligned using the replicated synchronization analysis of Chapter 4. This property allows us to sequentialize s into the "sequential simulator" $S^{\mathtt{w}}(s)$.

2. Knowing the communication distribution of the input program is key to obtaining precise bounds on communication costs. The second step analyzes each communication primitive and surrounding control structures in the polyhedral model [62]. This allows us to obtain precise bounds on communication costs that are inserted as work annotations into the sequential simulator, obtaining $S^{\mathtt{g}}(s)$.

3. In the third step we insert annotations for counting the number of synchronizations into the sequential simulator, obtaining $S^{\mathtt{l}}(s)$.

Finally, we apply the sequential cost analysis SCA on the resulting sequential program $S^{\mathtt{l}}(s)$ to obtain the parametric parallel cost.

5.3.1 Sequential Simulator

This section describes the transformation of a **BSPlite** program $s \in$ **Par** into a "sequential simulator" $S^{\mathtt{w}}(s) \in$ **Seq** of s, such that all global work traces of s can be produced by $S^{\mathtt{w}}(s)$. To do this, we require that the input program has *textually aligned* synchronization, replace parallel primitives, with no counterpart in

Seq, with `skip` instructions, and assign non-deterministic values to all variables affected by the parallelism. This will allow us to use the sequential cost analysis to get an upper bound on the local computation cost on the parallel program.

Sequentialization

The sequentialization transforms the parallel program so that it non-deterministically chooses the identity and state of one local process before the execution of each superstep. The underlying cost analysis will return the cost of the worst-case choice, coinciding with the definition of the local computation cost of one superstep.

The transformation relies on the *textually aligned synchronization* of the input program. As we seen, this intuitively corresponds to all processes starting execution at the same source code location in the beginning of each superstep, and then synchronizing at the same source code location at the end of the superstep. In this case we can repeatedly apply the non-deterministic identity in the beginning of each superstep.

The non-deterministic identity switch may switch to an identity that does not correspond to any feasible state of a local process. To restrict the non-determinism and improve precision, the identify switch does not modify the variables that are `pid`-independent. This set is identified by the textual alignment analysis.

Textual Alignment Analysis

We reuse the textual alignment analysis of Chapter 4 to statically under-approximate the set of `pid`-independent variables and the set of textually aligned statements. This also serves to verify that synchronization is textually aligned. We refer to the textual alignment analysis as RS, and treat it as a black box of the following functionality:

$$\textsc{rs} : \mathbf{Par} \to (\{\top\} \cup (\mathcal{P}(\mathbf{Lab}) \times (\mathbf{Lab} \to \mathcal{P}(\mathbf{Var}))))$$

If a program s can be statically verified to have textually aligned synchronization then $\textsc{rs}(s) = (\tau, \pi)$ where τ and π are under-approximations of textually aligned statements and the set of `pid`-independent variables at each program point, respectively.

If the analysis RS cannot verify statically the textual alignment of the program, then $RS(s) = \top$. In this case, the parallel cost analysis cannot move forward and

returns $\lambda u.\omega$. In the remainder of this chapter we assume that programs have statically verified textually aligned synchronization.

Example 4. *Consider the program s_{scan} in Figure 5.5. The textual alignment analysis gives:*

$$\begin{array}{lcl} RS(s_{scan}) & = & (\tau_{scan}, \pi_{scan}) \\ \tau_{scan} & = & \{1,2,3,5,6,8\} \\ \pi_{scan}(\ell) & = & \begin{cases} \{i, x, x_{in}\} & \text{if } \ell = 1 \\ \{i\} & \text{otherwise} \end{cases} \end{array}$$

The program has textually aligned synchronization, since all statements in this program are textually aligned except the statements labeled 4 and 7, corresponding to the body of the conditionals in the loop. The body of these conditionals will not be executed by all processes and this is statically detected since the value of the guard conditions depends on the `pid` *variable. The variables assigned at these statements and the variables affected by communication, namely x and x_{in}, will not be* `pid`*-independent at any statement reachable by these assignments and communications. However, i has no dependency on* `pid` *and so is* `pid`*-independent throughout the program.*

Sequential Simulator

The sequential simulator $S^{\mathtt{w}}(s)$ of a parallel program s with textually aligned synchronization is obtained by assigning a non-deterministic value to all variables that are not `pid`-independent (including *pid* itself) after each `sync` primitive, and then replacing all parallel primitives (`sync`, `get` and `put`) by a `skip` with the same label.

We first define the function *havoc* that creates a program that assigns a non-deterministic value to each variable given as argument:

$$\begin{array}{c} \mathit{havoc} : \mathcal{P}(\mathbf{Var}) \to \mathbf{Seq} \\ \mathit{havoc}(V) = (;)\ \{[x{:=}\mathtt{any}]^{\ell'} \mid x \in V\} \end{array}$$

where $(;)$ gives a sequential composition of a set of statements and ℓ' is a fresh label for each assignment.

Now assume $RS(s) = (\tau, \pi)$, that $\mathbf{Var}_s$ is the set of variables used in s and

again let ℓ' be a fresh label. Then $S^{\mathtt{w}}(s)$ is defined:

$$
\begin{array}{lcl}
S^{\mathtt{w}}, S' & : & \mathbf{Par} \to \mathbf{Seq} \\
S^{\mathtt{w}}(s) & = & [pid{:=}[0\,..\,nprocs-1]]^{\ell'}; S'(s) \\
S'(s') & = & \begin{cases}
[\mathtt{skip}]^{\ell} & \text{if } s' \in \{\mathtt{put}^{\ell}(e_1,e_2,x), \mathtt{get}^{\ell}(e_1,y,x)\} \\
[\mathtt{skip}]^{\ell}; havoc((\mathbf{Var}_s \cup \{pid\}) \setminus \pi(\ell)) & \text{if } s' = [\mathtt{sync}]^{\ell} \\
S'(s_1); S'(s_2) & \text{if } s' = s_1; s_2 \\
\mathtt{if}\ [E^{\mathtt{w}}(b)]^{\ell}\ \mathtt{then}\ S'(s_1)\ \mathtt{else}\ S'(s_2)\ \mathtt{end} & \text{if } s' = \mathtt{if}\ [b]^{\ell}\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end} \\
\mathtt{while}\ [E^{\mathtt{w}}(b)]^{\ell}\ \mathtt{do}\ S'(s_1)\ \mathtt{end} & \text{if } s' = \mathtt{while}\ [b]^{\ell}\ \mathtt{do}\ s_1\ \mathtt{end} \\
\{E^{\mathtt{w}}(e)\ u\}\ S'(s_1) & \text{if } s' = \{e\ u\}\ s_1 \\
[x{:=}E^{\mathtt{w}}(e)]^{\ell} & \text{if } s' = [x{:=}e]^{\ell} \\
[\mathtt{skip}]^{\ell} & \text{if } s' = [\mathtt{skip}]^{\ell}
\end{cases}
\end{array}
$$

As the `nprocs` and `pid` primitives do not exists in the expressions of **Seq**, we use the function $E^{\mathtt{w}}$ that replaces occurrences of `nprocs` with the variable *nprocs*, and `pid` with the variable *pid* in arithmetic and boolean expressions. The definition of this function is trivial and omitted. We assume that input programs do not manipulate the variables *pid* and *nprocs*. The variable *nprocs* is not assigned in the sequentialized program. This ensures that the sequentialized program is analyzed for an undetermined number of processes, and that the resulting cost function is parametric in **p**.

Intuitively, the sequential simulator will act as any process of the parallel program and will have the same series of values for `pid`-independent variables. For variables that are not `pid`-independent, it switches to any value after each synchronization using a non-deterministic assignment. In this way, the sequential simulator can assume the identity of any process at the beginning of each superstep and produce any global trace. This is formalized by the following conjecture:

Conjecture 1. For any $\mathbf{p} > 0$ and $s \in \mathbf{Par}$ such that $\textsc{rs}(s) = (\tau, \pi)$, and $\sigma \in \mathbf{State}$, if $\langle \langle s \rangle_i, \langle \sigma \rangle_i \rangle \longrightarrow^S \langle E, W, R \rangle$ then for all $w \in \{ \mathbin{+\!\!+} T \mid T \in G(W) \}$, $\exists \sigma', \langle S^{\mathtt{w}}(s), \sigma[nprocs \leftarrow \mathbf{p}] \rangle \to \langle \sigma', w \rangle$.

Note that the parallel program and its sequential simulator execute the same sequence of textually aligned instructions. That is, when executed with the same initial environment, the sequences of executed instructions of both programs will coincide after removing all labels that are not in τ.

$S^{\mathtt{w}}(s_{\text{scan}}) =$
  $[pid{:=}[0\,..\,nprocs-1]]^9$;
  $[i{:=}1]^1$;
  `while` $[i < nprocs]^2$ `do`
    `if` $[pid \geq i]^3$ `then`
      $[\mathtt{skip}]^4$
    `end`;
    $[\mathtt{skip}]^5$;
    $[pid{:=}[0\,..\,nprocs-1]]^{10}$; $[x{:=}\mathtt{any}]^{11}$; $[x_{in}{:=}\mathtt{any}]^{12}$;
    `if` $[pid \geq i]^6$ `then`
      $\{1\ \mathtt{w}\}\ [x{:=}x + x_{in}]^7$
    `end`;
    $[i{:=}i \times 2]^8$
  `end`

Figure 5.8 – *Sequential simulator* $S^{\mathtt{w}}(s_{scan})$

Obtaining the Local Computation Cost

As an immediate consequence of the previous conjecture the simulator also produces the maximum global work trace. Thus, we can now obtain an upper bound on the parallel cost of the computation of a program s by applying SCA to its sequential simulator:

Conjecture 2. For any $\mathbf{p} > 0$ and $s \in \mathbf{Par}$ such that $\text{RS}(s) = (\tau, \pi)$, and $\sigma \in \mathbf{State}$, if $\langle\langle s\rangle_i, \langle\sigma\rangle_i\rangle \longrightarrow^S \langle E, W, R\rangle$ then for all $u \in \mathbf{Unit} \setminus \{\mathtt{g}, \mathtt{l}\}$ we have $Cost_{\text{PAR}}(W, R)(u) \leq \mathcal{C}[\![\text{SCA}(S^{\mathtt{w}}(s))(u)]\!]\,\sigma[nprocs \leftarrow \mathbf{p}]$.

The non-determinism introduced by the sequential simulator potentially renders the obtained upper bound imprecise. However, we conjecture that the variables that have most influence on cost, namely those affecting control flow, are also those that are `pid`-independent and thus this imprecision should have limited influence on the upper bound. Indeed, this is true for data-oblivious programs, whose communication pattern does not depend on the communicated data, as our evaluation in Section 5.4 shows.

Example 5. *See Figure 5.8 for the sequential simulator* $S^{\mathtt{w}}(s_{scan})$*. Note the non-deterministic assignments to pid,* x *and* x_{in} *after the former synchronization at Label 5, and how the* `sync` *and* `get` *at Labels 4 and 5 have been replaced by* `skip`*. The effect of the former* `get` *is simulated by the non-deterministic update of* x_{in} *after the former* `sync` *at Label 5.*

5.3.2 Analyzing Communication Costs

The second transformation inserts an annotation $\{e\ \mathtt{g}\}$ for each communication primitive s in the simulator. This makes the underlying sequential cost analysis account for communication cost. The expression e must be an upper bound on the addition to the total communication cost of any processes executing s. Without further semantic analysis of the parallel control flow, we must assume that all processes execute the primitive, even if only a subset of them actually do so, and without knowing the exact value in all processes of the first (target) expression of the `put` or `get`, we must also assume that the communication is unbalanced, and thus more costly.

For instance, see the communication primitive at program point 4 in the program s_{scan} guarded by the conditional at program point 3. Without any semantic knowledge about the target expression $\mathtt{pid} - i$ and the guarding condition $\mathtt{pid} \geq i$, one must assume the worst-case addition of $\mathbf{p}\ \mathtt{g}$ to the program's total communication cost, obtained when all processes execute the `get` with the same target (for instance, when $i = \mathtt{pid}$). However, by knowing that i has the same value on all processes in each execution of this `get`, one can deduce that the target expression refers to one distinct process for each process executing the `get`, and thus a tighter bound of $1\ \mathtt{g}$ can be obtained.

Polyhedral Communicating Sections

This reasoning is automated by representing the communication primitive and surrounding code, called the *communicating section*, in the polyhedral model [20]. In this model, each execution of a statement that is nested in a set of loops and conditionals is represented as an integer point in a n-polyhedron, where n is the number of loops. A n-polyhedron is a set of points in $\mathbf{Int}^n$ vector space that is bounded by affine inequalities:

$$\mathcal{D} = \{\mathbf{x} \in \mathbf{Int}^n \mid A\mathbf{x} + \mathbf{a} \geq \mathbf{0}\}$$

The vector $\mathbf{x}$ corresponds to the loop iterators. Thus each point in the polyhedron corresponds to one valuation of the loop iterators. A is a constant matrix. The constant vector $\mathbf{a}$ can contain program variables not in $\mathbf{x}$ that are constant in the section, called *parameters*. This model requires that all loop bounds, iterator updates as well as conditionals in the section can be represented as affine combinations of loop iterators and parameters.

For the communicating section, our analysis adds two additional variables

```
s_scan =
  [i:=1]^1;
  while [i < nprocs]^2 do
     if [pid ≥ i]^3 then
        get^4(pid − i, x, x_in)
     end;
     [sync]^5;
     if [pid ≥ i]^6 then
        {1 w} [x:=x + x_in]^7
     end;
     [i:=2 ∗ i]^8
  end
```

Figure 5.9 – *The program s_{scan}, recalled*

s and d to **x**, corresponding to the `pid` of the source and destination process. The communicating section of a communication primitive is *analyzable* if the section's entry point is textually aligned, all its parameters are `pid`-independent and it does not contain a `sync`. We also require that the communication primitive has a target expression that is an affine combination of loop iterators

Finding sections of the code that is amenable to polyhedral representation is the subject of "polyhedral extraction" [195], which is outside the scope of this work. In our prototype, a simple algorithm is used to find the largest analyzable communicating section around each communication primitive. From this approach stems the requirement that programs are structured. The algorithm consists of starting with a communicating section containing only the communication primitive. Then, it adds as many contiguous statements around the section as possible, until adding another would make it no longer analyzable. If the section is nested in a loop, respectively a conditional, the algorithm attempts to include the whole body, respectively both branches, if possible to do so and keep the section analyzable. The algorithm continues thus recursively, until the section can no longer grow. But, the method used for polyhedral extraction is orthogonal to our contribution, and hence being structured is not an inherent limitation of our approach.

Interaction Sets

From a polyhedral communicating section the analysis obtains a symbolic representation of the exact set of communication requests that would be generated if it is executed by any number of processes **p**, called the *interaction set* [62]. This is not a literal set, but instead, a parameterized symbolic expression for which each valuation of the parameters gives rise to a set.

From a communication primitive $\texttt{put}^{\ell}(e_1, e_2, y)$, whose communicating section consists of n loops with the loop iterators $x_1, \ldots, x_n$, each with lower and upper bounds $L_1, \ldots, L_n$ and $U_1, \ldots, U_n$, and a set of guard expressions $C \subseteq \mathbf{BExp}_p$ from the conditionals, the analysis constructs the interaction set:

$$\mathcal{D} = \{[s, d, x_1, \ldots, x_n] \in \mathbf{Int}^{n+2} \mid 0 \leq s < \mathbf{p} \wedge d = e_1 \wedge \bigwedge_{k \in 1 \ldots n} L_k \leq x_k \leq U_k \wedge \bigwedge C\}$$

For $\texttt{get}^{\ell}(e_1, y, z)$ in an identical communicating section, the analysis constructs the interaction set:

$$\mathcal{D} = \{[s, d, x_1, \ldots, x_n] \in \mathbf{Int}^{n+2} \mid s = e_1 \wedge 0 \leq d < \mathbf{p} \wedge \bigwedge_{k \in 1 \ldots n} L_k \leq x_k \leq U_k \wedge \bigwedge C\}$$

In both cases, $[s, d, i_1, \ldots, i_n] \in \mathcal{D}$ means that process s will send data to d at the end of the superstep and that the loop iterators $x_1, \ldots, x_n$ have the values $i_1, \ldots, i_n$ when the communication primitive is executed. Any variables of the target expression e_1 of the communication primitives must be part of the parameters, as indicated above. These parameters have no intrinsic value. Instead each valuation of the parameters gives rise to an instance of the interaction set. We handle the interaction sets in this parameterized, symbolic form. This allows us to obtain a symbolic expression for the size of the interaction set, from which communication costs are derived. As the expression is symbolic, we obtain the size *as a function* of the parameters, which will simplify integrating the results of the polyhedral analysis in the rest of the cost analysis.

The constraints here are given as a conjunction, the transformation to the matrix inequalities representation is standard [20].

Example 6. *The analysis automatically extracts the polyhedron $\mathcal{D}_S$ representing the interaction set generated by the communicating section from program points* 3 *and* 4 *of s_{scan} (see Figure 5.9). A larger communicating section cannot be extracted without including the synchronization primitive of program point* 5*, which would render the communicating section non-analyzable. The constraints of $\mathcal{D}_S$ are shown first a boolean formula, then as the equivalent inequalities.*

```
...
if [pid ≥ i]^3 then
    get^4(pid − i, x, x_in)
end
...
```

$$\begin{aligned}\mathcal{D}_S &= \{[s,d] \in \mathbf{Int}^2 \mid s = d - i \wedge 0 \leq d < \mathbf{p} \wedge d \geq i\} \\ &= \left\{ \begin{pmatrix} s \\ d \end{pmatrix} \;\middle|\; \begin{pmatrix} s \\ d \end{pmatrix} \in \mathbf{Int}^2, \begin{bmatrix} 1 & -1 \\ 1 & 1 \\ 0 & 1 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} s \\ d \end{pmatrix} + \begin{pmatrix} -i \\ i \\ 0 \\ \mathbf{p}-1 \\ -i \end{pmatrix} \geq \mathbf{0} \right\}\end{aligned}$$

The two variables s and d of $\mathcal{D}_S$ respectively correspond to the identifier of the source respectively destination process of each request. This set is parameterized by the variable i, which is constant in the section, and the BSP parameter $\mathbf{p}$. The constraints are given by the target expression ($s = d - i$), the domain of the `pid` *variable ($d \geq 0$ and $\mathbf{p} - 1 \geq d$), and the condition on program point 3 ($d \geq i$).*

Example 7. *Some common communication patterns and the interaction sets the analysis obtains from these are illustrated in Figure 5.10.*

From Interaction Sets to *h*-relations

From the interaction set, the analysis extracts an upper bound on the section's addition to the total communication cost of the execution, which is inserted as an annotation at the section's entry. This is done by creating two relations from the interaction set $\mathcal{D}$: from process identifier to the set of outbound ($\mathcal{D}^+$) respectively inbound ($\mathcal{D}^-$) communication requests. The h-relation of this section is the largest of the upper bounds on the cardinality of the image of these relations. This is expressed by $\mathcal{H}$:

$$\mathcal{D}^+(i) = \{[s,d,\ldots] \in \mathcal{D} \mid s = i\} \qquad \mathcal{D}^-(i) = \{[s,d,\ldots] \in \mathcal{D} \mid d = i\}$$

$$\mathcal{H} = \max(\max_{i=0}^{\mathbf{p}-1} \#(\mathcal{D}^+(i)), \max_{i=0}^{\mathbf{p}-1} \#(\mathcal{D}^-(i)))$$

Implementation of Communication Analysis

The analysis uses `isl` [194] to create the interaction set $\mathcal{D}$ as described earlier and the two relations $\mathcal{D}^+$ and $\mathcal{D}^-$ using `isl`'s operations for creating relations and sets. It then asks `isl` to compute the expression corresponding to $\mathcal{H}$, which it does using integer volume counting techniques [196].

One-to-one	One-to-all	All-to-one	All-to-all

`if` $[\texttt{pid} = src]^1$ `then`
  $\texttt{put}^2(dest, e, x)$
`end`
	`if` $[\texttt{pid} = src]^1$ `then`
	  $[i{:=}0]^2$;
	  `while` $[i < \texttt{nprocs}]^3$ `do`
	  $\texttt{put}^4(i, e, x)$;
	  $[i{:=}i+1]^5$
	  `end`
	`end`
		$\texttt{put}^1(dest, e, x)$	$[i{:=}0]^1$;
			`while` $[i < \texttt{nprocs}]^2$ `do`
			  $\texttt{put}^3(i, e, x)$;
			  $[i{:=}i+1]^4$
			`end`

Pattern	Interaction set	h-relation
One-to-one	$\mathcal{D} = \{[s,d] \in \mathbf{Int}^2 \mid 0 \leq s < \mathbf{p} \wedge d = dest \wedge s = src\}$	1
One-to-all	$\mathcal{D} = \{[s,d,i] \in \mathbf{Int}^3 \mid 0 \leq s < \mathbf{p} \wedge d = dest \wedge 0 \leq i < \mathbf{p} \wedge s = src\}$	$\mathbf{p}$
All-to-one	$\mathcal{D} = \{[s,d] \in \mathbf{Int}^2 \mid 0 \leq s < \mathbf{p} \wedge d = dest\}$	$\mathbf{p}$
All-to-all	$\mathcal{D} = \{[s,d,i] \in \mathbf{Int}^3 \mid 0 \leq s < \mathbf{p} \wedge d = i \wedge 0 \leq i < \mathbf{p}\}$	$\mathbf{p}$

Figure 5.10 – *Common communication patterns, their corresponding interaction set and statically inferred h-relation.*

Example 8. *For the interaction set $\mathcal{D}_S$ from the example s_{scan}, this technique obtains the h-relation* 1*. The analysis inserts this bound before the* `if` *statement at program point* 5 *in the sequential simulator of s_{scan} (see Figure 5.11). Figure 5.10 contains common communication patterns and upper bounds extracted from their interaction sets using* `isl`*.*

Discussion

This method requires no pattern matching and automatically extracts a precise upper bound on the communication cost of analyzable any communicating section. When this is not the case, we fall back on the conservative but sound upper bound cost of **p** g, which is added as an annotation to the communication primitive in the sequential simulator.

Soundness

The following conjecture states that the sequential simulator with communication bounds soundly bounds from above the cost of the parallel program:

Conjecture 3. For any $\mathbf{p} > 0$ and $s \in \mathbf{Par}$ such that $\textsc{rs}(s) = (\tau, \pi)$, and any environment $\sigma \in \mathbf{State}$ and $\langle \langle s \rangle_i, \langle \sigma \rangle_i \rangle \longrightarrow^S \langle E, W, R \rangle$ then

$$Cost_{\textsc{par}}(W, R)(\mathrm{g}) \leq \mathcal{C}[\![\textsc{sca}(S^{\mathrm{g}}(s))(\mathrm{g})]\!]\,\sigma[nprocs \leftarrow \mathbf{p}].$$

5.3.3 Analyzing Synchronization Costs

Since we require that synchronization primitives are textually aligned in s, it suffices to annotate each instruction that was `sync` in the original program with $\{1\ \mathtt{l}\}$ in the sequential simulator $S^{\mathrm{g}}(s)$ to account for synchronization costs. We also add an annotated dummy `skip` instruction to the end of the program to account for the implicit synchronization barrier at the end of all executions. We refer to the resulting program as $S^{\mathtt{l}}(s)$.

Any execution of the parallel program evaluates the same sequence of textually aligned statements as the sequential simulator does on the same initial environment. Thus, the simulator will evaluate exactly as many annotations of unit `l` as there are synchronizations in the execution of the parallel program. This intuition is formalized by the following conjecture:

$S^{\mathtt{l}}(s_{\text{scan}}) =$
  $[pid{:=}[0\,..\,nprocs-1]]^{9}$;
  $[i{:=}1]^{1}$
  `while` $[i < nprocs]^{2}$ `do`
    $\{1\ \mathtt{g}\}$
    `if` $[pid \geq i]^{3}$ `then`
      $[\mathtt{skip}]^{4}$
    `end`
    $\{1\ \mathtt{l}\}$ $[\mathtt{skip}]^{5}$
    $[pid{:=}[0\,..\,nprocs-1]]^{10}$; $[x{:=}\mathtt{any}]^{11}$; $[x_{in}{:=}\mathtt{any}]^{12}$;
    `if` $[pid \geq i]^{6}$ `then`
      $\{1\ \mathtt{w}\}$ $[x{:=}x + x_{in}]^{7}$
    `end`
    $[i{:=}i \times 2]^{8}$
  `end`
  $\{1\ \mathtt{l}\}$ $[\mathtt{skip}]^{13}$

Figure 5.11 – *Sequential simulator $S^{\mathtt{l}}(s_{scan})$, with annotations for communication bounds and synchronization costs*

Conjecture 4. For any $\mathbf{p} > 0$ and $s \in \mathbf{Par}$ such that $\textsc{rs}(s) = (\tau, \pi)$, and $\sigma \in \mathbf{State}$ any environment, and $\langle \langle s \rangle_i, \langle \sigma \rangle_i \rangle \longrightarrow^{S} \langle E, W, R \rangle$. Then

$$Cost_{\textsc{par}}(W, R)(\mathtt{l}) \leq \mathcal{C}[\![\textsc{sca}(S^{\mathtt{l}}(s))(\mathtt{l})]\!]\,\sigma[nprocs \leftarrow \mathbf{p}].$$

Example 9. *The sequential simulator $S^{\mathtt{l}}(s_{scan})$ in Figure 5.11 is obtained by adding the communication bounds found in Section 5.3.2 to the conditional at Label 3, and annotating the* `sync` *at Label 5, as well as adding the dummy* `skip` *at Label 13 to account for the synchronization barrier terminating the execution.*

We can now submit the simulator $S^{\mathtt{l}}(s_{scan})$ to the sequential cost analyzer. The obtained cost is exactly the one obtained earlier by manual analysis, i.e.:

$$\textsc{sca}(S^{\mathtt{l}}(s_{scan})) = \lambda u. \begin{cases} \lceil \log_2 \mathbf{p} \rceil & \text{if } u = \mathtt{w} \\ \lceil \log_2 \mathbf{p} \rceil & \text{if } u = \mathtt{g} \\ \lceil \log_2 \mathbf{p} \rceil + 1 & \text{if } u = \mathtt{l} \end{cases}$$

5.3.4 Time Complexity of Analysis

We treat the time for sequentialization and communication analysis (T_{seq}) separately from the final sequential cost analysis ($T_{\textsc{sca}}$):

$$T_{analysis}(e, v) = T_{seq}(e, v) + T_{\textsc{sca}}(e, v)$$

Here, e is the number of edges of the program's control flow graph (which is proportional to its size) and v the number of variables of the program.

Sequentialization is done in linear time but uses the result of a data-flow analysis, which is computed in time bounded by $O(ev)$ [151]. The analysis time of each communication primitive is polynomial in the size of the polyhedra representing it [196], which in turn is bounded by the maximum nesting level of the program. The latter is often assumed to be bounded by some constant for realistic programs. Hence, T_{seq} is bounded by some polynomial.

Analysis time for the sequentialized program, T_{SCA}, depends on the details of the implementation of SCA. Our implementation translates the input program into "cost relations" [7]. This step involves a data-flow analysis bounded by $O(ev)$ and an abstract interpretation in the domain of convex polyhedra that is linear in e but exponential in the maximum number of variables in any scope [49].

Finally, the cost relations are solved into a closed form upper bound by PUBS [7] which is done in a time exponential in their bit size (Genaim, personal communication, 2017).

In sum, $T_{analysis}$ grows exponentially with the size of the program. This is due to our specific implementation of SCA that uses PUBS: another sound SCA with lower complexity could be used. Note that the analysis complexity only depends on the size of the analyzed program and is independent on run-time parameters such as the number of processors executing the program.

5.4 Implementation and Evaluation

A prototype of the analysis has been implemented for **BSPlite** programs in 3 KLOCs of Haskell. The underlying sequential cost analysis SCA is implemented as described in [6] and uses APRON [115] for abstract interpretation and PUBS [7] for solving cost equations. The polyhedral analysis of communicating sections uses `isl` [194].

We have performed two evaluations of the static upper bounds of the parallel cost given by the implementation on 8 **BSPlite** benchmarks. The first evaluates that they are indeed upper bounds and by what margin. The second evaluates the quality of their power to predict actual run times in seconds. While finding exact *Worst-Case Execution Times* [201] is not our goal, we demonstrate how BSP costs relate to concrete run times.

5.4.1 Benchmarks

Table 5.1 summarizes the benchmarks, their static bounds and analysis run times. The second column indicates whether the program's control flow is independent of the contents of the input arrays. We call such programs "data-oblivious", and when it is not the case, "data-dependent". Note that no attempt has been made to optimize the run time of the prototype. The benchmarks are written in a variant of **BSPlite** (Section 5.2), extended with arrays. Array contents are treated as non-deterministic values by the implementation. The benchmarks are inner product (`BspIp`); parallel prefix in logarithmic and constant number of supersteps (`Scan2` and `ScanCst2`); parallel reduction (`BspFold`); array compression (`Compress`); broadcast in one and logarithmic number of supersteps (`Bcast1` and `BcastLog`); and 2-phase broadcast (`Bcast2ph`).

Local computation is defined by work annotations added to costly array operations in loops. For simplicity, we only use the unit `w` and thus omit the normalization function w. The static bounds obtained by the analysis on local computation, communication and synchronization are given in the columns $W^\sharp$, $H^\sharp$, and $S^\sharp$ of Table 5.1 respectively. Benchmarks and static bounds are parameterized by BSP parameters and input sizes N.

5.4.2 Symbolic Evaluation

We verify whether that the static bounds are indeed bounds, and evaluate their precision by executing each benchmark in an interpreter simulating $\mathbf{p} = 16$. The interpreter is instrumented to return the parallel cost (as defined in Section 5.2.3) of each execution.

We found that the static bound is equal to the cost of each execution, except for the communication cost of the program `Compress`, which is overestimated by a factor of $\mathbf{p}$. The communication distribution of `Compress` depends on the values in the input array. The implementation treats these as non-deterministic values, and returns the pessimistic static bound $N\mathrm{g}$ on communication instead of the tighter bound $N/\mathbf{p}\mathrm{g}$ which can be found by analyzing the program manually.

5.4.3 Concrete Evaluation

To evaluate the quality of the static bounds' capacity to predict actual run times in seconds, we translate the benchmarks from **BSPlite** to C with BSPlib and com-

Benchmark (LOC)	Data-oblivious control flow	Statically inferred upper bound on parallel cost $W^\sharp(N)$	$H^\sharp(N)$	$S^\sharp(N)$	Analysis time
`BspIp` (9)	Yes	$(2N/\mathbf{p}+\mathbf{p})\mathtt{w}$	$\mathbf{p}\mathtt{g}$	$2\mathtt{l}$	1.09s
`Scan2` (16)	Yes	$(5N/\mathbf{p}+\log_2\mathbf{p}-3)\mathtt{w}$	$(\log_2\mathbf{p})\mathtt{g}$	$(\log_2\mathbf{p}+1)\mathtt{l}$	0.92s
`ScanCst2` (11)	Yes	$(5N/\mathbf{p}+\mathbf{p})\mathtt{w}$	$(\mathbf{p}-1)\mathtt{g}$	$2\mathtt{l}$	1.25s
`BspFold` (9)	Yes	$(N/\mathbf{p}+\mathbf{p}-2)\mathtt{w}$	$\mathbf{p}\mathtt{g}$	$2\mathtt{l}$	0.82s
`Compress` (19)	No	$(3N/\mathbf{p}+\mathbf{p}-2)\mathtt{w}$	$N\mathtt{g}$	$3\mathtt{l}$	2.13s
`Bcast1` (5)	Yes		$(\mathbf{p}-1)N\mathtt{g}$	$2\mathtt{l}$	0.63s
`BcastLog` (8)	Yes		$(\log_2\mathbf{p})N\mathtt{g}$	$(\log_2\mathbf{p}+1)\mathtt{l}$	0.61s
`Bcast2ph` (11)	Yes		$2(\mathbf{p}-1)N/\mathbf{p}\mathtt{g}$	$3\mathtt{l}$	1.16s

Table 5.1 – *Summary of benchmarks, static upper bounds of their parallel costs and analysis times*

pare their run times on two different parallel environments with those predicted by the static bounds in the BSP model.

Making such predictions is inherently difficult, especially when several translations are involved. For instance, our model supposes that the execution of one individual operation takes a fixed amount of time. In reality, the time taken depends on the state of caches, pipelines, and other hardware features. It also depends on optimizations applied by the compiler. Another issue in the model is the network. BSP assumes that the communication bottleneck will be at the end points and, thus, that the time to deliver an h-relation will scale linearly. However, this is not true for current multi-core architectures, which usually have tree-based network topologies, where bottlenecks can occur near the root. All considering, at best we can hope to obtain run time predictions that are not too far from the actual run time, but they may still be several factors off.

The first evaluation environment is a desktop computer with an 8-core, 3.20 GHz, Intel Xeon CPU E5-1660 processor, 32 GB RAM, and running Ubuntu 16.04. We use `gcc` 5.4.0. The second environment is an 8-node Intel Sandy Bridge cluster connected by FDR InfiniBand network cards. Each node has 2 Intel E5-2650 2 GHz CPUs with 8 cores each, 384 GB RAM and is running CentOS 7.2. Here we use `gcc` 6.1.0. We use a Huawei-internal BSPlib implementation in both environments.

The same method is used to obtain the BSP parameters of both environments. We modify `bspbench` to measure **r** as memory speed, which is the bottleneck in all the benchmarks. To obtain **g** and **l** we measure the minimum time taken to deliver all-to-all h-relations of size **p**, 2**p**, and h_{max}**p** over a large set of samples, where h_{max} is the size of the largest h-relation performed in the benchmarks. Then the y-intercept of the line passing through the first two data-points is taken as **l**, and the slope of the line passing through the last two is taken as **g**. Example BSP parameters are given above Figures 5.12 to 5.14.

We find that the run time of all benchmarks grow linearly with input size as predicted by the static bounds. See e.g. Figure 5.12. However, the static bounds do not always accurately predict the run times. See e.g. Figure 5.14.

We calculate the error in prediction using the formula $\mid T_{measured} - T_{predicted} \mid / \min(T_{measured}, T_{predicted})$. In this formulation, an overestimation of running time by a factor 2 as well as an underestimation by a factor 2 will correspond to an error of 100% [120]. The largest error factors for each environment-benchmark combination are summarized in Table 5.2. The large errors in predictions for `Compress` are explained by the inaccuracy of its statically found bound. For

Benchmark	Desktop $\mathbf{p}=2$	Desktop $\mathbf{p}=4$	Desktop $\mathbf{p}=8$	Cluster $\mathbf{p}=8$	Cluster $\mathbf{p}=128$	Worst prediction for cluster, $\mathbf{p}=8$ N	predicted	actual
`BspIp`	14.09%	7.98%	33.78%	15.49%	41.14%	$1.68 \cdot 10^8$	0.12s	0.10s
`Scan2`	10.64%	16.79%	47.70%	34.63%	25.26%	$1.68 \cdot 10^8$	0.29s	0.22s
`ScanCst2`	10.72%	16.85%	47.77%	35.36%	42.38%	$1.26 \cdot 10^8$	0.22s	0.16s
`BspFold`	11.86%	4.46%	39.72%	12.84%	48.76%	$1.68 \cdot 10^8$	0.12s	0.10s
`Compress`	396.78%	984.88%	2,449.87%	1,311.37%	15,388.60%	$4.8 \cdot 10^5$	0.12s	0.01s
`Bcast1`	44.11%	90.83%	153.11%	47.75%	372.52%	$1.6 \cdot 10^5$	0.03s	0.02s
`BcastLog`	35.41%	59.50%	101.04%	13.03%	31.71%	$1.6 \cdot 10^5$	0.01s	0.01s
`Bcast2ph`	32.84%	65.04%	97.08%	23.30%	48.58%	$1.92 \cdot 10^5$	0.01s	0.01s

Table 5.2 – *Maximal error in predictions per environment and benchmark. Sample times and predictions are for Cluster,* $\mathbf{p}=8$.

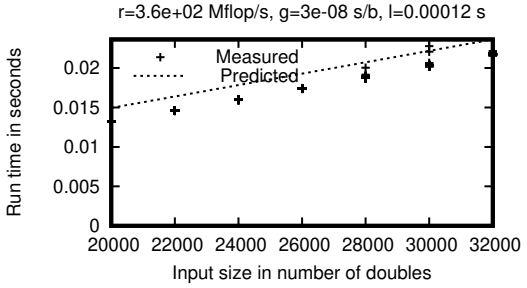


Figure 5.12 – `BcastLog` *on Cluster,* $\mathbf{p}=8$

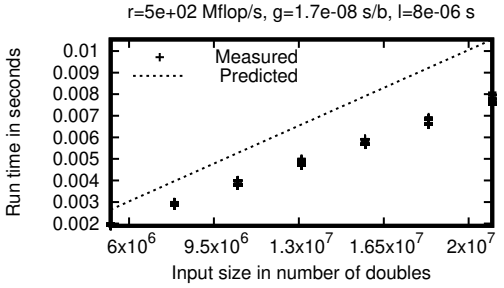


Figure 5.13 – `BspFold` *on Desktop,* $\mathbf{p}=8$

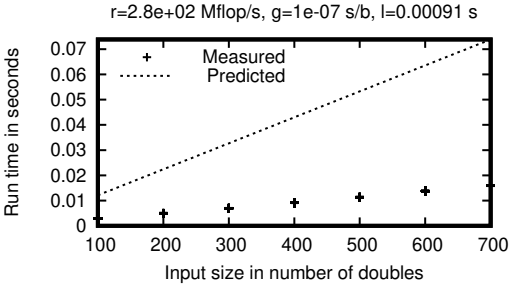


Figure 5.14 – `Bcast1` *on Cluster,* $\mathbf{p}=128$

the remaining benchmarks, error factors range from 4.46% for `BspFold` on the desktop with 4 processes to 372.5% for `Bcast1` on the cluster with 128 processes.

Indeed, `Bcast1` has the worst predictions of the data-oblivious benchmarks. This shows that in the considered environments, the communication pattern of this benchmark (one-to-all) is faster than the one used to estimate **g** (all-to-all). The discrepancy is even greater in the cluster with $\mathbf{p} = 128$ as a consequence of the cluster's hierarchical topology. The 128 processes correspond to 8 cluster nodes with 16 cores each, but the InfiniBand network is not 16 times faster than the internal node communication. Thus when only one process communicates with all other processes, it has much more bandwidth at its disposal then when all processes communicate outside the node. The former case corresponds to the communication pattern of `Bcast1`, and the latter to how the parameter **g** (which is an estimation of the full bisection bandwidth) is measured, explaining the difference between measured and predicted running time. The discrepancy of the other broadcast benchmarks, `BcastLog` and `Bcast2ph`, can also be explained by considering the topology of the networks and the communication patterns involved.

5.4.4 Conclusion of Evaluation

We find that (1) the static bounds of the implementation are indeed upper bounds of the parallel cost of all evaluated executions; (2) they are exact for data-oblivious benchmarks, but pessimistic for the one benchmark considered with data-dependent communication distribution; (3) the static bounds predict asymptotic behavior, and when tight static bounds are found, they accurately predict actual run times: the error is less than 50% for networks with full bisection bandwidth and for the others the error is never more than the ratio between the fastest link and the bisection bandwidth.

5.5 Related Work

Static cost analyses, as surveyed in Section 3.3.3, has previously been proposed for Resource Aware ML [105], for concurrent and distributed programs with dynamic task spawning [9], and for functional programs with divide-and-conquer parallelism [213]. But, to the best of our knowledge, no previous work exists on the automatic cost analysis of imperative BSP programs.

Closest to our work is Hayashis's cost analysis for shapely skeletal BSP programs [98]. However, this analysis is restricted to functional programs composed of parallel skeletons with *a priori* known cost functions, whereas our analysis deduces cost functions automatically.

Sequentialization has been used widely in the analysis and verification of parallel programs. We mention two examples, in the context of deductive verification. The Frama-C plugin Conc2Seq [24] uses sequentialization to verify concurrent C programs. The memory context of each process is simulated by translating each local variable to a **p**-vector. This vector maps process identifiers to the corresponding process's variable contents. A separate program counter is maintained per process. Atomic instructions are translated to functions. A global loop acts as a driver of the simulation, non-deterministically selecting a process and, depending on the instruction it should execute next, calls the corresponding function. This approach is aimed at fine-grained parallelism. For coarse-grained parallel programs, that consist of large atomic sections like in BSP, it is possible to create a less intrusive sequentialization. A translation that preserves more of the structure of the original program should be simpler to verify deductively, as the meaning of the original program is not lost in translation.

This idea is implemented in the tool BSP-Why [74], that uses sequentialization to verify imperative BSP programs. Again, a mapping is used to simulate the memory of each process. However, the instructions of the program are segmented into sequential blocks containing no parallel primitives. These are then transformed into loops that executes the sequential block **p** times: once per process and memory context.

Both of these sequentializations differ from ours in that we do not explicitly simulate the execution and memory of each process, but instead non-deterministically execute one single process. This is due to the notion of local computation cost in BSP. Sequential cost analysis applied to the sequentializations of Conc2Seq or BSP-Why would return an upper bound on the cumulative cost of executing each local computation phase. Applied to ours, it returns an upper bound on the local computation cost of executing the slowest process — which is the relevant measure in BSP.

The polyhedral model has seen widespread usage in areas such as automatic parallelization [127], verification and data-race analysis [37] and communication analysis [44, 30, 99]. Our work is in the same vein as Clauss' [44], who uses polyhedra to model load distribution in communicating parallel programs. The polyhedral model has also been used for automatically evaluating the communi-

cation volumes produced by loops and evaluating their different transformations by this measure [30, 30]. Our work differs in that we are first to exploit the polyhedral model to extract BSP communication costs and to integrate in a method for automatically obtaining BSP cost functions.

5.6 Concluding Remarks

The cost model is one of the key advantages of the *Bulk Synchronous Parallel* model. In this chapter we presented a method for *automatic cost analysis* of imperative BSP programs, in the aim of relieving the programmer of manually analyzing the cost of her algorithms. This method exploits the *textual alignment* property statically detected by replicated synchronization analysis of Chapter 4 to rewrite parallel programs into sequential programs and analyzing the communication distribution in the polyhedral model to obtain tight bounds on communication cost. The rewritten programs can then be treated by existing methods for cost analysis, obtaining the BSP cost of the original program.

We have evaluated the method and shown that the analysis obtains tight bounds on the cost of data-oblivious BSP programs that accurately predicts their actual run time in two different parallel environments. In addition to facilitating algorithm development, one possibility opened up by this development is online task scheduling in a system with evolving BSP parameters. Parallel *straightline* programs present another promising use case of the analysis in its current form. Such programs are common in signal processing and are characterized by simple control flow. However, they can scale to large sizes for which manual analysis is intractable.

Our method puts specific requirements on the analyzed program, namely the programs are structured and that all barriers are textually aligned. The former requirement is incidental. It is both inherited from the textual alignment analysis (developed in Chapter 4 and applied in Section 5.3.1) and due to the algorithm for polyhedral extraction used in the communication analysis (described in Section 5.3.2). Both of which require structured programs, but for which both proposals have been made for the analysis of non-structured programs [5, 90].

The latter requirement is inherent to the sequentialization approach of Section 5.3.1. To analyze the local computation cost of imperative BSP programs, one must deduce which sequences of instructions may execute in parallel in each superstep. The local computation cost is the costliest of these sequences.

The structure of textual alignment ensures that this sequence starts, and ends, at the same synchronization primitive in each process in each superstep. Consequently, those sequences will only execute in parallel with themselves and they can thus be analyzed in isolation. An alternative approach would require a more involved may-happen-in-parallel analysis, extracting and analyzing each potentially concurrent local computation phase.

The next step of our research includes the full implementation of the proposed method and evaluation on larger programs. One axis of future development is relaxing the constraints on the structure of the input programs, as well as treating a larger fragment of C with BSPlib. Our analysis gives imprecise costs for programs with data-dependent control flow. Treating such programs is an interesting venue of future research. Lastly, we would like to treat other measures on BSP costs (lower bound, average case, etc.) as well as treating costs of resources outside the BSP cost model, such as memory usage.

Safe Registration in BSPlib

6

Contents

6.1 BSPlib Registration and its Pitfalls 158
6.2 **BSPlite** with Registration 161
6.2.1 Local Semantics 162
6.2.2 Global Semantics 167
6.3 Instrumented Semantics 170
6.3.1 Instrumented Global Semantics 176
6.4 Correct Registration 179
6.4.1 Correctness 179
6.5 Sufficient Condition for Correct Registration 182
6.6 Related Work 183
6.7 Concluding Remarks 184

This chapter is extracted from the author's article [112].

In this chapter we study registration in BSPlib and exploit textual alignment to define a sufficient condition for its correct usage. This dynamic characterization of correct registration forms the formal underpinnings of future work towards a static analysis for verifying safe registration in BSPlib programs.

As introduced in Section 2.3.5, a BSPlib registration is an association between **p** memory addresses, one per process. It allows one process to reference memory objects on remote processes without knowing their address, thus enabling *Direct Remote Memory Access* (DRMA). At synchronization, the BSPlib runtime uses these registrations to route communication. Unfortunately, the BSPlib interface for manipulating registrations is informally defined with subtle corner cases that may provoke dynamic errors.

The first contribution of this chapter is an extension of **BSPlite**, formalizing BSPlib with registrations, with which we characterize correct registration. To our knowledge, ours is the first realistic formalization capturing the full idiosyncrasies of BSPlib registration.

The second contribution is a characterization of a subset of correct programs based on textual alignment. We exploited the notion of textual alignment to verify in synchronization Chapter 4 and to analyze BSP costs in Chapter 5. In this chapter, we generalize this notion to all collective operations, and in a restricted sense to memory locations. This requires an instrumentation of the semantics of programs which is slightly more complex. We believe this is the first work towards *static* verification of BSPlib registration.

This chapter proceeds as follows: In Section 6.1, we review the BSPlib registration mechanism and its pitfalls. Then, in Section 6.2, we extend our formalization of BSPlib to model registration. We instrument this semantics in Section 6.3, allowing us to define correct registration in Section 6.4. In Section 6.5, we describe and prove our sufficient condition for correct registration. We discuss related work in Section 6.6 and conclude in Section 6.7.

6.1 BSPlib Registration and its Pitfalls

BSPlib programs typically use DRMA for (buffered) communication, enabled by the `bsp_put` and `bsp_get` primitives. Before a process can issue DRMA requests to a remote memory area, this memory area must first have been associated to a local memory area using a *registration*. While a more extensive overview of this mechanism is given in Section 2.3.5, we here recall the basic notions.

A registration is an association between $\mathbf{p}$ addresses[1], one per process, that is stored in the *registration sequence*. Collectively calling the functions `bsp_push_reg`, or `bsp_pop_reg`, requests the addition, or removal, of a registration from the registration sequence. Logically, a registration can be seen as a $\mathbf{p}$-vector of addresses $\langle l_i \rangle_i$, where l_i is the argument of process i to `bsp_push_reg`. Registration requests (removals, and then additions) are executed at synchronization, and their effect is visible in the following superstep.

[1] In this chapter, we ignore the size of registered memory areas, which may vary per process. The size has no impact on the registration errors, so is left out to simplify the presentation.

Unfortunately, the registration mechanism has several important corner cases, and imprudent registration can cause dynamic errors. We informally characterize correct registration by the following rules:

(a) A registration $\langle l_i \rangle_i$ is created when all processes call `bsp_push_reg(`l_i`)` in the same superstep, and it becomes *active* in the next superstep.

(b) The addition and removal of registrations are collective actions to which all processes must participate. However, if a process does not need to expose any memory, it can pass `NULL` as the first argument to `bsp_push_reg`.

(c) The same address can be registered multiple times. Only the last registration of an address in the registration sequence is active. The motivation is modularity: to allow addresses to be reused for communication in different parts of the code, possibly unbeknownst to each other.

(d) The last active registration of $\langle l_i \rangle_i$ is removed when all processes call `bsp_pop_reg(`l_i`)`, and it becomes unavailable in the next superstep. A dynamic error occurs if the last pushed l_i is not at the same level in the registration sequence of all processes.

(e) Registration requests must be *compatible*: the order of all pushes must be the same on all processes, and for the pops likewise. However, it does not matter how requests are interleaved within one superstep.

	BSPlite Program	Pid	[Registrations ‖ push / $\overline{\text{pop}}$ requests] at synchronization
(1)	push &y; push &z; push &y; push &x; sync; pop &y; sync	0/1	$l_y l_z l_y l_x \longrightarrow l_y l_z \cancel{l_y} l_x \,\|\| \overline{l_y} \longrightarrow l_y l_z l_x$
(2)	push &x; pop &x; sync	0/1	$l_x \overline{l_x} \longrightarrow \Omega_R$
(3)	$p :=$ &y; if (pid $= 0$) then ($q :=$ &y) else ($q :=$ &x); push p; push q; sync; pop p; sync	0 1	$l_y l_y$ / $l_y l_x \longrightarrow l_y \cancel{l_y} \,\|\| \overline{l_y}$ / $\cancel{l_y} l_x \,\|\| \overline{l_y} \longrightarrow \Omega_R$
(4)	$p :=$ malloc pid; $q :=$ malloc pid; push p; push q; sync; pop p; sync	0 1	N N / $l_1 l_2 \longrightarrow$ N ~~N~~ / $\cancel{l_1} l_2 \,\|\| \overline{\text{N}}$ / $\overline{l_1} \longrightarrow \Omega_R$
(5)	if (pid $= 0$) then ($x := 0$) else (push &x); sync	0 1	$\longrightarrow \Omega_R$
(6)	push &y; sync; if (pid $= 0$) then (pop &y; push &x) else (push &x; pop &y); sync	0 1	l_y / $l_y \longrightarrow \cancel{l_y} \,\|\| \overline{l_y} l_x$ / $\cancel{l_y} \,\|\| l_x \overline{l_y} \longrightarrow l_x$ / l_x

Figure 6.1 – *Running examples illustrating registration in BSPlib. For each example an execution with* $\mathbf{p} = 2$ *is given depicting the registration sequence and requests before each synchronization. Here,* l_x *is the location of variable* x*,* l_i *is the* i*th address returned by* `malloc` *and* `N` *is* `NULL`*. The symbol* Ω_R *denotes a registration error. A struck through location,* $\cancel{l_y}$*, denotes the component of a registration that would be removed by a pop request. Program labels are omitted for legibility.*

The running examples in Figure 6.1, written in **BSPlite** extended with registration (detailed below in Section 6.2), illustrate these rules:

Example 1 Execution proceeds without error. The depicted registration sequence grows to the right. In the second superstep, the most recently pushed registration of `y` is removed.

Example 2 the program attempts to remove a registration of `x`. While a registration of this variable is requested in the same superstep, it does not become active until the next superstep, and so an error is produced (Rule (6.1)).

Example 3 A dynamic error occurs since the pop in the second superstep attempts to remove at different levels in the registration sequence (Rule (6.1)).

Example 4 This example is a simplified version of real BSPlib code, illustrating how the situation of Example 3 could be reproduced by dynamic allocation as `malloc` may return `NULL` [164, p. 143].

Example 5 A dynamic error occurs when only process 1 pushes (Rule (6.1)).

Example 6 This example illustrates how the interleaving of requests in one superstep is irrelevant as long as the requests are compatible (Rule (6.1)).

The goal of this chapter is to define a sufficient condition that forbids erroneous programs such as Examples 2 to 5. Just as textually aligned barriers is an intuitive, sufficient condition that ensures correct synchronization, the sufficient condition that we will develop in this chapter is intuitive and ensures correct registration.

As a first step, we formalize an extension of **BSPlite** that we use to characterize correct registration.

6.2 **BSPlite** with Registration

As we have done in the two previous chapters, we start by extending **BSPlite** to model the aspects of BSPlib that we are studying. The version of **BSPlite** used in this chapter has pointers, dynamic allocation, registration and communication (Figure 6.2). As in Chapter 5, we model DRMA communication, but in this chapter our modelization is closer to the BSPlib in that it implements transfers between *memory areas*, whereas the communication there is between *variables*.

$$
\begin{array}{llll}
\mathbf{AExp}_p & \ni e & ::= & pe \mid n \mid e_1\ op_a\ e_2 \mid \texttt{nprocs} \mid \texttt{pid} \mid \&x \\
\mathbf{PExp} & \ni pe & ::= & x \mid *e \\
\mathbf{BExp}_p & \ni b & ::= & \texttt{true} \mid \texttt{false} \mid e_1\ op_r\ e_2 \mid b_1\ op_b\ b_2 \mid !b \\
\mathbf{Par} & \ni s & ::= & [\texttt{skip}]^\ell \mid s_1; s_2 \mid \texttt{if}\ [b]^\ell\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ [b]^\ell\ \texttt{do}\ s_1 \\
 & & \mid & [\texttt{sync}]^\ell \mid [pe := e]^\ell \mid [pe := \texttt{malloc}\ e]^\ell \\
 & & \mid & [\texttt{free}\ e]^\ell \mid [\texttt{push}\ e]^\ell \\
 & & \mid & [\texttt{pop}\ e]^\ell \mid [\texttt{put}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell \mid [\texttt{get}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell
\end{array}
$$

$$x \in \mathbf{Var}, n \in \mathbf{Nat}, op_r \in \{=, <\}, op_b \in \{\mathbf{and}, \mathbf{or}\}, op_a \in \{+, -, \times\}$$

Figure 6.2 – *Syntax of* **BSPlite** *with registration*

Arithmetic expressions in $\mathbf{AExp}_p$ are as before with the addition of pointer expressions and the address-of operator. A pointer expression in **PExp** is either a variable or the dereferencement of an arithmetic expression. Boolean expressions are unchanged with respect to previous chapters.

Commands now also include dynamic allocation (`malloc`, whose argument indicates desired allocation size) and deallocation (`free`, whose argument indicates the memory area to deallocate). Assignments now assign to a location designated by the left-hand side pointer expression. The parallel primitives `sync`, `push`, `pop`, `put` and `get` and their arguments[2] model their BSPlib counterparts (detailed in Section 2.3.5). In other words, $[\texttt{put}\ e_{pid}\ e_{src}\ e_{dst}\ e_{offs}\ e_{nb}]^\ell$ requests the transfer of the memory area starting at e_{src} in the origin process and extending e_{nb} memory cells, into the memory area at the target process e_{pid} that is in an active registration with e_{dst}, at offset e_{offs}. The primitive $[\texttt{get}\ e_{pid}\ e_{src}\ e_{offs}\ e_{dst}\ e_{nb}]^\ell$ requests the transfer of the memory area at the target e_{pid} that is in an active registration with e_{src} at offset e_{offs} and extending e_{nb} bytes, into the memory area of the origin process at e_{dst}.

We no longer include the command `skip`. It served as a default continuation in previous chapters, but is no longer needed in this version of the language.

6.2.1 Local Semantics

In previous chapters we defined the local and global level of computation using big-step semantics. In this chapter, we instead define local computation using a small-step semantics. This change facilitates the instrumentation used to define correctness and our sufficient condition.

[2] With exception of the ignored size argument to `push`, as explained in the previous chapter.

The introduction of pointers to the language leads us to modify the domain by replacing states with an *environment* and *heaps*. The former maps variables to *locations*, and the latter locations to *values*. Whereas values were previously restricted to natural integers, they are now defined as the disjoint union of natural integers and locations. A location is a base address-offset pair. We distinguish the special location `N` modeling `NULL`, defined as $(b_{\mathtt{N}}, 0)$ where $b_{\mathtt{N}}$ is a distinguished base address.

We assume a fixed set of variables and assume that they are stored at the same location in each process. We thus simplify the semantics by parameterizing it by a global, fixed environment ρ that maps local variables to unique, non-`NULL` locations. This parameter remains implicit in the notations that follow.

Heaps are partial functions from (allocated) locations to values. We denote $\mathcal{H}_0$ the initial heap that allocates the location of all local variables **Var**.

$$
\begin{array}{lcll}
l & \in & \textbf{Loc} & = \textbf{Base} \times \textbf{Nat} \\
v & \in & \textbf{Val} & = \textbf{Nat} + \textbf{Loc} \\
\rho & \in & \textbf{Env} & = \textbf{Var} \to \textbf{Loc} \\
\mathcal{H} & \in & \textbf{Heap} & = \textbf{Loc} \hookrightarrow \textbf{Val} \\
\mathcal{H}_0 & \in & \textbf{Heap} \text{ such that} & \\
\multicolumn{4}{l}{\quad \forall l \in \textbf{Loc}. \mathcal{H}_0\, l \text{ is defined if } \exists x \in \textbf{Var}. \rho\, x = l,\ \mathcal{H}_0\, l = \text{undef} \ \text{ oth.}}
\end{array}
$$

The semantics of expressions, now incorporating these modifications to the domain, is given in Figure 6.3. We restrict pointer arithmetic to the pointer's base. Contrary to the modeled language C, our semantics does not allow the creation of an invalid pointer by offsetting a pointer into the memory area of another memory object. In other words, we do not consider invalid pointer usage, which can be precluded by other means. This restriction simplifies our proofs, by removing errors in the local semantics. The full semantics of boolean expressions is unchanged with respect to previous chapters and omitted.

The local semantics operates over configurations that consist of an (optional) program residue, a heap, a list of registration requests and a list of communication requests to execute at the next synchronization (see Figure 6.4). To distinguish requests (e.g. **push**) from the corresponding primitives (e.g. `push`), the former are typeset in bold. Registration requests contain only the location pushed or popped.

Communication requests are either **put** or **get** requests. The **put** requests contain the target process identifier, the list of values to transmit, a location from the origin process referring to a registration that identifies the destination location

$$
\begin{cases}
\mathcal{A}[\![\cdot]\!]^i & : \quad \mathbf{AExp}_p \to (\mathbf{Heap} \hookrightarrow \mathbf{Val}) \\
\mathcal{A}[\![pe]\!]^i\,\mathcal{H} & = \quad \mathcal{H}\,(\mathcal{P}[\![pe]\!]^i\,\mathcal{H}) \\
\mathcal{A}[\![n]\!]^i\,\mathcal{H} & = \quad n \\
\mathcal{A}[\![e_1\ op_a\ e_2]\!]^i\,\mathcal{H} & = \quad \begin{cases} (b_1, o_1\ [\![op_a]\!]\ n_2) & \text{if } \mathcal{A}[\![e_1]\!]^i\,\mathcal{H} = (b_1, o_1) \in \mathbf{Loc} \\ & \quad \text{and } \mathcal{A}[\![e_2]\!]^i\,\mathcal{H} = n_2 \\ & \quad \text{and } op_a \in \{+, -\} \\ n_1\ [\![op_a]\!]\ n_2 & \text{if } \mathcal{A}[\![e_1]\!]^i\,\mathcal{H} = n_1 \text{ and } \mathcal{A}[\![e_2]\!]^i\,\mathcal{H} = n_2 \end{cases} \\
\mathcal{A}[\![\texttt{nprocs}]\!]^i\,\mathcal{H} & = \quad \mathbf{p} \\
\mathcal{A}[\![\texttt{pid}]\!]^i\,\mathcal{H} & = \quad i \\
\mathcal{A}[\![\&x]\!]^i\,\mathcal{H} & = \quad \rho\,x
\end{cases}
$$

$$
\begin{cases}
\mathcal{P}[\![\cdot]\!]^i & : \quad \mathbf{PExp} \to (\mathbf{Heap} \hookrightarrow \mathbf{Loc}) \\
\mathcal{P}[\![x]\!]^i\,\mathcal{H} & = \quad \rho\,x \\
\mathcal{P}[\![*e]\!]^i\,\mathcal{H} & = \quad l \text{ if } \mathcal{A}[\![e]\!]^i\,\mathcal{H} = l, \text{ undef oth.}
\end{cases}
$$

$$
\begin{cases}
\mathcal{B}[\![\cdot]\!]^i & : \quad \mathbf{BExp}_p \to (\mathbf{Heap} \hookrightarrow \mathbf{Bool}) \\
& \ldots
\end{cases}
$$

Figure 6.3 – **BSPlite** *arithmetic, pointer and boolean expression semantics, where* $[\![op_a]\!]$ *gives the arithmetic denotation of the arithmetic operator* op_a *in the natural way.*

$\mathbf{RReq} \ni r ::=$ (Registration requests)
- $\mathbf{push}\,l$
- $\mathbf{pop}\,l$

$\mathbf{CReq} \ni c ::=$ (Communication requests)
- $\mathbf{put}\,j\,vs\,l\,n$ (Target pid, values, target loc., offset)
- $\mathbf{get}\,j\,l_1\,n_1\,l_2\,n_2$ (Target pid, source loc., offset, target loc., length)

$st = (\mathcal{H}, rrs, crs) \in \mathbf{State} = \mathbf{Heap} \times \mathbf{RReq}^* \times \mathbf{CReq}^*$

$\mathbf{LocalConf} \ni \gamma ::=$ (Local configuration)
- $\langle s, st \rangle$
- st

Figure 6.4 – *Configurations of the local semantics*

in the target process, and an offset into this registration, while **get** requests contain the target process identifier, a location from the origin process referring to a registration that identifies the source location in the target process, an offset into this registration, a destination location referring to the memory of the origin process, and the number of values to transfer.

Local Rules

As before, a reduction step from configuration γ to γ' is a judgment parameterized by the number of processes $\mathbf{p}$ and local process identifier $i \in \mathbf{Pid}$ and written $\vdash_1 \gamma \rightarrow^i_\alpha \gamma'$ where $\alpha \in \{\kappa, \iota\}$ denotes *termination type*. As we are now in a small-step semantics, there can be two reasons why the final configuration contains a residue program. The termination type allows us to distinguish the two cases. When $\alpha = \kappa$ the residue is the next step of local computation within the current superstep. However, when $\alpha = \iota$, local computation is suspended and requesting synchronization. Then the residue in γ' is the continuation to be executed in the next superstep.

We now comment upon the small-step rules (Figure 6.5) where they differ from the semantics of previous chapters. We will not describe the differences between a big-step and a small-step semantics and refer instead to a textbook [203].

Assignments now update the heap at the location denoted by the left-hand side pointer expression, which must be allocated (ASSIGN). Synchronization is initiated in SYNC. Dynamic allocation in MALLOC is handled by the predicate *alloc*, whose definition is omitted. Intuitively, $\mathit{alloc}\, n\, \mathcal{H}\, \mathcal{H}'\, b$ holds if $\mathcal{H}'$ is identical to $\mathcal{H}$ except that when n is positive, the locations $(b, 0) \ldots (b, n-1)$ are defined in the former and the base-address b does not occur in $\mathcal{H}$. If $n = 0$, then $b = \mathtt{N}$. The new memory's content is undetermined. This memory can be deallocated by `free` (rule FREE). This requires that the expression given as argument denotes a location that has been previously returned by `malloc`, that is, an allocated location with offset 0 that does not belong to a local variable. Intuitively, the predicate $\mathit{dealloc}\, \mathcal{H}\, b\, \mathcal{H}'$ holds if $\mathcal{H}$ and $\mathcal{H}'$ are identical, except that all locations of base b is undefined in the latter. The rules PUSH, POP, PUT and GET append registration requests and communication requests to the state.

Multi-Step Relation

We write $\gamma \rightarrow^i_\alpha \gamma'$ if there is a sequence of steps from γ such that γ' lacks a residue or the last step requests synchronization (defined in Figure 6.6).

$$\frac{}{\vdash_1 \langle [\mathtt{skip}]^\ell, st\rangle \to^i_\kappa st}\ \textsc{skip}$$

$$\frac{}{\vdash_1 \langle [\mathtt{sync}]^\ell, st\rangle \to^i_l st}\ \textsc{sync}$$

$$\frac{\mathcal{P}[\![pe]\!]^i\,\mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad \mathcal{A}[\![e]\!]^i\,\mathcal{H} = v \quad \mathcal{H}[l \leftarrow v] = \mathcal{H}'}{\vdash_1 \langle [pe := e]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}', rrs, crs)}\ \textsc{assign}$$

$$\frac{\mathcal{P}[\![pe]\!]^i\,\mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad alloc\,(\mathcal{A}[\![e]\!]^i\,\mathcal{H})\,\mathcal{H}\,\mathcal{H}'\,b}{\vdash_1 \langle [pe := \mathtt{malloc}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}'[l \leftarrow (b,0)], rrs, crs)}\ \textsc{malloc}$$

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = (b,0) \in \mathbf{Dom}(\mathcal{H}) \quad b \neq b_{\mathbb{N}} \quad \nexists x, \rho\, x = (b,0) \quad dealloc\,\mathcal{H}\,b\,\mathcal{H}'}{\vdash_1 \langle [\mathtt{free}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}', rrs, crs)}\ \textsc{free}$$

$$\frac{\vdash_1 \langle s_1, st\rangle \to^i_\alpha st'}{\vdash_1 \langle s_1; s_2, st\rangle \to^i_\alpha \langle s_2, st'\rangle}\ \textsc{seq\_1}$$

$$\frac{\vdash_1 \langle s_1, st\rangle \to^i_\alpha \langle s_1', st'\rangle}{\vdash_1 \langle s_1; s_2, st\rangle \to^i_\alpha \langle s_1'; s_2, st'\rangle}\ \textsc{seq\_2}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \mathtt{tt}}{\vdash_1 \langle \mathtt{if}\,[b]^\ell\,\mathtt{then}\,s_1\,\mathtt{else}\,s_2, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa \langle s_1, (\mathcal{H}, rrs, crs)\rangle}\ \textsc{if\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \mathtt{ff}}{\vdash_1 \langle \mathtt{if}\,[b]^\ell\,\mathtt{then}\,s_1\,\mathtt{else}\,s_2, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa \langle s_2, (\mathcal{H}, rrs, crs)\rangle}\ \textsc{if\_ff}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \mathtt{tt}}{\vdash_1 \langle \mathtt{while}\,[b]^\ell\,\mathtt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa \langle s_1; \mathtt{while}\,[b]^\ell\,\mathtt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle}\ \textsc{wh\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \mathtt{ff}}{\vdash_1 \langle \mathtt{while}\,[b]^\ell\,\mathtt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}, rrs, crs)}\ \textsc{wh\_ff}$$

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = l \quad rrs +\!\!+ [\mathbf{push}\,l] = rrs'}{\vdash_1 \langle [\mathtt{push}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}, rrs', crs)}\ \textsc{push}$$

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = l \quad rrs +\!\!+ [\mathbf{pop}\,l] = rrs'}{\vdash_1 \langle [\mathtt{pop}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}, rrs', crs)}\ \textsc{pop}$$

$$\frac{\begin{array}{c}(\mathcal{A}[\![e_1]\!]^i\,\mathcal{H}, \ldots, \mathcal{A}[\![e_5]\!]^i\,\mathcal{H}) = (j, (b, offs), l, n_1, n_2)\\ vs = [\mathcal{H}\,(b, offs), \ldots, \mathcal{H}\,(b, offs + n_2 - 1)]\end{array}}{\vdash_1 \langle [\mathtt{put}\,e_1\,e_2\,e_3\,e_4\,e_5]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}, rrs, crs +\!\!+ [\mathbf{put}\,j\,vs\,l\,n_1])}\ \textsc{put}$$

$$\frac{(\mathcal{A}[\![e_1]\!]^i\,\mathcal{H}, \ldots, \mathcal{A}[\![e_5]\!]^i\,\mathcal{H}) = (j, l_1, n_1, l_2, n_2)}{\vdash_1 \langle [\mathtt{get}\,e_1\,e_2\,e_3\,e_4\,e_5]^\ell, (\mathcal{H}, rrs, crs)\rangle \to^i_\kappa (\mathcal{H}, rrs, crs +\!\!+ [\mathbf{get}\,j\,l_1\,n_1\,l_2\,n_2])}\ \textsc{get}$$

Figure 6.5 – *Local semantics of commands in* **BSPlite** *with registration*

$$\frac{\vdash_1 \langle s, st\rangle \to^i_\kappa \langle s', st'\rangle \quad \langle s', st'\rangle \to^i_\alpha \gamma'}{\langle s, st\rangle \to^i_\alpha \gamma'}\ \textsc{step}$$

$$\frac{\vdash_1 \langle s, st\rangle \to^i_\iota \gamma}{\langle s, st\rangle \to^i_\iota \gamma}\ \textsc{susp} \qquad \frac{\vdash_1 \langle s, st\rangle \to^i_\alpha st'}{\langle s, st\rangle \to^i_\alpha st'}\ \textsc{term}$$

Figure 6.6 – *Local multi-step semantics of* **BSPlite** *commands*

$$\begin{cases} (\cdot)^{\mathtt{T}} & : \quad \forall A, (A^*)^{\mathbf{p}} \hookrightarrow (A^{\mathbf{p}})^* \\ \langle x_i : xs_i\rangle^{\mathtt{T}}_i & = \quad \langle x_i\rangle_i : \langle xs_i\rangle^{\mathtt{T}}_i \\ \langle\epsilon\rangle^{\mathtt{T}}_i & = \quad \epsilon \end{cases}$$

$$\begin{cases} \ominus & : \quad (\mathbf{RegSeq} \cup \{\Omega_R\}) \times \mathbf{Loc}^{\mathbf{p}} \to (\mathbf{RegSeq} \cup \{\Omega_R\}) \\ rs \ominus \langle l_i\rangle_i & = \quad \begin{cases} rs_1 +\!\!+ rs_2 & \text{if } rs = rs_1 +\!\!+ [\langle l_i\rangle_i] +\!\!+ rs_2 \text{ and} \\ & (\nexists i \in \mathbf{Pid}, k \in \mathbf{Nat}.\, rs_2[k][i] = l_i) \\ \Omega_R & \text{oth.} \end{cases} \end{cases}$$

$$\begin{cases} \mathcal{R}_\ominus, \mathcal{R}_\oplus & : \quad (\mathbf{RegSeq} \cup \{\Omega_R\}) \times (\mathbf{Loc}^{\mathbf{p}})^* \to (\mathbf{RegSeq} \cup \{\Omega_R\}) \\ \mathcal{R}_\ominus\, rs\, [L_1, \ldots, L_n] & = \quad ((rs \ominus L_1) \ominus L_2) \ominus \ldots \ominus L_n \\ \mathcal{R}_\ominus\, rs\, \epsilon & = \quad rs \\ \mathcal{R}_\oplus\, rs\, Ls_\oplus & = \quad rs +\!\!+ Ls_\oplus \text{ if } st \neq \Omega_R,\ \Omega_R \text{ oth.} \end{cases}$$

$$\begin{cases} \mathcal{R} & : \quad \mathbf{RegSeq} \times (\mathbf{RReq}^*)^{\mathbf{p}} \to (\mathbf{RegSeq} \cup \{\Omega_R\}) \\ \mathcal{R}\, rs\, \langle rrs_i\rangle_i & = \quad \mathcal{R}_\oplus\, Ls'_\oplus\, (\mathcal{R}_\ominus\, rs\, Ls'_\ominus) \text{ if } (Ls'_\ominus, Ls'_\oplus) = (Ls^{\mathtt{T}}_\ominus, Ls^{\mathtt{T}}_\oplus),\ \Omega_R \text{ oth.} \\ \quad \text{where } Ls_\ominus, Ls_\oplus = \langle [l \mid \mathbf{pop}\, l \in rrs_i]\rangle_i, \langle [l \mid \mathbf{push}\, l \in rrs_i]\rangle_i \end{cases}$$

Figure 6.7 – *The function $\mathcal{R}$ formalizes the effect of registration requests on a registration sequence.*

6.2.2 Global Semantics

In addition to initiating local computation in each process and treating the resulting communication as before, the global rules now also treat registration requests before executing following supersteps.

The global semantics operates over global configurations consisting of the habitual **p**-vectors of local configurations, but now also a *registration sequence*: a list of location **p**-vectors, where each vector is a registration:

$$\begin{array}{rcll} rs & \in & \mathbf{RegSeq} & = & (\mathbf{Loc}^{\mathbf{p}})^* \\ \Gamma & \in & \mathbf{GlobalConf} & = & \mathbf{LocalConf}^{\mathbf{p}} \times (\mathbf{RegSeq} \cup \{\Omega_R\}) \end{array}$$

We formalize how global computation applies the registration requests of a

$$
\begin{cases}
\mathcal{L} : \mathbf{RegSeq} \times \mathbf{Pid} \times \mathbf{Pid} \times \mathbf{Loc} \hookrightarrow \mathbf{Loc} \\
\mathcal{L}(rs, pid_1, pid_2, l) = \\
\quad \begin{cases}
l_{pid_2} & \text{if } \exists rs_1, rs_2.\, rs = rs_1 +\!\!+ [\langle l_i \rangle_i] +\!\!+ rs_2 \\
& \quad \text{and } l_{pid_1} = l \text{ and } \nexists k \in \mathbf{Nat}.\, rs_2[k][pid_1] = l \\
\text{undef} & \text{otherwise}
\end{cases}
\end{cases}
$$

$$
\begin{cases}
\mathcal{V} : \mathbf{GlobalState} \times \mathbf{Pid} \times \mathbf{Loc} \to \mathcal{P}(\mathbf{Val}) \\
\mathcal{V}((\langle \gamma_i \rangle_i, rs), i, (b, o)) = vs_{put} \cup vs_{get} \\
\text{where} \quad vs_{put} = \{v \mid \forall\, \mathbf{put}\, i\, vs\, l_j\, n_{offs} \in \pi_{crs}(\gamma_j) \\
\qquad \wedge\, \mathcal{L}(rs, j, i, l_j) = (b, o_i) \\
\qquad \wedge\, vs[o - (o_i + n_{offs})] = v\} \\
\qquad\quad vs_{get} = \{v \mid \forall\, \mathbf{get}\, j\, l_i\, n_{offs}\, (b, o_{dst})\, n_{len} \in \pi_{crs}(\gamma_i) \\
\qquad \wedge\, \mathcal{L}(rs, i, j, l_i) = (b_j, o_j) \\
\qquad \wedge\, o_{dst} \leq o < o_{dst} + n_{len} \\
\qquad \wedge\, \pi_{\mathcal{H}}(\gamma_j)\, (b_j, o_j + n_{offs} + (o - o_{dst})) = v\}
\end{cases}
$$

$$
\begin{cases}
\mathcal{C} : \mathbf{State^p} \times \mathbf{RegSeq} \times \mathbf{State^p} \\
\mathcal{C}(\langle \gamma_i \rangle_i, rs, \langle \gamma_i[\mathcal{H} \leftarrow \mathcal{H}'_i] \rangle_i) \iff \\
\quad \forall l \in \mathbf{Loc}.\, \mathcal{H}'_i\, l = v \iff \begin{cases}
v = \pi_{\mathcal{H}}(\gamma_i)\, l & \text{if } \mathcal{V}(\langle \gamma_i \rangle_i, rs, i, l) = \emptyset \\
v \in \mathcal{V}(\langle \gamma_i \rangle_i, rs, i, l) & \text{oth.}
\end{cases}
\end{cases}
$$

$$
\begin{cases}
Comm : \mathbf{GlobalState} \times \mathbf{GlobalState} \\
\qquad\qquad \dfrac{\mathcal{C}(\langle \gamma_i \rangle_i, st, \langle \gamma'_i \rangle_i) \quad rs' = \mathcal{R}(rs, \langle \pi_{rrs}(\gamma_i) \rangle_i)}{Comm((\langle \gamma_i \rangle_i, rs), (\langle \gamma'_i[rrs \leftarrow \epsilon, crs \leftarrow \epsilon] \rangle_i, rs'))}
\end{cases}
$$

Figure 6.8 – *Communication in* **BSPlite** *programs*

superstep to the registration sequence by the function $\mathcal{R}$ (Figure 6.7). It splits and transposes the list-vector into vector-lists of registrations to remove (by applying $\mathcal{R}_\ominus$) and registrations to add (by applying $\mathcal{R}_\oplus$). Registration sequences are manipulated by appending new lists of registrations and by popping registrations ($\ominus$-operator). Pop returns a dynamic error (Ω_R) if the popped registrations is not present in the sequence, or if the last appearance of each component is not at the same position. If two components of the transposition's operand do not have the same length, then the result is undefined and $\mathcal{R}$ returns a dynamic error, following the intuition of Rule (6.1) in Section 6.2.

$$\frac{rs \neq \Omega_R \quad \forall i \in \mathbf{Pid} \,.\, \gamma_i \rightarrow_\iota^i \gamma_i' \quad Comm((\langle \gamma_i' \rangle_i, rs), \Gamma'')}{(\langle \gamma_i \rangle_i, rs) \longrightarrow_\iota \Gamma''} \text{ GSUSP}$$

$$\frac{rs \neq \Omega_R \quad \forall i \in \mathbf{Pid} \,.\, \gamma_i \rightarrow_\kappa^i \gamma_i'}{(\langle \gamma_i \rangle_i, rs) \longrightarrow_\kappa (\langle \gamma_i' \rangle_i, rs)} \text{ GTERM}$$

$$\frac{}{Reach(\Gamma, \Gamma)} \text{ REFL} \qquad \frac{Reach(\Gamma, \Gamma'') \quad \Gamma'' \longrightarrow_\alpha \Gamma'}{Reach(\Gamma, \Gamma')} \text{ RSTEP}$$

Figure 6.9 – *Global big-step semantics of* **BSPlite** *programs and the reachability relation*

Global Rules

Compared to previous chapters, where a big-step semantics was used to define global computation, we here define a global step relation, and then define a reachability relation that relates two global configurations if the first (the initial) can be reached by a (potentially empty) sequence of global steps. Note also that, as in Chapter 4, we do not model synchronization errors, instead letting the step relation be undefined if there are incoherent termination states.

The global semantics of **BSPlite** programs with registration is given in Figure 6.9. One global step by $\mathbf{p}$ processes from global configuration Γ to Γ', written $\Gamma \longrightarrow_\alpha \Gamma'$, assumes processes have the same termination type α and that the registration sequence is not in an erroneous state.

The relation *Comm* is used to obtain the final configuration when $\alpha = \iota$, indicating that all local processes are requesting synchronization. The execution of communication is non-deterministic in the case of concurrent writes, and therefore *Comm* is a relation. Specifically, *Comm*, defined in Figure 6.8, executes (by $\mathcal{R}$) and removes the registration requests and the same of the communication requests (by $\mathcal{C}$). The projections $\pi_{rrs}(\gamma)$, $\pi_{crs}(\gamma)$ respectively $\pi_{\mathcal{H}}(\gamma)$ retrieves the registration requests, communication requests respectively heap from γ.

The relation $\mathcal{C}$ relates two environment vectors when the latter updates each heap of the former, according to the communication requests. As concurrent writes admit multiple resolutions of communication requests, $\mathcal{C}$ is a relation. In the updated heap, each location that is the target of some communication requests must contain one of the values specified by those communication requests. The auxiliary function $\mathcal{V}$ returns the set of values that can be written to a location in a given process as a result of communication requests in the global configuration. It uses $\mathcal{L}$ to translate origin address in the request to the corresponding address in the target process of the request.

Initial configuration The initial global configuration for a **BSPlite** program s with is now given by

$$\Gamma_s = (\langle\langle s, (\mathcal{H}_0, \epsilon, \epsilon)\rangle\rangle_{i \in \mathbf{Pid}}, \epsilon)$$

where the initial program is replicated and paired with the initial heap, empty request lists and registration sequence.

6.3 Instrumented Semantics

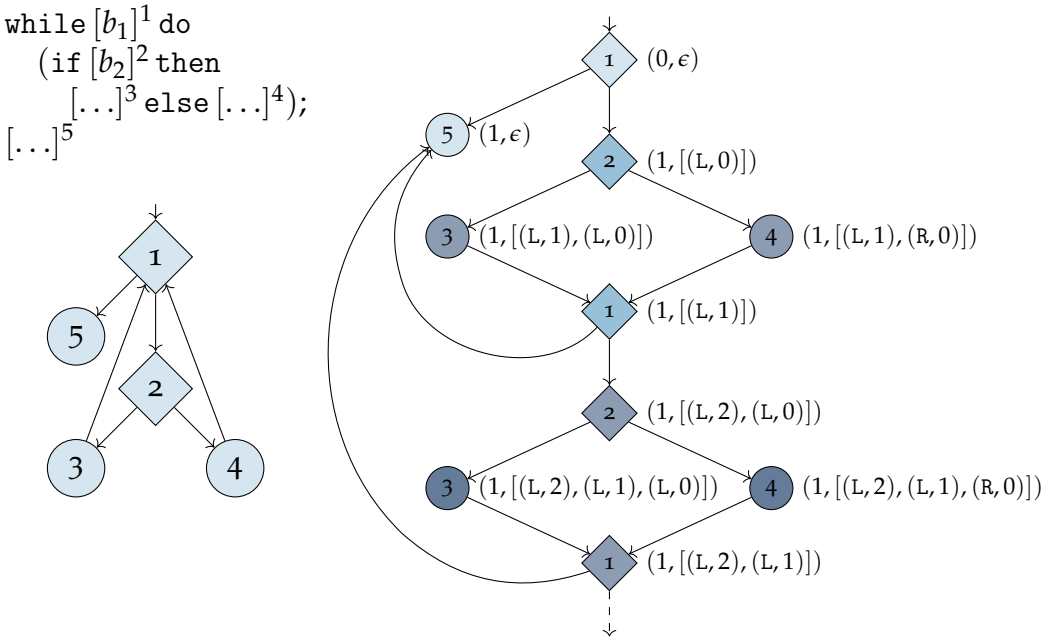
Instrumented Local State

We now *instrument* the semantics of the previous section. The instrumentation serves to capture the *trace of actions* taken by the program. Actions are of type `push!`, `pop!` or `sync!`, and are generated by the corresponding commands. The instrumentation has no impact on execution: it only serves as a basis for the definitions in Sections 6.4 and 6.5.

Each action stores the *path* [56] taken to reach the program point where it was generated. Intuitively, the path at a program point encodes the history of choices previously taken at control flow branches, trimming choices of fully executed branches. Trimming ensures that the path only contains a choice if it is relevant to the current position, such that if another branch would have been taken, execution could not be at this program point.

Formally, a path $\delta \in \mathbf{Path}$ is a pair (k, w) where k is the count of commands crossed on the outermost *nesting level*, and w is a list of choices. The nesting level is the number of conditionals in which a command nested. The commands on the outermost nesting level are those that are not nested in any conditionals. A choice is a pair of type $\{\mathtt{L}, \mathtt{R}\} \times \mathbf{Nat}$, and denotes the branch taken (`L` for true, `R` for false) and the count of commands crossed on the nesting level after that choice.

Consider the program depicted in Figure 6.10 and its execution. Like illustrated, the execution's loop iterations can be understood as unfolded conditionals. Program points are decorated with their corresponding path at execution. Each step increments the count on the current level. Iterations of the loop labeled 1 also add a choice to the path, all of which are trimmed when leaving the loop for point 5. Similarly, the inner conditional labeled 2 adds a choice that is trimmed after leaving the branches to program point 1. The choices made at the



a) Example program and its CFG **b) Execution**

Figure 6.10 – *Illustration of paths. Loop execution is visualized as unfolded conditionals. Nesting levels in the unfolding are color coded.*

conditional labeled 2 is irrelevant to how loop 1 is reached, and choices at loop labeled 1 is irrelevant to how point 5 is reached.

Unlike a big-step semantics, once the small-step semantics enters a branch, there is no memory of the encompassing structure, and we can no longer discern where the conditional's body ends and thus when to trim the path. To remedy, we maintain a *nesting stack* η of labels in the instrumentation:

$$\eta \in \mathbf{NestingStack} = \mathbf{Lab}^*$$

For each partially executed conditional, a label in the stack indicates the program point at which the corresponding choice must be trimmed from the path. Introducing artificial commands denoting the branch's end would solve the same problem, but pollute the syntax.

Actions `push!` and `pop!` store the concerned location and its *source*: the memory object from whence it was obtained. This is either a local variable x, an instance of dynamic allocation denoted by its path δ or *unknown* (for communicated pointers and integer values):

$$\mathbf{Src} \ni s ::= x \mid \delta \mid \mathit{unknown}$$

We refer to the first two types of sources as "known" sources. To track the source

of pointers during execution, we introduce a shadow store o, called *origin*, that associates heap locations of pointers, with the source of their content.

$$o \in \mathbf{Origin} = \mathbf{Loc} \to \mathbf{Src}$$

The intent is that if $\mathcal{H}\, l = l'$ then the source of l' is $o\, l$. To illustrate, we consider two examples, both executing in a global environment ρ that maps each variable x to the location l_x. First, consider the execution of a program $[y{:=}5]^1; [q{:=}\&y]^2$. The resulting heap and the intended origin are given by:

$$\begin{array}{lcllcl} \mathcal{H}\, l_y & = & 5 & o\, l_y & = & \mathit{unknown} \\ \mathcal{H}\, l_p & = & l_y & o\, l_p & = & y \end{array}$$

The origin of locations that does not correspond to pointers, such as y, should have an unknown source. The origin of the location that corresponds to the pointer q contains the source the its referee, that is y.

Now consider the execution of $[p := \texttt{malloc}\, 0]^1; [q := \texttt{malloc}\, 0]^2$. The dynamic allocations will return `N` twice, but the two pointers p and q should have different sources, distinguished by their path:

$$\begin{array}{lcllcl} \mathcal{H}\, l_p & = & \texttt{N} & o\, l_p & = & [0, \epsilon] \\ \mathcal{H}\, l_q & = & \texttt{N} & o\, l_q & = & [1, \epsilon] \end{array}$$

We can now define actions and the complete state of the instrumentation, containing the current path, label stack and origin:

$$\begin{array}{lcll} \textbf{Action} & \ni & a & ::= \texttt{push!}\, \delta\, (l, s) \mid \texttt{pop!}\, \delta\, (l, s) \mid \texttt{sync!}\, \delta \\ \textbf{InstrState} & \ni & I & ::= \langle \delta, \eta, o \rangle \end{array}$$

For actions, we also define the projection π_{path} that gives their path, along with π_{offs} and π_{src} that give the offset and source of the location (if any).

Consider the loop in Figure 6.10. Program point 1 in the second iteration of the loop is identified by the path $(1, [(\texttt{L}, 2), (\texttt{L}, 1)])$. If control moves to program point 5 after evaluating the guard of the loop, then the two choices must be trimmed from the path to obtain $(1, \epsilon)$. Thus the instrumentation state here should contain this path, and the label 5 twice in the nesting stack:

$$\langle (1, [(\texttt{L}, 2), (\texttt{L}, 1)]), [5, 5], o \rangle$$

$$\begin{cases} \oplus & : & \mathbf{Path} \times \mathbf{Nat} \to \mathbf{Path} \\ (k_0, \epsilon) \oplus n & = & (k_0 + n, \epsilon) \\ (k_0, w +\!\!+ [(ch, k)]) \oplus n & = & (k_0, w +\!\!+ [(ch, k + n)]) \end{cases}$$

$$\begin{cases} \odot & : & \mathbf{Path} \times (\mathbf{Nat} \times \{\mathtt{L}, \mathtt{R}\}) \to \mathbf{Path} \\ (k_0, w) \odot (ch, k) & = & (k_0, w +\!\!+ [(ch, k)]) \end{cases}$$

$$\begin{cases} trim & : & \mathbf{Lab} \to (\mathbf{Path} \times \mathbf{Lab}^*) \to (\mathbf{Path} \times \mathbf{Lab}^*) \\ trim\, \ell\, ((k_0, w +\!\!+ [(ch, k)]), \ell : \eta) & = & trim\, \ell\, ((k_0, w), \eta) \\ trim\, \ell\, (\delta, \eta) & = & (\delta, \eta) \end{cases}$$

Figure 6.11 – *Operators and functions on paths and nesting stack*

$$\begin{cases} src^i & : & \mathbf{AExp}_p \times \mathbf{Heap} \times \mathbf{Origin} \to \mathbf{Src} \qquad i \in \mathbf{Pid} \\ src^i(pe, \mathcal{H}, o) & = & o\,(\mathcal{P}[\![pe]\!]^i\,\mathcal{H}) \\ src^i(e_1\ op_a\ e_2, \mathcal{H}, o) & = & \begin{cases} src^i(e_1, \mathcal{H}, o) & \text{if } \mathcal{A}[\![e_2]\!]^i \notin \mathbf{Loc} \\ unknown & \text{oth.} \end{cases} \\ src^i(\&x, \mathcal{H}, o) & = & x \\ src^i(\_, \mathcal{H}, o) & = & unknown \end{cases}$$

Figure 6.12 – *Source of expressions*

for some origin o.

Finally, steps are instrumented with a *nesting flag* used to update the nesting stack. The flag indicates when execution enters a branch ($e = \blacksquare$) and otherwise ($e = \square$).

Instrumented Local Rules

An instrumented local small-step is written

$$\vdash_1 \gamma; I \to^i_\alpha \gamma'; I', as, e$$

where the initial and final instrumentation states are given by I and I', the list as is either a singleton action or empty (written ϵ) and e is the nesting flag. We visually distinguish the instrumentation from the underlying semantics by separating with a semi-colon and type setting it in blue.

This relation is defined by the rules in Figure 6.13. Simple instructions are instrumented to increment the path using the $\oplus$ operator, (e.g. rule ISYNC). Con-

$$\frac{}{\vdash_1 \langle [\texttt{skip}]^\ell, st\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa st; \langle \delta \oplus 1, \eta, o\rangle, \epsilon, \square} \textsc{iskip}$$

$$\frac{}{\vdash_1 \langle [\texttt{sync}]^\ell, st\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\iota st; \langle \delta \oplus 1, \eta, o\rangle, [\texttt{sync}!\,\delta], \square} \textsc{isync}$$

$$\frac{\mathcal{P}[\![pe]\!]^i\,\mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad \mathcal{A}[\![e]\!]^i\,\mathcal{H} = v \quad \mathcal{H}[l \leftarrow v] = \mathcal{H}'}{\begin{array}{l}\vdash_1 \langle [pe := e]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa \\ \quad (\mathcal{H}', rrs, crs); \langle \delta \oplus 1, \eta, o[l \leftarrow src^i(e, \mathcal{H}, o)]\rangle, \epsilon, \square\end{array}} \textsc{iassign}$$

$$\frac{\mathcal{P}[\![pe]\!]^i\,\mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad alloc\,(\mathcal{A}[\![e]\!]^i\,\mathcal{H})\,\mathcal{H}\,\mathcal{H}'\,b}{\begin{array}{l}\vdash_1 \langle [pe := \texttt{malloc}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa \\ \quad (\mathcal{H}'[l \leftarrow (b,0)], rrs, crs); \langle \delta \oplus 1, \eta, o[l \leftarrow \delta]\rangle, \epsilon, \square\end{array}} \textsc{imalloc}$$

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = (b,0) \in \mathbf{Dom}(\mathcal{H}) \quad b \neq b_{\mathbb{N}} \quad \nexists x, \rho\; x = (b,0) \quad dealloc\,\mathcal{H}\,b\,\mathcal{H}'}{\vdash_1 \langle [\texttt{free}\,e]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa (\mathcal{H}', rrs, crs); \langle \delta \oplus 1, \eta, o\rangle, \epsilon, \square} \textsc{ifree}$$

$$\frac{\vdash_1 \langle s_1, st\rangle; I \rightarrow^i_\alpha st'; \langle \delta'', \eta'', o'\rangle, as, e \quad (\delta', \eta') = trim\,(init\,s_2)\,(\delta'', \eta'')}{\vdash_1 \langle s_1; s_2, st\rangle; I \rightarrow^i_\alpha \langle s_2, st'\rangle; \langle \delta', \eta', o'\rangle, as, \square} \textsc{iseq\_1}$$

$$\frac{\begin{array}{l}\vdash_1 \langle s_1, st\rangle; I \rightarrow^i_\alpha \langle s_1', st'\rangle; \langle \delta', \eta', o'\rangle, as, e \\ \eta'' = (init\,s_2) : \eta' \text{ if } e = \blacksquare,\ \eta' \text{ oth.}\end{array}}{\vdash_1 \langle s_1; s_2, st\rangle; I \rightarrow^i_\alpha \langle s_1'; s_2, st'\rangle; \langle \delta', \eta'', o'\rangle, as, \square} \textsc{iseq\_2}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \texttt{tt}}{\begin{array}{l}\vdash_1 \langle \texttt{if}\,[b]^\ell\,\texttt{then}\,s_1\,\texttt{else}\,s_2, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa \\ \quad \langle s_1, (\mathcal{H}, rrs, crs)\rangle; \langle (\delta \oplus 1) \odot (\mathtt{L}, 0), \eta, o\rangle, \epsilon, \blacksquare\end{array}} \textsc{iif\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \texttt{ff}}{\begin{array}{l}\vdash_1 \langle \texttt{if}\,[b]^\ell\,\texttt{then}\,s_1\,\texttt{else}\,s_2, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa \\ \quad \langle s_2, (\mathcal{H}, rrs, crs)\rangle; \langle (\delta \oplus 1) \odot (\mathtt{R}, 0), \eta, o\rangle, \epsilon, \blacksquare\end{array}} \textsc{iif\_ff}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \texttt{tt}}{\begin{array}{l}\vdash_1 \langle \texttt{while}\,[b]^\ell\,\texttt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa \\ \quad \langle s_1; \texttt{while}\,[b]^\ell\,\texttt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle; \langle (\delta \oplus 1) \odot (\mathtt{L}, 0), \eta, o\rangle, \epsilon, \blacksquare\end{array}} \textsc{iwh\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^i\,\mathcal{H} = \texttt{ff}}{\vdash_1 \langle \texttt{while}\,[b]^\ell\,\texttt{do}\,s_1, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa (\mathcal{H}, rrs, crs); \langle \delta \oplus 1, \eta, o\rangle, \epsilon, \square} \textsc{iwh\_ff}$$

Figure 6.13 – *Local, instrumented, semantics of* **BSPlite** *commands*

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = l \quad rrs +\!\!+ [\mathbf{push}\,l] = rrs' \quad src^i(e,\mathcal{H},o) = s}{\vdash_1 \langle [\texttt{push}\ e]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa (\mathcal{H}, rrs', crs); \langle \delta \oplus 1, \eta, o\rangle, [\texttt{push!}\,\delta\,(l,s)], \square}\ \textsc{ipush}$$

$$\frac{\mathcal{A}[\![e]\!]^i\,\mathcal{H} = l \quad rrs +\!\!+ [\mathbf{pop}\,l] = rrs' \quad src^i(e,\mathcal{H},o) = s}{\vdash_1 \langle [\texttt{pop}\ e]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa (\mathcal{H}, rrs', crs); \langle \delta \oplus 1, \eta, o\rangle, [\texttt{pop!}\,\delta\,(l,s)], \square}\ \textsc{ipop}$$

$$\frac{\begin{array}{l}(\mathcal{A}[\![e_1]\!]^i\,\mathcal{H}, \ldots, \mathcal{A}[\![e_5]\!]^i\,\mathcal{H}) = (j, (b, offs), l, n_1, n_2)\\ vs = [\mathcal{H}\,(b, offs), \ldots, \mathcal{H}\,(b, offs + n_2 - 1)]\end{array}}{\begin{array}{l}\vdash_1 \langle [\texttt{put}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa\\ \quad (\mathcal{H}, rrs, crs +\!\!+ [\mathbf{put}\,j\,vs\,l\,n_1]); \langle \delta \oplus 1, \eta, o\rangle, \epsilon, \square\end{array}}\ \textsc{iput}$$

$$\frac{(\mathcal{A}[\![e_1]\!]^i\,\mathcal{H}, \ldots, \mathcal{A}[\![e_5]\!]^i\,\mathcal{H}) = (j, l_1, n_1, l_2, n_2)}{\begin{array}{l}\vdash_1 \langle [\texttt{get}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell, (\mathcal{H}, rrs, crs)\rangle; \langle \delta, \eta, o\rangle \rightarrow^i_\kappa\\ \quad (\mathcal{H}, rrs, crs +\!\!+ [\mathbf{get}\,j\,l_1\,n_1\,l_2\,n_2]); \langle \delta \oplus 1, \eta, o\rangle, \epsilon, \square\end{array}}\ \textsc{iget}$$

Figure 6.13 – *Local, instrumented, semantics of* **BSPlite** *commands, continued*

$$\frac{\vdash_1 \langle s, st\rangle; I \rightarrow^i_\kappa \langle s', st'\rangle; I'', as, e \quad \langle s', st'\rangle; I'' \rightarrow^i_\alpha \gamma; I', as'}{\langle s, st\rangle; I \rightarrow^i_\alpha \gamma; I', (as +\!\!+ as')}\ \textsc{istep}$$

$$\frac{\vdash_1 \langle s, st\rangle; I \rightarrow^i_\iota \gamma; I', as, e}{\langle s, st\rangle; I \rightarrow^i_\iota \gamma; I', as}\ \textsc{isusp} \qquad \frac{\vdash_1 \langle s, st\rangle; I \rightarrow^i_\alpha st'; I', as, e}{\langle s, st\rangle; I \rightarrow^i_\alpha st'; I', as}\ \textsc{iterm}$$

Figure 6.14 – *Local, instrumented, multi-step semantics of* **BSPlite** *commands*

ditionals increment and append the appropriate choice using the $\odot$ operator (e.g. rule IIF_TT). These operators are defined in Figure 6.11. Conditionals also set the nesting flag ■. When the reduction of the first component of a sequence sets this flag, the second component's label is pushed to the nesting stack (ISEQ_2), as a reminder to the *trim* function to remove a choice once the first component is fully reduced (ISEQ_1). This function is also defined in Figure 6.11.

The *src* function, defined in Figure 6.12, gives the source of arithmetic expressions that evaluates to locations. Intuitively, if the expression is the address-of operator, then the name of the operand is given; if the expression is a pointer expression, then the origin is consulted; etc. This function is used when updating the origin at assignments and allocations (e.g. rule IASSIGN), and consulted along with the path when generating actions (e.g. rule IPUSH).

$$\frac{rs \neq \Omega_R \quad \forall i \in \mathbf{Pid}\,.\,\gamma_i; I_i \rightarrow^i_\iota \gamma'_i; I'_i, as_i \quad IComm((\langle \gamma'_i; I'_i \rangle_i, rs), \Gamma'')}{(\langle \gamma_i; I_i \rangle_i, rs) \longrightarrow_\iota \Gamma' \,;\, \langle as_i \rangle_i} \text{ IGSUSP}$$

$$\frac{rs \neq \Omega_R \quad \forall i \in \mathbf{Pid}\,.\,\gamma_i; I_i \rightarrow^i_\kappa \gamma'_i; I'_i, as_i}{(\langle \gamma_i; I_i \rangle_i, rs) \longrightarrow_\kappa (\langle \gamma'_i; I'_i \rangle_i, rs) \,;\, \langle as_i \rangle_i} \text{ IGTERM}$$

$$\frac{}{Reach(\Gamma, \Gamma); \langle \epsilon \rangle_i} \text{ IREFL} \qquad \frac{Reach(\Gamma, \Gamma''); A \quad \Gamma'' \longrightarrow_\alpha \Gamma' \,;\, A'}{Reach(\Gamma, \Gamma'); A +\!\!+ A'} \text{ IRSTEP}$$

Figure 6.15 – *Instrumented global big-step semantics of* **BSPlite** *programs and the reachability relation*

Instrumented Multi-Step Relation

We define an instrumented multi-step relation in Figure 6.14, which propagates the instrumentation and accumulates actions. We write

$$\gamma; I \rightarrow^i_\alpha \gamma'; I', as$$

if there is a sequence of instrumented sequence of small-steps from γ such that γ' lacks a residue or is suspended, with I and I' as the sequence's initial and final instrumentation and where the final action trace is given by as.

6.3.1 Instrumented Global Semantics

Similarly, we instrument the global semantics, which is written:

$$\Gamma \longrightarrow_\alpha \Gamma' \,;\, A$$

Here, the **p**-vector A collects each process's action trace. The instrumented global semantics and reachability relation, are defined in Figure 6.15. The instrumentation of the global semantics does three things: it propagates the instrumented states, it accumulates the action traces vectors by concatenation, and it instruments communication so that the source of overwritten pointers is set to *unknown*. This is done by the function *IComm*, whose definition is omitted.

The trace vectors resulting from executing the running examples with the instrumentation is given in Figure 6.16. The trace vectors of the first two programs are unremarkable: registration is applied to the location of local variables with the corresponding source. In the third example, we note how the difference in source of the second push allows to distinguish registrations of distinct memory objects. This would not be directly possible from locations in the modeled

Program	Action trace vector	LC	TA	SA	SF	GC
Example 1	⟨ [push! … (l_y, y), push! … (l_z, z), push! … (l_y, y), push! … (l_x, x), sync! …, pop! (l_y, y), sync! …], [push! … (l_y, y), push! … (l_z, z), push! … (l_y, y), push! … (l_x, x), sync! …, pop! (l_y, y), sync! …] ⟩	✓ ✓	✓	✓	✓	✓
Example 2	⟨ [push! … (l_x, x), pop! (l_x, x), sync! …], [push! … (l_x, x), pop! (l_x, x), sync! …] ⟩	✗ ✗	✓	✓	✗	✗
Example 3	⟨ [push! … (l_y, y), push! … (l_y, y), sync! …, pop! (l_y, y), sync! …], [push! … (l_y, y), push! … (l_x, x), sync! …, pop! (l_y, y), sync! …] ⟩	✓ ✓	✓	✗	✗	✗
Example 4	⟨ [push! … $(\mathtt{N}, (0, \epsilon))$, push! … $(\mathtt{N}, (1, \epsilon))$, sync! …, pop! $(\mathtt{N}, (0, \epsilon))$, sync! …], [push! … $(l_1, (0, \epsilon))$, push! … $(l_2, (1, \epsilon))$, sync! …, pop! $(l_1, (0, \epsilon))$, sync! …] ⟩	✗ ✓	✓	✓	✗	✗
Example 5	⟨ [sync! $(1, \epsilon)$], [push! $(1, [(\mathtt{R}, 0)])$ (l_x, x), sync! $(1, \epsilon)$] ⟩	✓ ✓	✗	✗	✗	✗
Example 6	⟨ [push! $(0, \epsilon)$ (l_y, y), sync! $(1, \epsilon)$, pop! $(2, [(\mathtt{L}, 0)])$ (l_y, y), push! $(2, [(\mathtt{L}, 1)])$ (l_x, x), sync! $(2, \epsilon)$], [push! $(0, \epsilon)$ (l_y, y), sync! $(1, \epsilon)$, push! $(2, [(\mathtt{R}, 0)])$ (l_x, x), pop! $(2, [(\mathtt{R}, 1)])$ (l_y, y), sync! $(2, \epsilon)$] ⟩	✓ ✓	✗	✗	✗	✓

Figure 6.16 – *Trace vectors resulting from executing running examples with* $\mathbf{p} = 2$. *The right part of the table evaluates the traces' local correctness (LC), the vector's textual collective alignment (TA), source alignment (SA) and safety (SF), and global correctness (GC) as defined in Sections 6.4 and 6.5. Here,* l_x *is the location of variable* x *and* x *its source and* l_i *is the* i*th address returned by* `malloc`. *For legibility, the paths in actions have been replaced by ellipses, in all examples but Example 5 and Example 6.*

language, since the address of distinct local variables may be the same on different processes. Conversely, the address of the same local variable may differ between processes. This is demonstrated by the fourth example. Here the two pushes of each process concern different instances of memory allocation. Even though the location of the push, as in the trace of the first process, concur, the source allows to distinguish them. In the fifth example, we see immediately that process 1 does not participate in the push, indicating a dynamic error. In the final example, both processes participate in the push and the pop. However, the distinct paths of the `push!` actions respectively `pop!` actions indicate that those actions did not result from calls from the same instruction in the program.

An important property of the action trace vectors of the instrumentation is that each known source is associated to at most one base in each process. We call such action trace vectors **consistent**. Intuitively, the same base may be allocated twice (i.e. $b_{\mathbb{N}}$), but the path of each execution step is unique and so by extension, the source of each dynamic allocation is also unique. This is demonstrated in the trace of Example 5 in Figure 6.16.

Conversely, an action trace where two distinct bases appear with two known but different sources can not be generated by the instrumentation. It would either contradict the requirement that the global environment assigns unique bases to each local variable, or the fact that each distinct dynamic allocations comes from a different path.

Lemma 1. *If* $Reach(\Gamma, \Gamma'); A$ *then the same source* $s \neq unknown$ *never appears twice in the same component of* A *associated with two locations of different base.*

For the proof sketch of this lemma, and proof sketches of remaining properties in this chapter with omitted proofs, we refer to Appendix B.

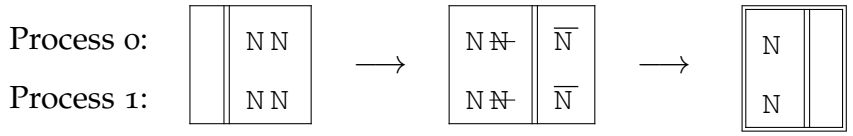
The semantics also ensures that the source of each local variable x is always associated with the fixed location of that variable, i.e. $\rho\, x$.

Instrumented initial configuration The initial instrumentation state has an empty path, empty nesting stack and initial origin $o_0 = (\lambda l.unknown)$. Hence, the instrumented initial global configuration is given by

$$\Gamma_c = (\langle\langle c, (\mathcal{H}_0, \epsilon, \epsilon)\rangle; \langle (0, \epsilon), \epsilon, o_0\rangle\rangle_i, \epsilon)$$

6.4 Correct Registration

To define correctness we reason on the action trace vectors generated by the instrumentation of the previous section. We impose a slight restriction compared to the BSPlib standard. Consider an execution of Example 4 where `malloc` returns `N` twice for both processes (this could be caused by space constraints):



There is no dynamic error, but arguably by "accident": the popped registration is probably not the one intended by the user and as the originally considered execution shows, an error could occur. We impose an additional **source restriction**: whenever a registration is popped, then the location in the **pop** request and the one in the registration sequence must come from the same known *source*: intuitively, the same local variable or same instance of dynamic allocation. This restriction ensures that a correct execution stays correct independently on the behavior of `malloc`.

6.4.1 Correctness

The action trace vectors generated by the instrumentation can be seen as programs with actions as instructions. From this point of view, we give a semantics of traces as functions over a state tracking the source and order registrations of each location, defining a local view of correctness, and then a global semantics for action trace vectors. Anticipating our sufficient condition for correctness, this will enable a definition of *global correctness from a local perspective*.

Local Correctness

Local action trace semantics is defined by $\mathcal{LC}!\llbracket\cdot\rrbracket$ (Figure 6.17) that symbolically executes the trace, tracks the source of pushed locations, and verifies that each popped location has been pushed and committed, and, by the source restriction, with the same source used in the `pop!` action. This is verified by the $\simeq$ operator, defined in the same figure. Local correctness of as amounts to $\mathcal{LC}!\llbracket as\rrbracket = \mathtt{tt}$. $\mathcal{LC}!\llbracket\cdot\rrbracket$ is defined by $\mathcal{LC}!_1$ giving the effect of one action on the state, $\mathcal{LC}!_{ss}$ giving the effect of all actions of one superstep and $\mathcal{LC}!'$ gives the effect of the whole trace. Any action in the last superstep has no effect and so as_i is ignored by $\mathcal{LC}!'$. By extension, we say an action trace vector is locally correct if each component is.

$$r \in \mathbf{Map} = \mathbf{Loc} \to \mathbf{Src}^*$$

$$\begin{cases} \mathcal{LC}!_1[\![\cdot]\!] & : \mathbf{Action} \to \mathbf{Map} \hookrightarrow \mathbf{Map} \\ \mathcal{LC}!_1[\![\texttt{pop!}\,\delta\,(l,s)]\!]\,r & = r[l \leftarrow ss] \text{ if } (r[l] = s' : ss) \wedge s \simeq s', \text{ undef oth.} \\ \mathcal{LC}!_1[\![\texttt{push!}\,\delta\,(l,s)]\!]\,r & = r[l \leftarrow s : r[l]] \\ \mathcal{LC}!_1[\![\texttt{sync!}\,\delta]\!]\,r & = \text{undef} \end{cases}$$

$$\begin{cases} \mathcal{LC}!_{ss}[\![\cdot]\!] & : \mathbf{Action}^* \to \mathbf{Map} \hookrightarrow \mathbf{Map} \\ \mathcal{LC}!_{ss}[\![as]\!]\,r & = \mathit{fold}\ \mathcal{LC}!_1[\![\cdot]\!]\ as'\ r \\ \quad \text{where } as' = [a \mid \forall a \in as, a = \texttt{pop!}\_\_] +\!\!+ [a \mid \forall a \in as, a = \texttt{push!}\_\_] \end{cases}$$

$$\begin{cases} \mathcal{LC}![\![\cdot]\!] & : \mathbf{Action}^* \hookrightarrow \mathbf{Map} \\ \mathcal{LC}![\![as]\!] & = \mathit{fold}\ \mathcal{LC}!_{ss}[\![\cdot]\!]\ [as_1, \ldots, as_{n-1}]\ (\lambda l.\epsilon) \\ \quad \text{where } as_1 +\!\!+ [\texttt{sync!}\,\delta_1] +\!\!+ \ldots +\!\!+ [\texttt{sync!}\,\delta_{n-1}] +\!\!+ as_n = as \\ \quad\quad \text{such that } \forall 1 \leq i \leq n, (\texttt{sync!}\,\_) \notin as_i \end{cases}$$

$$\begin{cases} \mathcal{LC}![\![\cdot]\!] & : \mathbf{Action}^* \to \mathbf{Bool} \\ \mathcal{LC}![\![as]\!] & = \texttt{tt} \text{ if } \mathcal{LC}!'[\![as]\!] \text{ is defined, } \texttt{ff} \text{ oth.} \end{cases}$$

$$\begin{array}{lcl} s_1 \simeq s_2 & \iff & \mathit{unknown} \notin \{s_1, s_2\} \wedge s_1 = s_2 \\ \mathit{fold}\ f\ \epsilon\ r & = & r \\ \mathit{fold}\ f\ [a_1, \ldots, a_n]\ r & = & (f\ a_n \circ \ldots \circ f\ a_1)\ r \end{array}$$

Figure 6.17 – *Local correctness of an action trace*

$$
\begin{array}{lcl}
r & \in & \mathbf{MapI} = \mathbf{Loc} \to (\mathbf{Nat} \times \mathbf{Src}^*) \\
m & \in & \mathbf{Matching} = (\mathbf{Nat} \times \mathbf{Nat}^*)^*
\end{array}
$$

$$
\begin{cases}
\mathcal{GC}!_1 : \mathbf{Action} \to (\mathbf{Nat} \times \mathbf{Nat}^* \times \mathbf{MapI}) \hookrightarrow (\mathbf{Nat} \times \mathbf{Nat}^* \times \mathbf{Map} \\
\mathcal{GC}!_1[\![\texttt{pop!}\,\delta\,(l,s)]\!]\,(k,ps,r) = \\
\quad \begin{cases} (k, k' : ps, r[l \leftarrow is]) & \text{if } r[l] = (k', s') : is \wedge s \simeq s' \\ \text{undef} & \text{otherwise} \end{cases} \\
\mathcal{GC}!_1[\![\texttt{push!}\,\delta\,(l,s)]\!]\,(k,ps,r) = (k+1, ps, r[l \leftarrow (k,s) : r[l]]) \\
\mathcal{GC}!_1[\![\texttt{sync!}\,\delta]\!]\,(k,ps,r) = \text{undef}
\end{cases}
$$

$$
\begin{cases}
\mathcal{GC}!_{ss} : \mathbf{Action}^* \to (\mathbf{Matching} \times \mathbf{MapI}) \hookrightarrow (\mathbf{Matching} \times \mathbf{MapI}) \\
\mathcal{GC}!_{ss}[\![as]\!]\,(m,r) = \text{let } k' = k \text{ if } m = m' +\!\!+ [(k,\_)],\ 0 \text{ oth. in} \\
\qquad\qquad\qquad\qquad \text{let } (k'', ps, r') = \mathit{fold}\ \mathcal{GC}!_1[\![\cdot]\!]\ as'\ (k', \epsilon, r) \text{ in} \\
\qquad\qquad\qquad\qquad (m +\!\!+ [(k'', ps)], r') \\
\quad \text{where } as' = [a \mid \forall a \in as, a = \texttt{pop!}\_\_] +\!\!+ [a \mid \forall a \in as, a = \texttt{push!}\_\_]
\end{cases}
$$

$$
\begin{cases}
\mathcal{GC}!' : \mathbf{Action}^* \hookrightarrow (\mathbf{Matching} \times \mathbf{MapI}) \\
\mathcal{GC}![\![as]\!] = \mathit{fold}\ \mathcal{GC}!_{ss}[\![\cdot]\!]\ [as_1, \ldots, as_{n-1}]\ (\epsilon, \lambda l.\epsilon) \\
\quad \text{where } as_1 +\!\!+ [\texttt{sync!}\,\delta_1] +\!\!+ \ldots +\!\!+ [\texttt{sync!}\,\delta_{n-1}] +\!\!+ as_n = as
\end{cases}
$$

$$
\begin{cases}
\mathcal{GC}! : (\mathbf{Action}^*)^{\mathbf{P}} \to \mathbf{Bool} \\
\mathcal{GC}![\![\langle as_i \rangle_i]\!] = \texttt{tt} \text{ if } \exists ms, \forall i \in \mathbf{Pid}. \mathcal{GC}!'[\![as_i]\!](\epsilon, \lambda l.\epsilon) = (ms, \_),\ \texttt{ff} \text{ oth.}
\end{cases}
$$

Figure 6.18 – *Global correctness of a trace vector*

Global Correctness

As the executions of Examples 3 and 5 demonstrate, local correctness of a trace vector does not suffice for its global correctness. They must also make compatible sequences of actions (unlike Example 5), and pops must occur at the same position in the registration sequence (unlike Example 3).

To capture these requirements, we abstract the effect of a trace into a *matching* (see Figure 6.18). The matching is one pair (k, ps) per superstep, where k counts the total number of pushes at the end of the superstep and ps is a list containing the index of each push removed by a pop in the superstep. $\mathcal{GC}!$ extends $\mathcal{LC}!$ to index the pushes added to the state, and to return the matching of locally correct traces. We then define a trace vector as being globally correct when each component has the same matching.

Before going further, we reconnect global correctness of traces with the se-

mantics of **BSPlite**, and confirm that executions with globally correct traces do not have registration errors:

Theorem 4. *If* $Reach(\Gamma_c, (\langle \gamma_i; I_i \rangle_i, rs)); A$ *and* $\mathcal{GC}![\![A]\!] = \texttt{tt}$ *then* $rs \neq \Omega_R$.

This trace-based characterization of correctness is independent of the underlying instrumented language. But, we have yet to simplify the task of writing correct programs. In the next section we define our sufficient condition that guarantees safe registration.

6.5 Sufficient Condition for Correct Registration

The sufficient condition for registration imposes 3 conditions: (1) collective calls to `sync`, `push` and `pop` should be textually aligned (2) the argument of collective calls to `push` and `pop` should be *source aligned*, i.e., refer to the same memory object: the same local variable or the same instance of dynamically allocated memory in all processes; (3) the trace of actions of each process should be locally correct. These conditions in conjunction guarantee global correctness.

Consider again the running examples. The execution of Example 2 is not locally correct in any process, and hence disqualified. Due to the source restriction, the execution of Example 4 by process 0 is not locally correct. In the execution of Example 3, the memory object referred to by q is not the same in each process, and hence the second push is not source aligned. The executions of Examples 5 and 6 are not textually aligned. Only Example 1 follows satisfies the sufficient condition.

We now formalize the sufficient condition using action trace vectors:

Definition 3. *A trace vector* $\langle as_i \rangle_i$ *has* ***textually aligned collectives*** *if each component has the same length and at each position, the same path:*

$$\exists m \in \mathbf{Nat}.\, \forall i, j \in \mathbf{Pid}.\, |as_i| = |as_j| = m \wedge \forall 0 \leq k < m.\, \pi_{path}(as_i[k]) = \pi_{path}(as_j[k])$$

Definition 4. *A trace vector with textually aligned collectives* $\langle as_i \rangle_i$ *is* ***source aligned*** *if all components have the same (known) source and offset at each position:*

$$\begin{aligned} &\forall i, j \in \mathbf{Pid}.\, \forall 0 \leq k < |as_i|.as_i[k] \neq \texttt{sync!}\_ \\ \implies\ &\pi_{src}(as_i[k]) \simeq \pi_{src}(as_j[k]) \wedge \pi_{offs}(as_i[k]) = \pi_{offs}(as_j[k]) \end{aligned}$$

Definition 5. *A consistent trace vector A that has textually aligned collectives, is source aligned and locally correct is called* ***safe****.*

By extension, a program in our sufficient condition is one that only produces safe action trace vectors. The action trace vectors of the running examples are evaluated against these conditions in Figure 6.16. As expected, the intuitions given above are consistent with the formalization. Finally, we prove that the sufficient condition guarantees global correctness:

Theorem 5. *If A is safe then $\mathcal{GC}!\llbracket A \rrbracket = \mathtt{tt}$.*

As an immediate corollary, an execution that produces a safe action trace vector does not have any registration errors:

Corollary 2. *If $Reach(\Gamma_c, (\langle \gamma_i; I_i \rangle_i, rs)); A$ and A is safe then $rs \neq \Omega_R$.*
Proof: Immediate by Theorem 5 and Theorem 4. □

This sufficient condition is inspired by examining realistic and correct BSPlib code. We have manually inspected our corpus of BSPlib programs, and found that programs appear to satisfy it. This intuition remains to be verified, preferably by the development and application of a static analysis targeting the condition. However, our observations leads us to conjecture that in addition to ensuring correctness, the condition is sufficiently permissive and coherent with the programmer's intuition of correctness.

6.6 Related Work

To the best of our knowledge, this is the first work towards automatically verifying registrations in BSPlib. Closest to our work is BSP-Why [74], a tool for deductive verification of BSPlib-like imperative BSP programs. While BSP-Why can verify programs using registrations for communication, it is unclear to which degree the modelization is true to the BSPlib standard, and while this work forms the basis for automatic verification, the BSP-Why user must manually prove their program correct.

Other languages and libraries for parallelism enjoy less error-prone schemes than registrations for creating associations between local and remote memory for DRMA communication. In the BSP paradigm, Yzelman et al. [207], use a communication container class to turn regular objects into distributed data-structures.

MPI [139, p. 403] uses window objects to allow a process to reference remote memory. Like BSPlib registrations, windows act as handles and are created and removed collectively. Unlike registrations, windows can be removed in any order. In OpenSHMEM [36], DRMA operations are only allowed on "symmetric" objects that the runtime system ensures have the same relative address in each process.

Previous authors, such as Tesson et al. [184], have formalized BSPlib, but did not consider registration. Gava et al. [80] formalize Paderborn's BSPlib [27], with DRMA but they do not formalize pointers and their modelization of registration is simplified. This leads us to believe that ours is the first realistic formalization of BSPlib registration.

6.7 Concluding Remarks

In this chapter, we have studied errors caused by registration in BSPlib. Registration is used to create associations between local and remote memory, but can provoke errors if done incorrectly. We have formalized BSPlib with registration, characterized correct executions, given a sufficient condition and proved that it guarantees correctness with respect to our semantics.

The logical continuation of this work is to develop a static analysis targeting this sufficient condition, and to evaluate it by verifying registration in real-world BSPlib programs. This will prove our intuition that the sufficient condition is permissive enough to include realistic programs.

Conclusion and Future Work 7

Contents

7.1 Context 185
7.2 Thesis 186
7.3 Contributions 186
7.4 Perspectives 187

7.1 Context

Parallel computing is an important component in achieving high computation capacities. Large calculations that require the application of parallel computation are now commonplace. Not exclusively, but notably, in the natural sciences where increasing fidelity in simulations enable a more precise understanding of subjects as diverse as the origin of the universe and the composition of matter [97]; the functioning of the earth system [137] — with important implications for climate science; the formation of galaxies [64]; and the human brain [136].

However, even more so than sequential computing, parallel computing is fraught with errors. The well-known difficulties of developing correct sequential programs, and the disastrous effects when approached lightly (of which spectacular examples abound [65]), are exacerbated by the exponential number of interactions between processes. Additionally, the hoped for increase in computation power when applying parallel computing does not come for free. In all but embarrassingly parallel cases, a performance increase requires a well-thought out strategy for how to parallelize the problem at hand, demanding significant effort. It is also difficult to *a priori* ensure scaling and portability of the parallelization.

The Bulk Synchronous Parallel model, and its commonly used implementation in the BSPlib library, answers some of these concerns by providing a structure of parallel computing that rules out certain classes of errors. Furthermore, it

enables reliable and portable performance predictions. However, care must still be taken to develop correct BSPlib programs, and manual program analysis is necessary to enjoy the BSP performance model.

Formal methods provide a framework for developing software that is mathematically guaranteed to be safe and efficient. Automatic push-button methods, such as static analysis, are especially promising as they do not require the intervention of experts in formal methods. It is the lack of such methods adapted for BSPlib that has motivated this thesis. We have aimed to develop automated tools to aid the development of BSPlib programs that are both correct and efficient.

7.2 Thesis

Our thesis is that the majority of BSPlib programs follows a structure called *textual alignment*. We have argued that textual alignment should be enforced in scalable parallel programs and that static analyses can be developed that exploit this property. This approach elegantly alleviates one of the principal problems when analyzing parallel programs, namely the large number of possible interactions between processes.

7.3 Contributions

To argue the importance of textual alignment, we have first conceived a static analysis verifying textual alignment of synchronization in BSPlib programs. We have formalized, proved sound in Coq, implemented in Frama-C and evaluated this analysis. Second, we have conceived, implemented as a prototype and evaluated a static cost analysis for BSPlib programs that exploits textual alignment. Third, we have conceived a sufficient condition, based on textual alignment, for BSPlib that we prove guarantees safe registration. Finally, these developments are based on a series of progressively more involved formalizations of BSPlib features, from synchronization to communication and registration.

7.4 Perspectives

We conclude by discussing some promising lines of research.

The precision of the static analysis for textual alignment can be improved. It is in particular the conservative assumptions about communication that require revision. A fine analysis of communication patterns in BSPlib, possibly based on polyhedral techniques, to expand the recognition of *pid*-independent expressions would reduce the amount of annotations currently required.

The current prototype implementation of the cost analysis should be implemented and evaluated on realistic BSPlib programs, to further validate its applicability. The analysis of communication costs is precise, but only for data-oblivious communication patterns. Further research is needed to devise static cost analyses of data-dependent communication.

The sufficient condition for safe registration should be the target of a static analysis. This analysis must be conceived, proved to statically approximate the sufficient condition and implemented to evaluate its practicality.

Textual alignment could also be exploited outside static analysis. Previous authors have initiated work towards deductive verification of scalable parallel programs, notably by the introduction of invariants over all processes [175]. These invariants are attached as assertions to synchronization primitives. We believe such assertions can be attached to any textual aligned program point, and serve as the basis of a new, compositional proof system for imperative, SPMD programs.

In this thesis we focus on BSPlib. BSPlib can be seen as a model of the Bulk Synchronous subset of other parallel libraries based on the SPMD model such as MPI. Finally, we propose applying the results in this thesis to MPI, and also, exploiting textual alignment to develop new static analyses for MPI. Static analysis for synchronization based on the work on Barrier Inference has already been implemented for MPI [209], and the authors note that MPI barriers tend to be textually aligned. However, to the best of our knowledge, there is no work on cost analysis for MPI. The BSP subset of MPI can be assumed to follow the same cost model as BSPlib, opening up an extensions of our cost analysis to this library.

The semantics of MPI windows are not afflicted by the same issues as the homologue of BSPlib registrations. This limits the applicability of our contributions towards safe registration in the context of MPI. However, a static analysis of registration is a first step towards verifying the correct usage of DRMA operations in BSPlib. The high-performance primitives of BSPlib are similar to the

RMA primitives of MPI, and notably, the intermingling of remote and local accesses unprotected by synchronization can give implementation specific or erroneous behavior in both libraries [89]. We think that textual alignment could serve as a framework towards formalizing a simplified programming model for safe DRMA, as the one suggested by Gropp [89, p. 91]. This model could then serves as the formal underpinnings for a static analysis verifying DRMA operation in both BSPlib and MPI.

Appendix

Proofs for Replicated Synchronization

A

Contents

A.1 Operational Semantics Simulates Denotational 191
A.1.1 Stable State Transformers 192
A.1.2 Simulation 194
A.2 Correctness of PI 203
A.2.1 Domain 204
A.2.2 Parameterized Constraint System 204
A.2.3 Constraint System Facts 205
A.2.4 Marked Path Abstractions and *pid*-independent Variables 205
A.2.5 Correctness of the Analysis 209
A.3 Correctness of *RS* 216
A.3.1 Safe State Transformers 216

In this section we give the soundness proof of the Replicated Synchronization analysis, detailed in Chapter 4. These proofs have been mechanized and verified in the Coq proof assistant. A natural language version of them follow.

In these proofs, we shall sometimes take the liberty of hiding program labels to improve legibility.

A.1 Operational Semantics Simulates Denotational

This section contains the proof of Theorem 1.

A.1.1 Stable State Transformers

Recall the definition of D_I: for $I \subseteq \mathbf{Pid}$, we note

$$D_I = \{\theta \in D \setminus \{\bot, \Omega_S\} \mid \theta\, i = \mathbf{0} \iff i \notin I\}$$

Definition 6. *A function $f : D \to D$ is stable if $\forall I \subseteq \mathbf{Pid}$, $\forall \theta \in D_I, F\,\theta \in D_I \Cup \{\bot, \Omega_S\}$, $f\ \Omega_S = \Omega_S$ and $f\ \bot = \bot$.*

The composition of two stable functions is stable.

Lemma 2 (Stable Composition)**.** *Let $f_1, f_2 : D \to D$ be two stable functions. Then $f_1 \circ f_2$ is also stable.*
Proof: Take any $I \subseteq \mathbf{Pid}$ and any $\theta \in D_I$, and show that $(f_1 \circ f_2)\,\theta \in D_I \cup \{\bot, \Omega_S\}$.

Since f_2 is stable, either $f_2\ \theta \in D_I$ or $f_2\ \theta \in \{\bot, \Omega_S\}$. In the latter case, since f_1 is stable, $f_1\ (f_2\ \theta) = (f_2\ \theta) \in \{\bot, \Omega_S\}$, and in the former case, since f_1 is stable, we have $f_1\ (f_2\ \theta) \in D_I \cup \{\bot, \Omega_S\}$. □

Lemma 3 (Stable Mask Combine)**.** *Let $f_1, f_2 : D \to D$ be two stable functions. Let $b \in \mathbf{BExp}$. Then $\lambda\theta.f_1\ (\partial_b\,\theta) \parallel f_2\ (\partial_{!b}\,\theta)$ is also stable.*
Proof: Let $f = \lambda\theta.f_1\ (\partial_b\,\theta) \parallel f_2\ (\partial_{!b}\,\theta)$. By the definition of ∂, the stability of f_1 and f_2 and the definition of $\parallel$ we have $f\,\Omega_S = \Omega_S$ and $f\ \bot = \bot$. So let $I \subseteq \mathbf{Pid}$ and $\theta \in D_I$, and show that $f\,\theta \in D_I \cup \{\bot, \Omega_S\}$.

Let $I' = \{i \in I \mid \mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt}\}$. Then $(\partial_b\,\theta) \in D_{I'}$ and $(\partial_{!b}\,\theta) \in D_{I \setminus I'}$. Let $\theta_1 = f_1\ (\partial_b\,\theta)$ and $\theta_2 = f_2\ (\partial_{!b}\,\theta)$. Since f_1 and f_2 are stable, $\theta_1 \in D_{I'} \cup \{\bot, \Omega_S\}$ and $\theta_2 \in D_{I \setminus I'} \cup \{\bot, \Omega_S\}$.

Either

- $\theta_1 \in \{\bot, \Omega_S\}$ or $\theta_2 \in \{\bot, \Omega_S\}$ and then $f\,\theta = (\theta_1 \parallel \theta_2) \in \{\bot, \Omega_S\}$ as well, or
- $\theta_1 \in D_{I'}$ and $\theta_2 \in D_{I \setminus I'}$. In this case, $f\,\theta = \lambda i.\theta_1[i] + \theta_2[i]$. We show that $f\,\theta \in D_I$. By the definition of D_I, we must show that $(f\,\theta)[i] = \mathbf{0} \iff i \notin I$.

($\Longrightarrow$) Then $(f\,\theta)[i] = \mathbf{0}$. If $(f\,\theta)[i] = \theta_1[i] + \theta_2[i] = \mathbf{0}$ then $\theta_1[i] = \mathbf{0}$ and $\theta_2[i] = \mathbf{0}$. Then by definition of $D_{I'}$ and $D_{I \setminus I'}$, $i \notin I'$ and $i \notin I \setminus I'$. Thus $i \notin I$.

($\Longleftarrow$) Then $i \notin I$, and by consequence $i \notin I'$ and $i \notin I \setminus I'$. Thus

$$\begin{aligned}(f\,\theta)[i] &= \theta_1[i] + \theta_2[i] \\ &= \mathbf{0} + \theta_2[i] && \text{since } i \notin I' \text{ and } \theta_1 \in D_{I'} \\ &= \theta_2[i] \\ &= \mathbf{0} && \text{since } i \notin I \setminus I' \text{ and } \theta_2 \in D_{I \setminus I'}\end{aligned}$$

□

The semantic functions of statements is stable.

Definition 7. *Let $f_\perp : D \to D$ be the function that returns $\perp$ for all arguments:*

$$f_\perp\,\theta = \perp$$

Let F^n denote the functional F applied n times, i.e., $F^0 = \lambda\theta.\theta$ and $F^{n+1} = F^n \circ F$. Then $fix\ F = \bigsqcup\{F^n\,f_\perp \mid n \geq 0\}$. Furthermore, if $fix\ F\,\theta = \theta'$, then there is n such that $F^n\,\theta = \theta'$ [152, p. 123, Theorem 5.37].

Lemma 4. *For all $s \in$ **Par**, $[\![s]\!]$ is stable.*
Proof: Let $I \subseteq$ **Pid** and $\theta \in D_I$. We then show that $[\![s]\!]\,\theta \in D_I \cup \{\perp, \Omega_S\}$ by structural induction on s.

- $s \equiv$ `skip`. Then $[\![s]\!]\,\theta = \theta \in D_I$.
- $s \equiv$ `sync`. Either
  - $I =$ **Pid** or $I = \emptyset$, and then $[\![\texttt{sync}]\!]\,\theta = \theta \in D_I$
  - otherwise, $[\![\texttt{sync}]\!]\,\theta = \Omega_S$
- $s \equiv x := e$. Then $[\![s]\!] = [x \leftarrow e]$. We show that $[x \leftarrow e]\,\theta \in D_I$, by showing $([x \leftarrow e]\,\theta)[i] = \mathbf{0} \iff i \notin I$.

($\Longrightarrow$) Then $([x \leftarrow e]\,\theta)[i] = \mathbf{0}$. By definition of $[x \leftarrow e]$, this implies that $\theta[i] = \mathbf{0}$. Since $\theta \in D_I$, we have $i \notin I$.

($\Longleftarrow$) Then $i \notin I$. Thus $\theta[i] = \mathbf{0}$ since $\theta \in D_I$, and $([x \leftarrow e]\,\theta)[i] = \mathbf{0}$.

- $s \equiv s_1; s_2$. Then $[\![s]\!] = [\![s_2]\!] \circ [\![s_1]\!]$, and by applying induction hypothesis twice we obtain that $[\![s_1]\!]$ and $[\![s_2]\!]$ are stable, so we can conclude by Lemma 2.

- $s \equiv \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$. Then

$$[\![s]\!] = \lambda\theta.[\![s_1]\!]\,(\partial_b\,\theta) \parallel [\![s_2]\!]\,(\partial_{!b}\,\theta)$$

  Since $[\![s_1]\!]$ and $[\![s_2]\!]$ are stable by the induction hypothesis, we have $[\![s]\!]$ from Lemma 3.

- $s \equiv \texttt{while}\ b\ \texttt{do}\ s_1\ \texttt{end}$. Then $[\![s]\!] = fix\ F = \bigsqcup\{F^n\,f_\perp \mid n \geq 0\}$. By induction on n, we first show that $F^n\,f_\perp$ is stable.

  - When $i = 0$, then $(F^0\,f_\perp) = f_\perp$, and for any $I \subseteq \mathbf{Pid}$ and any $\theta \in D_I$, $f_\perp\,\theta = \perp$.
  - Let $I \subseteq \mathbf{Pid}$ and $\theta \in D_I$, and show $(F^{n+1}\,f_\perp)\,\theta \in D_I \cup \{\perp, \Omega_S\}$. Either
    - $(\partial_b\,\theta) \notin D_\emptyset$, and then

$$F^{n+1}\,f_\perp = \lambda\theta.f_1\,(\partial_b\,\theta) \parallel f_2\,(\partial_{!b}\,\theta)$$

      with $f_1 = (F^n\,f_\perp) \circ [\![s_1]\!]$ and $f_2 = \lambda\theta.\theta$. We have that f_1 is stable by Lemma 2 since $F^n\,f_\perp$ is stable by the inner induction hypothesis and $[\![s_1]\!]$ is stable by the outer induction hypothesis. Since f_2 is trivially stable, we can apply Lemma 3 and obtain that $\lambda\theta.(((F^n\,f_\perp) \circ [\![s_1]\!])\,(\partial_b\,\theta)) \parallel \partial_{!b}\,\theta$ is stable. As a consequence, $(F^{n+1}\,f_\perp)\,\theta \in D_I \cup \{\perp, \Omega_S\}$.
    - $(\partial_b\,\theta) \in D_\emptyset$, and then $(F^{n+1}\,f_\perp)\,\theta = \theta \in D_I$ by definition.

  Now, let $I \subseteq \mathbf{Pid}$ and $\theta \in D_I$. If $[\![s]\!]\,\theta = \theta'$, then there is n such that $(F^n\,f_\perp)\,\theta = \theta'$. Since $F^n\,f_\perp$ is stable for all n, we have that $\theta' \in D_I \cup \{\perp, \Omega_S\}$ and thus $[\![s]\!]$ is stable.

□

A.1.2 Simulation

Some helpful lemmas about global rules.

Lemma 5 (Superstep Sequence). *For all $S_1, S_2 : \mathbf{Par^p}, \theta, \theta', \theta'' \in D_{\mathbf{Pid}}$, if $\langle S_1, \theta\rangle \longrightarrow \theta''$ and $\langle S_2, \theta''\rangle \longrightarrow \theta'$ then $\langle\langle S_1[i]; S_2[i]\rangle_i, \theta\rangle \longrightarrow \theta'$.*

Proof: By rule induction on $\langle S_1, \theta\rangle \longrightarrow \theta''$:

ALL-OK Then

$$\forall i \in \mathbf{Pid}, \langle S_1[i], \theta[i]\rangle \rightarrow^i \langle Ok, \theta''[i]\rangle$$

We do case distinction on the last rule applied in $\langle S_2, \theta'' \rangle \longrightarrow \theta'$. Either:

ALL-OK Then $\forall i \in \mathbf{Pid}, \langle S_2[i], \theta''[i] \rangle \rightarrow^i \langle Ok, \theta'[i] \rangle$.

Then

$$\forall i \in \mathbf{Pid}, \langle S_1[i]; S_2[i], \theta[i] \rangle \rightarrow^i \langle Ok, \theta'[i] \rangle$$

by SEQ-OK and the conclusion follows by ALL-OK.

ALL-WAIT Then $\forall i \in \mathbf{Pid}, \langle S_2[i], \theta''[i] \rangle \rightarrow^i \langle Wait(S_2[i]), \theta'''[i] \rangle$ and (A) $\langle S_2, \theta''' \rangle \longrightarrow \theta'$ for some vector of programs S_2. Then (B),

$$\forall i \in \mathbf{Pid}, \langle S_1[i]; S_2[i], \theta[i] \rangle \rightarrow^i \langle Wait(S_2'[i]), \theta'''[i] \rangle$$

by SEQ-OK and so

$$\langle \langle S_1[i]; S_2[i] \rangle_i, \theta \rangle \longrightarrow \theta'$$

by ALL-WAIT with (A) and (B) as premises.

ALL-WAIT Then

$$\forall i \in \mathbf{Pid}, \langle S_1[i], \theta[i] \rangle \rightarrow^i \langle Wait(S_1'[i]), \theta'''[i] \rangle$$

and

$$\langle S_1', \theta''' \rangle \longrightarrow \theta''$$

then by the induction hypothesis

$$\langle \langle S_1'[i]; S_2[i] \rangle_i, \theta''' \rangle \longrightarrow \theta' \qquad (*)$$

Since

$$\forall i \in \mathbf{Pid}, \langle S_1[i], \theta[i] \rangle \rightarrow^i \langle Wait(S_1'[i]), \theta'''[i] \rangle$$

we have

$$\forall i \in \mathbf{Pid}, \langle S_1[i]; S_2[i], \theta[i] \rangle \rightarrow^i \langle Wait(S_1'[i]; S_2[i]), \theta'''[i] \rangle$$

by the SEQ-WAIT local rule, so we apply the ALL-WAIT global rule with this and (*) obtain

$$\langle \langle S_1[i]; S_2[i] \rangle_i, \theta \rangle \longrightarrow \theta'$$

as desired.

GLB-ERR Vacuous since $\theta'' \in D_{\mathbf{Pid}}$.

□

Lemma 6 (Superstep While). *Let* $s = \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end}$, *and* $\forall i \in \mathbf{Pid}, \mathcal{B}[\![b]\!]^i\, \theta[i] = \texttt{tt}$, $\langle \langle s_1 \rangle_i, \theta \rangle \longrightarrow \theta''$ *and* $\langle \langle s \rangle_i, \theta'' \rangle \longrightarrow \theta'$ *then* $\langle \langle s \rangle_i, \theta \rangle \longrightarrow \theta'$.
Proof:[Sketchy] It can be shown that

$$\langle \langle \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end} \rangle_i, \theta \rangle \longrightarrow \theta'$$

if and only if

$$\langle \langle \texttt{if}\, b\, \texttt{then}\, s_1; \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end}\, \texttt{else}\, \mathit{skip}\, \texttt{end} \rangle_i, \theta \rangle \longrightarrow \theta'$$

And since $\mathcal{B}[\![b]\!]^i\, \theta[i] = \texttt{tt}$ for all i,

$$\langle \langle \texttt{if}\, b\, \texttt{then}\, s_1; \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end}\, \texttt{else}\, \mathit{skip}\, \texttt{end} \rangle_i, \theta \rangle \longrightarrow \theta'$$

if and only if

$$\langle \langle s_1; s \rangle_i, \theta \rangle \longrightarrow \theta'$$

which can be obtained by Lemma 5 using the hypothesis. □

Lemma 7 (Superstep Conditional). *If* $\forall i \in \mathbf{Pid}, \mathcal{B}[\![b]\!]^i\, \theta[i] = \texttt{tt}$, *and* $\langle S_1, \theta \rangle \longrightarrow \theta'$, *then*

$$\langle \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end} \rangle_i, \theta \rangle \longrightarrow \theta'$$

for $S_1, S_2 : \mathbf{Pid} \to \mathbf{Par}$.
Proof: Either the last rule of d_1 is:

ALL-OK Then

$$\forall i \in \mathbf{Pid}, \langle S_1[i], \theta[i] \rangle \to^i \langle \mathit{Ok}, \theta'[i] \rangle$$

and then

$$\forall i \in \mathbf{Pid}, \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end}, \theta[i] \rangle \to^i \langle \mathit{Ok}, \theta'[i] \rangle$$

is obtained, and so

$$\langle \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end} \rangle_i, \theta \rangle \longrightarrow \theta'$$

by the ALL-OK rule.

ALL-WAIT In this case,

$$\forall i \in \mathbf{Pid}, \langle S_1[i], \theta[i] \rangle \to^i \langle \mathit{Wait}(S_1'[i]), \theta''[i] \rangle$$

and

$$\langle S_1', \theta'' \rangle \longrightarrow \theta'$$

for some vector of programs S_1' and environments θ''. Then

$$\forall i \in \mathbf{Pid}, \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end}, \theta[i] \rangle \rightarrow^i \langle Wait(S_1'[i]), \theta''[i] \rangle$$

and since $\langle S_1', \theta'' \rangle \longrightarrow \theta'$ we have $\langle \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end} \rangle_i, \theta \rangle \rightarrow \theta'$ by the ALL-WAIT-rule.

GLB-ERR In this case $\theta' = \Omega_S$ and

$$\exists i \in \mathbf{Pid}, \langle S_1[i], \theta[i] \rangle \rightarrow^i \langle Ok, \sigma_i \rangle$$
$$\exists j \in \mathbf{Pid}, \langle S_1[j], \theta[j] \rangle \rightarrow^j \langle Wait(c'), \sigma_j \rangle$$

for some i, j, σ_i, c' and σ_j. Then

$$\langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end}, \theta[i] \rangle \rightarrow^i \langle Ok, \sigma_i \rangle$$
$$\langle \texttt{if}\, b\, \texttt{then}\, S_1[j]\, \texttt{else}\, S_2[j]\, \texttt{end}, \theta[j] \rangle \rightarrow^j \langle Wait(c'), \sigma_j \rangle$$

so

$$\langle \langle \texttt{if}\, b\, \texttt{then}\, S_1[i]\, \texttt{else}\, S_2[i]\, \texttt{end} \rangle_i, \theta \rangle \rightarrow \Omega_S$$

by the GLB-ERR-rule.

□

Some helpful facts about the *mask* and *combine* operators:

Lemma 8 (Empty state is identity of combine)**.** *For all $\theta \in D$ and $\theta' \in D_\emptyset$, $\theta \parallel \theta' = \theta$.*

Proof: If $\theta \in \{\bot, \Omega_S\}$, then the result is immediate from the definition. If not,

$$\begin{aligned} (\theta \parallel \theta') &= \lambda i.(\theta[i] + \theta'[i]) \\ &= \lambda i.\theta[i] && \text{since } \theta'[i] = \mathbf{0} \\ &= \theta \end{aligned}$$

□

Definition 8. *Let $D_\circ$ denote all partially visible environments, i.e.*

$$D_\circ = D \setminus (D_{\mathbf{Pid}} \cup \{\Omega_S, \bot\})$$

Then partial equivalence between a function over states, and the local operational semantics of a statement, written Psim, is defined:

$$\begin{aligned} Psim(f,s) \iff & \forall \theta \in D_\circ, f\,\theta = \theta' \notin \{\Omega_S, \bot\} \implies \\ & \forall i \in \mathbf{Pid}, \theta[i] \neq \mathbf{0} \implies \\ & \langle s, \theta[i]\rangle \to^i \langle Ok, \theta'[i]\rangle \end{aligned}$$

Lemma 9 (Loop partial simulation)**.** *Let F be the functional associated with $s =$* `while` b `do` s_1 `end`*, i.e.:*

$$F = \lambda f.\lambda\theta. \begin{cases} (f \circ [\![s_1]\!])\ (\partial_b\,\theta) \parallel \partial_{!b}\,\theta & \text{if } \partial_b\,\theta \notin D_\emptyset \cup \{\bot, \Omega_S\} \\ \theta & \text{otherwise} \end{cases}$$

and assume that $Psim([\![s_1]\!], s_1)$ then for all $n \geq 0$, $Psim(F^n\,f_\bot, s)$.
Proof: By induction on n:

- If $n = 0$. Then $F^0\,f_\bot = \bot$ for all θ, so $Psim(F^0\,f_\bot, s)$ vacuously holds.
- If $n + 1$. Take any $\theta \in D_\circ$, and any $i \in \mathbf{Pid}$ such that $\theta[i] \neq \mathbf{0}$. If no such i exists then the conclusion holds vacuously. Either
  - $\mathcal{B}[\![b]\!]^i\ \theta[i] =$ `tt`, and then

    $$(F^{n+1}\,f_\bot)\ \theta[i] = ((F^n\,f_\bot) \circ [\![s_1]\!])\ (\partial_b\,\theta)\ i = \theta'[i]$$

    We need to show $\langle s, \theta[i]\rangle \to^i \langle Ok, \theta'\ i\rangle$, which we do by the WH-TT-OK rule in the operational semantics. Let

    $$\theta''[i] = ([\![s_1]\!] \circ (\partial_b\,\theta))\ i = [\![s_1]\!]\ \theta[i]$$

    and

    $$\theta'[i] = (F^n\,f_\bot)\ \theta''[i]$$

    The premises of the rule WH-TT-OK which we need to prove are $\langle s_1, \theta[i]\rangle \to^i \langle Ok, \theta''[i]\rangle$, which follows from the assumption $Psim([\![s_1]\!], s_1)$, and $\langle s, \theta''[i]\rangle \to^i \langle Ok, \theta'[i]\rangle$ which follows from the induction hypothesis.
  - $\mathcal{B}[\![b]\!]^i\ \theta[i] =$ `ff`. Then

    $$(F^{n+1}\,f_\bot)\ \theta[i] = \theta'[i] = \theta[i]$$

We need to show $\langle s, \theta[i]\rangle \to^i \langle Ok, \theta[i]\rangle$ which we do by the WH-FF-rule.

□

Lemma 10 (Update assign). *For all* $e \in \mathbf{AExp}_p$, $x \in \mathbf{Var}$, $\theta \in D$ *and* $i \in \mathbf{Pid}$ *such that* $\theta[i] \neq \mathbf{0}$, $([x \leftarrow e]\,\theta)[i] = \theta[i][x \leftarrow \mathcal{A}[\![e]\!]^i\,\theta[i]]$
Proof: Immediate from the definition of $[x \leftarrow e]$. □

Lemma 11 (Semantics partial simulation). *For all* $s \in \mathbf{Par}$, $Psim([\![s]\!], s)$.
Proof: By structural induction on s. We take $\theta \in D_\circ$ such $[\![s]\!]\,\theta \notin \{\bot, \Omega_S\}$ and an $i \in \mathbf{Pid}$ such that $\theta[i] \neq \mathbf{0}$ and show $\langle s, \theta[i]\rangle \to^i \langle Ok, ([\![s]\!]\,\theta)[i]\rangle$.

- $s \equiv \texttt{skip}$. Then $[\![\texttt{skip}]\!]\,\theta = \theta$, and we also have $\langle \texttt{skip}, \theta[i]\rangle \to^i \langle Ok, \theta\,i\rangle$ by the SKIP-rule.
- $s \equiv \texttt{sync}$. Vacuous since $[\![\texttt{sync}]\!]\,\theta = \Omega_S$ since $\theta \notin D_{\mathbf{Pid}}$ and $\theta \notin D_\emptyset$.
- $s \equiv x := e$. Then

  $$[\![x := e]\!]\,\theta[i] = ([x \leftarrow e]\,\theta)\,i = \theta[i][x \leftarrow \mathcal{A}[\![e]\!]^i\theta[i]]$$

  by Lemma 10, and the result follows by rule ASSIGN.
- $s \equiv s_1; s_2$. By the induction hypothesis, $Psim([\![s_1]\!], s_1)$ and $Psim([\![s_2]\!], s_2)$. If $[\![s_1]\!]\,\theta = \Omega_S$ then $[\![s]\!]\,\theta = \Omega_S$ so assume not. Furthermore, since the semantic function of statements is stable by Lemma 4, $([\![s_1]\!]\,\theta)[i] \neq \mathbf{0}$ Then

  $$\begin{array}{rcll} \langle s_1, \theta[i]\rangle & \to^i & \langle Ok, ([\![s_1]\!]\,\theta)[i]\rangle & \text{and} \\ \langle s_2, ([\![s_1]\!]\,\theta)[i]\rangle & \to^i & \langle Ok, ([\![s_2]\!]\,([\![s_1]\!]\,\theta))[i]\rangle & \end{array}$$

  and so

  $$\langle s_1; s_2, \theta[i]\rangle \longrightarrow \langle Ok, [\![s]\!]\,\theta[i]\rangle$$

- $s \equiv \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ Then

  $$[\![s]\!] = \lambda\theta.[\![s_1]\!]\,(\partial_b\,\theta) \parallel [\![s_2]\!]\,(\partial_{!b}\,\theta)$$

  Either $(\partial_b\,\theta)[i] \neq \mathbf{0}$ or $(\partial_{!b}\,\theta)[i] \neq \mathbf{0}$. Assume the former, the latter case is symmetric. Furthermore, assume that $[\![s_2]\!]\,(\partial_{!b}\,\theta) \neq \Omega_S$, since otherwise $[\![s]\!]\,\theta = \Omega_S$. By the induction hypothesis, since $(\partial_b\,\theta)[i] \neq \mathbf{0}$, we have

  $$\langle s_1, (\partial_b\,\theta)[i]\rangle \to^i \langle Ok, ([\![s_1]\!]\,(\partial_b\,\theta))[i]\rangle$$

Then by the rule IF-TT, knowing that $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt}$,

$$\langle \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, \theta[i]\rangle \to^i \langle Ok, ([\![s_1]\!]\ (\partial_b\,\theta))[i]\rangle$$

and since $([\![s]\!]\,\theta)[i] = ([\![s_1]\!]\ (\partial_b\,\theta))[i]$ (since $(\partial_b\,\theta)[i] \neq \mathbf{0}$) we conclude.

- $s \equiv \mathtt{while}\ b\ \mathtt{do}\ s_1\ \mathtt{end}$. Then $[\![s]\!] = fix\ F = \bigsqcup\{F^n\,f_\perp \mid n \geq 0\}$.

  By Lemma 9, since $Psim([\![s_1]\!], s_1)$ by the induction hypothesis, we have $Psim(F^n\,f_\perp, s)$ for all n.

  Now if $[\![s]\!]\,\theta \neq \perp$, then there is n such that $(F^n\,f_\perp)\ \theta = [\![s]\!]\,\theta$, so

  $$\langle s, \theta[i]\rangle \to^i \langle Ok, ([\![s]\!]\,\theta)[i]\rangle$$

  by $Psim(F^n\,f_\perp, s)$.

□

Theorem 1. *Let $\theta \in D_{\mathbf{Pid}}$ be an unmasked environment vector. If $[\![s]\!]\,\theta = \theta' \notin \{\perp, \Omega_S\}$, then $\langle\langle s\rangle_i, \theta\rangle \longrightarrow \theta'$.*

Proof: By structural induction on s.

- $s \equiv \mathtt{skip}$. Then $[\![\mathtt{skip}]\!]\,\theta = \theta$, and since $\forall i \in \mathbf{Pid}, \langle \mathtt{skip}, \theta[i]\rangle \to^i \langle Ok, \theta[i]\rangle$, we obtain $\langle\langle \mathtt{skip}\rangle_i, \theta\rangle \longrightarrow \theta$ by applying the ALL-OK-rule.

- $s \equiv \mathtt{sync}$. Then $[\![s]\!]\,\theta = \theta$ since $\theta \in D_{\mathbf{Pid}}$. By the rule sync,

  $$\forall i \in \mathbf{Pid}, \langle \mathtt{sync}, \theta[i]\rangle \to^i \langle Wait(\mathtt{skip}), \theta[i]\rangle$$

  and by the rule of SKIP,

  $$\forall i \in \mathbf{Pid}, \langle \mathtt{skip}, \theta[i]\rangle \to^i \langle Ok, \theta[i]\rangle$$

  so by applying the ALL-WAIT and then the ALL-OK-rule, we obtain $\langle\langle s\rangle_i, \theta\rangle \longrightarrow \theta$.

- $s \equiv x := e$. By applying the ALL-OK-rule and showing that

  $$\forall i \in \mathbf{Pid}, \langle x := e, \theta[i]\rangle \to^i \langle Ok, [\![x := e]\!]\theta[i]\rangle$$

which is simple, since by Lemma 10

$$\begin{aligned}(\llbracket x := e \rrbracket\ \theta)[i] &= ([x \leftarrow e]\ \theta)[i] \\ &= \theta[i][x \leftarrow \mathcal{A}\llbracket e \rrbracket^i \theta[i]]\end{aligned}$$

And by the rule ASSIGN,

$$\langle x := e, \theta\, i\rangle \rightarrow^i \langle Ok, (\theta[i])[X \leftarrow \mathcal{A}\llbracket e \rrbracket^i\, \theta[i]]\rangle$$

for any i.

- $s \equiv s_1; s_2$. Then $\llbracket s \rrbracket = \llbracket s_2 \rrbracket \circ \llbracket s_1 \rrbracket$. Since $\llbracket s \rrbracket\, \theta \notin \{\bot, \Omega_S\}$, $\llbracket s_1 \rrbracket\, \theta \notin \{\bot, \Omega_S\}$. By the induction hypothesis twice we obtain

$$\begin{aligned}\langle\langle s_1\rangle_i, \theta\rangle &\longrightarrow \llbracket s_1 \rrbracket\, \theta \qquad \text{and} \\ \langle\langle s_2\rangle_i, \llbracket s_1 \rrbracket\, \theta\rangle &\longrightarrow \llbracket s_2 \rrbracket(\llbracket s_1 \rrbracket\, \theta)\end{aligned}$$

  The result follows by Lemma 5.

- $s \equiv \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ Then

$$\llbracket s \rrbracket = \lambda\theta.\llbracket s_1 \rrbracket\, (\partial_b\, \theta) \parallel \llbracket s_2 \rrbracket\, (\partial_{!b}\, \theta)$$

  Either

  - $\partial_b\, \theta = \theta \in D_{\mathbf{Pid}}$ and $\partial_{!b}\, \theta = \theta' \in D_\emptyset$. Since $\llbracket s \rrbracket \neq \{\bot, \Omega_S\}$, $\llbracket s_2 \rrbracket \neq \{\bot, \Omega_S\}$. Then by Lemma 4, $\llbracket s_2 \rrbracket\, \theta' = \theta'' \in D_\emptyset$ for some θ''. Then $\llbracket s \rrbracket\, \theta = \llbracket s_1 \rrbracket\, \theta$ by Lemma 8.

    By the induction hypothesis

$$\langle\langle s_1\rangle_i, \theta\rangle \longrightarrow \llbracket s_1 \rrbracket\, \theta$$

    We know $\forall i \in \mathbf{Pid}, \mathcal{B}\llbracket b \rrbracket^i\, \theta[i] = \texttt{tt}$, since $\partial_b\, \theta = \theta$. Then by Lemma 7,

$$\langle\langle s\rangle_i, \theta\rangle \rightarrow \llbracket s_1 \rrbracket\, \theta = \llbracket s \rrbracket\, \theta$$

  - $\partial_{!b}\, \theta = \theta \in D_{\mathbf{Pid}}$ and $\partial_b\, \theta = \theta' \in D_\emptyset$. This case is symmetric to the previous.
  - $\partial_b\, \theta \notin D_{\mathbf{Pid}}$ and $\partial_{!b}\, \theta \notin D_{\mathbf{Pid}}$. We apply the ALL-OK rule and show

$$\forall i \in \mathbf{Pid}, \langle s, \theta[i]\rangle \rightarrow^i \langle Ok, \llbracket s \rrbracket\, \theta[i]\rangle$$

Take any $i \in \mathbf{Pid}$. Assume $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt}$, the other case being symmetric, and we then have $(\partial_b\,\theta)\,i = \theta[i]$ and $[\![s]\!]\,\theta[i] = [\![s_1]\!]\,(\partial_b\,\theta)\,i$. By Lemma 11,

$$\begin{aligned} &\langle s_1, (\partial_b\,\theta)\,i\rangle \to^i \langle Ok, [\![s_1]\!]\,(\partial_b\,\theta)\,i\rangle \\ \iff &\langle s_1, \theta[i]\rangle \to^i \langle Ok, [\![s]\!]\,\theta[i]\rangle \end{aligned}$$

and so by the IF-TT-rule, $\langle s, \theta[i]\rangle \to^i \langle Ok, [\![s]\!]\,\theta[i]\rangle$.

- $s \equiv \mathtt{while}\,b\,\mathtt{do}\,s_1\,\mathtt{end}$. Then $[\![s]\!] = fix\ F = \bigsqcup\{F^n\,f_\perp \mid n \geq 0\}$. We proceed by induction on n, to show

$$\forall n \in \mathbf{Nat}, \theta \in D_{\mathbf{Pid}}, (F^n\,f_\perp)\,\theta = \theta' \notin \{\perp, \Omega_S\} \implies \langle\langle s\rangle_i, \theta\rangle \longrightarrow \theta'$$

  - $n = 0$. Holds vacuously since $(F^0 f_\perp)\,\theta = \perp$.
  - $n + 1$. Either
    - $\partial_b\,\theta = \theta \in D_{\mathbf{Pid}}$ and $\partial_{!b}\,\theta \in D_\emptyset$. Then $(F^{n+1}\,f_\perp)\,\theta = (F^n\,f_\perp)([\![s_1]\!]\,\theta)$. Let $\theta'' = [\![s_1]\!]\,\theta$. If $\theta'' \in \{\perp, \Omega_S\}$, then $\theta' \in \{\perp, \Omega_S\}$ and we are done. Otherwise, we have by the outer induction hypothesis, $\langle\langle s_1\rangle_i, \theta\rangle \longrightarrow \theta''$. Since the semantic function of statements is stable by Lemma 4, $\theta'' \in D_{\mathbf{Pid}}$, and so by the inner induction hypothesis, $\langle\langle s\rangle_i, \theta''\rangle \longrightarrow \theta'$. We conclude by Lemma 6.
    - $\partial_{!b}\,\theta = \theta \in D_{\mathbf{Pid}}$ and $\partial_b\,\theta \in D_\emptyset$. Then $(F^{n+1}\,f_\perp)\,\theta = \theta$. To show $\langle\langle s\rangle_i, \theta\rangle \longrightarrow \theta$, we apply the global ALL-OK-rule followed by the IF-FF local rule.
    - $\partial_b\ \theta \notin D_{\mathbf{Pid}}$ and $\partial_{!b}\,\theta \notin D_{\mathbf{Pid}}$. We apply the ALL-OK and show for all
$$\forall i \in \mathbf{Pid}, \langle s, \theta[i]\rangle \to^i \langle Ok, \theta'[i]\rangle$$
      We take some i, and do case distinction on the evaluation of the condition:
      - Either $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{ff}$. Then $\theta'[i] = \theta[i]$. Apply the WH-FF-rule to get $\langle s, \theta[i]\rangle \to^i \langle Ok, \theta[i]\rangle$.
      - Or $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt}$. Then
$$\theta'[i] = ((F^n\,f_\perp) \circ [\![s_1]\!])\,(\partial_b\,\theta)\,i$$

Apply the WH-TT-OK-rule and show:

(i) $\langle s_1, (\partial_b\,\theta)[i]\rangle \to^i \langle Ok, \theta''[i]\rangle$, and

(ii) $\langle s, \theta''[i]\rangle \to^i \langle Ok, \theta'[i]\rangle$, where $\theta'' = [\![s_1]\!]\,(\partial_b\,\theta)$.
(i) Clearly $\partial_b\,\theta \notin \{\bot, \Omega_S\} \cup D_{\mathbf{Pid}}$. By Lemma 11, we have $\langle s_1, (\partial_b\,\theta)[i]\rangle \to^i \langle Ok, \theta''[i]\rangle$.
(ii) Again $\theta'' \notin \{\bot, \Omega_S\} \cup D_{\mathbf{Pid}}$, since $[\![s]\!]\,\theta \notin \{\bot, \Omega_S\}$, and since the semantic function of statements is stable by Lemma 4. By Lemma 9 we obtain:

$$\langle s, \theta''[i]\rangle \to^i \langle Ok, (F^n\,f_\bot)\,\theta''[i]\rangle$$

and since $(F^n\,f_\bot)\,\theta''[i] = (F^n\,f_\bot)([\![s_1]\!]\,(\partial_b\,\theta))i = \theta'[i]$, we obtain

$$\langle s, \theta''[i]\rangle \to^i \langle Ok, \theta'[i]\rangle$$

The conclusion of the applied WH-TT-OK with the premises (i) and (ii) is

$$\langle s, (\partial_b\,\theta)\,i\rangle \to^i \langle Ok, \theta'[i]\rangle \iff \langle s, \theta[i]\rangle \to^i \langle Ok, \theta'[i]\rangle$$

since $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathtt{tt}$. This concludes this case.

□

A.2 Correctness of PI

Some general remarks about the relationship $\sim$ (defined in Definition 2) we want to be preserved by the analysis:

Definition 9. *Let $\sigma \sim_V \sigma'$ be the binary version of $\sim_V$ for local environments i.e. $\sigma \sim_V \sigma' \iff \forall x \in V, \sigma\,x = \sigma'\,x$. Note that $\sim_V$ is an equivalence relation.*

Lemma 12. *If $\sigma \sim_V \sigma'$ and $V \supseteq V'$ then $\sigma \sim_{V'} \sigma'$.*
Proof: Follows from the definition of $\sim_V$. □

Lemma 13. *If $\phi^d(e, V)$ for $e \in \mathbf{AExp}_p$ and $\sigma \sim_V \sigma'$ then $\exists v \in \mathbf{Nat}, \forall i, j \in \mathbf{Pid}, \mathcal{A}[\![e]\!]^i \sigma = \mathcal{A}[\![e]\!]^j \sigma' = v$. Similarly for boolean expressions, if $\phi^d(b, V)$ for $b \in \mathbf{BExp}$ and $\sigma \sim_V \sigma'$ then $\exists v \in \mathbf{Bool}, \forall i, j \in \mathbf{Pid}, \mathcal{B}[\![b]\!]^i \sigma = \mathcal{B}[\![b]\!]^j \sigma' = v$.*

Proof: We proceed by structural induction for $e \in \mathbf{AExp}_p$ on numerical expressions. If e is a constant (or `nprocs`), then n is that constant (or $\mathbf{p}$). The case where e is the special variable `pid` is absurd, since $\phi^d(e, V)$ forbids it. If e is the variable x, then we note that $x \in V$ since $\phi^d(e, V)$. The result then follows from $\sigma \sim_V \sigma'$. If e is some operation $e_1\ op\ e_2$ with $op \in \{+, -, \times\}$, then we apply the induction hypothesis on e_1 and e_2, and result follows from the fact that all operators are functions (and thus single-valued).

The same result can be shown for all $b \in \mathbf{BExp}$, by structural induction and by using the result on expressions in $\mathbf{AExp}_p$. □

A.2.1 Domain

We assume a fixed program $s_\star$. We assume the existence of a final edge for each node corresponding to a statement in the CFG, excluding merge nodes. This edge is given by $final_e^{s_\star}(s)$, defined below. However, the final node of the program, $finals_\star$ has no successor and hence no final edge by definition. To remedy, we add a dummy node labeled $\ell_\mathcal{F}$, that corresponds to no statement. We then define $final_e^{s_\star}(s)$:

Definition 10. *The final edge of the sub-program s of $s_\star$ is given by $final_e^{s_\star}(s)$:*

$$\begin{cases} final_e^{s_\star} & : \quad \mathbf{Par} \to \mathbf{Lab} \times \mathbf{Lab} \qquad\qquad s_\star \in \mathbf{Par} \\ final_e^{s_\star}(s) & = \begin{cases} (final(s), \ell') & \text{if } \exists \ell' \in \mathbf{Lab}, (final(s), \ell') \in flow(s_\star) \\ (inits, \ell_\mathcal{F}) & \text{otherwise} \end{cases} \end{cases}$$

In what follows, we will drop the superscript on $final_e$, and assume one fixed program $s_\star$.

A.2.2 Parameterized Constraint System

Definition 11. *$PI^\iota(s)$ is a constraint system parameterized by $\iota \in L$, so that:*

$$(pi_\circ, pi_\bullet) \vDash PI^\iota(s) \implies pi_\circ(init(s)) \sqsupseteq \iota$$

All other constraint in the parameterized system is as in the non-parameterized. The non-parameterized constraint system $PI(s)$ is equivalent to $PI^\iota(s)$ with $\iota = (\mathbf{Var}_s, \epsilon)$.

A.2.3 Constraint System Facts

Definition 12. *The subprograms $sub(s)$ of s are:*

$$sub(s) = \begin{cases} \{s_1, s_2\} & \text{if } c = s_1; s_2 \\ \{s_1, s_2\} & \text{if } c = \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \\ \{s_1\} & \text{if } c = \texttt{while}\ b\ \texttt{do}\ s_1\ \texttt{end} \\ \emptyset & \text{otherwise} \end{cases}$$

Lemma 14 (Solution extends to sub-programs). *If $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$ and $s' \in sub(s)$ and $\iota' = pi_\circ(init(s'))$ then $(pi_\circ, pi_\bullet) \vDash PI^{\iota'}(s')$.*
Proof: The labels and edges of s' are subsets of those of s. Except for $pi_\circ(init(s'))$, any constraint on them is the same as in $PI^\iota(s)$. The constraint on $init(s')$ must hold by construction of $PI^{\iota'}(s')$. □

Lemma 15 (Incoming state is coarser than incoming edge). *If $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$ and $\ell \in labels(s)$ such that ℓ is not a merge node, then $pi_\circ(\ell) \sqsupseteq pi_\bullet(\mathtt{n}(\ell))$.*
Proof: By structural induction on s. All cases follows trivially from the construction of the constraint system, except when $s = \texttt{while}\ [b]^\ell\ \texttt{do}\ s_1\ \texttt{end}$. In this case, the result follows from Lemma 19. □

Lemma 16 (Constraint system facts). *Let $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$. We then have:*

(i) *If $s \equiv s_1; s_2$, then $pi_\circ(init(s_2)) \sqsubseteq pi_\bullet(final_e(s_1))$.*

(ii) *$s \equiv \texttt{if}\ [b]^\ell\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$, then $pi_\circ(init(s_1)) \sqsubseteq pi_\bullet(\mathtt{t}(\ell))$ and $pi_\circ(init(s_2)) \sqsubseteq pi_\bullet(\mathtt{f}(\ell))$.*

(iii) *$s \equiv \texttt{while}\ [b]^\ell\ \texttt{do}\ s_1\ \texttt{end}$, then $pi_\circ(init(s_1)) \sqsubseteq pi_\bullet(\mathtt{t}(\ell))$.*

Proof: In the three cases respectively, this is equivalent to showing $pi_\circ(init(s_i)) \sqsubseteq pi_\bullet(\mathtt{n}(init(s_i)))$, for $i = 2$, $i = 1$, $i = 2$, and $i = 1$ respectively. This follows from the from the construction of the control flow graph, and in the first case, from the definition of $final_e$. Now all cases are proved by applying Lemma 15. □

A.2.4 Marked Path Abstractions and *pid*-independent Variables

Lemma 17 (Path concatenation is monotone). *For all $p, p' \in \mathbf{Path}^\sharp$ and $\ell \in \mathbf{Lab}$, If $p \preceq p'$ then $p.\ell \preceq p'.\ell$.*
Proof: Since, $p \preceq p'$, either:

- $p = \bot$, and so $p.\ell = \bot.\ell = \bot \preceq p'.\ell$.
- $p = p' = \epsilon$, $p.\ell = \epsilon : \ell = \ell \preceq \ell = \epsilon : \ell = p'.\ell$
- $p = p_0 : \ell_0$, $p' = p'_0 : \ell'_0$ and $p_0 \preceq p'_0$. Then $p.\ell = p : \ell \preceq p' : \ell = p'.\ell$.
- $p' = \top$, and so $p.\ell \preceq \top = \top.\ell = p'.\ell$.

□

Lemma 18 (Path merging is monotone)**.** *For all p_0, p_1 and p_2, if $p_0 \preceq p_1$ then $p_0 \triangledown p_2 \preceq p_1 \triangledown p_2$.*
Proof: By case distinction on $p_0 \preceq p_1$, and then case distinction on p_2. □

Lemma 19 (Merge Concat Order)**.** *For all ℓ, p and p_0, $p \preceq p.\ell \triangledown p_0$.*
Proof: If p or p_0 are in $\{\bot, \top\}$ the conclusion is immediate. Otherwise, we do case distinction on whether $p.\ell \succeq p_0$, $p.\ell \preceq p_0$ or neither. In the first two cases, the result follows by the monotonicity of $\sqcup$. In the latter, the result follows by the definition of $\triangledown$. □

Definition 13. *A path $p \in \mathbf{Path}^\sharp$ is **marked** if it is $\top$ or if it contains any marked labels. Formally, $p \notin (\{\bot\} \cup \mathbf{Lab}^*)$.*

Definition 14. *Let assigns : $\mathbf{Par} \to \mathcal{P}(\mathbf{Var})$ return the set of variable appearing on the left-hand side of an assignment the given program.*

Lemma 20 (Marked path kills all assigned)**.** *If $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$ and $\pi^2(pi_\circ(init(s)))$ is marked, then $\pi^1(pi_\bullet(final_e(s))) \subseteq \pi^1(pi_\circ(init(s))) \setminus assigns(s)$.*
Proof: Let $(V, p) = pi_\circ(init(s))$ and $(V', p') = pi_\bullet(final_e(s))$. We prove the stronger property $(V', p') \sqsupseteq (V \setminus assigns(s), p)$ by structural induction on s.

- $s \equiv [\texttt{skip}]^\ell$ and $s \equiv [\texttt{sync}]^\ell$. The constraint system gives us the following inequality:

  $$(V', p') = pi_\bullet(\mathtt{x}(\ell)) \sqsupseteq pi_\circ(\ell) = (V, p)$$

  Then since $assigns(s) = \emptyset$, we have $(V', p') \sqsupseteq (V \setminus assigns(s), p)$.

- $s \equiv [x{:=}e]^\ell$. The constraint system gives us the following inequalities:

  $$pi_\circ(\ell) = (V, p)$$
  $$(V', p') = pi_\bullet(\mathtt{x}(\ell)) \sqsupseteq (vdep(pi_\circ(\ell), e, \mathtt{X}), \pi^2(pi_\circ(\ell)))$$

In this case, note that p is marked, and thus $\phi^c(p)$ does not hold. Thus

$$vdep((V,p),e,x) = V \setminus \{x\} \qquad (*)$$

We then have:

$$\begin{array}{rl} (V',p') = pi_\bullet(\mathtt{x}(\ell)) & \text{by definition} \\ \sqsupseteq (V \setminus \{x\}, p) & \text{by } (*) \text{ and constraint system} \\ = (V \setminus assigns(s), p) & \text{since } assigns(s) = \{X\} \end{array}$$

- $s \equiv s_1; s_2$. Let $(V_1', p_1') = pi_\bullet(final_e(s_1)) = pi_\bullet(\mathtt{n}(init(s_2)))$, and $(V_2, p_2) = pi_\circ(init(s_2))$. Note that $(V_2, p_2) \sqsupseteq (V_1', p_1')$ by Lemma 16. Since $(pi_\circ, pi_\bullet) \vDash PI^\iota(s_1; s_2)$, we have $(pi_\circ, pi_\bullet) \vDash PI^\iota(s_1)$ and $(pi_\circ, pi_\bullet) \vDash PI^{\iota'}(s_2)$ with $\iota' = pi_\circ(init(s_2))$. We then have:

$$\begin{array}{rl} (V',p') \sqsupseteq (V_2 \setminus assigns(s_2), p_2) & \text{by the induction hypothesis} \\ \sqsupseteq (V_1' \setminus assigns(s_2), p_1') & \text{by the constraint system} \\ \sqsupseteq ((V \setminus assigns(s_1)) \setminus assigns(s_2), p) & \text{by the induction hypothesis} \\ = (V \setminus assigns(s), p) & \text{by definition of } assigns \end{array}$$

- $s \equiv \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}$. Let

$$\begin{array}{ll} (V_1, p_1) = pi_\circ(init(s_1)) & (V_1', p_1') = pi_\bullet(final_e(s_1)) \\ (V_2, p_2) = pi_\circ(init(s_2)) & (V_2', p_2') = pi_\bullet(final_e(s_2)) \end{array}$$

The constraint system gives the following inequalities:

$$\begin{array}{lcl} (V,p) & = & pi_\circ(\ell) \\ (V_1,p_1) \sqsupseteq pi_\bullet(\mathtt{t}(\ell)) & \sqsupseteq & (\pi^1(pi_\circ(\ell)), \pi^2(pi_\circ(\ell)).cdep(\ell, b, \pi^1(pi_\circ(\ell)))) \\ (V_2,p_2) \sqsupseteq pi_\bullet(\mathtt{f}(\ell)) & \sqsupseteq & (\pi^1(pi_\circ(\ell)), \pi^2(pi_\circ(\ell)).cdep(\ell, b, \pi^1(pi_\circ(\ell)))) \\ pi_\circ(\ell^m) & \sqsupseteq & (\pi^1(pi_\bullet(\mathtt{t}(\ell^m))) \cap \pi^1(pi_\bullet(\mathtt{f}(\ell^m))), \\ & & \pi^2(pi_\bullet(\mathtt{t}(\ell^m))) \triangledown \pi^2(pi_\bullet(\mathtt{f}(\ell^m)))) \\ & = & (V_1' \cap V_2', p_1' \triangledown p_2') \\ (V',p') = pi_\bullet(\mathtt{x}(\ell^m)) & \sqsupseteq & pi_\circ(\ell^m) \end{array}$$

Since $(pi_\circ, pi_\bullet) \vDash PI^{\iota}(s)$, we have $(pi_\circ, pi_\bullet) \vDash PI^{\iota_1}(s_1)$ and $(pi_\circ, pi_\bullet) \vDash PI^{\iota_2}(s_2)$ with $\iota_1 = pi_\circ(init(s_1))$ and $\iota_2 = pi_\circ(init(s_2))$. The induction hypothesis then gives us the following:

$$(V_1', p_1') \sqsupseteq (V_1 \setminus assigns(s_1), p_1) \tag{1}$$
$$(V_2', p_2') \sqsupseteq (V_2 \setminus assigns(s_2), p_2) \tag{2}$$

We show (i) $V' \subseteq V \setminus assigns(s)$ and (ii) $p' \succeq p$ separately.

(i) We have:

$$\begin{aligned}
V' &\subseteq (V_1' \cap V_2') && \text{by constraint system} \\
&\subseteq (V_1 \setminus assigns(s_1)) \cap (V_2 \setminus assigns(s_2)) && \text{by (1) and (2)} \\
&\subseteq (V \setminus assigns(s_1)) \cap (V \setminus assigns(s_2)) && \text{since } V_1 \subseteq V \text{ and } V_2 \subseteq V \\
&\subseteq V \setminus assigns(s) && \text{by definition of } assigns
\end{aligned}$$

(ii) Let $\ell' = cdep(\ell, b, \pi^1(pi_\circ(\ell)))$. Then

$$p_1' \succeq p_1 \succeq p.\ell' \tag{$*$}$$

by (1) and the constraint system. So we have

$$\begin{aligned}
p' &\succeq p_1' \triangledown p_2' && \text{by constraint system} \\
&\succeq p.\ell \triangledown p_2' && \text{by } (*) \text{ and Lemma 18} \\
&\succeq p && \text{by Lemma 19}
\end{aligned}$$

By (i) and (ii), we conclude $(V', p') \sqsupseteq (V \setminus assigns(s), p)$ as desired.

- $s \equiv \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end}$. Let

$$(V_0, p_0) = pi_\bullet(\mathtt{n}(\ell)) \quad (V_1, p_1) = pi_\circ(init(s_1)) \quad (V_1', p_1') = pi_\bullet(final_e(s_1))$$

The constraint system gives the following inequalities:

$$\begin{array}{lcl}
(V,p) = pi_\circ(\ell) & \sqsupseteq & (\pi^1(pi_\bullet(\mathtt{n}(\ell))) \cap \pi^1(pi_\bullet(\mathtt{b}(\ell))), \\
& & \pi^2(pi_\bullet(\mathtt{n}(\ell))).\ell \triangledown \pi^2(pi_\bullet(\mathtt{b}(\ell)))) \\
& = & (V_0 \cap V_1', p_0.\ell \triangledown p_1') \\
(V_1,p_1) \sqsupseteq pi_\bullet(\mathtt{t}(\ell)) & \sqsupseteq & (\pi^1(pi_\circ(\ell)), \pi^2(pi_\circ(\ell)).cdep(\ell, b, \pi^1(pi_\circ(\ell)))) \\
(V',p') = pi_\bullet(\mathtt{f}(\ell)) & \sqsupseteq & (\pi^1(pi_\circ(\ell)), \pi^2(pi_\circ(\ell)))
\end{array}$$

Since $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$, we have $(pi_\circ, pi_\bullet) \vDash PI^{\iota_1}(s_1)$ with $\iota_1 = pi_\circ(init(s_1))$. The induction hypothesis then gives us the following:

$$(V_1', p_1') \sqsupseteq (V_1 \setminus assigns(s_1), p_1) \tag{1}$$

We show (i) $V' \subseteq V \setminus assigns(s)$ and (ii) $p' \succeq p$ separately.

(i) We have:

$$\begin{array}{lr}
V' \subseteq \pi^1(pi_\circ(\ell)) = V & \\
\quad \subseteq V_0 \cap V_1' & \text{by the constraint system} \\
\quad \subseteq V_1' & \\
\quad \subseteq V_1 \setminus assigns(s_1) & \text{by (1)} \\
\quad \subseteq V \setminus assigns(s_1) & \text{since } V_1 \subseteq \pi^1(pi_\circ(\ell)) = V \\
\quad = V \setminus assigns(s) & \text{by definition of } assigns
\end{array}$$

(ii) We have

$$p' \succeq \pi^2(pi_\circ(\ell)) = p \qquad \text{by the constraint system}$$

By (i) and (ii), we conclude $(V', p') \sqsupseteq (V \setminus (assigns(s), p))$ as desired.

□

A.2.5 Correctness of the Analysis

We now turn to the correctness of the analysis:

Lemma 21. *If* $\langle s, \sigma\rangle \to^i \langle t, \sigma'\rangle$ *then* $\forall V \subseteq \mathbf{Var}, \sigma \sim_{V \setminus assigns(s)} \sigma'$.

Proof: By a trivial rule induction on $\langle s, \sigma\rangle \to^i \langle t, \sigma'\rangle$. □

Lemma 22. *If* $\langle s, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle$, $\langle s, \sigma_j\rangle \to^j \langle Ok, \sigma_j'\rangle$, $\sigma_i \sim_V \sigma_j$, *and* $V' \subseteq V \setminus assigns(s)$ *then* $\sigma_i' \sim_{V'} \sigma_j'$.
Proof: By Lemma 21, $\sigma_i \sim_{V'} \sigma_i'$ and $\sigma_j \sim_{V'} \sigma_j'$. Since $V' \subseteq V$, we also have $\sigma_i \sim_{V'} \sigma_j$ by Lemma 12. The result follows from the transitivity of $\sim_{V'}$. □

Theorem 6. *Let* $(pi_\circ, pi_\bullet) \vDash PI^\iota(s)$, *where* $\pi^2(\iota) \neq \bot$. *Let* $V = \pi^1(pi_\circ(init(s)))$, $V' = \pi^1(pi_\bullet(final_e(s)))$, *and* σ_i, σ_j *two environments such that* $\sigma_i \sim_V \sigma_j$, *and* $\langle s, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle$ *and* $\langle s, \sigma_j\rangle \to^j \langle Ok, \sigma_j'\rangle$. *Then* $\sigma_i' \sim_{V'} \sigma_j'$.
Proof: Proof by rule induction on $\langle s, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle$.

SKIP Then $s \equiv [\texttt{skip}]^\ell$. By the constraint system, we have:

$$V = \pi^1(pi_\circ(\ell)) \supseteq \pi^1(pi_\bullet(\mathtt{x}(\ell))) = V'$$

Since $\sigma_i' = \sigma_i$ and $\sigma_j' = \sigma_j$, by the rule for SKIP, and since $\pi^1(pi_\circ(\ell)) \supseteq \pi^1(pi_\bullet(\mathtt{x}(\ell)))$, we have $\sigma_i' \sim_{V'} \sigma_j'$ by Lemma 12.

SYNC Vacuous since $\langle s, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle$.

ASSIGN Then $s \equiv [x{:=}e]^\ell$. The predicate $\phi^d(e, \pi^1(pi_\circ(\ell))) \wedge \phi^c(\pi^2(pi_\circ(\ell)))$ gives rise to two cases:

- It does not hold. Then we have the following constraint after simplification of *vdep*:

$$V = \pi^1(pi_\circ(\ell))$$
$$\pi^1(pi_\circ(\ell)) \setminus \{x\} \supseteq \pi^1(pi_\bullet(\mathtt{x}(\ell))) = V'$$

  So $V' \subseteq V \setminus \{x\}$. Since $assigns(s) = \{x\}$, we conclude by Lemma 22.
- It holds. By Lemma 13, for some n, $\mathcal{A}[\![e]\!]^i \sigma_i = \mathcal{A}[\![e]\!]^j \sigma_j = n$. The constraint system is then

$$V = \pi^1(pi_\circ(\ell))$$
$$\pi^1(pi_\circ(\ell)) \cup \{x\} \supseteq \pi^1(pi_\bullet(\mathtt{x}(\ell))) = V'$$

  So $V' \subseteq V \cup \{x\}$. To show $\sigma_i' \sim_{V'} \sigma_j'$, take a $y \in V'$. Either $y = x$, and then $\sigma_i' y = \sigma_j' y = n$, or $y \neq x$ and then $\sigma_i' y = \sigma_i y$ and $\sigma_j' y = \sigma_j y$. Since $y \in V$ and $\sigma_i \sim_V \sigma_j$, we have $\sigma_i' y = \sigma_j' y$.

SEQ-OK In this case $s \equiv s_1; s_2$, and

$$\langle s_1, \sigma_i\rangle \to^i \langle Ok, \sigma_i''\rangle \qquad \langle s_2, \sigma_i''\rangle \to^i \langle Ok, \sigma_i'\rangle$$
$$\langle s_1, \sigma_j\rangle \to^j \langle Ok, \sigma_j''\rangle \qquad \langle s_2, \sigma_j''\rangle \to^j \langle Ok, \sigma_j'\rangle.$$

We apply the induction hypothesis to the execution of s_1. Let $V_1 = \pi^1(pi_\circ(init(s_1)))$, and $V_1' = \pi^1(pi_\bullet(final_e(s_1)))$. Note that since $init(s_1; s_2) = init(s_1)$, $V_1 = V$ and $\sigma_i \sim_{V_1} \sigma_j$. With $(pi_\circ, pi_\bullet) \vDash PI^{\iota'}(s_1)$ where $\iota' = pi_\circ(init(s_1))$, we obtain $\sigma_i'' \sim_{V_1'} \sigma_j''$.

Let $V_2 = \pi^1(pi_\circ(init(s_2)))$, and $V_2' = \pi^1(pi_\bullet(final_e(s_2)))$. By Lemma 16, $pi_\circ(init(s_2)) \sqsupseteq pi_\bullet(final_e(s_1))$, and so $V_2 \subseteq V_1'$. It follows by Lemma 12 that $\sigma_i'' \sim_{V_2} \sigma_j''$. We can then apply the induction hypothesis to s_2 and obtain $\sigma_i' \sim_{V_2'} \sigma_j'$. Since $final_e(s_1; s_2) = final_e(s_2)$, $V_2' = V'$ and we conclude $\sigma_i' \sim_{V'} \sigma_j'$.

SEQ-WAIT Vacuous since $\langle s, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle$.

IF-TT and IF-FF In these cases, $s \equiv \texttt{if}\ [b]^\ell\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ [\texttt{end}]^{\ell'}$. The predicate $\phi^d(b, \pi^1(pi_\circ(\ell)))$ gives rise to two cases:

- It holds, and then by Lemma 13 we have for some $v \in \mathbf{Bool}$, $\mathcal{B}[\![b]\!]^j\,\sigma_i = \mathcal{B}[\![b]\!]^j\,\sigma_j = v$. Assume $v = \texttt{tt}$, the other case being symmetric. Then

$$\langle s_1, \sigma_i\rangle \to^i \langle Ok, \sigma_i'\rangle \text{ and}$$
$$\langle s_1, \sigma_j\rangle \to^j \langle Ok, \sigma_j'\rangle$$

  Let $V_1 = \pi^1(pi_\circ(init(s_1)))$, and $V_1' = \pi^1(pi_\bullet(final_e(s_1)))$. Since $\pi^1(pi_\bullet(\texttt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) = V$, and by Lemma 16, $V_1 \subseteq \pi^1(pi_\bullet(\texttt{t}(\ell)))$, so $V_1 \subseteq V$ and since $\sigma_i \sim_V \sigma_j$, we have $\sigma_i \sim_{V_1} \sigma_j$. Since $(pi_\circ, pi_\bullet) \vDash PI^{\iota'}(s_1)$ with $\iota' = pi_\circ(init(s_1))$, we apply the induction hypothesis and obtain $\sigma_i' \sim_{V_1'} \sigma_j'$.

  By construction of the constraint system $V' = \pi^1(pi_\bullet(\texttt{x}(\ell'))) \subseteq \pi^1(pi_\circ(\ell')) \subseteq \pi^1(pi_\bullet(\texttt{t}(\ell'))) \cap \pi^1(pi_\bullet(\texttt{f}(\ell')))$ and by construction of the CFG, $\pi^1(pi_\bullet(\texttt{t}(\ell'))) = \pi^1(pi_\bullet(final_e(s_1))) = V_1'$, so $V' \subseteq V_1'$ and we obtain $\sigma_i' \sim_{V'} \sigma_j'$.

- It does not hold, then by the constraint system :

$$\pi^2(pi_\bullet(\texttt{t}(\ell))) \succeq \pi^2(pi_\circ(\ell)).\bar{\ell} \text{ and } \pi^2(pi_\bullet(\texttt{f}(\ell))) \succeq \pi^2(pi_\circ(\ell)).\bar{\ell}$$

and so $\pi^2(pi_\bullet(\mathtt{t}(\ell)))$ and $\pi^2(pi_\bullet(\mathtt{f}(\ell)))$ are marked. Then

$$\begin{aligned}&\pi^1(pi_\bullet(\mathtt{t}(\ell'))) \subseteq \pi^1(pi_\bullet(\mathtt{t}(\ell))) \setminus assigns(s_1) \text{ and}\\ &\pi^1(pi_\bullet(\mathtt{f}(\ell'))) \subseteq \pi^1(pi_\bullet(\mathtt{f}(\ell))) \setminus assigns(s_2) \qquad (*)\end{aligned}$$

by Lemma 20. Then, with $X_1 = assigns(s_1)$ and $X_2 = assigns(s_2)$, we have

$$\begin{aligned}V' &= \pi^1(pi_\bullet(\mathtt{x}(\ell')))\\ &\subseteq \pi^1(pi_\circ(\ell'))\\ &\subseteq \pi^1(pi_\bullet(\mathtt{t}(\ell'))) \cap \pi^1(pi_\bullet(\mathtt{f}(\ell')))\\ &\subseteq (\pi^1(pi_\bullet(\mathtt{t}(\ell))) \setminus X_1) \cap (\pi^1(pi_\bullet(\mathtt{f}(\ell))) \setminus X_2) && \text{by (*)}\\ &\subseteq (\pi^1(pi_\circ(\ell)) \setminus X_1) \cap (\pi^1(pi_\circ(\ell)) \setminus X_2)\\ &\qquad \text{since } \pi^1(pi_\bullet(\mathtt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) \text{ and } \pi^1(pi_\bullet(\mathtt{f}(\ell))) \subseteq \pi^1(pi_\circ(\ell))\\ &\subseteq \pi^1(pi_\circ(\ell)) \setminus (X_1 \cup X_2)\\ &\subseteq \pi^1(pi_\circ(\ell)) \setminus (assigns(s))\\ &= V \setminus (assigns(s))\end{aligned}$$

And so we conclude by Lemma 22.

WH-TT-OK and WH-FF. In these cases, $s \equiv \mathtt{while}\ [b]^\ell\ \mathtt{do}\ s_1\ \mathtt{end}$. The predicate $\phi^d(b, \pi^1(pi_\circ(\ell)))$ gives rise to two cases:

- It holds, and then for some $v \in \mathbf{Bool}$, $\mathcal{B}[\![b]\!]^j\,\sigma_i = \mathcal{B}[\![b]\!]^j\,\sigma_j = v$.
  - Assume $v = \mathtt{tt}$. Then for some σ_i'' and σ_j'', we have:

$$\begin{aligned}(A)\quad &\langle s_1, \sigma_i\rangle \to^i \langle Ok, \sigma_i''\rangle \qquad (B)\quad \langle s, \sigma_i''\rangle \to^i \langle Ok, \sigma_i'\rangle\\ &\langle s_1, \sigma_j\rangle \to^j \langle Ok, \sigma_j''\rangle \qquad\qquad \langle s, \sigma_j''\rangle \to^j \langle Ok, \sigma_j'\rangle\end{aligned}$$

    We know $(pi_\circ, pi_\bullet) \vDash PI^{\iota'}(s_1)$, with $\iota' = pi_\circ(init(s_1))$. Let $V_1 = \pi^1(pi_\circ(init(s_1)))$, and $V_1' = \pi^1(pi_\bullet(\mathtt{b}(\ell))) = \pi^1(pi_\bullet(final_e(s_1)))$. Since $pi_\circ(init(s_1)) \sqsupseteq pi_\bullet(\mathtt{t}(\ell))$ by Lemma 16, and $\pi^1(pi_\bullet(\mathtt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) = V$, we obtain $V_1 \subseteq V$.
    By Lemma 12, we have $\sigma_i \sim_{V_1} \sigma_j$. Then by applying the induction hypothesis on (A) we obtain $\sigma_i'' \sim_{V_1'} \sigma_j''$.
    Since $V_1' = \pi^1(pi_\bullet(\mathtt{b}(\ell)))$, and $V = \pi^1(pi_\circ(\ell)) \subseteq \pi^1(pi_\bullet(\mathtt{b}(\ell))) \cap \pi^1(pi_\bullet(\mathtt{n}(\ell)))$, we have $V \subseteq V_1'$. Thus $\sigma_i'' \sim_V \sigma_j''$.

We can now apply the induction hypothesis on (B) to obtain $\sigma_i' \sim_{V'} \sigma_j'$, which concludes this case.

* Assume $v = \texttt{ff}$. Then $\sigma_i' = \sigma_i$ and $\sigma_j' = \sigma_j$ and since $V' = \pi^1(pi_\bullet(\texttt{f}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) = V$ we conclude by Lemma 12.

– It does not hold, then by the constraint system

$$\pi^2(pi_\circ(init(s_1))) \succeq \pi^2(pi_\bullet(\texttt{t}(\ell))) \succeq \pi^2(pi_\circ(\ell)).\bar{\ell} \text{ and} \tag{A.1}$$
$$V = \pi^1(pi_\circ(init(s_1))) \subseteq \pi^1(pi_\bullet(\texttt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) \tag{A.2}$$

and so $\pi^2(pi_\circ(init(s_1)))$ is marked. Then

$$\begin{aligned}
V' &= \pi^1(pi_\bullet(\texttt{f}(\ell))) \\
&\subseteq \pi^1(pi_\circ(\ell)) \\
&\subseteq \pi^1(pi_\bullet(\texttt{b}(\ell))) && \text{by the constraint system} \\
&\subseteq \pi^1(pi_\circ(init(s_1))) \setminus assigns(s_1) && \text{by Lemma 20} \\
&\subseteq V \setminus assigns(s) && \text{by (A.2) and definition of } assigns
\end{aligned}$$

So we conclude by Lemma 22.

WH-TT-WAIT Vacuous since $\langle s, \sigma_i \rangle \rightarrow^i \langle Ok, \sigma_i' \rangle$.

□

To go from executing vectors of programs to two processes in isolation:

Definition 15. *For a program s, let $sf(s)$ be s with all* `sync`*-commands replaced by* `skip`*:*

$$sf(s) = \begin{cases} \texttt{skip} & \text{if } c = \texttt{sync} \\ sf(s_1); sf(s_2) & \text{if } c = s_1; s_2 \\ \texttt{if } b \texttt{ then } sf(s_1) \texttt{ else } sf(s_2) \texttt{ end} & \text{if } c = \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \\ \texttt{while } b \texttt{ do } sf(s_1) \texttt{ end} & \text{if } c = \texttt{while } b \texttt{ do } s_1 \texttt{ end} \\ c & \text{otherwise} \end{cases}$$

Lemma 23. *If $\langle s, \sigma \rangle \rightarrow^i \langle Ok, \sigma' \rangle$ then $\langle sf(s), \sigma \rangle \rightarrow^i \langle Ok, \sigma' \rangle$.*

Proof: Follows by a trivial rule induction on the hypothesis. □

Lemma 24. *If* $\langle s, \sigma\rangle \to^i \langle Wait(s'), \sigma''\rangle$ *and* $\langle sf(s'), \sigma''\rangle \to^i \langle Ok, \sigma'\rangle$ *then* $\langle sf(s), \sigma\rangle \to^i \langle Ok, \sigma'\rangle$.

Proof: By rule induction on the first hypothesis. We do case distinction on the last rule in the derivation.

SYNC Then $\sigma = \sigma' = \sigma''$, and $sf(\texttt{sync}) = \texttt{skip}$. By the SKIP-rule, $\langle sf(s), \sigma\rangle \to^i \langle Ok, \sigma\rangle$.

IF-TT Then $s \equiv \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ and $sf(s) \equiv \texttt{if}\ b\ \texttt{then}\ sf(s_1)\ \texttt{else}\ sf(s_2)\ \texttt{end}$. By the premises of this rule, $\mathcal{B}[\![b]\!]^j\,\sigma = \texttt{tt}$ and $\langle s_1, \sigma\rangle \to^i \langle Wait(s'), \sigma''\rangle$. By assumption $\langle sf(s'), \sigma''\rangle \to^i \langle Ok, \sigma'\rangle$. Thus, by the induction hypothesis, $\langle sf(s_1), \sigma\rangle \to^i \langle Ok, \sigma'\rangle$. We then obtain $\langle s, \sigma\rangle \to^i \langle Ok, \sigma'\rangle$ by the IF-TT rule.

IF-FF Analogous to last case.

WH-TT-OK Then $s \equiv \texttt{while}\ b\ \texttt{do}\ s_1\ \texttt{end}$ and $sf(s) \equiv \texttt{while}\ b\ \texttt{do}\ sf(s_1)\ \texttt{end}$. By the premises of this rule, $\mathcal{B}[\![b]\!]^j\,\sigma = \texttt{tt}$ and $\langle s_1, \sigma\rangle \to^i \langle Ok, \sigma_0''\rangle$ for some σ_0'' and $\langle s, \sigma_0''\rangle \to^i \langle Wait(s'), \sigma''\rangle$. By assumption $\langle sf(s'), \sigma''\rangle \to^i \langle Ok, \sigma'\rangle$. Thus, by the induction hypothesis, $\langle sf(s), \sigma_0''\rangle \to^i \langle Ok, \sigma'\rangle$. By Lemma 23, we have $\langle sf(s_1), \sigma\rangle \to^i \langle Ok, \sigma_0''\rangle$ and so we obtain $\langle sf(s), \sigma\rangle \to^i \langle Ok, \sigma'\rangle$ by the WH-TT-OK rule.

WH-TT-WAIT Then $s \equiv \texttt{while}\ b\ \texttt{do}\ s_1\ \texttt{end}$ and $sf(s) \equiv \texttt{while}\ b\ \texttt{do}\ sf(s_1)\ \texttt{end}$. By the premises of this rule, $\mathcal{B}[\![b]\!]^j\,\sigma = \texttt{tt}$ and $\langle s_1, \sigma\rangle \to^i \langle Wait(s_1'), \sigma''\rangle$ and so $\langle s, \sigma\rangle \to^i \langle Wait(s_1'; c), \sigma''\rangle$. By assumption $\langle sf(s_1'; c), \sigma''\rangle \to^i \langle Ok, \sigma'\rangle$. So there must be some σ_1'' so that $\langle sf(s_1'), \sigma''\rangle \to^i \langle Ok, \sigma_1''\rangle$ and $\langle sf(s), \sigma_1''\rangle \to^i \langle Ok, \sigma'\rangle$.

By the induction hypothesis, $\langle sf(s_1), \sigma\rangle \to^i \langle Ok, \sigma_1''\rangle$, and so we obtain $\langle sf(s), \sigma\rangle \to^i \langle Ok, \sigma'\rangle$ by the WH-TT-OK rule.

SEQ-OK $s \equiv s_1; s_2$ and $sf(s) \equiv sf(s_1); sf(s_2)$. By the premises of this rule,

$$\langle s_1, \sigma\rangle \to^i \langle Ok, \sigma_0''\rangle \qquad \langle s_2, \sigma_0''\rangle \to^i \langle Wait(s_2'), \sigma''\rangle$$

for some σ_0'', and by assumption

$$\langle sf(s_2'), \sigma''\rangle \to^i \langle Ok, \sigma'\rangle$$

By Lemma 23 we have

$$\langle sf(s_1), \sigma\rangle \to^i \langle Ok, \sigma_0''\rangle$$

and by the induction hypothesis

$$\langle sf(s_2), \sigma_0'' \rangle \to^i \langle Ok, \sigma' \rangle$$

We then apply the SEQ-OK rule, and obtain $\langle sf(s), \sigma \rangle \to^i \langle Ok, \sigma' \rangle$.

SEQ-WAIT $s \equiv s_1; s_2$ and $sf(s) \equiv sf(s_1); sf(s_2)$. We then have:

$$\frac{\langle s_1, \sigma \rangle \to^i \langle Wait(s_1'), \sigma'' \rangle}{\langle s_1; s_2, \sigma \rangle \to^i \langle Wait(s_1'; s_2), \sigma'' \rangle}$$

and by assumption

$$\langle sf(s_1'; s_2), \sigma'' \rangle \to^i \langle Ok, \sigma' \rangle$$

Then there is σ_0'' so that

$$\langle sf(s_1'), \sigma'' \rangle \to^i \langle Ok, \sigma_0'' \rangle \qquad \langle sf(s_2), \sigma_0'' \rangle \to^i \langle Ok, \sigma' \rangle$$

By the induction hypothesis, we obtain

$$\langle sf(s_1), \sigma \rangle \to^i \langle Ok, \sigma_0'' \rangle$$

and so by SEQ-OK we have $\langle sf(s), \sigma \rangle \to^i \langle Ok, \sigma' \rangle$.

SKIP , ASSIGN and WH-FF. Vacuous since the termination state is not Ok.

□

Lemma 25. *If* $\langle S, \theta \rangle \longrightarrow \theta'$ *then for all* $i \in$ **Pid**, $\langle sf(S[i]), \theta[i] \rangle \to^i \langle Ok, \theta'[i] \rangle$.
Proof: By rule induction on the hypothesis. If the last rule used was:

ALL-OK Take any $i \in$ **Pid**. By the premises of this rule, $\langle S[i], \theta[i] \rangle \to^i \langle Ok, \theta'[i] \rangle$. By Lemma 23, $\langle sf(S[i]), \theta[i] \rangle \to^i \langle Ok, \theta'[i] \rangle$

ALL-WAIT Again, take any $i \in$ **Pid**. By the premises of this rule,

$$\langle S[i], \theta[i] \rangle \to^i \langle Wait(S'[i]), \theta''[i] \rangle \quad \text{and} \quad \langle S', \theta'' \rangle \longrightarrow \theta'$$

By the induction hypothesis, $\langle sf(S'[i]), \theta''[i] \rangle \to^i \langle Ok, \theta'[i] \rangle$. Then by Lemma 24, $\langle sf(S[i]), \theta[i] \rangle \to^i \langle Ok, \theta'[i] \rangle$.

□

Knowing that we can look at two processes in isolation, and that the analysis is correct for two processes, we can extend the result to vectors and show the main theorem:

Theorem 2. *Let s be a program, $\theta, \theta' \in D_{\mathbf{Pid}}$ two environment vectors and $(pi_\circ, pi_\bullet) \models PI(s)$. Let $V = \pi^1(pi_\circ(init(s)))$ and $V' = \pi^1(pi_\bullet(final_e(s)))$. If $\sim_V \theta$ and $[\![s]\!]\,\theta = \theta'$, then $\sim_{V'} \theta'$.*

Proof: By Theorem 1, $\langle \langle s \rangle_i, \theta \rangle \longrightarrow \theta'$.

Take any $i, j \in \mathbf{Pid}$. Then by Lemma 25, we have $\langle sf(s), \theta[i] \rangle \rightarrow^i \langle Ok, \theta'[i] \rangle$ and $\langle sf(s), \theta\ j \rangle \rightarrow^j \langle Ok, \theta'[j] \rangle$.

Note that $(pi_\circ, pi_\bullet) \vDash PI(s) \iff (pi_\circ, pi_\bullet) \vDash PI^\iota(s)$ with $\iota = (\mathbf{Var}, \epsilon)$. By Theorem 6, $\theta'[i] \sim_{V'} \theta'[j]$. Since this is true for any i and j, we conclude $\sim_{V'} \theta'$ □

A.3 Correctness of *RS*

A.3.1 Safe State Transformers

Next, we show that programs that are syntactically synchronization free (see Section 4.3.2) are also textually aligned.

Lemma 26. *Let $s \in \mathbf{Par}$ such that $sf^\sharp(s)$, then s is textually aligned for $D \setminus \{\Omega_S\}$.*

Proof: The proof proceeds by structural induction on s. The only clause which gives rise to Ω_S in the definition of $[\![s]\!]$ is that for `sync`. However this clause cannot be applied since s is statically synchronization free, and thus does not contain the `sync`-command. In all other clauses, the lemma follows from definition and by applying the induction hypothesis to each commands constituents. □

Lemma 27 (Single-valued case distinction)**.** *Let $V \subseteq \mathbf{Var}$, $\theta \in D \setminus \{\bot, \Omega_S\}$ and $b \in \mathbf{BExp}$. If $\sim_V \theta$ and $\phi^d(b, V)$, then*

(i) $\partial_b\,\theta = \theta$ *and* $\partial_{!b}\,\theta \in D_\emptyset$ *or*

(ii) $\partial_b\,\theta \in D_\emptyset$ *and* $\partial_{!b}\,\theta = \theta$.

Proof: By Lemma 13 it follows that for all i and j such that $\theta[i] \neq \mathbf{0}$ and $\theta[j] \neq \mathbf{0}$, either $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathcal{B}[\![b]\!]^j\,\theta[j] = \mathtt{tt}$ or $\mathcal{B}[\![b]\!]^i\,\theta[i] = \mathcal{B}[\![b]\!]^j\,\theta[j] = \mathtt{ff}$. Then by the definition of ∂, if it is the former, case *(i)* applies, and if it is the latter, case *(ii)* applies, □

Theorem 3. *If* $(pi_\circ, pi_\bullet) \vDash PI(s)$, $RS(s, pi)$, *and* $D^V_{\mathbf{Pid}} = \{\theta \in D_{\mathbf{Pid}} \mid \sim_V \theta\}$ *where* $V = \pi^1(pi_\circ(init(s)))$, *then* s *is textually aligned for any environment in* $D^V_{\mathbf{Pid}}$.

Proof:

We prove the stronger property $P(s)$ by structural induction on s:

$$P(s) \iff \forall \iota \in L, \forall pi = (pi_\circ, pi_\bullet), pi \vDash PI^\iota(s) \wedge RS(s, pi) \wedge \forall \theta \in D^{\pi^1(pi_\circ(init(s)))}_{\mathbf{Pid}} \implies [\![s]\!]\,\theta \neq \Omega_S$$

We assume some ι and pi such that $pi \vDash PI^\iota(s)$, $RS(s, pi)$, let $V = \pi^1(pi_\circ(init(s)))$, $\theta \in D^V_{\mathbf{Pid}}$, and show that $[\![s]\!]\,\theta \neq \Omega_S$.

- $s \equiv \texttt{skip}$. Then $[\![s]\!]\,\theta = \theta \neq \Omega_S$.
- $s \equiv \texttt{sync}$. Since $\theta \in D_{\mathbf{Pid}}$, $[\![s]\!]\,\theta = \theta$ and $\theta \neq \Omega_S$.
- $s \equiv X := e$. Then $[\![s]\!] = [X \leftarrow e]$, and since $\theta \in D_{\mathbf{Pid}}$ we have by the definition of $[X \leftarrow e]$, $[X \leftarrow e]\,\theta \neq \Omega_S$.
- $s \equiv s_1; s_2$. In this case, $[\![s]\!] = [\![s_2]\!]\,([\![s_1]\!]\,\theta)$. We apply the induction hypothesis twice: (i) first to show that $([\![s_1]\!]\,\theta) \neq \Omega_S$, and (ii) to show that $[\![s_2]\!]\,([\![s_1]\!]\,\theta) \neq \Omega_S$.

  (i) By Lemma 14, $pi \vDash PI^{\iota'}(s_1)$, with $\iota' = pi_\circ(init(s_1))$. From the premises of $RS(s, pi)$ we have $RS(s_1, pi)$. Since $init(s_1; s_2) = init(s_1)$, $V = \pi^1(pi_\circ(init(s_1)))$ as well. Let $[\![s_1]\!]\,\theta = \theta'$. Then from the induction hypothesis we have that $\theta' \neq \Omega_S$.

  (ii) If $\theta' = \bot$ then $[\![s]\!]\,\theta = [\![s_2]\!]\,\theta' = \bot \neq \Omega_S$ and we are done, so assume $\theta' \neq \bot$. Then by Theorem 2

  $$\sim_{V'} \theta' \text{ where } V' = \pi^1(pi_\bullet(final_e(s_1))) \supseteq \pi^1(pi_\circ(init(s_2)))$$

  where the last inclusion is follows from the construction of the constraint system.

  By Lemma 14, $pi \vDash PI^{\iota''}(s_2)$, with $\iota'' = pi_\circ(init(s_2))$. From the premises of $RS(s, pi)$ we have $RS(s_2, pi)$. Let $V'' = \pi^1(pi_\circ(init(s_2)))$. By the induction hypothesis we get that s_2 is textually aligned for $D^{V''}_{\mathbf{Pid}}$.

  From the above inclusion, we have $V'' \subseteq V'$, i.e. $\sim_{V''} \theta'$, and since the semantic function of statements is stable by Lemma 4, i.e. $\theta' \in D_{\mathbf{Pid}}$, as a consequence $\theta' \in D^{V''}_{\mathbf{Pid}}$, and so $[\![s]\!]\,\theta = [\![s_2]\!]\,\theta' \neq \Omega_S$.

- $s \equiv \texttt{if}\, b\, \texttt{then}\, s_1\, \texttt{else}\, s_2\, \texttt{end}$. By the premises of $RS(s, pi)$, either:
  - $\phi^d(b, V)$. Since $\sim_V \theta$, by Lemma 27 we have $\partial_b\,\theta = \theta$ and $\partial_{!b}\,\theta \in D_\emptyset$ (or analogously, $\partial_{!b}\,\theta = \theta$ and $\partial_b\,\theta \in D_\emptyset$). Then

    $$[\![s]\!]\,\theta = [\![s_1]\!]\,(\partial_b\,\theta) \parallel [\![s_2]\!]\,(\partial_{!b}\,\theta) = [\![s_1]\!]\,\theta$$

    follows by Lemma 8.

    We have $pi \vDash PI^\iota(s_1)$, with $\iota' = pi_\circ(init(s_1))$ by Lemma 14, and $RS(s_1, pi)$ by the premises. Let $V_1 = \pi^1(pi_\circ(init(s_1)))$. Then s_1 is textually aligned for $D^{V_1}_{\mathbf{Pid}}$ by the induction hypothesis. By construction of the constraint system, we have

    $$V_1 = \pi^1(pi_\circ(init(s_1))) \subseteq \pi^1(pi_\bullet(\mathtt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) = V$$

    and since $\sim_V \theta$, $\sim_{V_1} \theta$. Then $\theta \in D^{V_1}_{\mathbf{Pid}}$, and since s_1 is textually aligned for $D^{V_1}_{\mathbf{Pid}}$, $[\![s_1]\!]\,\theta \neq \Omega_S$.
  - $sf^\sharp(s_1)$ and $sf^\sharp(s_2)$. Then $sf^\sharp(s)$, and we conclude by Lemma 26.
- $s \equiv \texttt{while}\, b\, \texttt{do}\, s_1\, \texttt{end}$. Then $[\![s]\!] = fix\, F = \bigsqcup\{F^n\, f_\perp \mid n \geq 0\}$, and

  $$F = \lambda f.\lambda\theta. \begin{cases} ((f \circ [\![s_1]\!])\,(\partial_b\,\theta)) \parallel (\partial_{!b}\,\theta) & \text{if } \partial_b\,\theta \notin D_\emptyset \cup \{\perp, \Omega_S\} \\ \theta & \text{otherwise} \end{cases}$$

  By the premises of $RS(s, pi)$, either $sf^\sharp(s_1)$. Then $sf^\sharp(s)$ and the conclusion follows by Lemma 26.

  Otherwise $\phi^d(b, V)$. To show $[\![s]\!]\,\theta \neq \Omega_S$ we show that all $F^n\, f_\perp$ are textually aligned for $D^V_{\mathbf{Pid}}$, by induction on n. Let this property $P'(n)$:

  $$P'(n) \Leftrightarrow \forall \theta \in D^V_{\mathbf{Pid}}, (F^n\, f_\perp)\,\theta \neq \Omega_S$$

  $P'(0)$ Let $\theta \in D^V_{\mathbf{Pid}}$. Then $(F^0\, f_\perp)\,\theta = f_\perp\,\theta = \perp \neq \Omega_S$.

  $P'(n+1)$ Let $\theta \in D^V_{\mathbf{Pid}}$. Since $\phi^d(b, V)$, by Lemma 27, either $\partial_b\,\theta \in D_\emptyset$ and $\partial_{!b}\,\theta = \theta$. Then $F^{i+1}\, f_\perp = \lambda\theta.\theta$ which is trivially textually aligned for $D^V_{\mathbf{Pid}}$.

  Otherwise, $\partial_b\,\theta = \theta \notin D_\emptyset$ and $\partial_{!b}\,\theta \in D_\emptyset$. In this case,

  $$(F^{n+1}\, f_\perp)\,\theta = F\,(F^n\, f_\perp)\,\theta = ((F^n\, f_\perp) \circ [\![s_1]\!])\,\theta$$

The outer induction hypothesis gives $P(s_1)$ and, since $pi \vDash PI^{\iota'}(s_1)$ with $\iota' = pi_\circ(init(s_1))$ by Lemma 14, and $RS(s_1, pi)$ by the premises of $RS(s, pi)$. Let $V_1 = \pi^1(pi_\circ(init(s_1)))$, and then we obtain that s_1 is textually aligned for $D^{V_1}_{\mathbf{Pid}}$.

The constraint system gives

$$V_1 = \pi^1(pi_\circ(init(s_1))) \subseteq \pi^1(pi_\bullet(\mathtt{t}(\ell))) \subseteq \pi^1(pi_\circ(\ell)) = V$$

so since $\sim_V \theta$ we have $\sim_{V_1} \theta$.

Let $[\![s_1]\!]\,\theta = \theta'$. Since $\sim_{V_1} \theta$ and s_1 is textually aligned for $D^{V_1}_{\mathbf{Pid}}$, $\theta' \neq \Omega_S$. Let $V_1' = \pi^1(final_e(s_1))$. By Theorem 2, $\sim_{V_1'} \theta'$.

By construction of $PI^\iota(s)$, we have the following:

$$V = \pi^1(pi_\circ(\ell)) \subseteq \pi^1(pi_\bullet(\mathtt{n}(\ell))) \cap \pi^1(pi_\bullet(\mathtt{b}(\ell))) \subseteq \pi^1(pi_\bullet(\mathtt{b}(\ell))) = V_1'$$

And so by $\sim_{V_1'} \theta'$ we have $\sim_V \theta'$, thus $\theta' \in D^V_{\mathbf{Pid}}$.

By the induction hypothesis on n we have that $F^n\, f_\perp$ is textually aligned for $D^V_{\mathbf{Pid}}$, and so

$$[\![s]\!]\,\theta = (F^n\, f_\perp)([\![s_1]\!]\,\theta) = (F^n\, f_\perp)\,\theta' \neq \Omega_S$$

□

Proof Sketches for Safe Registration in BSPlib

B

Contents

B.1 Proof Sketch For Lemma 1 221
B.2 Proof Sketch For Theorem 4 222
B.3 Proof Sketch For Theorem 5 224

This section sketches the proofs of Lemma 1 and Theorems 4 and 5. While the full proofs have been developed, we do not include them here due to their length and lack of time. Instead, we defer their typesetting in a technical report to future work.

B.1 Proof Sketch For Lemma 1

We recall Lemma 1:

Lemma 1. *If* $Reach(\Gamma, \Gamma'); A$ *then the same source* $s \neq$ *unknown never appears twice in the same component of* A *associated with two locations of different base.*

Proof sketch: Before proving this lemma, we show an auxiliary fact. Namely, that the path of each local execution step is a "fresh" path that has not appeared before. We formalize freshness by establishing a strict order on paths with the intention that, $\delta_1 < \delta_2$ if δ_1 is produced "before" δ_2 in the semantics and thus that δ_2 is fresh. We then show, by a standard rule induction on the local semantics, that the initial and final path of each local step is strictly ordered.

The proof of Lemma 1 consists of showing that for each instrumented state $\mathcal{H}, o$ and as in a local execution there exists a partial mapping ρ from sources to bases such that for each location-source pair (l, s) where $s \neq$ *unknown* and either

1. (l, s) appears in as, or

2. for some location l', $\mathcal{H}\, l' = l$ and $o\, l = s$, or
3. for some variable x, $s = x$ and $l = \sigma\, x$.

we have $\rho\, s = \pi_{base}(l)$. We then say that the instrumented state is ρ-consistent.

Clearly from (1) follows that each source that appears in as is associated with at most one base, by the single-valuedness of ρ, as Lemma 1 requires.

We first show that the initial state of a local process is ρ-consistent with ρ being the empty map. We then show by rule induction on the local semantics that an appropriate ρ' exists that preserves ρ-consistent of the instrumented state.

- For all cases but IMALLOC we take $\rho' = \rho$. In the case MALLOC we take $\rho' = \rho[(\delta, \ell) \leftarrow \pi_{base}(l')]$ where l' is the newly allocated location and δ the current path.

  Since the path is fresh, (δ, ℓ) cannot already be defined in ρ. ρ'-consistency of the new instrumented state follows by considering any (l, s) pair that occurs in the new instrumented state.

- If a new action is added to the trace (rules IPUSH, IPOP and ISYNC) and it contains the location-source pair (l, s) then $\rho\, s = \pi_{base}(l)$ follows by the consistency of the instrumented state.

- For the case IASSIGN, we prove an additional fact: in a ρ-consistent instrumented state, the source given by the src-function for a location applied to ρ returns the base of that same location. Using this fact, we show the instrumented state resulting from the assignment ρ-consistent.

- Remaining cases are trivial.

To conclude, we show that ρ-consistency follows for multi-step execution by standard rule induction. Then, that the concept can be extended to global executions by showing the existence of p-vectors of mappings $\langle \rho_i \rangle_i$ such that the state of each processor i is ρ_i-consistent. We show that all reachable global configurations are $\langle \rho_i \rangle_i$-consistent. The result follows. □

B.2 Proof Sketch For Theorem 4

We recall Theorem 4:

Theorem 4. *If* $Reach(\Gamma_c, (\langle \gamma_i; I_i \rangle_i, rs)); A$ *and* $\mathcal{GC}![\![A]\!] = \mathtt{tt}$ *then* $rs \neq \Omega_R$.

Proof sketch: The proof strategy consists of establishing a correspondence between a **p**-vector of the maps in **MapI** that $\mathcal{GC}!$ acts on and the registration sequences of **RegSeq** in the concrete semantics. In particular, the correspondence consists of a function from a **p**-vector of **MapI** to an element of **RegSeq**. Thus by definition, the correspondence only holds when the registration sequence is not in the error state.

Trivially, the initial empty registration sequence and an vector of empty maps correspond to each other.

Before considering reachable configurations, we establish an auxiliary fact relating $\mathcal{GC}!_{ss}$ and $\mathcal{R}$. First, the former is defined for action traces and the latter over vectors of lists of registration requests. We define a function *reqsToTrace* from registration requests to action traces in the obvious way.

We then show that if rs and a vector of maps $\langle r_i \rangle_i$ correspond, if $\langle rrs_i \rangle_i$ is a vector of registration requests such as resulting from the execution of one superstep, then the vector of maps that results from applying $\mathcal{GC}!_{ss}\,(reqsToTrace(rrs_i))$ pointwise to $\langle r_i \rangle_i$ conserves the correspondence with $\mathcal{R}\, rs\, \langle rrs_i \rangle_i$. This follows from a showing similar relationship between $\mathcal{R}_\ominus, \mathcal{R}_\oplus$ and $\mathcal{GC}!_1$ followed by standard induction.

We then proceed by rule induction to show that all reachable configurations preserve the correspondence between registration sequence and the map-vector obtained by applying $\mathcal{GC}!'$ pointwise to the reached the action vector. The reflection case of *Reach* is trivial. In the step case of *Reach* we consider the termination type α of the global step. In the case $\alpha = \kappa$, then the correspondence trivially holds. When $\alpha = \iota$ we note that the action trace of each process i is on the form $as_i = as_i' +\!\!+ [\mathtt{sync!\_}]$ such that as_i' does not contain any synchronization actions, and that $as_i' = reqsToTrace(rrs_i)$ where rrs_i correspond to the list of registration requests engendered by process i and *reqsToTrace*, three facts which follows from a standard rule induction on the local semantics. Preservation of the correspondence now follows since it is preserved by $\mathcal{GC}!_{ss}$ and $\mathcal{R}$.

As the correspondence is preserved by all reachable configurations with an action vector for which $\mathcal{GC}!$ holds, and since the correspondence by definition only holds if the registration sequence of that configuration is not in the error state, we have the desired result. □

B.3 Proof Sketch For Theorem 5

We recall Theorem 5:

Theorem 5. *If A is safe then $\mathcal{GC}!\llbracket A \rrbracket = \mathtt{tt}$.*

Proof sketch: We first show that a trace that is locally correct also has a matching, which is simple given the similarity of the functions $\mathcal{LC}!$ and $\mathcal{GC}!$.

It then suffices to show that that all pairs of action traces that are ρ-consistent, textually aligned, source aligned and where both traces have a matching actually have the same matching, implying that all traces in the vector have the same matching and thus that the vector is globally correct.

This is done by showing that two such traces are equivalent modulo bases, and thus exactly equal if ρ of one trace is used to substitute each source-(base-offset) pair $(s, (b, o))$ appearing in the other with $(s, (\rho\, s, o))$. This follows from the definition of

1. textual alignment: from which we know that the traces have the same length and their action have pointwise the same path. We can then show that two actions with the same path from the same program must be the same type of action (since they originate from the same instruction).

2. source alignment: which ensures that the two actions at the same position in the traces have the same source and the same offset.

Hence, only bases differ between the traces.

From there we show that applied to $\mathcal{GC}!$, two such traces have the same matching. Specifically, we show that a correspondence can be established between two elements of **MapI** that are in this way equivalent modulo bases. Trivially, two empty **MapI** are equivalent modulo base. We then show that $\mathcal{GC}!_1$, applied to actions and maps equivalent modulo bases return the same matching and new maps that are also equivalent modulo bases. The result then follows from by induction on the length of the trace. □

Bibliography

[1] S. Aananthakrishnan, G. Bronevetsky, M. Baranowski, and G. Gopalakrishnan. ParFuse: Parallel and Compositional Analysis of Message Passing Programs. In C. Ding, J. Criswell, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 24–39. Springer International Publishing, 2017. ISBN 978-3-319-52709-3. 77

[2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229471. 74, 75, 76

[3] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable Temporal Order Analysis for Large Scale Debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654104. 73

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986. 70

[5] A. Aiken and D. Gay. Barrier Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268974. 70, 73, 79, 82, 92, 116, 154

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *European Symposium on Programming*, pages 157–172. Springer, 2007. 120, 121, 125, 126, 147

[7] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011. 147

[8] E. Albert, J. Correas, and G. Román-Díez. Peak cost analysis of distributed systems. In *International Static Analysis Symposium*, pages 18–33. Springer, 2014. 78

[9] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. Resource Analysis: From Sequential to Concurrent and Distributed Programs. In *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 3–17. Springer International Publishing, June 2015. doi: 10.1007/978-3-319-19249-9_1. 78, 152

[10] V. Allombert. *Functional Abstraction for Programming Multi-Level Architectures : Formalisation and Implementation*. PhD thesis, Université Paris-Est, July 2017. 56, 79

[11] V. Allombert, F. Gava, and J. Tesson. Multi-ML: Programming Multi-BSP Algorithms in ML. *International Journal of Parallel Programming*, page 20, 2015. 59, 61

[12] V. Allombert, F. Gava, and J. Tesson. Toward Performance Prediction for Multi-BSP Programs in ML. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 159–174. Springer, 2018. 79

[13] R. Alur, J. Devietti, O. S. Navarro Leija, and N. Singhania. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 507–525. Springer International Publishing, 2017. ISBN 978-3-319-63387-9. 71, 73, 75

[14] J. Anderson, P. J. Burns, D. Milroy, P. Ruprecht, T. Hauser, and H. J. Siegel. Deploying RMACC Summit: An HPC Resource for the Rocky Mountain Region. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 8:1–8:7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5272-7. doi: 10.1145/3093338.3093379. v, 2

[15] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, Feb. 1975. ISSN 0022-0000. doi: 10.1016/S0022-0000(75)80018-3. 65

[16] M. Assaf. *From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow*. PhD thesis, Université Rennes 1, 2015. 70

[17] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *ACM SIGPLAN Notices*, volume 31, pages 149–159. ACM, 1996. 70, 73, 106

[18] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In P. Trinder and G. Michaelson, editors, *Proceedings of the First Scottish Functional Programming Workshop*, Technical Report, pages 43–52, Edinburgh, Aug. 1999. Heriot-Watt University. viii, 42, 61

[19] E. Bardsley and A. F. Donaldson. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *NASA Formal Methods*, pages 230–245. Springer, Cham, Apr. 2014. doi: 10.1007/978-3-319-06200-6_18. 66

[20] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable than You Think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010. 121, 140, 142

[21] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013. 65, 83

[22] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A verifier for GPU kernels. In *ACM SIGPLAN Notices*, volume 47, pages 113–132. ACM, 2012. 66, 73

[23] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press on Demand, 2004. 14, 30, 114

[24] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 67–72. IEEE, 2016. 153

[25] S. Blazy, D. Bühler, and B. Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In A. Bouajjani and D. Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture

Notes in Computer Science, pages 112–130. Springer International Publishing, 2017. ISBN 978-3-319-52234-0. 53

[26] O. Bonorden, B. Juurlink, I. Von Otte, and I. Rieping. The Paderborn university BSP (PUB) library-design, implementation and performance. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 99–104, Apr. 1999. doi: 10.1109/IPPS.1999.760442. 75

[27] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, Feb. 2003. ISSN 0167-8191. doi: 10.1016/S0167-8191(02)00218-1. 39, 59, 63, 184

[28] V. Botbol, E. Chailloux, and T. Le Gall. Static Analysis of Communicating Processes Using Symbolic Transducers. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 73–90. Springer, 2017. 71

[29] L. Bougé. The Data-Parallel Programming Model: A Semantic Perspective (Final Version). Report RR-3044, INRIA, 1996. 67

[30] P. Boulet and X. Redon. Communication Pre-evaluation in HPF. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 263–272, London, UK, UK, 1998. Springer-Verlag. ISBN 978-3-540-64952-6. 78, 153, 154

[31] W. Bousdira, F. Loulergue, and J. Tesson. A verified library of algorithmic skeletons on evenly distributed arrays. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 218–232. Springer, 2012. 67

[32] G. Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.32. 71, 72, 73, 74

[33] J.-W. Buurlage, T. Bannink, and A. Wits. Bulk-synchronous pseudo-streaming algorithms for many-core accelerators. *arXiv:1608.07200 [cs]*, Aug. 2016. 39

[34] J.-W. Buurlage, T. Bannink, and R. H. Bisseling. Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs. In *Euro-Par 2018: Parallel Processing*, pages 519–532. Springer, Cham, Aug. 2018. doi: 10.1007/978-3-319-96983-1_37. 39

[35] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011. 74

[36] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2, New York, NY, USA, 2010. ACM. 41, 184

[37] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection. In *Languages and Compilers for Parallel Computing*, pages 106–120. Springer, Cham, 2017. doi: 10.1007/978-3-319-52709-3_10. 74, 76, 153

[38] K. C. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ proof system. *arXiv preprint arXiv:0811.1914*, 2008. 66

[39] Y. Chen and J. W. Sanders. Top-down design of bulk-synchronous parallel programs. *Parallel Processing Letters*, 13(03):389–400, 2003. 67

[40] Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(2):221–262, 2004. 67

[41] T. Christiansen, L. Wall, and J. Orwant. *Programming Perl: Unmatched Power for Text Processing and Scripting*. O'Reilly Media, Inc., 2012. x, 6

[42] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli: A comprehensive overview. Technical report, Working Papers, ERCIS-European Research Center for Information Systems, 2009. 63

[43] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999. 68

[44] P. Clauss. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pages 278–285, New York, NY, USA, 1996. ACM. ISBN 978-0-89791-803-9. doi: 10.1145/237578.237617. 78, 153

[45] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009. 65, 66

[46] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman London, 1989. 62

[47] B. Cook. Principles of program termination. *Engineering Methods and Tools for Software Safety and Security*, 22:161, 2009. 125

[48] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. 43, 53

[49] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96. ACM, 1978. 147

[50] R. Couturier and D. Méry. An experiment in parallelizing an application using formal methods. In *Computer Aided Verification*, pages 345–356. Springer, Berlin, Heidelberg, June 1998. doi: 10.1007/BFb0028757. 66

[51] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993. 56, 57

[52] P. Cuoq, P. Hilsenkopf, F. Kirchner, S. Labbé, N. Thuy, and B. Yakobowski. Formal verification of software important to safety using the Frama-C tool suite. In *Proceedings of the 8th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies*, 2012. 53

[53] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012. 65, 66

[54] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, A Program Analysis Perspective. In *The 10th*

International Conference on Software Engineering and Formal Methods (SEFM 2012), volume 7504 of *LNCS*, pages 233–247. Springer, 2012. 52

[55] F. Dabrowski. A Denotational Semantics of Textually Aligned SPMD Programs. In *International Symposium on Formal Approaches to Parallel and Distributed Systems (4PAD 2018)*, 2018. 73, 83, 87, 116

[56] F. Dabrowski. Textual Alignment in SPMD Programs. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1046–1053, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5191-1. doi: 10.1145/3167132.3167254. 73, 79, 84, 116, 170

[57] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 133–144, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328457. 122

[58] S. Darabi, S. C. C. Blom, and M. Huisman. A Verification Technique for Deterministic Parallel Programs. In *NASA Formal Methods*, pages 247–264. Springer, Cham, May 2017. doi: 10.1007/978-3-319-57288-8_17. 66

[59] F. Darema. SPMD computational model. In *Encyclopedia of Parallel Computing*, pages 1933–1943. Springer, 2011. 24

[60] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 65

[61] M. Delahaye, N. Kosmatov, and J. Signoles. Common Specification Language for Static and Dynamic Analysis of C Programs. In *The 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, pages 1230–1235. ACM, 2013. 53

[62] B. Di Martino, A. Mazzeo, N. Mazzocca, and U. Villano. Parallel Program Analysis and Restructuring by Detection of Point-to-Point Interaction Patterns and Their Transformation into Collective Communication Constructs. *Science of Computer Programming*, 40(2–3):235–263, July 2001. ISSN 0167-6423. doi: 10.1016/S0167-6423(01)00017-X. 74, 77, 135, 141

[63] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975. 53

[64] Y. Dubois and R. Teyssier. On the onset of galactic winds in quiescent star forming galaxies. *Astronomy and Astrophysics*, 477:79–94, Jan. 2008. ISSN 0004-6361. doi: 10.1051/0004-6361:20078326. xiv, 185

[65] J. Eloff and M. B. Bella. Software Failures: An Overview. In J. Eloff and M. Bihina Bella, editors, *Software Failure Investigation: A Near-Miss Analysis Approach*, pages 7–24. Springer International Publishing, Cham, 2018. ISBN 978-3-319-61334-5. doi: 10.1007/978-3-319-61334-5_2. xiv, 185

[66] K. Emoto, F. Loulergue, and J. Tesson. A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, number 8558 in Lecture Notes in Computer Science, pages 258–274. Springer International Publishing, July 2014. ISBN 978-3-319-08969-0 978-3-319-08970-6. doi: 10.1007/978-3-319-08970-6_17. 67

[67] A. Ernstsson, L. Li, and C. Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, 2018. 63

[68] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *Programming Languages and Systems*, pages 125–128. Springer, 2013. 65, 67

[69] C. Flanagan and S. Qadeer. Thread-modular model checking. In *International SPIN Workshop on Model Checking of Software*, pages 213–224. Springer, 2003. 71

[70] L. Flon and N. Suzuki. Consistent and complete proof rules for the total correctness of parallel programs. In *Foundations of Computer Science, 1978., 19th Annual Symposium On*, pages 184–192. IEEE, 1978. 65

[71] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972. 25

[72] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(4):15, 2017. 69

[73] J. Fortin and F. Gava. Towards Mechanised Semantics of HPC: The BSP with Subgroup Synchronisation Case. In G. Wang, A. Zomaya, G. Martinez, and K. Li, editors, *Algorithms and Architectures for Parallel Processing*,

Lecture Notes in Computer Science, pages 222–237. Springer International Publishing, 2015. ISBN 978-3-319-27161-3. 117

[74] J. Fortin and F. Gava. BSP-Why: A Tool for Deductive Verification of BSP Algorithms with Subgroup Synchronisation. *International Journal of Parallel Programming*, 44(3):574–597, June 2016. ISSN 0885-7458, 1573-7640. doi: 10.1007/s10766-015-0360-y. 66, 67, 68, 153, 183

[75] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978. 56

[76] M. I. Frank, A. Agarwal, and M. K. Vernon. Lopc: Modeling contention in parallel algorithms. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '97, pages 276–287, New York, NY, USA, 1997. ACM. ISBN 0-89791-906-8. doi: 10.1145/263764.263803. URL `http://doi.acm.org/10.1145/263764.263803`. 57

[77] M. Frieb, A. Stegmeier, J. Mische, and T. Ungerer. Employing MPI Collectives for Timing Analysis on Embedded Multi-Cores. In M. Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-025-5. doi: 10.4230/OASIcs.WCET.2016.10. 78

[78] Z. Ganjei, A. Rezine, L. Henrio, P. Eles, and Z. Peng. On Reachability in Parameterized Phaser Programs. *arXiv:1811.07142 [cs]*, Nov. 2018. 74

[79] F. Gava. Formal proofs of functional bsp programs. *Parallel Processing Letters*, 13(03):365–376, Sept. 2003. ISSN 0129-6264. doi: 10.1142/S0129626403001343. 67

[80] F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. In *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008*, pages 269–276, Piscataway, NJ, USA, Dec. 2008. IEEE Press. doi: 10.1109/PDCAT.2008.43. 67, 68, 117, 184

[81] F. Gava and F. Loulergue. A static analysis for Bulk Synchronous Parallel ML to avoid parallel nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005. 78

[82] A. V. Gerbessiotis and S.-Y. Lee. Remote memory access: A case for portable, efficient and library independent parallel programming. *Scientific Programming*, 12(3):169–183, 2004. 43

[83] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. https://www.hindawi.com/journals/sp/2014/571902/abs/, 2014. 24

[84] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic development of correct bulk synchronous parallel programs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference On*, pages 334–340. IEEE, 2010. 67

[85] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010. 62

[86] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama. Report of the HPC Correctness Summit, Jan 25–26, 2017, Washington, DC. *arXiv preprint arXiv:1705.07478*, 2017. 79

[87] S. Gorlatch and M. Cole. Parallel Skeletons. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_24. 62

[88] M. W. Goudreau, K. Lang, S. B. Rao, and T. Tsantilas. The green BSP library. *Report CS TR*, 95(11), 1995. 24, 63

[89] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2014. 2, 42, 188

[90] T. Grosser, A. Groesslinger, and C. Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012. 154

[91] I. Grudenic and N. Bogunovic. Modeling and verification of MPI based distributed software. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 123–132. Springer, 2006. 69

[92] A. Gustavsson, J. Gustafsson, and B. Lisper. Timing Analysis of Parallel Software Using Abstract Execution. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes

in Computer Science, pages 59–77. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54013-4. 77

[93] G. Hains. Subset synchronization in BSP computing. In *PDPTA*, volume 98, pages 242–246, 1998. 15

[94] G. Hains. *Algorithmes et programmation parallèles : Théorie avec BSP et pratique avec OCaml*. Ellipses Marketing, May 2018. ISBN 978-2-340-02466-3. 14

[95] G. Hains and A. Domínguez. Real-time parallel routing for telecom networks: Graph algorithms and bulk-synchronous parallel acceleration. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–4. IEEE, 2016. 114

[96] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In *Second International Workshop on Constructive Methods for Parallel Programming, CMPP*, Ponte de Lima, Portugal, 2000. Uni. Passau (D) TR MIP-0007. 64

[97] V. Halyo, P. LeGresley, and P. Lujan. Massively parallel computing and the search for jets and black holes at the LHC. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 744:54–60, Apr. 2014. ISSN 0168-9002. doi: 10.1016/j.nima.2014.01.038. v, xiv, 1, 185

[98] Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Parallel Programs. *PARALLEL ALGORITHMS AND APPLICATION*, 17(1):59–84, 2002. 79, 153

[99] F. Heine and A. Slowik. Volume Driven Data Distribution for NUMA-Machines. In *Euro-Par 2000 Parallel Processing*, pages 415–424. Springer, Berlin, Heidelberg, Aug. 2000. doi: 10.1007/3-540-44520-X_53. 153

[100] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, Dec. 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00093-3. viii, 23, 30, 41, 63

[101] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 65

[102] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 60

[103] T. Hoefler, J. Dinan, R. Thakur, B. W. Barrett, P. Balaji, W. Gropp, and K. D. Underwood. Remote Memory Access Programming in MPI-3. *TOPC*, 2: 9–9, 2015. doi: 10.1145/2780584. 24, 66, 77

[104] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*, pages 287–306. Springer, 2010. 78

[105] J. Hoffmann and Z. Shao. Automatic Static Cost Analysis for Parallel Programs. In *European Symposium on Programming Languages and Systems*, pages 132–157. Springer, 2015. 78, 152

[106] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, volume 38, pages 185–197. ACM, 2003. 122

[107] W. Hu, N. Huang, and T. Chiueh. Software Defined Radio Implementation of an LTE Downlink Transceiver for Ultra Dense Networks. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. doi: 10.1109/ISCAS.2018.8351109. v, 1

[108] Y. Huang and E. Mercer. Detecting MPI Zero Buffer Incompatibility by SMT Encoding. In *NASA Formal Methods*, pages 219–233. Springer, Cham, Apr. 2015. doi: 10.1007/978-3-319-17524-9_16. 69

[109] J. Hückelheim, Z. Luo, S. H. K. Narayanan, S. Siegel, and P. D. Hovland. Verifying Properties of Differentiable Programs. In *Static Analysis*, pages 205–222. Springer, Cham, Aug. 2018. doi: 10.1007/978-3-319-99725-4_14. 69

[110] A. Jakobsson. Automatic Cost Analysis for Imperative BSP Programs. *International Journal of Parallel Programming*, 47(2):184–212, Apr. 2019. ISSN 0885-7458, 1573-7640. doi: 10.1007/s10766-018-0562-1. 119

[111] A. Jakobsson, F. Dabrowski, W. Bousdira, F. Loulergue, and G. Hains. Replicated Synchronization for Imperative BSP Programs. *Procedia Computer Science*, 108:535–544, Jan. 2017. ISSN 1877-0509. doi: 10.1016/j.procs.2017.05.123. 81

[112] A. Jakobsson, F. Dabrowski, and W. Bousdira. Safe Usage of Registers in BSPlib. In *Proceedings of the 34th Annual ACM Symposium on Applied Computing*, SAC '19, Limassol, Cyprus, Apr. 2019. ACM. ISBN 978-1-4503-5933-7. doi: 10.1145/3297280.3297421. 157

[113] F. A. Jakobsson. Optimized Support of Memory-Related Annotations for Runtime Assertion Checking with Frama-C. Master's thesis, Université de Bordeaux, Aug. 2014. 52

[114] N. Javed and F. Loulergue. OSL: Optimized bulk synchronous parallel skeletons on distributed arrays. In *Advanced Parallel Processing Technologies*, pages 436–451. Springer, 2009. 64

[115] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009. 147

[116] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *IFIP PACT*, pages 171–180. Citeseer, 1994. 72, 74

[117] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in Lecture Notes in Computer Science, pages 359–368. Springer Berlin Heidelberg, Aug. 1996. ISBN 978-3-540-61627-6 978-3-540-70636-6. doi: 10.1007/BFb0024724. 67

[118] A. Jones, R. Melhem, and S. Shao. A compiler-based communication analysis approach for multiprocessor systems. In *Parallel and Distributed Processing Symposium, International(IPDPS)*, page 65, Apr. 2006. ISBN 978-1-4244-0054-6. doi: 10.1109/IPDPS.2006.1639322. 74

[119] G. Jones and M. Goldsmith. *Programming in Occam*. Prentice-Hall International New York, NY, 1987. 60

[120] B. H. H. Juurlink and H. A. G. Wijshoff. A Quantitative Comparison of Parallel Computation Models. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 13–24, New York, NY, USA, 1996. ACM. ISBN 978-0-89791-809-1. doi: 10.1145/237502.241604. 150

[121] A. Kamil and K. Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-69329-1. doi: 10.1007/978-3-540-69330-7_13. xi, 6, 73, 117

[122] A. Kamil and K. Yelick. Enforcing Textual Alignment of Collectives Using Dynamic Checks. In G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 368–382. Springer Berlin Heidelberg, Oct. 2009. ISBN 978-3-642-13373-2 978-3-642-13374-9. doi: 10.1007/978-3-642-13374-9_25. 73

[123] R. M. Keller. Formal Verification of Parallel Programs. *Commun. ACM*, 19 (7):371–384, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360251. 65

[124] I. Laguna and M. Schulz. Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. IEEE Press, Nov. 2016. ISBN 978-1-4673-8815-3. 73

[125] L. Lamport. The 'Hoare logic'of concurrent programs. *Acta Informatica*, 14 (1):21–37, 1980. 65

[126] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *International Static Analysis Symposium*, pages 5–23. Springer, 2012. 74

[127] C. Lengauer. Loop Parallelization in the Polytope Model. In *International Conference on Concurrency Theory*, pages 398–416. Springer, 1993. 153

[128] C. Li and G. Hains. A simple bridging model for high-performance computing. In *High Performance Computing and Simulation (HPCS), 2011 International Conference On*, pages 249–256. IEEE, 2011. 56, 58

[129] G. Li, R. Palmer, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Formal Specification of MPI 2.0: Case Study in Specifying a Practical Concurrent Programming API. *Sci. Comput. Program.*, 76(2):65–81, Feb. 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2010.03.007. 61, 66, 69

[130] Y. Lin. Static Nonconcurrency Analysis of OpenMP Programs. In M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss,

editors, *OpenMP Shared Memory Parallel Programming*, Lecture Notes in Computer Science, pages 36–50. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68555-5. 74

[131] B. Lisper. Towards Parallel Programming Models for Predictability. In T. Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 48–58, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-41-5. doi: 10.4230/OASIcs.WCET.2012.48. 78

[132] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004. ISSN 1573-7640. doi: 10.1023/B:IJPP.0000029272.69895.c1. 24

[133] F. Loulergue. BSλppp: Functional BSP Programs on Enumerated Vectors. In M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, editors, *High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer Berlin Heidelberg, Oct. 2000. ISBN 978-3-540-41128-4 978-3-540-39999-5. doi: 10.1007/3-540-39999-2_34. 64, 67

[134] F. Loulergue. A Verified Accumulate Algorithmic Skeleton. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 420–426, Nov. 2017. doi: 10.1109/CANDAR.2017.108. 67

[135] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010. 63

[136] H. Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153, 2006. xiv, 185

[137] J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research: Oceans*, 102(C3):5753–5766, 1997. ISSN 2156-2202. doi: 10.1029/96JC02775. xiv, 185

[138] A. J. McPherson, V. Nagarajan, and M. Cintra. Static Approximation of MPI Communication Graphs for Optimized Process Placement. In *Lan-*

guages and Compilers for Parallel Computing, Lecture Notes in Computer Science, pages 268–283. Springer, Cham, Sept. 2014. ISBN 978-3-319-17472-3 978-3-319-17473-0. doi: 10.1007/978-3-319-17473-0_18. 74

[139] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), Sept. 2012. 24, 33, 40, 42, 64, 184

[140] J. Midtgaard, F. Nielson, and H. R. Nielson. Process-local static analysis of synchronous processes. In *International Static Analysis Symposium*, pages 284–305. Springer, 2018. 72

[141] R. Miller. A library for bulk-synchronous parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, pages 100–108, 1993. 24, 39, 63, 114

[142] R. Milner. *A Calculus of Communicating Systems*. Springer, Berlín, 1980. ISBN 978-3-540-10235-9 978-0-387-10235-1. OCLC: 911368693. 60

[143] A. Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation*, pages 39–58. Springer, Berlin, Heidelberg, Jan. 2014. doi: 10.1007/978-3-642-54013-4_3. 71

[144] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 978-1-58113-297-7. doi: 10.1145/378239.379017. 65

[145] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN Notices*, volume 42, pages 327–338. ACM, 2007. 76

[146] R. Nakade, E. Mercer, P. Aldous, and J. McCarthy. Model-Checking Task Parallel Programs for Data-Race. In *NASA Formal Methods*, pages 367–382. Springer, Cham, Apr. 2018. doi: 10.1007/978-3-319-77935-5_25. 69

[147] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer, 2002. 53

[148] N. Ng, N. Yoshida, O. Pernet, R. Hu, and Y. Kryftis. Safe parallel programming with session java. In *International Conference on Coordination Languages and Models*, pages 110–126. Springer, 2011. 75

[149] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 202–218. Springer, 2012. 75

[150] N. Ng, N. Yoshida, and W. Luk. Scalable Session Programming for Heterogeneous High-Performance Systems. In *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 82–98. Springer, Cham, Sept. 2013. ISBN 978-3-319-05031-7 978-3-319-05032-4. doi: 10.1007/978-3-319-05032-4_7. 67

[151] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Science & Business Media, Dec. 2004. ISBN 978-3-540-65410-0. 43, 44, 47, 51, 70, 83, 102, 111, 113, 147

[152] H. R. Nielson and F. Nielson. *Semantics with Applications*. Springer, 2007. 193

[153] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM*, 19(5):279–285, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. 65

[154] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic wcet analysis of real-time parallel applications. In *13th Workshop on Worst-Case Execution Time Analysis (WCET 2013)*, pages pp–11, 2013. 77

[155] S. Pelagatti. *Structured Development of Parallel Programs*, volume 102. Taylor & Francis London, 1998. 62

[156] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium On*, pages 46–57. IEEE, 1977. 68

[157] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer, 2004. 125

[158] S. Pophale, O. Hernandez, S. Poole, and B. M. Chapman. Extending the OpenSHMEM Analyzer to Perform Synchronization and Multi-valued

Analysis. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, number 8356 in Lecture Notes in Computer Science, pages 134–148. Springer International Publishing, Mar. 2014. ISBN 978-3-319-05214-4 978-3-319-05215-1. doi: 10.1007/978-3-319-05215-1_10. 73

[159] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. d Supinski, and D. J. Quinlan. Detecting Patterns in MPI Communication Traces. In *2008 37th International Conference on Parallel Processing*, pages 230–237, Sept. 2008. doi: 10.1109/ICPP.2008.71. 79

[160] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan. Using MPI Communication Patterns to Guide Source Code Transformations. In M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *Computational Science – ICCS 2008*, number 5103 in Lecture Notes in Computer Science, pages 253–260. Springer Berlin Heidelberg, June 2008. ISBN 978-3-540-69388-8 978-3-540-69389-5. doi: 10.1007/978-3-540-69389-5_29. 74

[161] J. Randmets. Static cost analysis, 2012. 125

[162] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 0002-9947, 1088-6850. doi: 10.1090/S0002-9947-1953-0053041-6. 43

[163] M. Rinard. Analysis of multithreaded programs. In *Static Analysis*, pages 1–19. Springer, 2001. 77

[164] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, USA, 1988. 161

[165] E. Saillard, P. Carribault, and D. Barthou. PARCOACH: Combining static and dynamic validation of MPI collective communications. *The International Journal of High Performance Computing Applications*, 28(4):425–434, 2014. 73, 79

[166] C. Santos, F. Martins, and V. T. Vasconcelos. Deductive Verification of Parallel Programs Using Why3. *Electronic Proceedings in Theoretical Computer Science*, 189:128–142, Aug. 2015. ISSN 2075-2180. doi: 10.4204/EPTCS.189.11. 67, 68

[167] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. *Specification, IBM, janvier*, 2012. 73

[168] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 265–273. Springer, 2008. 69

[169] D. R. Shires, L. L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of MPI programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA*, pages 1847–1853, 1999. 71, 72

[170] S. F. Siegel. Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 413–429. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-30579-8. 69

[171] S. F. Siegel. Model Checking Nonblocking MPI Programs. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, number 4349 in Lecture Notes in Computer Science, pages 44–58. Springer Berlin Heidelberg, Jan. 2007. ISBN 978-3-540-69735-0 978-3-540-69738-1. doi: 10.1007/978-3-540-69738-1_3. 69

[172] S. F. Siegel. Verifying Parallel Programs with MPI-Spin. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 13–14. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75416-9. 69

[173] S. F. Siegel and G. S. Avrunin. Verification of Halting Properties for MPI Programs Using Nonblocking Operations. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 326–334. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75416-9. 69

[174] S. F. Siegel and T. K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, Dec. 2011. ISSN 1661-8289. doi: 10.1007/s11786-011-0100-7. 69

[175] S. F. Siegel and T. K. Zirkel. Loop Invariant Symbolic Execution for Parallel Programs. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, V. Kuncak, and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148, pages 412–427. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27939-3 978-3-642-27940-9. xvi, 187

[176] A. Spector and D. Gifford. The Space Shuttle Primary Computer System. *Commun. ACM*, 27(9):872–900, Sept. 1984. ISSN 0001-0782. doi: 10.1145/358234.358246. 25

[177] G. Staple and K. Werbach. The end of spectrum scarcity [spectrum allocation and utilization]. *IEEE spectrum*, 41(3):48–52, 2004. v, 1

[178] T. Sterling, M. Anderson, and M. Brodowicz. Chapter 8 - The Essential MPI. In T. Sterling, M. Anderson, and M. Brodowicz, editors, *High Performance Computing*, pages 249–284. Morgan Kaufmann, Boston, Jan. 2018. ISBN 978-0-12-420158-3. doi: 10.1016/B978-0-12-420158-3.00008-3. 2

[179] A. Stewart, M. Clint, and J. Gabarró. Axiomatic frameworks for developing BSP-style programs. *Parallel Algorithms And Application*, 14(4):271–292, 2000. 42, 67

[180] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *Parallel Processing, 2006. ICPP 2006. International Conference On*, pages 175–184. IEEE, 2006. 71, 72

[181] W. Suijlen. Mock BSPlib for Testing and Debugging Bulk Synchronous Parallel Software. *Parallel Processing Letters*, 27(01):1740001, 2017. 79

[182] W. J. Suijlen and P. Krusche. BSPonMPI. *URL: http://bsponmpi. sourceforge. net*, 2013. 39

[183] W. J. Suijlen and A. N. Yzelman. Lightweight Parallel Foundations: A new communication layer. Technical Report DPSL-PARIS-TR-2018-09, Huawei Technologies France / 2012 Laboratories / CSI / DPSL / PADAL, 20 Quai du Point du Jour, 92100 Boulogne-Billlancourt, France, 2018. 39, 40, 41, 62, 63, 102

[184] J. Tesson and F. Loulergue. Formal Semantics of DRMA-Style Programming in BSPlib. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and

J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967, pages 1122–1129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68105-2 978-3-540-68111-3. 67, 117, 132, 184

[185] J. Tesson and F. Loulergue. A verified bulk synchronous parallel ML heat diffusion simulation. *Procedia Computer Science*, 4:36–45, 2011. 67

[186] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD Thesis, University of Oxford, 1998. viii, 4, 59

[187] S. Tripakis, C. Stergiou, and R. Lublinerman. Checking Equivalence of SPMD Programs Using Non-Interference. Technical Report UCB/EECS-2010-11, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, Jan. 2010. 74

[188] A. Turing. Checking a large routine. In *The Early British Computer Conferences*, pages 70–72. MIT Press, 1989. 65

[189] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *International Conference on Computer Aided Verification*, pages 66–79. Springer, 2008. 69

[190] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345258. 69

[191] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. viii, 4, 56

[192] L. G. Valiant. A Bridging Model for Multi-core Computing. *J. Comput. Syst. Sci.*, 77(1):154–166, Jan. 2011. ISSN 0022-0000. doi: 10.1016/j.jcss.2010.06.012. 56, 58, 59

[193] M. van Duijn. Extending the BSP model to hierarchical heterogeneous architectures. Master's thesis, Utrecht University, 2018. 39

[194] S. Verdoolaege. *Isl*: An Integer Set Library for the Polyhedral Model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010. 143, 147

[195] S. Verdoolaege and T. Grosser. Polyhedral Extraction Tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012. 141

[196] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica*, 48(1):37–66, 2007. 143, 147

[197] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 978-0-7803-9802-3. 79

[198] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. *ACM Sigplan Notices*, 44(4):261–270, 2009. 69

[199] P. Wang, Y. Du, H. Fu, X. Yang, and H. Zhou. Static Analysis for Application-Level Checkpointing of MPI Programs. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 548–555, Sept. 2008. doi: 10.1109/HPCC.2008.39. 74

[200] B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, Sept. 1975. ISSN 0001-0782. doi: 10.1145/361002.361016. 120, 121

[201] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The Worst-Case Execution-Time Problem—overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008. 77, 147

[202] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In M. Dal Cin, M. Kaâniche, and A. Pataricza, editors, *Dependable Computing - EDCC 5*, volume 3463 of *LNCS*, pages 281–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32019-7. 53

[203] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT press, 1993. 45, 46, 65, 165

[204] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):

825–836, Sept. 1998. ISSN 1096-9128. doi: 10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H. xi, 6

[205] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array dataflow analysis for polyhedral X10 programs. In *ACM SIGPLAN Notices*, volume 48, pages 23–34. ACM, 2013. 74, 76

[206] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Checking Race Freedom of Clocked X10 Programs. *arXiv:1311.4305 [cs]*, Nov. 2013. 74, 76

[207] A. Yzelman and R. H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, Apr. 2012. ISSN 1532-0634. doi: 10.1002/cpe.1843. 39, 40, 183

[208] A. N. Yzelman, R. H. Bisseling, and D. Roose. MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming. *International Journal of Parallel Programming*, 42(4):619–642, Aug. 2014. ISSN 0885-7458, 1573-7640. doi: 10.1007/s10766-013-0262-9. 39, 40, 59, 63

[209] Y. Zhang and E. Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229472. 73, 75, 79, 82, 116, 187

[210] Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 95–109. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85261-2. 74

[211] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: Formal verification of parallel programs. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference On*, pages 830–835. IEEE, 2015. 69, 79

[212] J. Zhou and Y. Chen. Generating C Code from LOGS Specifications. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. Van Hung,

and M. Wirsing, editors, *Theoretical Aspects of Computing – ICTAC 2005*, volume 3722, pages 195–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-29107-7 978-3-540-32072-2. 79

[213] W. Zimmermann. *Automatic Worst Case Complexity Analysis of Parallel Programs*. International Computer Science Institute, 1990. 78, 152

[214] S. F. S. Ziqing Luo. Towards Deductive Verification of Message-Passing Parallel Programs, 2018. 67

[215] T. K. Zirkel, S. F. Siegel, and T. McClory. Automated Verification of Chapel Programs Using Model Checking and Symbolic Execution. In *NASA Formal Methods*, pages 198–212. Springer, Berlin, Heidelberg, May 2013. doi: 10.1007/978-3-642-38088-4_14. 69, 70

This document has been written in the GNU Emacs editor and the LaTeX 2ε document preparation system.

Filip Arvid JAKOBSSON

Analyse statique des programmes BSPlib

Résumé : La programmation parallèle consiste à utiliser des architectures à multiples unités de traitement, de manière à ce que le temps de calcul soit inversement proportionnel au nombre d'unités matérielles. Le modèle de BSP (*Bulk Synchronous Parallel*) permet de rendre le temps de calcul prévisible. BSPlib est une bibliothèque pour la programmation BSP en langage C. En BSPlib on entrelace des instructions de contrôle de la structure parallèle globale, et des instructions locales pour chaque unité de traitement. Cela permet des optimisations fines de la synchronisation, mais permet aussi l'écriture de programmes dont les calculs locaux divergent et masquent ainsi l'évolution globale du calcul BSP.

Toutefois, les programmes BSPlib réalistes sont *syntaxiquement alignés*, une propriété qui garantit la convergence du flot de contrôle parallèle. Dans ce mémoire nous étudions les trois dimensions principales des programmes BSPlib du point de vue de l'alignement syntaxique : la synchronisation, le temps de calcul et la communication. D'abord nous présentons une analyse statique qui identifie les instructions syntaxiquement alignées et les utilise pour vérifier la sûreté de la synchronisation globale. Cette analyse a été implémentée en Frama-C et certifiée en Coq. Ensuite nous utilisons l'alignement syntaxique comme base d'une analyse statique du temps de calcul. Elle est fondée sur une analyse classique du coût pour les programmes séquentiels. Enfin nous définissons une condition suffisante pour la sûreté de l'enregistrement des variables. L'enregistrement en BSPlib permet la communication par accès aléatoire à la mémoire distante (DRMA) mais est sujet à des erreurs de programmation. Notre développement technique est la base d'une future analyse statique de ce mécanisme.

Mots-clés : Programmation parallèle • Bulk Synchronous Parallelism • SPMD • Analyse statique • Synchronisation • Analyse de coût • Communication

Static Analysis for BSPlib Programs

Abstract: The goal of scalable parallel programming is to program computer architectures composed of multiple processing units so that increasing the number of processing units leads to an increase in performance. *Bulk Synchronous Parallel* (BSP) is a widely used model for scalable parallel programming with predictable performance. BSPlib is a library for BSP programming in C. In BSPlib, parallel algorithms are expressed by intermingling instructions that control the global parallel structure, and instructions that express the local computation of each processing unit. This lets the programmer fine-tune synchronization, but also implement programs whose diverging parallel control flow obscures the underlying BSP structure. In practice however, the majority of BSPlib program are *textually aligned*, a property that ensures parallel control flow convergence. We examine three core aspects of BSPlib programs through the lens of textual alignment: synchronization, performance and communication. First, we present a static analysis that identifies textually aligned statements and use it to verify safe synchronization. This analysis has been implemented in Frama-C and certified in Coq. Second, we exploit textual alignment to develop a static performance analysis for BSPlib programs, based on classic cost analysis for sequential programs. Third, we develop a textual alignment-based sufficient condition for safe registration. Registration in BSPlib enables communication by Direct Remote Memory Access but is error prone. This development forms the basis for a future static analysis of registration.

Keywords: Parallel programming • Bulk Synchronous Parallelism • SPMD • Static Analysis • Synchronization • Cost analysis • Communication

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO), Université d'Orléans. Faculté des Sciences, Bâtiment IIIA. Rue Léonard de Vinci B.P. 6759 F-45067 ORLEANS Cedex 2, France