Thesis submitted in total fulfillment
of the requirements for the award of

DOCTOR OF PHILOSOPHY OF THE UNIVERSITY OF BORDEAUX

at the École doctorale de Mathématique et Informatique de Bordeaux
Specialized in: **Computer Science**

By **Emmanuelle SAILLARD**

Static/Dynamic Analyses for Validation and Improvement of Multi-models HPC Applications

---

Hosted by:
CEA, DAM, DIF
F-91297 Arpajon, France

After obtaining the opinion of:

Pr. Matthias S. Müller	Professor at University of RWTH Aachen	Referee
Dr. Fabrice Rastello	Director of research, INRIA	Referee

Defended the 24th of September 2015 in front of the following doctoral Committee:

Dr. Emmanuel Jeannot	Director of research, INRIA	Jury President
Pr. Denis Barthou	Professor at University of Bordeaux, INRIA	Advisor
Dr. Patrick Carribault	Research Scientist at CEA, HDR	Advisor
Pr. Matthias S. Müller	Professor at University of RWTH Aachen	Referee
Dr. Fabrice Rastello	Director of research, INRIA	Referee
Dr. Torsten Hoefler	Assistant Professor, ETH Zürich	Examiner

Thèse présentée
pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

École doctorale de Mathématique et Informatique de Bordeaux
Spécialité : **Informatique**

Présentée par **Emmanuelle SAILLARD**

Analyse statique/dynamique pour la validation et l'amélioration des applications parallèles multi-modèles

---

Organisme d'accueil:
CEA, DAM, DIF
F-91297 Arpajon, France

Après avis de :

M. Matthias S. Müller	Professeur à l'université de RWTH Aachen	Rapporteur
M. Fabrice Rastello	Chargé de recherche, INRIA	Rapporteur

Soutenue le 24 septembre 2015 devant la commission d'examen composée de :

M. Emmanuel Jeannot	Directeur de recherche, INRIA	Président
Pr. Denis Barthou	Professeur à l'université de Bordeaux, INRIA	Directeur de thèse
M. Patrick Carribault	Ingénieur-chercheur au CEA, HDR	Encadrant CEA
M. Matthias S. Müller	Professeur à l'université de RWTH Aachen	Rapporteur
M. Fabrice Rastello	Chargé de recherche, INRIA	Rapporteur
M. Torsten Hoefler	Professeur Assistant, ETH Zürich	Examinateur

Abstract

Supercomputing plays an important role in several innovative fields, speeding up prototyping or validating scientific theories. However, supercomputers are evolving rapidly with now millions of processing units, posing the questions of their programmability. Despite the emergence of more widespread and functional parallel programming models, developing correct and effective parallel applications still remains a complex task. Although debugging solutions have emerged to address this issue, they often come with restrictions.

However programming model evolutions stress the requirement for a convenient validation tool able to handle hybrid applications. Indeed as current scientific applications mainly rely on the Message Passing Interface (MPI) parallel programming model, new hardwares designed for Exascale with higher node-level parallelism clearly advocate for an MPI+X solutions with X a thread-based model such as OpenMP. But integrating two different programming models inside the same application can be error-prone leading to complex bugs - mostly detected unfortunately at runtime. In an MPI+X program not only the correctness of MPI should be ensured but also its interactions with the multi-threaded model, for example identical MPI collective operations cannot be performed by multiple non-synchronized threads.

This thesis aims at developing a combination of static and dynamic analysis to enable a scalable verification of hybrid HPC applications. The first pass statically verifies the thread level required by an MPI+OpenMP application and outlines execution paths leading to potential deadlocks. Thanks to this analysis, the code is selectively instrumented, displaying an error and synchronously interrupting all processes if the actual scheduling leads to a deadlock situation. Our method has been implemented as a plugin in the GCC compiler to avoid compiler recompilation.

Key words: High Performance Computing, Static analysis, Collective, MPI, OpenMP, Debugging

Résumé

L'utilisation du parallélisme des architectures actuelles dans le domaine du calcul hautes performances oblige à recourir à différents langages parallèles. Ainsi, l'utilisation conjointe de MPI pour le parallélisme gros grain, à mémoire distribuée et OpenMP pour du parallélisme de thread fait partie des pratiques de développement d'applications pour supercalculateurs. Des erreurs liées à l'utilisation conjointe de ces langages de parallélisme sont actuellement difficiles à détecter et cela limite l'écriture de codes permettant des interactions plus poussées entre ces niveaux de parallélisme.

Des outils ont été proposés afin de pallier ce problème. Cependant, ces outils sont généralement focalisés sur un type de modèle et permettent une vérification dite statique (à la compilation) ou dynamique (à l'exécution). Pourtant une combinaison statique/dynamique donnerait des informations plus pertinentes. En effet, le compilateur est en mesure de donner des informations relatives au comportement général du code, indépendamment du jeu d'entrée. C'est par exemple le cas des problèmes liés aux communications collectives du modèle MPI.

Cette thèse a pour objectif de développer des analyses statiques/dynamiques permettant la vérification d'une application parallèle mélangeant plusieurs modèles de programmation afin de diriger les développeurs vers un code parallèle multi-modèles correct et performant.

Nous avons créé la plateforme PARCOACH, un plugin GCC qui détecte des erreurs dans les applications parallèlisées avec MPI, OpenMP et MPI+OpenMP, combinant des analyses statiques et dynamiques. La vérification se fait en deux étapes. Premièrement, de potentielles erreurs sont détectées lors de la phase de compilation. Ensuite, un test au runtime est ajouté pour savoir si le problème va réellement se produire. Grâce à ces analyses combinées, PARCOACH renvoie des messages précis aux utilisateurs et évite les situations de blocage.

Mots clés: Calcul haute performance, Analyse statique, Collectives, MPI, OpenMP, Débogage

Acknowledgement

TODO

Preamble

In the High Performance Computing (HPC) field, one of the major issue consists in debugging parallel applications. With progress to exascale systems, this is particularly true. Parallel programming models evolve with new features and are getting more complex. Furthermore with the increase of core number per node, the trend is to make applications hybrid by mixing models. As scientific applications mainly rely on the MPI parallel programming model, progress to exascale systems advocates for MPI+X solutions with X a thread-based approach like OpenMP. But integrating two different programming models inside the same application makes the debugging phase more challenging (complex models, new errors, ...). Although debugging solutions have emerged to address this issue, they often come with restrictions. Developers then face errors more difficult to locate and correct with few efficient help.

To fill this gap, this thesis aims at developing a combination of static and dynamic analysis to enable a precise and early verification of hybrid HPC applications. That is why we designed the PARallel COntrol flow Anomaly CHecker framework. The framework detects errors in two steps. The first pass statically verifies the thread level required by an MPI+OpenMP application and outlines all execution paths leading to potential deadlocks. With the help of this analysis, the code is then selectively instrumented, displaying an error and synchronously interrupting all processes if the actual scheduling leads to a deadlock situation.

Manuscript Outline

This thesis is organized in five chapters. The first chapter describes the context of the thesis. After a brief introduction to the high performance computing domain, the first chapter points out exascale challenges induced by supercomputer architecture evolution (**Section 1.1**). Then the chapter focuses on the parallel applications debugging challenge and gives an overview of the current state of work in this area especially on MPI and OpenMP, that are the two most used models in HPC applications (**Section 1.2**).

The three following chapters present the contributions of the thesis:

1. Verification of the compliance of MPI+OpenMP applications with MPI thread levels defined in the MPI-2 standard (**Chapter 2**);

2. Detection of MPI collective errors origin in MPI applications (**Chapter 3**);

3. Detection of the misuse origin of OpenMP barriers and worksharing constructs in OpenMP applications (**Chapter 3**);

4. Detection of MPI collective errors origin in a multi-threaded context (**Chapter 4**);

5. Based on the fact that no benchmark exists to highlight functionnalities of debugging tools, we propose an error benchmark suite reflecting common errors made when using MPI and OpenMP programming models (**Chapter 2**);

Finally the last chapter opens a discussion and concludes the thesis (**Chapter 5**).

Contents

Preamble **7**

1 Introduction **9**

1.1 Introduction to High Performance Computing . . . 9
1.1.1 Modern Supercomputers . . . 9
1.1.2 Thesis Computing Environment . . . 10
1.1.3 Programming Models for HPC . . . 11
1.1.4 MPI: Message Passing Interface . . . 12
1.1.5 PGAS . . . 14
1.1.6 Pthreads . . . 14
1.1.7 OpenMP . . . 15
1.1.8 Heterogeneous Architectures Programming . . . 16
1.1.9 Exascale Challenges . . . 16
1.1.10 Summary . . . 17
1.2 Debugging Parallel Constructs of HPC applications . . . 18
1.2.1 Software Life-Cycle Models . . . 18
1.2.2 Debugging Parallel Applications . . . 20
1.2.3 Verification of MPI Applications . . . 24
1.2.4 Verification of OpenMP Applications . . . 27
1.2.5 Verification of MPI+OpenMP Applications . . . 29
1.3 Outline . . . 30

2 Interaction Between MPI and shared memory models **33**

2.1 MPI Thread-Level Checking for MPI+OpenMP Applications . . . 33
2.1.1 Analysis of the Multithreaded Context . . . 34
2.1.2 Thread-Level Compliance Checking . . . 38
2.2 PARallel COntrol flow Anomaly CHecker (PARCOACH) . . . 44
2.3 Revealing PARCOACH Functionalities . . . 45
2.4 Summary . . . 46

3 Detection of Collective Errors Origin in Parallel Applications **51**

3.1 Combining Static and Dynamic Analyses to Find the Origin of MPI Collective Errors . . . 51
3.1.1 Compile-Time Verification . . . 53
3.1.2 Static Instrumentation for Execution-Time Verification . . . 56
3.1.3 Evaluation . . . 57
3.2 Combining Static and Dynamic Analyses to Find the Origin of OpenMP Collective Errors . 61
3.2.1 Checking OpenMP Directives and Control Flow . . . 62
3.2.2 Intra-Procedural Analysis . . . 63
3.2.3 Static Instrumentation for Execution-Time Verification . . . 65

3.2.4 Inter-Procedural Analysis 66
3.2.5 Evaluation 67
3.3 Summary 69

4 Detection of Collective Errors Origin in Applications Mixing Parallel Programming Models 71
4.1 Static and Dynamic Validation of MPI Collective Communications in Multi-threaded Context 71
4.1.1 Problem Statement 73
4.1.2 Compile-Time Verification 74
4.1.3 Static Instrumentation for Execution-Time Verification 78
4.1.4 Evaluation 81
4.2 Summary 83

5 Conclusion and Perspectives 85
5.1 Conclusion 85
5.2 Work in progress 86
5.3 Perspectives 91
5.3.1 Short-term Improvements 91
5.3.2 General Perspectives 93

Appendices 95

A GCC, the GNU Compiler Collection 97
A.1 Structure of GCC 97
A.2 GCC's history 98

B Key concepts 101
B.1 Dominance/Postdominance 101
B.2 Dominance/Postdominance Frontier 102

C Details on benchmarks 103
C.1 NAS Parallel benchmarks 103
C.2 Coral 104

Bibliography 105

List of Figures 115

List of Tables 117

Chapter 1

Introduction

> "The continued drive for higher- and higher-performance systems [...] leads us to one simple conclusion: the future is parallel".
>
> Michael J. Flynn and Kevin W. Rudd in Parallel Architectures [1]

The purpose of this chapter is to set the context of the thesis. **Section 1.1** introduces basic concepts related to the High Performance Computing (HPC) field. The section exposes supercomputers architecture and their evolution, makes a short summary of parallel programming models and ends with the challenges induced by supercomputer evolution and the issues targeted in the scope of this thesis. **Section 1.2** discusses development methodology and more precisely the debugging phase (error location and correction). Then the section summarizes common errors in MPI and OpenMP applications and gives existing debugging solutions for these parallel programming models.

1.1 Introduction to High Performance Computing

Computation is an essential tool in solving many of today's highly complex scientific and engineering problems. Performing realistic simulations requires a sufficient computational power to obtain results with the desirable degree of accuracy in a reasonable time. Furthermore since the amount of memory consumed by large and complex problems is high, a lot of storage capacity is also required. These requirements cannot be bought by common computers. That is why supercomputers were born. The term supercomputer was defined in [2] by C. Morris as any of a category of extremely powerful, large-capacity mainframe computers that are capable of manipulating massive amounts of data in an extremely short time. Thus a supercomputer can be seen as one of the largest, fastest, and most powerful computer available at a given time.

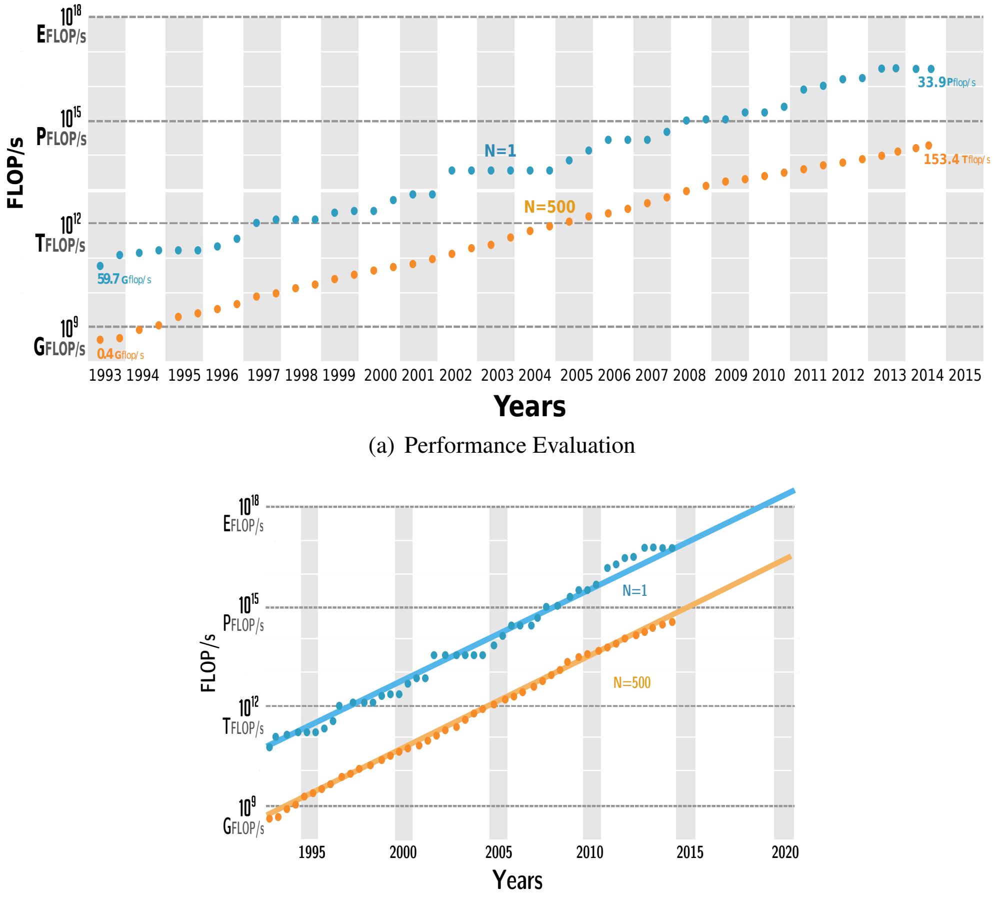
The first supercomputer, a 64-processor SIMD machine called ILLIAC IV, was designed in 1967 by the University of Illinois and the Burroughs Corporation. At that time, ILLIAC IV was the fastest computer achieving record speeds of 200 million instructions per second (Mips). It contained four control units: each one was controlling a 64 arithmetic and logic unit array processor. Since then, huge technological progress has been achieved. Today most supercomputers are massively parallel machines and contain thousands of processors, heterogeneous processing clusters, and can compute quadrillions calculations per second.

1.1.1 Modern Supercomputers

Twice a year, the most powerful systems are tested and ordered based on their Linpack[1] performance to form the TOP500 list [3]. Supercomputers are ranked by their ability to solve a dense system of linear

[1] The Linpack Benchmark is a measure of a computer's floating-point rate of execution

equations. In November 2014 the Chinese supercomputer Tianhle-2[2] was recorded as the most powerful supercomputer with a performance of 33.86 petaflop/s ($10^{15} Flop/s$) on the Linpack benchmark. As a comparison, the most powerful supercomputer reached a performance of 170 gigaflop/s ($10^{9} Flop/s$) in November 1994 and a performance of 70.7 teraflop/s ($10^{12} Flop/s$) in November 2004, almost one thousand times less powerful than today's supercomputers. **Figure 1.1(a)** shows the performance evolution of the first and the 500th supercomputer since the TOP500 project was launched in 1993. The predicted performance evolution until 2020 is presented on **Figure 1.1(b)**. As pointed out in the figure, supercomputing power is growing exponentially and exascale systems (10^{18} FLOP/s) are coming shortly. There are however good chances that this evolution does not go forever as many argue the end of Moore's law.



(a) Performance Evaluation

(b) Projected Performance Evaluation

Figure 1.1: *Growth of supercomputing power of the first (N=1) and the last (N=500) supercomputer recorded by TOP500 list (see http://www.top500.org)*

1.1.2 Thesis Computing Environment

All experiments of this thesis were conducted on Tera 100 [4] and Curie [5] petaflopic supercomputers and Inti, a prototype of Tera 100. Tera 100 was developed by Bull in 2008 for the French Atomic Energy Commission (CEA) and designed for the French nuclear simulation program. Tera 100 hosts 4 370 compute nodes for a total of 138 368 cores. Each compute node gathers four eight-core Nehalem Ex processors at 2.27 GHz and 64 GB of RAM. Tera 100 aggregated a peak performance of 1.05 petaflop/s and was the sixth

[2] Tianhle-2 (Milkyway-2) - TH-IVB-FEP cluster, Intel Xeon E5-2692 12C 2.200GHZ, TH Express-2, Intel Xeon Phi 31S1P

supercomputer of the TOP500 list in november 2010. Curie was ordered by GENCI[3] and was operated into the TGCC[4] by CEA. Curie is open to scientists through the French participation into the PRACE[5] research infrastructure. Curie was the ninth supercomputer in november 2012 with a peak performance of 1.359 petaflop/s. Tera 100 and Curie are both manufactured by Bull and thus feature a similar design. **Table 1.1** summarizes a description of these machines (current rank in the TOP500 list, total number of cores, ...).

Table 1.1: *Characteristics of Tera 100, Curie and Inti supercomputers (2014, [3])*

Characteristic	**Tera 100**	**Curie**	**Inti**
TOP 500 rank	47	33	-
Total Number of cores	138 368	77 184	872
Linpack Performance (R_{max})	1 050 TFlop/s	1 359 TFlop/s	-
Theoretical Peak (R_{peak})	1 254.5 TFlop/s	1 667.17 TFlop/s	-
Processor type	Intel Xeon 7500	Intel Xeon E5-2680	Intel Xeon E5640
Memory per core	2 GB	4 GB	2.725 GB
Total Memory	276 736 GB	308 736 GB	2 376.2
Operating System	Linux (Redhat)	Linux (Redhat)	Linux (Redhat)
Interconnect	Infiniband	Infiniband	Infiniband
Network Topology	Fat-tree	Fat-tree	Fat-tree

Building such powerful systems is not sufficient. Dedicated parallelism programming is needed to obtain the desired performance. The next section describes the possibilities to exploit supercomputers.

1.1.3 Programming Models for HPC

Wilkinson *et al.* characterize in [6] the parallel programming as programming multiple computers, or computers with multiple internal processors. The aim is to solve a problem at a greater computational speed than it is possible with a single computer. A parallel program gathers concurrently executing processes, which may be connected to one another through either message-passing or accesses to shared data. A parallel programming model specifies what data can be named by the processes, what operations can be done on the named data, and what ordering exists among these operations. There is today a large variety of programming models exposing these two prevailing parallel computing memory architectures.

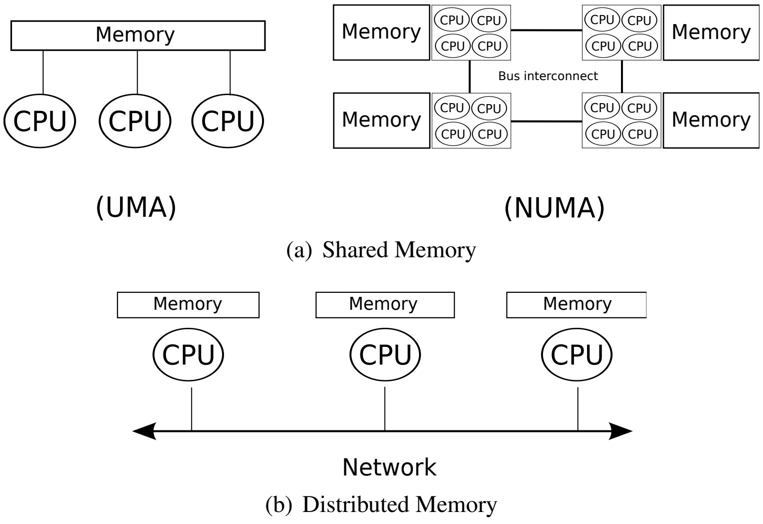
Shared memory machines have been classified as UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access), depending on memory access times (see **Figure 1.2(a)**). UMA is commonly represented by Symmetric Multiprocessor (SMP) machines when processors are identical. NUMA is made by physically linking several SMPs. One SMP can directly access memory of another SMP. In shared memory approaches as Pthreads or OpenMP, data are shared and it is the user responsibility to ensure the coherency of concurrent accesses in global memory. This issue can not arise in distributed memory as each instance has its private copy of data. Data are moved from the address space of a process to another process address space through cooperative operations on each process. The programmer has to specify what data to send and where. These architectures are mostly managed with message passing models like PVM or MPI. Distributed memory systems require a communication network to connect inter-processor memory (see **Figure 1.2(b)**).

Laments about the difficulty of using MPI (e.g., the lack of compile or runtime help, complexity of nonblocking communication) have bought new approaches that enables a tradeoff between distributed and shared memory approaches as it is the case for PGAS models. Some PGAS languages include UPC, Co-Array Fortran, Chapel, X10, Phalanx, Titanium as will be seen in more details later.

[3] **GENCI**: *Grand Equipement National de Calcul Intensif*

[4] **TGCC**: *Très Grand Centre de Calcul*

[5] **PRACE**: *Partnership for Advanced Computing in Europe*



(a) Shared Memory

(b) Distributed Memory

Figure 1.2: *Shared and Distributed Memory Architectures*

One of the most challenging characteristics of today's parallel environment is the emergence of heterogeneity systems. The growth of computer games have encouraged the development of graphics co-processors especially attractive to the HPC field for energy efficiency. Heterogeneous architectures have lead to parallel programming models devoted for accelerators (OpenACC, CUDA, OpenCL).

1.1.4 MPI: Message Passing Interface

The Message Passing Interface [7] is a message-passing library interface specification created in 1993. MPI is a specification, not a language. All MPI operations are expressed as functions, subroutines or methods for C and Fortran languages. Each MPI process executes a parallel instance of a program in a private address space and exchanges data across distributed memory systems via messages. MPI exposes multiple ways to express communications between tasks/processes including point-to-point and collective. While point-to-point functions involve only two tasks, collective communications involve a group (called communicator) of processes. **Figure 1.3** shows an example of a MPI program with three MPI processes. Each MPI process runs on one CPU[6] core. In the figure, a process sends a message to another process.

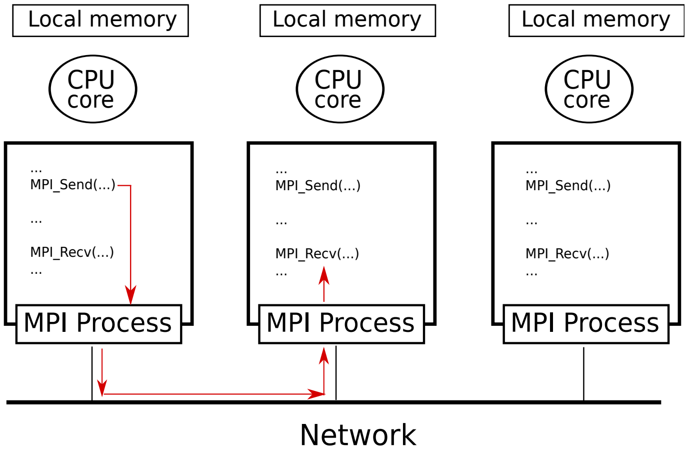


Figure 1.3: *Example of three MPI processes executing point-to-point communications (send/recv).*

[6] **CPU**: Central Processing Unit

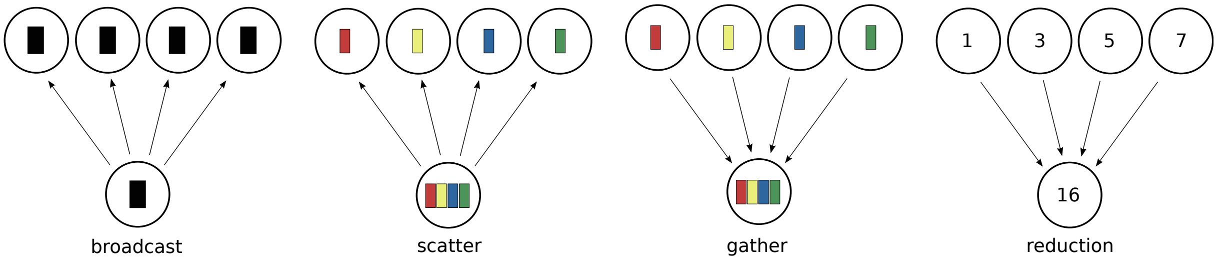


Figure 1.4: *Collective Communication Routines [8].*

MPI Collective Operations

A collective operation involves all processes in a MPI communicator. By default, all MPI processes are in MPI_COMM_WORLD. Then they can be partitioned in new communicators (e.g., `MPI_Comm_split`, `MPI_Comm_create`). The MPI specification requires that all processes must call blocking and non-blocking collective operations in the exact same order. If all processes in a communicator do not participate in the collective, unexpected behavior, including program failure, can occur. Non-blocking collectives are available since the MPI-3 standard. They follow the same principle as non-blocking point-to-point communication. A call to a non-blocking collective operation on a communicator initiates the collective without completing it. However while it is possible to match a blocking send (resp. non-blocking send) with a non-blocking receive (resp. blocking receive), it is not possible to do the same with collective operations. For example, a call to the blocking barrier `MPI_Barrier` on some processes of the communicator cannot match with a call to the non-blocking barrier `MPI_Ibarrier` on the remaining processes. All processes have to call `MPI_Barrier` or `MPI_Ibarrier` to be correct.

There are three main types of collective operations: Synchronization (barrier), Data Movement (broadcast, scatter/gather, all to all) and Collective Computation (reductions). **Figure 1.4** shows the fourth basic collectives. `MPI_Bcast` broadcast a message from the process with rank root to all processes of the group, itself included. With the `MPI_Gather` function, each process, root process included, sends the contents of its send buffer to the root process. Conversely, the root process scatters its send buffer to all other processes in the group with `MPI_Scatter`. The `MPI_Reduce` function performs a global reduction operation (sum, min, max, etc.). One member of the group collects data from the other members and effectuates the reduction operation on that data. The reduction operation can be either one of the predefined list of operations, or a user-defined operation.

MPI Implementations

Two main MPI libraries have been developed and are open source: MPICH [9] (and its recent successor MPICH3) and Open MPI [10]. Some machine constructors have created their own MPI implementation. Some examples are IntelMPI [11], BullxMPI [12], IBM Platform MPI [13] and Cray MPI. The MultiProcessor Computing Framework (MPC) [14, 15] implements a thread-based MPI runtime. It provides a unified runtime with its own implementation of the POSIX threads, OpenMP [16] and MPI. Since the project started in 2003, it now conforms to POSIX threads, OpenMP 2.5 standard and is fully MPI 1.3 compliant. It also supports the MPI_THREAD_MULTIPLE level. The compilation of a program is done through a patched version of GCC called MPI_GCC. This one converts standard C, C++ and Fortran MPI codes into a thread-based MPI implementation [17]. As other MPI implementation, MPC is constantly evolving and is freely available [18].

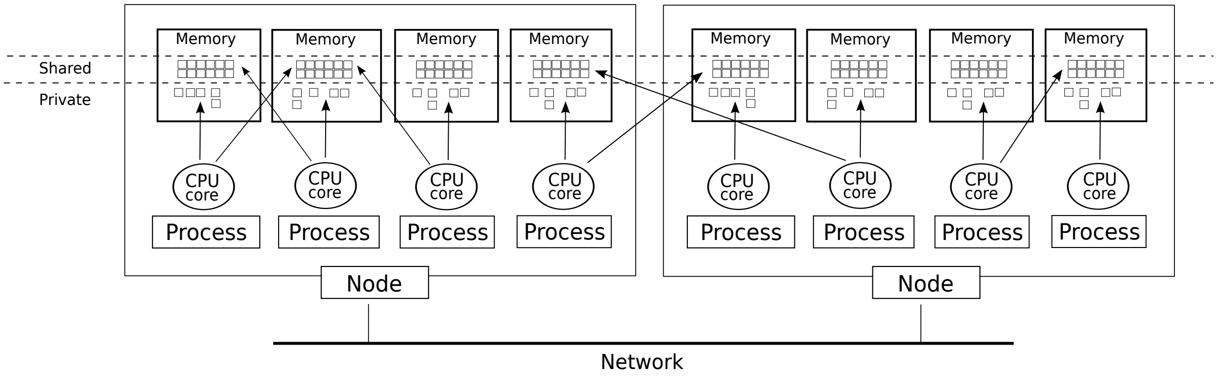


Figure 1.5: *Example of the PGAS model.*

The Success of MPI

MPI is widely used in scientific applications. Its success is explained by W. D. Gropp in [19]. In this article, the author gives six properties that make MPI so successful: *portability*, *performance*, *simplicity and symmetry*, *modularity*, *composability* and *completeness*. According to the author, these properties should be satisfied by any parallel programming model to succeed.

MPI has the ability to run on most parallel platforms (*portability*) which ensures parallel applications durability and to deliver the available performance of the underlying hardware (*performance*). MPI is simple in the sense that using MPI requires learning only a few concepts (*simplicity*). Indeed many MPI programs can be written with only a few routines. In addition to being a complete programming model (*completeness*), which means any parallel algorithm can be implemented with it, MPI enables the exploitation of other tools improvements (*composability*) and supports a large enough community (*modularity*). MPI was designed to work with other tools such as compilers and debuggers.

1.1.5 PGAS

Partitioned Global Address Space (PGAS) [20] is a distributed shared-memory model. The global address space means that threads may directly read/write remote data and partitioned means that data is designated as local or global. It allows a global view of data by an abstracted shared address space and hides the distinction between shared and distributed memory. Common PGAS languages include Unified Parallel C [21] (an extension of the C programming language for SPMD parallel programming), Co-Array Fortran (a small set of extensions to Fortran 95 for SPMD parallel programming), Titanium [22] , X-10 [23] (a language developed by IBM) and Chapel [24] (a language led by Cray).

We can mention two UPC projects: the Berkeley Unified parallel C project [25] and the GNU UPC [26]. The last one provides a compilation and execution environment for UPC programs. UPC combines the programmability advantages of the shared memory programming paradigm, the control over data layout and performance of the message passing programming paradigm. In a UPC program, threads operate in a partitioned global address space logically distributed among them. Each thread has affinity with a portion of the globally shared address space and has a private space. UPC provides synchronization mechanisms (barriers, locks, fence, spinlocks). As an example, the barrier synchronization is achieved by the function `upc_barrier`. Like MPI, UPC requires that all threads/processes have the same sequence of collective operations.

1.1.6 Pthreads

The POSIX Threads, commonly referred to as Pthreads [27] stands for the official IEEE POSIX 1003.1c standard (1995), which was established by the IEEE standard committee [28]. It uses the shared memory approach where a unique memory is shared between CPUs. A Pthreads program starts with a single default thread. All other threads must be explicitly created by the programmer and have direct access to data inside a node (but can have their own private data) and have the same address space. The Pthreads interface offers a set of C functions for thread management (e.g., `pthread_create`, `pthread_exit`, `pthread_join`), mutexes (e.g., `pthread_mutex_init`, `pthread_mutex_lock`), condition variables (e.g., `pthread_cond_init`) and thread synchronization (e.g., `pthread_barrier_init,pthread_barrier_wait`).

1.1.7 OpenMP

OpenMP [29] is a shared memory programming model and is based on the fork-join model of parallel execution. An OpenMP program begins as a single thread of execution (initial thread) which creates a team of itself when it encounters a `parallel` construct (fork). At the end of the parallel region, all threads are asleep except the initial thread (join). Nested parallelism and orphaned directives are allowed [30].

The following typical work-sharing constructs are available.

`For` construct

The first construct that has been implemented since OpenMP 1.0 and probably what OpenMP is best known for, enables loop parallelization. The user can control how the runtime handles the loop using additional clauses, which can change the scheduler and/or the granularity of the loop.

`Tasking` construct

Since OpenMP 3.0, the `task` construct has been added. The programmer can use this construct to create tasks explicitly, and can create synchronization points between them using a `taskwait` directive. As of OpenMP 4.0, the programmer can also specify data dependencies between tasks instead of using an explicit synchronization, leaving this responsibility to the runtime. As for the `for` construct, the programmer can specify clauses for the `task` construct, which provide more control on how tasks are created (e.g., control the grain when using task-generating loops).

`Target` construct

In order to address a wider variety of computer devices, a support for accelerators has been added since OpenMP 4.0. The `target` directive can be used to specify a portion of code which can be executed either on the host or offloaded to a device. The OpenMP specification initially supported only one kind of accelerator device, but support for multiple accelerators will be added in OpenMP 4.1 (e.g., CPU + GPU + Intel MIC[7]). There is also quite a few clauses to specify how and when the data needed for a target region should be transfered to the device, which can have an important impact on performances, as transfer costs to external devices are usually very high.

For all work-sharing construct, the programmer can specify how the data should be handled in each threads, this is known as data-sharing information. Some common data-sharing clauses which can be applied on variables are the following :

- shared: the variable will be shared by all threads, thus allowing for data races on the variable.

[7] **MIC:** Many Integrated Core

- private: each thread will have its own uninitialized copy of the variable, which will be destroyed at the end of the construct.
- firstprivate: the same as above, but for each thread the variable is initialized to the value it had when the construct was created.

When inside a parallel region, synchronization constructs (e.g., barrier) coordinate threads in a team and data accesses. The specification requires all threads of a team executing a parallel region to execute a barrier or none at all. An implicit barrier is called at the end of a `parallel`, `sections`, `single`, `workshare` regions and at the end of a loop construct unless a `nowait` clause is specified. Thus the OpenMP specification forces all threads of a team to encounter the same sequence of worksharing constructs.

1.1.8 Heterogeneous Architectures Programming

Heterogeneity is emerging in supercomputers. It is then important to understand how to program on these architectures (e.g., how to create an adherence between codes and devices). Most known and used models devoted to work on accelerators are presented below.

OpenACC [31] is a high-level implicit programming model that enables offloading of compute-intensive loops and code regions from a host CPU to an accelerator using simple compiler directives in C, C++ and Fortran. Programmers create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator. All of these are managed by the OpenACC API-enabled compilers and runtimes.

CUDA[8] [32] is a lower-level explicit programming model developed by NVIDIA. CUDA is an extension of the C programming language. OpenCL[9] [33] is similar to CUDA. It is a low-level API that runs on CUDA-powered GPUs.

1.1.9 Exascale Challenges

Why exascale? Even if powerful supercomputers are now available, we still face "Grand Challenges" problems [34]. A Grand Challenge problem is a problem that cannot be solved in a reasonable time with today's computers [6]. Scientific and engineering challenges in both simulation and data analysis already exceed petaflops. This is especially due to more complex scientific models with higher resolution, large-scale computation and lots of data. Exascale could facilitate more realistic and accurate simulations and also benefit to other fields such as economics and medecine. Studies [35–37] have highlighted this and give scientific challenges requiring exascale computing resources (e.g., modeling global climate change over long periods [38]).

This increase in the processor power of supercomputers presents many challenges for applications developers and programming models particularly concerning performance, correctness and portability. It also rises practical problems as a need to reduce the power consumption from supercomputers (the electric power would likely exceed a gigawatt).

Exascale programming models The increase of cores per node raises the issue of the right programming model to use for the future. The creation of new programming languages adapted for future machines can be considered. However rewrite applications from scratch is a laborious task and requires considerable time. Moreover developers can not be forced to start over with a new programming model when a new feature is needed. Thus any application and programming model should be prepared to evolve in order to run effectively on many generations of parallel computers. This is already happening as existing scientific

[8] **CUDA**: Compute Unified Device Architecture

[9] **OpenCL**: Open Computing Language

applications are modified to make them gradually hybrid (by mixing parallel programming models). This leads to consider interoperability of models.

It seems that using a shared memory model inside shared memory nodes and message passing across nodes tends to be a good fit to fully exploit the performance of future machines. E. Lusk and A. Chan report for instance some successful use cases of OpenMP threads exploiting multiple cores per node with MPI communicating among the nodes [39]. As most HPC applications are parallelized with MPI, combining MPI with a shared memory model is becoming a standard. **Figure 1.6** shows an example of a MPI+OpenMP execution model. In this figure, we suppose the machine has four-cores nodes. Each MPI process runs on a node and contains four OpenMP threads.

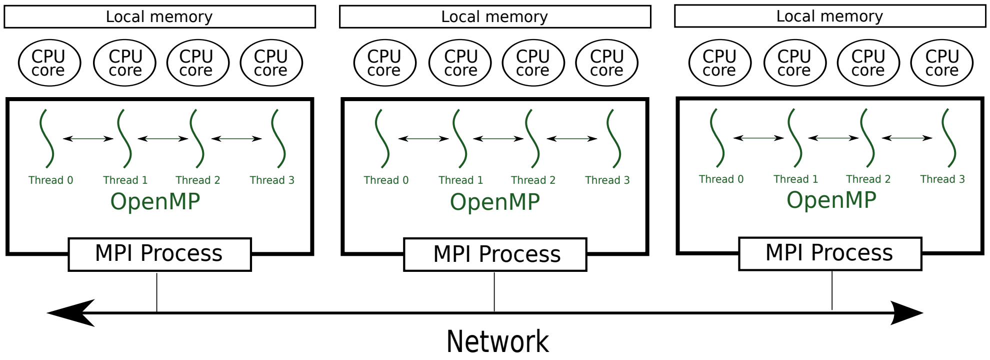


Figure 1.6: *Example of a hybrid model combining MPI and OpenMP programming models. In this example, each MPI task is executed on a node and contains four OpenMP threads.*

Interoperability of parallel programming models - Focus on MPI The adoption of MPI+X programming has made the MPI committee to produce solutions that enables full interoperability between MPI and system-level programming models (e.g., X10, Chapel, UPC) as well as node-level programming models (e.g., OpenMP, threads, TBB). The MPI-2 standard has progressed in this direction by defining degrees of thread support. The first thread support `MPI_THREAD_SINGLE` specifies MPI calls are performed outside threaded regions. The second thread support, `MPI_THREAD_FUNNELED` ensures only the main thread makes MPI calls while the `MPI_THREAD_SERIALIZED` level ensures multiple threads may make MPI calls but only one at a time. The highest thread support `MPI_THREAD_MULTIPLE` enables multiple threads to call MPI, with no restriction. This thread support should be used with caution to respect thread safety within processes. Mixing parallel programming models does not facilitate the debugging phase.

1.1.10 Summary

This section summarizes parallel programming models used in HPC and provides a picture of what are HPC challenges today. Regarding these models, supercomputers evolution questions applications duration as well as models interoperability. The reducing amount of memory per compute core tends more to mix parallel programming models instead of using one model to avoid a complete rewrite of applications. This makes the debugging process of an application harder. Besides, using a supercomputer is only relevant if results obtained are correct. It does not matter how fast the application can run if the application is erroneous. That is why it is crucial to provide effective debugging tools to help developers especially for hybrid applications. But as set out further, very few debugging tools exist to debug hybrid applications (non-reproducibility of experiments execution, more problems in hybrid programs). The next section paints a state of the art about approaches to detect errors in parallel applications for the purpose of solving the debugging exascale challenge. In the scope of this thesis we target MPI and OpenMP models as they are widely used in the HPC world. We then will focus on MPI, OpenMP and MPI+OpenMP applications. Problems adressed here can also be found when using other parallel solutions like MPI+Pthreads or PGAS.

1.2 Debugging Parallel Constructs of HPC applications

The previous section introduced supercomputer evolution and related challenges. In this section we take interest in the area of parallel constructs error detection in parallel applications.

1.2.1 Software Life-Cycle Models

Software life-cycle models define the different activities in the software development, their relative order and their relations in the software development process. The software life-cycle typically includes a requirement phase, design phase, implementation phase, testing phase and an installation and maintenance phase. There exist a wide variety of life-cycle models with strengths and weaknesses. We generally distinguish three types of models: linear, cyclic and multi-cyclic models. Among linear models, we can mention waterfall models, transformational and V-form models. Among cyclic models we find prototyping, spiral, incremental and evolutionary models [40]. The Waterfall model, the spiral model and the V-model are part of the main life-cycle models and are described below. It shall be noted that our descriptions of software life-cycle models will remain brief as the purpose here is to solely give an overview in order to provide a sufficient context of the work done in this thesis.

Waterfall Model

The Waterfall model, first introduced by Royce in 1970 [41], is the most common approach. It can be seen as a chronological sequence of activities, progressing downwards (like a waterfall). The model as described by Boehm is presented **Figure 1.7** [40]. It contains five main activities: Requirements, Design, Implementation, Testing and Maintenance. In this model, an activity cannot be done if the previous one has not successfully completed. This model works well for projects that are relatively simple.

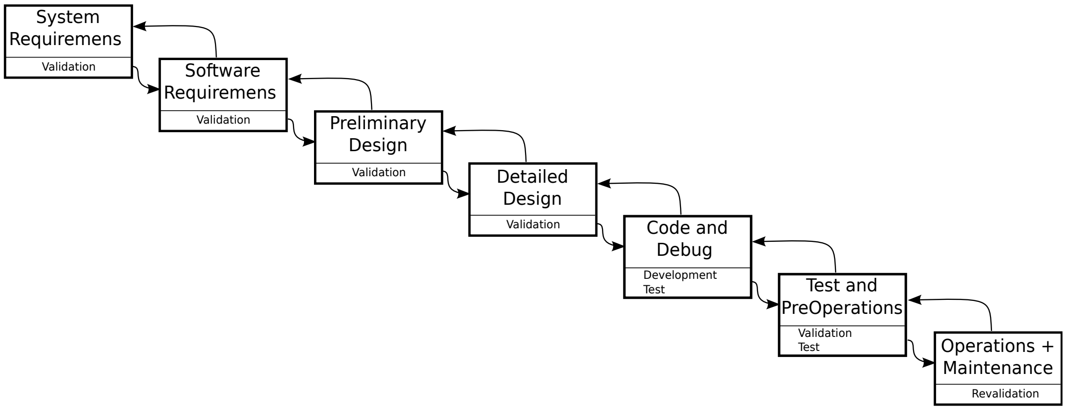


Figure 1.7: *Boehm's version of the Waterfall Model*

V-Model

The V-Model can be seen as a derivation of the waterfall model. Despite its V shape, like presented **Figure 1.8**, the model is still a sequence of consecutive phases. The V form delimits early analyzing and design steps from the integration and verification steps.

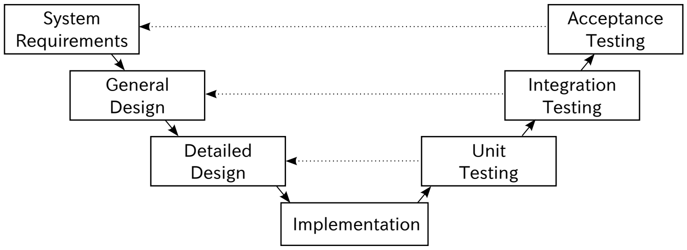


Figure 1.8: *Example of V-Model development cycle*

Spiral Model

The spiral model combines elements of various models and particularly the waterfall and prototyping[10] models. It is composed of four phases: Planning, Risk Analysis, Engineering and Evaluation. These phases are repeatedly passed in cycles [42]. The model is described **Figure 1.9**.

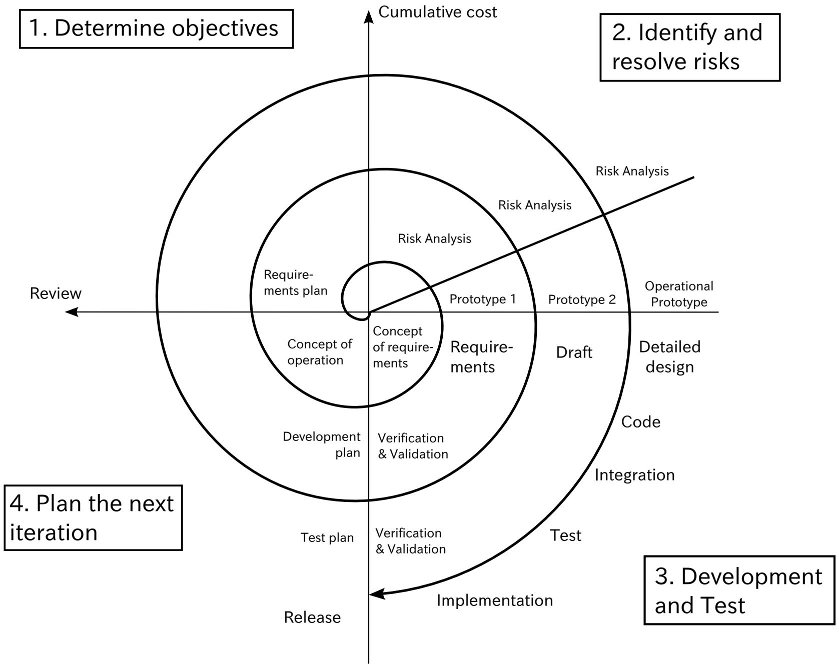


Figure 1.9: *Example of Spiral development cycle model*

Testing and Debugging

In the scope of this thesis we are interested in the testing phase of the software life-cycle. The testing phase consists in the verification and the validation of a program. Verification is used to prove the program correctness according to its specification and validation checks the program execution for a limited sets of inputs. The goal of the testing phase is correctness or reliability. Boehm gives different types of automated aids for the testing phase, between the point at which the code has been written and the point at which it is pronounced acceptable [43]:

- Static Code Analysis

[10] **The software prototyping Model** relies on creating and testing prototypes of software applications. An initial prototype is tested and reviewed. Then the feedback is used to improve the prototype.

- Test Case Preparation
- Test Monitoring and Output Checking
- Fault isolation, debugging
- Retesting (once a presumed fix has been made)
- Integration of routines into systems
- Stopping

Static code analysis gathers information without requiring program execution. It includes usual compiler diagnostics and data-type checking. Moreover control flow and reachability analysis are applied with structural analysis programs. As an extension of structural analysis, test case preparation provides assistance in choosing data values to make the program execute along a desired path. This tactic succeeds on simple cases and only helps generate inputs as the expected outputs have to be manually calculated. Various kinds of dynamic data-type checking and assertion checking, and for timing and performance analysis have been developed to enable automatic testing. Debugging is probably the most important part of testing as programmers generally spend a lot of time in it. Furthermore, a program is useless if it is incorrect. Debugging, as defined in the ANSI/IEEE standard glossary of software engineering terms [6], is the process of locating, analyzing, and correcting suspected errors.

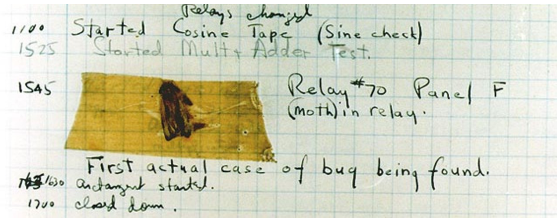
Once the code is fixed, retesting is used to check the differences in codes, inputs and outputs between corrected codes and previous test cases. Even far from complete criteria for determining when to stop testing, tools exist to quantify the amount of testing performed.

1.2.2 Debugging Parallel Applications

Developers generally spend a lot of time to debug an application. This section aims at understanding the debugging phase of the application development cycle in order to help selecting best tools and best practices to reduce this time.

History of Debugging

It is said that the term "bug" was first used in 1945. On September 9, 1945, Grace Hopper was working on Mark II calculator at the Computation Laboratory of Harvard University when the machine abruptly stopped for no apparent reason. She and her Harvard technical team discovered the computer glitch was due to a moth. The moth was removed from the machine and Grace added the caption "First actual case of bug being found" to her manual logbook (see **Figure 1.10**). Since, the term "bug" was popularized to signify any system malfunction [44].



1100 Started Cosine Tape (Sine check)
Relays changed
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Figure 1.10: *First computer "bug"*

Definitions

Applied to programs, a bug can mean an incorrect program code or an incorrect program state or an incorrect program execution. This lack of precision can cause confusions and has lead to the use of more precise terms. This section aims at summarizing existing terms and definitions used in the debugging field.

An informal definition of the basic term "error" for computer science is given below.

Definition 1. *An error is defined as the computation (calculation and outputting) of one or more incorrect results by a computer.*

The ambiguity of the term *bug* has lead
Zeller defines the terms *defect*, *infection* and *failure* in [45]. His definitions are given below.

Definition 2. *A defect is an incorrect program code.*

Definition 3. *An infection denotes an incorrect program state.*

Definition 4. *The term failure is used for an observable incorrect program behavior.*

The IEEE standards define a *fault* as Zeller defines a *defect* and a *bug* as the synonym of Zeller's definition of a *defect*. *Debugging* is thus the activity of removing defects in the software.

The terms *error* and *fault* are frequently used as a synonym of *infection* and for mistakes made by the programmer. *Failures* are called *issues* or *problems*.

In view of the above, we define a correct parallel program as follows:

Definition 5. *A correct parallel program is a program that terminates, complies to the semantics and the specification of the parallel model used and for all given inputs reacts with correct outputs.*

Debugging Cycle

The testing and debugging phase of the software life-cycle operates in two cycles represented by a flow-diagram presented **Figure 1.11**. The testing cycle (on the left in the figure) is used to detect incorrect computations and the debugging cycle (on the right in the figure) allows the user to get information about program states and intermediate results during execution. The debugging phase is only applied in case of the existence of an error to identify, locate and correct.

An error can be either a program failure or incorrect results. In case of a program failure, the program abruptly ends at an incorrect state while in case of incorrect results, the program ends correctly but outputs wrong results. This last case requires a check of output validity. In both cases, as the programmer does not know what he is looking for, the program error could be solved by getting back in time in order to find what caused it. Thus after an error is noticed, each step in reverse time order is checked. However this solution is in fact hard or even impossible to achieve. As a consequence the classic approach to debugging resorts to backtracking by re-executing a program with the illusion to get back in time. To that end, a breakpoint is added where the error is supposed to be in the program and this one is re-executed from the beginning until reaching the breakpoint. From the breakpoint, the user can get information needed to identify and correct the problem. If this is not the case, the user must find another possible location of the error and remake the procedure.

This style of debugging is called cyclical debugging [47]. The method is not efficient as it requires a lot of time especially in large programs.

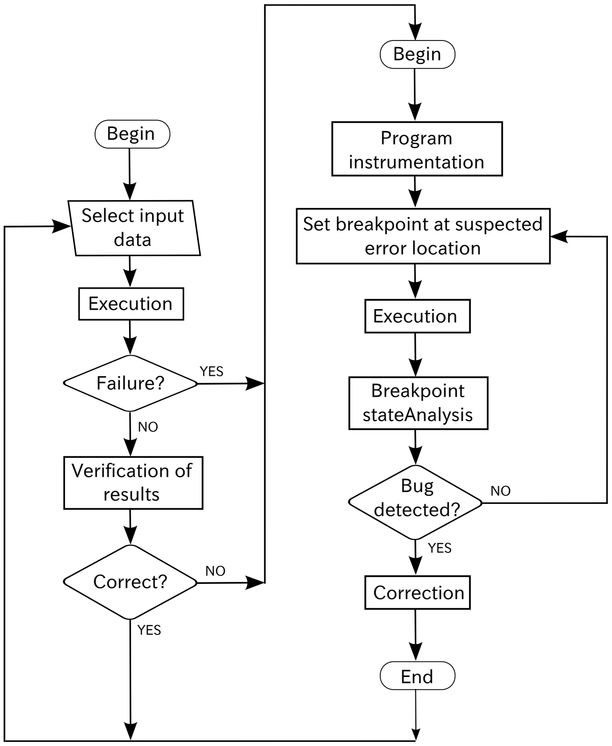


Figure 1.11: *Testing and debugging cycle [46]*

Once an error is reproducible, this process of bug-tracking is described by Zeller [45] in seven steps (which initial letters form the word `TRAFFIC`): Track (create an entry in the problem database), Reproduce the failure, Automate and simplify the test case, Find origins by following back the dependences from the failure to possible infection origins, Focus to possible origins, Isolate the origin of the infection and Correct the defect [45].

Several tools have been developed to assist users in the testing and debugging phases of sequential programs. Some users tried to use them in parallel programs but faced obstacles. These obstacles are discussed in the next section.

Sequential Debugging VS Parallel Debugging

In practice, a parallel program can be seen as several concurrent sequential programs which can communicate. In this way the difficulties encountered to debug sequential programs are the same in parallel programs. However parallel programs have specific characteristics that must be taken into consideration. For example, even run with the same inputs, parallel programs do not always have reproducible behavior and can be completely different from run to run. As a result, the cyclical debugging approach is not applicable on parallel programs (an undesirable behavior may not appear when the program would be reexecuted). In addition, the order of events occuring in distinct, concurrently processors may hardly be determined.
Parallel debugging then requires the rethinking of traditional debugging goals in the context of parallelism.

A first difference between sequential and parallel debugging is the amount of debugging data to be analyzed and the required storage. Debugging is based on program results and the states of the program during execution. In parallel programs, analyzing the results and the state in which parallel instances are is difficult. Another difference is that at large scale, anomalous effects can appear in parallel programs. Furthermore, synchronization and communication between concurrently executing tasks can increase the occurrence of unexpected event and results.

These difficulties have lead to the emergence of parallel debugging tools to help parallel application developers to fix errors.

Parallel Debugging Tools

A debugging tool is defined in [44] as follows:

Definition 6. *A debugging tool is anything that provides useful knowledge about a program execution and its occuring program state changes.*

Like regular debuggers, a parallel debugging tool has some essential prerequisites. First any debugging tool should take a reasonable time. Then as an error can occur much earlier in a program than the manifested effects, a debugging tool should provide truthful information to step the user in the right direction for correcting errors. In addition of sequential debugging tools prerequisites, a parallel debugging tool should be valid for any scale of parallel programs and support different programming models and hardware architectures. The necessity of parallel programs to exchange data and to synchronize parallel instances introduces effects that do not exist in sequential programs. Therefore, additional functionality for handling these communication effects is a basic requirement for parallel debugging tools.

Tactics of Debugging

In principle, there are two main activities that have to be carried out during debugging: (1) the detection of faulty behavior and (2) the location of code responsible for the faulty behavior. In order to manage these two activities, we can mention three tactics of debugging: (i) debuggers, (ii) static analysis and (iii) dynamic analysis (including post-mortem analysis) [48].

(i) Debuggers are the most common used tactic of debugging. They help tracking down, isolating and removing bugs. As we have seen previously, programmers generally start to debug their program from a faulty state and re-launch their program using a debugger in order to explore its state.

(ii) Static analysis examines a program without executing it. It enables the consideration of all possible states of a program and typically detects synchronization (e.g., deadlock) and data-usage errors (e.g., usual sequential data-usage errors, processes simultaneously updating a shared variable).

(iii) Contrary to static analysis, dynamic analysis observes the execution of a program. Dynamic analysis is limited to few input sets. The trace-based approach consists in storing events separately of the program in a tracefile. This method allows the analysis of the program from the beginning to its termination. Managing large tracefiles can be challenging as they grow rapidly with both event verbosity and the number of cores.

A combination of these tactics could improve the debugging phase as they have different purpose and cannot catch all errors.

The following sections describe existing tools to debug applications parallelized with the most used parallel models for distributed and shared memory. **Section 1.2.3** describes common MPI errors and related work on MPI debugging analysis, focusing on deadlocks due to collective operations. **Section 1.2.4** provides a summary of common OpenMP errors and existing debugging tools for OpenMP programs.

1.2.3 Verification of MPI Applications

As stated in **Section 1.1.4**, HPC applications are mainly parallelized with MPI. This section describes common MPI errors and existing tools to prevent and/or correct them. In the MPI debugging tools section we focus on deadlocks caused by MPI collective operations.

Classification of Common MPI Errors

In [49], authors give a number of properties that any MPI program should satisfy. Among these properties, we can mention that *the program [should be] input-output deterministic* and *the program contains no unnecessary barriers*. A classification of common MPI errors has been made by authors in [48]. In this paper, MPI errors are grouped into six categories:

- Deadlocks
- Data races
- Memory
- Mismatches
- Resource handling
- Portability

Deadlocks are generally caused by the non occurrence of something (mismatched send/recv operations or collective calls). The program stays blocked in an infinite waiting state preventing the program to terminate. We make a distinction between two types of deadlock: real deadlock, which corresponds to deadlock that necessarily occurs and potential deadlock, which occurs under certain circumstances. Code 1 **Figure 1.12** depicts a case of real deadlock. In this example, both processes of rank 0 and 1 perform a `MPI_Recv` before a `MPI_Send`. Data races can be caused by various reasons like the use of a receive call with the wildcard `MPI_ANY_SOURCE` as source argument as presented in Code 2. In this code, an error may occur depending on the order of the receives made by the process with rank 0. Memory contains all incorrect codes involving memory as a code where a memory still in use is reused. For example, Code 3 shows a code where `MPI_Alloc` is not followed by a `MPI_Free_mem`. Allocated datatypes, communicators, requests, etc should always be freed. Mismatches addresses calls with wrong type or number of arguments. Code 4 presents a datatype mismatch: `MPI_Type_contiguous`[11] is called on NULL datatype. This results in an error. Another example is to match a send `MPI_INT` with a receive `MPI_DOUBLE`. Resource handling corresponds to all incorrect construction, usage and destruction of MPI resources (e.g., communicators, groups). Finally, in the portability category are all decisions made by implementors.

MPI Debugging tools

Related work on MPI code verification can be organized in five main categories: (i) debuggers, (ii) static analyses, (iii) online dynamic analyses, (iv) special MPI libraries, and (v) trace-based dynamic analyses.

MPI Debuggers Although it has no support for MPI, it is possible to attach the well-known debugger gdb [50] to each MPI process of an application. The mpirun command that runs MPI processes under the control of gdb is called mpigdb. The same can be done with Valgrind. Debugging MPI applications with mpigdb succeeds with a limited number of MPI processes. More convenient parallel debuggers like TotalView [51] and DDT [52] provide usual functionality of debuggers but also allows the user to monitor and act on groups of processes in a single debugging session [48]. TotalView is a debugger from RogueWave software, Inc [51]. It can be used to debug both serial and parallel programs. It works with C, C++ and Fortran applications, supports several HPC platforms and can handle multiple types of HPC parallel coding

[11] **MPI_Type_contiguous:** is the simplest datatype constructor which allows a replication of a datatype into contiguous location [7]

Code 1

```
 ...
  // Communications between process 0 and 1
  if(rank==0){
    MPI_Recv(&Recv_buf, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status[0]);
    MPI_Send(&Send_buf, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  }else if(rank==1){
    MPI_Recv(&Recv_buf, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status[1]);
    MPI_Send(&Send_buf, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
 ...
```

Code 2

```
...
if(rank==0){
     MPI_Recv(&Recv_buf, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status
         [0]);
     MPI_Recv(&Recv_buf, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status[1]);
}
else if(rank==1)
     MPI_Send(&Send_buf, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
else if(rank==2)
     MPI_Send(&Send_buf, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
...
```

Code 3

```
char *ap;
...

MPI_Alloc_mem(count, MPI_INFO_NULL, &
    ap);
for(j=0; j<count; j++) {
    ap[j] = (char)(j & 0x7f);
}
...
```

Code 4

```
...
MPI_Datatype typeNULL, type;
typeNULL = MPI_DATATYPE_NULL;
...
MPI_Type_contiguous(1, typeNULL, &
    type);
...
MPI_Type_free(&type);
...
```

Figure 1.12: *Examples of erroneous MPI codes.*

(MPI, OpenMP, UPC and GA, OpenACC and CUDA). TotalView provides key features like a graphical data visualization and a memory leaks and malloc errors debugging. The Allinea DDT debugger [52] also supports various HPC platforms and models. This tool has shown that it could work at petascale.

Static tools This class of tools is mainly based on model checking and requires symbolic program execution, at the expense of combinatorial number of schedules or reachable states to consider, making this approach challenging. TASS [53], a successor of MPI-SPIN [54] follows this approach: using model checking and symbolic execution, it checks numerous program properties explicitly annotated with pragmas. If a property is violated (such as an incorrect order of collective calls) by exploring reachable states of the model built, an explicit counter-example is returned to the user in the form of a step-by-step trace through the program showing the values of variables at each state of the synthesized model.

MPI-Checker [55] is a static analysis checker for MPI codes that uses Clang[12]'s Static Analyzer [56]. It enables AST (e.g., unmatched point-to-point calls, unreachable calls, type mismatch, invalid argument type, collective call in rank branch) and path-sensitive (e.g., double request usage of nonblocking calls, missing wait) checks. The analysis emits bug reports only if an invariant is surely violated but is limited to C programs.

More focused on send-receive matching and dataflow analysis, [57] presents a compiler analysis framework that extends traditional dataflow analyses to message passing applications. This framework is defined for a communication model with unbounded number of processes that communicate exclusively via `send` and deterministic `receive` operations. Authors define a parallel control flow graph (pCFG) and dataflow equations to analyze applications. pCFG is an extension of CFG by representing all possible control-flow states and state transitions that may be performed by multiple sets of processes. For example, it represents possible send-receive matching and rules for dataflow propagation. Currently intra-procedural, the framework could be extended to support context-sensitive interprocedural dataflow analyses. Authors suggest to compute a function summary for every pCFG node where some process set performs a function call.

Online dynamic tools Dealing with dynamic tools, we can mention DAMPI [49, 58], Marmot [59, 60], Umpire [60, 61], MPI-CHECK [60, 62] and MUST [63, 64].

Umpire, Marmot and MUST rely on a dynamic analysis of MPI calls instrumented through the MPI profiling interface (PMPI). They are able to detect mismatching collectives either with a timeout approach (DAMPI, Marmot and MPI-CHECK) or with a scheduling validation (Umpire and MUST). Methods performing deadlock detections through a timeout approach are known to produce false positives, for example in case of abnormal latencies. DAMPI uses a scalable algorithm based on Lamport Clocks (vector clocks focused on call order) to capture possible non deterministic matches. For each MPI collective operation, participating processes update their clock, based on operation semantics. Umpire, limited to shared memory platforms, relies on dependency graphs with additional arcs for collective operations to detect deadlocks. In Marmot, an additional MPI process performs a global analysis of function calls and communication patterns. Both of these approaches, however, have limited scalability, forwarding MPI call information to a central manager for collective correctness. MUST overcomes such limitation by relying on a tree-based layout [65, 66]. Finally, MPI-CHECK [62] instruments the source code at compile time adding extra arguments to MPI calls. The resulting instrumented program is then compiled and produces an instrumented executable which outputs errors and warnings upon execution.

MPI libraries Validation can also be done inside MPI libraries or as an extension of a library (as for MPICH for instance or NEC-MPI), allowing collective verification for the full MPI-2 standard [67–70]. The detection of runtime deadlock causes is however limited to the information available to the MPI routines.

Trace-based dynamic tools Intel® Message Checker (IMC) [60, 71] collects all MPI-related information in trace files and performs the post-mortem analysis of these traces. This tends to be difficult and with limited scalability due to the trace sizes, correlated to the number of cores and the execution time of the application. IMC was recently replaced by Intel® Trace Analyzer and Intel® Trace Collector (ITAC) [72] which are part of Intel Cluster Tools (ICT). Intel® Trace Analyzer analyzes event trace data generated by the Intel® Trace Collector to detect performance problems and programming errors.

Summary A comparative study of all tools mentioned above is presented **Table 1.2**. The table categorizes tools according to their debugging tactic.

[12] **Clang**: a C language family front-end for LLVM compiler

Table 1.2: *Classification of MPI Debugging Tools. T: Trace-based, OD: Online dynamic analysis, L: MPI library*

Tool	Debugger	Static Analysis	Dynamic Analysis
DDT	x		
mpigdb	x		
MPI-SPIN		x	
TASS		x	
MPI-Checker		x	
Umpire			OD
Marmot			OD
MUST			OD
MPI-CHECK			OD
DAMPI			OD
NEC-MPI			L
MPICH			L
IMC			T
ITAC			T

1.2.4 Verification of OpenMP Applications

Node-level parallelization is becoming increasingly important with the emergence of multi- and manycore architectures. OpenMP has emerged as the most widely used standard for shared memory parallel programming. Furthermore, efforts to establish error handling capabilities in the OpenMP standard encouraged studies of common OpenMP errors to classify them. This section first exposes the classification of possible mistakes in OpenMP applications and then details the state of the art on existing debugging OpenMP tools.

Classification of Common OpenMP Errors

A study of most frequent errors made by students in OpenMP programs and best practices to adopt in order to avoid them was done in [73]. When this article was published, OpenMP 2.5 was the most recent standard specification. In this article, authors distinguish two main types of mistakes: *Correctness mistakes* and *Performance mistakes*. Correctness mistakes are errors impacting the correctness of a program (e.g., access to shared variables not protected) while performance mistakes are errors impacting the speed of a program (e.g., too much work inside a `critical` region).

A more detailed description of OpenMP usage errors that covers newer OpenMP constructs has been recently published in [74]. This work distinguishes syntactic and semantic programming errors (defect) and performance issues. The latter affects the efficiency of the application and are not necessarily specific to OpenMP. A syntactic error adresses code non compliant with the OpenMP grammar (e.g., mistyped directives). A semantic error is a programming mistake that can cause execution aborts, deadlocks or incorrect results. These errors can be divided in four subclasses: violation of the standard, conceptual defect, race condition and deadlock. **Figure 1.13** presents a visual classification of OpenMP errors. Violation of the OpenMP standard and conceptual defects are programming errors. Violation of the OpenMP standard includes worksharing constructs not encountered by all threads of a team (see Code 1 **Figure 1.14**) and invalid nesting of regions like a `single` region nested in a `single` region (see Code 2 **Figure 1.14**). These kind of prohibition can be detected by compilers. For example, the GCC compiler (version 4.9.1) returns the following warning for Code 2 **Figure 1.14**.

```
example2.c: In function "main":
example2.c:12:12: Warning : work-sharing region may not be closely nested
inside of work-sharing, critical, ordered, master or explicit task region
```

and the ICC compiler (14.0.3 20140422) returns the folowing error message:

```
example2.c(12): error: "single" region may not be closely nested inside a
                "single" region
                #pragma omp single
                        ^

compilation aborted for example2.c (code 2)
```

A conceptual defect is when a code does not explicitly violate the OpenMP standard but results in unwanted behavior. An example is to use a `parallel` directive instead of a `parallel for` (see Code 3 **Figure 1.14**).

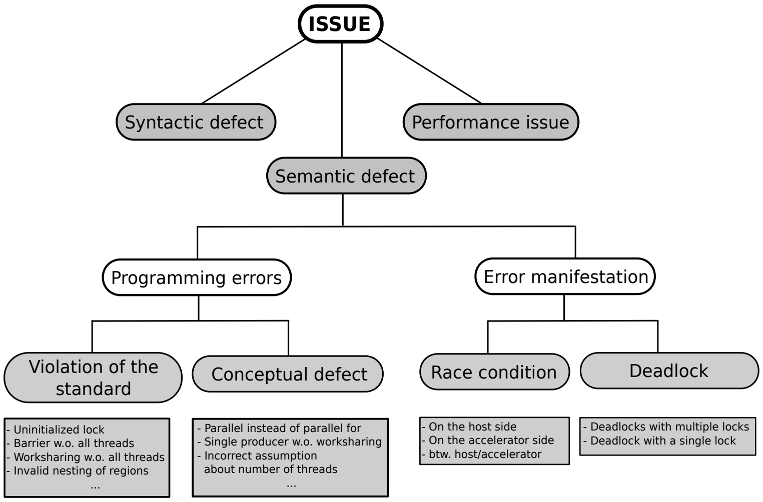

ISSUE
Syntactic defect
Performance issue
Semantic defect
Programming errors
Error manifestation
Violation of the standard
Conceptual defect
Race condition
Deadlock
- Uninitialized lock
- Barrier w.o. all threads
- Worksharing w.o. all threads
- Invalid nesting of regions
...
- Parallel instead of parallel for
- Single producer w.o. worksharing
- Incorrect assumption about number of threads
...
- On the host side
- On the accelerator side
- btw. host/accelerator
- Deadlocks with multiple locks
- Deadlock with a single lock

Figure 1.13: *Classification of Common Issues in OpenMP Applications [74]*

Code 1

```
#pragma omp parallel
{
  if(omp_get_num_threads()
      %2)
  {
    #pragma omp for
    for(int i=0; i<N; i++)
      ...
  }
}
```

Code 2

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp single
    {
      ...
    }
  }
}
```

Code 3

```
double a[N], b[N], c[N
    ];
...
#pragma omp parallel
  for(int i=0;i<N;i++)
     a[i]=b[i]*c[i];
```

Figure 1.14: *Examples of semantic defects (Violation of the standard and Conceptual defect).*

OpenMP Debugging Tools

OpenMP applications are prone to concurrency errors such as data races and deadlocks. Debugging tools generally check the correctness of OpenMP programs either at compile-time or during the execution of a program, both methods having advantages and inconveniences. This section summarizes some existing tools to detect data races and deadlocks in OpenMP applications.

Static tools The well-defined semantics of OpenMP makes static analyses common to check the correctness of OpenMP applications. Several static approaches exist: First we can mention the OpenMP Analysis Toolkit [75] (OAT) that uses symbolic analysis to detect concurrency errors. It relies on the ROSE compiler infrastructure to encode every parallel region into Satisfiability Modulo Theories (SMT) formulae. Those formulae are then solved with a SMT-solver like Yices [76]. OAT terminates its analysis by instrumenting the source code with fault injection techniques to confirm the reported errors. OmpVerify [77] is a static tool integrated in Eclipse IDE using the polyhedral model to detect data races in OpenMP parallel loops. This tool is restricted to program fragments called Affine Control Loops but it has the advantage of reporting accurate errors to the user. Lin [78] describes a concurrency analysis technique to detect whether two statements will not be executed concurrently by different threads in a team. The method is an intra-procedural analysis based on phase partioning using an OpenMP Control Flow Graph (OMPCFG) that models the transfer of control flow in an OpenMP program.

Similarly, Zhang *et al.* [79] use a concurrency analysis to detect unaligned barriers in OpenMP C programs. This inter-procedural method consists in four phases: A CFG construction to model the various OpenMP constructs, a barrier matching to find threads barriers that synchronize together, a program division into phases (sequence of basic blocks separated by barriers) and an aggregation of phases with matching barriers. Any two basic blocks from the same aggregated phase are said to be concurrent. For the verification of the whole program Zhang *et al.* export a barrier tree. Detection can also be done by compilers like GCC when lowering the OpenMP constructs to GOMP function calls [80] (e.g., GCC issues a warning for wrong nested parallelism as seen earlier).

Dynamic tools Among dynamic tools we can mention the Adaptative Dynamic Analysis Tool [81] (ADAT) and RaceStand [82, 83] for focused data races detection and Intel Thread Checker [84, 85] and Sun Thread Analyzer [86] for both data races and deadlocks detection. ADAT is a data-race detection tool using classification and adaptation mechanisms. The tool creates a pseudo-instrumented source code and an Engine Code Property Selector (ECPS) table and then transforms the pseudo-instrumented source code into an executable by using the ECPS table information. With a C compiler supporting OpenMP, the instrumented source code is compiled and executed to detect data races. RaceStand by GNU utilizes an on-the-fly dynamic monitoring approach to detect data races and has recently improved its check with a dynamic binary instrumentation technique based on Pin software framework. This tool detects the existence of races and locates races between two accesses not causally preceded by other accesses also involved in races (first races) for each shared variable in a program. Intel® Thread Checker and Sun Thread Analyzer both require an application instrumentation and trace references to memory and synchronization operations during the application execution. Sun Thread Analyzer necessitates program recompilation with the Sun compilers. To find data races the program must be executed with two or more threads. Unlike Sun Thread Analyzer, Intel Thread Checker does not depend on the number of threads used. It dynamically detects data races using a projection technology which exploits relaxed OpenMP programs. More precisely, the projection technology checks the data dependency of accesses to shared variables using sequentially traced information. But Intel Thread Checker does not consider OpenMP program specifications and can therefore report false positives. Li *et al.* present in [87] an online-offline model to test the correctness of every OpenMP parallel region. The online correctness testing model is used to find parallel regions with incorrect execution results (not corresponding to serial execution results), identify all places that caused errors (directives used improperly or located wrongly) and correct them. Then the offline correctness testing model tests the correctness of regions with corrected directives.

1.2.5 Verification of MPI+OpenMP Applications

Even if it is now possible to profile and visualize profiles and traces (for instance with Jumpshot [88]) for MPI+OpenMP programs, debugging tools especially those detecting collective operations errors and thread levels compliance are practically non-existent. Among profiling and tracing tools we can mention Intel® Thread Checker [85], Totalview [89], TAU [90], Scalasca [91], Vampir [92], DDT [93] and ompP [94]. Although it is possible to debug a program with a trace file, the post-mortem approach does not allow early errors detection. Indeed an error is noticed at the same time it occurs. Only a debugging tool could help to detect errors and find what caused them. The ISP dynamic debugger for MPI programs has been adapted to verify a large hybrid MPI/Pthread program. It uses a "record/replay" mechanism inspired by Output Deterministic Replay (ODR) [95]. To our knowledge, Marmot [96] is the only tool that provides a support for detecting collective errors in MPI+OpenMP programs. Marmot uses the MPI profiling interface (PMPI) to introduce artificial data races only occuring when some constraints are violated and detect them with the Intel Thread Checker tool. The authors define five constraints based on the definition of the thread levels mentioned in the MPI standard. These constraints imply violations, including simultaneous multiple collective calls on the same communicator (referred to constraint IV). Marmot generates HTML reports witch output error messages with the line of the collective and the nesting level between MPI and OpenMP the programmer should use. Marmot has been included in both the DDT debugger and the CUBE visualization tool. To our knowledge, Marmot [96] is the only tool that provides a support for detecting violations in MPI+OpenMP programs.

1.3 Outline

This chapter helps understanding the context of this thesis. It sums up existing parallel programming models, exascale challenges and the importance of the debugging phase of an application development cycle.

Debugging summarizes the activities of error detection and performance analysis of the software lifecycle. Both activities can be tedious and difficult, especially in parallel programs. The state of the art about debugging solutions for MPI, OpenMP and MPI+OpenMP programs has shown a lack concerning precise and early detection of errors in parallel applications. Indeed, existing debugging tools whether static or dynamic are able to detect the line in the source code where an error occured but rarely the line responsible for this situation. Although the compile-time offers the possibility to detect and correct errors earlier than at runtime, few tools rely on purely static analysis because of the combinatory aspect of methods used like model checking (e.g., TASS for MPI verification). Dynamic tools detect an error when the corresponding effects occur and are limited to the input dataset of a run. Indeed, even if dynamic tools return no false positive, they can miss errors as they are correlated to one execution of a program (e.g., one input set).

The chapter shows how hybrid debugging tools are crucial for the Exascale era. Indeed, the MPI+OpenMP approach is one solution to tackle the increasing node-level parallelism and the decreasing amount of memory per compute unit. But writing hybrid programs is driven by the availability of hybrid debugging tools and there is a complete lack in helping developers to find errors in hybrid programs.

This thesis investigates the combination of static and dynamic analysis to enable a scalable detection of control flow anomaly in parallel programs. More precisely, the two-step analysis provides an early and precise detection of collectives errors origins in MPI, OpenMP and MPI+OpenMP parallel programs. The contributions are:

1. The first contribution checks the compliance of MPI+OpenMP applications. We designed a novel approach in which the compiler is a valuable partner in solving problems occuring in parallel applications. The idea was to analyze applications at compile-time to verify the compliance of hybrid applications with the MPI thread level provided. Then only potential non-compliant portion of codes are instrumented to verify the non-compliance at runtime. We validate the analysis on computational benchmarks and applications showing a small impact on performance and the ease integration of our techniques in the development process. This contribution enables us to realize the interest of an error benchmark suite to verify the good detection of errors in parallel applications. (**Chapter 2**);

2. The second contribution adapts the previous analysis to detect the origin of *collective errors* in MPI and OpenMP applications. This contribution is based on the fact that MPI collective communications and OpenMP barriers and worksharing constructs have similar constraints. All MPI processes (resp. OpenMP threads of a team) have to call the same sequence of collective communications (resp. the same sequence of barriers and worksharing constructs). (**Chapter 3**);

3. The last contribution extends the combining approach to detect the origin of MPI *collective errors* in MPI+OpenMP applications. The idea is to verify the sequence of MPI collective communications of all MPI processes and to detect concurrent calls within a process. To this end we highlight the multithreaded context in which MPI collective operations are called. (**Chapter 4**).

These contributions have been published and presented in the European MPI Users' Group Meeting (EuroMPI) 2013 [97], the International Journal of High Performance Computing Applications (IJHPCA) 2014 [98], the International Workshop on OpenMP (IWOMP) 2014 [99], the Symposium on Principles and Practice of Parallel Programming (PPoPP) 2015 [100], the European Conference on Parallel and Distributed Computing (EuroPar) 2015 [101] and the European MPI Users' Group Meeting (EuroMPI) 2015 (to appear).

Chapter 2

Interaction Between MPI and shared memory models

As stated **Section 1.1.4**, MPI is one of the most widely used model in HPC applications. However as we progress from petascale to exascale systems, MPI evolves to be mixed with shared-memory approaches like OpenMP to efficiently exploit future systems. The adoption of MPI+X raises two crucial issues: the interoperability of models and how to assist developers to write conform hybrid applications. Mixing models means one model feature can be used in the context of the other. To avoid an interference between models, a context of use must be defined. Up to now debugging tools were mainly focused on one programming model and one step of the application development cycle (mostly compilation time or execution time). However a combination of compile-time and runtime informations could give more precise and efficient analyses.

Based on this fact, this chapter introduces a two-step method which aims at helping application developers to check which model interaction support is required for a specific hybrid code. The method is simple to use, compliant to hybrid MPI+X[1] programs and compatible with existing dynamic tools. The chapter focuses on MPI+OpenMP applications and reproduces a published article [101].

2.1 MPI Thread-Level Checking for MPI+OpenMP Applications

To address the challenges of exascale systems, MPI+OpenMP applications are becoming the norm in high performance computing. In such cases, special care is required for MPI calls to ensure the multi-threaded model does not interfere with MPI. As an example, within a process, the same communicator may not be concurrently used by two different MPI collective calls. This means MPI collective operations may not be called by multiple parallel threads. For instance, the function `MPI_Allreduce` Example 1 **Figure 2.1** will be called by each thread created in the parallel region (line 2). In that case, the sequence of MPI collective calls is not deterministic as it should be. The MPI-2 standard defines four thread-safety levels to indicate how MPI should interact with threads. These levels are from the most restrictive to the less restrictive levels: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE`. According to the MPI standard, *it is the user responsibility to prevent races when threads within the same application post conflicting communication calls* ([7], page 482 lines 45-46). This should be checked above all for the fully multithreaded case (`MPI_THREAD_MULTIPLE`).

Figure 2.1 illustrates some of the possible issues related to MPI communications in a multithreaded context through six examples written in C. `MPI_Send` in Example 2 is called outside the threaded region. This example is compliant to the `MPI_THREAD_SINGLE` level if the function `f` is not called in a parallel construct. `MPI_Allreduce` in Example 3 is called in a `single` block, `MPI_THREAD_SERIALIZED`

[1] X should be a shared-memory approach with perfect nested parallelism like OpenMP

Example 1

```
1  void f(){
2   #pragma omp parallel
3   {
4     MPI_Allreduce(..)
5   }
6  }
```

Example 2

```
1  void f(){
2
3   #pragma omp parallel
4   {
5     /*...*/
6   }
7
8   MPI_Send(..)
9  }
```

Example 3

```
1  void f(){
2   #pragma omp parallel
3   {
4    /***/
5    #pragma omp single
6    {
7     MPI_Allreduce(..)
8    }
9   }
10 }
```

Example 4

```
1  void f(){
2   #pragma omp parallel
3   {
4    #pragma omp single \
5          nowait
6    {
7     MPI_Reduce(..)
8    }
9    /***/
10   #pragma omp single
11   {
12    MPI_Reduce(..)
13   }
14  }
15 }
```

Example 5

```
1  void f(){
2   #pragma omp parallel
3   {
4    #pragma omp single
5    {
6     MPI_Reduce(..)
7    }
8    /***/
9    #pragma omp single
10   {
11    MPI_Reduce(..)
12   }
13  }
14 }
```

Example 6

```
1  void f(){
2   /***/
3   if(...){
4    #pragma omp parallel
5    {
6     /***/
7     #pragma omp master
8     {
9      MPI_Recv(..)
10     MPI_Send(..)
11    }
12   }
13  }
14  /***/
15 }
```

Figure 2.1: *MPI+OpenMP examples showing different uses of MPI calls.*

then corresponds to the minimum level of compliance. However if the function f is called itself in a parallel construct, the collective is executed in a nested parallel region, possibly leading to more than one concurrent call to this collective. This erroneous situation always occurs unless only one thread is created in the first parallel region or in both regions. Example 4 illustrates a more complex case: two MPI_Reduces are executed in single constructs in the same OpenMP parallel region. As the first construct contains a nowait clause, both MPI_Reduces can be executed concurrently by different threads. This requires a thread-level equal to MPI_THREAD_MULTIPLE, assuming the communicators used by the two collectives are different. If they are identical, the code is incorrect. On the contrary, in Example 5, the two MPI_Reduces are called one after the other because of the implicit thread synchronization at the end of the first single region (line 7). This piece of code requires a minimum thread-level equal to MPI_THREAD_SERIALIZED. In Example 6, function f is compliant with the MPI_THREAD_FUNNELED level. However, if the master directive line 7 is replaced by a single directive, the MPI_THREAD_SERIALIZED level is the minimum thread-level required. Thus, these examples illustrate the difficulty for a developer to ensure that MPI calls are correctly placed inside an hybrid MPI+OpenMP application whatever the required thread-level support.

2.1.1 Analysis of the Multithreaded Context

This subsection details the static analysis that helps the application developer to check which thread-level support is required for a specific code. The analysis proposed does not depend on one particular run and finds all possible situations of non-compliance to a given thread level. As it is conservative, it can be complemented by an instrumentation phase that checks the occurrence of these situations. An essential part of this analysis consists in determining the multi-threaded context in which MPI calls (Point-to-point and Collectives) are performed. The method described in this section computes a parallelism word to characterize this context in each point of the function analyzed. Programs are supposed to be SPMD (Single Program Multiple Data) MPI programs. It means that every MPI rank calls the same functions in the same order. This covers a large amount of scientific simulation applications for High-Performance Computing.

The Augmented Control Flow Graph

The analysis operates on the code represented as a *control-flow graph* (CFG) [102, 103]. The *control-flow graph* is an intermediate representation built in almost all compilers. It models the control flow of a function and is defined as a directed graph (V, E) where V represents the set of basic blocks[2] and E is the set of edges. Each edge $u \longrightarrow v \in E$ depicts a potential flow of control from node u to v. Each node in V has a set of successors and a set of predecessors denoted as $SUCC(u)$ and $PRED(u)$. Moreover two unique artificial nodes are appended for entry and exit points. Each function has one entry but can have several returns. To ensure one entry and one exit per function, all returns are bring together into one node. An example of a *control-flow graph* and its associated C code (`example.c`) is presented **Figure 2.2**. The code `example.c.013t.cfg` is obtained by the command `gcc -fdump-tree-cfg example.c -o example`. It shows basic blocks built by the GCC compiler. This CFG contains four nodes: the first one represents the variable c affectation and the `if` statement evaluation while the second and third ones respectively contain the `if` body and the `else` body. Finally the last one denotes the `return` instruction. Statements in node 2 are executed one after the other in a sequential order. Depending of the value of the conditional statement at the end of node 2, the variable c equals $a - b$ (node 3) or $b - a$ (node 4). The execution of the conditional results in a choice which gives rise to two paths: node 2 ⟶ node 3 and node 2 ⟶ node 4. From nodes 3 and 4, paths converge in node 5.

example.c

```c
int f( int a, int b )
{
    int c=0;

    if(a>b)
        c=a-b;
    else
        c=b-a;

    return c;
}
```

example.c.013t.cfg

```
<bb 2>:
  c = 0;
  if (a > b)
    goto <bb 3>;
  else
    goto <bb 4>;

<bb 3>:
  c = a - b;
  goto <bb 5>;

<bb 4>:
  c = b - a;

<bb 5>:
  D.1600 = c;
  return D.1600;
```

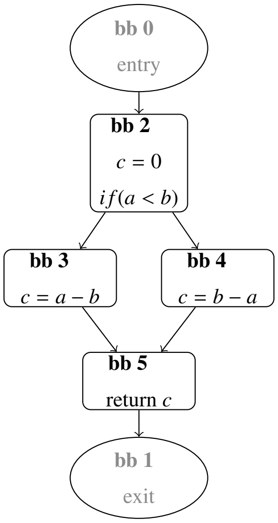


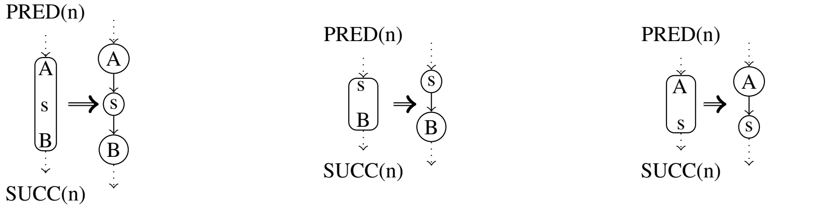
Figure 2.2: *Example of a simple C code with its associated CFG.*

[2] **Basic block**: Maximal sequence of linear code

The compile-time verification then consists in a static analysis of the CFG for each function of a program. The CFG is augmented to highlight nodes containing MPI calls (collectives and P2P) and as for some compilers like GCC, OpenMP directives are put into separate basic blocks. Hence new nodes are added for explicit and implicit thread barriers. For sake of clarity in figures, implicit thread barriers at the end of parallel regions are denoted by `end parallel`. **Algorithm 1** details how the CFG is augmented. The algorithm takes as input the CFG of the current processed function built by a compiler and returns the augmented CFG G^+. When a node with a MPI call is encountered, the node is tagged (line 6 in the algorithm). When a node contains an OpenMP construct, the node is split as shown **Figure 2.3** (line 9 in the algorithm). If the OpenMP directive is the first statement or the last statement of the node, the node is split into two nodes and three nodes otherwise. The sets of nodes and edges are up to date as well as the successors and predecessors of each node. Assuming the splitting phase takes time $O(T)$, the augmented CFG construction takes time $O(T.|V|)$.

Algorithm 1 Building the augmented CFG G^+

```
 1: function CFG+_CONSTRUCTION($G = (V, E)$)                        ▷ $G$: CFG
 2:     $V^+ \leftarrow V$ , $E^+ \leftarrow E$
 3:     for each $n \in V$ do
 4:         for each statement $s \in n$ do
 5:             if $s$ is a MPI call then
 6:                 Tag $n$                                  ▷ $n$ contains a MPI call
 7:             end if
 8:             if $s$ is a OpenMP construct then    ▷ includes explicit and implicit barriers
 9:                 Split $n$ into $n_1, n_2$ and $n_3$ such as $n_2$ only contains the statement $s$
10:                 Tag $n_2$                          ▷ $n_2$ contains an OpenMP directive
11:                 Up to date $V^+$ and $E^+$:
12:                     $V^+ \leftarrow V^+ - n$
13:                     $V^+ \leftarrow V^+ \cup \{n_1, n_2, n_3\}$
14:                     $E^+ \leftarrow E^+ - \{PRED(n) \rightarrow n, n \rightarrow SUCC(n)\}$
15:                     $E^+ \leftarrow E^+ \cup \{PRED(n) \rightarrow n_1, n_1 \rightarrow n_2, n_2 \rightarrow n_3, n_3 \rightarrow SUCC(n)\}$
16:                 Up to date successors and predecessors of $n_1, n_2, n_3$ and each node in $SUCC(n)$ and $PRED(n)$
17:             end if
18:         end for
19:     end for
20:     return $G^+ = (V^+, E^+)$
21: end function
```



$n_1 = A, n_2 = s, n_3 = B$ $n_1 = \emptyset, n_2 = s, n_3 = B$ $n_1 = A, n_2 = s, n_3 = \emptyset$

Figure 2.3: *Different splitting phases according to the origin node. The statement s is supposed to be an OpenMP construct or an implicit or explicit barrier. The origin node is presented on the left and the result of the splitting phase on the right.*

Parallelism Words Construction

To highlight the thread context in which a MPI call is performed, we define a *parallelism word* for a node in G^+ as the sequence of OpenMP parallel constructs (pragma `parallel`, `single`, ...) surrounding this block and the barriers traversed from the beginning of a function to the block. Parallel regions containing the block are denoted by P^i, with i the id of the basic block with the OpenMP construct (CFG+_CONSTRUCTION **Algorithm 1** ensures there is only one OpenMP directive per node) . Similarly, regions executed by the master thread are denoted by M^i and other single threaded regions are denoted S^i. Finally, `barrier` corresponds to B. OpenMP defines a perfectly-nested parallelism, thus the control flow has no impact on the parallelism word.

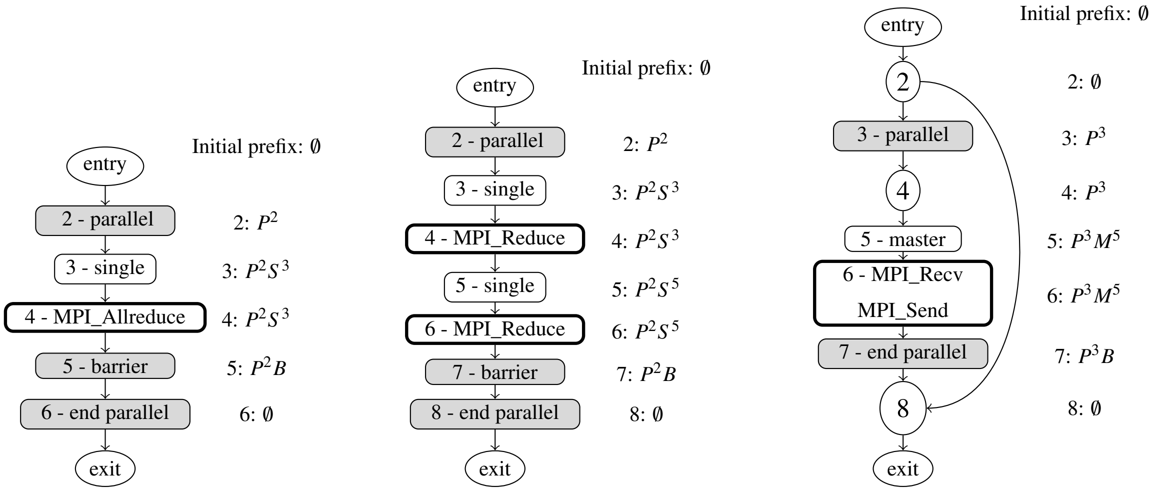
Algorithm 2 Building the parallelism word for all CFG nodes

```
function DEPTH_FIRST_SEARCH($G^+ = (V^+, E^+), n$)        ▷ $G^+$: CFG augmented
    Tag $n$
    SET_PARALLELISM_WORD($n$)
    for $u \in SUCC(n)$ do
        if $u$ is not tagged then
            $pw[u] = pw[n]$
            DEPTH_FIRST_SEARCH($G^+, u$)
        end if
    end for
end function
```

Algorithm 3 Parallelism word construction of a node

```
function SET_PARALLELISM_WORD($n$)        ▷ $n$: a node of the CFG
    switch $n$
      case #pragma omp parallel
        push(P,$pw[n]$)
        break
      case #pragma omp barrier or implicit barrier
        push(B,$pw[n]$)
        break
      case #pragma omp single/section/task
        push(S,$pw[n]$)
        break
      case #pragma omp master
        push(M,$pw[n]$)
        break
      case end of parallel region
        while head($pw[n]$)≠P pop($pw[n]$) end while
        pop($pw[n]$)
        break
      case end of single/master/section/task region
        pop($pw[n]$)
        break
      default
        break
    end switch
end function
```

Algorithm 2 presents the depth-first search that traverses the entire graph to construct the parallelism word of each node of the augmented CFG. Each node n is associated to a parallelism word denoted $pw[n]$. The *Depth_First_Search* function starts the depth-first search with the unique successor of the entry node and explores each branch of the CFG as far as possible. Each node sets its parallelism word depending on its predecessor and the OpenMP directives it contains. **Algorithm 3** presents the parallelism word construction of a node: P is added when a parallel region is encountered, S is added when a single, section or task region is traversed, M is added when a master construct is traversed and B is added when an implicit or explicit thread barrier is met. A simplification is done when OpenMP regions end. The *Depth_First_Search* procedure is linear in the size of G^+ and takes time $O(|E^+|)$ and the parallelism word construction in a node takes time $O(1)$. The entire parallelism word construction for a G^+ then takes time $O(|E^+|)$. **Figure 2.4** shows examples of CFG with their associated parallelism words. Functions presented are supposed to be called in a sequential thread context, the initial parallelism word at the function entry is then empty.



(a) CFG of Example 3 (b) CFG of Example 4 (c) CFG of Example 6

Figure 2.4: *Augmented Control Flow Graph and parallelism words of codes in **Figure 2.1***

Parallelism Words Analysis

The automaton **Figure 2.5** defines the possible parallelism words. Nestings forbidden by the OpenMP specification (SS, MS,...) are not considered by the automaton. If such forbidden nested regions are obtained, our analysis returns the error message: *invalid state, error*. The language of accepted parallelism words will depend on the specified thread level. As we check each function independently, the level of parallelism in which a function is called is unknown. To provide an accurate picture of the level of thread parallelism in which function occurrence is called, statistics on the NAS Parallel Benchmarks multizone (NASPB-MZ) using class B [104] have been collected and are shown in **Table 2.1** per thread, in each process. We notice that functions are mainly called within one level of multithreading.

Thus to consider all possible initial conditions, each callsite is instrumented in order to capture the initial parallelism word of each function. This word corresponds to a prefix P_i for all basic blocks of the called function and defines an initial state in Automaton **Figure 2.5** (all states are possible initial states). The user can choose the initial state at compile-time.

2.1.2 Thread-Level Compliance Checking

This subsection describes how the non-compliance of thread levels can be detected at compile-time. For that purpose we use the parallelism words introduced in the previous subsection to check the placement of MPI calls within a process.

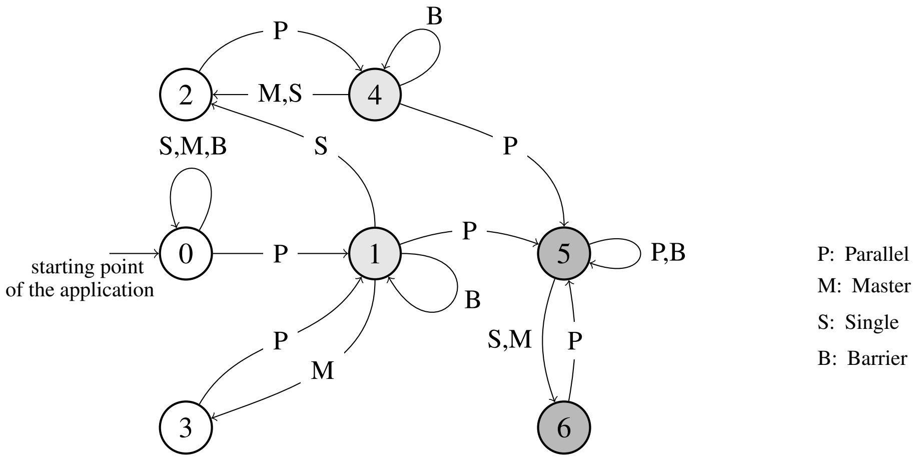


Figure 2.5: *Automaton of possible parallelism words. Nodes 0, 2 and 3 correspond to code executed by the master thread or a single thread. Nodes 1 and 4 correspond to code executed in a parallel region, and 5 and 6 to code executed in nested parallel region.*

Table 2.1: *Level of threads parallelism at function entries for NASPB-MZ*

Benchmark	**# function calls**	**# calls in sequential (state 0,2,3)**	**# calls in parallel (state 1,4)**	**# calls in nested (state 5,6)**
BT-MZ	396,918,403	45,379	396,873,024	0
SP-MZ	15,479,425	116,161	15,363,264	0
LU-MZ	3,017,513	40,745	2,976,768	0

Static Analysis and Interface to Dynamic Checkings

For each possible thread level we define a language of valid parallelism words based on the automaton **Figure 2.5**. For a given basic block, its parallelism word consists in the prefix (obtained from the callsite of the function or user-defined) and the word computed from previous analysis. The analysis verifies if the nodes containing MPI calls (point-to-point and collectives) are associated with an accepted word. Thread barriers can be safely ignored as they do not influence the level of thread parallelism. In case of the detection of a possible error, a warning related to the initial level with the name of the call is returned to the programmer. **Algorithm 4** takes as input the CFG augmented and the language L of correct parallelism words and outputs two sets: S and S_{ipw}. These sets respectively contain the nodes violating the input language and the nodes that dominate these nodes before the execution/control flow changes (see appendice B.1 page 101 for the definition of domination). This set will be given as one of the input parameters of the dynamic analysis. In the algorithm, line 7, the node u corresponds to the node preceeding n in the CFG and that is the immediate successor of a control flow node (with two successors) or of a pragma node (changing the parallelism word). Assuming the dominators before execution/control flow changement of each node are known, this algorithm takes time $O(|V^+|)$ in the worst case.

The nodes in the set S_{ipw} correspond to execution points where compliance should be tested at runtime, in order to handle possible false-positives detected statically. A unique parallelism word pw_e is computed at runtime and updated after each OpenMP construct. Compared to the compile-time parallelism words, parallel regions created with only one thread correspond to the parallelism word ϵ. This implies that such region has no impact on the current multithreaded context.

Algorithm 4 Detection of *parallelism words* for multithreaded regions

```
function Multithreaded_regions($G^+ = (V^+, E^+), L$)        ▷ $G^+$: augmented CFG, $L$: accepted language
    $S_{ipw} \leftarrow \emptyset$
    $S \leftarrow \emptyset$
    for each $n \in V^+ | n$ contains a MPI call do
        if $pw[n] \notin L$ then
            $S \leftarrow S \cup \{n\}$
            $u \leftarrow$ Node that dominates $n$ before execution/control flow changement
            $S_{ipw} \leftarrow S_{ipw} \cup u$
        end if
    end for
    Output $S_{ipw}$ and nodes in $S$ as warnings
end function
```

MPI_THREAD_SINGLE

By setting the MPI_THREAD_SINGLE level, the user ensures only one thread will execute MPI calls ([7], page 486 line 1). This means all MPI calls should be performed outside multi-threaded regions. Thus all nodes of the CFG containing a MPI call must be associated with an empty parallelism word. The language L of accepted parallelism words is then defined by $L = \{\epsilon\}$. **Algorithm 4** with $L = \{\epsilon\}$ returns the non-compliant MPI calls (set S). **Algorithm 5** details the entire compliance verification.

Algorithm 5 Verification of the MPI_THREAD_SINGLE level compliance

```
function SINGLE_Verification($G^+, L$)                        ▷ $G^+$: augmented CFG, $L = \{\epsilon\}$
    Depth_First_Search($G^+, SUCC(entry)$)          ▷ Building the parallelism word for all CFG nodes
    Multithreaded_regions($G^+, L$)          ▷ Detection of parallelism words for multithreaded regions
end function
```

MPI_THREAD_FUNNELED

The use of MPI_THREAD_FUNNELED level means the process may be multi-threaded but the application must ensure that only the thread that initialized MPI can make MPI calls ([7], page 486 lines 3-5). For this level, State 3 in Automaton **Figure 2.5** is the accepting state and the language $L = (PB^*M)^+$ describes the accepted words. With **Algorithm 4** and L, our analysis detects MPI calls that are not executed in a master region. **Algorithm 6** details the entire compliance verification.

Algorithm 6 Verification of the MPI_THREAD_FUNNELED level compliance

```
function FUNNELED_Verification($G^+, L$)                    ▷ $G^+$: augmented CFG, $L = (PB^*M)^+$
    Depth_First_Search($G^+, SUCC(entry)$)          ▷ Building the parallelism word for all CFG nodes
    Multithreaded_regions($G^+, L$)          ▷ Detection of parallelism words for multithreaded regions
end function
```

MPI_THREAD_SERIALIZED

The MPI_THREAD_SERIALIZED level means the process may be multi-threaded but only one thread at a time can perform MPI calls ([7], 12.4.3). The accepting states in Automaton **Figure 2.5** are states 2 and 3. Thus, the language $L = (PB^*S|PB^*M)^*$ describes the accepted words. This language contains parallelism words ending by S or M without a repeated sequence of P. Critical sections and locks are not supported here.

To verify the compliance of this level, **Algorithm 4** is used to make sure all MPI calls are performed in a monothreaded context. Different MPI calls in the same monothreaded region are sequentially performed as only one thread executes it. However, calls in different monothreaded regions may be called simultaneously if monothreaded regions are executed in parallel (no thread synchronization between monothreaded regions). Special care is requested for MPI collective operations. All MPI processes should execute the same sequence of MPI collective operations in a deterministic way. That means there is a total order between MPI collective calls. **Algorithm 7** shows the detection of concurrent calls. It takes as input the CFG and outputs two sets: S and S_{cc}. When nodes containing a MPI call with the same number of B are detected these nodes are put in the set S and the nodes that begin the monothreaded regions are put in the set S_{cc} for the dynamic analysis. A warning is then issued for nodes in S. **Algorithm 8** details the entire compliance verification.

Algorithm 7 Detection of potential concurrent calls

```
function Concurrent_calls($G^+ = (V^+, E^+)$)                                ▷ $G^+$: CFG
    $S_{cc} \leftarrow \emptyset$
    $S \leftarrow \emptyset$
    if $\exists\, u, v \in$ nodes in concurrent monothreaded regions then
        $S \leftarrow S \cup \{u, v\}$
        $i, j \leftarrow$ nodes immediate successors of nodes creating monothreaded regions
        $S_{cc} \leftarrow S_{cc} \cup \{i, j\}$
    end if
    Output $S_{cc}$ and nodes in $S$ as warnings
end function
```

Algorithm 8 Verification of the MPI_THREAD_SERIALIZED level compliance

```
function SERIALIZED_Verification($G^+, L$)           ▷ $G^+$: augmented CFG, $L = (PB^*S|PB^*M)^*$
    Depth_First_Search($G^+, SUCC(entry)$)          ▷ Building the parallelism word for all CFG nodes
    Multithreaded_regions($G^+, L$)                 ▷ Detection of parallelism words for multithreaded regions
    Concurrent_calls($G^+$)                         ▷ Detection of potential concurrent calls
end function
```

MPI_THREAD_MULTIPLE

This level is the least restrictive level. It enables multiple threads to call MPI with no restriction ([7], page 486 line 10). However MPI calls should be thread safe, meaning that when two concurrently running threads make MPI calls, the outcome will be as if the calls executed sequentially in some order. Special care is requested for MPI collective operations. Indeed it is harder to ensure thread safety with this type of communication. The verification of this level follows the same analyses as for the `MPI_THREAD_SERIALIZED` level. Both levels are subject to the same constraints. **Algorithm 9** details the entire compliance verification.

Selective Static Instrumentation

Previous sections detailed the static detection of possible MPI thread-level non-compliance. To dynamically verify the total order of MPI calls sequences in each MPI process, validation functions are inserted in nodes in the sets S_{ipw} and S_{cc} generated by **Algorithms 4 and 7**: CC_{ipw} and CC_{cc}. These functions are

Algorithm 9 Verification of the MPI_THREAD_MULTIPLE level compliance

```
function MULTIPLE_Verification($G^+$, $L$)      ▷ $G^+$: augmented CFG, $L = (PB^*S|PB^*M)^*$
    Depth_First_Search($G^+$, $SUCC(entry)$)     ▷ Building the parallelism word for all CFG nodes
    Multithreaded_regions($G^+$, $L$)     ▷ Detection of parallelism words for multithreaded regions
    Concurrent_calls($G^+$)     ▷ Detection of potential concurrent calls
end function
```

Algorithm 10 Library Functions To Check MPI calls

```
function $CC_{ipw}$(L)      ▷ Detect calls in multithreaded regions
    if $pw_e \notin L$ then      ▷ $pw_e$: execution parallelism word
        MPI_Abort($com$, 0)
    end if
end function

function $CC_{cc}$(L)      ▷ Detect concurrent calls
    $CC_{ipw}$(L)
    if $collective\_lock = 1$ then
        MPI_Abort($com$, 0)
    else
        #pragma omp atomic write
        $collective\_lock = 1$
    end if
end function
```

depicted **Algorithm 10**. Function CC_{ipw} detects incorrect execution parallelism words (pw_e) and Function CC_{cc} detects concurrent collective calls.

For each node n in S_{ipw}, if the corresponding execution parallelism words $pw_e[n]$ is not in L the program stops through a call to `MPI_Abort` and an error message is returned to the programmer. For each node n in S_{cc}, a check is done to ensure the node is actually in a monothreaded region. Counting the number of threads concurrently executing a given basic block cannot be done by the simple use of `omp_get_num_thread()`. Indeed, the control flow may select only a subset of the total number of threads for the execution of a basic block. Similarly to the previous case, we resort to a shared variable *collective_lock*. This variable is used to prevent another thread from entering the region in S_{cc}. The shared variable is reset to 0 right after the barrier(s) (if any) successor of the region concerned. The intuition is indeed to reset the lock at the first barrier following the possible concurrent monothreaded regions. Each function of a program is instrumented by **Algorithm 11**. If an error is about to occur the program is stopped and an error message is returned with error type information.

In **Figure 2.4(b)**, nodes 4 and 6 have the same number of thread barriers in their parallelism words (node 4: P^2S^3, node 6: P^2S^5) so the collective operations involved are potential concurrent collective calls. The algorithm outputs a warning for collective calls located nodes 4 and 6 ($S = \{4, 6\}$) and flags nodes 4 and 6 for dynamic checks ($S_{cc} = \{4, 6\}$). CC_{cc} functions are then inserted in nodes 4 and 6 as shown **Figure 2.6**. Suppose function f of example 3 **Figure 2.4** is called in a parallel construct. Node 4 has an incorrect parallelism word for a collective node. The algorithm outputs a warning for the collective that can be called by multiple processes: $S = \{4\}$ and $S_{ipw} = \{3\}$. Thus a CC_{ipw} function is inserted node 3.

Correctness Proof

The instrumentation depicted **Algorithm 11** is correct if all situations are captured and if the inserted code does not generate errors or deadlocks.

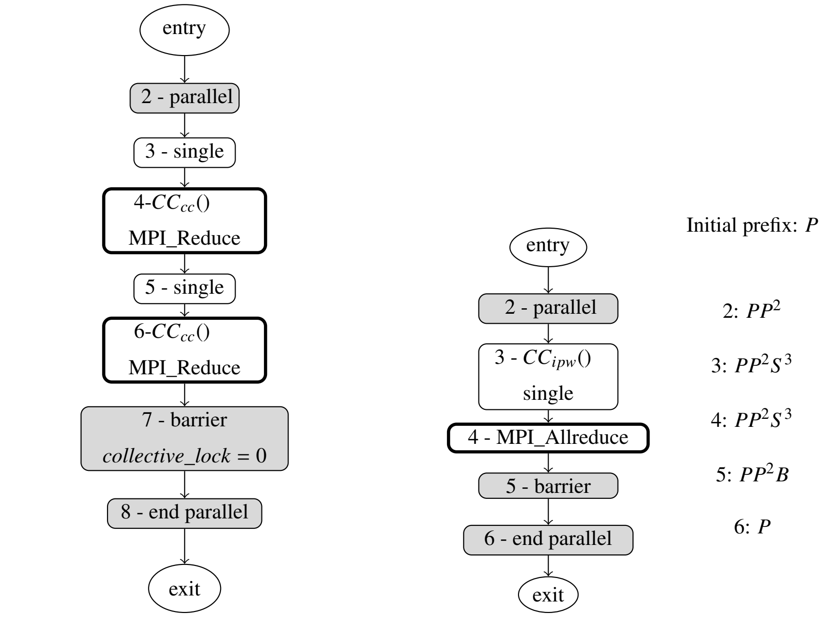
Algorithm 11 Selective Static Instrumentation

```
function INSTRUMENTATION($G^+, S, S_{ipw}, S_{cc}$)
                                   ▷ $G^+$: augmented CFG, $S, S_{ipw}, S_{cc}$: sets created at compile-time

    if $S \neq \emptyset$ then
        STEP 1: Parallelism words for multithreaded regions detection
        for $n \in S_{ipw}$ do
            Insert call to $CC_{ipw}()$ as the first statement of $n$
        end for

        STEP 2: Concurrent MPI calls detection
        for $n \in S_{cc}$ do
            Insert call to $CC_{cc}()$ as the first statement of $n$
            Insert $collective\_lock = 0$ after the barrier(s) successors of the region created by $n$
        end for
    end if

end function
```



(a) **Figure 2.4(b)** CFG instrumented

(b) **Figure 4.5(a)** CFG instrumented with an initial prefix P

Figure 2.6: *Instrumented CFG* ***Figures 2.4(b) and 4.5(a) (Algorithm 11)***

- If no potential error is reported from the compile-time analysis then no validation function is added. No error or deadlock is added in the code.

- Suppose the non-compliance of one thread-level is detected at compile-time. **Algorithm 11** then inserts CC_{cc} and CC_{ipw} functions. CC_{cc} and CC_{ipw} functions are inserted as soon as possible before MPI calls. All threads do not have to call CC_{cc} and CC_{ipw} functions, only one thread detects that an error is going to occur in a program. CC_{ipw} prevents from the execution of a call in a multithreaded context, *i.e.,* when multiple threads are about to execute the same operation simultaneously. For CC_{cc}, the second thread to execute CC_{cc} raises an error and avoids the unordered execution of different collective operations. As the *collective_lock* variable is only reset after a barrier following the parallel region where the check occurs, this detection does not depend on the thread scheduling. Whenever a thread enters a monothreaded region containing collective calls while another thread is already executing a monothreaded region also containing a collective call the thread detects an error and stops the program. In both cases the program is stopped before an error can occur.

We validate a thread-level given by the compiler or parsed at compile-time.

2.2 PARallel COntrol flow Anomaly CHecker (PARCOACH)

The two-step method checking MPI thread-level compliance in MPI+OpenMP applications was implemented in GCC 4.7.0 [105] as a plugin. It is simple to deploy in existing environments as it does not modify the whole compilation chain. The plugin called PARCOACH for PARallel COntrol flow Anomaly CHecker is located in the middle of the compilation chain where the source code is represented in an intermediate form (CFG). PARCOACH performs a new pass inserted inside the compiler pass manager, after generating the CFG information. The GNU Compiler Collection (GCC) was chosen because of its wide use but the method works with all compilers using a CFG representation. The analysis is language independent and thus enables the verification of C, C++ and Fortran programs. It is written in GIMPLE [106], a tree-address representation derived from GENERIC (see **Figure 2.7**). **Figure 2.8** gives an overview of PARCOACH.

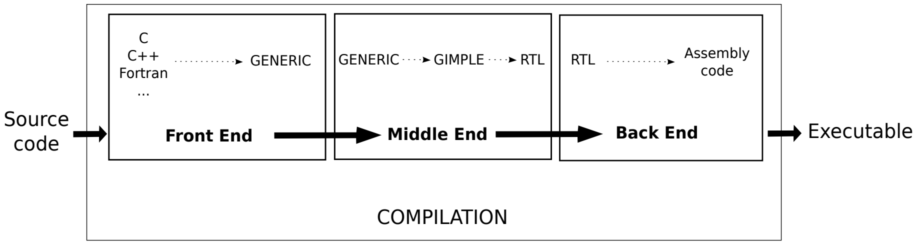


Figure 2.7: *Architecture of GCC*

To show the impact of PARCOACH on the compilation time, we present experimental results obtained on the NAS Parallel benchmarks multizone (NAS-MZ v3.2) using class B [104], five MPI+OpenMP Coral benchmarks [107] (AMG2013, LULESH, HACC, SNAP, miniFE) and a large multi-physics 2D/3D AMR hydrocode platform named HERA [108], which is a production test case.

Table 2.2 shows the language and the number of lines of each tested benchmark. The 4th and 5th columns depict the thread level provided (level actually returned to the user, might be lower than the desired level, depending on the MPI implementation) and the minimum thread level required by the application (thread-level the user should use). The last column displays the compliance our analysis returned. For each benchmark, the overhead obtained at compile-time is presented **Figure 2.9**. This overhead corresponds to the difference

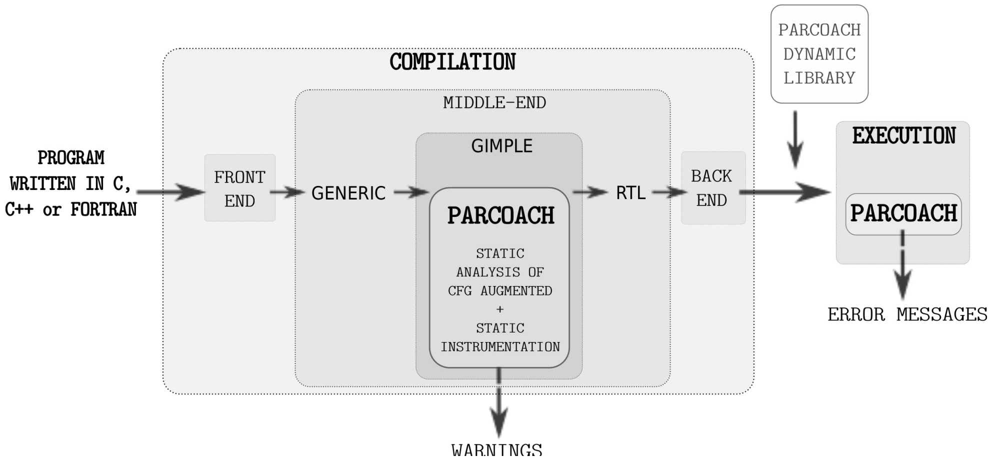


Figure 2.8: *Overview of PARCOACH*

Table 2.2: *Compliance Results*

Benchmark	Language	Lines of code	Thread level provided	Min thread level required	Compliant
BT-MZ	Fortran	6,779	SINGLE	SINGLE	yes
SP-MZ	Fortran	4,862	SINGLE	SINGLE	yes
LU-MZ	Fortran	6,542	SINGLE	SINGLE	yes
AMG2013	C	75,000	SINGLE	SINGLE	yes
LULESH	C	5,000	SINGLE	SINGLE	yes
miniFE	C++	50,000	SINGLE	SINGLE	yes
HACC	C++	35,000	SINGLE	SINGLE	yes
SNAP	Fortran	3,000	SINGLE	SINGLE	yes
HERA	C++	500,000	SERIALIZED	SERIALIZED	yes

between a basic serial compilation and a serial compilation with PARCOACH. The overhead obtained is acceptable as it does not exceed 6%.

The analysis issues warnings at compile-time with potential error information (lines of MPI calls, line where the dynamic check is inserted,...). With existing benchmarks, we can quantify the overhead of our analysis but we can't promote its functionality. As benchmarks tested are correct, PARCOACH did not find non-compliance of thread levels. That is why we created a microbenchmarks suite containing purposely non-compliance of MPI thread-levels to reveal PARCOACH MPI thread-levels compliance checking capability.

2.3 Revealing PARCOACH Functionalities

To assure the functionality of PARCOACH, we created a microbenchmarks suite containing MPI+OpenMP programs written in C and MPI thread-level non-compliant. This section shows results on four hybrid programs from this microbenchmarks suite (coll_single, coll_funneled, coll_serialized, p2p_multiple). The code of these benchmarks is depicted **Figures 2.10 and 2.11**.

Notice that the `MPI_THREAD_MULTIPLE` level was not supported by the MPI implementation we used. **Table 2.3** shows our analysis was able to find the thread-level non-compliance in the error microbenchmarks suite. All programs has been detected as in conformity with the standard.

The following example shows what a user can read on *stderr* when compiling the program *coll_serialized*

```
in function ’f’:
Warning: PARCOACH: possible non-compliance of MPI_THREAD_SERIALIZED level
```

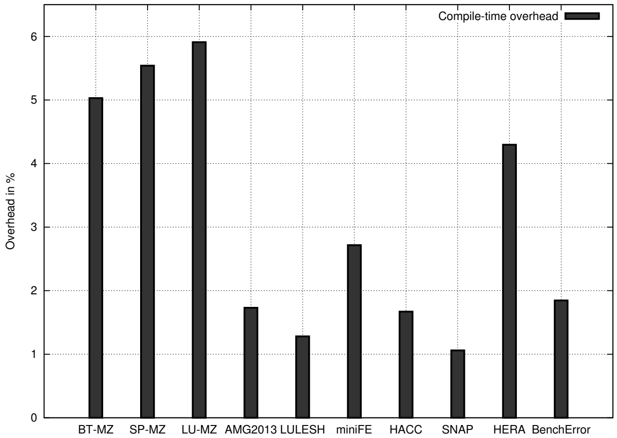


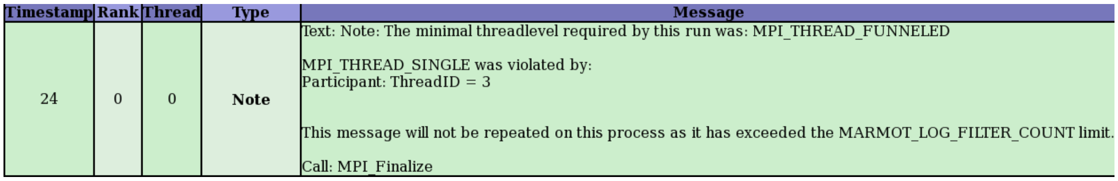
Figure 2.9: *Overhead of average compilation time*

Table 2.3: *Compliance Results*

Benchmark	Lines of code	Thread level provided	Thread level required	Compliant
coll_single	29	SINGLE	FUNNELED	no
coll_funneled	36	FUNNELED	SERIALIZED	no
coll_serialized	47	SERIALIZED	MULTIPLE	no
p2p_multiple	45	SERIALIZED	MULTIPLE	no

```
Potential concurrent MPI collective calls within a process : MPI_Reduce l.11
may be called simultaneously with MPI_Reduce l.6
PARCOACH: Minimum thread-level required: MPI_THREAD_MULTIPLE
PARCOACH inserted a check after the single directive l.4
PARCOACH inserted a check after the single directive l.9
```

In this example the `MPI_Reduces` are in fact called on different communicators. As PARCOACH does not check communicators both single regions are instrumented to check if the non-compliance of the thread level is confirmed at runtime. In comparison, the error message returned by the dynamic tool Marmot at runtime is the following:



Timestamp	Rank	Thread	Type	Message
24	0	0	**Note**	Text: Note: The minimal threadlevel required by this run was: MPI_THREAD_FUNNELED MPI_THREAD_SINGLE was violated by: Participant: ThreadID = 3 This message will not be repeated on this process as it has exceeded the MARMOT_LOG_FILTER_COUNT limit. Call: MPI_Finalize

Marmot finds that the code should be executed within the `MPI_THREAD_FUNNELED` thread level whereas PARCOACH finds the level `MPI_THREAD_MULTIPLE`. The reason comes from the fact that Marmot detects conformance w.r.t. one execution, and in particular to one parallel schedule. During the execution monitored by Marmot, the `SINGLE` constructs are executed by the master thread leading to a serialized sequence of these constructs. However, from a conformance point of view, this is not correct and the thread level `MPI_THREAD_MULTIPLE` as analyzed by PARCOACH should be chosen.

2.4 Summary

Supercomputers evolution encourages the development of hybrid applications. As most HPC applications are parallelized with MPI, the main solution adopted is to mix MPI with a shared memory model like

coll_single.c

```c
int main(int argc, char **argv)
{
   int provided, required;
   required=MPI_THREAD_SINGLE;

   MPI_Init_thread(&argc, &argv,
       required ,&provided);

   #pragma omp parallel
   {
      MPI_Barrier(MPI_COMM_WORLD);
   }

   MPI_Finalize();
   return 0;
}
```

coll_funneled.c

```c
int main(int argc, char **argv)
{
   int provided, required;
   required=MPI_THREAD_FUNNELED;

   MPI_Init_thread(&argc, &argv,
       required ,&provided);

   #pragma omp parallel
   {
      #pragma omp single
      {
         MPI_Barrier(MPI_COMM_WORLD);
      }
   }

   MPI_Finalize();
   return 0;
}
```

coll_serialized.c

```c
int main( int argc, char **argv)
{
     int rank,res=0,temp=0, provided, required;
     required=MPI_THREAD_SERIALIZED;

     MPI_Init_thread(&argc, &argv, required, &provided);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

     temp=rank;

     #pragma omp parallel
     {
          #pragma omp single nowait
          {
               MPI_Reduce(&temp,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
          }
          #pragma omp single
          {
               MPI_Reduce(&temp,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
          }
     }

     MPI_Finalize();
     return 0;
}
```

Figure 2.10: *Examples of MPI thread-level non-compliant codes.*

OpenMP. But this does not facilitate the debugging phase and raises the issue of models interoperability.

One of the MPI challenges is its interoperability with other programming models. Even if it is now possible to profile and visualize profiles and traces for MPI+OpenMP programs, debugging tools especially

p2p_multiple.c

```
int main( int argc, char **argv)
{
    int rank, provided, required, tag=1000, buff_size=1;
    MPI_Status status;
    int* buff_addr1= malloc(sizeof(int)*buff_size);
    int* buff_addr2= malloc(sizeof(int)*buff_size);
    required=MPI_THREAD_MULTIPLE;

    MPI_Init_thread(&argc, &argv, required, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    #pragma omp parallel
    {
        if(rank==0){
            if(omp_get_thread_num()==0){
              MPI_Recv(buff_addr1,buff_size,MPI_INT,1,0,MPI_COMM_WORLD,&status);
            }
            MPI_Send(buff_addr2,buff_size,MPI_INT,1,tag,MPI_COMM_WORLD);
        }
        if(rank==1){
            if(omp_get_thread_num()==0){
              MPI_Recv(buff_addr2,buff_size,MPI_INT,0,0,MPI_COMM_WORLD,&status);
            }
            MPI_Send(buff_addr1,buff_size,MPI_INT,0,tag,MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Figure 2.11: *Examples of MPI thread-level non-compliant codes.*

those detecting thread levels compliance are practically non-existent. To our knowledge, Marmot [96] is the only tool that provides a support for detecting violations in MPI+OpenMP programs. Marmot uses the MPI profiling interface (PMPI) to introduce artificial data races only occuring when some constraints are violated and detect them with the Intel Thread Checker tool. The authors define five restrictions for hybrid MPI applications based on the definition of the thread levels mentioned in the MPI standard. The fifth restriction is the non-violation to the provided thread level. However, as Marmot only relies on profiling, it may find for one run that the program is non compliant to a given thread level, and for another run find its compliance (so defining a compliance per run). The same happens for bugs, where detection may require many runs in a profile-only approach. On the contrary, PARCOACH finds statically the possible non-compliance of the code, pinpointing non-compliant code fragments and situations. The runtime instrumentation only checks whether these situations occur.

We have designed a static analysis followed by a static instrumentation of MPI+OpenMP programs to check which model interaction support is required for such an application. The analysis proposed finds the right MPI thread level to be used and identifies code fragments that may prevent conformance to a given level and presents a small impact on compilation-time with an overhead lower than 6%. Our method, named PARallel COntrol flow Anomaly CHecker, has been implemented in GCC as a plugin to avoid compiler recompilation. It supports applications written in C, C++ and Fortran and can be extended to any MPI+X applications with X a shared memory model with perfect nested parallelism.

PARCOACH was integrated in GCC but can be used in all compilers using a CFG representation. It was designed to be complementary to existing debugging tools. It then could be integrated in PMPI-based tools like Marmot or MUST to cover other errors like calls arguments (e.g., communicators) or to report warnings concerning the execution path responsible for bugs related to MPI thread-level compliance.

Many benchmarks exist to highlight the performance of debugging tools but there is no benchmark to demonstrate their efficiency and compare them in term of errors detection. This lack leads us to create a microbenchmarks suite containing wrong MPI+OpenMP programs. This error microbenchmarks suite will be used to validate all PARCOACH functionnalities.

This chapter aims at detecting a wrong MPI initialization in MPI+OpenMP applications. Verify the interaction between programming models is the first step to debug an hybrid application. Once the right interaction of models ensured, the correctness of each model can be checked separately. The next chapter suggests an adaptation of the method detecting MPI thread-level non-compliance to detect misuse of MPI and OpenMP collectives (MPI collective communications, OpenMP barriers and worksharing constructs) as soon as possible.

Chapter 3

Detection of Collective Errors Origin in Parallel Applications

When a bug occurs in an application it is generally difficult to identify what caused it. This is particularly true in parallel applications. **Sections 1.2.3 and 1.2.4** have provided a state of the art about debugging tools for MPI and OpenMP applications. We noticed a lack regarding the detection of errors as soon as possible. Thus in this chapter, we took an active interest in the identification of errors origin in the source code as well as in the development cycle of an application.

MPI and OpenMP models have *collectives constraints* meaning that tasks of the considered model have to encounter the same sequence of *collectives* in the same order. In MPI, *collectives* are blocking and non-blocking collective operations and in OpenMP, *collectives* are threads synchronizations and worksharing constructs. This chapter describes the extension of the method introduced in the previous chapter to detect the origin of collective errors in MPI and OpenMP applications. Its content has been published in articles [97–99] and is reproduced in this chapter.

3.1 Combining Static and Dynamic Analyses to Find the Origin of MPI Collective Errors

Nowadays most of scientific applications are parallelized based on MPI communications. The MPI standard requires that MPI collective communications have to be executed in the same order by all processes in their communicator and the same number of times, otherwise they do not conform to the standard and a deadlock or other undefined behavior can occur. As soon as the control flow involving these collective operations becomes more complex, in particular including conditionals on process ranks, ensuring the correction of such code is error-prone. In this section, we focus on detecting blocking and non-blocking MPI collective operations errors in Single Program Multiple Data applications, assuming MPI calls are not nested in multithreaded regions.

Based on the distributed-memory paradigm, this model exposes multiple ways to express communications between tasks/processes including point-to-point and collective. While point-to-point functions involve only two tasks, collective communications require that all processes in a communicator invoke the same operation. Each process does not have to statically invoke such collective function at the same line of the source code, but the sequences of collective calls in all MPI processes must be the same and corresponding function calls should have a compatible set of arguments. Due to the control flow inside an MPI program, processes may execute different execution paths. Such behavior may cause errors and deadlocks difficult for the user to detect and analyze.

A simple example

```
void f( int r ) {
  if( r == 0 )
    MPI_Barrier(
        MPI_COMM_WORLD);
  return;
}
void g( int r ) {
  f(r);
  MPI_Barrier(MPI_COMM_WORLD
      );
  exit(0);
}
```

The instrumented simple example

```
void f( int r ) {
   int res;
   if( r == 0 ) {
      MPI_Reduce(1,&res,1,MPI_INT,equalsop
          ,0,MPI_COMM_WORLD);
      if ( rank==0 && res == -1 )
        MPI_Abort(MPI_COMM_WORLD,0);
      MPI_Barrier(MPI_COMM_WORLD);
   }
   MPI_Reduce(0,&res,1,MPI_INT,equalsop,0,
       MPI_COMM_WORLD);
   if ( rank==0 && res == -1 )
     MPI_Abort(MPI_COMM_WORLD,0);
   return;
}
```

Figure 3.1: *A simple example and its instrumentation*

The following simple example (Listing 3.1) illustrates the potential issues with collective communications. Assume here that g is called by all processes. Depending on the value of the input parameter r, a process will execute or not the barrier in the if statement in f. If r is not uniformly true or false among MPI processes, some tasks will be blocked in f while the remaining process ranks will reach the barrier in g. These processes will then terminate, while the first ones will be in a deadlock situation at the barrier in g. The machine state when the deadlock occurs does not help to identify the cause of the deadlock. As the value of r is unknown at compile time and might be the same for every MPI process, the dynamic state of control flow has to be checked in order to prevent from entering a deadlock state. Transforming the previous example would lead to the code presented in Listing 3.2. Notice that the function g does not need to be transformed since it does not introduce a collective that may be the cause of a deadlock or an unspecified behavior. In order to partition processes according to their behavior regarding the conditional in f, two calls to the collective MPI_Reduce with the equalsop operation (bit equality checking) are inserted in the code: One before the barrier operation with the input value 1 (1^{st} parameter of the call), and one before the return statement with the input value 0. All processes call the MPI_Reduce collective, whatever their execution path. However, input values should be the same, otherwise the function is incorrect and MPI_Abort is issued in order to prevent from deadlocking. We consider only monothreaded MPI programs (the analysis also works on multithreaded programs if all MPI collectives are performed in monothreaded regions). In our context a function is said to be *correct* regarding collective communications if all MPI processes entering the function eventually exit without leaving any process blocked inside a collective operation or in a completion call for non-blocking operations.

While it is possible to match a blocking send (resp. nonblocking send) with a nonblocking receive (resp. blocking receive), it is not possible to do the same with collective operations. The MPI-3 API clearly states that *Nonblocking collective operations do not match with blocking collective operation [...]. All processes must call collective operation (blocking and nonblocking) in the same order per communicator* ([7], page 198 lines 1-4). For example, a call to a MPI_Barrier on some processes can not match with a MPI_Ibarrier on the remaining processes. They should encounter at some point a call to MPI_Barrier for the collective operation to be valid.

As our analysis is focused on the detection of mismatching collectives, other possible sources of deadlock (e.g., infinite loops, blocking IOs and other deadlocks) which would require dedicated analysis are not checked. While all types of blocking and non-blocking collectives are handled, collective operations are assumed to be called on the same communicators, with compatible arguments.

To prove an MPI program is correct, the method is decomposed into two phases: A compile-time verification and an execution-time verification. All source code functions are either proved statically correct, or potentially incorrect, depending on the control flow. Correct functions are filtered out and the code of the remaining functions is transformed to prevent deadlock situations. Only functions with collective calls that can deadlock are instrumented. This filtering approach avoids systematic instrumentation, thus reducing the overhead due to the dynamic analysis. When a deadlock situation occurs in a run, an error message is returned with information gathered at compile-time: The location and the type of the collective and the control-flow code responsible for this situation.

3.1.1 Compile-Time Verification

The principle of the proposed static analysis is to detect functions that have paths with different sequences of collectives (either not the same number or not the same collectives). When two such paths are found, the node responsible for this possible control flow divergence leading to deadlock is identified. First, the CFG is augmented as in **Algorithm 1, page 36** to only highlight *collective nodes* (nodes containing a collective operation). **Algorithm 12** details the compile-time analysis to detect if a function is correct. The algorithm takes as input the CFG of the current function and outputs nodes that may lead to collective errors and their collectives that may deadlock (set O). This set will be given as parameter for a code instrumentation. Note that the algorithm can handle any MPI collective operation, and based on our context, only the name of the collective is used in the algorithm.

Algorithm 12 Step 1 - Static Pass

```
function STATIC_PASS($G = (V, E)$)                                   ▹ $G$: CFG
    $O \leftarrow \emptyset$                                          ▹ Output set
    Remove loop backedges in $G$ to compute execution orders for nodes with collectives
    for $r$ in node orders do
        for $c$ in collective names of execution order $r$ do
            $C_{r,c} \leftarrow \{u \in V | r$ is the max. execution order of $u$, $u$ executes a collective with name $c\}$
            if $PDF^{+}(C_{r,c}) \neq \emptyset$ then
                $O \leftarrow O \cup (c, PDF^{+}(C_{r,c}))$
            end if
        end for
    end for
    Output nodes in $O$ as warnings and for Step 2.
end function
```

The main steps of the algorithm are the following: we compute for each node of the CFG the number of collectives on the execution paths from the function entry to the node. This number is 0 for nodes before the first collective (including the node with the first collective), 1 for nodes reached after one collective and so on. When multiple paths exist, nodes can have multiple numbers, at most the number of collectives in the function. Loop backedges are removed to have a finite numbering and the algorithm is applied to the CFG of each loop separately. For nodes with collectives, these numbers define an *execution order* between collectives within a function. Collectives with different numbers are executed sequentially while directives with the same number can be executed in parallel. These numbers are called in the following *execution orders*. If the same node has multiple execution orders, only the highest one is considered.

In a correct function (from a static point of view), for any given order k, all execution paths from *entry* to *exit* should traverse the nodes of order k with the same collective operation. Conversly a function is not correct if there are nodes with out-going paths traversing nodes of execution order k and other paths that do not traverse nodes of order k or with different collectives. It should be noted that whenever a function contains a collective call in a loop, it is considered as statically incorrect. The analysis does not account the number of iteration as this information is unknown at compile time. To address that, a data-flow analysis is needed.

Nodes corresponding to possible control-flow divergence leading to deadlocks can be computed using the iterated postdominance frontier [109]. A node u postdominates a node v if all paths from v to *exit* go through u. We extend this relation to sets: A set U postdominates a node v if all paths from v to *exit* go through at least one node of U. The postdominance frontier of a node u, $PDF(u)$ is the set of all nodes v such that u postdominates a successor of v but does not strictly postdominate v. If $\gg$ denotes the postdominance relation,

$$PDF(u) = \{v \mid \exists\, w \in SUCC(v), u \gg w \text{ and } u \not\gg v\}$$

In other words all paths from w to the exit node go through u. On the contrary v is not postdominated by u so there exists a path from v to exit node that does not traverse u.

This notion is extended to a set of nodes U,

$$PDF(U) = \{v \mid \exists\, w \in SUCC(v), \forall u \in U, u \gg w \text{ and } u \not\gg v\}.$$

All paths from w to the exit node go through a node u in the set U. On the contrary v is not postdominated by any node u in U so there exists a path from v to exit node that does not traverse a node in U. The iterated postdominance frontier PDF^+ is defined as the transitive closure of PDF, when considered as a relation [109].

Algorithm 12 describes this computation applied to each function and loop, entry and exit being then defined as loop entry and exit. The execution order computation corresponds to a simple traversal of the acyclic CFG, counting traversed nodes with collectives. Then for each execution order r, the nodes calling the same function c, at order r are clustered into $C_{r,c}$. The iterated postdominance frontier of this set corresponds to nodes that can lead both to the execution of such collective or not.

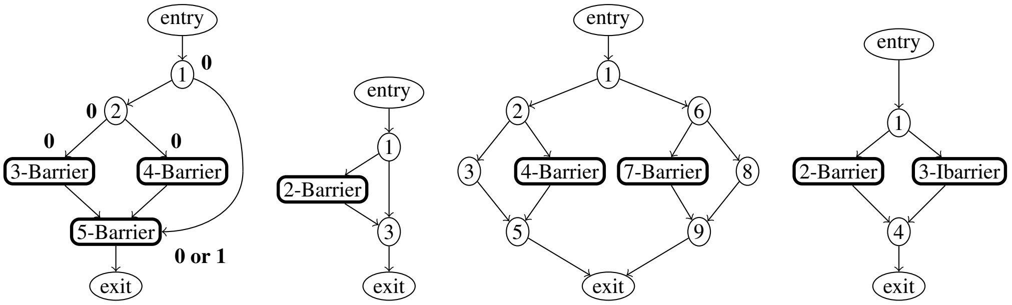
Lemma 1. ***Algorithm 12*** *detects all deadlock situations due to a collective operation.*

Proof. We prove that the algorithm computes a non-empty set O if and only if the function is incorrect, and nodes in O correspond exactly to the nodes that can lead to a deadlock.

Consider an element (c, S) of O, with c a collective and S the set $PDF^+(C_{r,c})$ for some order r. If u denotes a node from S, there is an outgoing path from u that goes through c of order r, and another path that reaches the exit node without going through a collective c of same order. If the second path never reaches a collective c (any order) and if both paths are executed by different tasks, then some tasks will wait at the collective c while the other tasks will either wait at another collective (a deadlock) or exit the function (incorrect function). In both cases, the function is incorrect. If both paths traverse the same collective c, since the orders are different, one of the paths has more collectives c than the other. Again, this leads to an incorrect function.

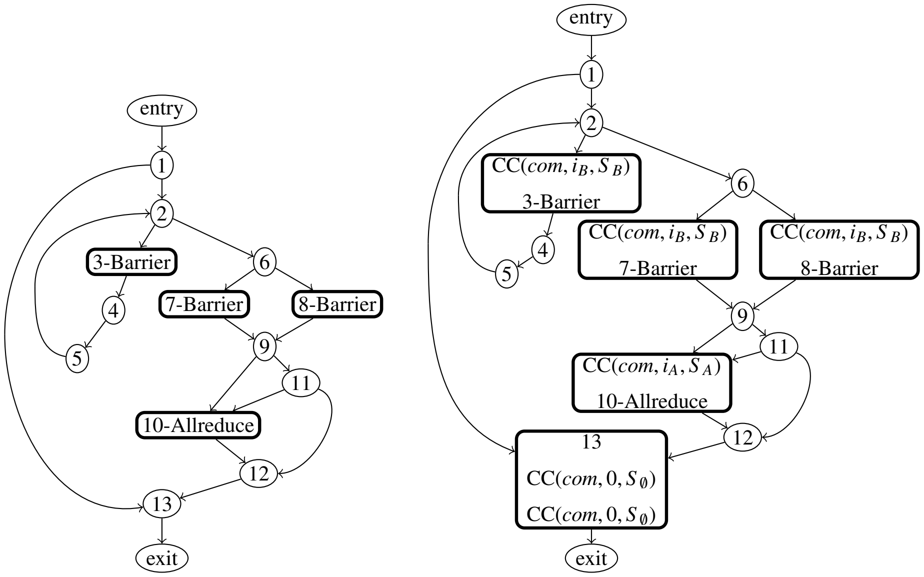
The algorithm is applied on each loop separately. This separate analysis identifies at least loop exit nodes as control-flow nodes that may be responsible for deadlocks, when the loop calls collectives. Indeed, static analysis does not count iterations and collectives in loops may be executed a different number of times for each process.

Now consider an incorrect function: when executing this function with multiple tasks, some tasks may reach the exit node while other tasks are waiting at a collective c inside the function. If this collective is not inside a loop, by definition of the execution order, this implies that the exit node and the node with the collective c have the same execution order r. As nodes may have multiple execution orders, let us consider the smallest r. There is a collective c' of order r such that the set of nodes $C_{r,c'}$ is not empty. Besides, $PDF^+(C_{r,c'}) \neq \emptyset$ since there is a path to the exit that does not traverse the r^{th} collective. Hence Algorithm 1 detects the function as incorrect. A similar proof holds for the case where the tasks are executing two different type of collectives. □



(a) CFG 1 (b) CFG 2 (c) CFG 3 (d) CFG 4

Figure 3.2: *Example of Control Flow Graphs. From the left, a CFG showing execution orders, CFG of function f and two other CFGs*



(a) CFG 5 before Algorithm 12 pass (b) CFG 5 instrumented (see **Algo.** 13)

Figure 3.3: *Example of a CFG from a Benchmark and their instrumentation (see Algorithm 13)*

Figure 3.2(a) shows an example of execution orders (numbers near each node) computation for a simple CFG. `MPI_Barrier` node 5 can be the first collective called (path *entry*→1→5) or the second one (path *entry*→1→2→3 or 4→ 5). On this CFG, the set of nodes $\{3, 4\}$ postdominates node 2: $\{3, 4\} \gg 2$ but $\{3, 4\} \not\gg 1$ so node 1 is in the iterated postdominance frontier of the set of nodes $\{3, 4\}$: $PDF^+(\{3, 4\}) = \{1\}$. This node may be responsible for a deadlock, while 2 is not. **Figure 3.2(b)** depicts the CFG extracted from the simple example **Figure 3.1**. It contains 3 nodes: The first one represents the `if` statement while the second one contains the `if` body with the collective call. Finally the last one denotes the `return` instruction. The algorithm considers the set $C_{0,\texttt{Barrier}} = \{2\}$ corresponding to the collective `MPI_Barrier`. As its iterated postdominance frontier is node 1, the algorithm outputs a warning for the condition located in node 1 and flags the collective `MPI_Barrier` for the following dynamic analysis (set O). **Figure 3.2(c)** presents a CFG containing two `MPI_Barrier` nodes 4 and 7. These nodes are of order 0. $C_{0,\texttt{Barrier}} = \{4, 7\}$ and $PDF^+(C_{0,\texttt{Barrier}}) = \{2, 6, 1\}$. The CFG **Figure 3.2(d)** contains two collective nodes. Node 2 has a `MPI_Barrier` while node 3 has a `MPI_Ibarrier`. The algorithm computes $C_{0,\texttt{Barrier}} = \{2\}$ and $C_{0,\texttt{Ibarrier}} = \{3\}$. The iterated postdominance frontiers of both sets contain node 1. Node 1 can potentially be the cause of a deadlock. **Figure 3.3(a)** presents another CFG extracted from a real benchmark. This example contains 2 collectives: MPI_Barrier (nodes 3, 7 and 8) and MPI_Allreduce (node 10). The algorithm first removes the backedge 5 → 2 from the loop and computes orders. Nodes $7, 8$ are of order 0, 10 of order 1. For the collectives in $C_{0,\texttt{Barrier}} = \{7, 8\}$, the iterated postdominance frontier corresponds to node 1. Note that node 6 is postdominated by the set $\{7, 8\}$ according to the definition of previous section. $C_{1,\texttt{Allreduce}}$ contains only node 10 and $PDF^{+}(C_{1,\texttt{Allreduce}}) = \{1, 9, 11\}$. Indeed from these nodes, it is possible to execute the MPI_Allreduce or not. Finally, the same algorithm is applied once more on the graph with nodes $\{2, 3, 4, 5\}$ corresponding to the loop, without the backedge. Node 2 is marked as entry and exit. This node is the only one in the iterated postdominance frontier of the barrier in node 3. To sum up, node 1 decides of the number of execution of barriers in $7, 8$, nodes $9, 11$ decide of the number of execution of MPI_Allreduce and node 2 is responsible for the number of barriers executed in node 3.

Potential errors reported by the static analysis can be false positives relatively to the global CFG that is possibly not correlated with the actual control flow. Our static analysis only returns candidate nodes which can possibly lead to a deadlock as we favoured a filtering approach in order to avoid the combinatorial aspect of execution simulation. To deal with false positive results caused by the compiler, a static instrumentation of the source code takes place to check at execution-time if warnings outputed by the static analysis will eventually lead to an error.

3.1.2 Static Instrumentation for Execution-Time Verification

The code fragments leading potentially to incorrect functions and detected with the previous analysis are transformed in order to raise an error message at the execution time: whenever MPI processes take execution paths that cannot lead to the same number of collectives, in the same order, the program stops. This subsection presents the code transformation involved.

Some potential errors may depend on the control flow taken by the different processes. The main idea is to modify the code so that before each MPI collective call, we check that all processes within the communicator are about to call the same collective. Besides, we also check that when a process is going to exit the function, all processes are exiting. This is achieved by a function, CC that counts the number of processes that are going to execute a given collective operation or to exit the function in which the MPI collective operation is invoked. CC is also a collective operation, as it gathers the processes of the communicator into groups depending on what they are going to call (collective type or exit). Function CC is depicted in **Algorithm 14**. It takes as input the communicator related to the collective call c, an integer i_c identifying the type of collective and the set of nodes generated by the previous algorithm (see the compile-time verification). We define a new MPI operator named *equalsop* which returns -1 if there is at least two different integer among processes. Relying on the CC function, **Algorithm 13** describes the instrumentation for the execution-time verification. The function INSTRUMENTATION is called on MPI_COMM_WORLD. For each node n containing a call to the collective c, MPI_Reduce is called just before calling c. The root process provides the combined value and test if all processes have the same input value. If input values are different among all processes, an error is issued and the program is aborted through a call to MPI_Abort. This process is repeated for each collective operation c in the set O. Finally, in the closest node of collective nodes that postdominates and joins all paths of the CFG, MPI_Reduce with the input value 0 is added to eventually catch up processes not calling any additional collective. An example of CFG instrumented is presented **Figure 3.3(b)**.

Lemma 2. ***Algorithm 13*** *is correct: all deadlock situations are captured by the instrumentation and the new collectives inserted do not generate a deadlock themselves.*

Proof. We define a *control sequence* as the sequence of collective calls executed by a process in a program execution. For an execution of a given function, a control sequence is denoted as $c_1 c_2 .. c_n$ with c_i the i-th col-

Algorithm 13 Step 2 - Selective Static Instrumentation

```
function INSTRUMENTATION($communicator, G, O$)        ▷ $G$: CFG, $O$: set created by Algorithm 12
    for $(c, S) \in O$ do
        for $n$ in nodes containing a call to collective $c$ do
            Insert call to CC($communicator, i_c, S$) before the call to $c$
        end for
    end for
    Insert call to CC($communicator, 0, \emptyset$) before return statements
end function
```

Algorithm 14 Library Function To Check Collectives (CC)

```
function CC($communicator, i_c, S$)
    int $rank$, $res$
    MPI_COMM_RANK($communicator$, &$rank$)
    MPI_REDUCE(&$i_c$, &$res$, 1, $MPI\_INT$, $equalsop$, 0, $communicator$)
    if $rank == 0$ && $res == -1$ then
        Display error for all nodes in $S$
        MPI_ABORT($communicator$,0)
    end if
end function
```

lective called. **Algorithm 13** rewrites each collective c_j from the set O into $s_j c_j$ corresponding to the function `MPI_Reduce` called by *CC* based on the color j and the initial collective c_j. The function `MPI_Reduce` with color 0 denoted as s_0 is added after all collective nodes. To ease the proof, we will assume that this conditional rewriting, performed only for collectives found by the static analysis, is conducted for all collectives of the control sequence. Consequently, a sequence $c_1..c_n$ becomes $s_1 c_1..s_n c_n s_0$. If all control sequences are the same for all processes, the function executes with no deadlock. By applying **Algorithm 13**, the modified control sequences are still identical, this algorithm does not introduce deadlocks. If a function deadlocks due to collective operations,

- Either a process calls a collective communication c_i while another process calls a collective function c_k with $k \neq i$. The control sequence of both processes differ only with their last collective, c_k and c_i, and both are prefixed by $c_1..c_{i-1}$.
- Or a process calls a collective communication while another one exits the function (a deadlock may occur at a later point in the execution or outside of the function). The control sequence of the process exiting the function is $c_1..c_{i-1}$ and the process inside the function executes the same prefix sequence with one more collective c_i.

In the first case, the algorithm changes both control sequences into $s_1 c_1..s_{i-1} c_{i-1} s_i$ and $s_1 c_1..s_{i-1} c_{i-1} s_k$. These sequences stop with s_i and s_k since CC(x,i) and CC(x,k) lead to an error detection and abort. Hence the modified function no longer deadlocks. In the second case, the algorithm changes both control sequences into $s_1 c_1..s_{i-1} c_{i-1} s_i$ for the process inside the function, and $s_1 c_1..$ $s_{i-1} c_{i-1} s_0$ for the one trying to leave the function. Note that the process is stopped before leaving the function since CC(x,i) and CC(x,0) both abort, generating an error message. Again, the modified function does not deadlock anymore.

To conclude, **Algorithm 13** is indeed correct and prevents all deadlock situations. □

3.1.3 Evaluation

The analyses were integrated in the PARCOACH tool. The pass applies **Algorithms 12 and 13**. This section presents experimental results obtained on representative C++ MPI applications: EulerMHD [110], solving the Euler and ideal magnetohydrodynamics equations both at high order on a 2D Cartesian mesh and HERA [108], a large multi-physics AMR hydrocode platform. We also selected six benchmarks from the MPI NAS Parallel benchmarks [104] (NASPB v3.2) using class C to test both C and Fortran programs. The error benchmark suite introduced in the previous chapter was extended with MPI programs following the errors categories depicted in [48] (deadlocks, data races, mismatches, resource handling, memory and portability).

CC Function Implementation

The goal of the CC function that checks MPI collective operations at runtime is to gather MPI processes calling the same collective call. For that purpose, we can split the communicator through a call to `MPI_Comm_split`, gather or reduce information about collectives with `MPI_Allgather`, `MPI_Gather`, `MPI_Allreduce` or `MPI_Reduce`, the best approach depending on the implementation. To determine which MPI function performs better, we used the Intel MPI Benchmark suite [111] (IMB) v3.2.3 with a 4B message as we only want to send an integer related to the type of the collective about to be called. **Figure 3.4** shows the time spent in each candidate function for a range of MPI processes. In this figure, `MPI_Reduce` seems to be the most scalable [112]. Hence we opted for this function.

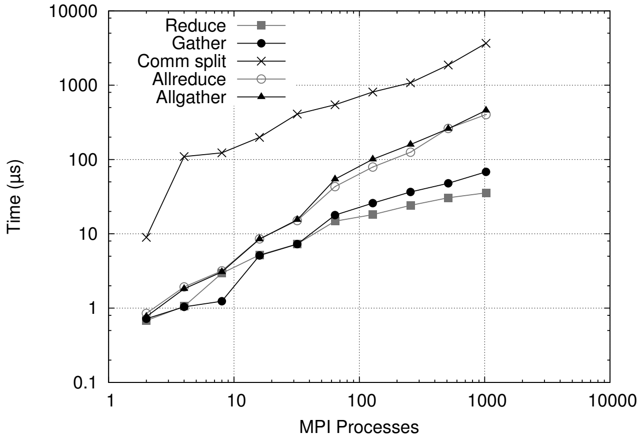


Figure 3.4: *Execution time of collective calls from IMB*

Benchmark	#coll. calls	#nodes in S	% instrumented collectives	#calls to CC
EulerMHD	14	14	36%	26
BT	10	5	78%	8
LU	16	2	14%	6
SP	9	5	75%	7
IS	5	2	40%	3
CG	2	0	0%	0
FT	10	0	0%	0
HERA	644	578	84%	3,255

Table 3.1: *Compilation and Execution Results of MPI Applications*

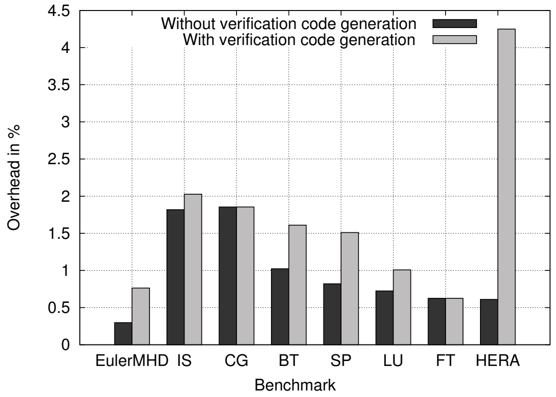


Figure 3.5: *Overhead of average compilation time with and without verification code generation (CC functions insertion)*

Static Check Results

At compile time, a warning is returned to the programmer when a potential deadlock situation is detected. The following example shows what a user can read on `stderr` for NAS benchmark IS:

```
is.c:In function 'main':
is.c:1093:1: warning: STATIC-CHECK: MPI_Reduce may not be called by all
processes in the communicator because of the conditional line 923 - Check
inserted before MPI_Reduce line 994
```

This warning provides the name of the collective that may deadlock (`MPI_Reduce`) and the line of the conditional leading to the collective call (line 923). This collective call is instrumented at line 994 as described in **Algorithm 13**. In this case, a test over the number of processes which if higher than 512 produces an error requesting a lower number of processes, before finalizing the program. This particular warning does not lead to a deadlock as all processes inside the same communicator share the same value for `MPI_Comm_size`. However, notice that the line number where the control flow divergence may occur is not close to the collective call: The conditional that may be responsible for a deadlock in a `MPI_Reduce` is 71 lines far from the collective.

Figure 3.5 details the overhead of compilation time when activating our GCC plugin. This overhead remains acceptable as it does not exceed 5% for HERA. It is presented with and without the code generation which accounts for the insertion of `CC` function calls (see **Algorithm 14**). This specific step is mainly responsible for the overhead except for CG and FT. Indeed, according to the static analysis, these benchmarks are correct, so no collective operation is instrumented. For each benchmark, **Table 3.1** presents the number of static calls to a collective communication and the number of nodes found by **Algorithm 12** (set $S = \cup(PDF^{+}(C_{r,c}) \in O)$). The location of the static analysis in the compilation chain explains the high number of collective calls found in HERA. Indeed, C++ templates are instantiated and, therefore, duplicated before entering the middle-end part of GCC. For all nodes in S, the control-flow does not depend on process ranks and the functions are correct. Nevertheless, this table shows that the static analysis is able to reduce the amount of instrumentation needed to check the collective patterns (third column). Reducing further the number of instrumented collectives would require an inter-procedural data-flow analysis on the nodes in S.

Execution Results

Figure 3.7(a) shows the overhead obtained for NASPB class C from 4 to 512 cores (CG and FT have no overhead as no collective is instrumented). The overhead does not exceed 18% and tends to slightly increase with the number of cores. **Figure 3.7(b)** presents weak-scaling results for EulerMHD from 1 to 1,280 cores

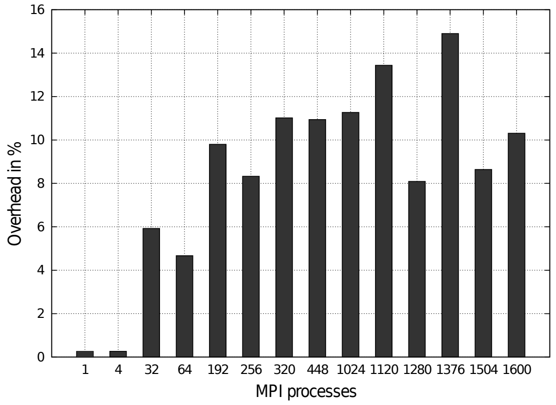
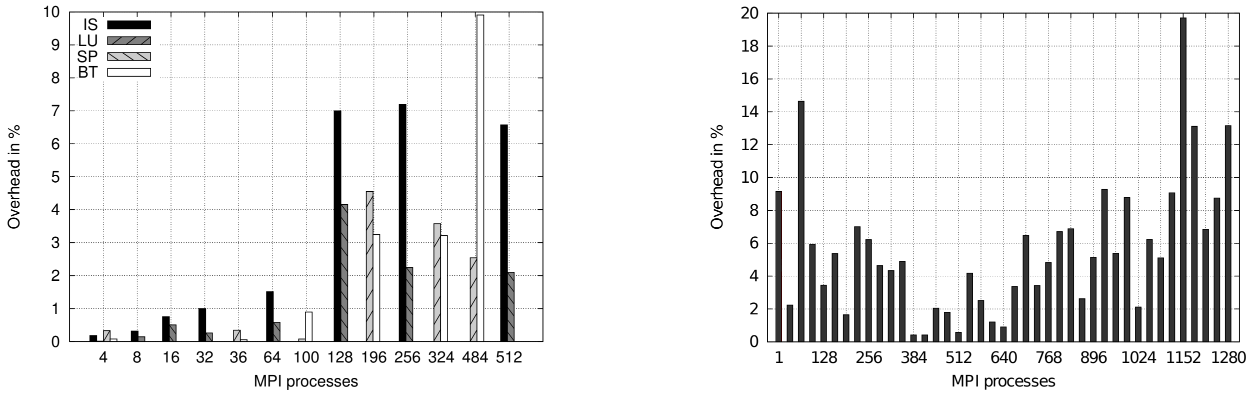


Figure 3.6: *Execution time overhead for HERA with strong scaling*



(a) Execution time overhead for NASPB

(b) Execution time overhead for EulerMHD

Figure 3.7: *Execution time overhead for NASPB class C with strong scaling and for EulerMHD with weak scaling*

where the overhead remains comparable with a higher overhead as it is related to the number of CC calls, with the same increasing trend. **Figure 3.6** presents the overhead obtained for HERA from 1 to 384 cores. The overhead also increases with the number of processes and does not exceed 12%. Highest execution times are of the order of 10 min for the benchmarks. The last column of **Table 3.1** depicts the number of calls to the CC function during the execution of the benchmarks. Processes about to call collectives identified as potential deadlock sources are counted. If some processes are missing, the abort function is called to stop the program before deadlocking. An error is printed to stderr with the line number, the collective name and conditionals responsible (informations gathered at compile-time):

```
DYNAMIC-CHECK: Error detected on rank 0 - Abort is invoking before MPI_Barrier
               line 47 in function f (program.c)
DYNAMIC-CHECK: See warnings about conditional(s) line(s) 45
```

This section presented an application of the PARCOACH two-phase analysis to detect incorrect collective patterns in MPI programs. The first pass statically identifies the reduced set of MPI collective communications that may eventually lead to potential deadlock situations, and issues warnings. Using this analysis, a selective instrumentation of the code is achieved, displaying an error, synchronously interrupting all processes, if the schedule leads to a deadlock situation. The compile-time overhead obtained is very low (5%). Dealing with the runtime overhead, it could be non-negligible at larger scale as PARCOACH adds collectives

Deadlock Situation 1

```
1  void f( ){
2    if(...)
3    {
4      #pragma omp parallel
5      {
6        if(...)
7        {
8           /*...*/
9           #pragma omp single
10          {
11             /*...*/
12          }
13       }
14       #pragma omp barrier
15       /*...*/
16     }
17   }
18 }
```

Deadlock Situation 2

```
1  void f() {
2    /*...*/
3    #pragma omp barrier
4    return;
5  }
6
7  int main(  ) {
8    int r;
9    #pragma omp parallel private(r)
10   {
11     r=...;
12     if( r == 0 )
13       f();
14   }
15   exit(0);
16 }
```

Figure 3.8: *Examples of deadlock situation in OpenMP programs*

for instrumentation. However, with the help of collective selection, the runtime overhead remains acceptable (less than 20%) at a representative scale on a C++ application.

The next section presents the detection of the origin of improper uses of OpenMP barriers and worksharing constructs (`single, for, section, workshare`) in OpenMP applications. Thread barriers and worksharing constructs are considered as *collectives* and present the same constraint as MPI collective operations.

3.2 Combining Static and Dynamic Analyses to Find the Origin of OpenMP Collective Errors

OpenMP is a popular parallel programming model for shared memory machines that aims at making parallel programming easier. However there are a number of improper uses of worksharing constructs and barriers that are not statically detected by compilers and may lead to deadlock or unspecified behavior. This section presents how the two-step method used in PARCOACH can detect the control flow codes responsible of improper uses of barriers and worksharing constructs in OpenMP applications.

The OpenMP specification requires that all threads of a team must execute the same sequence of worksharing constructs and barriers [29]. However in practice no error occurs when all threads of a team do not execute exactly the same barrier. That is why we authorize threads synchronizations with different barriers and defined two verbosity levels (0 and 1) defining soft and hard barriers verifications. To show the difficulty to enforce this constraint in OpenMP codes, consider the motivating examples in **Figure 3.8**. In function `f` of the deadlock situation 1, each thread may or may not encounter the `single` construct line 9, depending on the control flow (line 6). According to the OpenMP specification, all threads in a team should encounter the same `single`, or none of them. However, compiling this code and executing it does not lead to a syntactic error but to a deadlock. Indeed, if the result of the conditional is not the same among all threads, the first barrier executed will be for some threads the implicit barrier line 12 (end of `single`) while for others, it will be the explicit barrier line 14. Then the first group of threads will stop at the explicit barrier line 14 while the second group will stop at the barrier related to the end of the parallel region. Finally, the first set of threads will be released and eventually deadlock at this last barrier. Note that if we modify this example by adding an else statement with another single, the code is still potentially erroneous since all threads should encounter the same single. A more complex case appears in the deadlock situation 2. A deadlock can occur at the end of the parallel region of function main because of the conditional line 12. Depending on the control flow the barrier in f may be not encountered by all threads. The error is more difficult to detect and an interprocedural analysis is required. This illustrates the fact that the machine state does not help to identify the cause of deadlocks (in these two examples, conditionals).

3.2.1 Checking OpenMP Directives and Control Flow

In OpenMP programs, the threads of a team can synchronize through the #pragma omp barrier directive or at an implicit barrier at the end of worksharing regions (unless a nowait clause is specified). Worksharing constructs distribute the execution of the associated region among the threads of a team [29]. Worksharing constructs are loop, sections, single and workshare constructs. The OpenMP specification gives some restrictions to barriers and worksharing constructs. Indeed, each barrier/worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region ([29] Sections 2.7, p.53 and 2.12.3, p.124). However, due to the control flow inside an OpenMP program, the threads may execute different execution paths with different numbers of barriers and worksharing regions. Such behavior can lead to a deadlock or unspecified behaviors.

The principle of the static analysis we propose is the following. For each function of the code, we check that for all threads entering the function and for all teams created within it, the same number of barriers are executed, whatever the execution path taken by the threads. If the number of barriers may depend on the control flow, the control structures responsible for this are shown with a warning. This is a conservative approach, since we do not check that the conditional of an if statement for instance is dependent on the ID of the threads. Moreover, we check that worksharing constructs may not be conditionally executed, potentially leading to unspecified behaviors. This intra-procedural analysis on barriers and worksharing constructs is complemented by a simple inter-procedural analysis: User-defined functions are subsumed by the number of worksharing constructs and barriers executed by the entering threads. This captures all potential improper uses of barriers and worksharing constructs.

The program to analyze is represented using the OMPCFG intermediate representation, briefly described in the following section. Then the intra- and inter-analyses are presented.

Intermediate Representation: OMPCFG

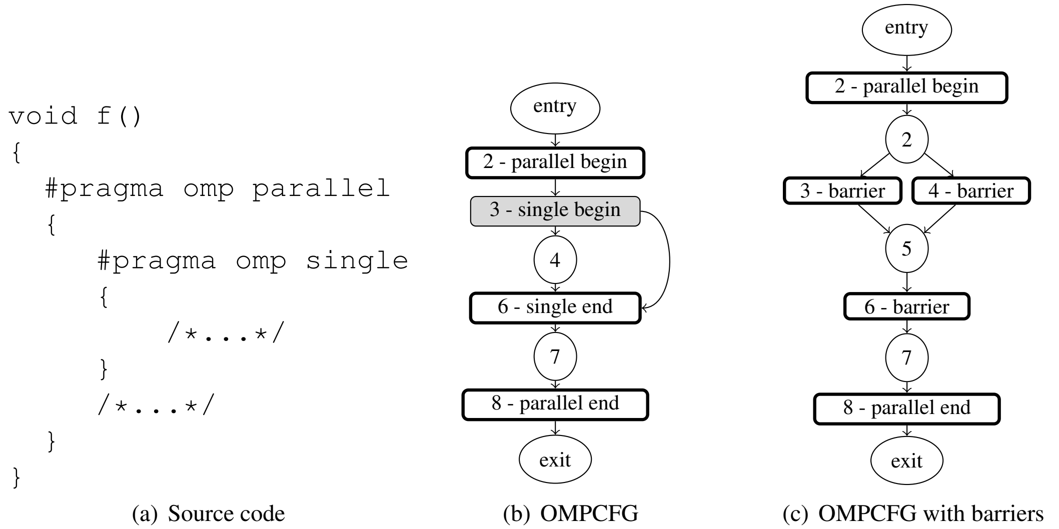
Lin [78] extended the notion of CFG to a representation for parallel OpenMP programs, called OMPCFG. Each node of the OMPCFG represents a basic block (basic nodes) or an individual block containing an OpenMP directive (directive nodes). In the OMPCFG, implicit barriers are made explicit and each combined parallel worksharing construct is separated into a nowait worksharing construct nested in a parallel region. Moreover the OMPCFG has a single *Entry* and single *Exit* nodes. New edges are inserted between basic nodes and directive nodes according to OpenMP semantics. As a result, the master directive is represented as a conditional. **Table 3.2** lists the OpenMP directives and their corresponding directive node in the OMPCFG. Note that Lin also adds edges from the end construct directive to the begin construct directive nodes denoted as construct edges. These edges are not considered here as they do not reflect any control flow.

Figure 3.9 shows examples of OMPCFG. All directive nodes containing a barrier are represented as thick nodes and all directive nodes containing a worksharing construct are colored in gray. Directive nodes containing a parallel construct are considered as barriers but are not considered in our Algorithms. Out of clarity, implicit barriers at the end of worksharing and parallel regions are not designated by barriers but

Table 3.2: *Directive nodes in the OMPCFG*

Directive name	Control flow	Worksharing construct
parallel, critical, atomic, section, barrier, ordered, task, taskwait, taskyield	linear	
master	if/else	
for, single	if/else	*
sections, workshare	switch/case	*

by *region-name* end.



```
void f()
{
  #pragma omp parallel
  {
     #pragma omp single
     {
        /*...*/
     }
     /*...*/
  }
}
```

(a) Source code (b) OMPCFG (c) OMPCFG with barriers

Figure 3.9: *Example of a simple code (a) with its corresponding OMPCFG (b) and an example of OMPCFG containing barriers (c)*

This representation is the base of the PARCOACH compiler analysis. GCC uses a graph representation similar to the OMPCFG from version 4.2.

3.2.2 Intra-Procedural Analysis

This section details the static verification of barriers and worksharing constructs for each function of a program. We define two levels of verbosity for barriers verification: level 0 that returns warnings only if there may be an execution error and level 1 that returns warnings in strict accordance with the specification.

For the verbosity level 0, we identify barrier statements that synchronize together. To that purpose, we introduce a number, the *sequential order*, counting the number of barriers traversed before reaching a barrier. This number is assigned to each node in the OMPCFG. Two nodes with different sequential order are sequentially ordered thanks to barriers. This number is 0 for nodes before the first barrier (including the node with the first barrier), 1 for nodes reached after one barrier and so on. When multiple paths exist, nodes can have multiple numbers, at most the number of barriers in the function. Loop backedges are removed to have a finite numbering. A function is not correct if there are nodes with multiple orders. These nodes correspond to possible control-flow divergence leading to deadlocks. In Zhang *et al.* [79], this notion of sequential order corresponds to phases, computed through an inter-procedural liveness analysis and a barrier aggregation step. While both methods can be used for our goal, our approach is simpler, more adapted to the verification of barriers. The computation of the execution order uses the **Algorithm 12, page 53**.

This algorithm detects possible control-flow divergence leading to a deadlock in a MPI barrier. MPI barriers are numbered by so-called *execution ranks* (similar to sequential order here). MPI barriers of same execution rank r are put into a set $C_{r,c}$ as matching MPI barriers. c is used to differentiate MPI collective operations names. In our case, only barriers are considered so the c is useless and only C_r sets are created. **Algorithm 15** is an adaptation of this method for OpenMP barriers, from line 6 to line 12. Barriers with multiple sequential orders are put in the set C_r with r corresponding to their maximal sequential order. For example the OMPCFG **Figure 3.9(c)** contains three explicit barriers nodes 3, 4 and 6 and one implicit barrier node 8. The sequential order for nodes 3 and 4 is 0, for node 6 , 1 and for node 8, 2. The algorithm computes $C_0 = \{3,4\}$, $C_1 = \{6\}$ and $C_2 = \{8\}$.

For the verbosity level 1, we verify each barrier is encountered by all threads of a team. This is described from line 14 to line 16 in **Algorithm 15**.

Algorithm 15 Step 1 - OpenMP Intra-procedural Control-flow Analysis

```
 1: function FUNCTION_VERIFICATION($f, v$)                    ▷ $f$: a function of the application
 2:                                                           ▷ $v$: level of verbosity
 3:     Compute $G = (V, E)$ the OMPCFG of $f$
 4:     $O \leftarrow \emptyset$, $O' \leftarrow \emptyset$    ▷ Output sets: Conditional nodes
 5:     if $v = 0$ then                                       ▷ level 0 of verbosity
 6:         Remove loop backedges in $G$ and Compute sequential order of all nodes
 7:         for $r$ in node orders do
 8:             for barriers of sequential order $r$ do
 9:                 $C_r \leftarrow \{u \in V | u \text{ of order } r\}$
10:                 $O \leftarrow O \cup (barrier, PDF^+(C_r))$
11:             end for
12:         end for
13:     else                                                  ▷ level 1 of verbosity
14:         for $u \in V$ s.t. $u$ contains an explicit barrier do
15:             $O \leftarrow O \cup (barrier, PDF^+(u))$
16:         end for
17:     end if
18:
19:     for $u \in V$ s.t. $u$ contains a worksharing construct $w$ do
20:         $O' \leftarrow O' \cup (w, PDF^+(u))$
21:     end for
22:     Output nodes in $O$ and $O'$ as warnings
23: end function
```

The algorithm takes the OMPCFG of the current function and the verbosity level as input parameters and outputs a message error for conditional nodes that may lead to a deadlock in a barrier (set O). The core of the algorithm is based on the postdominance frontier. If barriers with the same sequential order r have a non-empty PDF^+ set, then some threads may not perform the n^{th} synchronization. Due to the representation of all worksharing constructs (as if/else or switch), barriers inside these worksharing constructs are detected as incorrect.

The lines 19 to 21 of the algorithm detect if worksharing constructs may not be executed by all threads of a team. For each node u containing a worksharing construct, we compute the iterated postdominance frontier of u. If the $PDF^+(u)$ is not empty then some threads may execute the construct while others may avoid it. The set of nodes detected are put in the set O' for warnings.

Lemma 3. ***Algorithm 15*** *is correct: it detects all deadlock situations due to* `barrier` *and worksharing regions.*

Proof. The levels of verbosity enable a strict verification of barriers in compliance with the specification. In that purpose **Algorithm 15** detects if all threads of a team have strictly the same sequence of barriers. A soft verification is also possible. The algorithm then verifies all threads of a team encounter the same number of barriers. The proof has been done Section 3.1.1. Then **Algorithm 15** computes the set O' of control-flow nodes that have execution paths with different number or type of worksharing constructs from the node to the *Exit* node. We prove that nodes in O' correspond exactly to the nodes that lead to a deadlock. □

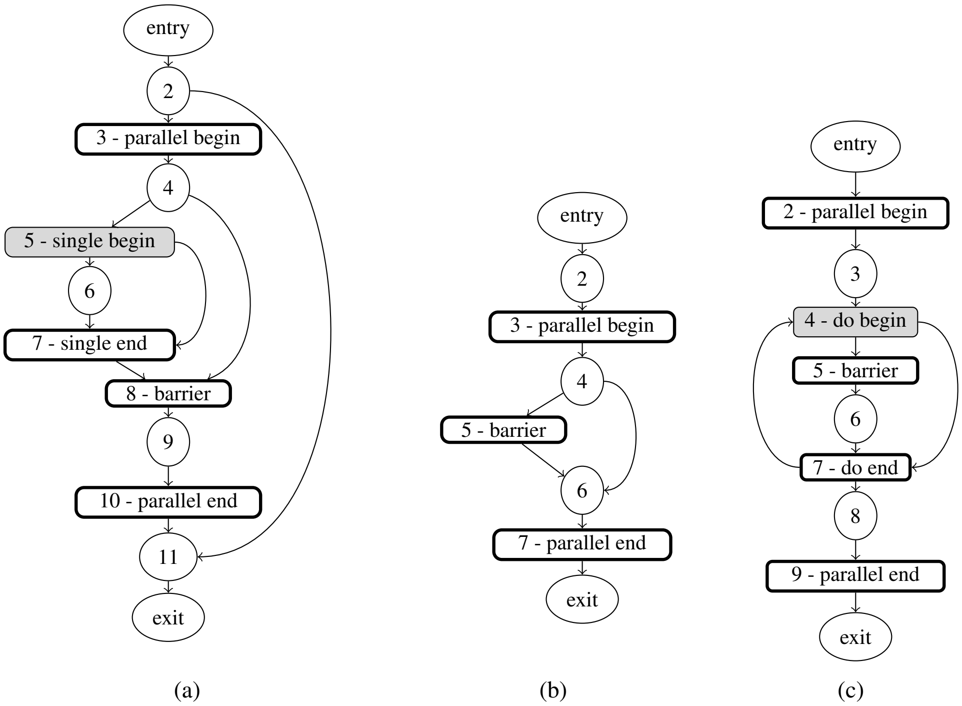


Figure 3.10: *Functions `f` OMPCFG of Listing 2.3 ((a)) and `main` OMPCFG of Listing 2.6 after function `f` replacement ((b), see **Algorithm 18**) and an example of an OMPCFG with a loop ((c))*

As an example, the first OMPCFG **Figure 3.10(a)** contains one explicit barrier (node 8), two implicit barriers (nodes 7 and 10) and one worksharing construct: `single` (node 5). **Algorithm 15** computes sequential orders. Node 7 is of sequential order 0, node 8 is of sequential orders 0 and 1 and finally node 10 is of sequential orders 1 and 2. Thus we have $C_0 = \{7\}$, $C_1 = \{8\}$ (node 8 is in C_1 as it has multiple sequential orders) and $C_2 = \{10\}$. $PDF^+(C_1) = \emptyset$ and $PDF^+(C_2) = \emptyset$ but $PDF^+(C_0) = \{4\}$. Node 4 is the only node in the iterated postdominance frontier of node 7 as the conditional node 2 is outside the parallel region. Then the conditional node 4 is returned as the possible cause of a deadlock in a `barrier`. For node 5, $PDF^+(5) = 4$. To sum up for Listing 2.3, a warning is issued for the conditional located in node 4 as potentially leading to different barriers and worksharing constructs sequence among threads. The OMPCFG **Figure 3.10(b)** contains one explicit barrier node 5 and one implicit barrier node 7. The algorithm computes $C_0 = \{5\}$, $C_1 = \{7\}$ and $PDF^+(C_0) = \{4\}$. Last, the OMPCFG **Figure 3.10(c)** contains one worksharing construct node 4, one explicit barrier node 5, two implicit barriers nodes 7 and 9 and a loop (composed of nodes 4, 5, 6, 7). First **Algorithm 15** removes the loop backedge from node 7 to node 4. Then sequential orders are computed: $C_0 = \{5\}$, $C_1 = \{7\}$ and $C_2 = \{9\}$. A warning is issued for the conditional node 4 as $PDF^+(C_0) = \{4\}$. For the loop construct node 4, the iterated postdominance frontier is empty.

3.2.3 Static Instrumentation for Execution-Time Verification

A test verifying if all tasks are going to synchronize or encounter the same worksharing construct is done before each collective OpenMP by the function CC (see **Algorithm 17**). Relying on the CC function,

Algorithm 16 describes the instrumentation for the execution-time verification. If there is only one thread in the team, the code is correct. Otherwise, each thread updates i_c, an integer associated to the type of the collective they are about to call. Barriers and the end of parallel regions are differentiated. If threads of the team are not all going to call a collective, abort is called. The `atomic` directive ensures shared variables are updated by only one thread at a time.

Algorithm 16 Step 2 - Selective Static Instrumentation

```
function INSTRUMENTATION($G, O, O'$)                ▷ $G$: CFG, $O, O'$: sets created by Algorithm 15
    for $(c, S) \in O, O'$ do
        for $n$ in nodes containing a call to collective $c$ do
            Insert call to CC($c, S$) before the call to $c$
        end for
    end for
    Insert call to CC($0, \emptyset$) before return statements
end function
```

Algorithm 17 Library Function To Check Collectives (CC)

```
function CC($c, S$)
    int $i_c$
    if OMP_GET_NUM_THREADS()!=1 then
        # pragma omp atomic
        $i_c + +$                              ▷ each task updates the variable relied to its barrier
        # pragma omp barrier
        if $i_c ! =$ OMP_GET_NUM_THREADS() then
            Display error for all nodes in $S$
            ABORT()
        end if
        RESET_COLLECTIVE_INTEGERS()
    end if
end function
```

3.2.4 Inter-Procedural Analysis

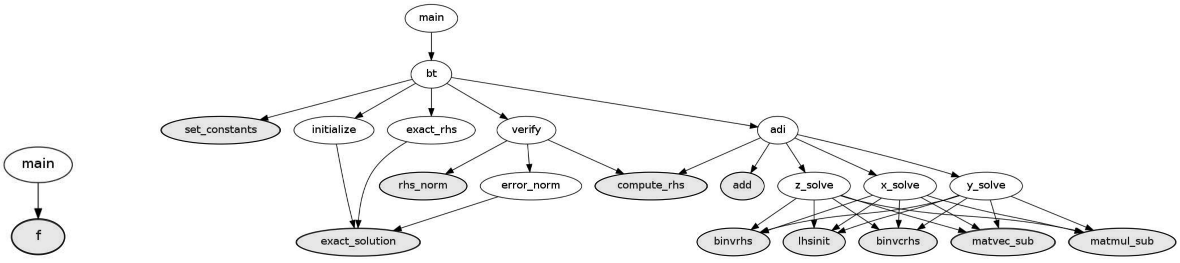
This section describes the analysis for the whole application code. We assume the application is not using recursion, meaning the callgraph of the application has no cycle.

The method iterates through the callgraph, in reverse topological order. It starts with functions that do not call other functions in the code, then callers of these functions, and so forth. After the previous analysis of **Algorithm 15**, each function retains the minimal number of barriers executed by the team of threads entering the function (excluding the barriers executed by teams created inside the function), as well as the number of worksharing constructs executed by this same team. These numbers are denoted $n_{\texttt{barrier}}$ for the number of barriers, n_d for worksharing constructs (among `for`, `worksharing`, `sections`, `single`). They are obtained through a simple traversal of the OMPCFG of the function. When a function g is called from a function f, g is replaced by as many barriers and worksharing constructs as these values. For worksharing constructs, only the number of constructs matters for the analysis. Indeed, we verify each callee function with worksharing constructs are not depending on the control flow in caller functions. Then the analysis `Function_verification` is called on f. These steps are described in **Algorithm 18**.

Figure 3.11 shows callgraphs of Listing 3.4 and BT from the NAS parallel benchmarks OpenMP. Nodes colored in gray are first nodes considered by **Algorithm 18**. In the example of the Listing 3.4 callgraph, **Algorithm 18** computes $n_{worksharing}(f) = 0$ and $n_{barrier}(f) = 0$ which are the minimal numbers of barriers and worksharing constructs in function `f`. Function f is then replaced by these numbers in `main`.

Algorithm 18 Step 1 - OpenMP Inter-Procedural Analysis

```
function CODE_VERIFICATION($CG, v$)        ▷ $CG$: call graph        ▷ $v$: level of verbosity
    Sort $CG$ in reverse topological order
    for $f \in CG$ do
        for $g$ a callee in $f$ do
            Compute $n_d(g)$ for $d$ =barrier and worksharing constructs        ▷ $n_d$: minimal number of directives $d$ executed by entering threads
            Replace $g$ in $f$ by $n_d(g)$ empty worksharing constructs and $n_{\texttt{barrier}}(g)$ barriers.
        end for
        Compute Function_verification($f, v$)
    end for
end function
```



(a) (b)

Figure 3.11: *Callgraph of Listing 3.4 (a) and BT from NASPB-OMP (b)*

3.2.5 Evaluation

PARCOACH was enhanced to detect collective errors in OpenMP applications. The pass applies **Algorithms 15 and 16**. This section presents experimental results on the NAS parallel Benchmarks OpenMP (NASPB-OMP) [104] v3.2 using class B and HERA [108]. Even if the test case used by HERA is parallelized with MPI+OpenMP, only the correctness of OpenMP barriers and worksharing constructs have been checked. The error benchmark suite was extended with OpenMP programs containing the *Violations of the Standard/Nonconforming Program* class depicted in [74]. The number of lines and the language of each benchmark is presented **Table 3.3**.

PARCOACH issues warnings for barriers and worksharing constructs potentially not encountered by all threads of a team. The name of the OpenMP directive with potential improper use and the line of the conditional leading to this situation are returned to the programmer. The following example shows what a user can read on `stderr` when compiling Listing 2.3 with our plugin.

```
in function 'f':
example.c: warning: STATIC-CHECK: #pragma omp single line 9 is
possibly not called by all threads because of the condition line 6
```

Table 3.3 shows the number of barriers and worksharing constructs found in each benchmark and the number of nodes in the sets S and S' generated by **Algorithm 15** with the two verbosity levels. For all these nodes, the control flow does not depend on thread ID and therefore functions are correct. A data-flow analysis could be done to complement our analysis to reduce the number of false positives. Indeed, a check on the conditionals in $S \cup S'$ could help the plugin to detect control flow not depending on threads ID and avoid false positives. This is left for future work. The table also presents first results for the inter-procedural analysis by giving the number of functions executed in parallel with a non null minimal number of barriers

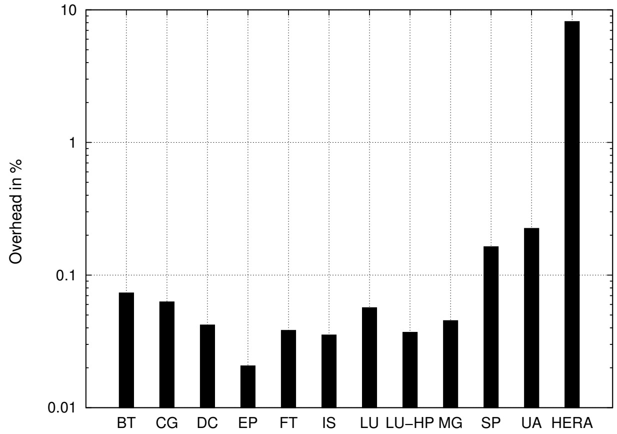


Figure 3.12: *Overhead of average compilation time for NASPB-OMP and HERA*

or worksharing constructs. These functions may be replaced by their callee functions in the source code to report errors considering the entire program.

Table 3.3: *Static Results for each benchmark (F=FORTRAN)*

Benchmark	**NASPB-OMP**											**HERA**
	BT	**CG**	**DC**	**EP**	**FT**	**IS**	**LU**	**LU-HP**	**MG**	**SP**	**UA**	
Language	F	F	C	F	F	C	F	F	F	F	F	C++
# lines	3,835	1,204	3,295	294	1,336	940	3,921	3,875	1,497	3,309	8,375	827,739
# explicit barriers	0	0	0	0	0	2	3	0	0	0	0	92
# worksharing	31	18	0	1	8	5	37	37	15	35	77	1,622
# nodes in S ∪ S′	**Verbosity 0**											
	0	0	0	0	0	0	0	0	0	0	0	564
	Verbosity 1											
	0	0	0	0	0	0	0	0	0	0	0	587
# functions with $(n_{barrier} + n_d) \neq 0$	3	3	0	0	0	0	8	4	2	3	15	398

The compile-time overhead obtained when compiling the applications and activating our plugin is shown **Figure 3.12**. The overhead remains acceptable as it does not exceed 0.25% for NASPB-OMP and 10% for HERA (caused by the size of the code, it takes 52.3 minutes to compile HERA with PARCOACH).

This section presented an application of the PARCOACH two-phase analysis to detect misuse of OpenMP collectives (barriers and worksharing constructs). The intra-procedural analysis proposed verifies if all threads entering a function and created in it have the same sequence of worksharing regions and depending of the verbosity level, the same sequence of barriers (verbosity 1) or the same number of barriers (verbosity 0) in OpenMP applications. Potential errors are automatically returned to the user with the line of the erroneous conditionals by a simple analysis of the OMPCFG. Thus the user knows exactly what can cause a deadlock and correct it. The compilation-time overhead obtained is low as it is under 10%. An inter-procedural analysis complements the analysis for checking the whole application by considering functions interconnections.

3.3 Summary

This chapter explains how the first version of the GCC plugin PARCOACH is extended to detect collective errors in MPI and OpenMP applications. Our method detects incorrect functions by combining both static and dynamic approaches.

The static analysis detects all incorrect functions of a program and issues warnings for potential errors. Then because these potential errors might not appear during execution, the code is transformed in order to check only the reported warnings. In the case of an actual deadlock situation, the program aborts allowing a program state exploration with a debugger.

Like all static tools, our method has the advantage of not requiring execution of the program but can produce false positives. PARCOACH is based on the MPI semantics and is thus independent on MPI implementation. Unlike the MPI tool TASS, our static check analyzer requires no source-code modifications since it is integrated within a compiler. Potential errors are automatically returned to the programmer with their context (including the line of the erroneous conditional) through a low-complexity control-flow graph analysis. However a pragma-based approach could be useful to improve our static analysis (for example by tagging MPI rank dependent variables), thus reducing false-positive possibilities. Besides, in our approach, the combinatorial aspect of detecting effective mismatch is avoided by the runtime check. We perform a runtime check, taking advantage of the compile time analysis results (code locus and potential error filtering) in order to scale to large programs, avoiding instrumentation of the whole MPI interface or systematic code instrumentation like in MPI-CHECK. Compared to other dynamic analysis tools, our method provides more precise errors including the conditionals responsible with our static check help. Our analysis complements existing MPI-only debugging tools in that it focuses on detecting and avoiding collective errors.

Contrary to MPI, PARCOACH is based on GCC optimizations and thus is limited in the use of OpenMP. Compared to existing work, in particular the method of Zhang *et al.* [79], our OpenMP verification technique is fast (introducing little overhead) and able to scale to large applications. Furthermore, our analysis is language independent and verifies woksharing-construct placements in a program. To detect possible deadlocks we use the graph representation defined in [78]. Potential errors are automatically returned to the user with the line of the erroneous conditionals by a simple analysis of the OMPCFG. Thus the user knows exactly what can cause a deadlock and correct it.

Although it satisfies both scalability and functional requirements, PARCOACH is only intra-procedural with the possible drawback of missing conditional statements out of function boundaries. A solution to move towards inter-procedural analyses was presented for OpenMP applications. The solution consists in a traversal of an application callgraph in reverse topological order. Then each function is analyzed and summarized by the minimal number of collectives (barriers and worksharing constructs) in it. This approach follows bottom-up approaches.

This chapter focuses on MPI and OpenMP models. However every model following the same collective constraints can be checked by the analyses. For instance, the UPC language based on the PGAS model (see **Section 1.1.5**) has a `UPC_barrier` that should be called by all threads. Furthermore, PARCOACH can be seen as a preliminary work setting the basis for a wider set of analyses combining static and dynamic aspects and extended to hybrid (OpenMP + MPI) parallelisms. MPI and OpenMP analyses can be performed one after the other to detect errors in MPI+OpenMP applications if these models are distinct (corresponding to MPI_THREAD_SINGLE and MPI_THREAD_FUNNELED levels in an MPI+OpenMP program). But this can be insufficient as some errors can occur only when models are mixed. For example concurrent MPI collective calls within a MPI process. The next chapter exposes an extension of the two-step analysis to detect collective errors in MPI+OpenMP applications.

Chapter 4

Detection of Collective Errors Origin in Applications Mixing Parallel Programming Models

Even if hybrid applications are more and more common, most debugging tools are focused on one type of parallelism model. However errors in hybrid programs (whatever the thread-level support) can result from the combination of both forms of parallelism. Besides integrating two different programming models inside the same application can generate complex bugs very hard to identify. These kind of errors cannot be found by one-model debugging tools. This chapter proves the needs of dedicated analyses and supplies them to detect bugs related to misuse of MPI collective operations inside and outside threaded regions. More specifically the chapter localizes the source of MPI collective errors in MPI+X applications with X a shared memory model. We target SPMD (Single Program Multiple Data) MPI programs and explicit fork/join multi-threaded models with perfectly nested regions. OpenMP corresponds to this kind of model. Thus throughout the chapter, examples provided are parallelized with MPI+OpenMP. This work has been published in a poster [100] which content is reproduced in this chapter.

4.1 Static and Dynamic Validation of MPI Collective Communications in Multi-threaded Context

Three thread levels enable MPI communications inside OpenMP parallel regions. In particular, the MPI_THREAD_MULTIPLE level authorizes multiple threads to call MPI functions simultaneously. This configuration allows a full exploitation of resources with concurrent OpenMP threads and MPI communications. However, according to the standard, it is the user's responsibility to ensure that MPI communications (including collective calls) are correctly placed, according to the thread level used. More specifically, if the number of expected calls to a collective operation or their sequence is not the same for all processes, this can lead to errors or deadlocks. For example in the simple code given previously:

```
#pragma omp parallel
{
   MPI_Allreduce(...);
}
```

there will be as many calls to MPI_Allreduce as the number of threads created by the parallel region. This is an issue because the sequence of MPI collective calls should be deterministic within each MPI process, or the order of collective calls is not guaranteed. Finding such bugs and, moreover, the source of the errors may be challenging.

The previous simple example illustrates the case where the multithreaded programming model may change the execution flow leading to a deadlock due to one misuse of MPI collective communications. In an MPI+OpenMP program, not only the correctness of MPI should be ensured but also the multi-threaded model should not interfere with MPI. **Figure 4.1** illustrates some of the possible issues related to MPI collective communications in a multithreaded context through six examples. We focus here only on such errors. Issues related to OpenMP workshare constructs can be detected by other existing tools.

Code 1

```
void f(){
  #pragma omp parallel
  {
     #pragma omp parallel
     {
       /***/
       #pragma omp single
       {
          MPI_Allreduce(...)
       }
     }
   }
}
```

Code 2

```
void f(){
  #pragma omp parallel
  {
     #pragma omp single nowait
     {
        MPI_Reduce(...)
     }
     /***/
     #pragma omp single
     {
        MPI_Reduce(...)
     }
  }
}
```

Code 3

```
void f(){
  #pragma omp parallel
  {
     #pragma omp single
     {
        MPI_Reduce(...)
     }
     /***/
     #pragma omp single
     {
        MPI_Reduce(...)
     }
  }
}
```

Code 4

```
void f(){
  #pragma omp parallel
  {
    #pragma omp single
    {
      if(...)
        MPI_Allreduce(...)
      else
        /***/
    }
    /***/
  }
}
```

Code 5

```
void f(){
  if(...)
  {
    #pragma omp parallel
    {
      /***/
      #pragma omp single
      {
        MPI_Allreduce(...)
      }
    }
  }
}
```

Code 6

```
void f(){
  #pragma omp parallel
  {
    /***/
    if(...) {
       #pragma omp parallel
       {
          /***/
          #pragma omp single
          {
            MPI_Allreduce(...)
          }
       }
    }
    /***/
  }
}
```

Figure 4.1: *MPI+OpenMP examples showing different uses of MPI collectives.*

First of all, within a process the same communicator may not be used concurrently by two different MPI collective calls. This means MPI collective operations should not be called by multiple parallel threads. Although `MPI_Allreduce` in Code 1 is called in a `single` block, the nested parallel regions could lead to more than one concurrent call to this function. This erroneous situation always occurs unless only one thread is created in the first parallel region or in both regions. However, executing collectives in a monothreaded region is not sufficient: multiple monothreaded regions may be executed in parallel, creating bugs, as illustrated by the function `f` in Code 1: It contains two `single` constructs in one OpenMP parallel region. Even if this piece of code seems to require only a low thread-level support, the first construct contains a `nowait` clause, therefore both `MPI_Reduce` can be called simultaneously by different threads. On the contrary, the threads in Code 3 synchronize at the implicit OpenMP barrier at the end of the `single` construct. In this case the different `MPI_Reduce` functions are not performed concurrently, unless the function `f` is called from a worksharing region.

In addition to check that collective calls cannot be concurrently executed by different threads, the sequence of such calls is also of interest. The order of MPI collectives have to be the same for all MPI processes. A function is said to be correct if all processes entering a function do not exit leaving a process blocked inside a collective operation. This property still have to be ensured in MPI+OpenMP programs as those shown in Codes 4, 5 and 6. In Code 4, the thread executing the single region may not call `MPI_Allreduce` depending on the previous conditional statement (line 6). If the result of the conditional is not the same for all processes, some MPI tasks will call the collective while others will not. In the same way, `MPI_Allreduce` in Code 5 may not be called by all processes because of the conditional line 2.

These examples illustrate the difficulty for a developer to ensure that MPI collective operations are correctly used inside an hybrid MPI+OpenMP application whatever the required thread-level support. To tackle this issue, we propose a compile-time method to check that MPI collective calls are executed with the same deterministic sequence for each process in presence of a thread-based programming model, for all thread levels proposed by MPI.

4.1.1 Problem Statement

In our context, the problem statement can be expressed as follows: A hybrid program is correct if all MPI processes execute the same MPI collective operations in the same order in a deterministic way. This means there is a total order between MPI collective calls within each process and this order is the same for all MPI processes.

The compile-time analysis we propose in the following section analyses the code in order to prove this property, and when this depends on runtime conditions (such as the control flow or the number of threads executing a statement), these conditions are inserted in the code. Note that whenever the compile-time analysis is able to prove the condition statically, no code is inserted into the program, lightening the impact of the transformation on the execution time. In order to prove that a hybrid program is correct, the analysis is decomposed into three phases proving the following statements:

1. Within an MPI process, all collectives are executed in a monothreaded context;

2. Within an MPI process, if (1) holds, two collective executions are ordered sequentially, either because they belong to the same monothreaded region or because they are separated by a thread synchronization;

3. Once (1) and (2) have been proved, the sequence of collectives for all MPI processes does not depend on the control flow.

(1) and (2) ensure that collective operations are executed in an order that does not depend on the number of threads, nor on their execution schedule. (3) shows that the same sequence of collectives is executed for all MPI processes, and when the compile-time analysis is not able to prove this property due to some control flow statements, checks are inserted at these statements. This corresponds to the instrumentation required for the verification. **Figure 4.2** subsumes this decomposition of the problem statement into the analysis of categories of errors corresponding to the previous items.

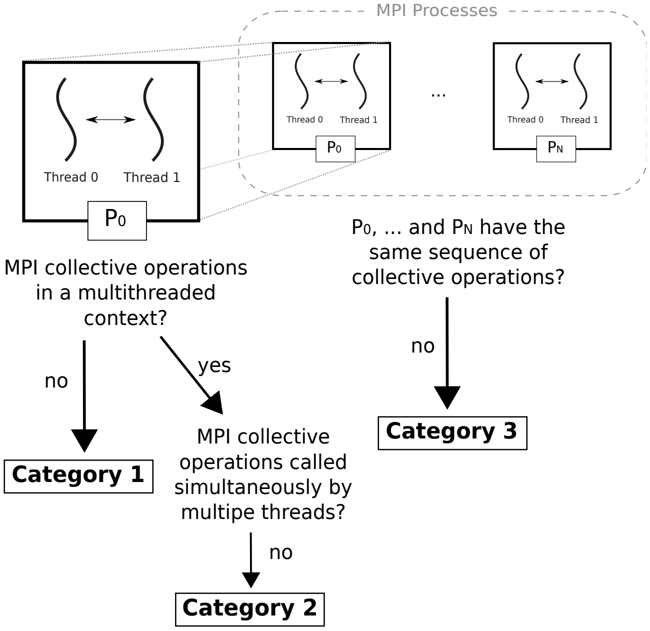

MPI Processes
Thread 0
Thread 1
P0
PN
...
P0, ... and PN have the same sequence of collective operations?
MPI collective operations in a multithreaded context?
no
yes
Category 1
MPI collective operations called simultaneously by multipe threads?
no
Category 2
no
Category 3

Figure 4.2: *Possible errors in a hybrid program with N MPI processes and 2 threads per process*

A function is said to be incorrect if at least one of the three categories presented in **Figure 4.2** is verified. Functions f in Codes 1 and 6 verify the first category whereas the function f in Code 2 is in the second category and in Codes 4, 5 and 6, f is in the third category. To conclude functions presented in Codes 1, 2, 4, 5 and 6 are considered as incorrect functions. Function f in Code 3 is the only correct function regarding these categories, provided the function is not called within a multi-threaded context (such as a `parallel for` block).

4.1.2 Compile-Time Verification

This section describes how categories depicted in **Figure 4.2** can be detected at compile-time.

The CFG is augmented as to highlight collective nodes (represented as thick nodes) and, as for the GCC compiler, OpenMP directives are put into separate basic blocks. Hence new nodes are added for explicit and implicit thread barriers. For greater clarity, implicit thread barriers at the end of parallel regions are referred as `end parallel`. **Figure 4.3(a)** depicts the CFG of Code 1.

As stated in the previous section, the static analysis aims at verifying the total order of MPI collective calls within a process as well as between processes. To this end, we use the notion of *parallelism word* for a basic block defined Section 2.1.1.

Detecting Category 1 Situations

This phase of the compile-time analysis corresponds to the detection of MPI collectives that are not executed in a monothreaded region.

For this part, thread barriers can be safely ignored as they do not influence the level of thread parallelism. Checking that a collective is executed in a monothreaded region boils down to check the parallelism word of its basic block. This requires to use MPI with at least a level defined by `MPI_THREAD_SERIALIZED`. If a collective is executed in a multithreaded region, this requires further the use of the level `MPI_THREAD_MULTIPLE` and the code is then correct if only one thread execute the collective. To be in a monothreaded region, the parallelism word has to end with an *S* (ignoring *B*s). Moreover, if the parallelism word has a

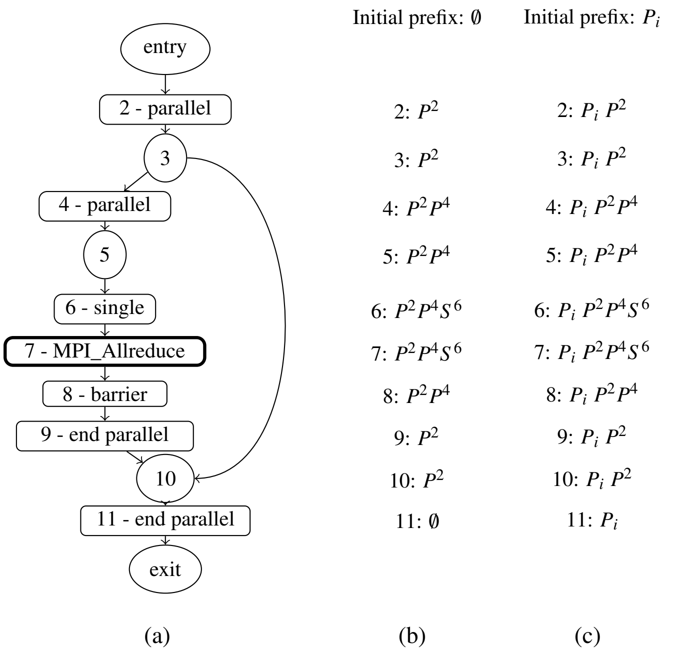


Figure 4.3: *CFG of Code 6 and the parallelism words associated at each node*

sequence of two or more P with no S in-between, it implies the parallelism is nested: Even if the word ends with an S, one thread for each thread team can execute the collective. We assume then the collective is not executed in a monothreaded region.

For instance, in Code 2 the function `MPI_Reduce` line 6 is called in a single construct inside a `parallel` region. Its associated parallelism word is PS. PS defines a monothreaded context as the `single` construct creates a portion of code executed by one thread in the current team. The function `MPI_Allreduce` line 9 of Code 1 is called in a `single` construct inside two nested parallel regions. Its associated word is PPS. There will be as many threads executing the single region as the number of threads created at the first parallel region. This parallelism word thus defines a multithreaded region.

We define a finite-state automaton **Figure 4.4** that recognizes the language of parallelism words corresponding to monothreaded regions. This automaton is a variation of automaton **Figure 2.5, page 39** where master sections are not differentiated from single sections. The state 0 is the accepting state and the language L defined by $L = (S|PB^*S)^*$ describes the accepted parallelism words. This language contains parallelism words ending by S without a repeated sequence of P.

Lemma 4. *A node n is in a monothreaded context if $pw[n] \in L$.*

Proof. We suppose $pw[n] \notin L$. Hence the automaton with the input $pw[n]$ stops either in the state corresponding to $(S|PB^*S)^*PB^*$ or in the state corresponding to $(S|PB^*S)^*PB^*P(P|S|B)^*$. It implies that:

- Either $pw[n]$ ends with a word in PB^*: Multiple threads may execute n.
- Or $pw[n]$ contains a string in PB^*P. This corresponds to a nested parallelism region.

Both cases do not correspond to a monothreaded region. We conclude that n is in a monothreaded context if its corresponding parallelism word $pw[n]$ is in L. □

To consider all possible initial conditions, the initial parallelism word of the function is an initial prefix P_i unknown at compile-time. **Figure 4.3(c)** presents parallelism words of the CFG **Figure 4.3(a)** with an initial

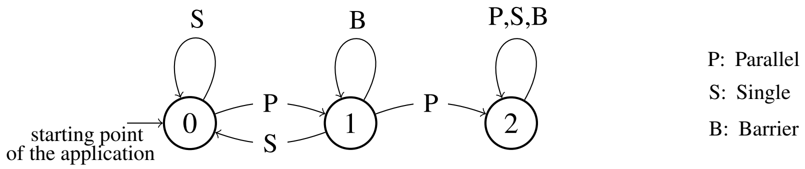


Figure 4.4: *Automaton of possible parallelism words. Node 0 corresponds to code executed by the master thread or a single thread. Node 1 correspond to code executed in a parallel region and 3 to code executed in nested parallel region.*

prefix. Considering level 1 as the initial condition, the language of correct parallelism words recognized by automaton **Figure 4.4** is $L = S(S|PB^*S)^*$. A level superior or equal to 2 as initial condition has no accepted language.

In order to limit the number of warnings emitted during this analysis, the programmer can select with an option given to the analysis the initial level to consider at compile-time. Based on the language of correct parallelism words, **Algorithm 4** detects if an MPI collective operation is called in a multithreaded context. In that case a warning related to the initial level with the name of the collective is returned to the programmer. The algorithm takes as input the CFG and the language L of correct parallelism words and outputs two sets: S and S_{ipw}.

The CFG **Figure 4.3(a)** contains only one MPI collective: `MPI_Allreduce` (node 7). This collective may be called by multiple threads at runtime as $pw[7] = P^2P^4S^6$. Indeed, the parallelism word of node 7 is not in L. As $pw[7]$ corresponds to a multithreaded execution context,, **Algorithm 4** gets the uppest node that dominates the collective node before the execution or control flow changes. From node 4 the execution flow changes. Finally the algorithm outputs a warning for node 7 ($S = \{7\}$) and flags the node 5 for the dynamic analysis ($S_{ipw} = \{5\}$). If indeed both `parallel` constructs create teams of strictly more than 1 thread, then there is an error.

Detecting Category 2 Situations

For this analysis, the MPI collectives are assumed to be called in monothreaded regions, as defined in previous section. However, different MPI collectives can still be executed simultaneously if monothreaded regions are executed in parallel.

Different MPI collective operations in the same monothreaded region are sequentially performed as only one thread executes it. However, collective operations in different monothreaded regions may be called simultaneously. The following lemma defines this difference.

Lemma 5. *Two nodes n_1 and n_2 are said to be in concurrent monothreaded regions if they are in monothreaded regions and if $pw[n_1] = wS^ju$ and $pw[n_2] = wS^kv$ where w is a common prefix, $j \neq k$, u and v words in $(P|S|B)^*$. Two nodes in monothreaded regions can be executed simultaneously if and only if they are in concurrent monothreaded regions.*

Proof. Consider two nodes n_1 and n_2 in monothreaded regions. Let w_1, w_2 be their resp. parallelism words. We first show that these nodes can be executed in parallel only if they are in concurrent monothreaded regions.

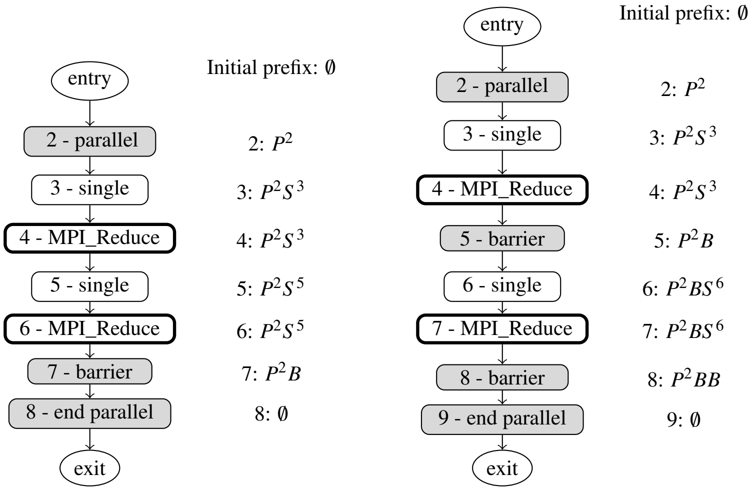
To be executed simultaneously, the two nodes need to have parallel region (a P in both parallelism words). If w is their common prefix (possibly empty), w_1 and w_2 can therefore be written either $w_1 = w.P^ib_1S^k.u$ and $w_2 = w.P^ib_2S^h.v$ (same parallel region) with b_1, b_2 words of B^*, or $w_1 = w.P^ib_1S^k.u$ and $w_2 = w.P^jb_2S^k.u$ (different parallel regions). Indeed, n_1 and n_2 are in monothreaded regions, enforcing a sequence of alternative S and P, interleaved with B.

When n_1 and n_2 are in the same parallel region (first case), the nodes can be executed simultaneously only if $b_1 = b_2$. In this case, the common prefix is wP^ib_1, corresponding to the fact that n_1 is in n_2 concurrent monothreaded region. When n_1 and n_2 are in different parallel regions (second case), there exists a h such that w ends with S^h (monothreaded region). It implies that the parallel constructs P^i and P^j are sequentially ordered. n_1 and n_2 cannot be executed simultaneously. This shows the result.

Showing that two concurrent monothreaded regions can be executed simultaneously is obvious. □

For example, the CFG in **Figure 4.5(a)** contains two collective nodes: 4 and 6 and three thread barriers colored in gray (the parallel construct is considered as a thread barrier but does not appear in parallelism words). Parallelism words of nodes 4 and 6 are $pw[4] = P^2S^3$ and $pw[6] = P^2S^5$. Then nodes 4 and 6 are in concurrent monothreaded regions respectively created by nodes 3 and 5. Both regions can be executed simultaneously.

Algorithm 7 shows the detection of concurrent collective calls. It takes as input the CFG and outputs two sets: S and S_{cc}. When collective nodes with the same number of B are detected these nodes are put in the set S and the nodes that begin the monothreaded regions are put in the set S_{cc} for the dynamic analysis. A warning is issued for nodes in S.



(a) CFG of Code 2 (b) CFG of Code 3

Figure 4.5: *Examples of CFG with monothreaded regions highlighting thread barriers*

In **Figure 4.5(a)**, nodes 4 and 6 have the same number of thread barriers in their parallelism words so the collective operations involved are potential concurrent collective calls. The algorithm outputs a warning for collective calls located nodes 4 and 6 ($S = \{4, 6\}$) and flags nodes 4 and 6 for dynamic checks ($S_{cc} = \{4, 6\}$). On the contrary, **Figure 4.5(b)**, collective nodes 4 and 7 are in different monothreaded regions separated by a thread synchronization (node 5) so the collective operations cannot be called simultaneously ($S = \emptyset$ and $S_{cc} = \emptyset$). There is no need for a dynamic check.

Detecting Category 3 Situations

Once the sequence of MPI collective calls is verified in each process we must check that all sequences are the same for all processes. For this purpose, we resort to the **Algorithm 12**. It detects MPI collective mismatches by identifying conditionals potentially leading to a deadlock (set S) and flags MPI collective operations that can deadlock (set O). A warning is also issued for collective calls located in a loop as they can be called different times if the number of iterations is not the same for all MPI processes.

For example, conditionals line 6 and 2 in Codes 4 and 5 are reported as potentially leading to a deadlock in `MPI_Allreduce`. This function is then flagged as a collective that can deadlock.

Static Pass Algorithm

To wrap-up all static algorithms, **Algorithm 19** shows how the different analyses are combined. Note that each function of the program is verified independently. First the algorithm constructs the parallelism words of the function CFG. Then the placement of MPI collective operations within a process is checked and finally the algorithm detects collective calls mismatches between processes. **Algorithm 19** issues a warning when a potential error is detected and creates sets S, S_{ipw} and S_{cc} for the following dynamic analysis.

Algorithm 19 Step 1: Static Pass of hybrid programs

```
function HYBRID_STATIC_PASS($G = (V, E), L$)
                        ▷ $G$: CFG, $L$: language of correct parallelism words

    DFS($G, entry(G)$) (Algo. 2)
                        ▷ parallelism words construction
    MULTITHREADED_REGIONS($G, L$) (Algo. 4)
                        ▷ creates sets $S_{ipw}$ and $S$
    CONCURRENT_CALLS($G$) (Algo. 7)
                        ▷ creates sets $S_{cc}$ and $S$
    STATIC_PASS($G$) (Algo. 12)
                        ▷ creates sets $O$ and $S$
end function
```

4.1.3 Static Instrumentation for Execution-Time Verification

The compile-time analyses presented in the previous section outputs warnings for MPI collective operations that may lead to an error or deadlock. Nevertheless the static analysis could lead to false positives relatively to the CFG that is possibly not correlated with the actual control flow. Besides, parallel construct can create parallel regions with only one thread, that is a sequential region. To deal with false positives results, a dynamic instrumentation is added. A unique parallelism word is computed at runtime and updated after each OpenMP construct. Compared to the compile-time parallelism words, we introduce a new letter, P_1, accounting for parallel regions created with only one thread. P_1PS is for instance a correct execution parallelism word and is differentiated from PPS.

This section describes the code transformation that checks if all warnings returned by the static analysis will eventually lead to an error.

Algorithm Description

To dynamically verify the total order of MPI collective sequences in each MPI process, validation functions are inserted in nodes in the sets S_{ipw} and S_{cc} generated by **Algorithms 4 and 7**: CC_{ipw} and CC_{cc}. These functions are depicted **Algorithm 21**. Function CC_{ipw} detects incorrect execution parallelism words and Function CC_{cc} detects concurrent collective calls.

Based on the alphabet $\{P_1, P, S, B\}$, correct execution parallelism words are in the language L_e of regular expression $(S|P_1|PB^*S)^*$ recognized by Automaton **Figure 4.4**. For each node n in S_{ipw}, if the corresponding $pw_e[n]$ is not in L_e the program stops through a call to `MPI_Abort` and an error message is returned to the programmer. For each node n in S_{cc}, a check is done to ensure the node is actually in a monothreaded region. Counting the number of threads concurrently executing a given basic block cannot be done by the simple use of `omp_get_num_thread()`. Indeed, the control flow may select only a subset of the total number of threads for the execution of a basic block. Similarly to the previous case, we resort to a shared variable *collective_lock*. This variable is used to prevent another thread from entering the region in S_{cc}. The shared variable is reset to 0 right after the barrier(s) (if any) successor of the region concerned. The intuition is indeed to reset the lock at the first barrier following the possible concurrent monothreaded regions.

To dynamically verify the total order of MPI collective sequences between processes, a check function *CC* is inserted before each MPI collective operation and before `return` statements. *CC* takes as input the communicator com_c related to the collective call c and a color i_c specific to the type of collective. As multiple threads may call *CC* before `return` statements, this function is wrapped into a `single` pragma. *CC* is depicted **Algorithm 14**. Each function of a program is instrumented by **Algorithm 20**. If an error is about to occur the program is stopped and an error message is returned with error type information.

Algorithm 20 Step 2: Selective Static Instrumentation

```
function INSTRUMENTATION($communicator, G, S, S_{ipw}, S_{cc}$)
                                   ▷ $G$: CFG, $S, S_{ipw}, S_{cc}$: sets created at compile-time

    if $S \neq \emptyset$ then
        STEP 1: Control flow errors detection
        for $n$ in nodes containing a call to collective $c$ do
            Insert call to CC($com_c, i_c$) before the call to $c$
        end for
        Before return statements insert
          if(omp_in_parallel()){
           # pragma omp single
             CC($communicator$, 0)}
         else
           CC($communicator$, 0)

        STEP 2: Collectives in multithreaded regions
        for $n \in S_{ipw}$ do
            Insert call to $CC_{ipw}()$ as the first statement of $n$
        end for

        STEP 3: Concurrent MPI calls detection
        for $n \in S_{cc}$ do
            Insert call to $CC_{cc}()$ as the first statement of $n$
            Insert $collective\_lock = 0$ after the barrier(s) successors of the region created by $n$
        end for
    end if

end function
```

Figure 4.6 presents the transformation achieved by **Algorithm 20** on CFG **Figures 4.3(a) and 4.5(a)**. The CFG **Figure 4.6(a)** has neither collective in O nor node in S_{ipw} but two potential concurrent calls. Thus only CC_{cc} functions are inserted. The CFG **Figure 4.6(b)** has no potential concurrent call. However it contains a collective call that may be called by multiple threads of a process and may not be called by all processes. Thus CC_{ipw} and CC functions are inserted.

Correctness Proof

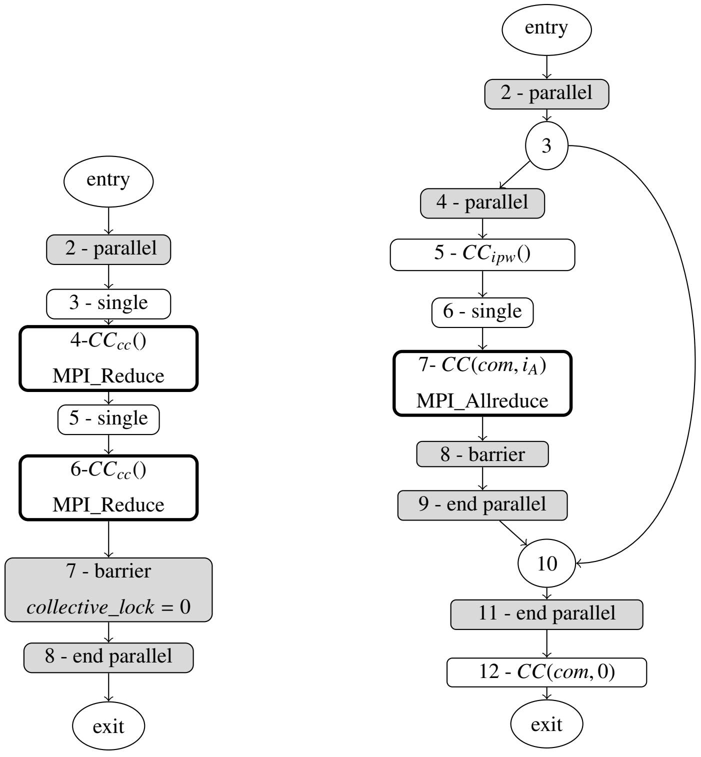
The instrumentation depicted **Algorithm 20** is correct if all **Figure 4.2** situations are captured and if the code inserted does not generate errors or deadlocks. We provide here a short correctness proof for **Algorithm 20**.

Algorithm 21 Library Functions To Check MPI collectives

```
function $CC_{ipw}$                              ▷ Detect collectives in multithreaded regions
    if $pw_e \notin L_e$ then
        MPI_Abort($com, 0$)
    end if
end function

function $CC_{cc}$                               ▷ Detect concurrent collective calls
    $CC_{ipw}$
    if $collective\_lock = 1$ then
        MPI_Abort($com, 0$)
    else
        #pragma omp atomic write
        $collective\_lock = 1$
    end if
end function

function $CC(com_c, i_c)$                        ▷ Detect collective calls mismatches
```



(a) Fig. 4.5(a) CFG instrumented

(b) Fig. 4.3(a) CFG instrumented

Figure 4.6: *Instrumented CFG* ***Figures 4.3(a) and 4.5(a) (Algorithm 11)***

If no potential error is reported from the compile-time analysis or only CC functions are inserted, no error is added and all category 3 situations are detected as proven Chapter 3.

Suppose all category situations are detected at compile-time. **Algorithm 20** inserts CC, CC_{cc} and CC_{ipw} functions. CC functions are inserted just before collective calls whereas CC_{cc} and CC_{ipw} functions are inserted as soon as possible before collective calls. CC_{cc} and CC_{ipw} are then executed before CC functions. If CC_{cc} and CC_{ipw} codes capture all errors due to the multithreaded context, CC functions do not add bugs and detect deadlock situations between MPI processes.

All threads do not have to call CC_{cc} and CC_{ipw} functions, only one thread detects that an error is going to occur in a program. CC_{ipw} prevents from the execution of a collective in a multithreaded context, *i.e.* when multiple threads are about to execute the same collective operation simultaneously. For CC_{ipw}, the second thread to execute CC_{ipw} raises an error and avoids the unordered execution of different collective operations. As the *collective_lock* variable is only reset after a barrier following the parallel region where the check occurs, this detection does not depend on the thread scheduling. Whenever a thread enters a monothreaded region containing collective calls while another thread is already executing a monothreaded region also containing a collective call the thread detects an error and stops the program. In both cases the program is stopped before an error can occur.

Finally, since the CC function uses the same communicator as the collective it precedes, if there are errors in the use of these communicators, the bugs are propagated inside the CC function. In such case, others tools such as MUST can be used to detect such situations.

4.1.4 Evaluation

This section presents experimental results obtained on the NAS Parallel benchmarks multizone (NASPB-MZ v3.2) using class B [104], a mixed mode MPI/OpenMP benchmark suite v1.0 [113, 114] (EPCC suite), HERA [108], five MPI+OpenMP Coral benchmarks and one code of the error microbenchmark suite (`coll_deadlock`).

At compile-time PARCOACH issues warnings for potential MPI collective errors within an MPI process and between MPI processes. The type of each potential error is specified (collective mismatch, concurrent collective calls in an MPI process,..) with the names and lines in the source code of MPI collective calls involved. The following example shows what a user can read on `stderr` when compiling Code 2.

```
in function 'f':
Warning: STATIC-CHECK: Concurent MPI collective calls in a process :
MPI_Reduce l.11 may be called  simultaneously with MPI_Reduce l.6
STATIC-CHECK check inserted after the single directive l.4
STATIC-CHECK check inserted after the single directive l.9
```

In this example, both single regions are instrumented as described in **Algorithm 20**.

For each benchmark, the overhead obtained at compile-time with and without code generation (corresponding to CC_{ipw} , CC_{cc} and CC functions insertions) is presented in **Figure 4.7**. This overhead is acceptable as it does not exceed 6%. **Table 4.1** shows the number of static MPI collective calls and the number of nodes in the set S found by **Algorithm 12**. The language of correct parallelism words was adapted for each benchmark depending on statistics done about the initial condition in which functions are generally called in it (see **Algorithm 4**). A data flow analysis on nodes in S as a complement to our static phase could reduce further this set as it can be shown that the control flow for all conditional nodes in S does not depend on process ranks since the benchmarks checked are correct. The 4th column depicts the percentage of the benchmarks functions instrumented. For the most part we notice a good impact of the static analysis on the selective instrumentation. The two last columns show the number of expected errors and the number of errors found by the analysis. As all benchmarks tested are correct no error was found except for the `coll_deadlock` program from the error benchmark suite.

The execution overhead obtained for each benchmark is presented in **Figures 4.8 and 4.9**. **Figures 4.8(a) and 4.9** show respectively the overhead obtained for NASPB-MZ from 1 to 16 MPI processes and 1 to 64 MPI processes with eight threads each (up to 512 threads) and the overhead obtained for the mixed mode MPI/OpenMP benchmark from 1 to 128 MPI processes with 8 threads each (up to 1024 threads with a target time of 20 sec). The overhead for the NASPB-MZ tends to sightly increase with the number of MPI processes (and thus the number of threads). The overhead remains under 25% (for an initial execution time

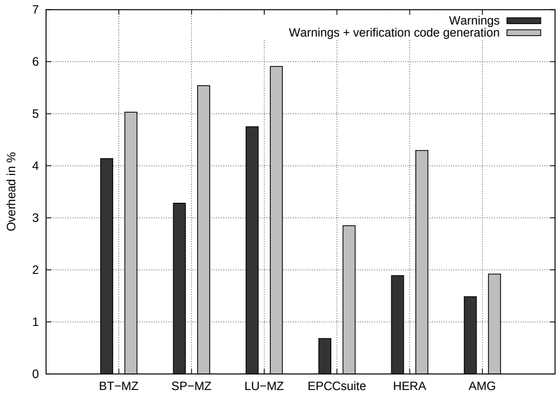


Figure 4.7: *Overhead of average compilation time with and without verification code generation*

Table 4.1: *Compilation and Execution Results*

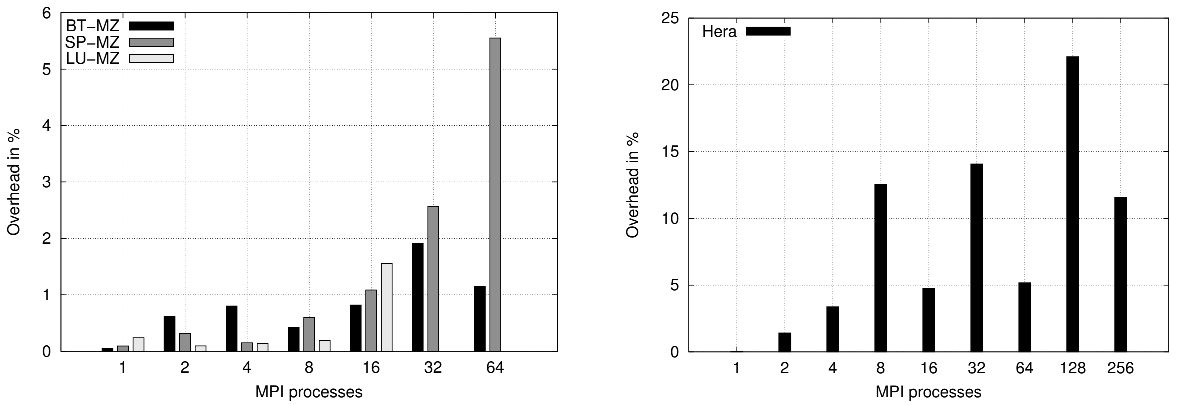
Benchmark	# coll. calls	# nodes in *S*	% functions instrumented	# expected errors	# errors found
NASPB-MZ				0	0
BT-MZ	15	7	8,57%	0	0
SP-MZ	15	7	8,57%	0	0
LU-MZ	20	7	8,82%	0	0
EPCC suite			58,2%	0	0
barrier	5	5	100%	0	0
alltoall	6	5	40%	0	0
broadcast	6	5	66,7%	0	0
scatter	7	9	66,7%	0	0
allreduce	7	9	100%	0	0
HERA	574	375	<1%	0	0
AMG2013	86	75	13.33%	0	0
LULESH	3	1	1.44%	0	0
miniFE	4	6	2.56%	0	0
HACC	26	11	1.41%	0	0
SNAP	9	13	10%	0	0
coll_deadlock	1	1	100%	1	1

of 18.36 sec for the highest overhead) for HERA. **Figure 4.8(b)** presents strong scaling results for HERA from 1 to 256 MPI processes with 8 threads per process (up to 2048 threads).

If multiple threads are about to call an MPI collective operation in a process, the program stops through a call to `MPI_Abort` and an error message is returned to the programmer with the names of the collectives and their lines in the source code. The error message reported at runtime to the programmer for Code 2 is presented below.

```
DYNAMIC-CHECK: Error detected in f (example3.c)
DYNAMIC-CHECK: Two or more threads are about to call the same MPI collective
               operation
DYNAMIC-CHECK: Abort is invoking before calling MPI_Reduce l.6 MPI_Reduce l.11
```

The amount of messages generated dynamically is reduced compared to the amount of messages generated statically. No false positive are generated in this case. Similarly, if a deadlock due to a collective mismatch is about to occur, an error is printed to stderr with the collective name, the line number and conditionals responsible. The error message reported at runtime to the programmer for Code 4 is presented bellow.



(a) Execution-Time Overhead for NASPB-MZ (Strong scaling)

(b) Execution-Time Overhead for HERA (Strong scaling)

Figure 4.8: *Execution-time overhead for HERA and NASPB-MZ Class B with 8 threads per MPI process*

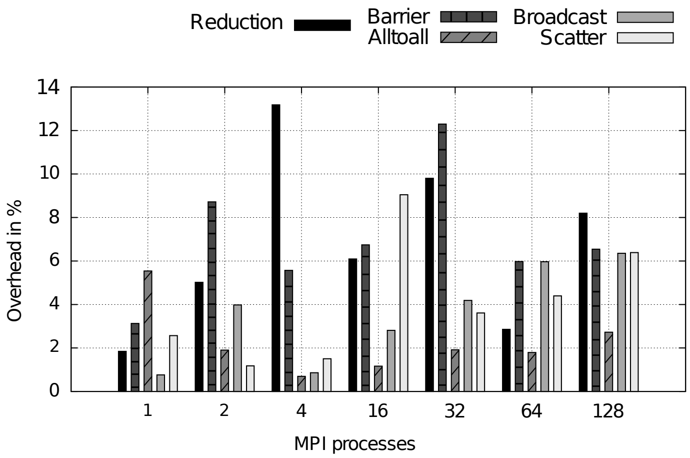


Figure 4.9: *Execution-Time Overhead for the EPCC suite (Weak scaling) (8 threads per process). The benchmark suite contains measurements for the following operations : Barrier, Reduce, Broadcast, Scatter and Alltoall.*

```
DYNAMIC-CHECK: Error detected in f (example1.c)
DYNAMIC-CHECK: Abort is invoking before calling MPI_Allreduce l.7
DYNAMIC-CHECK: see warnings about conditional line 6
```

4.2 Summary

Although large hybrid applications appear, the lack of debugging tools for hybrid programs does not encourage the development of such applications and limits the use of thread levels. This chapter presented a debugging method to overcome this issue, extending the GCC plugin PARCOACH. First PARCOACH statically identifies MPI collective operations that can deadlock or be performed by multiple non-synchronized threads. Then these potential errors/deadlocks are validated during execution by a code transformation. The cost of the runtime checks is limited by a selective instrumentation, thanks to the compile-time analysis, avoiding unnecessary checks. We have shown a small impact on performance with a compile-time overhead less than 6%. Besides with the help of the static phase, the runtime overhead remains acceptable (less than

25%) for a large application.

Chapter 5

Conclusion and Perspectives

> "Debugging is the dirty little secret of computer science".
>
> Henry Lieberman in The Debugging Scandal and What to Do About It [115]

5.1 Conclusion

Supercomputer evolution shows that exascale will be reached by 2020. This evolution advocates for hybrid solutions like MPI+OpenMP. But mixing parallel models within the same application leads to more complex codes. It is then necessary to better support users in the debugging phase of such applications. The time spent on debugging an application is generally due to the lack of an advanced support to efficiently guide debugging activities. Moreover the later errors are detected, the higher are the costs to rework the system in order to function correctly. We therefore created the static/dynamic platform named PARallel COntrol flow Anomaly CHecker (PARCOACH) that takes advantage of the compiler to enable an early detection of errors and helps HPC developers to correct them with truthful information. PARCOACH verifies HPC programs (MPI, OpenMP and MPI+OpenMP programs) in two steps:

1. It detects potential errors with an intra-procedural analysis during the compilation-time (detection per function);

2. To deal with false positives returned at compile-time, it transforms only potential erroneous codes to stop the execution whenever a deadlock is about to occur at runtime.

The method tends to better help users by returning warnings and error messages with precise information. This two-phase method was easily integrated into the GCC compiler as a plugin allowing the verification of C, C++ and Fortran applications. As a plugin, PARCOACH avoids the compiler recompilation. It is also linked to a light dynamic library for runtime checks.

Contributions Summary

These contributions have been published and presented in the European MPI Users' Group Meeting (EuroMPI) 2013 [97], the International Journal of High Performance Computing Applications (IJHPCA) 2014 [98], the International Workshop on OpenMP (IWOMP) 2014 [99], the Symposium on Principles and Practice of Parallel Programming (PPoPP) 2015 [100], the European Conference on Parallel and Distributed Computing (EuroPar) 2015 [101] and the European MPI Users' Group Meeting (EuroMPI) 2015 (to appear).

- In an MPI+OpenMP application, MPI is initialized with a so-called MPI thread level to indicate the interaction between MPI processes and OpenMP threads. The first step in the debugging process of hybrid applications is to verify if this interaction is respected. To that purpose PARCOACH helps application developers to check which interaction support is required for a specific hybrid code.

  Once the correct MPI initialization is ensured, we can check if each model is well used.

- On the MPI side, PARCOACH focuses on an important class of MPI communication errors. It verifies sequences of collective operations (blocking and non-blocking) of all MPI processes in SPMD MPI programs. PARCOACH uses the two-step method to detect mismatching collective calls between processes, assuming MPI collective calls are performed with compatible arguments.

  PARCOACH is based on the MPI semantics and is thus independent on MPI implementation. Our analysis complements existing MPI-only debugging tools.

- On the OpenMP side, PARCOACH verifies if all threads entering a function and created in it have the same sequence of worksharing regions and, depending of the verbosity level, the same sequence of barriers (verbosity 1) or the same number of barriers (verbosity 0) in OpenMP applications. It uses the graph representation defined in [78] for the compilation verification phase.

  PARCOACH is restricted to optimizations done by the compiler. PARCOACH would not work on ICC compiler.

- One would think that detecting errors in hybrid applications means to detect errors of each model within the application. However there are specific errors only resulting in the mixing of models. This is the case of MPI collective communications in multi-threaded context. PARCOACH detects bugs related to misuse of MPI collective communications in MPI SPMD programs mixed with multi-threaded model with perfect nested parallelism like OpenMP.

  This focus on collective bugs in hybrid codes is one of the key issues of a wider use of hybrid codes with MPI_THREAD_MULTIPLE. Developers are reluctant to use such level of parallelism when deadlocks difficult to find are possible, with no tool to help them (the dynamic tool Marmot can only detect errors related to a specific run in MPI+OpenMP programs).

- The creation of PARCOACH has naturally bought the development of an error benchmark suite to verify its functionnality. The error benchmark suite contains common MPI and OpenMP errors as well as non-compliance of MPI thread-level in MPI+OpenMP codes.

The main advantage of the PARCOACH method is to highlight the statements responsible for the execution path potentially leading to future deadlocks or unspecified behaviors. For all analysis, we have shown a small impact on performance with low compile-time and runtime overheads.

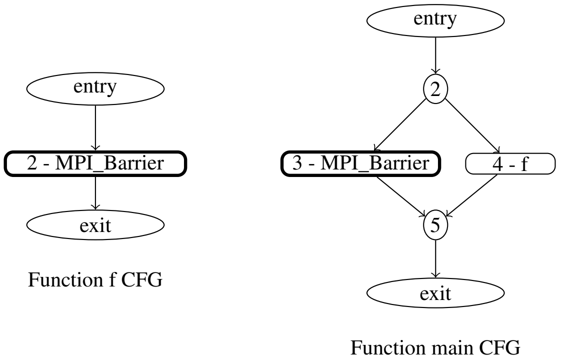
5.2 Work in progress

Our MPI verification analysis is intra-procedural as it checks a program function by function. Thus it can miss errors out of function boundaries and produce false positive and false negative. A false positive is presented at **Figure 5.1**. The figure shows a C program parallelized with MPI with two functions: `f` and `main` and the CFG associated to these functions. The MPI verification performed by PARCOACH checks `f` and `main` separately. It considers `f` as a correct function regarding MPI collective communications but returns a warning for the collective `MPI_Barrier` in function `main` ($C_{0,barrier}$ = {3} and $PDF^+(C_{0,barrier})$ = {2}). However a `MPI_Barrier` is hiding behind function `f`. This means no matter the path taken in function `main` CFG, all MPI processes will encounter a barrier. PARCOACH could avoid emitting the warning by knowing this information.

Code

```
MPI_Comm CommA;
...
int f()
{
    ...
    MPI_Barrier(CommA);
    ...
}

int main(int argc, char **argv)
{
    ...
    if()
       f();
    else
       MPI_Barrier(CommA);
    ...
    return 0;
}
```



Function f CFG

Function main CFG

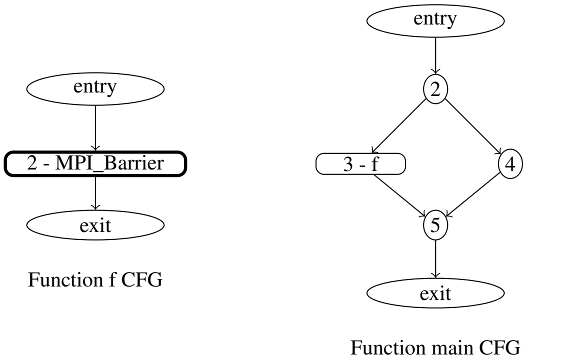
Figure 5.1: *Example of a false positive.*

Figure 5.2 depicts an example of a false negative. The figure shows the same previous C program parallelized with MPI but without the `MPI_Barrier` in the else branch in function `main`. In this example, no warning is returned by PARCOACH. However all MPI processes may not encounter the barrier in function `f` because of the conditional node 2 function `main`. Likewise the previous example, reporting this information would help PARCOACH to manage this situation.

Code

```
MPI_Comm CommA;
...
int f()
{
    ...
    MPI_Barrier(CommA);
    ...
}

int main(int argc, char **argv)
{
    ...
    if()
       f();
    else
       ...;
    ...
    return 0;
}
```



Function f CFG

Function main CFG

Figure 5.2: *Example of a false negative.*

To extend PARCOACH analyses into inter-procedural analyses and then have a global view of a program, we use a callgraph (CG). In a callgraph, nodes represent procedures in the program and edges represent possible calls. An edge $f \longrightarrow g$ then depicts a call to g in f. Each node in CG has a set of successors $SUCC_{CG}$ and a set of predecessors $PRED_{CG}$ that can be empty [102]. For example, the CG of the two previous programs contains two nodes (`f` and `main`) and one edge ($main \longrightarrow f$).

The simplest approach to extend PARCOACH is to use methods like inlining or cloning. The inlining process replaces callee functions by their entire code in their caller functions. This avoids the loss of information between callee and caller functions. When the Control Flow Graph (CFG) of a function is built, it then contains the CFG of each callee functions. An additional method is cloning. It estimates the frequency of functions calls and builds a simplified version of functions related to the context in which they are called. It is then possible to clone functions with their most frequent value. Although simple, these solutions have a high complexity.

Another solution is to replace each callee function by a summary [116]. This raises the issue of finding what is important to keep about each function. This is currently explored during the internship of Hugo Brunie at CEA.

First approach

The idea is to compute and reuse summaries of functions through a traversal of a program callgraph. There are two possible solutions for the CG traversal: top-down and bottom-up analyses. A top-down analysis starts at root procedures of a program and proceeds from callers to callees whereas a bottom-up analysis begins from leaf procedures and proceeds from callees to callers [117,118]. As PARCOACH analyses each function independently, the more intuitive solution is to use a bottom-up approach. This provides an analysis of all contexts in which functions are called.

The intra-procedural analysis performed by PARCOACH is changed in order to return the summary needed for the inter-procedural analysis. Summaries should contain just enough information to be helpful and not overload functions. For a first approach, we choose to keep the valid sequence of collective operations of each function. The valid sequence of collective operations corresponds to collective operations all MPI processes will encounter for sure. The algorithm now returns the set of conditional nodes potentially leading to a deadlock situation O and the valid sequence of collective operations $ValidSeq$. **Algorithm 22** describes the first approach to an inter-procedural analysis. In this approach, the summary of a function includes the valid sequence of MPI collective calls. The algorithm first analyses functions that do not call any function, then callers of these functions and so on. For every function analysed, callee functions are replaced by the valid sequence of collective calls they contain. To handle recursivity, we can limit the number of the recursive function calls to one. We assume potential errors can be found after one step of the analysis.

For the example presented **Figure 5.1**, **Algorithm 22** first analyses function `f`. The *INTRAPROCEDURAL_ANALYSIS* procedure is called on `f` and returns the set ($O_f = \emptyset, ValidSeq = \{MPI\_Barrier\}$). Then the *INTERPROCEDURAL_ANALYSIS* procedure sets $f.ValidSeq = \{MPI\_Barrier\}$ and $O = \emptyset$. During the inter-procedural analysis of function `main`, `f` is replaced by $\{MPI\_Barrier\}$. Thanks to this replacement, no warning is emitted for function `main`. The same is applied to **Figure 5.2**. `f` is also replaced by $\{MPI\_Barrier\}$ in function `main`. The *INTRAPROCEDURAL_ANALYSIS* procedure then returns a warning to the user for the collective.

Second approach

The first approach reduces the number of false positives returned by PARCOACH and finds potential errors impossible to detect and locate with an intra-procedural approach. However the first inter-procedural analysis is a bit light. **Figure 5.3** illustrates this point. The possible deadlock in `MPI_Allreduce` reported by the intra-procedural analysis in function `g` is not reported in `f`. However this information could be useful

Algorithm 22 Inter-procedural anlysis: first solution

```
function INTRAPROCEDURAL_ANALYSIS($CFG$)                          ▷ $CFG$: Control Flow Graph
    $O \leftarrow \emptyset$
    $ValidSeq \leftarrow \emptyset$                                              ▷ Output set
    Remove loop backedges in $G$ to compute execution orders for nodes with collectives
    for $r$ in node orders do
        for $c$ in collective names of execution order $r$ do
            $C_{r,c} \leftarrow \{u \in V | r$ is the max. execution order of $u$, $u$ executes a collective with name $c\}$
            if $PDF^+(C_{r,c}) \neq \emptyset$ then
                $O \leftarrow O \cup (c, PDF^+(C_{r,c}))$
            else
                $ValidSeq \leftarrow ValidSeq \cup \{c\}$
            end if
        end for
    end for
    return $(O, ValidSeq)$
end function

function INTERPROCEDURAL_ANALYSIS($\bigcup_f CFG_f, CG$)     ▷ $CFG$: Control Flow Graph, $CG$: Callgraph
    $Seq \leftarrow \{\}$ , $O \leftarrow \emptyset$
    for each $n \in CG$ in reverse topological order do
        $n.ValidSeq \leftarrow \{\}$
        $O_n \leftarrow \emptyset$
        for each $f \in SUCC_{CG}(n)$  do
            Replace $f$ in n by $f.ValidSeq$
        end for
        $(O_n, ValidSeq) \leftarrow$ INTRAPROCEDURAL_ANALYSIS($CFG_n$)
        $n.ValidSeq \leftarrow ValidSeq$
        $O \leftarrow O \cup O_n$
    end for
    return $O$
end function
```

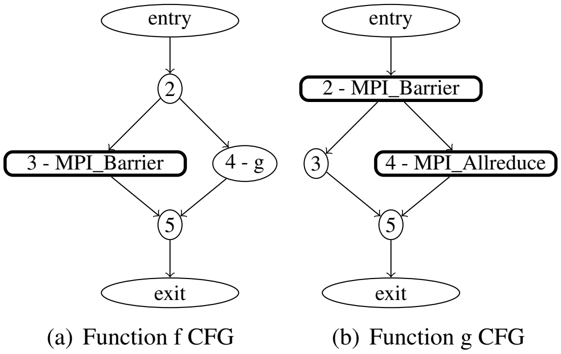
for debugging.

Reporting the entire CFG can be so expensive (see **Table 5.1**) that we resort to report a reducted CFG. This reducted CFG contains only collective nodes and nodes in their iterated postdominance frontier. As an example, **Figure 5.4(a)** shows the reducted CFG of function g. g contains two collective nodes (nodes 2 and 4) and one node in the iterated postdominance frontier of node 4 (node 2). **Algorithm 23** depicts the second solution adopted for the inter-procedural analysis. Each callee function is now replaced by its CFG reducted as presented **Figure 5.4(b)** where g is replaced by its reducted CFG.

Static instrumentation The intra-procedural analysis of PARCOACH allows a selective instrumentation. Whenever a potential deadlock is found, all collectives in the function are instrumented: validation functions are inserted before MPI collective functions and before return statements. With the inter-procedural analysis this is no longer possible. Whenever a potential deadlock is found, all collectives in the program are instrumented. This boilds down to instrument a program as dynamic tools like MUST do. Yet validation functions could be inserted at divergence points to avoid systematic instrumentation. Validation functions may compare MPI collective sequences of each path directly from divergent points. This instrumentation enables a deadlock detection as soon as possible at runtime.

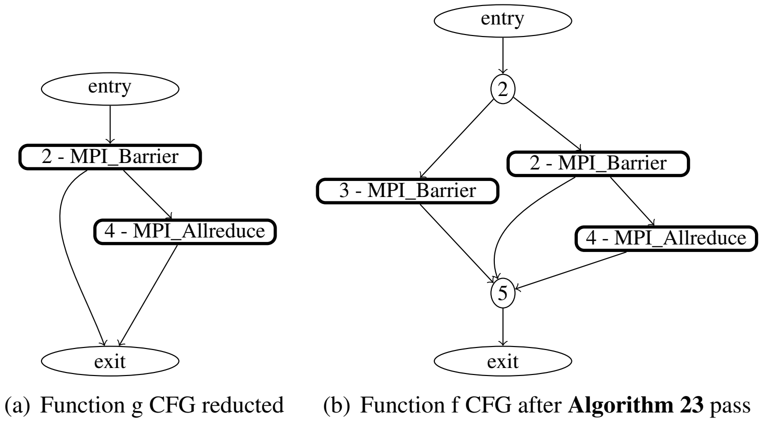
Benchmark	**Number of CFG nodes** **Max.**	 **Min.**	**Number of functions**
EulerMHD	391	3	4 847
NASPB-MPI			
BT	135	3	55
CG	57	3	11
FT	63	3	32
MG	77	3	26
SP	147	3	27
DT	64	3	27
IS	67	6	6
EP	49	3	2
LU	147	3	30

Table 5.1: *Statistics on EulerMHD and NASPB-MPI v.3.3 class B.*



(a) Function f CFG (b) Function g CFG

Figure 5.3: *Example of a problematic case for the first inter-procedural approach.*



(a) Function g CFG reducted (b) Function f CFG after **Algorithm 23** pass

Figure 5.4: *The applied second inter-procedural approach.*

Application to OpenMP and MPI+OpenMP applications This process is made out of GCC. That way, CFGs are not really modified. We suggest two approaches to the inter-procedural analysis of MPI programs. Results on real benchmarks and applications are missing to validate the approaches. The same principle can be applied to OpenMP and MPI+OpenMP applications. Only functions summaries will change.

Algorithm 23 Inter-procedural analysis: second solution

```
function INTRAPROCEDURAL_ANALYSIS($CFG$)                              ▷ $CFG$: Control Flow Graph
    $O \leftarrow \emptyset$
    $CFGreducted \leftarrow \emptyset$                                                   ▷ Output set
    Remove loop backedges in $G$ to compute execution orders for nodes with collectives
    for $r$ in node orders do
        for $c$ in collective names of execution order $r$ do
            $C_{r,c} \leftarrow \{u \in V | r$ is the max. execution order of $u$, $u$ executes a collective with name $c\}$
            if $PDF^{+}(C_{r,c}) \neq \emptyset$ then
                $O \leftarrow O \cup (c, PDF^{+}(C_{r,c}))$
            end if
        end for
    end for
    $CFGreducted \leftarrow$ COMPUTE_CFG_REDUCTED($CFG$)
    return $(O, CFGreducted)$
end function

function INTERPROCEDURAL_ANALYSIS($\bigcup_f CFG_f, CG$)      ▷ $CFG$: Control Flow Graph, $CG$: Callgraph
    $Seq \leftarrow \{\}$ , $O \leftarrow \emptyset$
    for each $n \in CG$ in reverse topological order do
        $O_n \leftarrow \emptyset$
        for each $f \in SUCC_{CG}(n)$ do
            Replace $f$ in n by $(CFGreducted_f)$
        end for
        $(O_n, CFGreducted_n) \leftarrow$ INTRAPROCEDURAL_ANALYSIS($CFG_n$)
    end for
    return $O_{main}$
end function
```

5.3 Perspectives

Further work on PARCOACH would largely improve the platform for debugging purpose. We first consider short-term improvements and then more general perspectives of this work.

5.3.1 Short-term Improvements

We have noted some limitations of PARCOACH (e.g., not handling MPI functions arguments in MPI programs, critical sections and locks in OpenMP programs). This section presents short-term improvements that could reduce those limitations.

(a) Data-flow analysis

Some warnings returned by PARCOACH at compile-time are false positives as the conditionals reported do not involve processes rank variables. This is for example the case of the warning presented page 59. The code associated to this warning is depicted **Figure 5.5**. The `comm_size` variable line 6 has the same value for all MPI processes and is thus independent on processes rank. A data-flow analysis regarding parallel-related variables could enhance the static control-flow analysis done in PARCOACH by reducing the amount of useless warnings. This could be done by tagging variables depending on MPI rank and OpenMP thread ID. **Figure 5.6** (on the left) suggests a pragma-based approach. The variable `comm_size` is tagged as CONST PARALLEL which means it does not depend on MPI rank. All conditionals on non-tagged variables would be removed from the set of conditionals that can potentially lead to a deadlock situation. This is all the more interesting with an inter-procedural analysis.

Function main in is.c (original lines: l.923-930 | l.994)

```
int main( int argc, char **argv )
{
    ...

    /*  Check to see whether total number of processes is within bounds.
        [...]                                                          */
    if  ( comm_size > MAX_PROCS)
    {
       if( my_rank == 0 )
           printf( \n ERROR: number of processes %d exceeds maximum %d
                   \n Exiting program!\n\n, comm_size,  MAX_PROCS);
       MPI_Finalize();
       exit( 1 );
    }
    ...
    /*  End of timing, obtain maximum time of all processors */
    MPI_Reduce( &timecounter,&maxtime,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD );

   ...
}
```

Figure 5.5: *Function main of the NASPB-MPI IS v3.2.*

Function main in is.c

```
int main( int argc, char **argv )
{
    ...
    CONST PARALLEL comm_size;
    if  ( comm_size > MAX_PROCS)
    {
       ...
    }
   ...
}
```

MPI collective calls on different communicators

```
if( ... ){
    MPI_Barrier(com1);
}
else{
    MPI_Barrier(com2);
}
```

Figure 5.6: *Pragma-based approach in function main of the NASPB-MPI IS v3.2 and Example of MPI communications on different communicators (com1 and com2).*

(b) Cover more MPI and OpenMP verification

PARCOACH is focused on the detection of deadlocks related to MPI collective communications, OpenMP barriers and worksharing constructs. It would also be interesting to extend PARCOACH to detect problems that occur with MPI point-to-point functions.

For our analyses, we supposed all MPI collective communications are called with compatible arguments. PARCOACH currently returns no warning for the portion of code presented **Figure 5.6** (on the right). With a data-flow analysis, this assumption is cleared up and collective communications arguments can be checked (communicators, root, ...).

5.3.2 General Perspectives

HPC applications are still favourable for errors either because of new features that make programming models more complex or because of the adoption of hybrid solutions. Thus debugging HPC applications remains crucial and challenging. It is important to continue working further on new solutions to help developers. This section gives general perspectives to continue in that direction.

(a) Extensions for parallel programming models

Although applied to MPI and OpenMP models the PARCOACH framework can be applicable to any models with collective constraints. Like MPI, UPC requires that the order of collective operations must be the same on all threads/processes. UPC-CHECK [119] and UPC-SPIN [120] are the only tools that can detect deadlocks in UPC programs. UPC-CHECK is based on an algorithm using a distributed shared WFG to detect deadlocks in collective operations (order and consistency of arguments) and locks.

Besides, beyond parallelism issue, PARCOACH can detect any semantic problems. Since an order should be guaranteed (an instruction proceeds before another one), PARCOACH can be used to ensure the order is respected and find potential divergence. For example, PARCOACH can be used to verify sequences of lock-/unlock in OpenMP applications or for each MPI non-blocking communication that there exists at least a completion function (`MPI_Wait`, `MPI_Test` or their {*All, some, any*} derivated flavor) associated. **Figure 5.7(a)** shows a CFG with one non-blocking collective and one completion call. In this example, there may be a problem with the completion call.

(b) Full Integration into an existing tool

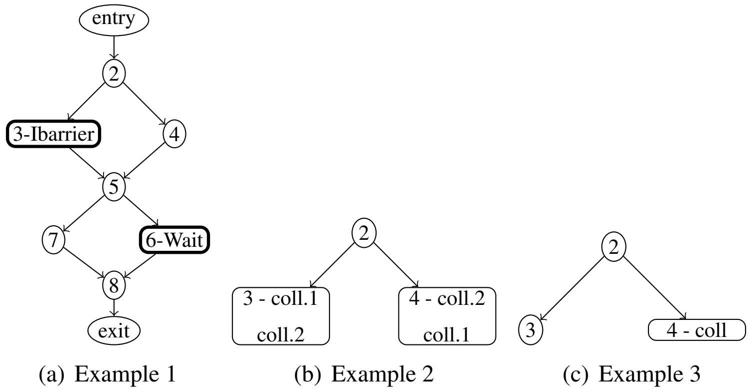
As a plugin, PARCOACH can be combined with an existing tool for early detection of bugs and verification at runtime. For that we might expect a compilation with PARCOACH and an execution with a dynamic debugging tool. PARCOACH could be integrated in existing tools like Marmot or MUST. Besides, our static analysis can be seen as a valuable advantage to integrate in a dynamic tool by reporting warnings concerning the execution path responsible for bugs.

It could be interesting to go further with the integration and enable a full interaction between PARCOACH and a dynamic tool. The selective instrumentation we propose could help the dynamic tool to reduce the amount of checks during the execution. MUST would have the advantage of warnings returned by our static check and all debug information (line of the cause of a deadlock,...) to avoid systematic instrumentation. Thus validation functions inserted by PARCOACH for runtime checking could also be interpreted by the dynamic tool. Associated with MUST, PARCOACH does not need to insert validation functions before return statements as MUST can detect MPI processes calling `MPI_Finalize` while other MPI processes calls for another MPI function.

(c) Adaptation to other compilers

PARCOACH was integrated in the GCC compiler but could be integrated in all compilers using the CFG representation like the LLVM compiler framework [121]. LLVM (Low Level Virtual Machine) is an open-source compilation platform that competes with GCC in terms of compilation speed and performance of the generated code. As GCC, LLVM allows compiler extensions through new LLVM passes.

Going further with the compiler help, whenever PARCOACH finds a potential error at compile-time, the compiler could help to reorder instructions. For instance, the compiler could switch collective statements node 3 in **Figure 5.7(b)** or append `coll` node 3 in **Figure 5.7(c)**.



(a) Example 1 (b) Example 2 (c) Example 3

Figure 5.7: *Examples for semantic problems detection ((a)) and errors correction by compilers ((b) and (c)).*

(d) Improvement of the Error Benchmark Suite

The error benchmark suite could provide a large and complete comparison of debugging tools in term of bugs detection. We may expand the work done in [60] with more tools and parallel programming models. The benchmark suite could also be used to test new features of models and find new possible errors.

Appendices

Appendix A

GCC, the GNU Compiler Collection

GCC is an integrated distribution of compilers for several major programming languages (C, C++, Objective-C, Objective-C++, Java, Fortran, Ada and Go). GCC is a part of the GNU project and aims at improving the compiler used in the GNU system including the GNU/Linux variant. Most of the containt here is from [122].

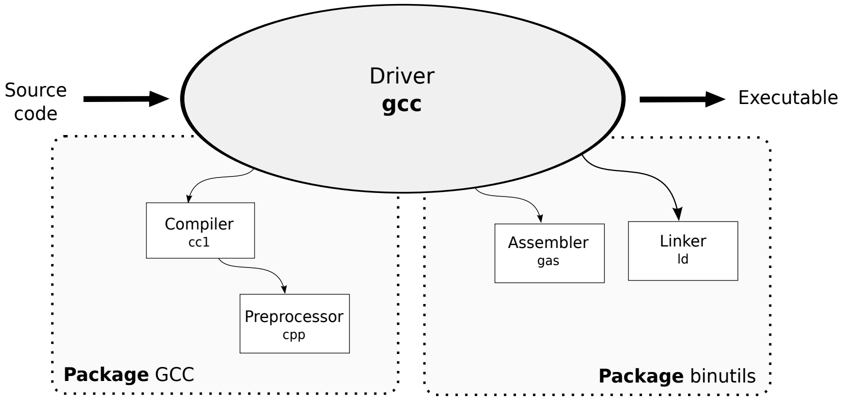


Figure A.1: *Architecture of GCC*

A.1 Structure of GCC

As most compilers, a compilation with GCC is split in three phases: the Front End, the Middle End and the Back End (see **Figure A.2**).

The Front End takes the source code and does whatever is needed to translate that source code into a semantically equivalent, language independent abstract syntax tree (AST). The syntax and semantics of this AST are defined by the GIMPLE language. GIMPLE is the highest level language independent intermediate representation GCC has.

The AST is then run through a list of target independent code transformations that take care of such things as constructing a control flow graph, and optimizing the AST for optimizing compilations, lowering to non-strict RTL and running RTL based optimizations for optimizing compilations. The non-strict RTL is handed over to more low-level passes.

The low-level passes are the passes that are part of the code generation process. These passes turn the non-strict RTL representation into strict RTL. Strict RTL passes include scheduling, doing peephole optimizations, and emitting the assembly output.

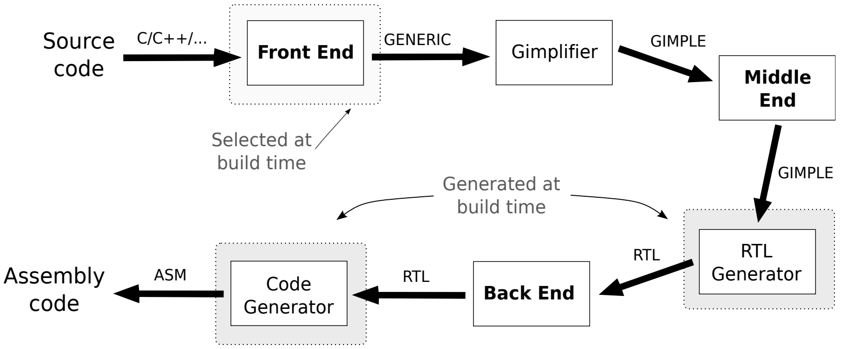


Figure A.2: *Architecture of GCC*

A.2 GCC's history

The very first beta of GCC, then known as the GNU C Compiler, was released on 22 March 1987. Since then, many improvements were made. GCC has grown to support more languages, more architectures and has many optimisations. This section gives a GCC timeline with general improvements done in the different releases of GCC.

- **0.9**: March 22, 1987
  - First beta release
- **GCC 1.0**: May 23, 1987
- **GCC 3.0**: June 18, 2001
  - JAVA support in GCC
- **GCC 4.0**: April 20, 2005
  - Merge of the tree ssa branch
  - Swing Modulo Scheduling (SMS), an RTL level instruction scheduling optimization intended for loops
  - Intermediate representation GIMPLE
- **GCC 4.2.0**: May 13, 2007
  - OpenMP support for the C, C++ and Fortran compilers
- **GCC 4.5.0**: April 14, 2010
  - New link-time optimizer (LTO)
  - Plugins to extend the compiler without modifying its source code
- **GCC 4.6.0**: March 25, 2011
  - Better memory usage and cache locality
  - Support for the Go and CAF programming languages
- **GCC 4.7.0**: March 22, 2012
  - OpenMP 3.1

    - C++11 standard

- **GCC 4.8.0**: March 22, 2013
    - GCC now uses C++ as its implementation language
    - Full support of C++11 standard
- **GCC 4.9.0**: April 22, 2014
    - OpenMP 4.0
    - Experimental support for C++14
    - Intel AVX-512 support (inline assembly support, new registers and extending existing ones, new intrinsics, and basic autovectorization)
    - Complete implementation of the Go 1.2.1 release
- **GCC 5.0**: December 23, 2014
    - Support for all C++14 language features
    - Preliminary implementation of the OpenACC 2.0a specification
- Current version: **GCC 5.1** released in April 22, 2015

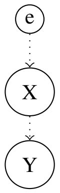
Appendix B

Key concepts

G denotes a *Control Flow Graph* (CFG) of a program. V is the set of vertex and E the set of edges. Entry and exit nodes of the CFG are respectively denoted by e and s. X and Y are nodes in the CFG. Each node X has a set of predecessors (preds(X)) and a set of successors (succs(X)). Most of the definitions are from *Efficiently Computing Static Single Assigment Form and the Control Dependence Graph* [109] and [102,103].

B.1 Dominance/Postdominance

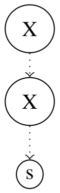
Definition 7. *X dominates Y if X appears on every path from Entry to Y. We write* $X \underline{\gg} Y$. *Each node dominates itself.*



$$Dom(e) = e$$

$$Dom(Y) = \Big(\bigcap_{p \in preds(Y)} Dom(p)\Big) \cup \{Y\}$$

Definition 8. *X postdominates Y if X appears on every path from Y to Exit. We write* $X \underline{\gg_p} Y$. *Each node postdominates itself.*



$$PDom(Y) = \Big(\bigcap_{p \in succs(Y)} PDom(p)\Big) \cup \{Y\}$$

Definition 9. *If X dominates Y and* $X \neq nY$ *then X strictly dominates Y. We write* $X \gg Y$.

$$sDom(Y) = Dom(Y) - \{Y\}$$

Definition 10. *If X postdominates Y and* $X \neq nY$ *then X strictly postdominates Y. We write* $X \gg_p Y$.

$$sPDom(Y) = PDom(Y) - \{Y\}$$

Definition 11. *The immediate dominator of Y is the closest strict dominator of Y on any path from Entry to Y.*

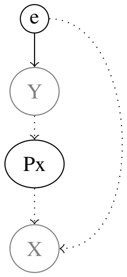
$$iDom(Y) = \{X | X \in sDom(Y) \text{and} \forall p \in sDom(Y) \text{with} p \neq= X, X \neq\in sDom(p)\}$$

Definition 12. *The immediate postdominator of Y is the closest strict postdominator of Y on any path from Y to Exit.*

$$iPDom(Y) = \{X | X \in sPDom(Y) \text{and} \forall p \in sPDom(Y) \text{with} p \neq= X, X \neq\in sPDom(p)\}$$

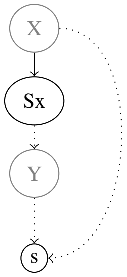
B.2 Dominance/Postdominance Frontier

Definition 13. *The dominance frontier DF(Y) of a node Y is the set of all nodes X such that Y dominates a predecessor of X but does not strictly dominate X.*



$$DF(Y) = \{X | \exists Px \in preds(X), Y \underline{\gg} Px \text{and} Y \neq\gg X\}$$

Definition 14. *The postdominance frontier PDF(Y) of a node Y is the set of all nodes X such that Y postdominates a successor of X but does not strictly postdominate X.*



$$PDF(Y) = \{X | \exists Sx \in succs(X), Y \underline{\gg_p} Sx \text{and} Y \neq\gg_p X\}$$

Appendix C

Details on benchmarks

We validated the work done in this thesis on several benchmarks and production applications: the NAS Parallel Benchmarks, EulerMHD, Coral Benchmarks, HERA, EPCC and the error benchmark suite created in the scope of this thesis.

C.1 NAS Parallel benchmarks

The NAS Parallel Benchmarks (NPB) [104] were developed by the Numerical Aerodynamic Simulation group at the National Aeronautic and Space Administration (NASA). The benchmarks are a small set of C and Fortran programs derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications in the original "pencil-and-paper" specification (NPB 1). The benchmark suite has been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids.

The original eight benchmarks specified in NPB 1 mimic the computation and data movement in CFD applications:

- five kernels
  - IS - Integer Sort, random memory access
  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient, irregular memory access and communication
  - MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT - discrete 3D fast Fourier Transform, all-to-all communication
- three pseudo applications
  - BT - Block Tri-diagonal solver
  - SP - Scalar Penta-diagonal solver
  - LU - Lower-Upper Gauss-Seidel solver

Multi-zone versions of NPB (NPB-MZ) are designed to exploit multiple levels of parallelism in applications and to test the effectiveness of multi-level and hybrid parallelization paradigms and tools. There are three types of benchmark problems derived from single-zone pseudo applications of NPB:

- BT-MZ - uneven-size zones within a problem class, increased number of zones as problem class grows

- SP-MZ - even-size zones within a problem class, increased number of zones as problem class grows
- LU-MZ - even-size zones within a problem class, a fixed number of zones for all problem classes

The extension of the benchmark suite includes unstructured computation, parallel I/O, and data movement.

- UA - Unstructured Adaptive mesh, dynamic and irregular memory access
- BT-IO - test of different parallel I/O techniques
- DC - Data Cube
- DT - Data Traffic

C.2 Coral

The Coral benchmark suite is composed of scalable science benchmarks (LSMS, QBOX, HACC and Nekbone), throughput benchmarks (CAM-SE, UMT2013, AMG2013, MCB, QMCPACK, NAMD, LULESH, SNAP and miniFE), data-centric benchmarks (Graph500, Integer Sort, Hash and SPECint2006), skeleton benchmarks (CLOMP, IOR, CORAL MPI benchmarks, Memory benchmarks, LCALS, Pynamic, HACC IO, FTQ, XSBench and MiniMADNESS) and microkernel benchmarks (NEKbonemk, HACCmk, UMTmk, AMGmk, MILCmk and GFMCmk). In this thesis we tested the following five Coral benchmarks:

- HACC: Compute intensity, random memory access, all-to-all communication. (written in C++)
- AMG2013: Algebraic Multi-Grid linear system solver for unstructured mesh physics packages. (written in C)
- LULESH: Shock hydrodynamics for unstructured meshes. Fine-grained loop level threading. (written in C)
- SNAP: Deterministic radiation transport for structured meshes. (written in Fortran)
- miniFE: Finite element code. (written in C++)

The HACC (Hardware Accelerated Cosmology Code) framework uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. The main scientific goal is to simulate the evolution of the Universe from its early times to today and to advance our understanding of dark energy and dark matter, the two components that make up 95% of our Universe [123].

AMG is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [124].

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a shock hydro mini-app. It performs a hydrodynamics stencil calculation using both MPI and OpenMP to achieve parallelism [125].

SNAP serves as a proxy application to model the performance of a modern discrete ordinates neutral particle transport application. This benchmark stresses the memory subsystem and total memory capacity. It also has the ability to use newer MPI and OpenMP features, such as nested threads and thread multiple communication when available [126].

miniFE is a Finite Element mini-application which implements a couple of kernels representative of implicit finite-element applications [127].

Bibliography

[1] Michael J. Flynn and Kevin W. Rudd. Parallel Architectures. *ACM Comput. Surv.*, 28(1):67–70, March 1996. pages 9

[2] C.G. Morris and Academic Press. *Academic Press Dictionary of Science and Technology*. Academic Press, 1992. pages 9

[3] TOP 500 List. http://www.top500.org/lists/. pages 9, 11, 117

[4] TOP 500 - Tera 100 supercomputer. http://top500.org/system/10589. pages 10

[5] TOP 500 - Curie supercomputer. http://top500.org/system/177818. pages 10

[6] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999. pages 11, 16, 20

[7] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. http://www.mpi-forum.org/docs/docs.html. pages 12, 24, 33, 40, 41, 52

[8] MPI Tutorials. https://computing.llnl.gov/tutorials/mpi/. pages 13, 115

[9] MPICH. https://www.mpich.org/. pages 13

[10] Openmpi : Open source high performance computing. http://www.open-mpi.org/. pages 13

[11] Intel MPI Library. http://software.intel.com/en-us/intel-mpi-library. pages 13

[12] Bullx Cluster Suite - Application Developer @ Ys Guide. 2.1-MPIBull2. pages 13

[13] IBM Platform MPI. http://www-03.ibm.com/systems/platformcomputing/products/mpi/. pages 13

[14] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In Emilio Luque, Tomàs Margalef, and Domingo Benitez, editors, *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, volume 5168 of *Euro-Par '08*, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag. pages 13

[15] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 94–103. Springer Berlin Heidelberg, 2009. pages 13

[16] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In Sato et al. [128], pages 1–14. pages 13

[17] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications. In Chapman et al. [129], pages 80–93. pages 13

[18] The MultiProcessor Computing Framework download page. http://mpc.paratools.com/Download. pages 13

[19] William D. Gropp. Learning from the Success of MPI. In Burkhard Monien, ViktorK. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing — HiPC 2001*, volume 2228 of *Lecture Notes in Computer Science*, pages 81–92. Springer Berlin Heidelberg, 2001. pages 14

[20] PGAS : Partitioned Global Address Space. http://www.pgas.org. pages 14

[21] Unified Parallel C. http://upc.gwu.edu/. pages 14

[22] Titanium. http://titanium.cs.berkeley.edu/. pages 14

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005. pages 14

[24] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. pages 14

[25] Berkeley unified parallel C project. http://upc.lbl.gov. pages 14

[26] GNU unified parallel C. http://www.gccupc.org. pages 14

[27] POSIX threads. https://computing.llnl.gov/tutorials/pthreads/. pages 14

[28] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). System Interfaces. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Syst*, 2004. pages 14

[29] The OpenMP API specification for parallel programming - version 4.0. http://www.openmp.org, 2008. pages 15, 61, 62

[30] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. pages 15

[31] OpenAcc Directives for Accelerators. http://www.openacc.org. pages 16

[32] NVIDIA Corporation. *NVIDIA CUDA, Compute Unified Device Architecture Programming Guide, v5.0*, 2012. pages 16

[33] OpenCL - the open standard for parallel programming of heterogeneous systems. http://khronos.org/opencl. pages 16

[34] K. G. Wilson. Grand Challenges to Computational Science. *Future Gener. Comput. Syst.*, 5(2-3):171–189, September 1989. pages 16

[35] DC Department of Energy, Washington. Exascale Workshop Panel Meeting Report. page 46, January 2010. pages 16

[36] Jack Dongarra, Pete Beckman, and al. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011. pages 16

[37] Stephen S. Pawlowski. Exascale science: the next frontier in high performance computing. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 1–1, New York, NY, USA, 2010. ACM. pages 16

[38] DC Department of Energy, Washington. Challenges in climate change science and the role of computing at the Extreme Scale. page 98, November 2008. pages 16

[39] Ewing Lusk and Anthony Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *Intl. Conf. on OpenMP in a New Era of Parallelism*, pages 36–47, Berlin, Heidelberg, 2008. Springer-Verlag. pages 17

[40] A. Zendler. *Advanced Concepts, Life Cycle Models and Tools for Object-oriented Software Development*. Reihe Softwaretechnik. Tectum, 1997. pages 18

[41] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. pages 18

[42] Barry W. Boehm. Spiral Development: Experience, Principles, and Refinements. In Wilfried J. Hansen, editor, *Spiral Development Workshop*, Pittsburgh, 2000. pages 19

[43] Richard W. Selby. *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research (Practitioners)*. Wiley-IEEE Computer Society Pr, 2007. pages 19

[44] M. Stitt. *Debugging: Creative Techniques and Tools for Software Repair*. Wiley professional computing. Wiley, 1992. pages 20, 23

[45] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009. pages 21, 22

[46] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Instut für Technische Informatik und Telematik, 2000. pages 22, 115

[47] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989. pages 21

[48] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Runtime Checking of MPI Applications with MARMOT. ParCo, 2005. pages 23, 24, 58

[49] Stephen F. Siegel and Ganesh Gopalakrishnan. Formal analysis of message passing. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 2–18. Springer Berlin Heidelberg, 2011. pages 24, 26

[50] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/gdb.html`. pages 24

[51] Totalview debugger. `http://www.roguewave.com/products-services/totalview`. pages 24

[52] Allinea DDT Debugger. `http://www.allinea.com/products/ddt/features`. pages 24, 25

[53] Stephen Siegel and Timothy Zirkel. Automatic Formal Verification of MPI Based Parallel Programs. In *PPoPP*, pages 309–310, 2011. pages 25

[54] Stephen F. Siegel. Verifying Parallel Programs with MPI-Spin. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007. pages 25

[55] MPI-Checker. https://github.com/0ax1/MPI-Checker. pages 26

[56] Clang Static Analyzer. http://clang-analyzer.llvm.org/. pages 26

[57] Greg 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. pages 26

[58] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de de Supinski, Martin Schulz, and Greg Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. pages 26

[59] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI Analysis and Checking Tool. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter, editors, *PARCO*, volume 13 of *Advances in Parallel Computing*, pages 493–500. Elsevier, 2003. pages 26

[60] Subodh Sharma, Ganesh Gopalakrishnan, and Robert M; Kirby. A survey of MPI related debuggers and tools. 2007. pages 26, 94

[61] Jeffrey S. Vetter and Bronis 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. pages 26

[62] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a Tool for Checking Fortran 90 MPI Programs. Concurrency and Computation: Practice and Experience, pages 15:93–100, 2003. pages 26

[63] T. Hilbrich, M. Schulz, B.R. de Supinski, and M. Müller. MUST: A scalable approach to runtime Error Detection in MPI programs. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Parallel Tools Workshop*, pages 53–66. Springer, 2009. pages 26

[64] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. pages 26

[65] Tobias Hilbrich, Fabian Hänsel, Martin Schulz, Bronis R. de Supinski, Matthias S. Müller, and Wolfgang E. Nagel. Runtime MPI Collective Checking with Tree-Based Overlay Networks. In *European MPI Users' Group Meeting*, pages 129–134, New York, NY, USA, 2013. ACM. pages 26

[66] Tobias Hilbrich, Joachim Protze, Bronis R. de de Supinski, Martin Schulz, Matthias S. Müller, and Wolfgang E. Nagel. Intralayer Communication for Tree-Based Overlay Networks. In *Intl. Conf. on Parallel Processing*, pages 995–1003, Washington, DC, USA, 2013. IEEE Computer Society. pages 26

[67] Christopher Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In *Proceedings of the 12th European PVM/MPI Users' Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'05, pages 138–147, Berlin, Heidelberg, 2005. Springer-Verlag. pages 26

[68] Christopher Falzone, Anthony Chan, Ewing L. Lusk, and William Gropp. A Portable Method for Finding User Errors in the Usage of MPI Collective Operations. *IJHPCA*, 21(2):155–165, 2007. pages 26

[69] Jesper Larsson Träff and Joachim Worringen. Verifying Collective MPI Calls. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27. PVM/MPI, Springer Berlin Heidelberg, 2004. pages 26

[70] Christopher Falzone, Anthony Chan, Ewing Lusk, and William Gropp. A Portable Method for Finding User Errors in the Usage of MPI Collective Operations. *Int. J. High Perform. Comput. Appl.*, 21(2):155–165, May 2007. pages 26

[71] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM. pages 26

[72] https://software.intel.com/en-us/intel-trace-analyzer. pages 26

[73] Michael Süss and Claudia Leopold. Common Mistakes in OpenMP and How To Avoid Them A Collection of Best Practices. In Matthias S. Müller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, and Michael Voss, editors, *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323, Berlin, Heidelberg, 2008. Springer-Verlag. pages 27

[74] Tim Cramer, Christian Terboven, Matthias Müller, Felix Münchhalfen, Joachim Protze, and Tobias Hilbrich. Classification of Common Errors in OpenMP Applications. In Luiz DeRose, BronisR. de Supinski, StephenL. Olivier, BarbaraM. Chapman, and MatthiasS. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 58–72. Springer International Publishing, 2014. pages 27, 28, 67, 115

[75] Hongyi Ma, Steve Diersen, Liqiang Wang, Chunhua Liao, Daniel J. Quinlan, and Zijiang Yang. Symbolic Analysis of Concurrency Errors in OpenMP Programs. In *ICPP*, pages 510–516. IEEE, 2013. pages 29

[76] Yices: An SMT solver. http://yices.csl.sri.com. pages 29

[77] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, IWOMP'11, pages 37–53, 2011. pages 29

[78] Yuan Lin. Static Nonconcurrency Analysis of OpenMP Programs. In Matthias S. Müller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, and Michael Voss, editors, *IWOMP*, volume 4315 of *LNCS*, pages 36–50. Springer, 2005. pages 29, 62, 69, 86

[79] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In Vikram S. Adve, María Jesús Garzarán, and Paul Petersen, editors, *LCPC*, volume 5234 of *LNCS*, pages 95–109. Springer, 2007. pages 29, 63, 69

[80] GOMP site. gcc.gnu.org/projects/gomp. pages 29

[81] Young-Joo Kim, Sejun Song, and Yong-Kee Jun. ADAT: An Adaptable Dynamic Analysis Tool for Race Detection in OpenMP Programs. In *ISPA*, pages 304–310. IEEE, 2011. pages 29

[82] Young-Joo Kim, Mi-Young Park, So-Hee Park, and Yong-Kee Jun. A Practical Tool for Detecting Races in OpenMP Programs. In Victor E. Malyshkin, editor, *PaCT*, volume 3606 of *LNCS*, pages 321–330. Springer, 2005. pages 29

[83] Ying Meng, Ok-Kyoon Ha, and Yong-Kee Jun. Dynamic Instrumentation for Nested Fork-join Parallelism in OpenMP Programs. In *Proceedings of the 4th International Conference on Future Generation Information Technology*, FGIT'12, pages 154–158. Springer-Verlag, 2012. pages 29

[84] Paul Petersen and Sanjiv Shah. OpenMP Support in the Intel Thread Checker. In Michael Voss, editor, *WOMPAT*, volume 2716 of *LNCS*, pages 1–12. Springer, 2003. pages 29

[85] Young-Joo Kim, Kim Daeyoung, and Yong-Kee Jun. An Empirical Analysis of Intel Thread Checker for Detecting Races in OpenMP Programs. In Roger Y. Lee, editor, *ACIS-ICIS*, pages 409–414. IEEE Computer Society, 2008. pages 29, 30

[86] Christian Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 669–676. IOS Press, 2007. pages 29

[87] Jianjiang Li, Dan Hei, and Lin Yan. Correctness Analysis based on Testing and Checking for OpenMP Programs. In *ChinaGrid Annual Conference, 2009. ChinaGrid '09. Fourth*, pages 210–215. Fourth ChinaGrid Annual Conference, IEEE, Aug 2009. pages 29

[88] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *ACM/IEEE Intl. Conf. on SuperComputing*, November 2000. pages 29

[89] James Cownie and Shirley Moore. Portable OpenMP debugging with TotalView. 2000. pages 30

[90] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Intl. J. on High Performance Computing Applications*, 20:287–331, 2006. pages 30

[91] M. Geimer, F. Wolf, B.J.N. Wylie, D. Becker, D. Böhme, W. Frings, M.-A. Hermanns, B. Mohr, and Z. Szebenyi. Recent Developments in the Scalasca Toolset. In *Intl. Workshop on Parallel Tools for High Performance Computing*, 2010. Record converted from VDB: 12.11.2012. pages 30

[92] Holger Brunst and Bernd Mohr. Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG. In Matthias S. Müller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, and Michael Voss, editors, *IWOMP*, volume 4315 of *Lecture Notes in Computer Science*, pages 5–14. Springer, 2005. pages 30

[93] David Lecomber and Patrick Wohlschlegel. Debugging at Scale with Allinea DDT. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Parallel Tools Workshop*, pages 3–12. Springer, 2012. pages 30

[94] K. Fürlinger and M. Gerndt. ompP: A Profiling Tool for OpenMP. In *Intl. Conf. on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, pages 15–23, Berlin, Heidelberg, 2008. Springer-Verlag. pages 30

[95] W.-F. Chiang, G. Szubzda, G. Gopalakrishnan, and R. Thakur. Dynamic Verification of Hybrid Programs. In *European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 298–301. Springer-Verlag, 2010. pages 30

[96] T. Hilbrich, M. S. Müller, and B. Krammer. Detection of Violations to the MPI Standard in Hybrid OpenMP/MPI Applications. In *Intl. Conf. on OpenMP in a New Era of Parallelism*, pages 26–35. Springer-Verlag, 2008. pages 30, 48

[97] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Combining Static and Dynamic Validation of MPI Collective Communications. In *Proceedings of the European MPI Users' Group Meeting*, EuroMPI'13, pages 117–122, New York, NY, USA, September 2013. ACM. pages 31, 51, 85

[98] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. PARCOACH: Combining static and dynamic validation of MPI collective communications. *International Journal of High Performance Computing Applications*, page 10.1177/1094342014552204, 2014. pages 31, 51, 85

[99] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Static Validation of Barriers and Worksharing Constructs in OpenMP Applications. In *International Workshop on OpenMP*, pages 73 – 86, Salvador, Brazil, September 2014. pages 31, 51, 85

[100] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Static/Dynamic Validation of MPI Collective Communications in Multi-threaded Context. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 279–280, New York, NY, USA, 2015. ACM. pages 31, 71, 85

[101] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. MPI Thread-Level Checking for MPI+OpenMP Applications. In *EuroPar*, 2015. pages 31, 33, 85

[102] Steven S. Muchnick. *Advanced compiler design implementation*. Academic Press, 1997. pages 35, 88, 101

[103] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, second edition*. Pearson Education, Inc, 2007. pages 35, 101

[104] NASPB site: http://www.nas.nasa.gov/software/NPB. pages 38, 44, 58, 67, 81, 103

[105] GCC 4.7. `gcc.gnu.org/gcc-4.7/`. pages 44

[106] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *in Proc. GCC Developers Summit*, pages 171–180. GCC summit, 2003. pages 44

[107] CORAL Benchmarks. `https://asc.llnl.gov/CORAL-benchmarks/`. pages 44

[108] H. Jourden. HERA: A hydrodynamic AMR Platform for Multi-Physics Simulations. In Tomasz Plewa, Timur Linde, and V. Gregory Weirs, editors, *Adaptive Mesh Refinement - Theory and Applications*, pages 283–294. Springer, 2003. pages 44, 58, 67, 81

[109] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM TOPLAS*, pages 13(4):451–490, 1991. pages 54, 101

[110] M. Wolff, S. Jaouen, and H. Jourdren. High-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. In *Discrete and Continuous Dynamical Systems Series S*. NMCF, 2009. pages 57

[111] IMB. software.intel.com/en-us/articles/intel-mpi-benchmarksl. pages 58

[112] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jasper Larsson Träff. MPI on a million Processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag. pages 58

[113] J. M. Bull, J. P. Enright, X. Guo, C. Maynard, and F. Reid. Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations. *Intl. J. of Parallel Programming*, 38(5-6):396–417, 2010. pages 81

[114] Lorna Smith and Mark Bull. Development of Mixed Mode MPI / OpenMP Applications. *Sci. Program.*, 9(2,3):83–98, 2001. pages 81

[115] Henry Lieberman. The Debugging Scandal and What to Do About It (Introduction to the Special Section). *Commun. ACM*, 40(4):26–29, 1997. pages 85

[116] S. Gulwani and A. Tiwari. Computing Procedure Summaries for Interprocedural Analysis. In R. De Nicola, editor, *European Symp. on Programming, ESOP 2007*, volume 4421 of *LNCS*, pages 253–267, 2007. pages 88

[117] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid Top-down and Bottom-up Interprocedural Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 249–258, New York, NY, USA, 2014. ACM. pages 88

[118] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid Top-down and Bottom-up Interprocedural Analysis. *SIGPLAN Not.*, 49(6):249–258, June 2014. pages 88

[119] I. Roy, G. R. Luecke, J. Coyle, and M. Kraeva. A scalable deadlock detection algorithm for UPC collective operations. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS'13, pages 2–15, The University of Edinburgh, 2013. pages 93

[120] A. Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS'11, 2011. pages 93

[121] The LLVM Compiler Infrastructure. http://llvm.org/. pages 93

[122] GCC. https://gcc.gnu.org/. pages 97

[123] HACC summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACC_Summary_v1.5.pdf. pages 104

[124] AMG summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/AMG2013_Summary_v2.3.pd. pages 104

[125] LULESH summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/LULESH_Summary_v1.pdf. pages 104

[126] SNAP summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/SNAP_Summary_v1.3.pd. pages 104

[127] miniFE summary. https://asc.llnl.gov/CORAL-benchmarks/Summaries/MiniFE_Summary_v2.0.pdf. pages 104

[128] Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski, editors. *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings*, volume 6132 of *Lecture Notes in Computer Science*. Springer, 2010. pages 105

[129] Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors. *OpenMP in the Petascale Era - 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings*, volume 6665 of *Lecture Notes in Computer Science*. Springer, 2011. pages 106

List of Figures

1.1 Growth of supercomputing power of the first (N=1) and the last (N=500) supercomputer recorded by TOP500 list (see http://www.top500.org) 10
1.2 Shared and Distributed Memory Architectures 12
1.3 Example of three MPI processes executing point-to-point communications (send/recv). . . . 12
1.4 Collective Communication Routines [8]. 13
1.5 Example of the PGAS model. 14
1.6 Example of a hybrid model combining MPI and OpenMP programming models. In this example, each MPI task is executed on a node and contains four OpenMP threads. 17
1.7 Boehm's version of the Waterfall Model 18
1.8 Example of V-Model development cycle 19
1.9 Example of Spiral development cycle model 19
1.10 First computer "bug" 20
1.11 Testing and debugging cycle [46] 22
1.12 Examples of erroneous MPI codes. 25
1.13 Classification of Common Issues in OpenMP Applications [74] 28
1.14 Examples of semantic defects (Violation of the standard and Conceptual defect). 28

2.1 MPI+OpenMP examples showing different uses of MPI calls. 34
2.2 Example of a simple C code with its associated CFG. 35
2.3 Different splitting phases according to the origin node. The statement *s* is supposed to be an OpenMP construct or an implicit or explicit barrier. The origin node is presented on the left and the result of the splitting phase on the right. 36
2.4 Augmented Control Flow Graph and parallelism words of codes in **Figure 2.1** 38
2.5 Automaton of possible parallelism words. Nodes 0, 2 and 3 correspond to code executed by the master thread or a single thread. Nodes 1 and 4 correspond to code executed in a parallel region, and 5 and 6 to code executed in nested parallel region. 39
2.6 Instrumented CFG **Figures 2.4(b) and 4.5(a)** (**Algorithm 11**) 43
2.7 Architecture of GCC 44
2.8 Overview of PARCOACH 45
2.9 Overhead of average compilation time 46
2.10 Examples of MPI thread-level non-compliant codes. 47
2.11 Examples of MPI thread-level non-compliant codes. 48

3.1 A simple example and its instrumentation 52
3.2 Example of Control Flow Graphs. From the left, a CFG showing execution orders, CFG of function f and two other CFGs 55
3.3 Example of a CFG from a Benchmark and their instrumentation (see Algorithm 13) 55
3.4 Execution time of collective calls from IMB 58

3.5 Overhead of average compilation time with and without verification code generation (CC functions insertion) 59
3.6 Execution time overhead for HERA with strong scaling 60
3.7 Execution time overhead for NASPB class C with strong scaling and for EulerMHD with weak scaling 60
3.8 Examples of deadlock situation in OpenMP programs 61
3.9 Example of a simple code (a) with its corresponding OMPCFG (b) and an example of OMPCFG containing barriers (c) 63
3.10 Functions `f` OMPCFG of Listing 2.3 ((a)) and `main` OMPCFG of Listing 2.6 after function `f` replacement ((b), see **Algorithm 18**) and an example of an OMPCFG with a loop ((c)) . . 65
3.11 Callgraph of Listing 3.4 (a) and BT from NASPB-OMP (b) 67
3.12 Overhead of average compilation time for NASPB-OMP and HERA 68

4.1 MPI+OpenMP examples showing different uses of MPI collectives. 72
4.2 Possible errors in a hybrid program with N MPI processes and 2 threads per process 74
4.3 CFG of Code 6 and the parallelism words associated at each node 75
4.4 Automaton of possible parallelism words. Node 0 corresponds to code executed by the master thread or a single thread. Node 1 correspond to code executed in a parallel region and 3 to code executed in nested parallel region. 76
4.5 Examples of CFG with monothreaded regions highlighting thread barriers 77
4.6 Instrumented CFG **Figures 4.3(a) and 4.5(a)** (**Algorithm 11**) 80
4.7 Overhead of average compilation time with and without verification code generation 82
4.8 Execution-time overhead for HERA and NASPB-MZ Class B with 8 threads per MPI process 83
4.9 Execution-Time Overhead for the EPCC suite (Weak scaling) (8 threads per process). The benchmark suite contains measurements for the following operations : Barrier, Reduce, Broadcast, Scatter and Alltoall. 83

5.1 Example of a false positive. 87
5.2 Example of a false negative. 87
5.3 Example of a problematic case for the first inter-procedural approach. 90
5.4 The applied second inter-procedural approach. 90
5.5 Function main of the NASPB-MPI IS v3.2. 92
5.6 Pragma-based approach in function main of the NASPB-MPI IS v3.2 and Example of MPI communications on different communicators (com1 and com2). 92
5.7 Examples for semantic problems detection ((a)) and errors correction by compilers ((b) and (c)). 94

A.1 Architecture of GCC 97
A.2 Architecture of GCC 98

List of Tables

1.1 Characteristics of Tera 100, Curie and Inti supercomputers (2014, [3]) 11
1.2 Classification of MPI Debugging Tools. T: Trace-based, OD: Online dynamic analysis, L: MPI library 27

2.1 Level of threads parallelism at function entries for NASPB-MZ 39
2.2 Compliance Results 45
2.3 Compliance Results 46

3.1 Compilation and Execution Results of MPI Applications 58
3.2 Directive nodes in the OMPCFG 63
3.3 Static Results for each benchmark (F=FORTRAN) 68

4.1 Compilation and Execution Results 82

5.1 Statistics on EulerMHD and NASPB-MPI v.3.3 class B. 90