Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.
Communicated by Prof. Dr. rer. nat. Matthias S. Müller

Darstellungskonzepte für OpenMP in der SPMD IR und ihre Realisierung in MLIR

Towards Representing OpenMP in SPMD IR: Concepts and realization in MLIR

Bachelorarbeit

Frederic J. Malmendier
Matrikelnummer: 447861

Aachen, den 9. Juni 2026

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Dr. rer. nat. Stefan Lankes (*)
Betreuer:	Semih Burak, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Betriebssysteme, RWTH Aachen University

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 9. Juni 2026

Kurzfassung

Diese Arbeit erweitert die SPMD Intermediate Representation (IR) im MLIR-Framework um das Programmiermodell OpenMP. Die ursprüngliche IR bildet eine Abstraktionsschicht für API-basierte parallele Programmiermodelle (PPMs), die auf einem Single Program, Multiple Data (SPMD) Modell basieren. Sie orientiert sich dabei insbesondere an Konzepten von MPI und nutzt explizite Kommunikations- und Synchronisationsmechanismen innerhalb einer verteilten Speicherarchitektur (Distributed Memory). Im Gegensatz dazu folgt OpenMP einem direktivenbasierten Ansatz, bei dem Synchronisation an den Kontrollfluss des Programms gekoppelt ist und die Kommunikation über einen gemeinsamen Speicher (Shared Memory) erfolgt.

Um diese semantische Lücke zu schließen, entwickelt die vorliegende Arbeit einen ersten Design-Entwurf sowie eine prototypische Implementierung zur Integration von OpenMP in die IR. Ein spezieller Conversion Pass übersetzt OpenMP-Direktiven in kollektive, regionenbasierte Operationen, wodurch `parallel`- und Worksharing-Loop-Regionen auf neu eingeführte Operationen im `spmd`-Dialekt abgebildet werden können.

Die Evaluation der Erweiterung erfolgt mithilfe des bestehenden Collective Verification Pass (der IR) auf einer eigens dafür erstellten Benchmark-Suite, die gezielt OpenMP-Synchronisationsfehler simuliert. Die Ergebnisse zeigen, dass die erweiterte IR in der Lage ist, OpenMP-Programme erfolgreich zu verifizieren, ohne auf modellspezifische Abstraktionen angewiesen zu sein.

Stichwörter: OpenMP, MPI, MLIR, SPMD, Intermediate Representation, statische Analyse

Abstract

This thesis extends the SPMD IR within the MLIR framework by incorporating the OpenMP programming model. The original IR provides an abstraction layer for API-based parallel programming models (PPMs) that share a Single Program, Multiple Data (SPMD) execution style. At its core, the abstraction is derived from MPI concepts and relies on explicit communication and synchronization mechanisms within distributed memory domains. In contrast, OpenMP's directive-based model synchronizes execution with the program's control flow and handles communication via shared memory.

To bridge this semantic gap, this work presents an initial design and a prototype implementation for representing OpenMP within the IR. Specifically, a dedicated conversion pass models OpenMP directives as collective, region-based operations, mapping `parallel` and worksharing-loop regions to newly introduced operations in the `spmd` dialect.

The extension was evaluated by applying the IR's existing Collectives Verification pass to a specifically created benchmark suite targeting OpenMP synchronization errors. The results demonstrate that the IR can successfully verify OpenMP programs without relying on model-specific abstractions.

Keywords: OpenMP, MPI, MLIR, SPMD, Intermediate Representation, Static Analysis

Contents

List of Figures

List of Tables

1. Introduction

The increasing demand for computational performance has driven modern high-performance computing (HPC) systems toward highly parallel architectures. To efficiently exploit these systems, parallel programming models (PPMs) provide abstractions that enable developers to express and manage concurrency. However, the resulting complexity of parallel applications significantly increases the need for flexible analysis tools that are reusable across different programming models.

Among the most widely used of these models are the Message Passing Interface (MPI) [1], an API-based distributed-memory model following the Single Program, Multiple Data (SPMD) paradigm, and OpenMP [2], a directive-based shared-memory model employing a fork-join execution pattern. In practice, many HPC applications combine both approaches in hybrid configurations to exploit parallelism across multiple levels of the hardware hierarchy.

The complexity of parallel programs makes tool support for correctness verification essential. However, existing tools are typically designed for a specific PPM, which limits their reusability across models and complicates their extension to new programming paradigms. The SPMD IR [3, 4] addresses this limitation by providing a unifying abstraction for SPMD-based PPMs. Implemented as a dialect within the MLIR [5] compiler framework, it transforms model-specific API calls into a common intermediate representation, enabling analysis tools to operate independently of the underlying programming model.

In this work, we propose an initial design to extend the SPMD IR to represent the shared-memory OpenMP programming model. This requires bridging a fundamental semantic gap: while SPMD-based PPMs operate on independent processes with private memory and explicit communication, OpenMP relies on threads within a shared address space and implicit communication. Despite these differences, OpenMP's `parallel` construct exhibits an SPMD-like execution pattern, where threads execute the same code concurrently and synchronize via collective barriers.

Unlike API-based PPMs, OpenMP expresses parallelism through compiler directives. Therefore, our approach builds upon MLIR's existing OpenMP dialect (`omp`) [6], which provides a structured representation of these directives and serves as the basis for conversion into the SPMD IR. For evaluation, we focus on the Collectives Verification use case of the SPMD IR, as it naturally aligns with synchronization-related errors in OpenMP programs.

This work makes the following contributions:

1. An analysis of core OpenMP features – specifically the fork-join execution model and worksharing mechanisms – and their representation in the `omp` dialect, highlighting similarities and differences with the SPMD paradigm (Section 4.1).

2. A design strategy for representing these features within the SPMD IR, including a discussion of alternative modeling approaches and the proposal of a concrete design (Section 4.2).

3. A prototype implementation including a conversion pass from the `omp` dialect to the SPMD IR, extensions to existing analysis passes, and an evaluation using a dedicated benchmark suite (Section 4.3 and 5)

1.1. Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background on parallel programming models and compiler infrastructures, covering MPI, OpenMP, MLIR, and the SPMD IR. Chapter 3 reviews related work on intermediate representations for parallel programs and OpenMP correctness tools. Chapter 4 presents the core contributions of this thesis, including directive analysis, design, and implementation. Chapter 5 evaluates the approach using the Collectives Verification on representative OpenMP error patterns. Finally, Chapter 6 concludes the thesis and outlines directions for future work.

2. Background

This chapter introduces the theoretical foundations necessary to understand the conceptual work of representing OpenMP in the SPMD IR and the implementation efforts later presented in this thesis. Section 2.1 presents the programming models relevant to this thesis, covering the SPMD paradigm with MPI and the shared-memory model of OpenMP. Section 2.2 introduces the compiler infrastructure, beginning with MLIR as the framework in which the SPMD IR is realized, followed by the SPMD IR itself.

2.1. Parallel Programming Models

This section introduces the parallel programming models relevant to this thesis, focusing on the SPMD paradigm with MPI and the shared-memory model of OpenMP.

2.1.1. The SPMD paradigm and MPI

The SPMD paradigm is a prominent execution model in high-performance computing, with widespread adoption among many parallel programming models [7]. In an SPMD program, multiple processing elements independently execute the same source code. While the SPMD paradigm itself does not dictate a specific memory model, it is traditionally associated with distributed-memory architectures. In such environments, each process operates within its own private memory space, meaning no process can directly access the memory of another. Consequently, they must rely on explicit communication mechanisms to exchange data and synchronize their states.

Despite executing the exact same program, the processing elements diverge in their control flow based on their unique identifiers – commonly referred to as *ranks* in MPI. This divergence allows each process to determine the specific portion of a computation it is responsible for. A common pattern for exploiting this parallelism is data decomposition: a large dataset is partitioned, each process computes its local portion, and the partial results are subsequently combined.

It is important to note that SPMD is fundamentally an abstract execution model. To build actual software, this model must be implemented via a concrete

programming interface. The Message Passing Interface (MPI) [1] is the de facto standard for distributed-memory parallel programming and serves as a prominent realization of the SPMD paradigm. MPI is strictly an API-based library; therefore, all communication, synchronization, and process management are expressed through explicit function calls embedded directly into the program.

The following provides an overview of the core MPI concepts from the MPI Standard [8] that are particularly relevant to this work:

Communicators and rank identification. In MPI, a communicator defines a group of ranks that can communicate with each other. The default communicator, `MPI_COMM_WORLD`, encompasses all ranks in a program. Each rank obtains its unique identifier within a communicator via `MPI_Comm_rank` and the total number of participating ranks via `MPI_Comm_size`. Communicators can be split into subgroups using operations like `MPI_Comm_split`, enabling hierarchical or domain-specific communication patterns.

Collective communication. MPI provides a set of collective operations that must be called by all ranks within a communicator. These include:

- `MPI_Barrier`: synchronizes all ranks, blocking until every rank has reached the API call.
- `MPI_Bcast`: distributes data from one rank to all others.
- `MPI_Scatter` and `MPI_Gather`: divide data among ranks and collect distributed data back to a single rank.
- `MPI_Allreduce`: combines partial results from all ranks using a specified operation (e.g., sum) and distributes the result to every rank.

Furthermore, all mentioned collective operations receive a communicator handle as input parameter and their execution involves all ranks within the given communicator. Additionally, they are blocking by default, meaning that the function call will only fully return when all ranks finished their local execution and require that the same collective is called in the same order by all ranks of the communicator. Violations of this requirement may lead to deadlocks or undefined behavior and are referred to as collective errors.

Non-blocking communication and RMA. Beyond collective operations, MPI supports point-to-point communication (`MPI_Send`/`MPI_Recv`), non-blocking communication operations that return immediately and require explicit completion (`MPI_Isend`/`MPI_Wait`), and remote memory access (RMA) for one-sided communication.

2.1.2. OpenMP

OpenMP [9] is a directive-based shared-memory programming model, which is a widely adopted standard for node-level parallelization in HPC [10]. Unlike MPI's distributed-memory model, where parallelism is expressed via ranks that execute on separate memory spaces, OpenMP expresses parallelism as multiple threads executing an annotated region of the code in a shared address space. The programmer can leverage compiler directives (pragmas in C/C++) to define parallel regions which will then be executed in compliance with the specification of OpenMP's execution model.

OpenMP organizes its features with compiler directives, each representing a mechanism which can be used by the programmer to specify a program's behavior. Additionally, clauses can be added to a directive, altering the behavior in compliance with the clause specification. In C/C++, directives are expressed as `#pragma omp` annotations. When referring to a construct, such as the `parallel` construct, this includes both the directive (e.g., `#pragma omp parallel`) and its associated structured block. Constructs specify the region of the structured block. The execution model specifies the different execution patterns within regions. In Figure 2.1, the `parallel` construct consists of the compiler directive `#pragma omp parallel` and the associated structured block – representing a parallel region. Inside the structured block lies the worksharing-loop construct which consists of the directive `#pragma omp for` and the associated for-loop – representing a worksharing-loop region.

```
1 #pragma omp parallel
2 {
3     #pragma omp for
4     for (i=0; i<N; ++i){
5         // ...
6     }
7 }
```

Figure 2.1.: OpenMP worksharing-loop construct in C

To later understand the Directives Analysis (see Section 4.1), the following will elaborate on the execution model and how the shared-memory model can be leveraged by the programmer.

Execution model. A program begins with a single initial thread executing sequentially. When this thread encounters a `parallel` construct, it spawns a thread team from a pool of available threads which will then continue the enclosed parallel

region. All threads in the team concurrently execute the enclosed structured block of the `parallel` construct and are identified by unique thread numbers ranging from 0 to $n-1$. At the end of the parallel region, an implicit barrier synchronizes the team, after which only the initial thread continues execution. The `parallel` construct serves as basis for all parallel patterns discussed in the remainder of this work.

Data scoping. OpenMP distinguishes between shared and private data. The scope of a variable determines whether threads hold their own private copy or access the common address space, therefore sharing access with other threads. By default, variables declared outside a parallel region are shared within, while variables declared on the inside are private. This can be overridden through explicit data-sharing clauses, allowing fine-grained control over variable visibility.

2.2. Compiler Infrastructure

2.2.1. MLIR

Intermediate representations (IRs) are central components of modern compilers, serving as the medium on which analyses, optimizations, and transformations operate. A typical compilation pipeline, as depicted in Figure 2.2, translates source code through one or more IRs before generating target-specific machine code.

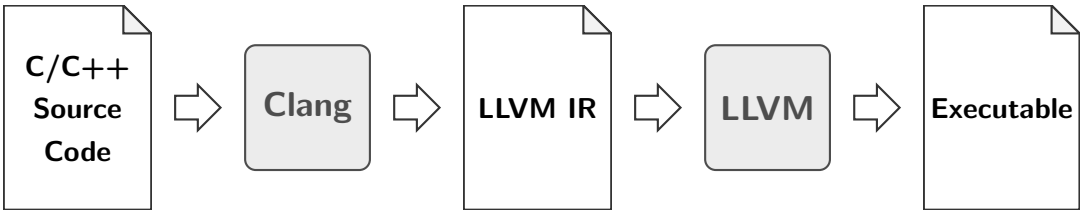

Figure 2.2.: LLVM toolchain compilation pipeline

LLVM IR [11] is a well-established IR that provides a common representation for languages such as C/C++, Fortran, and Julia within the LLVM toolchain. It defines a fixed set of low-level operations and enables a broad range of optimization passes to be applied uniformly. However, LLVM IR's fixed instruction set limits its ability to represent domain-specific semantics at higher levels of abstraction.

The Multi-Level Intermediate Representation (MLIR) [5] addresses this limitation. MLIR is a compiler framework within the LLVM project that supports extensible, multi-level program representations. A core strength of MLIR is its ability to use transformation passes to mix different levels of abstraction within a

single representation. For instance, when compiling source code, a frontend like Clang traditionally generates LLVM IR as a low-level abstraction. As illustrated in Figure 2.3, utilizing MLIR's infrastructure allows this LLVM IR to be transformed into a representation that combines the low-level semantics of the `llvm` dialect with higher-level abstractions provided by other MLIR dialects. This flexible mixing of abstractions preserves domain-specific information that can then be used for further analysis or optimization.

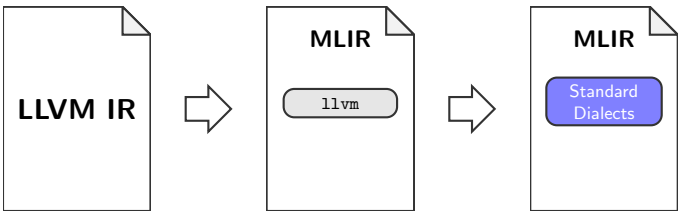

Figure 2.3.: MLIR generation pipeline

In MLIR, dialects act as namespaces that group related operations, types, and attributes to form a domain-specific abstraction (e.g., the OpenMP dialect). Multiple dialects can interact freely with one another, allowing a single intermediate representation to mix abstractions from different domains. MLIR provides several built-in dialects, among which the relevant dialects for this thesis are `arith` for arithmetic operations, `scf` for structured control flow (e.g., loops and branches), and `llvm` as an embedding of the LLVM IR into MLIR.

Operations provide the primary abstraction for representing domain-specific semantics. For instance, in the `scf` dialect, `scf.if` represents a conditional branch and `scf.for` a `for`-loop. Each operation can have input values, called operands, and produces result values. All of these values strictly follow the Static Single Assignment (SSA) form. This means that a value is created exactly once by a single operation and can never be changed afterwards. To pass these values between different blocks, MLIR uses *block arguments*. While operands represent values computed at runtime, to store static information known at compile-time, operations can also hold constant data known as attributes.

Additionally, operations may contain *regions*, which in turn consist of a list of *blocks*. A block is a sequential list of operations and may define block arguments that serve as entry parameters for that specific block. This nested structure enables hierarchical IR designs, where the control flow semantics within and among regions are strictly defined by the enclosing parent operation. For example, the `scf.if` operation contains two regions (a *then* and an *else* region), of which only one is executed based on a boolean operand. Similarly, the `scf.for` operation contains a single region representing the loop body, receiving the dynamically updated induction variable as a block argument.

Furthermore, common properties and constraints of operations can be enforced by attaching *traits* during dialect definition. For example, the `scf.if` operation utilizes the `SingleBlockImplicitTerminator<scf::YieldOp>` trait. This specifies that the region must always terminate with an `scf.yield` operation, which will be inserted implicitly if left out by the programmer. In the implementation phase of this thesis, the standard `IsolatedFromAbove` trait will be utilized. This trait establishes a strict scoping boundary, specifying that any reference to SSA values defined outside the operation's lexical scope is illegal.

To round up MLIR's infrastructure, *passes* are the standard mechanism for traversing and modifying the IR. Passes are modular, self-contained compiler phases that either perform structural transformations – such as the lowering (conversion) of one dialect into another – or execute analysis tasks. To serve as a foundation for the design presented in the subsequent chapters, it is essential to understand how OpenMP is represented within this ecosystem.

The OpenMP MLIR Dialect The `omp` dialect provides an abstraction for a subset of OpenMP features, modeling its specific directives and clauses. The operations provided by this dialect directly correspond to OpenMP directives in the source code; for example, the `omp.parallel` operation represents the `#pragma omp parallel` construct. By providing a native MLIR representation of OpenMP semantics, this domain-specific dialect captures the original execution model of the pragmas. Consequently, the `omp` dialect serves as the ideal foundation for this thesis, acting as the starting point for the representation of OpenMP in the SPMD IR.

2.2.2. SPMD IR

Tools for analyzing or optimizing parallel programs are typically developed for a single PPM, making them neither reusable across PPMs nor easily expandable to support new ones. The SPMD IR [3] addresses this problem by introducing a unifying intermediate representation for SPMD-based PPMs, realized as the `spmd` dialect in MLIR. At its core, the dialect uses transformation passes to convert API function calls of various programming models – including the distributed-memory model MPI, PGAS models SHMEM and NVSHMEM, and the GPU communication library NCCL – into a common set of IR operations. Figure 2.4 illustrates the IR's pipeline for converting API calls into the `spmd` dialect.

While these models vary in their specific memory semantics, they all fundamentally rely on explicit communication between independent execution units (ranks) with disjoint memory spaces. The resulting abstraction unifies these PPMs and can be used by further analysis tools.

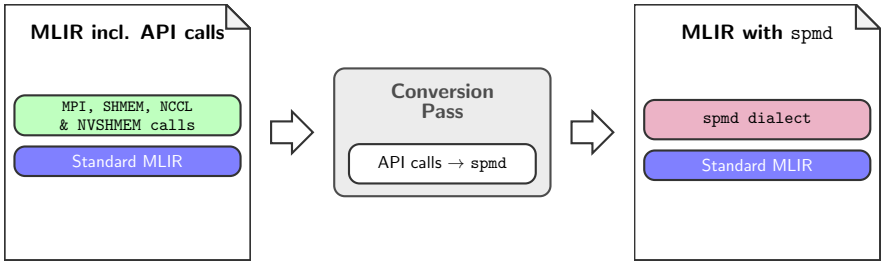

Figure 2.4.: SPMD IR conversion pipeline for API-based PPMs

The `spmd` dialect provides operations that encapsulate the core functionality of SPMD-based programming models. Its operations are primarily communicator-based, meaning they operate within or manage a specified set of active ranks. Relevant operations for this thesis are:

- `spmd.commWorld`: Initializes and returns the global communicator, representing all active ranks (analogous to `MPI_COMM_WORLD`).
- `spmd.commDup`: Duplicates an existing communicator provided as an operand and returns a new, communication handle represented as a unique SSA operand.
- `spmd.getRankInComm`: Takes a communicator as an operand and returns the unique rank identifier of the executing process within that specific context.
- `spmd.barrier`: Performs a collective synchronization across a given communicator. It features an `isBlocking` attribute to specify whether the operation halts execution until all participating ranks have reached the barrier.

In addition, the `spmd` dialect provides counterparts to the previously introduced MPI collective functions including `spmd.scatter`, `spmd.gather`, `spmd.allreduce` and `spmd.commDup` (see Section 2.1.1). In a recent extension, Remote Memory Access (RMA) operations, including `spmd.put` and `get`, and a corresponding data race detection pass, were introduced to the dialect [4].

The SPMD IR features a dedicated Multi-Value (MV) analysis pass that leverages the dialect's unified abstraction to statically determine an operation's active set of ranks – specifically, the subset of ranks actively executing it within a given communicator. To formalize this level of parallelism, the analysis assigns an `executionKind` attribute to each operation, classifying whether it is executed by `All` ranks, a statically known subset (`Static`), a dynamically determined subset (`Dynamic`), or a single rank (`One`).

To determine these execution states, the analysis tracks multi-values throughout the program's control flow. A multi-value is defined as a variable whose runtime

value can vary across different ranks within the same communicator. These values originate from designated MV-seed operations. A primary example is `spmd.getRankInComm`, which returns a unique identifier for each executing rank. When the control flow encounters a branching operation, such as an `scf.if`, whose condition depends on a multi-value, the execution diverges. This divergence implies that not all ranks will evaluate the condition identically, resulting in only a subset of ranks entering the operations region. Because the exact subset of participating ranks can only be resolved at runtime, the analysis assigns the `Dynamic executionKind` to the branching operation. Consequently, all operations nested within its regions inherit this `Dynamic` state as well.

The Collective Verification pass then utilizes the computed `executionKind` attributes to check whether an operation with the `Collective` trait is nested within a divergent structured control flow region. Such nesting would result in a potential deadlock or undefined behavior, as not all ranks in a communicator would reach the collective operation.

3. Related Work

The SPMD IR was introduced as a unifying intermediate representation for parallel programming models within the MLIR framework. It was initially established by Burak et al. [3] to support the distributed-memory model MPI, the PGAS model SHMEM, and the GPU communication library NCCL. A subsequent extension integrated Remote Memory Access (RMA) semantics and added support for NVSHMEM [4].

While these PPMs share an SPMD execution pattern, they differ in their memory models and communication mechanisms. Despite these differences, they all rely on disjoint memory spaces and explicit communication. This thesis integrates a programming model that fundamentally differs from these previously supported models. OpenMP is a directive-based shared-memory model in which threads communicate implicitly through a common address space rather than through explicit message passing.

In OpenMP, thread synchronization is coupled to the program's underlying control flow. Consequently, OpenMP is vulnerable to structural synchronization errors, particularly when divergent execution paths cause only a subset of threads to encounter a barrier or worksharing construct. Münchhalfen et al. [12] provide a systematic classification of such OpenMP programming errors, formally categorizing these exact structural defects.

Several tools have been developed to detect such deadlocks and structural violations. Saillard, Carribault, and Barthou [13] propose a static analysis for OpenMP that checks whether all threads execute the same number of barriers and verifies that worksharing constructs are not conditionally executed. Their approach operates on a custom control-flow graph (OMPCFG) specifically extended to represent OpenMP directives. Building upon this, PARCOACH [14] generalizes the analysis to support both MPI and OpenMP by constructing a parallel program control-flow graph (PPCFG) and detecting collective errors through execution-order analysis. To reduce false positives, Huchant et al. [15] further extend PARCOACH with a multi-valued expression analysis. By tracking data dependencies from seed operations – such as `MPI_Comm_rank` or `omp_get_thread_num` – through a program dependence graph (PDCG), the analysis accurately identifies which control flow branches diverge based on process or thread identifiers.

A common characteristic of all three approaches is their reliance on constructing separate, graph-based representations (such as the OMPCFG, PPCFG, and

PDCG) to perform their verification. While the extension by Huchant et al. [15] is conceptually similar to the SPMD IR's Multi-Value analysis, the SPMD IR embeds both the unified program representation and the multi-value information directly within the IR. Therefore, by mapping OpenMP's barriers and worksharing constructs onto collective operations in the SPMD IR, this work can reuse the IR's Collectives Verification and Multi-Value analysis with only minimal adaptations. The SPMD IR authors [3, 4] argued that supporting new programming models only requires additional conversion passes. However, this claim had previously only been demonstrated for API-based models, where library function calls serve as the entry point for conversion. This thesis validates this claim for a fundamentally different paradigm, applying the principle to a dialect-based input by converting region-based operations from the `omp` dialect.

4. Representing OpenMP in the SPMD IR

This chapter presents the conceptual design for representing OpenMP constructs within the SPMD IR architecture. Rather than covering the full breadth of the OpenMP specification, this work focuses on two foundational paradigms: the fork-join model, realized through the parallel construct, which provides the structural basis for OpenMP's multithreaded execution; and worksharing mechanisms, which build upon this foundation by distributing work across the threads of a parallel region. Further OpenMP paradigms, such as tasking, device offloading, and SIMD, were investigated but are not part of this approach (see Section 4.1.3). The chapter is structured in three parts. Section 4.1 provides a directive analysis that examines the selected constructs, their representation in the `omp` dialect, and their conceptual similarities with the SPMD paradigm. Section 4.2 presents the design, mapping the analyzed constructs onto existing `spmd` operations where possible and introducing new operations where necessary. Finally, Section 4.3 details the implementation of this design within the existing SPMD IR toolchain.

4.1. Directive Analysis

This section provides an in-depth analysis of selected OpenMP constructs and their representation within the OpenMP dialect in MLIR, referred to as the `omp` dialect. The primary goal of this analysis is to examine the semantic behavior of these constructs, outline their conceptual similarities with classical SPMD-based models like MPI, and present their design in the `omp` dialect. These findings serve as the foundation for the proposed conceptual representation of OpenMP in the SPMD IR (see Section 4.2).

Rather than covering the entirety of the OpenMP specification, this work focuses on a distinct subset of constructs, which are summarized in Table 4.1. This selection is conceptually oriented toward the "OpenMP Common Core" presented by Mattson et al. [2], focusing on the most fundamental paradigms of shared-memory programming: the fork-join model and worksharing mechanisms. Further constructs and paradigms that extend beyond this core are discussed subsequently in the scope limitations (see Section 4.1.3).

Table 4.1.: Selected OpenMP constructs

Construct	Paradigm	MLIR `omp` Operation
`parallel`	Fork-Join	`omp.parallel`
`for` / Worksharing-Loop	Worksharing	`omp.wsloop`
`single`	Worksharing	`omp.single`
`master` / `masked`	Execution Control	`omp.master` / `omp.masked`
`barrier`	Synchronization	`omp.barrier`

A critical technical constraint of this analysis is defined by the underlying compiler infrastructure, specifically the OpenMP to the SPMD IR conversion pipeline. As illustrated in Figure 4.1, this transformation pipeline requires the input to be an MLIR representation utilizing the `omp` dialect, which is then mapped to the `spmd` dialect (further implementation details are discussed in Section 4.3).

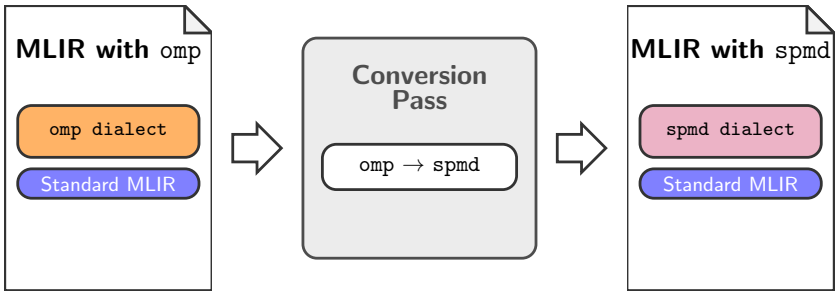

Figure 4.1.: Conceptual OpenMP to SPMD IR conversion pipeline

Consequently, the structural capabilities of the SPMD IR – and by extension, the scope of this analysis–are strictly limited to OpenMP features that are explicitly supported and represented within the `omp` dialect. Therefore, this work focuses exclusively on constructs that have a direct representation in the `omp` dialect (a comprehensive list of all available `omp` operations is provided in Appendix A.1). Furthermore, alongside these compiler directives, the analysis also briefly addresses relevant library routines from OpenMP's runtime API.

The remainder of this section is structured as follows: Section 4.1.1 introduces the foundational fork-join paradigm and its realization through the `parallel` construct. Section 4.1.2 analyzes worksharing mechanisms, encompassing loop distributions as well as the `single` and `masked` execution models. Finally, Section 4.1.3 delineates the scope limitations, distinguishing between features feasible for future work (such as the `teams` construct and mutual exclusion) and those that are out of scope for this work (like tasking, target offloading, and SIMD).

Unless otherwise specified, the semantic definitions, scoping rules, and execution

models of the OpenMP constructs discussed throughout this analysis are based on the official OpenMP API Specification, Version 6.0 [9].

4.1.1. Fork-Join

Compared to unstructured approaches like POSIX threads[1], the fork-join model imposes a structured paradigm for multithreaded execution. Early works by Conway and Dennis and Van Horn introduced *fork* and *join* as mechanisms for process creation and termination synchronization. In later adaptations such as the Cilk-5 language [19], the fork-join model was refined by dividing the execution into distinct sequential and parallel phases. Since the fork-join concept has been used repeatedly in various forms, it is difficult to define the model precisely; however, there are some fundamental characteristics. An execution typically begins with a single initial thread executing a sequential phase. This thread can perform a *fork* operation to spawn a set of concurrent threads, thereby entering a parallel phase. A subsequent *join* operation establishes a synchronization point where these threads converge, allowing the single initial thread to resume sequential execution. OpenMP adopts a team-based interpretation [20] of this model, in which a group of threads jointly executes a parallel region.

In OpenMP, this paradigm is realized through the `parallel` construct. As illustrated in Figure 4.2, it serves as the entry point to a fork-join style execution. The construct encompasses a parallel region – a structured block of code – that is executed concurrently by all threads within the newly spawned team.

```
1 #pragma omp parallel
2 {
3     // ...
4 }
```

Figure 4.2.: OpenMP `parallel` construct in C

Execution Model

Building upon the fundamental OpenMP execution mechanisms introduced in Section 2.1.2, the `parallel` construct dictates the semantics of the region. When a thread encounters a `parallel` construct, it acts as the initial thread and forms a new team by recruiting unassigned threads from the runtime pool, establishing a fork-join execution flow as illustrated in Figure 4.3. The encountering thread

[1] POSIX Threads (pthreads) represent a standardized C language programming interface, offering a low-level API for explicit thread management and synchronization [16].

is strictly bound to the construct; consequently, its newly formed team provides the active *execution context* for the region – that is, the set of threads that jointly execute the structured block, identified by their thread numbers and synchronized at the region's boundaries. The size of this team is determined dynamically at runtime, but remains constant once the region has been entered.

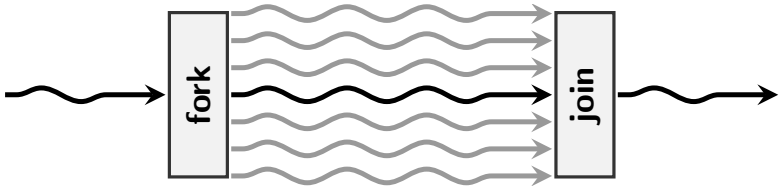

Figure 4.3.: OpenMP fork-join execution model

Within this execution context, all threads of the team concurrently execute the exact same structured code block. This semantic behavior closely mirrors the SPMD paradigm discussed in Section 2.1.1. In both paradigms, the execution logic relies on unique identification to distribute work or branch the control flow. Similar to ranks in SPMD-based models like MPI, OpenMP distinguishes executing entities by assigning a unique thread number (ranging from 0 to $n-1$) to each thread in the team. This identifier can be queried dynamically via the `omp_get_thread_num()` API routine.

The active execution context is strictly bounded by the structural limits of the parallel region. Its end is marked by an implicit barrier, serving as a mandatory synchronization point. Only when all threads of the team have arrived at this barrier does the lifetime of the team conclude. The worker threads are released back into the unassigned pool, and the initial thread resumes the sequential execution phase. Furthermore, OpenMP allows an active execution context to spawn nested parallel phases. The maximum depth of such nested regions is globally constrained by the `OMP_MAX_ACTIVE_LEVELS` environment variable. In addition to the implicit barrier at the end of the region, OpenMP provides an explicit `barrier` construct, which enforces synchronization at arbitrary points within the execution context.

Clauses

OpenMP is fundamentally based on a shared-memory execution model (see Section 2.1.2). Within this model, a variable's scope is determined by the placement of its declaration in the code. By default, variables declared outside of a parallel region are implicitly shared among all threads within the region's execution context, while those declared locally within the region are strictly private.

The `parallel` construct accepts several data-sharing clauses that explicitly override this default behavior. The `private` clause allocates an uninitialized,

thread-local instance of an externally declared variable for each thread in the team, isolating it from the original memory location. The `firstprivate` clause extends this semantics by initializing the local copy with the value of the original variable upon entering the region. Conversely, the `shared` clause explicitly asserts that all threads reference the same shared memory location. This explicit declaration becomes necessary when developers modify the default scoping rules, for instance, by applying the `default(none)` clause. Furthermore, the `reduction` clause allows threads to safely aggregate thread-local computations into a single global result; however, since reductions are more frequently used in combination with loop constructs, they will be discussed in Section 4.1.2.

In a second category, the OpenMP specification defines clauses that dictate the configuration of a parallel region's execution context. The `num_threads` clause dynamically restricts the number of threads participating in the team, while the `if` clause allows for conditional parallelism, forcing a sequential execution with a team size of one if the evaluated condition is false. The `proc_bind` clause defines the thread affinity policy, determining how the runtime pins threads to hardware resources such as NUMA domains. Importantly from an IR perspective, the `proc_bind` policy is provided as a constant value known at compile time, whereas clauses such as `num_threads` or `if` evaluate to dynamic values at runtime. These configuration and data-scoping mechanisms are demonstrated in the expanded example in Figure 4.4.

```
1  int x; // shared by default
2  #pragma omp parallel num_threads(4) private(x)
       proc_bind(spread)
3  {
4      // ...
5  }
```

Figure 4.4.: OpenMP `parallel` construct with clauses in C

Representation in the `omp` dialect

The `omp` dialect introduces the `omp.parallel` operation to represent the semantics of the OpenMP `parallel` construct. It is a *region-based* operation, meaning it contains a single MLIR region that encapsulates the structured code block to be executed.

The operation itself establishes the active execution context for this region. When lowering C code to MLIR, the code block enclosed by the `#pragma omp parallel` directive is mapped directly into the region of the `omp.parallel` operation. Because

the semantics of the execution context are inherently tied to this enclosing operation, structural OpenMP features, such as the implicit barrier and team termination at the end of the region, are modeled implicitly by the region's terminator operation.

In addition, the explicit barrier generated by the `barrier` construct is represented by the `omp.barrier` operation in the dialect. Structurally, the `omp.parallel` operation does not enforce isolation from its enclosing scope – external SSA values can be freely referenced from within the region, which directly reflects OpenMP's default shared-memory semantics[2].

Furthermore, OpenMP runtime API routines that query the execution context, such as `omp_get_thread_num()`, do not require dedicated operations in the dialect; they are simply represented as standard function calls within the region's body.

To capture the configuration clauses introduced in the previous section, the dialect differentiates between dynamic and static properties, as detailed in Section 2.2.1. Clauses that are evaluated at runtime, such as `num_threads`, are represented as dynamic SSA operands attached to the `omp.parallel` operation. Conversely, clauses known at compile-time, such as `proc_bind`, are attached as constant attributes.

Regarding data-scoping, the dialect leverages MLIR's lexical scoping rules. The `shared` scope is represented implicitly by allowing operations inside the region to capture and reference SSA operands defined in the enclosing outer scope. For `private` scopes, the necessary memory allocations are placed directly inside the region's body. When an externally declared variable is explicitly privatized using the `private` clause, the `omp` dialect maps the original operand to a new, thread-local allocation. Once a variable is privatized, referencing the original external operand from inside the region is strictly forbidden; only the thread-local copy may be used. Figure 4.5 illustrates how the clauses from Figure 4.4 are conceptually mapped to MLIR.

```
1    // %x defined in outer scope
2    %threads = // ...
3    omp.parallel num_threads(%threads) proc_bind(spread)
         private(@priv %x -> %x_priv) {
4        // ...
5        omp.terminator
6    }
```

Figure 4.5.: Representation of OpenMP clauses in the `omp` dialect

[2]Further implementation details regarding the specific traits and terminators of the `omp` dialect can be found in the official MLIR documentation [21].

4.1.2. Worksharing

While OpenMP's implementation of the fork-join paradigm via the `parallel` construct (see Section 4.1.1) establishes the necessary execution context – spawning a thread team and providing thread identification – and the OpenMP memory model facilitates implicit communication between these threads, the construct itself does not inherently distribute work. Without any additional directives, the structured block of a parallel region is executed redundantly by every thread in the team [20]. To transition from this redundant behavior, OpenMP provides worksharing directives. These directives exploit data parallelism by dividing the overall workload into distinct chunks, assigning them to the threads within the active execution context.

In SPMD-based PPMs like MPI, such workload distribution is traditionally realized through explicit *data* distribution. Patterns such as `MPI_Scatter` divide a global data array into discrete chunks and send them to the individual executing ranks, often followed by an `MPI_Gather` operation to collect the partial results back to a root rank (see Section 2.1.1). In contrast, OpenMP's shared-memory model relies on *control flow* distribution. Because all threads inherently share the same memory space, OpenMP worksharing constructs do not need to move or distribute data; instead, they partition the execution of the structured block itself across the available threads. Similar to the `parallel` construct, OpenMP worksharing regions natively imply a synchronization point at their end, forcing all threads to wait at an implicit barrier before the execution continues.

Worksharing-Loops

The most prominent worksharing mechanism in OpenMP is the distribution of `for`-loops (in C/C++) [20]. The worksharing-loop construct divides the iteration space of a loop into distinct chunks and assigns them to the threads within the current execution context. Consequently, instead of every thread executing the entire loop, each thread processes only a specific subset of the total iterations. Figure 4.6 demonstrates the C representation of this construct using the `#pragma omp for` directive nested inside a parallel region.

By default, the data-scoping rules of a worksharing-loop remain in sync with its enclosing parallel region, unless explicitly overridden by data-sharing attribute clauses (as discussed in Section 4.1.1). A particularly relevant clause in the context of worksharing-loops is the `reduction` clause. It allows threads to compute local partial results, which are subsequently aggregated with the variable's original value upon completion of the region. For a standard addition, the global reduction result is computed as $sum_{result} = sum_{original} + \sum sum_{private}$, where $sum_{private}$ represents the private, thread-local partial sums.

```
omp.wsloop schedule(Static) nowait {
    omp.loop_nest (%iv) : index = (%lb) to (%ub)
       step (%step) {
        // ...
        omp.yield
    }
    omp.terminator
}
```

Figure 4.6.: Representation of a worksharing-loop in the `omp` dialect

In distributed-memory models like MPI, achieving a comparable result typically requires an explicit data distribution step (e.g., `MPI_Scatter`) followed by a collective aggregation (e.g., `MPI_Allreduce`). OpenMP's shared-memory model renders the distribution step obsolete, as all threads already operate on the same address space. Furthermore, the construct's execution behavior can be fine-tuned using the `schedule` and `nowait` clauses. The `schedule` clause dictates the iteration distribution policy for workload balancing, and the `nowait` clause eliminates the implicit barrier at the end of the worksharing region.

In the `omp` dialect, the semantics of a worksharing-loop are structurally decoupled into two distinct operations. The `omp.wsloop` operation acts as a wrapper that captures the worksharing semantics and OpenMP-specific clauses. Nested directly inside its region is the `omp.loop_nest` operation, which defines the fundamental loop properties such as lower bounds, upper bounds, and step sizes. Figure 4.7 illustrates this composite representation.

```
omp.parallel {
    omp.wsloop {
        // ...
    }
}
```

Figure 4.7.: Nested `parallel` and worksharing-loop constructs in the `omp` dialect

The execution control clauses are mapped directly to the `omp.wsloop` wrapper. Similar to `proc_bind` in the `parallel` construct, the `schedule` policy is attached as a compile-time attribute. The `nowait` semantic is explicitly represented by the presence of a `nowait` attribute. Reductions are modeled using a symbolic reference: the `reduction` clause on the `omp.wsloop` operation refers to a separate `omp.reduction.declare` operation, which isolates the initialization and combination logic for the specified reduction operator.

Single, Master, and Masked

The `single` construct is formally classified as a worksharing directive because it fulfills the structural characteristics of this category: it is bound to the execution context of a parallel region and concludes with an implicit barrier. Conceptually, however, it does not distribute work across the team. Instead, the construct dynamically selects a single encountering thread to execute its structured block at runtime, while the other threads bypass the block and wait at the implicit barrier, unless a `nowait` clause is specified. This behavior can, for example, be used for executing unparallelizable tasks within an active parallel region – such as thread-unsafe I/O operations or initializing shared data structures – without the overhead of tearing down and recreating the thread team.

A related concept is provided by the `master` construct, which strictly restricts execution of its structured block to the initial thread (i.e., `thread_num` = 0). Notably, unlike `single`, it does not imply a synchronization barrier at its end. In recent OpenMP specifications (since version 5.1), the `master` construct has been deprecated and replaced by the more generic `masked` construct, which allows a subset of threads to enter the region via the `filter` clause.

Ultimately, these constructs complete the spectrum of execution behaviors within a parallel context: while a pure parallel region causes all threads to redundantly execute the code, and worksharing-loops distribute the workload among them, the `single` and `masked` constructs restrict the execution to exactly one or a specifically filtered subset of threads.

4.1.3. Scope Limitations

While the previous sections analyzed the core OpenMP constructs required for the foundational `omp` to `spmd` translation, the OpenMP specification encompasses a vast array of additional directives. This section briefly outlines constructs that conceptually align with the SPMD paradigm but are left for future work, as well as features that fundamentally differ from the proposed IR abstraction and are therefore considered out of scope.

The `teams` construct extends the traditional fork-join principle to better accommodate multi-layered parallelism. Instead of spawning a single thread team, it generates a *league of teams*, acting primarily as a structural container for nested `parallel` constructs. The teams within a league are strictly isolated: they belong to separate contention groups, do not share a synchronized address space, and cannot perform global barriers to synchronize with one another. Conceptually, this isolation between multiple execution contexts shares similarities with the use of

disjoint communicator groups in MPI (a visual representation of this hierarchy is provided in Appendix A.2).

Mutual Exclusion. The `critical` construct and OpenMP runtime locks provide mutual exclusion semantics, ensuring that a specific structured block or shared resource is accessed by exactly one thread at a time. While mutual exclusion is inherently tied to shared-memory data protection, the underlying concept is not exclusive to it. Distributed memory PPMs also employ mutual exclusion mechanisms, such as window locks in MPI's one-sided communication interface. Consequently, translating OpenMP's `critical` and locking semantics into the `spmd` dialect is conceptually a natural candidate for future extension and further discussion and implementation of these constructs are deferred to future work (see Section 6.1).

In addition to the aforementioned features, several other OpenMP paradigms were considered in the context of this work. However, they were deemed unfeasible for the proposed design approach (see Section 4.2) due to a fundamental lack of semantic similarity with the SPMD model.

Foremost is the OpenMP tasking model. Unlike the fork-join paradigm, which relies on a statically sized team of threads with explicit, constant identifiers, tasking is designed around the dynamic creation of dependent, deferred work units. This dynamic execution model inherently clashes with the static rank and communication structures typical in SPMD-based PPMs. Similarly, the `target` construct introduces an entirely different architectural paradigm. It relies on an explicit data transfer model via `map` clauses to orchestrate execution across two distinct memory address spaces: the host and an accelerator device. This separation introduces a completely new conceptual level that goes beyond a unified shared memory, which would likely require a fundamental extension of the underlying SPMD model itself. Finally, while the `simd` construct also exploits data parallelism, it operates on a fundamentally different abstraction level. SIMD directives dictate the utilization of hardware-level vector lanes rather than independent execution threads, and therefore occur below the conceptual scope of ranks.

4.2. Design

In the previous section, we provided an in-depth analysis of the selected OpenMP subset. In this section, we propose a conceptual design for representing OpenMP within the SPMD IR. The semantic and structural findings from the directive analysis serve as the foundational baseline for the mapping strategies proposed herein.

The architecture of the SPMD IR was originally designed to abstract SPMD-based programming models, such as MPI and OpenSHMEM. Consequently, the primary challenge addressed in this design is bridging the conceptual gap between these distributed communication models and OpenMP's shared-memory parallelism. This mapping approach is conceptually feasible because OpenMP's fundamental fork-join paradigm dictates a redundant execution mechanism across a team of threads, where each thread can be uniquely identified by its thread number and synchronizes via collective barriers. To represent the expression of OpenMP within the IR, a critical global design decision was made: OpenMP threads are abstracted as ranks, and thread teams are represented as communicators. This core abstraction embeds the OpenMP execution model into the IR's existing unification layer. The representation of OpenMP's shared-memory semantics is addressed by making data flow between execution contexts explicit within the IR, thereby reconciling shared-memory communication with the IR's distributed-memory framework (detailed in Section 4.2.1).

Building upon this abstraction, the proposed mapping strategy is guided by three core design principles:

1. **Minimal semantic loss:** The structural representation of OpenMP constructs must retain as much information as possible from the original directives. Minimizing the loss of semantics ensures that crucial details remain available for subsequent compiler analysis passes and future code generation.

2. **Minimally invasive:** The mapping strategy should maximize the reuse of existing standard MLIR operations and established `spmd` dialect primitives, avoiding the unnecessary proliferation of new, highly specific operations.

3. **Unification:** Any newly introduced operations must be designed with unification in mind, meaning they should ideally serve as common abstractions that benefit both shared and distributed memory paradigms.

In practice, fulfilling all these criteria simultaneously is not always possible. Therefore, the proposed representations strive to find pragmatic compromises that balance these objectives in an effective manner.

The remainder of this section details the application of these design principles. Section 4.2.1 explores the mapping of the fork-join paradigm, which establishes the primary structural foundation for the OpenMP representation. Building on this, Section 4.2.2 details the integration of worksharing mechanisms, focusing in particular on worksharing-loops and the `single` construct. Note that synchronization mechanisms are conceptually intertwined with these execution paradigms and are therefore discussed directly within their respective sections. Finally, Section 4.2.3 provides a brief outlook on the fundamental concepts of modeling hybrid parallelism

(e.g., MPI+OpenMP). While integrating hybrid OpenMP models is beyond the scope of this work, this section outlines how the newly introduced abstractions can pave the way for future unification.

4.2.1. Fork-Join Modeling

The following sections detail the design of a new spmd.parallel operation, covering its structural representation, execution model, synchronization semantics, and data-scoping mechanisms.

spmd.parallel as a Region-Based Operation

To accurately represent OpenMP within the SPMD IR, the fundamental design must encapsulate the paradigm's core execution model: fork-join parallelism (as analyzed in Section 4.1.1). Fork-join semantics must be represented with a region-based operation, as OpenMP's execution model relies on hierarchical structured blocks to define parallel regions. Modeling the `omp.parallel` construct with the existing flat, non-region-based operations would force the IR to explicitly model the execution context and data-scoping, which would otherwise be implicitly declared by a region's boundaries. In addition, representing composite constructs (e.g., `#pragma omp parallel for`) – which the `omp` dialect models through nested regions (see Figure 4.7) – would add another layer of complexity. Finally, the SPMD IR's existing analysis capabilities rely on region boundaries. An OpenMP parallel region establishes a new execution context where control-flow can diverge. For instance, nested constructs like `single` or `masked` restrict execution to a subset of the team. Using a region-based operation ensures that the analysis can identify the boundaries of this divergent execution context.

Because the existing SPMD IR operations primarily model flat, API-specific calls, they are non-region-based by design. Instead, the IR relies on operations from the `scf` dialect to model divergent control-flow. However, the `scf` dialect contains a set of general, high-level operations, none of which provide the specialized semantics required to model an OpenMP parallel region (the limitations of generic `scf` operations for OpenMP representations are further discussed in Section 4.2.2). Therefore, a dedicated `spmd.parallel` operation is introduced to the dialect, as shown in Figure 4.8.

Execution Model

In order to represent OpenMP's fork-join execution model, the proposed design maps OpenMP threads directly to the IR's ranks, and teams to communicators.

```
spmd.parallel{
    // ...
}
```

Figure 4.8.: The region-based `spmd.parallel` operation

This mapping leverages that the execution of a parallel region is SPMD-like (see Section 2.1.1), as each thread of a team independently executes the region. Furthermore, the executing threads can be identified their thread numbers that are integers starting at zero. This aligns with the rank identification mechanism in the `spmd` dialect. Therefore, OpenMP's thread identification mechanism (e.g., `omp_get_thread_num`) is mapped to the SPMD IR's `spmd.getRankInComm` operation. It should be noted that, while this mapping accurately represents the concurrent execution within a parallel region, it does not account for OpenMP's sequential execution before and after the region. However, in most cases, OpenMP constructs are bound to a parallel region and therefore accurately represented. Edge cases, such as orphaned worksharing directives which are called without an enclosing `parallel` construct or cases involving constructs deemed out of scope for this work, e.g., `task` or `teams`, require further investigation.

In OpenMP, barriers block a team until they are reached by all threads of the team. Similar to this, the `spmd.barrier` operation forces all ranks to wait until it is reached by all ranks in the given communicator. The proposed design assigns a unique communicator to each `spmd.parallel` region. This unique communicator is then passed to all operations representing OpenMP constructs or API calls nested within the region. Furthermore, OpenMP's `barrier` construct is mapped to the `spmd.barrier` operation. When for example a `barrier` construct is nested within a parallel region, in the IR an `spmd.barrier` will reside within the region of an `spmd.parallel` operation. This accurately represents OpenMP's synchronization behavior, as the `spmd.barrier` forces ranks in the communicator of the parallel region to wait until it is reached by all.

The `spmd.parallel` region's communicator is instantiated before the region and then passed into it as an SSA operand. This structure is demonstrated in Figure 4.9.

The design, shown in the previous figure, uses an `spmd.commWorld` operation (see Section 2.2.2). This approach prevents the IR from differentiating between distinct thread teams, as every communicator assigned to a parallel region will always represent the world communicator, and thus holding all ranks. For instance, when analyzing the execution context of operations, two consecutive `spmd.parallel` operations would be treated as having identical execution contexts.

```
%comm = spmd.commDup(%world_comm)
spmd.parallel(%comm) {
    // ...
}
```

Figure 4.9.: Explicit communicator passing

To resolve this, each region's thread team must be represented by a uniquely identifiable communicator. To create this communicator, the proposed design leverages the existing `spmd.commDup` operation. This establishes a new execution context for each parallel region, as the communicator handles are separated by distinct SSA values (see Figure 4.10).

```
%comm1 = spmd.commDup(%world_comm)
spmd.parallel(%comm1) {
    // ...
}

%comm2 = spmd.commDup(%world_comm)
spmd.parallel(%comm2) {
    // ...
}
```

Figure 4.10.: Isolating execution context using `spmd.commDup`

Ensuring that *orphaned directives* – constructs that appear in a separate function called from a parallel region – receive the communicator poses a structural challenge, as their position prevents the explicit passing of the communicator's SSA value without altering function signatures. Therefore, a dedicated operation is introduced to dynamically retrieve the active communicator from the implicit execution environment of the call tree (see Section 4.3.1).

This approach is unproblematic for the current implementation of the Collectives Verification, as it only checks the distinct SSA operands to compare the execution context. However, this design approach would fail for a future extension that supports hybrid MPI+OpenMP programs, as duplicating the world communicator would incorrectly mix the ranks originating from MPI processes and from OpenMP threads. These implications are further discussed in Section 4.2.3.

Synchronization

In addition to the `barrier` construct, which provides a mechanism for explicit synchronization, OpenMP's `parallel` construct has an implicit barrier at the end of its region, synchronizing all threads before exiting the region. While this implicit barrier could be modeled by appending an explicit `spmd.barrier` after the `spmd.operation`'s region, the design proposes to model this implicit synchronization with the `isBlocking` attribute. An illustration of this design is shown in Figure 4.11.

```
1  spmd.parallel(%comm) <isBlocking = true> {
2      // ...
3  }
```

Figure 4.11.: Implicit synchronization via the `isBlocking` attribute

Furthermore, the IR is instructed to handle `spmd.parallel` as a collective operation (see Section 2.2.2). This design choice integrates the operation in the Collectives Verification pass. It should be noted that unlike most collective operations within the IR, the `spmd.parallel` operation can be called from a diverging branch without violating OpenMP's specification. This is because in OpenMP, the synchronization mechanism of a `parallel` construct only applies to the thread team that is spawned by the encountering thread. Execution units not entering the branch with the parallel region would therefore not be affected by its implicit barrier. However, a program triggering this correct branching behavior would involve either nested parallel regions or hybrid MPI+OpenMP, both out of scope for this thesis.

Shared Memory

In MPI's distributed memory model (see Section 2.1.1), communication among ranks requires explicit message passing via collective operations. In contrast, OpenMP threads communicate with concurrent reads and writes to variables in a shared memory address space. Therefore, the design needs to represent OpenMP's shared memory and data-scoping rules.

In the proposed design, all variables allocated within a parallel region are handled as if they are located in a distributed memory space and remain rank-local. This aligns with OpenMP's default scoping, where variables allocated in a region are private. In addition, references to variables allocated in the enclosing scope are strictly forbidden. The IsolatedFromAbove trait on spmd.parallel enforces this restriction at the IR level, ensuring that no external values can be implicitly captured. Instead, variables for which concurrent access should be granted are

modeled by explicitly passing a memory reference for each variable into the region. Inside the region, those memory references are then visible as block arguments and can then be referenced from within. This representation makes shared memory access apparent in the IR, and subsequent analysis passes can treat all accesses to block arguments as shared accesses. Figure 4.12 shows an example IR using the proposed design for modeling OpenMP's memory model in the `spmd` dialect.

```
1    %x = alloc()
2    spmd.parallel(%comm, %x) <isBlocking = true> {
3    ^bb0(%x_shared):
4        %val = load(%x_shared)  // shared access
5        %y = alloc()            // private alloc
6        // ...
7    }
```

Figure 4.12.: Modeling shared and private memory access in the SPMD IR

Alternatively, shared memory access could be modeled using SPMD IR's RMA mechanisms. In this approach, shared variables could be represented as distributed memory windows, while read and write instructions could be represented with `spmd.get` and `spmd.put`. This would be a feasible alternative, as it would potentially allow the IR to detect data races by reusing the existing data race verification pass for RMA. However, as the proposed design maps threads to ranks, a write by a thread to a shared variable would need to be modeled as an `spmd.put` targeting the memory of other ranks. A concurrent read by another thread would then be a local load from the rank holding the variable. This would be a conflict with the incoming remote write – resulting in a remote data race that cannot be detected by the IR [4].

Clauses

The preliminary analysis (Section 4.1.1) established that the `parallel` construct's behavior can be altered with clauses. In the `omp` dialect, clauses – such as the `shared` clause – are implicitly represented. Others are modeled by additional operations or attributes. Figure 4.13 shows an overview of clauses applicable to the `omp.parallel` operation.

Translating these data-scoping semantics into the SPMD IR requires replacing clause attributes with memory allocations. As shown in the previous figure, the `omp` dialect does not provide a `shared` clause and rather models shared memory variables by referencing a variable declared in the outer scope from within a parallel region. In the `omp` dialect, shared variables are modeled by implicit captures from

```
%shared_var = alloc()
%priv_var = alloc()
%fpriv_var = alloc()
store(42, %fpriv_var)

omp.parallel private(%priv_var -> %arg_priv)
   firstprivate(%fpriv_var -> %arg_fpriv) {
    // %shared_var implicitly captured
}
```

Figure 4.13.: Data-scoping allocations in the `omp` dialect

the enclosing scope. In the `spmd` representation, this implicit capture is made explicit by passing the corresponding values as operand to `spmd.parallel` and exposing them as block arguments inside the region. Variables explicitly marked as private are modeled by allocating them inside the `spmd.parallel` region. This guarantees that every rank in the region's execution context instantiates its own uninitialized version of the variable.

Furthermore, as there is no explicit capturing of the variable marked as private from the outside scope, shared access is not possible. The `firstprivate` clause is modeled using the same principle with the addition that this time, it is instantiated with the original external value upon entering the region. In order to ensure that each rank within the region's execution context has its own initialized version of the variable, the region captures the value of the original variable.

The `num_threads` clause has an experimental implementation, in which the clause's value is carried over to the `spmd.parallel` operation as an operand. This prevents a possible semantic loss in the conversion process. However, as this value can only be resolved at runtime, it is only of limited relevance for a static analysis. Similarly, the `proc_bind` clause could be modeled as an attribute to the `spmd.parallel` operation, as its value can be derived at compile time. A corresponding implementation is left for future work. Figure 4.14 illustrates the proposed design.

```
spmd.parallel(%comm) num_threads(%threads) <
   proc_bind = spread; isBlocking = true> {
    // ...
}
```

Figure 4.14.: Proposed representation of static configuration clauses

Other clauses such as `if` or `reduction` require a more structural mapping

approach. For instance, the `if` clause alters the execution flow by determining whether a region is executed by a thread team or by a single thread. Mapping the `if` clause's behavior requires modeling the shift in the execution context. If the `if` condition evaluates to false, the parallel region must be executed strictly sequentially. This behavior can be achieved by branching the communicator creation: While the true branch performs a standard `spmd.commDup` to instantiate a team of N ranks, the false branch provides a singleton communicator containing only the current rank. Currently, the IR does not provide an operation for creating a singleton communicator. A future extension could introduce a dedicated `spmd.commSelf` operation, that aligns with MPI's `MPI_COMM_SELF` function. By passing the resulting communication handle to the `spmd.parallel` operation, all nested operations – including barriers and worksharing constructs – can adapt to the team size. Figure 4.15 illustrates this design approach.

```
%world_comm = spmd.commWorld()
if (%condition) {
    %comm = spmd.commDup(%world_comm, N)
} else {
    %comm = spmd.commSelf()
}
spmd.parallel(%comm) <isBlocking = true> {
    // ...
}
```

Figure 4.15.: Proposed dynamic execution context derivation for the if clause

4.2.2. Worksharing Extension

Building on the directive analysis in Section 4.1, this section presents the representation of OpenMP worksharing-loops within the SPMD IR, as well as presenting design strategies for modeling the `single` construct.

Worksharing-Loops

As analyzed in Section 4.1.2, OpenMP worksharing-loops distribute the iteration space across threads via control-flow rather than explicit data distribution as in SPMD-based PPMs. Accurately representing this behavior requires semantics beyond the SPMD IR's existing set of collective operations.

Analogous to the `parallel` construct (Section 4.2.1), a region-based operation is needed for the Multi-Value analysis to track divergent control-flow. The `scf` dialect

provides the `scf.parallel` operation which expresses a loop with independent iterations that may execute concurrently. However, `scf.parallel` lacks a binding to an execution context (e.g., a thread team or communicator), provides no implicit synchronization semantics, and cannot express OpenMP-specific scheduling policies. Therefore, this design extends the SPMD IR with an operation explicitly representing worksharing-loops.

The directive analysis (Section 4.1.2) established that the `omp` dialect represents worksharing-loops using a wrapper operation (`omp.wsloop`) and a nested loop (`omp.loop_nest`). The proposed design adopts this structure for the SPMD IR: `spmd.wsloop` captures the execution context and OpenMP-specific clause semantics, while `spmd.loop` represents the loop-associated iteration space. This separation provides a clear structure which can then be used for validation purposes, such as detecting illegal nesting of worksharing constructs. An additional benefit of this design decision is the simplification of the conversion.

Synchronization follows the same `isBlocking` mechanism as for `spmd.parallel` (see Section 4.2.1). The `nowait` clause is modeled by setting `isBlocking` to `false`. Figure 4.16 illustrates this complete architectural mapping, demonstrating how the `spmd.wsloop` and its nested `spmd.loop` are embedded within the active communicator's parallel region.

```
%comm = spmd.commDup(%world_comm)
    spmd.parallel(%comm) <isBlocking = true> {
        spmd.wsloop(%comm) <isBlocking = true> {
            spmd.loop {
                // ...
            }
        }
    }
```

Figure 4.16.: Structural representation of a worksharing-loop in the SPMD IR

The wrapper design offers a natural structure for representing OpenMP-specific clauses. The `schedule` clause, which dictates the chunking and distribution of loop iterations, can be directly attached as an attribute to the `spmd.wsloop` operation, preserving its semantics for future code generation pipeline extensions. Data-scoping clauses such as `private` and `firstprivate` follow the same allocation strategy as for `spmd.parallel` (see Section 4.2.1), with variables allocated inside the `spmd.wsloop` wrapper while strictly remaining outside the nested `spmd.loop`.

A design approach for representing reductions requires further elaboration. While the translation of the `reduction` clause is not part of the current implementation pipeline, a possible approach for integrating this aggregation would be to use the

`spmd.allreduce` operation by placing it after the nested loop. This placement leverages the structure of the `spmd.wsloop` operation. Partial results computed within the loop are aggregated collectively by `spmd.allreduce` and the execution is synchronized at the end by the `isBlocking` attribute of the wrapper operation. OpenMP dictates that reduction must incorporate the initial value in their final aggregations (see Section 4.1.2). Therefore, a final aggregation, carried out by a single rank, after the `spmd.wsloop` operation is needed. The proposed design is illustrated in Figure 4.17).

```
1  %shared_var = alloc()
2  spmd.parallel(%comm, %shared_var) <isBlocking = true
       > {
3      %priv_var = alloc()
4      spmd.wsloop(%comm) <isBlocking = true> {
5          spmd.loop {
6              // write partial result to %priv_var
7          }
8      }
9      %reduced_val = spmd.allreduce(%comm, %priv_var)
10     %rank = spmd.getRankInComm(%comm)
11     if (%rank == 0) {
12         %shared_var = %shared_var + %reduced_val //
               final accumulation
13     }
14 }
```

Figure 4.17.: Proposed reduction clause representation using `spmd.allreduce`

It should be noted that any appended `nowait` clause would be overridden by the `spmd.allreduce`, as it is a blocking collective operation. Forcing a non-blocking execution to satisfy the `nowait` semantics would require mapping the reduction to a non-blocking collective (e.g., `Iallreduce`) and deferring the synchronization. Because the OpenMP specification mandates that all threads must eventually synchronize before accessing the final reduction variable, determining the correct placement for this deferred synchronization within the IR is challenging. Resolving this conflict between non-blocking reductions and data race prevention requires further consideration and is therefore deferred to future work.

Single

In addition to worksharing-loops, the directive analysis discussed the `single` construct and its representation in the `omp` dialect (see Section 4.1.2). This section discusses potential designs for representing the `single` construct in the SPMD IR. Note that the `single` construct is not implemented in the prototype implementation of this thesis.

A straightforward approach to modeling the single construct is using the existing `scf.if` operation followed by an `spmd.barrier`, as illustrated in Figure 4.18. To ensure that only one rank enters the branch, an MV-seed operation such as `spmd.getRankInComm` could evaluate the branching condition statically (e.g., `rank == 0`). The primary benefit of this approach is that the existing Multi-Value analysis and Collectives Verification would seamlessly support it, and no further modifications to the IR infrastructure would be needed.

```
1    %rank = spmd.getRankInComm(%comm)
2    if (%rank == 0) {
3        // ...
4    }
5    spmd.barrier(%comm)
```

Figure 4.18.: Static `single` construct representation

Notably, this mapping aligns with the semantics of the OpenMP `master` construct, which mandates execution by the primary thread (rank 0). This approach derives a standard MV-seed for the existing Multi-Value analysis without requiring any OpenMP-specific modification to the analysis framework. However, the `master` construct has been deprecated in OpenMP 5.1 [9] and replaced by the `masked` construct. While `masked` operates similarly by default, it introduces the ability to filter the executing threads based on specific IDs (e.g., via the `filter` clause). Because this requires a flexible evaluation of the executing entity, a condition that statically choose a rank (e.g., `spmd.getRankInComm == 0`) cannot accurately represent the generalized `masked` construct.

For `single`, the executing thread is not predetermined at compile time; rather, the first thread to encounter the construct at runtime typically executes it. Consequently, choosing the executing rank by a static rank ID would be incorrect.

Furthermore, utilizing the generic `scf.if` operation limits the IR's ability to identify structural worksharing errors, such as the illegal nesting of a `single` construct within a worksharing-loop, as it cannot distinguish a genuine OpenMP `single` region from user-defined control-flow logic.

Finally, the representation of OpenMP clauses poses a significant challenge.

While the semantic effect of the `nowait` clause can be modeled by simply removing the trailing explicit `spmd.barrier`, private data-sharing clauses complicate the conversion.

Similar to the representation of data-scoping clauses for `spmd.parallel` regions, the `private` clause requires explicit memory re-allocations localized to the construct's scope. Handling this data-scoping within a generic control-flow operation complicates the conversion process, as the same structural mechanisms utilized by the `spmd.parallel` operation cannot be used.

To overcome these limitations, two alternative conceptual approaches are discussed. In accordance with the mapping of the `omp.wsloop` representation, the first alternative proposes the introduction of a region-based `spmd.single` operation (see Figure 4.19). This would accurately represent the `single` construct's execution context and support clause semantics. On the downside, this new operation would duplicate the control-flow logic already provided by `scf.if`, conflicting with the IR's unification approach.

```
1  spmd.single(comm) <isBlocking = true> {
2      // ...
3  }
```

Figure 4.19.: Dedicated `single` construct representation

The second alternative proposes a synthesis that dynamically selects a rank while retaining standard control-flow semantics, as demonstrated in Figure 4.20. This approach retains the straightforward structure of `scf.if` and `spmd.barrier` but introduces a new dynamic selection operation that evaluates to true for exactly one rank at runtime. The dynamic selection could be realized by a `spmd.dynamicComm` operation, which dynamically splits the active team's communicator into a singleton (for the elected rank) and a complement communicator. By retaining the communicator representation of a region's execution context, this new operation aligns with the design strategies for other constructs' IR representations. A simpler solution would be an `spmd.electRank` operation which takes the team's communicator and directly evaluates to a boolean that is true for one dynamically selected rank. Regardless of the chosen selection operation, this approach derives an MV-seed for the Multi-Value analysis, which can then statically deduce that the branch is executed by a single rank (`executionKind` = `One`), even if the specific rank ID is not known at compile time. This ensures a correct representation of the `single` construct's execution semantics, while retaining the unification approach of the IR. However, the structural difficulties for representing data-scoping clauses of the straightforward approach still remain.

Overall, the worksharing design balances retaining OpenMP-specific semantics –

```
%is_elected = spmd.electRank(%comm)
if (%is_elected) {
    // ...
}
spmd.barrier(%comm)
```

Figure 4.20.: Dynamic `single` construct representation

such as scheduling policies and reduction logic – with reusing the IR's established collective and synchronization primitives. However, extending this model to hybrid execution scenarios introduces additional challenges, as discussed in the following section.

4.2.3. Outlook: Hybrid Parallelism

The introduction of the region-based `spmd.parallel` operation opens the possibility of representing hybrid MPI+OpenMP programs by nesting parallel regions. This section explores whether the `parallel` operation is suitable for abstracting parallel execution in hybrid cases and identifies challenges that remain for a complete hybrid representation.

A modeling approach, using nested `spmd.parallel` operations for representing hybrid MPI+OpenMP in the IR is illustrated in Figure 4.21. The underlying PPM abstracted by each `spmd.parallel` operation can be identified via the `usedModel` attribute.

```
spmd.parallel(%commMPI) <usedModel = "MPI"> {
    // ... MPI ranks
    spmd.parallel(%commOpenMP) <usedModel = "OpenMP"
       > {
        // ... OpenMP threads
    }
}
```

Figure 4.21.: Conceptual hybrid MPI+OpenMP execution via nested `spmd.parallel`

In this conceptual figure, each MPI rank (represented by `commMPI`) spawns its own OpenMP thread team (represented by `commOpenMP`). At this level, all threads within a team share the address space of *their* parent MPI rank, but have no direct access to the memory of other MPI ranks. When only modeling OpenMP, the

communicator representing the parallel region's execution context can be derived by applying `spmd.commDup` on the world communicator. However, in a hybrid case, this approach intertwines two fundamentally different memory architectures: distributed-memory for the MPI ranks and shared-memory for the OpenMP threads.

In hybrid MPI+OpenMP programs, each MPI rank owns a physically separate memory space. When an MPI rank encounters a `parallel` construct, the resulting OpenMP thread team shares the memory domain of that specific MPI rank. Threads spawned by different MPI ranks cannot access each other's memory without explicit communication. The proposed design models shared memory by capturing and explicitly passing all shared variables into the region of the `spmd.parallel` operation, granting all ranks within the execution context access to the same memory reference (see Section 4.2.1). In a hybrid scenario, however, the same mechanism is used at both levels of nesting – for the outer MPI region and the inner OpenMP region – despite their fundamentally different memory semantics. The IR cannot distinguish whether a captured value originates from a memory domain shared among all inner ranks (because they are threads of the same MPI rank) or from a distributed memory domain belonging to a different MPI rank. This information is lost when values are passed into the region, making the proposed shared memory design insufficient for hybrid programs. Specifying the semantics for nesting `spmd.parallel` operations is deferred to future work. The challenges of representing shared and distributed memory domains in hybrid programs require further discussion.

4.3. Implementation

The following sections present the prototype implementation of the proposed design for representing OpenMP concepts within the SPMD IR infrastructure[3]. The current pipeline does not support automatic lowering from C/C++ with OpenMP pragmas (e.g., `#pragma omp parallel`) into the `omp` dialect, as the SPMD IR's frontend relies on the Clang compiler, which lowers OpenMP pragmas directly into runtime calls (e.g., `__kmpc_fork_call`) rather than into the `omp` dialect. An alternative approach using LLVM's Flang compiler to lower Fortran programs into the MLIR with the `omp` dialect was investigated [22]. However, the approach was found to be in a prototypical state and was therefore not suitable for this work. Nonetheless, this demonstrates that future support for automatic code lowering is feasible and remains future work. For this thesis, we manually constructed MLIR input programs using the `omp`, `scf`, `memref`, and `llvm` dialects. As illustrated in

[3]The complete source code and development environment for this implementation can be found in the associated Git repository: `https://git-ce.rwth-aachen.de/spmd-ir/spmd.git`, Branch: `OMP_extension`, Commit: 9644d649fc890f04caa7a83f9b70326137565bc6.

Figure 4.22, the conversion pass transforms the OpenMP-specific operations into a representation using the SPMD dialect, after which the program can be further processed by the existing SPMD IR analysis infrastructure.

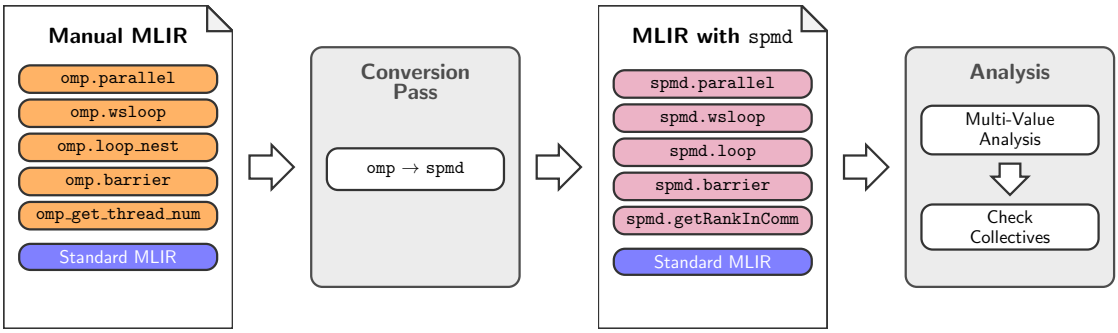

Figure 4.22.: Implementation of the OpenMP to SPMD conversion pipeline

The remainder of this section is structured as follows. Section 4.3.1 introduces the new operations added to the SPMD dialect. Section 4.3.2 presents the conversion pass that transforms `omp` dialect operations into their SPMD counterparts. Finally, Section 4.3.3 describes the adaptations made to the existing Multi-Value analysis and Collectives Verification passes to support the newly introduced operations.

4.3.1. SPMD IR Extensions

The design concepts introduced in Section 4.2 are realized through a set of new operations added to the SPMD dialect. In addition to the operations proposed in the design, several structural operations required by the MLIR framework were introduced. Table 4.2 provides an overview of all new operations; their complete TableGen definitions are provided in Appendix A.3.

Table 4.2.: New operations in the `spmd` dialect

Operation	Key Traits	Region
`spmd.parallel`	`IsolatedFromAbove`, `Collective`	✓
`spmd.wsloop`	`Collective`	✓
`spmd.loop`	—	✓
`spmd.getParallelComm`	—	—
`spmd.terminator`	`Terminator`	—
`spmd.yield`	`Terminator`	—

Two implementation details merit attention beyond what was discussed in the design. First, the `IsolatedFromAbove` trait on `spmd.parallel` not only enforces the data-scoping semantics proposed in Section 4.2.1, but also serves as a safeguard

against MLIR's general-purpose optimization passes. Without this trait, passes such as canonicalization or common subexpression elimination could implicitly capture external values across region boundaries, undermining the explicit data-scoping established during conversion. Second, the `Collective` trait on `spmd.wsloop` and `spmd.parallel` registers the operations with the Collectives Verification pass, allowing them to be recognized as collective operations that require participation of all ranks in the associated communicator. The `spmd.getParallelComm` operation addresses orphaned directives as discussed in Section 4.2.1: it retrieves the active communicator from the implicit execution context when no enclosing `spmd.parallel` region is present in the local scope.

4.3.2. Conversion Pass

The conversion pass is implemented as a partial conversion using MLIR's `applyPartialConversion` infrastructure, where the `scf`, `arith`, `memref`, and `llvm` dialects remain legal and are preserved in the output IR. The converted OpenMP operations are marked as illegal, ensuring that the pass fails if any unconverted instances remain. In addition to the structural conversions discussed below, the pass performs two direct one-to-one mappings: `omp.barrier` is converted to `spmd.barrier` with the addition of a communicator operand, and the OpenMP runtime call `omp_get_thread_num` (represented as `llvm.call` in the input IR) is replaced by `spmd.getRankInComm`.

Communicator Management All conversion patterns require access to an active SPMD communicator. When the operation being converted resides within an already converted `spmd.parallel` region, the communicator is retrieved from the last block argument of the region's entry block. When converting a parallel region at function scope, the global `spmd.commWorld` operation is created (or reused if it already exists) at the function's entry point and subsequently duplicated via `spmd.commDup`. This establishes a distinct execution context for each parallel region, as discussed in Section 4.2.1. For orphaned directives, neither approach is applicable, as no `spmd.parallel` region or global communicator exists in the local scope. In this case, the pass inserts an `spmd.getParallelComm` operation to retrieve the active communicator from the surrounding execution context.

Parallel Region Conversion Because `spmd.parallel` carries the `IsolatedFromAbove` trait, all values defined outside the region that are referenced from within must be explicitly passed as operands. The conversion first collects all such values via MLIR's `getUsedValuesDefinedAbove` utility and passes them as the variadic `args` operand to the new `spmd.parallel` operation. The original region body is

then moved using `inlineRegionBefore`. Since the moved region still references the original external values – which are no longer accessible due to `IsolatedFromAbove` – corresponding block arguments are added to the entry block, one per captured value plus the communicator as the final argument. All uses of the original values within the region are rewritten to reference these block arguments. This mechanism directly implements the shared memory design approach discussed in Section 4.2.1. The structural `isBlocking` attribute proposed in Section 4.2.1 is then added and set to true, modeling a parallel region's implicit barrier. Finally, the `omp.terminator` is replaced by `spmd.terminator`.

Worksharing-Loop Conversion The `omp.wsloop` operation is converted to `spmd.wsloop`, inheriting the communicator from the enclosing `spmd.parallel` region or, for orphaned directives, from an `spmd.getParallelComm` operation. The `nowait` attribute is inverted to match the `isBlocking` semantics (i.e., `nowait = true` results in `isBlocking = false`). The nested `omp.loop_nest` is converted separately to `spmd.loop`, directly transferring bounds and step values. In both cases, region bodies are moved via `inlineRegionBefore`, and `omp.yield` terminators are replaced by `spmd.yield`.

A complete example illustrating the conversion of an `omp.parallel` region to its `spmd.parallel` counterpart is provided in Appendix A.4.

4.3.3. Analysis Pass Adaptations

The existing Multi-Value analysis and Collectives Verification passes were extended to support the newly introduced operations. All adaptations extend the existing pass structure without altering the original analysis logic, thereby maintaining compatibility with the existing analysis framework while adding support for OpenMP-specific execution semantics.

Multi-Value Analysis. The primary challenge in adapting the Multi-Value analysis arose from the `IsolatedFromAbove` trait attached to the `spmd.parallel` operation. This trait dictates that external values must be explicitly passed as block arguments into the region's entry block. However, the existing dataflow solver does not automatically propagate uniformity information across this specific structural boundary, leaving the lattice states – data structures used by the solver to track whether an SSA value is uniform or divergent – uninitialized for these block arguments. Consequently, the original helper functions, such as uniformity queries and the detection of spmd.getRankInComm, performed null-pointer dereferences when attempting to access the missing states. To resolve this, null-safe variants of the affected helper functions were introduced (e.g., `isUniformSafe`), which

explicitly verify the existence of a lattice state before accessing it and conservatively treat values with unresolved states as potentially divergent. Additionally, the traversal now marks all operations nested within an `spmd.wsloop` region as `ExecKind::Dynamic`, making it impossible for the static analysis to predetermine which rank executes which iteration.

Collectives Verification. The Collectives Verification pass required more substantial adaptations to support the new collective operations `spmd.parallel` and `spmd.wsloop`. First, because the pass originally treated collective operations as flat instructions (e.g., without a corresponding nested region), the traversal logic was extended to now also handle region-based collectives, by recursively descending into the region and treating the regions boundary analogously to the `spmd.barrier` operation.

Second, for `spmd.parallel`, the traversal switches to the inner communicator – retrieved from the region's block arguments – ensuring that nested collectives are verified against the execution context provided by the correct communicator. Without this adaptation, the `IsolatedFromAbove` trait would prevent the pass from analyzing any operations inside a parallel region.

Third, the original pass did not account for interprocedural function calls and therefore relied on prior function inlining. Because the conversion pass (introduced in Section 4.3.2) does not provide any inlining mechanisms, handling OpenMP's orphaned directives required an extension of the analysis. Specifically, the adapted pass now recognizes any communicators obtained via `spmd.getParallelComm` as belonging to the enclosing parallel region's execution context.

Finally, a structural nesting validation was introduced to actively detect the illegal nesting of barriers and worksharing constructs within other worksharing regions. To ensure robust detection even for orphaned directives, the pass tracks this contextual constraint via a state flag that is recursively propagated throughout the entire traversal process.

5. Evaluation

In this chapter, we evaluate the extension of the SPMD IR by applying the IR's existing analytical framework to detect OpenMP-specific error cases. Specifically, we use the IR's Collectives Verification to detect error patterns where not all threads reach worksharing constructs or barriers. In addition, we also evaluate the detection of structural nesting violations in OpenMP.

5.1. Methodology

The covered error categories all refer to the OpenMP error classification provided by Münchhalfen et al. [12]. While their work presents an array of relevant error classes, the two that stand out the most are *Barriers not reached by all threads in a team* and *Worksharing constructs not reached by all threads in a team* – also referred to as `bwat` and `wswat` in the remainder of this chapter. These map naturally onto collective errors in the proposed design, because explicit barriers and the implicit barriers at the end of parallel and worksharing regions are modeled as collective operations via the `isBlocking` attribute. Consequently, the Collectives Verification pass can detect these OpenMP errors by checking if all ranks in a communicator reach the same collective operation. Furthermore, cases that represent errors from the class *Invalid nesting of regions* will also be evaluated.

To evaluate the IR's capability of detecting OpenMP errors in practice, a benchmark suite was developed[1]. The suite consists of 26 test cases: 10 featuring `bwat` violations, 9 featuring `wswat` violations, and 7 correct cases that are used to validate the absence of false positives. Rather than forming a standalone category, cases representing the *Invalid nesting of regions* class are embedded directly into the `bwat` and `wswat` datasets. Specifically, they are assigned based on the illegally nested construct: nesting a barrier inside a worksharing region is evaluated as a `bwat` case, while nesting a worksharing construct inside another is evaluated as a `wswat` case.

Because the current pipeline lacks the capability to automatically lower OpenMP compiler directives into the `omp` dialect, all test cases were manually constructed as MLIR programs. Each error case is implemented in two variants: an *inlined*

[1]The benchmark suite can be found in the listed sources of this thesis and in the associated Git repository with the path `extension/omp-benchmark-suite`.

variant, where the OpenMP constructs appear directly in the main function, and a *function-call* variant. The latter exercises the IR's capability of handling orphaned directives via the `spmd.getParallelComm` mechanism. Both variants are expected to yield identical verification results.

5.2. Collectives Verification Results

This section presents the results of the benchmark suite. Following the established methodology, the evaluation is structured along the three test categories: *Barriers without all threads* (`bwat`), *Worksharing without all threads* (`wswat`), and the correct OpenMP programs (*Correct cases*). Throughout this section, individual test cases are referenced by their category abbreviation followed by a numerical index (e.g., `bwat_01` or `wswat_01`).

5.2.1. Barrier without all Threads

Because the `omp.barrier` directive translates directly into an `spmd.barrier` operation, errors from the `bwat` category are represented as collective mismatches within the SPMD IR. Table 5.1 summarizes the verification results for the ten benchmark cases. With the exception of `bwat_04`, which is excluded due to its reliance on the unsupported `critical` construct, the Collectives Verification detects all errors in the inlined variants and eight out of nine in the function-call variants.

Table 5.1.: Evaluation results: Barriers without all threads (bwat)

Variant	Total	TP	FN	N/A
Function-call	10	8	1	1[a]
Inlined	10	9	0	1[a]

[a] Involves the `critical` construct (`bwat_04`), whose conversion is deferred to future work.

The core detection mechanism is best illustrated by cases `bwat_01`, `02`, and `03`, which feature explicit barriers inside thread-dependent conditional branches (see Figure 5.1).

```
1  if (omp_get_thread_num() % 2 == 0) {
2    #pragma omp barrier  // ERROR
3  }
```

Figure 5.1.: `bwat_01`: Barrier within a conditional branch

The existing analysis framework seamlessly handles these scenarios. The Multi-Value analysis evaluates the `spmd.getRankInComm` operation as a multi-valued SSA value, marking the subsequent `scf.if` condition as divergent. As demonstrated in Figure 5.2, the Collectives Verification then finds the `spmd.barrier` operation within this divergent region and correctly flags the collective mismatch.

```
1 %rank = spmd.getRankInComm(%comm) <spmd.executionKind
     = All>
2 scf.if (%rank % 2 == 0) {
3     spmd.barrier(%comm) <spmd.executionKind = Dynamic>
4 } <spmd.executionKind = All, spmd.isMultiValued = true
     >
```

Figure 5.2.: Divergent barrier representation in the SPMD IR after Multi-Value analysis

Data-dependent conditionals, as evaluated in `bwat_05`, are also reliably caught. When a thread's local calculation dictates its control flow, only a subset of threads may reach the nested barrier. The Multi-Value analysis successfully tracks this local dependency and marks the conditional branch as divergent. Consequently, the Collectives Verification detects the collective operation within this divergent region and flags the error.

Furthermore, in cases featuring asymmetric control flow, such as `bwat_06` where one thread spins on a lock while others wait at a barrier, the Multi-Value analysis marks both branches as divergent and the Collectives Verification flags a collective operation within a divergent branch.

Cases featuring orphaned directives are reliably detected (`bwat_07`, `08`). When an `omp.barrier` is hidden within a separate function and is called from a divergent control flow region, the analysis can step into the callee's body and then identify that the orphaned `spmd.barrier` is executed divergently.

However, deeply nested function calls expose a limitation in the current implementation. In the function-call variant of `bwat_09`, the analysis produces a false negative. When the Collectives Verification steps into a deeply nested function call, it is not able to connect the innermost communicator with the communicator of the caller's region. This is likely an implementation limitation rather than a fundamental architectural constraint, as the `spmd.getParallelComm` mechanism is designed to retrieve the enclosing communicator regardless of call depth. Resolving this requires extending the pass to correctly propagate the communicator through multiple levels of function calls. This error is likely the result of a limitation in the implementation and does not reveal a constraint in the proposed design. The Collectives Verification pass recursively descends into the function callees' bodies,

where the `spmd.getParallelComm` operation derives the communicator of the enclosing parallel region. The Multi-Value analysis does not track the `executionKind` attribute through function calls. When the analysis handles a collective operation in the callee's body, it cannot derive whether it is in a divergent execution context from the attributes appended to the operations. To overcome this, the prototype implementation added a parameter to the Collectives Verification that tracks divergent execution context through function calls. The produced false negative shows that the current implementation does not correctly propagate this parameter through multiple levels of function calls, causing the analysis to lose the divergence information before reaching the innermost collective operation.

Finally, structural violations, such as nesting a barrier within a worksharing-loop (`bwat_10`) can be detected using the structural nesting validation introduced in Section 4.3.3. While the existing Multi-Value analysis alone would flag a barrier inside a loop as a collective within a dynamic execution context, the dedicated structural check can provide a more accurate error message that explicitly highlights the prohibited OpenMP nesting behavior. Note that the current design marks all operations within an `spmd.wsloop` region as divergent, which would incorrectly flag legitimate MPI collective operations nested inside an OpenMP worksharing region in a hybrid case.

5.2.2. Worksharing without all Threads

In the SPMD IR, the *Worksharing without all threads* (`wswat`) category is handled similarly to the `bwat` class. Because the `omp.wsloop` operation translates into an `spmd.wsloop` collective, the verification pass applies the same structural rules to check for collective errors as it does for explicit barriers.

Table 5.2 summarizes the verification results. Out of the nine cases, six are evaluated. Case `wswat_04` is excluded due to its reliance on the unsupported `critical` construct, while `wswat_08` and `09` rely on other OpenMP features that currently fall outside the conversion pipeline's scope. For the six applicable cases, the Collectives Verification detects all errors across both the inlined and function-call variants.

Table 5.2.: Evaluation results: Worksharing without all threads (wswat)

Variant	**Total**	**TP**	**FN**	**N/A**
Function-call	9	6	0	3[a,b]
Inlined	9	6	0	3[a,b]

[a] Involves the `critical` construct (`wswat_04`), whose conversion is deferred to future work.
[b] Relies on unsupported features (`wswat_08`, `wswat_09`) that are currently out of scope.

Cases `wswat_01`, `02`, and `03` illustrate this detection. In these scenarios, a worksharing construct is nested within a thread-dependent conditional branch (see Figure 5.3).

```
1  if (omp_get_thread_num() % 2 == 0) {
2      #pragma omp for  // ERROR
3      for (int i = 0; i < N; i++) {
4          data[i] = i * 2;
5      }
6  }
```

Figure 5.3.: `wswat_01`: Worksharing construct nested inside a conditional branch.

As demonstrated in Figure 5.4, the Multi-Value analysis tracks this thread dependency and marks the `scf.if` region as divergent. Consequently, the verification pass flags the nested `spmd.wsloop` as an illegal collective operation.

```
1  %rank = spmd.getRankInComm(%comm) <spmd.executionKind
       = All>
2  scf.if (rank % 2 == 0) {
3    spmd.wsloop(%comm) <spmd.executionKind = Dynamic> {
4      // ...
5  } <spmd.executionKind = All, spmd.isMultiValued = true
       >
```

Figure 5.4.: Divergent worksharing representation in the SPMD IR after Multi-Value analysis

Consistent with the findings from the previous section, data-dependent conditionals (`wswat_05`) are reliably caught. The Multi-Value analysis tracks the local data dependency, marks the conditional branch as divergent, and enables the verification pass to flag the nested `spmd.wsloop`. Furthermore, orphaned worksharing directives hidden within separate function calls (`wswat_06`, `07`) are successfully detected. When a function is called from a divergent control flow region, the analysis steps into the callee's body while preserving the divergent structural context, accurately identifying the illegal collective operation.

5.2.3. Correct Cases

To confirm that the analysis does not flag valid OpenMP programs as errors, the benchmark suite includes seven correct test cases. These semantically valid programs serve as a baseline to verify the absence of false positives.

Table 5.3 summarizes the verification results for this category. The cases cover unconditional barriers, multiple sequential barriers, worksharing-loops with and without the nowait clause, and orphaned directives at various call depths.

Table 5.3.: Evaluation results: Semantically valid programs

Variant	Total	TN	FP	N/A
Function-call	7	7	0	0
Inlined	7	7	0	0

The Collectives Verification successfully processes all seven cases across both the inlined and function-call variants. The analysis correctly identifies the absence of divergent collective executions, resulting in zero false positives (FP). This confirms that the SPMD IR infrastructure reliably differentiates between valid OpenMP execution patterns and actual structural synchronization defects.

5.3. Limitations

While the verification of collectives demonstrates the practical applicability of the proposed design, several limitations remain. First, the existing data race detection pass cannot be applied to detect potential race conditions in OpenMP programs. Further, additional error classes involving OpenMP mutual exclusion semantics, such as locks and critical sections, as presented by Münchhalfen et al. [12], remain unsupported. In addition, the error detection capabilities for unsupported constructs, such as `single`, remain uninvestigated as the current implementation focuses on a subset of OpenMP constructs. Furthermore, the benchmark suite does not cover nested parallel regions nor dynamic team sizes, as these features are not yet supported by the current prototype. Hybrid programming approaches combining both MPI and OpenMP are also not covered, and the current divergence marking within worksharing regions would conflict with legitimate hybrid usage (see Section 5.2.1). Finally, current toolchain constraints limit the pipeline to supporting only MLIR programs that use the `omp` dialect as a starting point for the conversion pass.

6. Conclusion

In summary, this thesis proposed a design for representing the OpenMP programming model within the SPMD IR. A core subset of OpenMP constructs – specifically the `parallel` construct, worksharing-loops, and barriers – can now be represented within the IR's unified abstraction layer. The central contribution of this work is a design that maps OpenMP's fork-join model onto the SPMD IR – abstracting threads as ranks and thread teams as communicators. In addition, to capture OpenMP's hierarchical execution model, region-based operations with explicit data-scoping were introduced to the dialect, alongside initial approaches for representing OpenMP's shared-memory semantics.

To assess the practical applicability of this design, a benchmark suite containing various OpenMP error classes was developed. The results demonstrate that the proposed representation is compatible with the IR's existing analysis framework. Notably, established passes such as the Multi-Value analysis and the collective verification could be reused with minimal adaptations to detect synchronization errors involving worksharing-loops and explicit barriers. Furthermore, the extension of the Collectives Verification to detect invalid nesting of worksharing constructs showcases that the IR's existing infrastructure provides a robust foundation for identifying additional OpenMP-specific error cases.

Beyond the implemented subset, design propositions for further constructs such as `single` and `master/masked` were made, as well as for the `reduction` clause. In addition, initial steps were taken to investigate the support for a potential hybrid MPI and OpenMP representation.

In conclusion, the prototype developed in this work demonstrates that OpenMP's fork-join execution, synchronization, and worksharing semantics can be reconciled with the SPMD IR's distributed-memory abstractions, confirming the viability of the IR's unification approach for shared-memory PPMs.

6.1. Future Work

A complete representation of OpenMP within the SPMD IR remains an open challenge. The following outlines directions for extending the approach presented in this work.

Extended Construct Coverage. Several OpenMP constructs that were conceptually designed in this work remain unimplemented. The `single` and `master/masked` constructs were discussed in Section 4.2 but not realized in the conversion pipeline. Beyond these, OpenMP's mutual exclusion semantics, such as `critical` sections and locks, were investigated as they share conceptual similarities with the lock operations already present in the SPMD IR for RMA synchronization. A preliminary proof-of-concept exists in the form of an experimental `CheckSynchronization` pass, but a full-fledged extension remains an open task. Further OpenMP paradigms such as tasking, device offloading, and SIMD were deemed out of scope for this work (see Section 4.1.3). Additionally, support for nested parallel regions, which require managing hierarchical communicator scoping, remains unexplored.

Data Race Detection. Detecting data races in OpenMP programs requires reasoning about concurrent accesses to shared variables. The SPMD IR's existing data race detection operates exclusively on RMA windows and is therefore not applicable to the shared-memory model proposed in this work. It remains an open question whether OpenMP's shared-memory semantics can be modeled within the IR in a way that enables static data race detection. In particular, since OpenMP's data races occur through cross-thread accesses, which, under the proposed thread-to-rank mapping, translate into conflicts between multiple ranks – a category the current analysis does not cover.

Hybrid Parallelism. An initial design approach for representing hybrid MPI and OpenMP programs was proposed in Section 4.2.3. However, a fundamental challenge remains: the SPMD dialect currently cannot distinguish between shared and distributed memory domains when values are passed into nested `spmd.parallel` regions.

Automatic Lowering Pipeline. A practical limitation of the current prototype is the reliance on manually constructed MLIR input files. In the context of this work, a Flang-based approach by Brown [22] for automatically lowering Fortran programs with OpenMP into MLIR using the OpenMP dialect was investigated. While the approach was not yet mature enough to serve as a reliable frontend for this work, it demonstrated that automatic code generation for this pipeline is feasible in principle.

A. Supplementary Material

A.1. Table for all constructs supported in the `omp` dialect

Table A.1.: Constructs supported in the `omp` dialect

Paradigm	**Constructs**
Fork-Join parallelism	`parallel`, `teams`
Worksharing	`for`, `sections`, `section`, `single`, `scope`, `distribute`, `loop`, `workdistribute`, `scan`
Tasking	`task`, `taskloop`, `task_iteration`, `taskgraph`, `taskyield`
SIMD	`simd`, `declare simd`
Device Offloading	`target`, `target data`, `target enter data`, `target exit data`, `target update`
Synchronization	`barrier`, `critical`, `taskgroup`, `taskwait`, `atomic`, `flush`, `ordered`
Masking	`masked`, `master`
Cancellation	`cancel`, `cancellation point`
Loop-Transformation	`tile`, `fuse`, `unroll`

A.2. Figure illustrating the `teams` construct

A.3. TableGen Definitions for OpenMP Support Operations

All added operations are excerpt from `SPMDOps.td` and can be found in the associated git repository of the SPMD IR or listed under `sourcecode/MyWork/SPMDOps.td` in this thesis' repository.

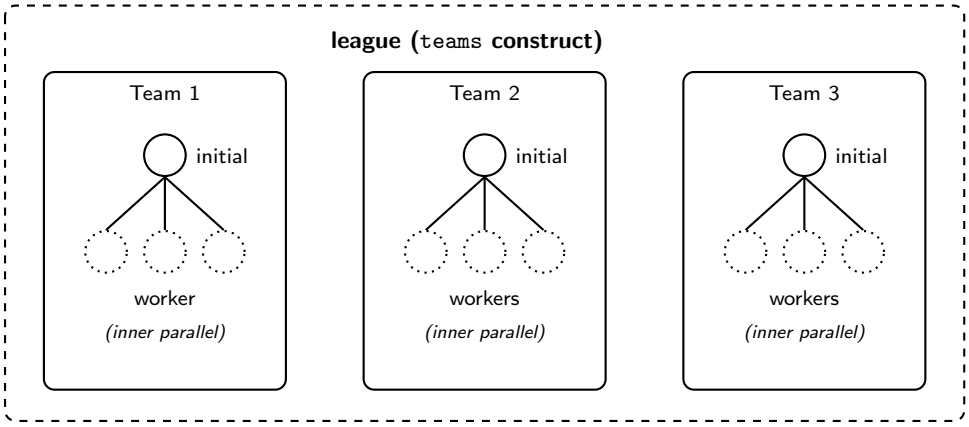

Figure A.1.: Execution model of the OpenMP teams construct

Listing A.1: Parallel region: `spmd.parallel`

```
1
2  def SPMD_ParallelOp : SPMD_Op<"parallel", [
3        Collective,
4        IsolatedFromAbove,
5        SingleBlockImplicitTerminator<"TerminatorOp">,
6        AttrSizedOperandSegments,
7        AutomaticAllocationScope,
8        DeclareOpInterfaceMethods<RegionBranchOpInterface,
9                                  ["
                                      getEntrySuccessorOperands
                                      "]>
10     ]> {
11   let summary = "Defines a parallel region executed by a
         team of ranks";
12   let description = [{
13     The ‘spmd.parallel‘ operation represents a fork-join
           parallel region.
14     A team of ranks executes the enclosed structured
          block.
15
16     The ‘IsolatedFromAbove‘ trait enforces that all
          externally defined
17     values must be explicitly passed via ‘$args‘,
          implementing OpenMP’s
18     data-scoping semantics (shared and firstprivate
```

```
      variables).
19
20    The ‘Collective‘ trait registers this operation with
          the collectives
21    verification pass, as all ranks must reach the
         region boundary.
22  }];
23
24  let arguments = (ins SPMD_CommType:$comm,
25                       Optional<I32>:$num_threads,
26                       Variadic<AnyType>:$args,
27                       BoolAttr:$isBlocking,
28                       I32Attr:$usedModel);
29
30  let regions = (region SizedRegion<1>:$region);
31 }
```

Listing A.2: Worksharing-loop operations: `spmd.wsloop` + `spmd.loop`

```
1  def SPMD_WorkshareLoopOp : SPMD_Op<"wsloop", [
2        Collective,
3        SingleBlock,
4        NoTerminator
5      ]> {
6    let summary = "Worksharing loop wrapper distributing
        iterations among ranks";
7    let description = [{
8      Semantic wrapper for ‘spmd.loop‘ that carries the
         execution context
9      and synchronization semantics of a worksharing loop.
          The ‘isBlocking‘
10     attribute encodes whether an implicit barrier is
         enforced at the end.
11
12     The ‘Collective‘ trait allows the collectives
         verification pass to
13     recognize this operation as requiring participation
         of all ranks.
14   }];
15
16   let arguments = (ins SPMD_CommType:$comm, BoolAttr:
       $isBlocking);
```

```
17    let regions = (region SizedRegion<1>:$region);
18    let skipDefaultBuilders = 1;
19    let builders = [
20      OpBuilder<(ins "::mlir::Value":$comm, "bool":
          $isBlocking)>
21    ];
22  }
23
24  def SPMD_LoopOp : SPMD_Op<"loop", [
25        SingleBlock,
26        SingleBlockImplicitTerminator<"YieldOp">,
27        AttrSizedOperandSegments
28      ]> {
29    let summary = "N-dimensional loop representing the
        iteration space";
30    let description = [{
31      Represents the iteration space of a loop. When
          nested inside an
32      `spmd.wsloop`, iterations are distributed among
          ranks. Supports
33      multi-dimensional bounds for collapse semantics.
          Each dimension
34      introduces a block argument serving as the induction
          variable.
35    }];
36
37    let arguments = (ins Variadic<AnyInteger>:$lowerBounds
        ,
38                         Variadic<AnyInteger>:$upperBounds
                             ,
39                         Variadic<AnyInteger>:$steps);
40
41    let regions = (region SizedRegion<1>:$region);
42    let skipDefaultBuilders = 1;
43    let builders = [
44      OpBuilder<(ins "Value":$lb, "Value":$ub, "Value":
          $step)>,
45      OpBuilder<(ins "ValueRange":$lbs, "ValueRange":$ubs,
          "ValueRange":$steps)>
46    ];
```

```
}
```

Listing A.3: Region terminators: `spmd.terminator` + `spmd.yield`

```
def SPMD_TerminatorOp : SPMD_Op<"terminator", [
    Terminator]> {
  let summary = "Terminates an spmd.parallel region";
  let description = [{
    The `spmd.terminator` operation marks the end of a
       control flow
    block within an `spmd.parallel` region.
  }];

  let arguments = (ins);
  let results = (outs);
}

def SPMD_YieldOp : SPMD_Op<"yield", [Pure, Terminator,
    ReturnLike]> {
  let summary = "Terminates an spmd.loop region";
  let description = [{
    Terminates a block in an `spmd.loop` region and
       optionally passes
    values to the next iteration or as the final result
       of the loop.
  }];

  let arguments = (ins Variadic<AnyType>:$results);

  let builders = [
    OpBuilder<(ins), [{ /* empty yield */ }]>
  ];
}
```

Listing A.4: Communicator retrieval for orphaned directives: `spmd.getParallelComm`

```
def SPMD_GetParallelCommOp : SPMD_Op<"getParallelComm",
    [Pure]> {
  let summary = "Retrieves the active communicator from
     the implicit context";
  let description = [{
    Retrieves the active SPMD communicator from the
```

```
        implicit execution
    context of the enclosing parallel region. Used to
        model orphaned
    OpenMP directives without modifying function
        signatures.
  }];

  let arguments = (ins);
  let results = (outs SPMD_CommType:$res);
}
```

A.4. Conversion Example: OMP Dialect to SPMD Dialect

Note: The following "Output IR" code listings (as in Listings A.6 and A.8) are manually translated into MLIR's Custom Assembly Format for better readability. The native IR outputs are in Generic MLIR format and can be found in the git repository associated with this thesis.

A.4.1. Parallel Region and Worksharing Loop

Listing A.5: Input IR: Standard OpenMP parallel region with a nested worksharing loop

```
module {
  llvm.func @omp_get_thread_num() -> i32

  llvm.func @main() -> i32 {
    %c16_i32 = arith.constant 16 : i32
    %c1_i32 = arith.constant 1 : i32
    %c0_i32 = arith.constant 0 : i32

    %data_ptr = llvm.alloca %c16_i32 x i32 : (i32) -> !
        llvm.ptr
    %result_ptr = llvm.alloca %c16_i32 x i32 : (i32) ->
        !llvm.ptr

    omp.parallel {
      omp.wsloop {
```

```
14          omp.loop_nest (%j) : i32 = (%c0_i32) to (%
               c16_i32) step (%c1_i32) {
15            // result[j] = data[j] + 1;
16            %offset_data_read = llvm.getelementptr %
                 data_ptr[%j] : (!llvm.ptr, i32) -> !llvm.
                 ptr, i32
17            %loaded_val = llvm.load %offset_data_read : !
                 llvm.ptr -> i32
18
19            %add_val = arith.addi %loaded_val, %c1_i32 :
                 i32
20            %offset_result = llvm.getelementptr %
                 result_ptr[%j] : (!llvm.ptr, i32) -> !llvm.
                 ptr, i32
21            llvm.store %add_val, %offset_result : i32, !
                 llvm.ptr
22
23            omp.yield
24          }
25        }
26        omp.terminator
27      }
28      llvm.return %c0_i32 : i32
29    }
30  }
```

Listing A.6: Output IR: Converted SPMD representation demonstrating explicit communicator management and shared memory emulation via block arguments

```
1  module {
2    llvm.func @main() -> i32 {
3      %comm_world = spmd.commWorld : !spmd.comm
4
5      %c16 = arith.constant 16 : i32
6      %c2 = arith.constant 2 : i32
7      %c1 = arith.constant 1 : i32
8      %c0 = arith.constant 0 : i32
9
10     %data_ptr = llvm.alloca %c16 x i32 : (i32) -> !llvm.
          ptr
```

```
11      %result_ptr = llvm.alloca %c16 x i32 : (i32) -> !
            llvm.ptr
12
13      %team_comm, %error = spmd.commDup(%comm_world) : (!
            spmd.comm) -> (!spmd.comm, !spmd.error)
14
15      spmd.parallel(%team_comm, %data_ptr, %c1, %
            result_ptr, %c0, %c16) {
16      ^bb0(%arg_data: !llvm.ptr, %arg_c1: i32, %arg_result
            : !llvm.ptr, %arg_c0: i32, %arg_c16: i32, %
            arg_comm: !spmd.comm):
17        spmd.wsloop(%arg_comm) {isBlocking = true} {
18          spmd.loop (%j) : i32 = (%arg_c0) to (%arg_c16)
                step (%arg_c1) {
19            %offset_data = llvm.getelementptr %arg_data[%j
                ] : (!llvm.ptr, i32) -> !llvm.ptr, i32
20            %loaded_val = llvm.load %offset_data : !llvm.
                ptr -> i32
21
22            %add_val = arith.addi %loaded_val, %arg_c1 :
                i32
23            %offset_result = llvm.getelementptr %
                arg_result[%j] : (!llvm.ptr, i32) -> !llvm.
                ptr, i32
24            llvm.store %add_val, %offset_result : i32, !
                llvm.ptr
25
26            spmd.yield
27          }
28        }
29        spmd.terminator
30      } {isBlocking = true}
31      llvm.return %c0 : i32
32    }
33  }
```

A.4.2. Orphaned Directive Conversion

Listing A.7: Input IR: OpenMP orphaned directive where a barrier is located inside a separate, detached function

```
module {
  llvm.func @omp_get_thread_num() -> i32

  // --- Orphaned Function ---
  llvm.func @do_work_and_sync(%data: !llvm.ptr, %tid:
     i32) {
    %c3_i32 = arith.constant 3 : i32
    %c100_i32 = arith.constant 100 : i32

    // data[tid] = tid * 3
    %mul = arith.muli %tid, %c3_i32 : i32
    %ptr_offset = llvm.getelementptr %data[%tid] : (!
       llvm.ptr, i32) -> !llvm.ptr, i32
    llvm.store %mul, %ptr_offset : i32, !llvm.ptr

    // ORPHANED BARRIER: Lexically detached from the
       parallel region
    omp.barrier

    // data[tid] += 100
    %val = llvm.load %ptr_offset : !llvm.ptr -> i32
    %add = arith.addi %val, %c100_i32 : i32
    llvm.store %add, %ptr_offset : i32, !llvm.ptr

    llvm.return
  }

  llvm.func @main() -> i32 {
    %c8_i32 = arith.constant 8 : i32
    %c0_i32 = arith.constant 0 : i32
    %data_ptr = llvm.alloca %c8_i32 x i32 : (i32) -> !
       llvm.ptr

    // --- OpenMP Parallel Region ---
    omp.parallel {
      %tid = llvm.call @omp_get_thread_num() : () -> i32

      // Interprocedural call passing the execution
         context
```

```
      llvm.call @do_work_and_sync(%data_ptr, %tid) : (!
          llvm.ptr, i32) -> ()

      omp.terminator
    }

    llvm.return %c0_i32 : i32
  }
}
```

Listing A.8: Output IR: Converted orphaned directive utilizing dynamic communicator retrieval alongside proper rank ID mapping

```
module {
  llvm.func @omp_get_thread_num() -> i32

  llvm.func @do_work_and_sync(%arg_data: !llvm.ptr, %
      arg_tid: i32) {
    %c3_i32 = arith.constant 3 : i32
    %c100_i32 = arith.constant 100 : i32

    %mul = arith.muli %arg_tid, %c3_i32 : i32
    %ptr_offset = llvm.getelementptr %arg_data[%arg_tid]
        : (!llvm.ptr, i32) -> !llvm.ptr, i32
    llvm.store %mul, %ptr_offset : i32, !llvm.ptr

    // 1. Dynamic retrieval of the executing team's
       context
    %implicit_comm = spmd.getParallelComm : !spmd.comm
    %error = spmd.barrier(%implicit_comm) {isBlocking =
       true}

    %val = llvm.load %ptr_offset : !llvm.ptr -> i32
    %add = arith.addi %val, %c100_i32 : i32
    llvm.store %add, %ptr_offset : i32, !llvm.ptr

    llvm.return
  }

  llvm.func @main() -> i32 {
    %comm_world = spmd.commWorld : !spmd.comm
```

```
25      %team_comm, %error = spmd.commDup(%comm_world) : (!
            spmd.comm) -> (!spmd.comm, !spmd.error)
26
27      %c8_i32 = arith.constant 8 : i32
28      %c0_i32 = arith.constant 0 : i32
29      %data_ptr = llvm.alloca %c8_i32 x i32 : (i32) -> !
            llvm.ptr
30
31      // 2. Region execution context capturing external
            data
32      spmd.parallel(%team_comm, %data_ptr) {
33      ^bb0(%arg_data: !llvm.ptr, %arg_comm: !spmd.comm):
34
35        // 3. Thread ID conversion utilizing the team
              communicator
36        %rank, %err_rank = spmd.getRankInComm(%arg_comm) :
              (!spmd.comm) -> (i32, !spmd.error)
37
38        llvm.call @do_work_and_sync(%arg_data, %rank) : (!
              llvm.ptr, i32) -> ()
39
40        spmd.terminator
41      } {isBlocking = true}
42
43      llvm.return %c0_i32 : i32
44    }
45  }
```

Abkürzungsverzeichnis

API Application Programming Interface.

GPU Graphics Processing Unit.

HPC High Performance Computing.

IR Intermediate Representation.

MLIR Multi-Level Intermediate Representation.

MPI Message Passing Interface.

MV Multi-Value.

NCCL NVIDIA Collective Communications Library.

NUMA Non-Uniform Memory Access.

NVSHMEM NVIDIA Shared Memory Library.

omp OpenMP MLIR Dialect.

PGAS Partitioned Global Address Space.

PPM Parallel Programming Model.

RMA Remote Memory Access.

SHMEM Shared Memory Library.

SIMD Single Instruction, Multiple Data.

SPMD Single Program, Multiple Data.

SSA Static Single Assignment.

Bibliography

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[2] Timothy G Mattson, Yun Helen He, and Alice E Koniges. *The OpenMP common core: making OpenMP simple again*. MIT Press, 2019.

[3] Semih Burak et al. "SPMD IR: unifying SPMD and multi-value IR showcased for static verification of collectives". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2024, pp. 3–20.

[4] Semih Burak et al. "Extending the SPMD IR for RMA Models and Static Data Race Detection". In: *European MPI Users' Group Meeting*. Springer. 2025, pp. 143–164.

[5] Chris Lattner et al. "MLIR: Scaling compiler infrastructure for domain specific computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14.

[6] *omp Dialect Documentation*. Apr. 2026. URL: `https://mlir.llvm.org/docs/Dialects/OpenMPDialect/`.

[7] Frederica Darema. "The spmd model: Past, present and future". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2001, pp. 1–1.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*. June 2025. URL: `https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf`.

[9] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Specification Version 6.0 November 2024*.

[10] Tim Jammer, Christian Iwainsky, and Christian Bischof. "Survey of OpenMP Practice in General Open Source Software". In: *International Workshop on OpenMP*. Springer. 2024, pp. 97–110.

[11] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[12] Jan Felix Münchhalfen et al. “Classification of common errors in OpenMP applications”. In: *International Workshop on OpenMP*. Springer. 2014, pp. 58–72.

[13] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. “Static validation of barriers and worksharing constructs in OpenMP applications”. In: *International Workshop on OpenMP*. Springer. 2014, pp. 73–86.

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

[15] Pierre Huchant et al. “Multi-valued expression analysis for collective checking”. In: *European Conference on Parallel Processing*. Springer. 2019, pp. 29–43.

[16] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1993.

[17] Melvin E Conway. “A multiprocessor system design”. In: *Proceedings of the November 12-14, 1963, fall joint computer conference*. 1963, pp. 139–146.

[18] Jack B Dennis and Earl C Van Horn. “Programming semantics for multiprogrammed computations”. In: *Communications of the ACM* 9.3 (1966), pp. 143–155.

[19] Matteo Frigo, Charles E Leiserson, and Keith H Randall. “The implementation of the Cilk-5 multithreaded language”. In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 212–223.

[20] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[21] *MLIR: Code Documentation*. Apr. 2026. URL: `https://mlir.llvm.org/docs/`.

[22] Nick Brown. “Fully integrating the Flang Fortran compiler with standard MLIR”. In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2024, pp. 939–949.