SPMD IR: Unifying SPMD and Multi-value IR Showcased for Static Verification of Collectives

Semih Burak, I. Ivanov, J. Domke, M. Müller

EuroMPI 2024
25 - 27.09.24

Motivation

- HPC systems get more complex and heterogeneous; the software and hardware

Motivation

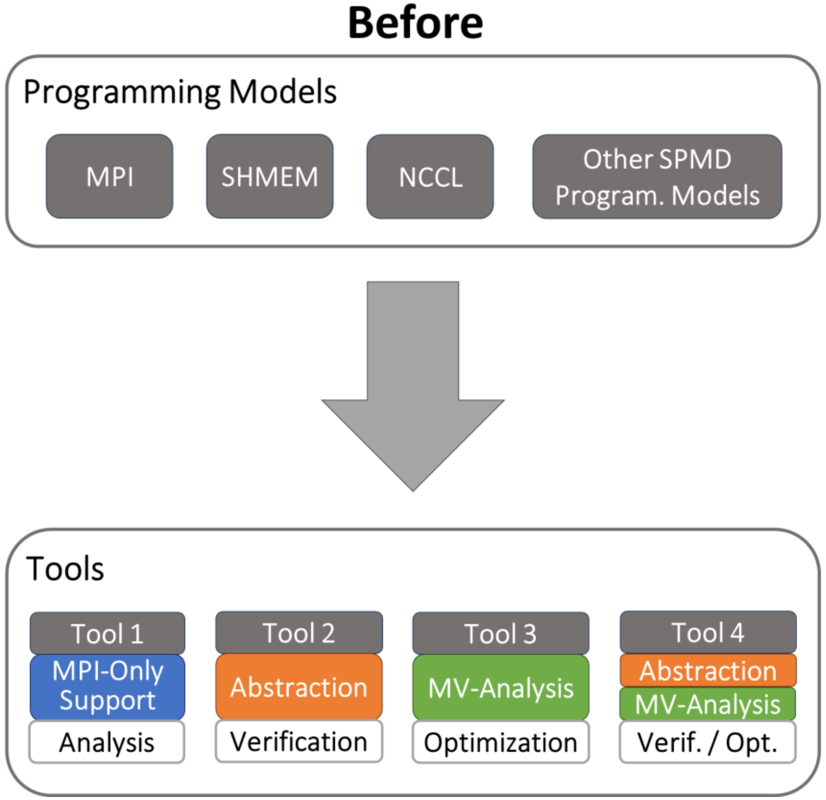
- HPC systems get more complex and heterogeneous; the software and hardware
- For effective utilization of HPC clusters,
  - inter-node comm. and SPMD programming models such as MPI are inevitable

Motivation

- HPC systems get more complex and heterogeneous; the software and hardware
- For effective utilization of HPC clusters,
  - inter-node comm. and SPMD programming models such as MPI are inevitable
- There exist different programming models with distinct specialization but with common features

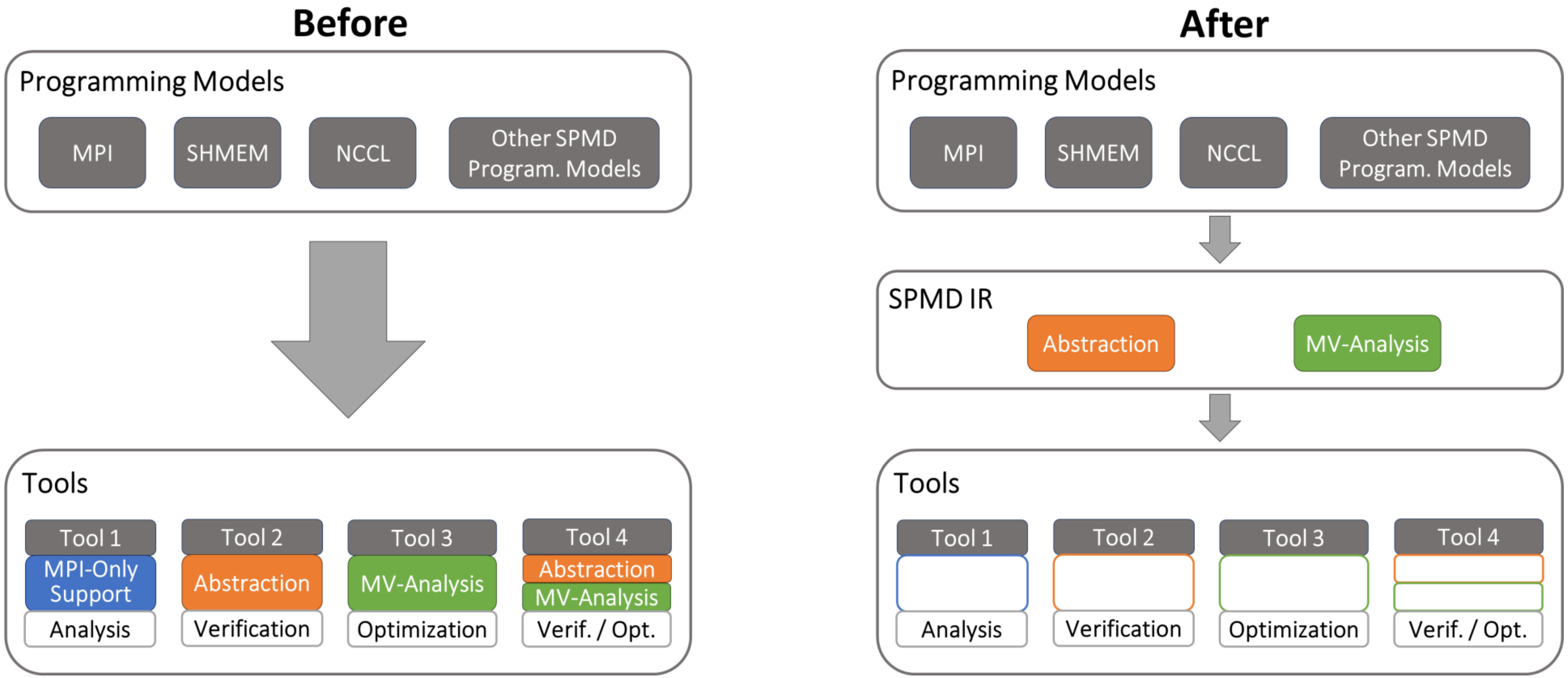
Motivation

- HPC systems get more complex and heterogeneous; the software and hardware
- For effective utilization of HPC clusters,
  - inter-node comm. and SPMD programming models such as MPI are inevitable
- There exist different programming models with distinct specialization but with common features



Motivation

- HPC systems get more complex and heterogeneous; the software and hardware
- For effective utilization of HPC clusters,
  - inter-node comm. and SPMD programming models such as MPI are inevitable
- There exist different programming models with distinct specialization but with common features



Before

Programming Models

MPI

SHMEM

NCCL

Other SPMD Program. Models

Tools

Tool 1

MPI-Only Support

Analysis

Tool 2

Abstraction

Verification

Tool 3

MV-Analysis

Optimization

Tool 4

Abstraction

MV-Analysis

Verif. / Opt.

After

Programming Models

MPI

SHMEM

NCCL

Other SPMD Program. Models

SPMD IR

Abstraction

MV-Analysis

Tools

Tool 1

Analysis

Tool 2

Verification

Tool 3

Optimization

Tool 4

Verif. / Opt.

Outline

- Background
  - MLIR
  - Multi-Values

Outline

- Background
  - MLIR
  - Multi-Values
- SPMD IR
  - Extended Multi-Value Analysis
  - Scope
  - Example
  - Workflow

Outline

- Background
  - MLIR
  - Multi-Values

- SPMD IR
  - Extended Multi-Value Analysis
  - Scope
  - Example
  - Workflow

- Evaluation
  - Static Verification of Collectives
  - Comparison to PARCOACH

Outline

- Background
  - MLIR
  - Multi-Values

- SPMD IR
  - Extended Multi-Value Analysis
  - Scope
  - Example
  - Workflow

- Evaluation
  - Static Verification of Collectives
  - Comparison to PARCOACH

- Discussion

Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO ’21

- IR and compiler framework building upon and part of LLVM, published 2021

Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO '21

- IR and compiler framework building upon and part of LLVM, published 2021

- The purpose of IRs in compilers is generally to facilitate optimizations
  - before lowering to low-level or machine code

Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO ’21

- IR and compiler framework building upon and part of LLVM, published 2021

- The purpose of IRs in compilers is generally to facilitate optimizations
  - before lowering to low-level or machine code

- Compared to LLVMIR, MLIR
  - is more extensible, reusable
  - can mix low-level and high-level abstractions

Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO '21

- IR and compiler framework building upon and part of LLVM, published 2021

- The purpose of IRs in compilers is generally to facilitate optimizations
  - before lowering to low-level or machine code

- Compared to LLVMIR, MLIR
  - is more extensible, reusable
  - can mix low-level and high-level abstractions
  - can encapsulate domain-specific semantics by dialects
    - a collection of custom operations, types, and attributes

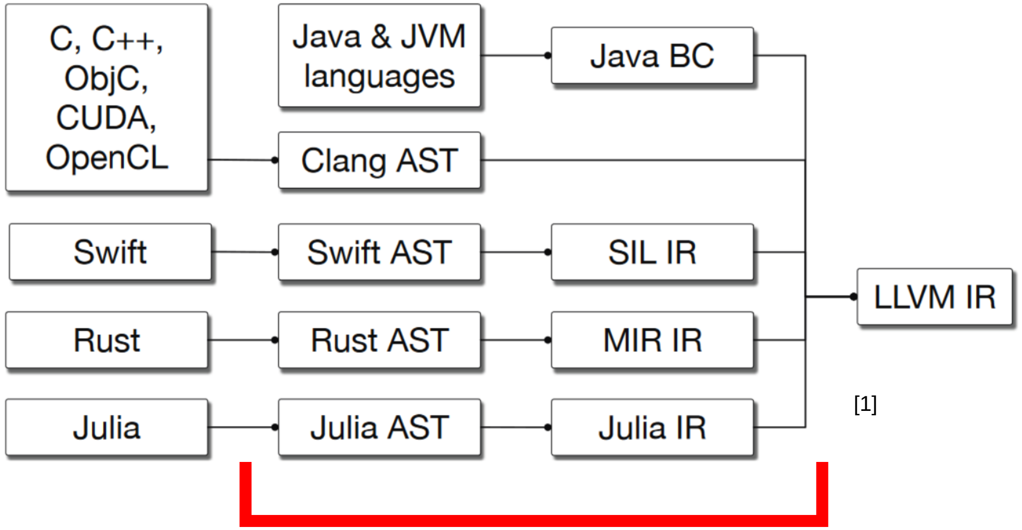
Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO '21

- IR and compiler framework building upon and part of LLVM, published 2021

- The purpose of IRs in compilers is generally to facilitate optimizations
  - before lowering to low-level or machine code

- Compared to LLVMIR, MLIR
  - is more extensible, reusable
  - can mix low-level and high-level abstractions
  - can encapsulate domain-specific semantics by dialects
    - a collection of custom operations, types, and attributes



MLIR also helps here, replacing custom high-level IRs

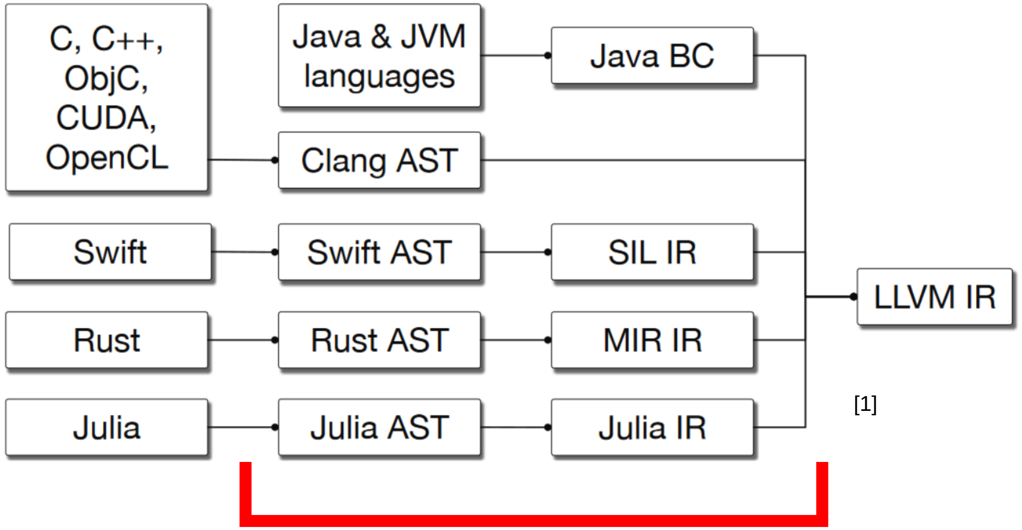
Background: MLIR (Multi-Level Intermediate Representation)

[1] Lattner et al: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO '21

- IR and compiler framework building upon and part of LLVM, published 2021

- The purpose of IRs in compilers is generally to facilitate optimizations
  - before lowering to low-level or machine code

- Compared to LLVMIR, MLIR
  - is more extensible, reusable
  - can mix low-level and high-level abstractions
  - can encapsulate domain-specific semantics by dialects
    - a collection of custom operations, types, and attributes



- Idea:
  - Express SPMD-related information in its own dialect and a set of attributes
  - Next to other special-purpose dialects, e.g., for shared-memory parallelism

MLIR also helps here, replacing custom high-level IRs

Background: MLIR (2)

```
func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

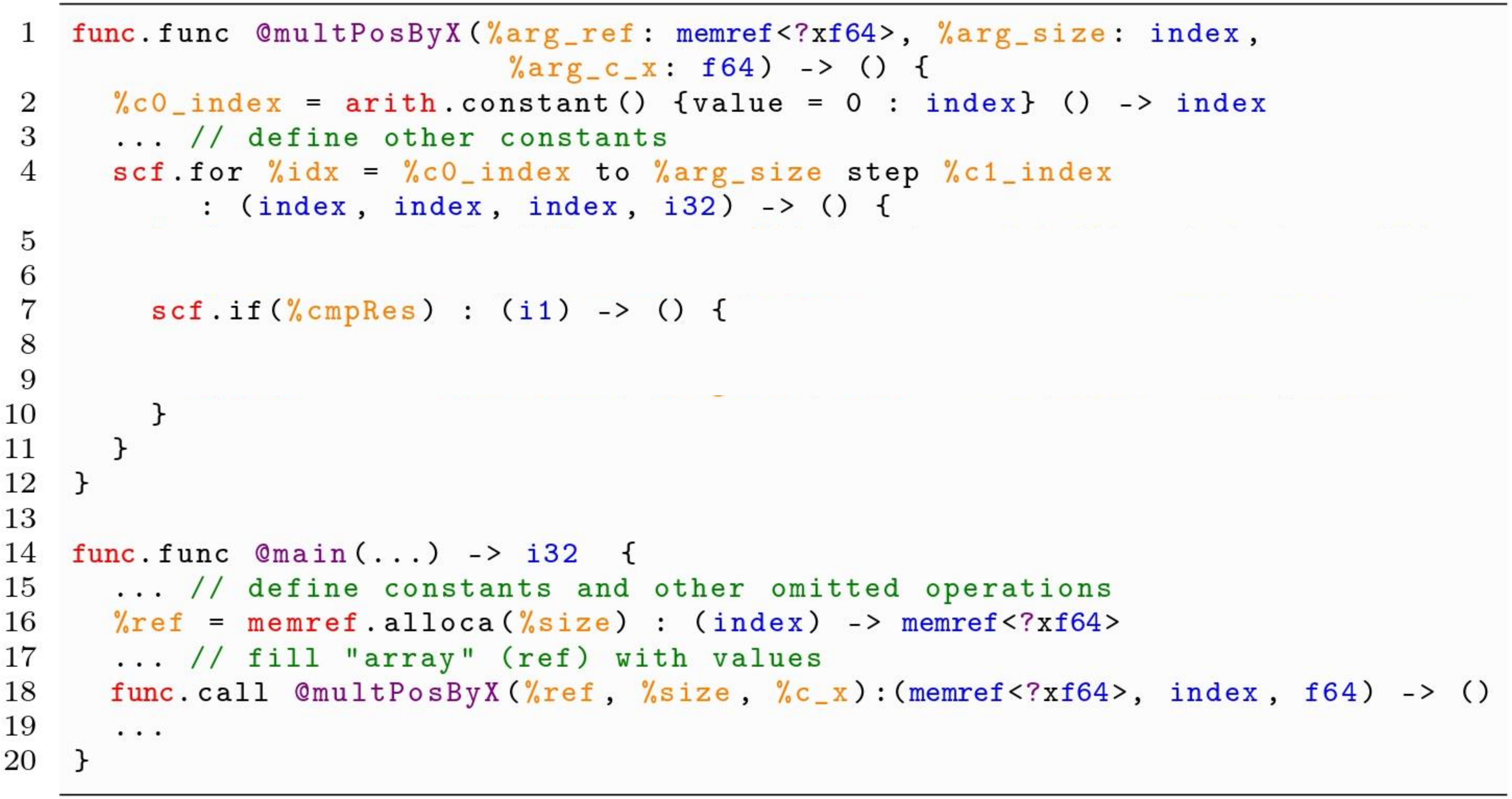
- Red highlights the dialects
- Orange the values
- Blue the types

Background: MLIR (2)

```
func.func @multPosByX(%arg_ref: memref<?xf64>, %arg_size: index,
                              %arg_c_x: f64) -> () {
  %c0_index = arith.constant() {value = 0 : index} () -> index
  ... // define other constants

}

func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

- Red highlights the dialects
- Orange the values
- Blue the types

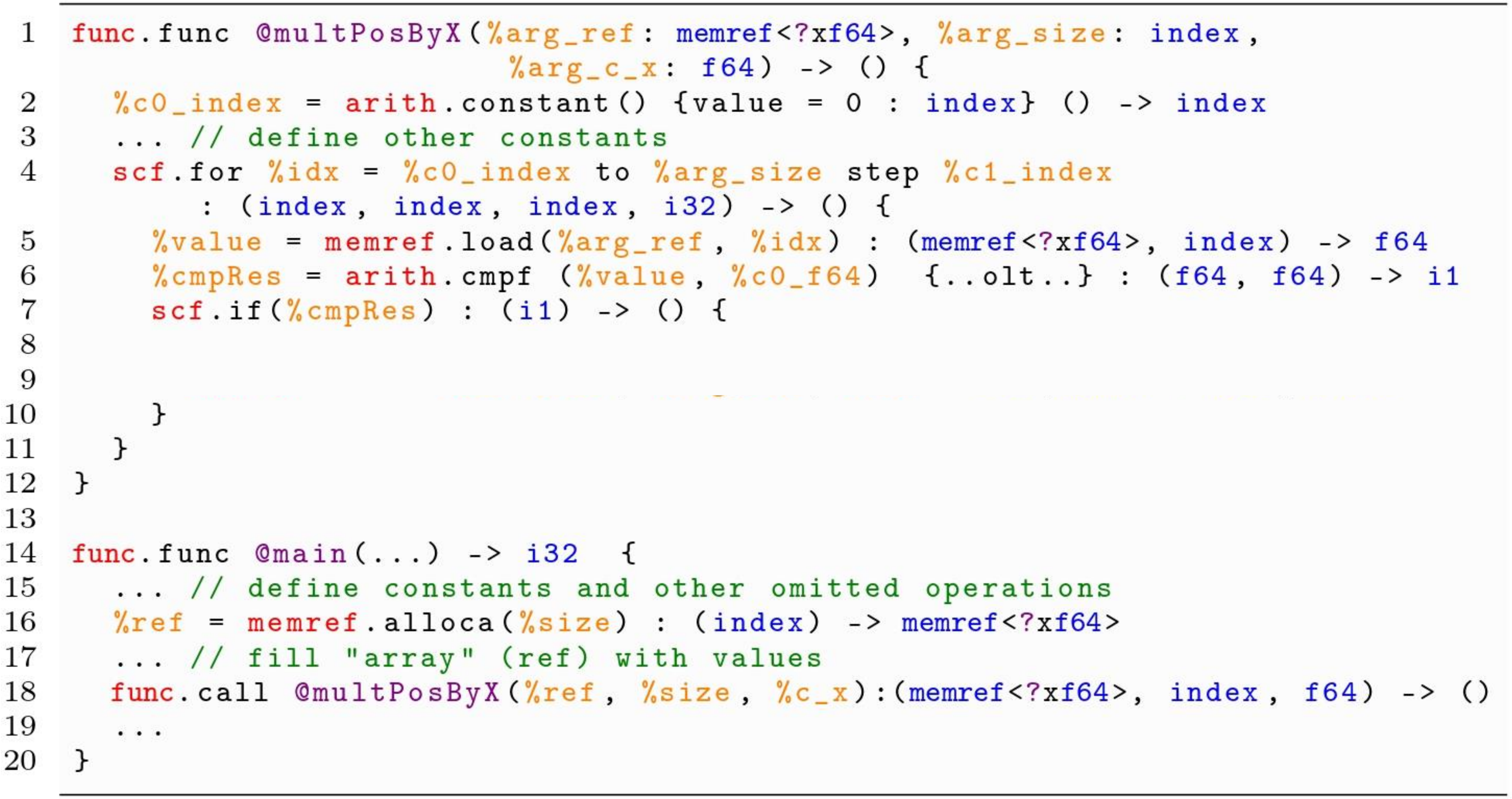
Background: MLIR (2)



```
func.func @multPosByX(%arg_ref: memref<?xf64>, %arg_size: index,
                                %arg_c_x: f64) -> () {
  %c0_index = arith.constant() {value = 0 : index} () -> index
  ... // define other constants
  scf.for %idx = %c0_index to %arg_size step %c1_index
       : (index, index, index, i32) -> () {


    scf.if(%cmpRes) : (i1) -> () {


    }
  }
}

func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

- Red highlights the dialects
- Orange the values
- Blue the types

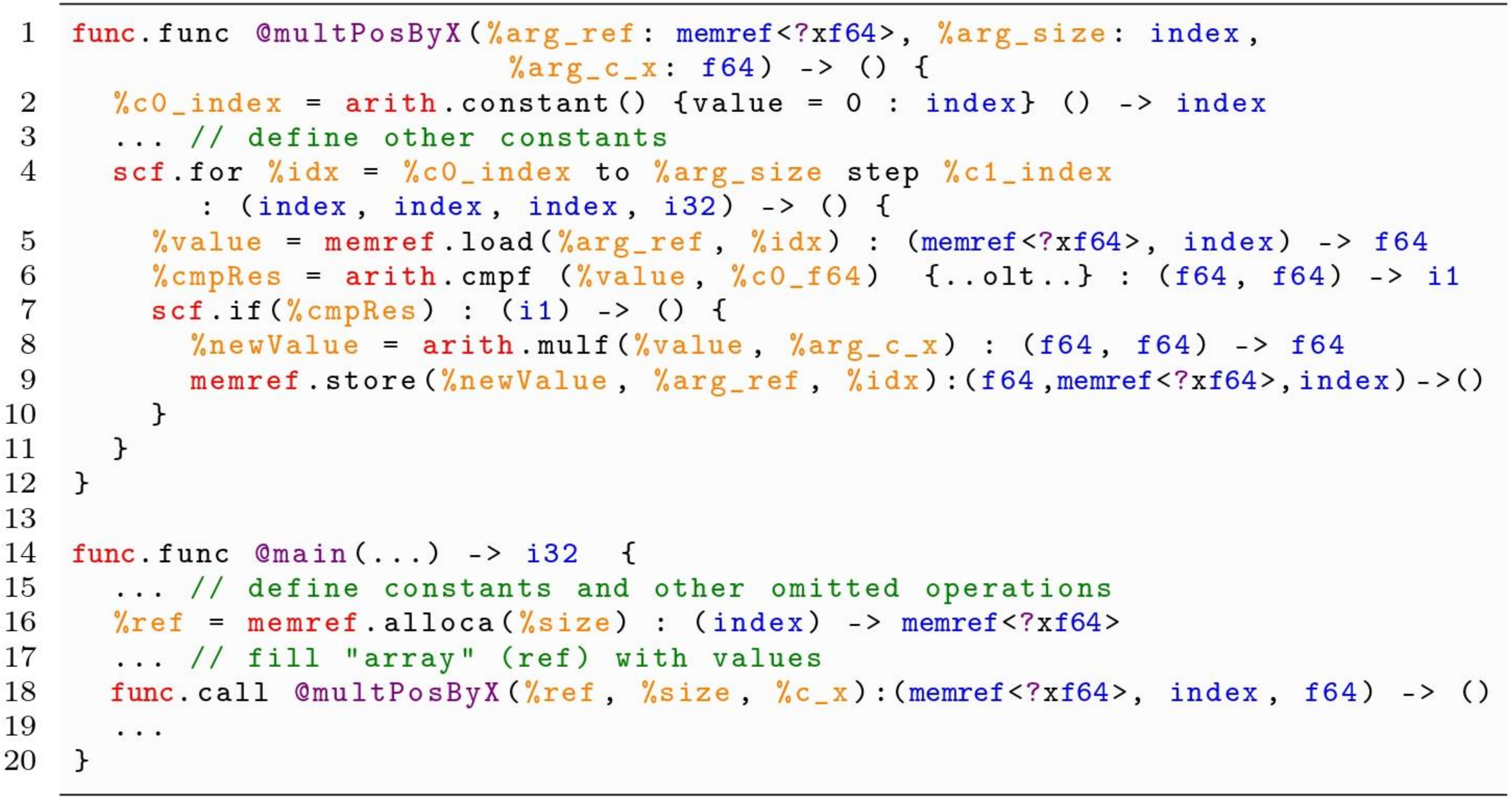
Background: MLIR (2)



```
func.func @multPosByX(%arg_ref: memref<?xf64>, %arg_size: index,
                              %arg_c_x: f64) -> () {
  %c0_index = arith.constant() {value = 0 : index} () -> index
  ... // define other constants
  scf.for %idx = %c0_index to %arg_size step %c1_index
        : (index, index, index, i32) -> () {
    %value = memref.load(%arg_ref, %idx) : (memref<?xf64>, index) -> f64
    %cmpRes = arith.cmpf (%value, %c0_f64)  {..olt..} : (f64, f64) -> i1
    scf.if(%cmpRes) : (i1) -> () {


    }
  }
}

func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

- Red highlights the dialects
- Orange the values
- Blue the types

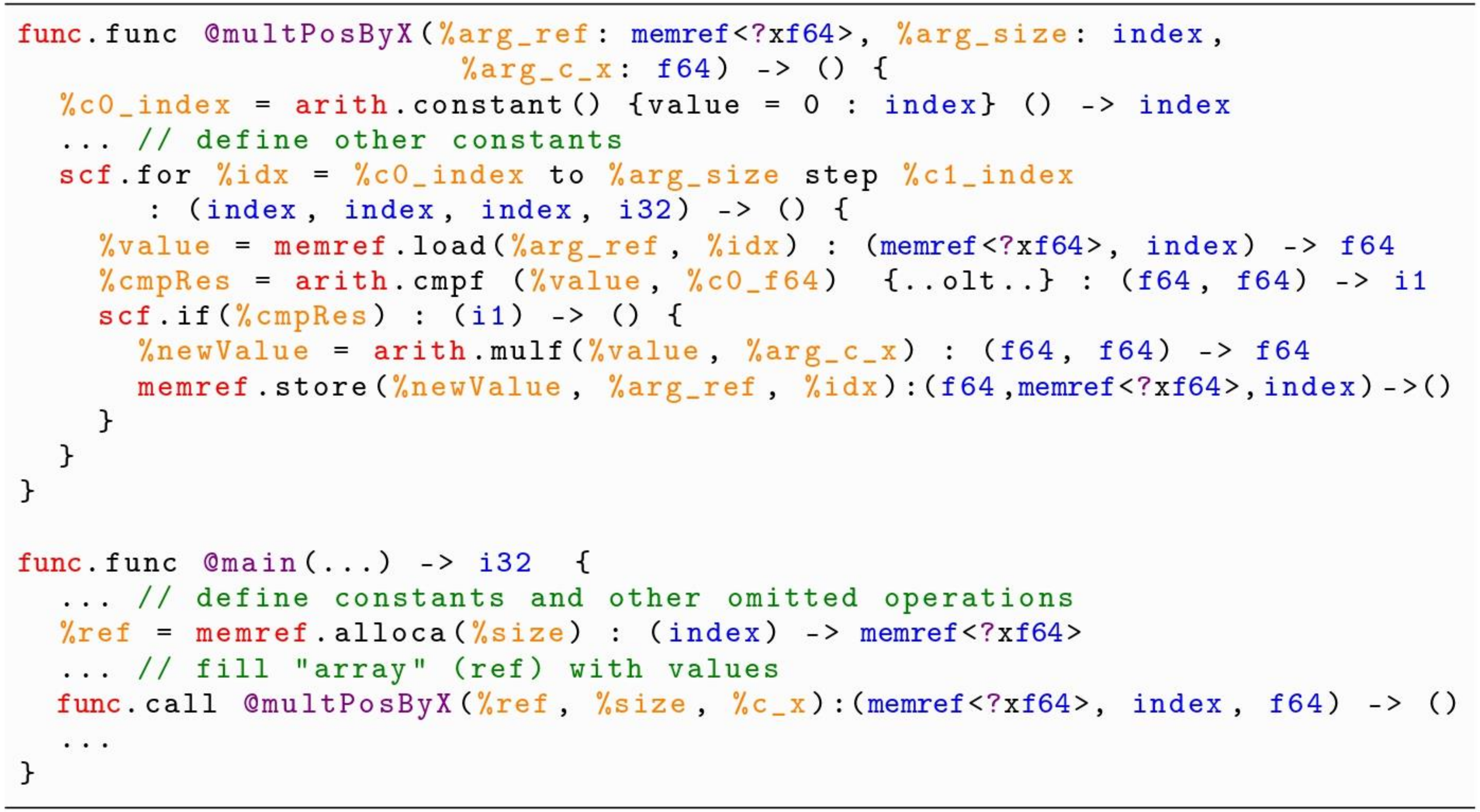
Background: MLIR (2)



```
func.func @multPosByX(%arg_ref: memref<?xf64>, %arg_size: index,
                                %arg_c_x: f64) -> () {
  %c0_index = arith.constant() {value = 0 : index} () -> index
  ... // define other constants
  scf.for %idx = %c0_index to %arg_size step %c1_index
        : (index, index, index, i32) -> () {
    %value = memref.load(%arg_ref, %idx) : (memref<?xf64>, index) -> f64
    %cmpRes = arith.cmpf (%value, %c0_f64)  {..olt..} : (f64, f64) -> i1
    scf.if(%cmpRes) : (i1) -> () {
      %newValue = arith.mulf(%value, %arg_c_x) : (f64, f64) -> f64
      memref.store(%newValue, %arg_ref, %idx):(f64,memref<?xf64>,index)->()
    }
  }
}

func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

- Red highlights the dialects
- Orange the values
- Blue the types

Background: MLIR (2)



```
func.func @multPosByX(%arg_ref: memref<?xf64>, %arg_size: index,
                                %arg_c_x: f64) -> () {
  %c0_index = arith.constant() {value = 0 : index} () -> index
  ... // define other constants
  scf.for %idx = %c0_index to %arg_size step %c1_index
       : (index, index, index, i32) -> () {
    %value = memref.load(%arg_ref, %idx) : (memref<?xf64>, index) -> f64
    %cmpRes = arith.cmpf (%value, %c0_f64)  {..olt..} : (f64, f64) -> i1
    scf.if(%cmpRes) : (i1) -> () {
      %newValue = arith.mulf(%value, %arg_c_x) : (f64, f64) -> f64
      memref.store(%newValue, %arg_ref, %idx):(f64,memref<?xf64>,index)->()
    }
  }
}

func.func @main(...) -> i32  {
  ... // define constants and other omitted operations
  %ref = memref.alloca(%size) : (index) -> memref<?xf64>
  ... // fill "array" (ref) with values
  func.call @multPosByX(%ref, %size, %c_x):(memref<?xf64>, index, f64) -> ()
  ...
}
```

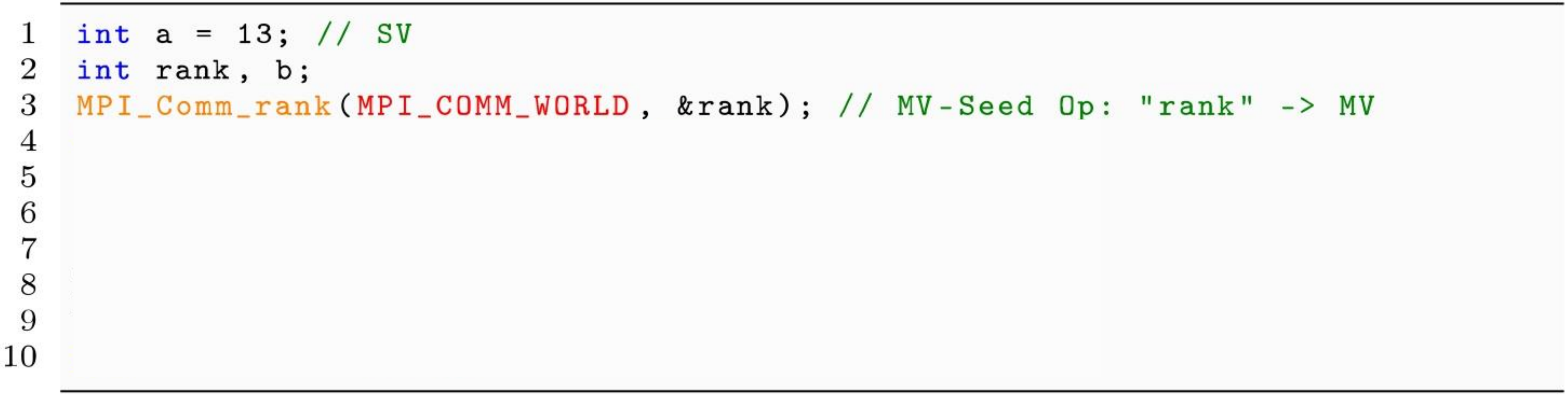
- Red highlights the dialects
- Orange the values
- Blue the types

- As each dialect can seamlessly interact with other dialects
  - MLIR provides a toolset to develop an expressive IR composed of multiple levels and semantics

Background: Multi-Values

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise

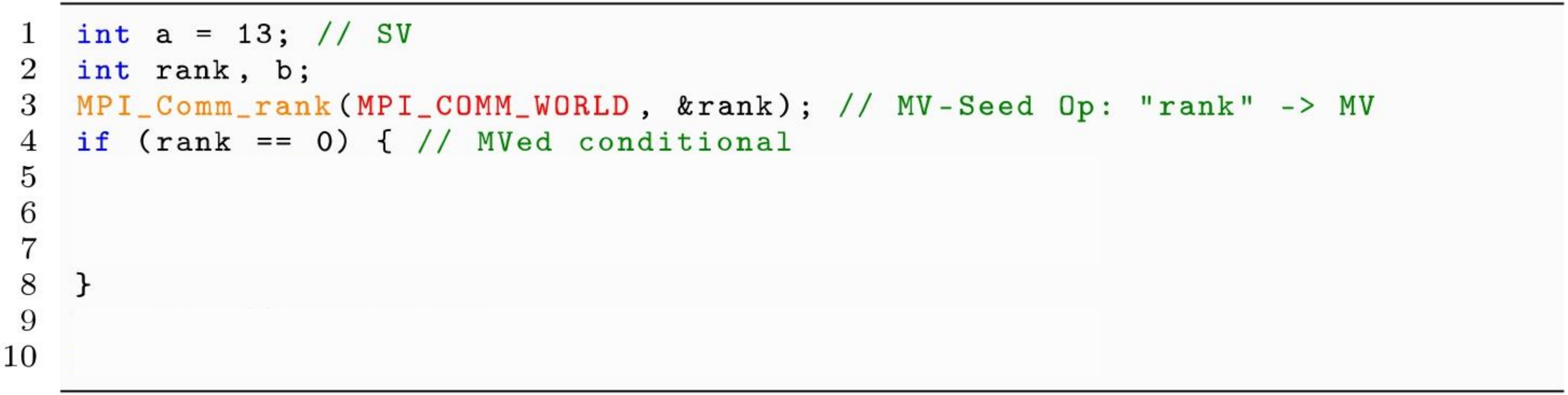
Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise

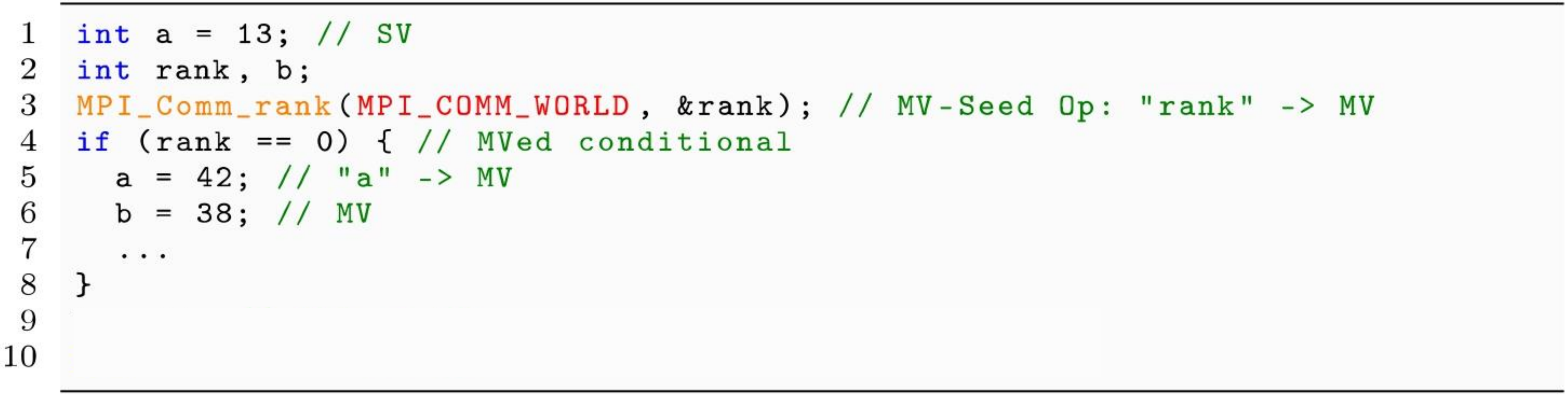
Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
if (rank == 0) { // MVed conditional



}


```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise

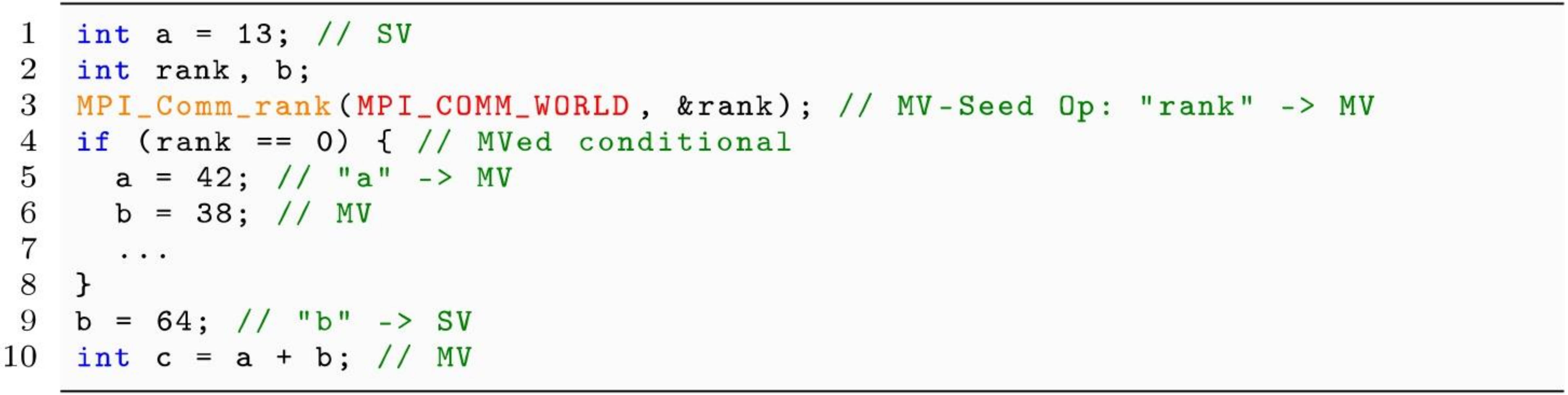
Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
if (rank == 0) { // MVed conditional
  a = 42; // "a" -> MV
  b = 38; // MV
  ...
}
```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise

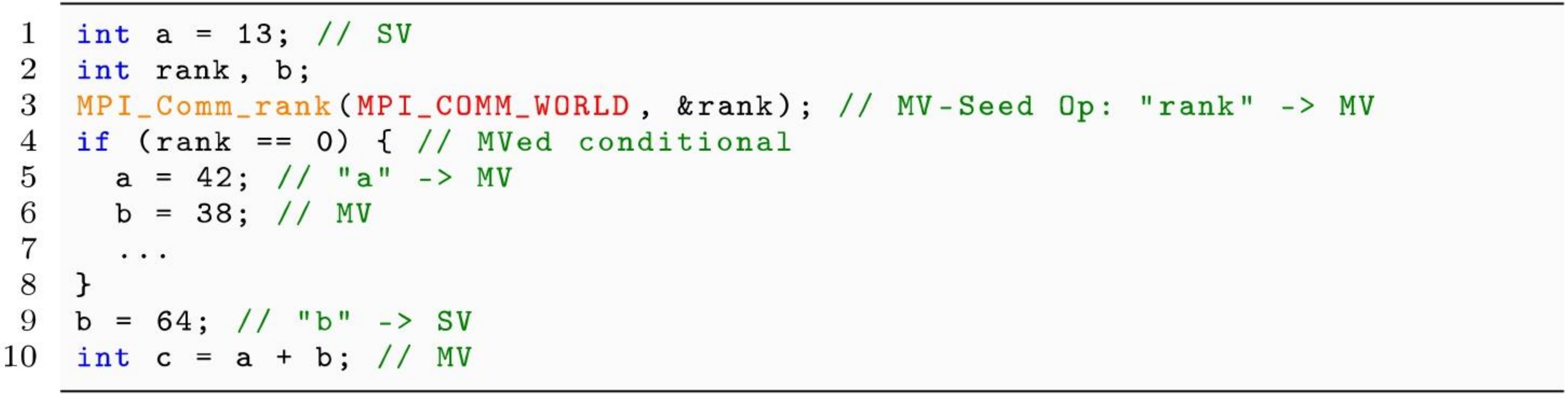
Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
if (rank == 0) { // MVed conditional
  a = 42; // "a" -> MV
  b = 38; // MV
  ...
}
b = 64; // "b" -> SV
int c = a + b; // MV
```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise

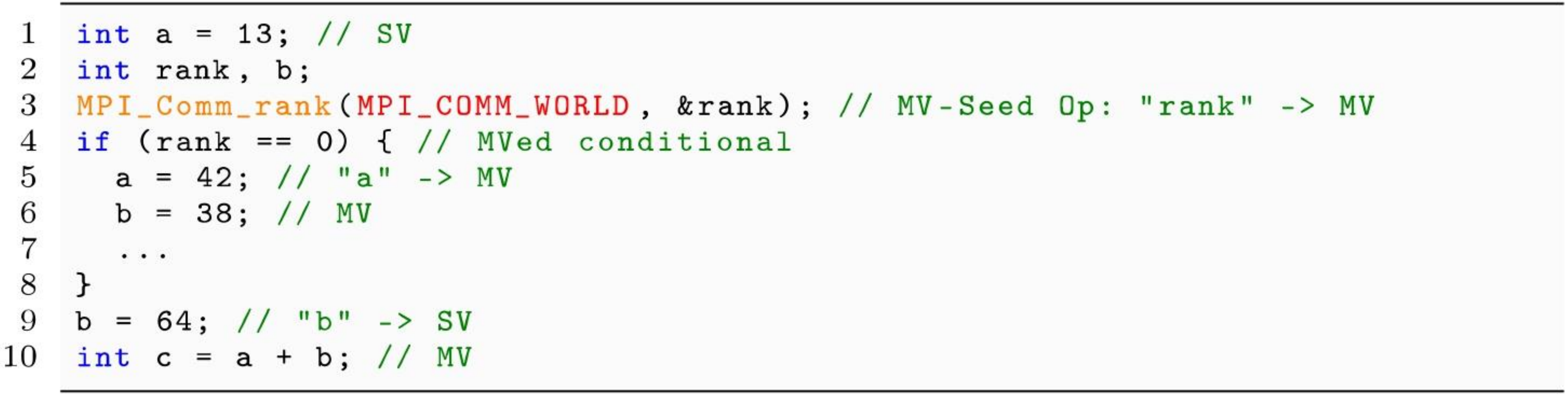
Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
if (rank == 0) { // MVed conditional
  a = 42; // "a" -> MV
  b = 38; // MV
  ...
}
b = 64; // "b" -> SV
int c = a + b; // MV
```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise
- MV-seed operations yield MVs per definition

Background: Multi-Values



```
int a = 13; // SV
int rank, b;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // MV-Seed Op: "rank" -> MV
if (rank == 0) { // MVed conditional
  a = 42; // "a" -> MV
  b = 38; // MV
  ...
}
b = 64; // "b" -> SV
int c = a + b; // MV
```

- A variable or value is called multi-value (MV)
  - if its runtime-value may differ for a subset of processes
  - and called single-value (SV) otherwise
- MV-seed operations yield MVs per definition
- New MVs can result from operations using MVs
  - or through control-flow operations

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO ’24

- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO '24

- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR
- Add execution kind and executing processes analysis

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO ’24

- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR
- Add execution kind and executing processes analysis
- Express the results of this extended MV-analysis in the IR by attributes

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO '24

- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR

- Add execution kind and executing processes analysis

- Express the results of this extended MV-analysis in the IR by attributes
  - isMultiValued: True or False
    - Tagging conditionals as depending on an MV

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO ’24

- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR

- Add execution kind and executing processes analysis

- Express the results of this extended MV-analysis in the IR by attributes
  - isMultiValued: True or False
    - Tagging conditionals as depending on an MV
  - executionKind: All, AllBut, Static, Dynamic, One, AllButOne
    - Specify for each operation the information given at compile time

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO '24

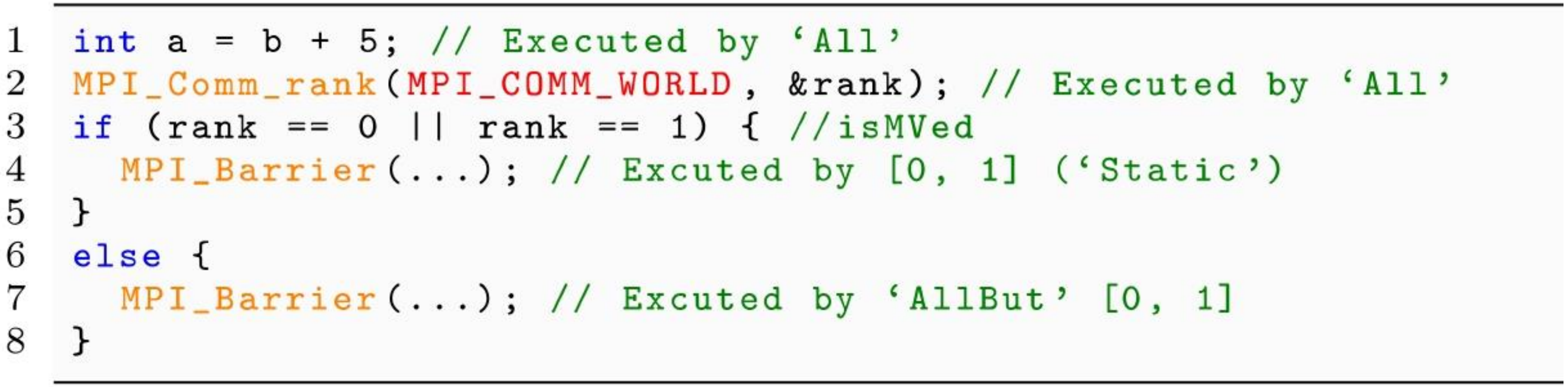
- Adapt existing work[2] for MV-analysis for SPMD programs in MLIR

- Add execution kind and executing processes analysis

- Express the results of this extended MV-analysis in the IR by attributes
  - isMultiValued: True or False
    - Tagging conditionals as depending on an MV
  - executionKind: All, AllBut, Static, Dynamic, One, AllButOne
    - Specify for each operation the information given at compile time
  - executed(Not)By: List of process IDs (not) executing the operation
    - Given for Static and AllBut cases

SPMD IR: Extended Multi-Value Analysis

[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO ’24



```
int a = b + 5; // Executed by ‘All’
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Executed by ‘All’
if (rank == 0 || rank == 1) { //isMVed
  MPI_Barrier(...); // Excuted by [0, 1] (‘Static’)
}
else {
  MPI_Barrier(...); // Excuted by ‘AllBut’ [0, 1]
}
```

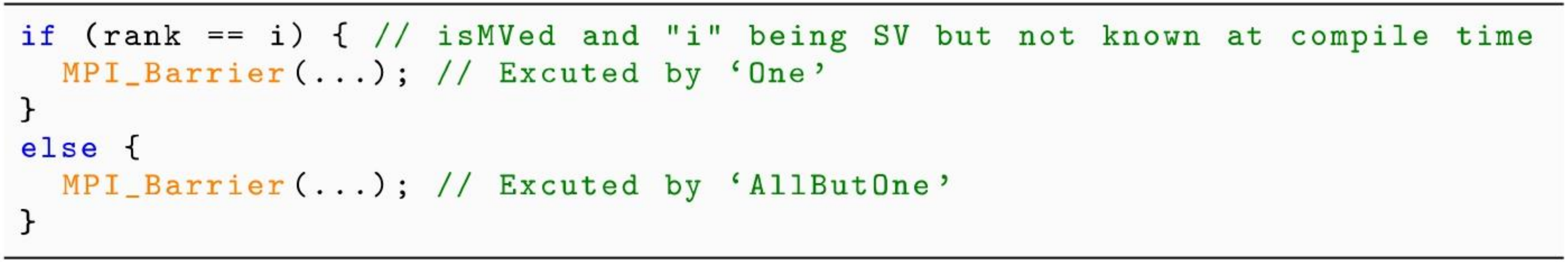
- isMultiValued: True or False
  - Tagging conditionals as depending on an MV

- executionKind: All, AllBut, Static, Dynamic, One, AllButOne
  - Specify for each operation the information given at compile time

- executed(Not)By: List of process IDs (not) executing the operation
  - Given for Static and AllBut cases

SPMD IR: Extended Multi-Value Analysis

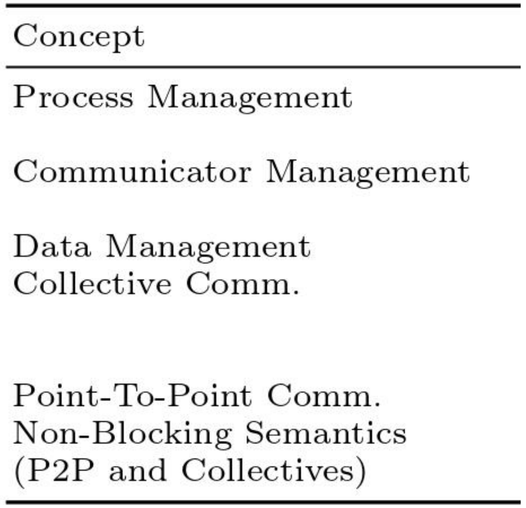
[2] Tiotto et al.: Experiences Building an MLIR-Based SYCL Compiler. CGO ’24



```
if (rank == i) { // isMVed and "i" being SV but not known at compile time
  MPI_Barrier(...); // Excuted by ‘One’
}
else {
  MPI_Barrier(...); // Excuted by ‘AllButOne’
}
```

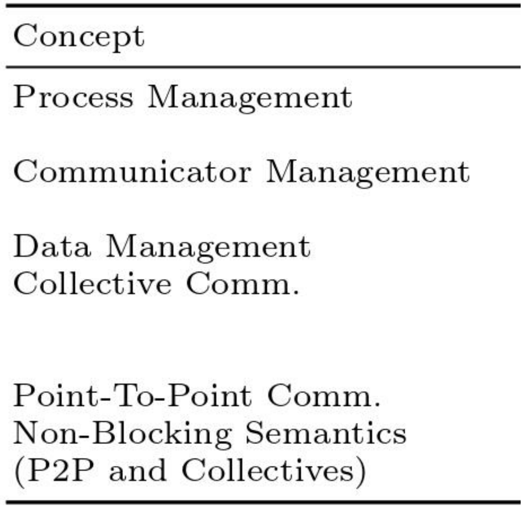
- isMultiValued: True or False
  - Tagging conditionals as depending on an MV
- executionKind: All, AllBut, Static, Dynamic, One, AllButOne
  - Specify for each operation the information given at compile time
- executed(Not)By: List of process IDs (not) executing the operation
  - Given for Static and AllBut cases

SPMD IR: Scope



Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics (P2P and Collectives)

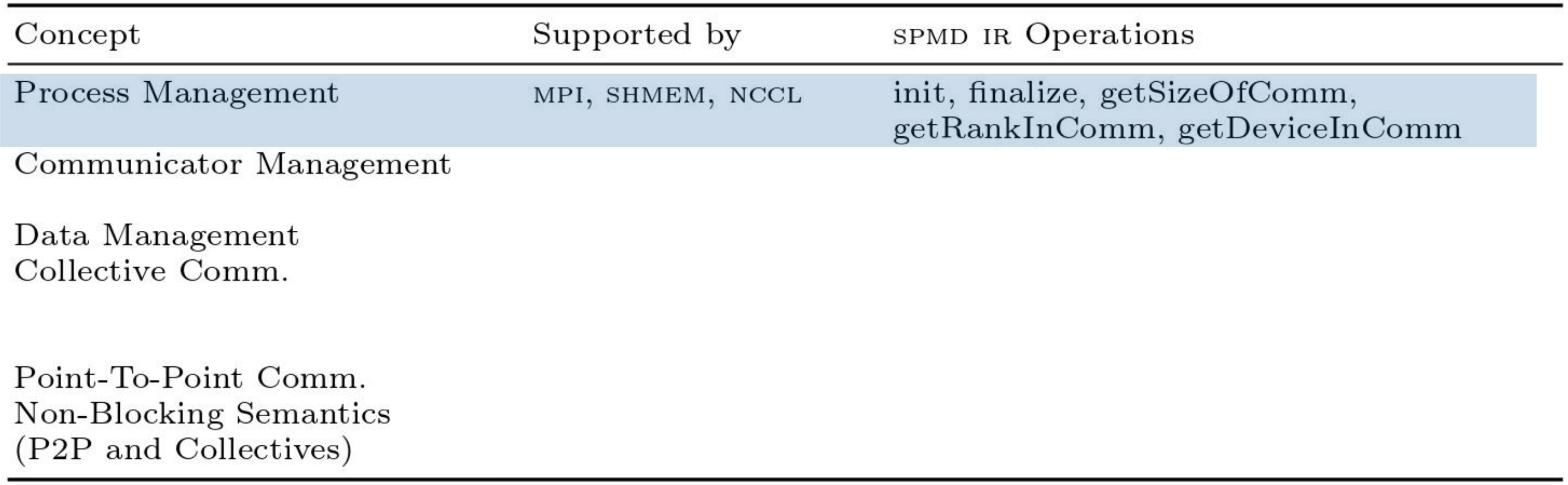
SPMD IR: Scope



Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics (P2P and Collectives)

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

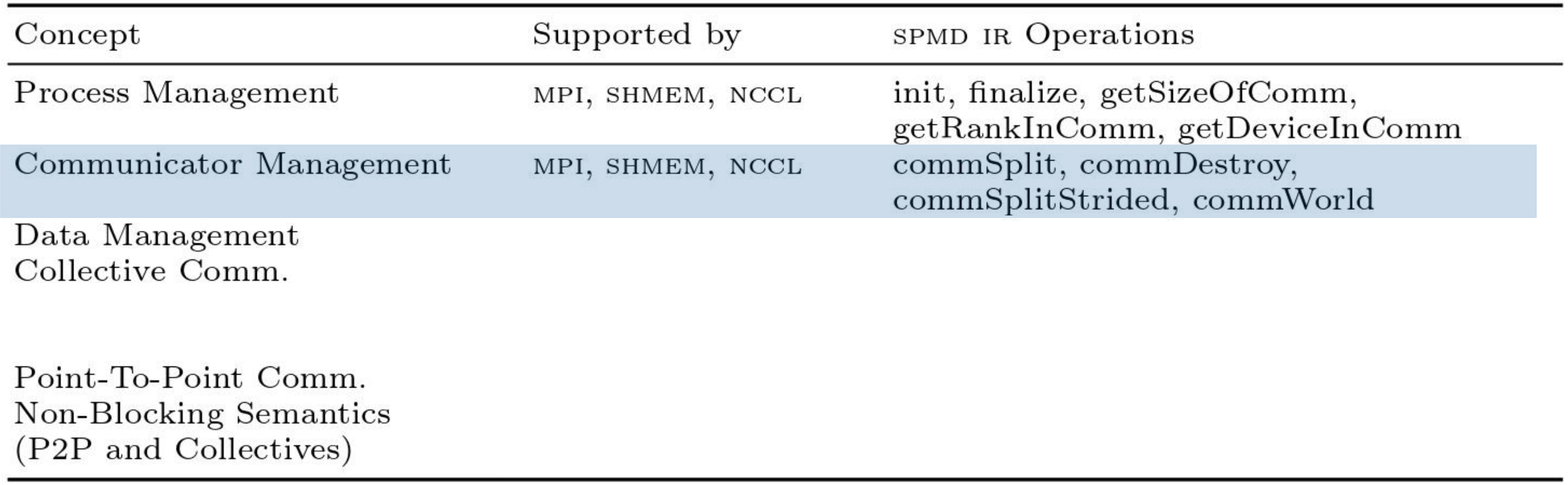
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management		
Data Management		
Collective Comm.		
Point-To-Point Comm.		
Non-Blocking Semantics (P2P and Collectives)		

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

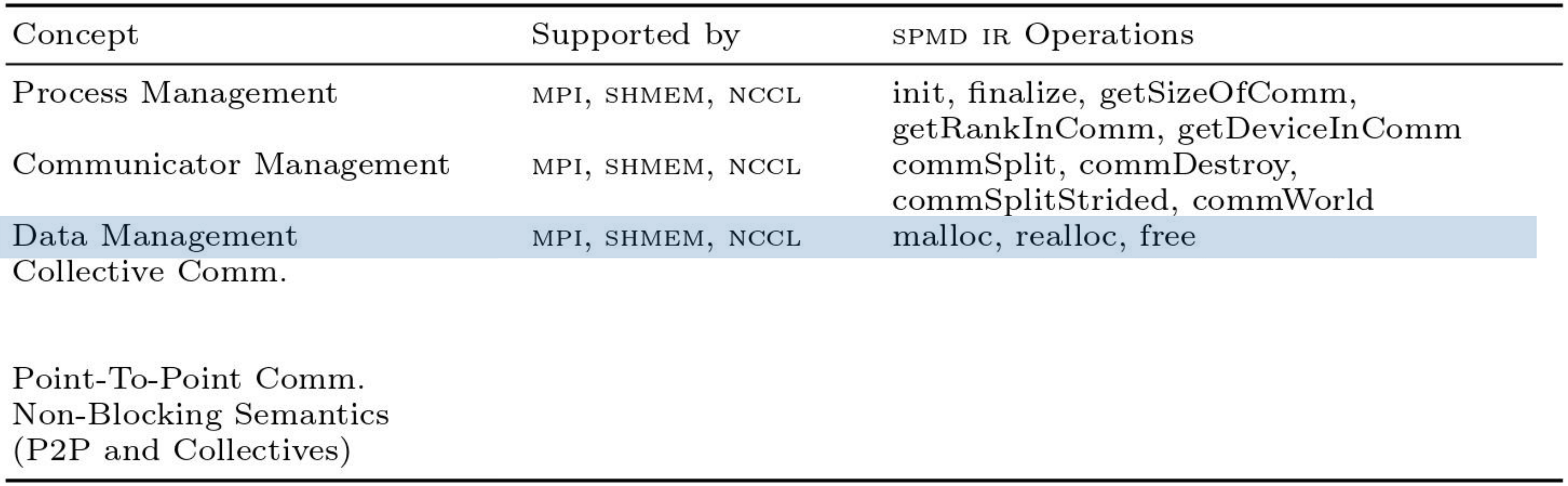
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management		
Collective Comm.		
Point-To-Point Comm.		
Non-Blocking Semantics (P2P and Collectives)		

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

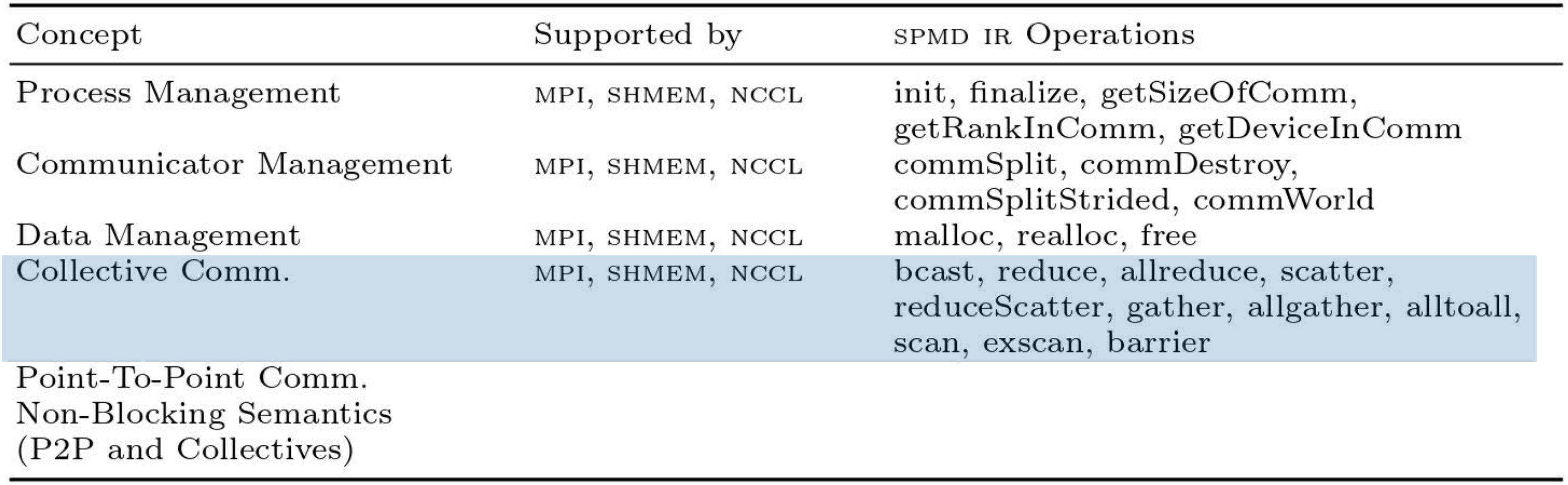
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.		
Point-To-Point Comm.		
Non-Blocking Semantics (P2P and Collectives)		

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

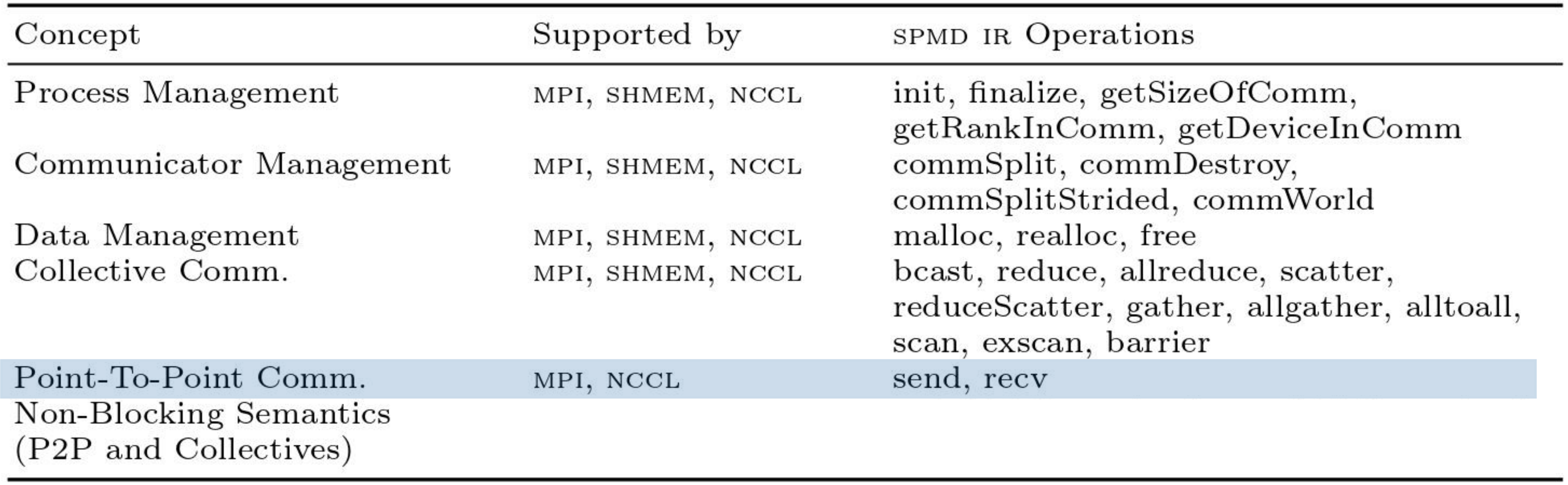
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.		
Non-Blocking Semantics (P2P and Collectives)		

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

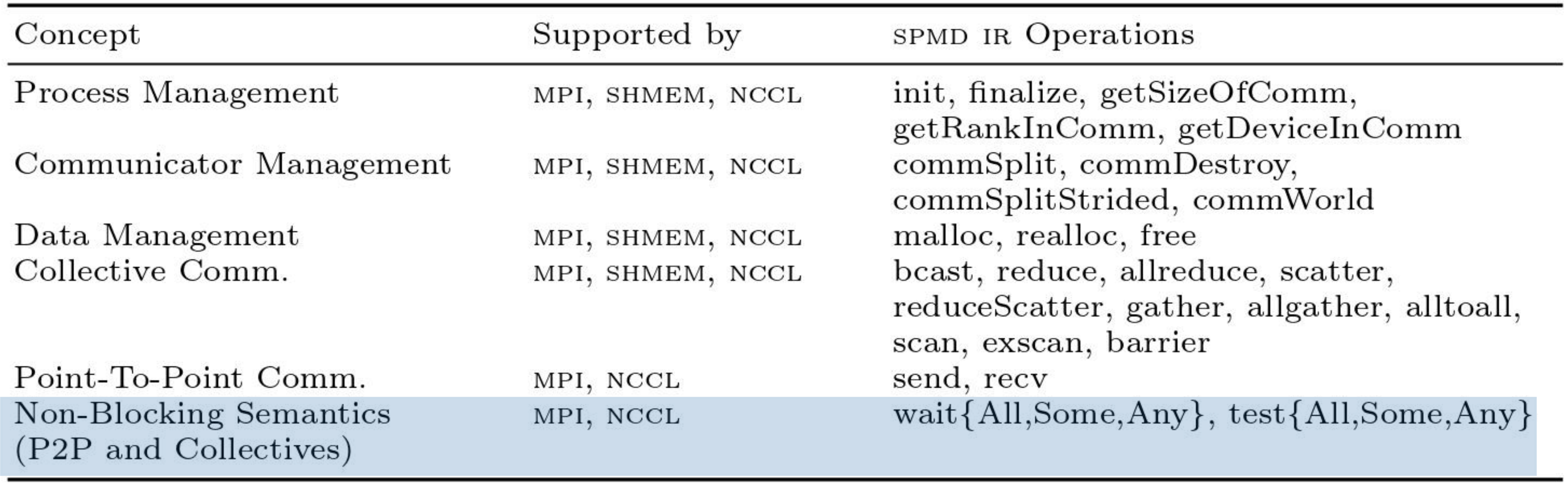
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)		

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

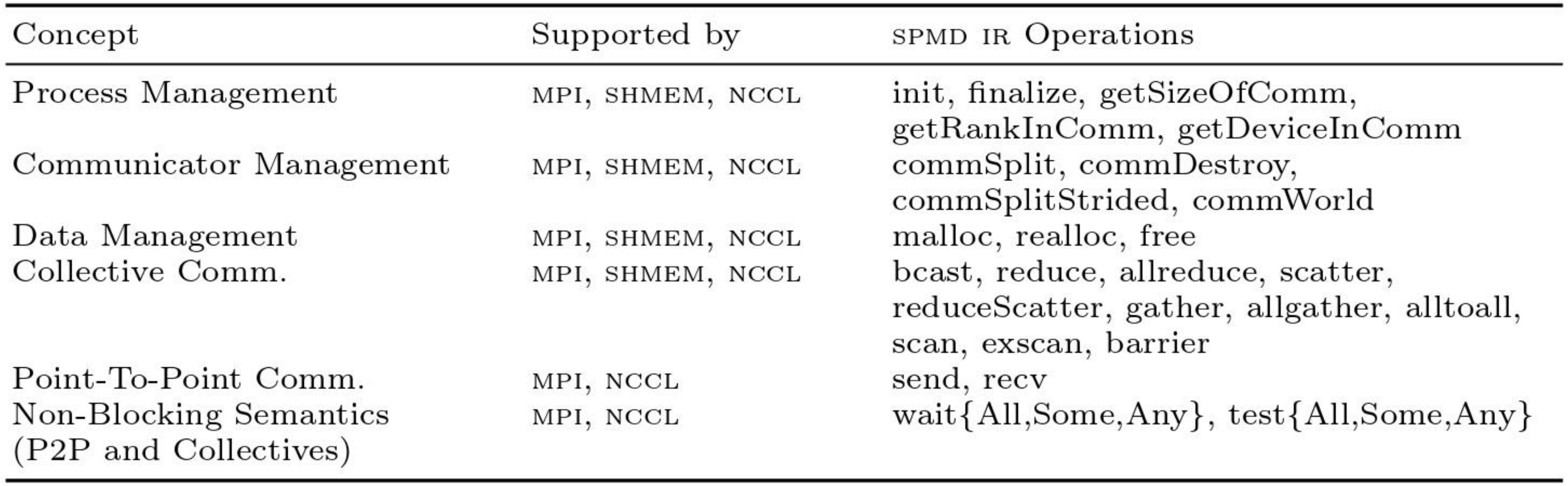
SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

SPMD IR: Scope

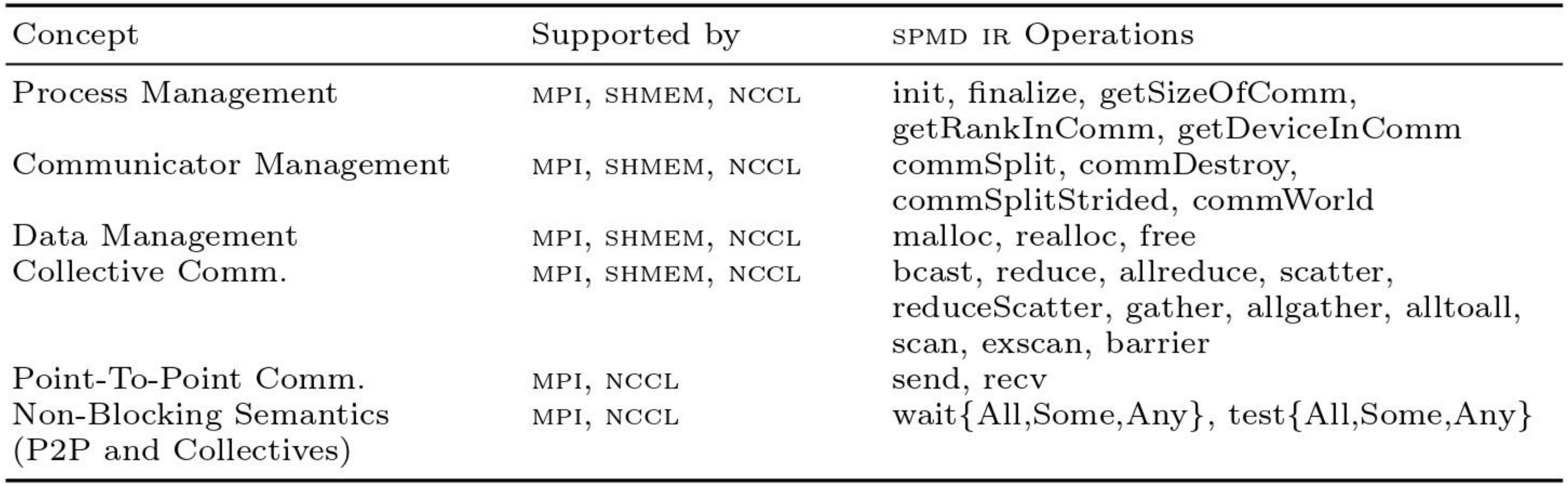


Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

SPMD IR: Scope



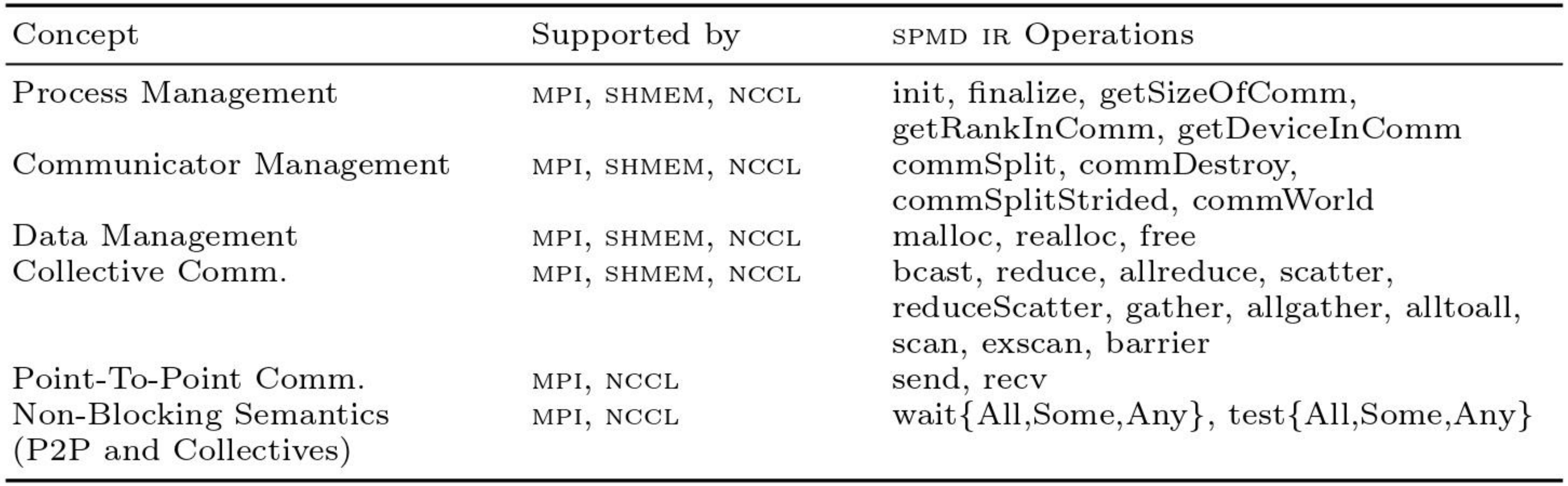
Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags

SPMD IR: Scope



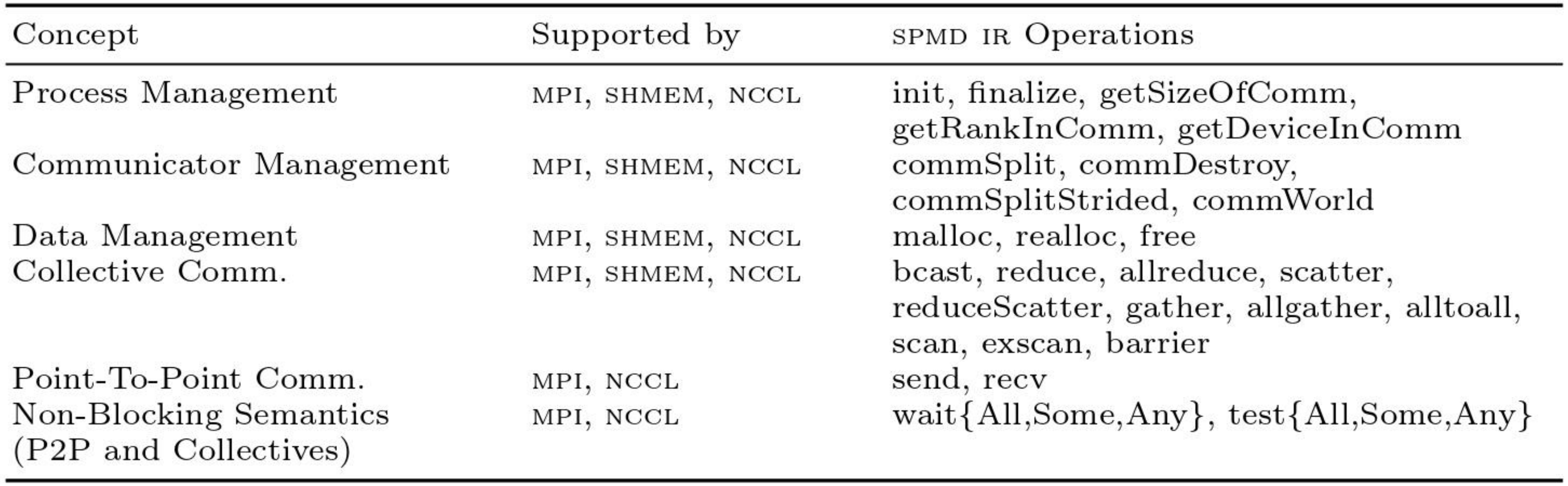
Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data

SPMD IR: Scope



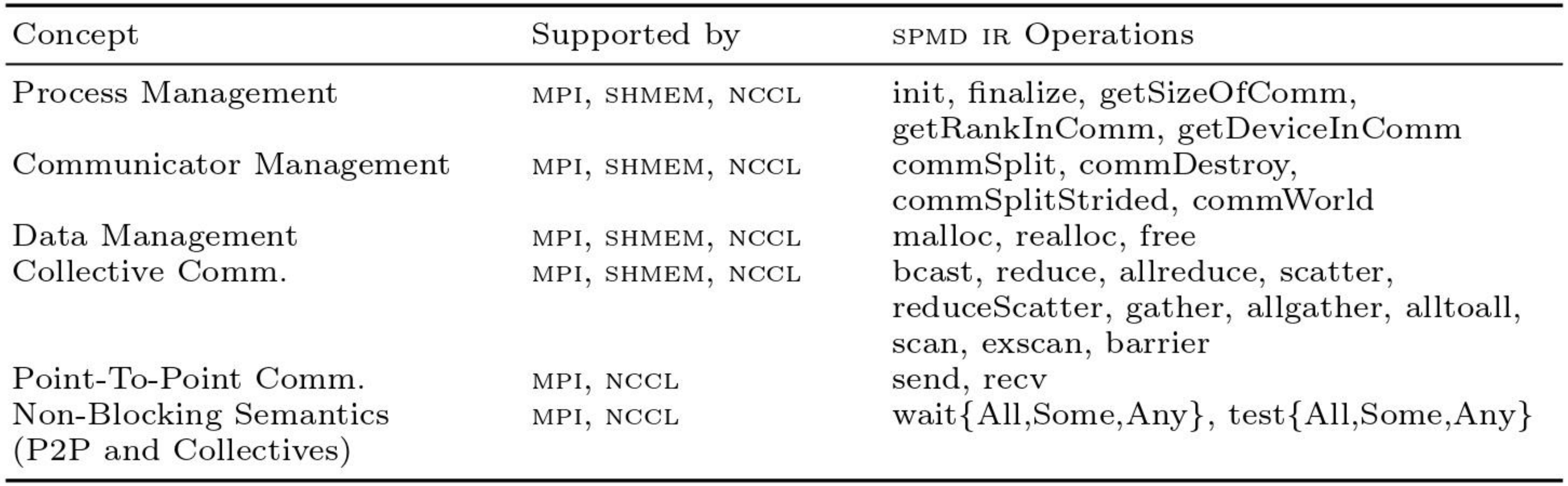
Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data
  - custom dialect (spmd) types for the remaining, e.g.:

SPMD IR: Scope



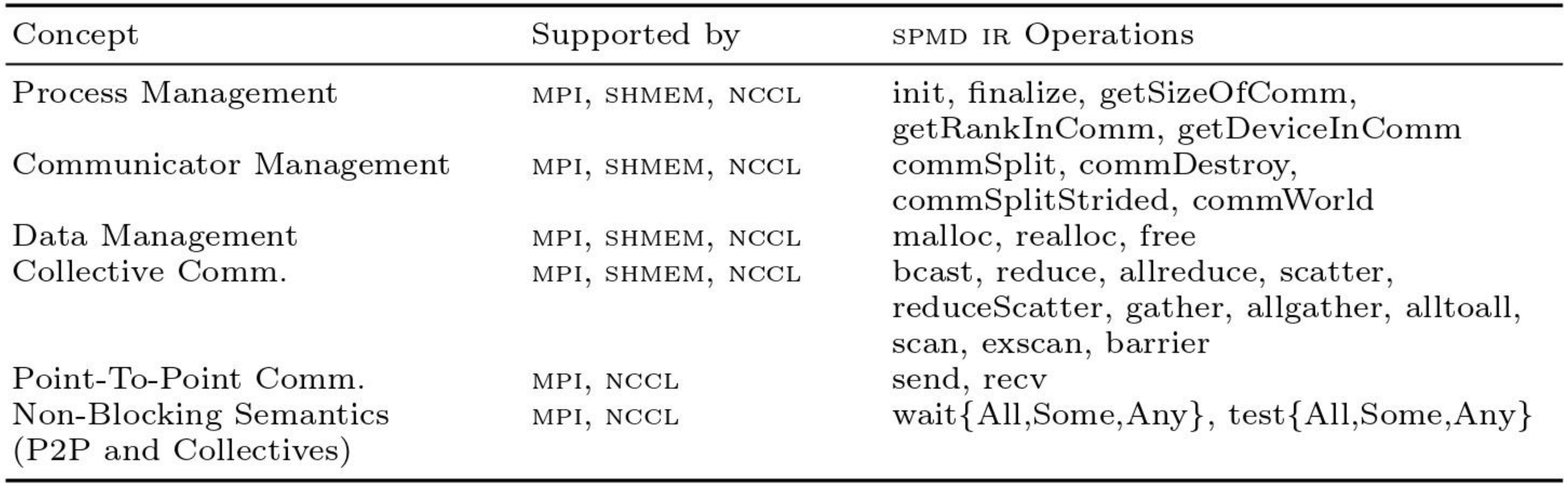
Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data
  - custom dialect (spmd) types for the remaining, e.g.:
    - a communicator of type spmd comm

SPMD IR: Scope



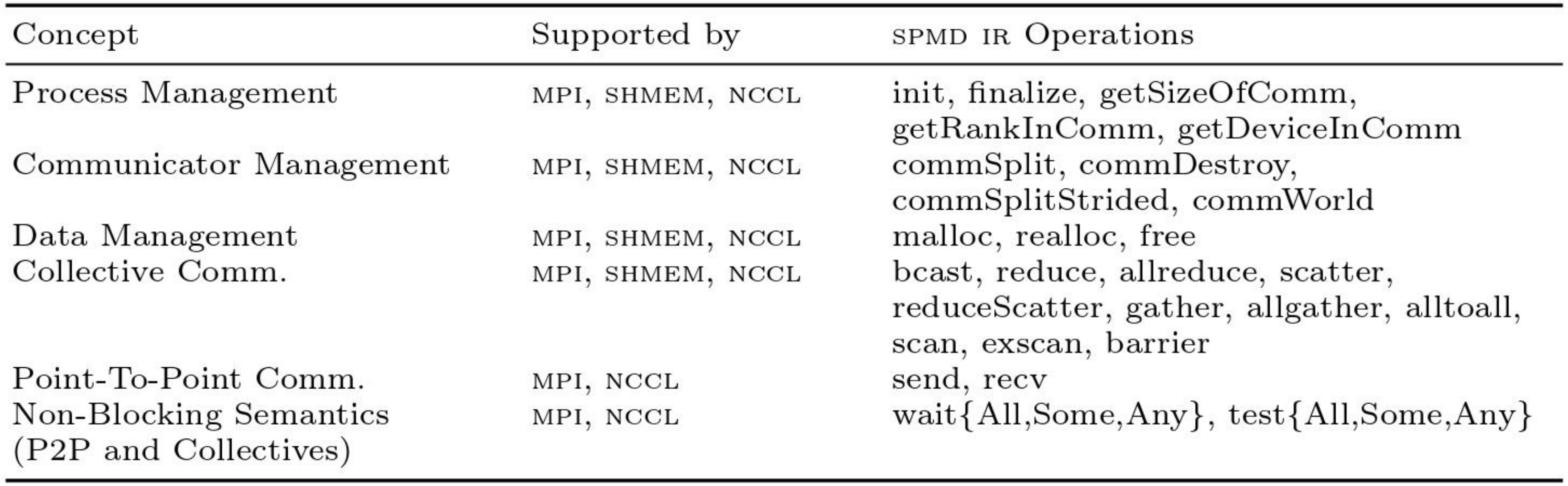
Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data
  - custom dialect (spmd) types for the remaining, e.g.:
    - a communicator of type spmd comm
    - an error/success value of type spmd error

SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

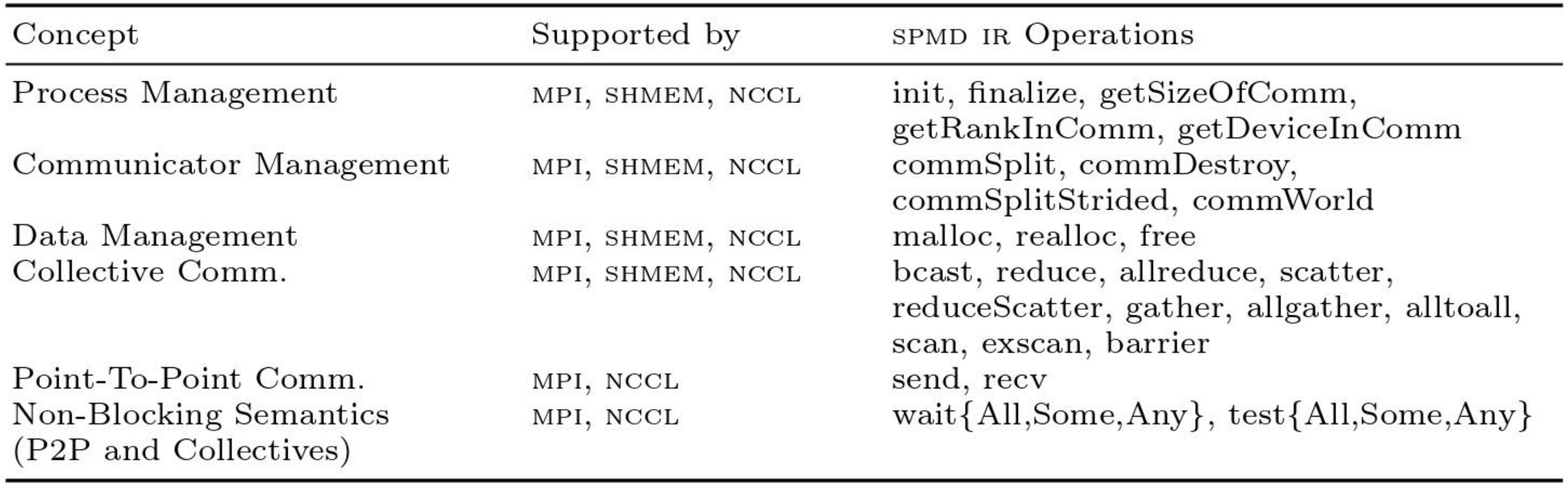
- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data
  - custom dialect (spmd) types for the remaining, e.g.:
    - a communicator of type spmd comm
    - an error/success value of type spmd error

- For memory references, IR specifies side-effects, e.g., read-only

SPMD IR: Scope



Concept	Supported by	SPMD IR Operations
Process Management	MPI, SHMEM, NCCL	init, finalize, getSizeOfComm, getRankInComm, getDeviceInComm
Communicator Management	MPI, SHMEM, NCCL	commSplit, commDestroy, commSplitStrided, commWorld
Data Management	MPI, SHMEM, NCCL	malloc, realloc, free
Collective Comm.	MPI, SHMEM, NCCL	bcast, reduce, allreduce, scatter, reduceScatter, gather, allgather, alltoall, scan, exscan, barrier
Point-To-Point Comm.	MPI, NCCL	send, recv
Non-Blocking Semantics (P2P and Collectives)	MPI, NCCL	wait{All,Some,Any}, test{All,Some,Any}

- Idea:
  - Having a minimal set of operations, types, and attributes that can cover those features for unification purposes

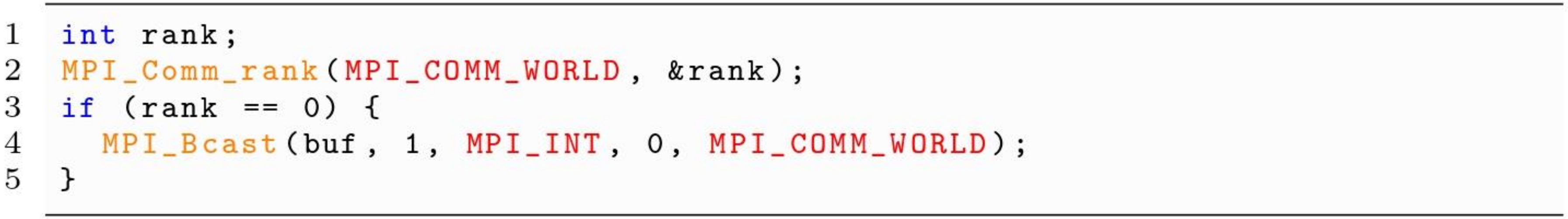
- Attributes for blocking or buffered communication

- Types:
  - integers for process and device IDs, or tags
  - any memref type for communicated data
  - custom dialect (spmd) types for the remaining, e.g.:
    - a communicator of type spmd comm
    - an error/success value of type spmd error

- For memory references, IR specifies side-effects, e.g., read-only

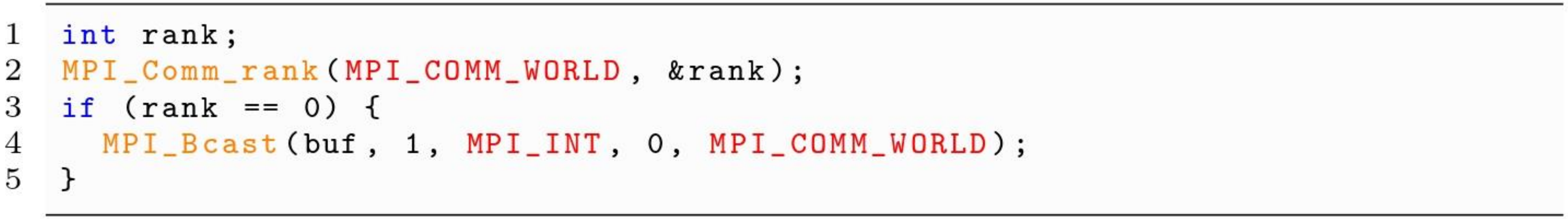
- RMA is work in progress

SPMD IR: Example

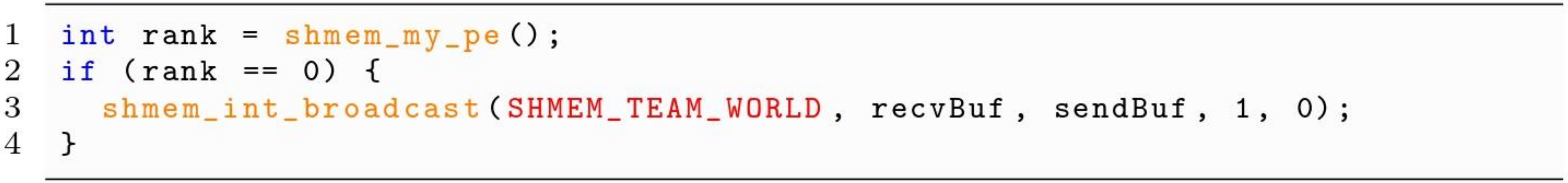


```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

SPMD IR: Example



```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```



```
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```

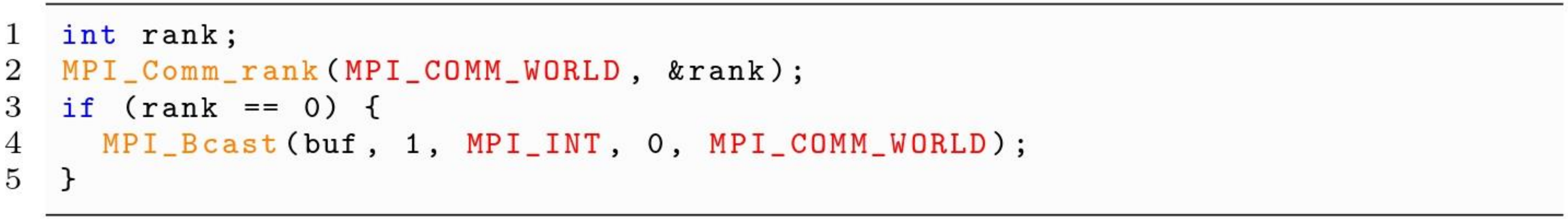
SPMD IR: Example

```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

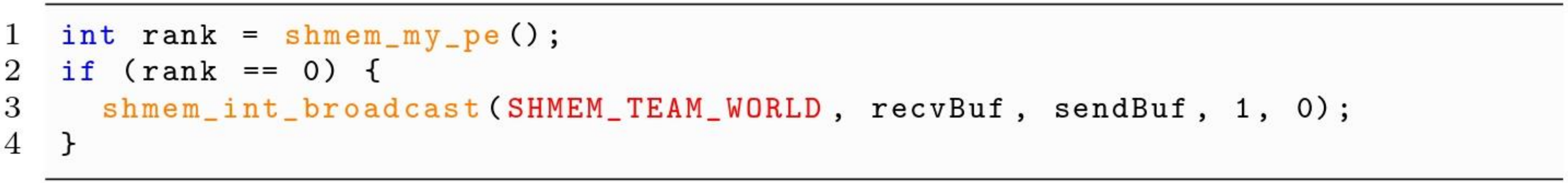
```c
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```

```
%rank, %error1 = spmd.getRankInComm(%comm)

%cmpRes = arith.cmpi eq (%rank, %c0)
scf.if (%cmpRes) {
  %error2 = spmd.bcast (%comm, %sendBuf, %recvBuf, %count, %i32Type, %c0)


}
```

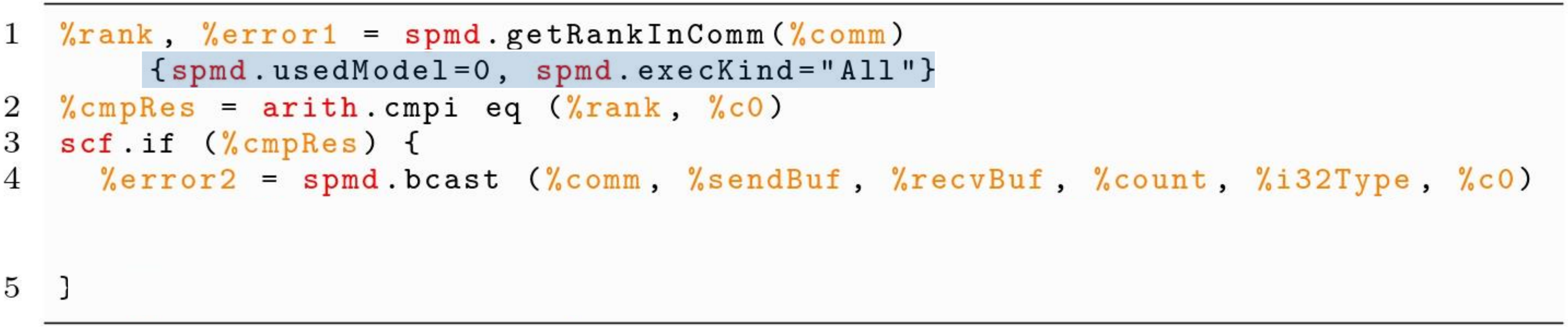
SPMD IR: Example



```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```



```
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```



```
%rank, %error1 = spmd.getRankInComm(%comm)
      {spmd.usedModel=0, spmd.execKind="All"}
%cmpRes = arith.cmpi eq (%rank, %c0)
scf.if (%cmpRes) {
  %error2 = spmd.bcast (%comm, %sendBuf, %recvBuf, %count, %i32Type, %c0)


}
```

SPMD IR: Example

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

```
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```

```
%rank, %error1 = spmd.getRankInComm(%comm)
     {spmd.usedModel=0, spmd.execKind="All"}
%cmpRes = arith.cmpi eq (%rank, %c0) {spmd.execKind="All"}
scf.if (%cmpRes) {
  %error2 = spmd.bcast (%comm, %sendBuf, %recvBuf, %count, %i32Type, %c0)



}
```

SPMD IR: Example

```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

```c
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```

```
%rank, %error1 = spmd.getRankInComm(%comm)
      {spmd.usedModel=0, spmd.execKind="All"}
%cmpRes = arith.cmpi eq (%rank, %c0) {spmd.execKind="All"}
scf.if (%cmpRes) {
  %error2 = spmd.bcast (%comm, %sendBuf, %recvBuf, %count, %i32Type, %c0)
        {spmd.usedModel=0, spmd.isBlocking=true, spmd.executedBy=[0],
        spmd.execKind="Static"}
}
```

SPMD IR: Example

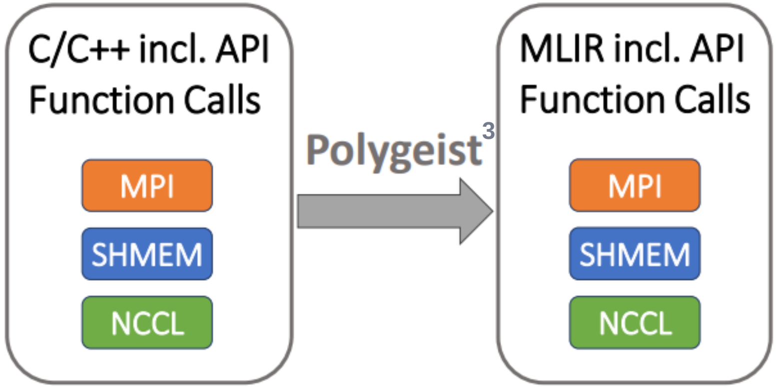
```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Bcast(buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

```c
int rank = shmem_my_pe();
if (rank == 0) {
  shmem_int_broadcast(SHMEM_TEAM_WORLD, recvBuf, sendBuf, 1, 0);
}
```

```
%rank, %error1 = spmd.getRankInComm(%comm)
     {spmd.usedModel=0, spmd.execKind="All"}
%cmpRes = arith.cmpi eq (%rank, %c0) {spmd.execKind="All"}
scf.if (%cmpRes) {
  %error2 = spmd.bcast (%comm, %sendBuf, %recvBuf, %count, %i32Type, %c0)
       {spmd.usedModel=0, spmd.isBlocking=true, spmd.executedBy=[0],
       spmd.execKind="Static"}
} {spmd.execKind="All", spmd.isMV=true}
```

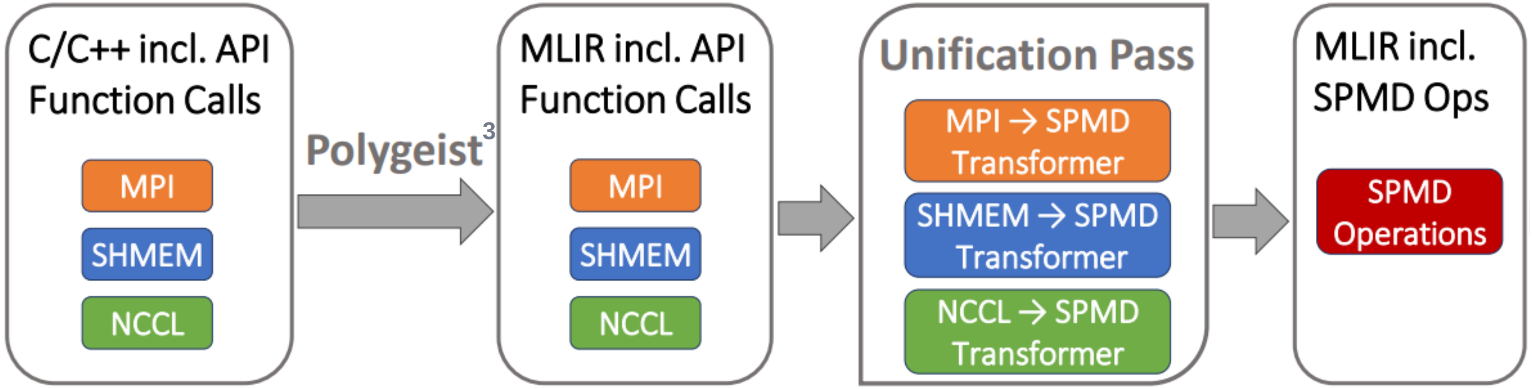
SPMD IR: Workflow

[3] Moses et al.: Polygeist: Raising C to Polyhedral MLIR. PACT ’21



SPMD IR: Workflow

[3] Moses et al.: Polygeist: Raising C to Polyhedral MLIR. PACT '21



C/C++ incl. API Function Calls
- MPI
- SHMEM
- NCCL

Polygeist[3]

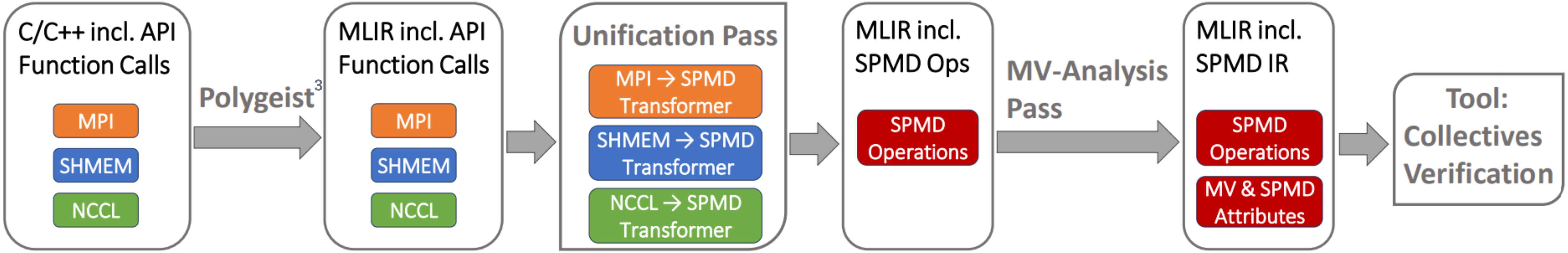
MLIR incl. API Function Calls
- MPI
- SHMEM
- NCCL

Unification Pass
- MPI → SPMD Transformer
- SHMEM → SPMD Transformer
- NCCL → SPMD Transformer

MLIR incl. SPMD Ops
- SPMD Operations

SPMD IR: Workflow

[3] Moses et al.: Polygeist: Raising C to Polyhedral MLIR. PACT ’21



C/C++ incl. API Function Calls
- MPI
- SHMEM
- NCCL

Polygeist[3]

MLIR incl. API Function Calls
- MPI
- SHMEM
- NCCL

Unification Pass
- MPI → SPMD Transformer
- SHMEM → SPMD Transformer
- NCCL → SPMD Transformer

MLIR incl. SPMD Ops
- SPMD Operations

MV-Analysis Pass

MLIR incl. SPMD IR
- SPMD Operations
- MV & SPMD Attributes

Tool: Collectives Verification

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

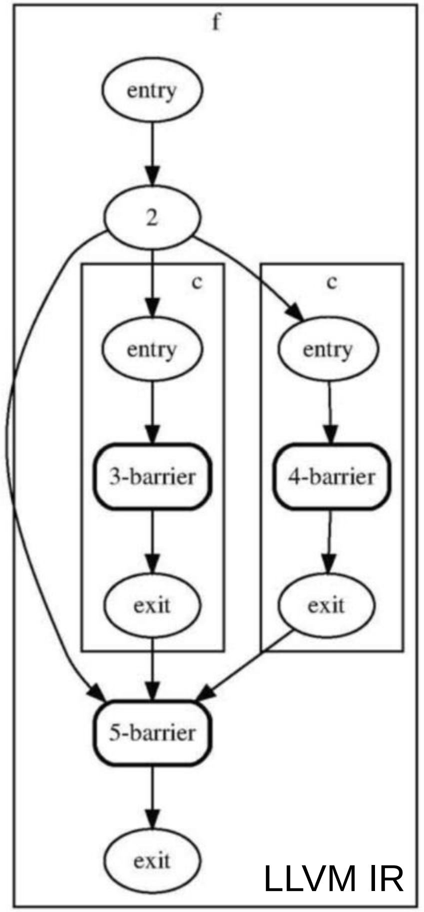
Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

Evaluation

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

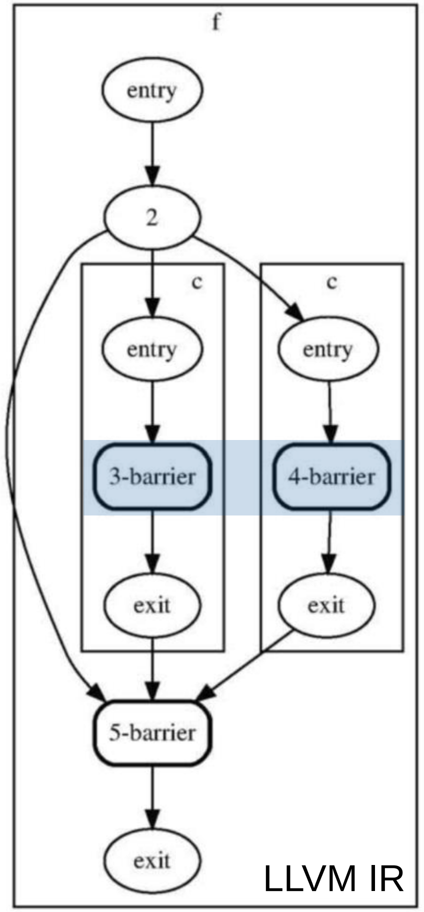


[5]

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness '18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

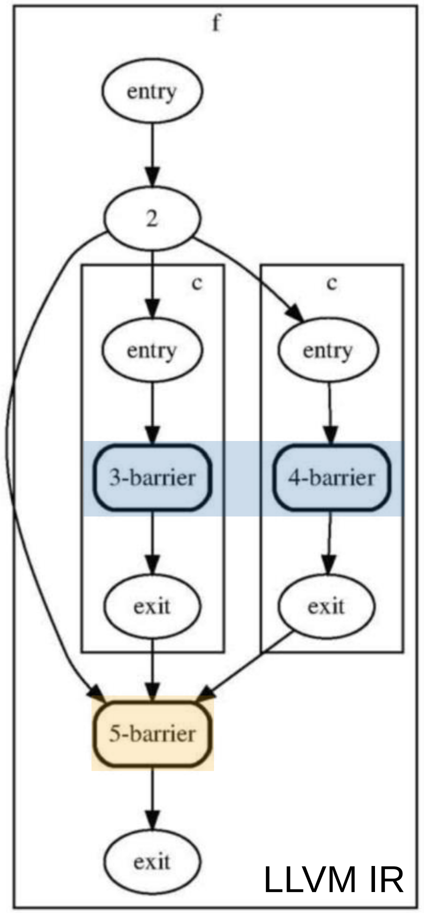


[5]

Evaluation

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

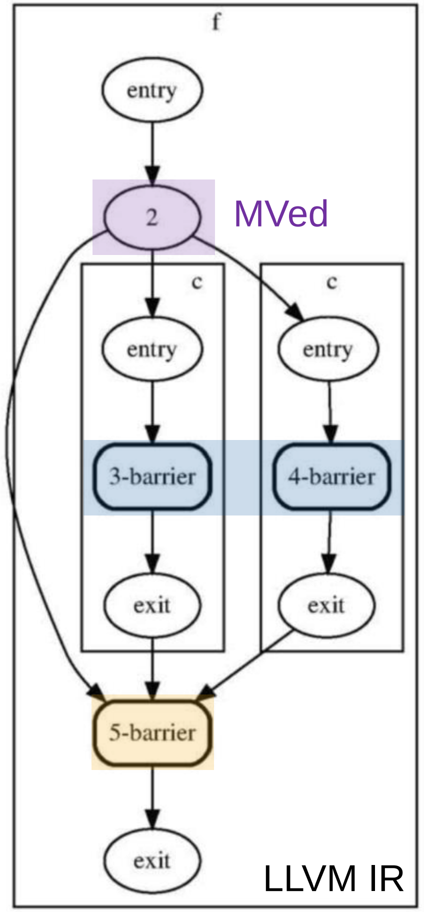


[5]

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness '18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

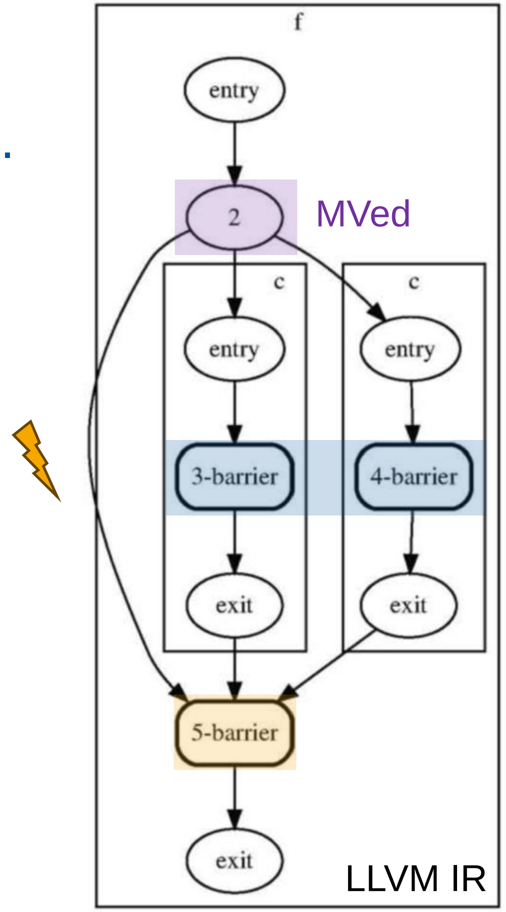


[5]

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

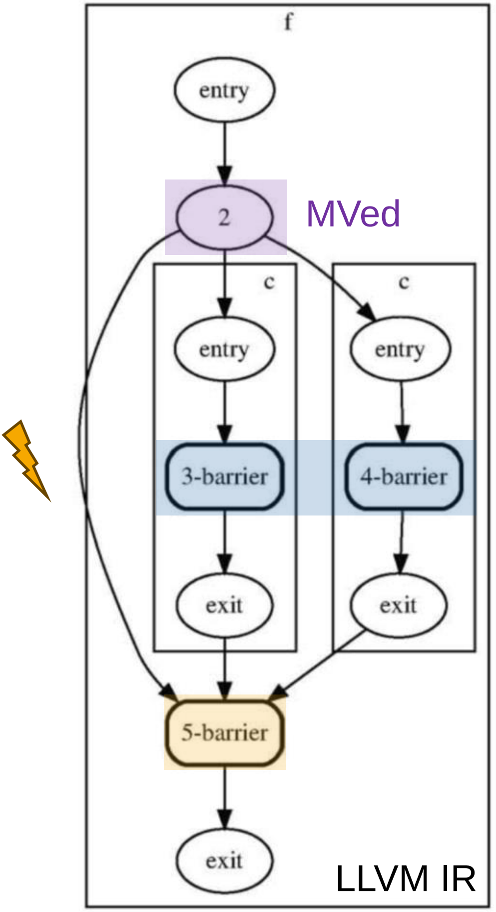


[5]

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.
- Port the approach of PARCOACH[4] to the SPMD IR with two extensions

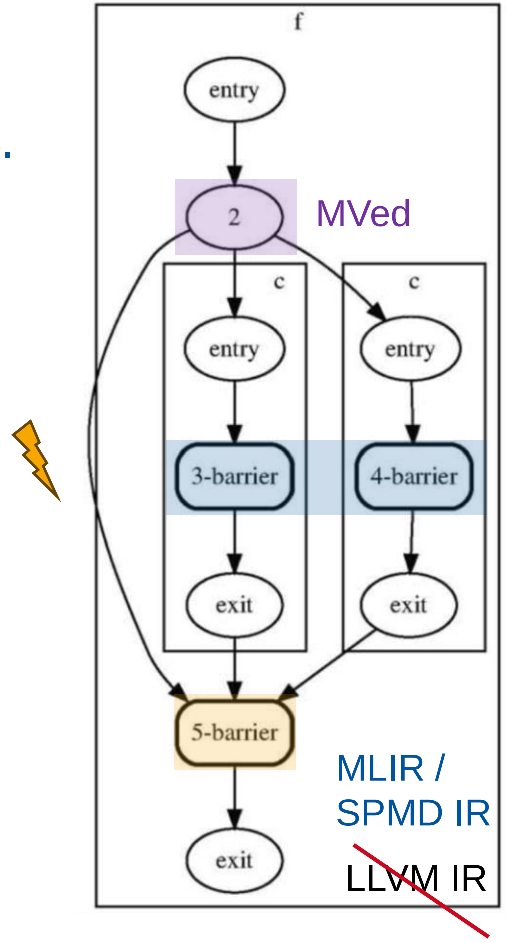


[5]

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness '18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.
- Port the approach of PARCOACH[4] to the SPMD IR with two extensions



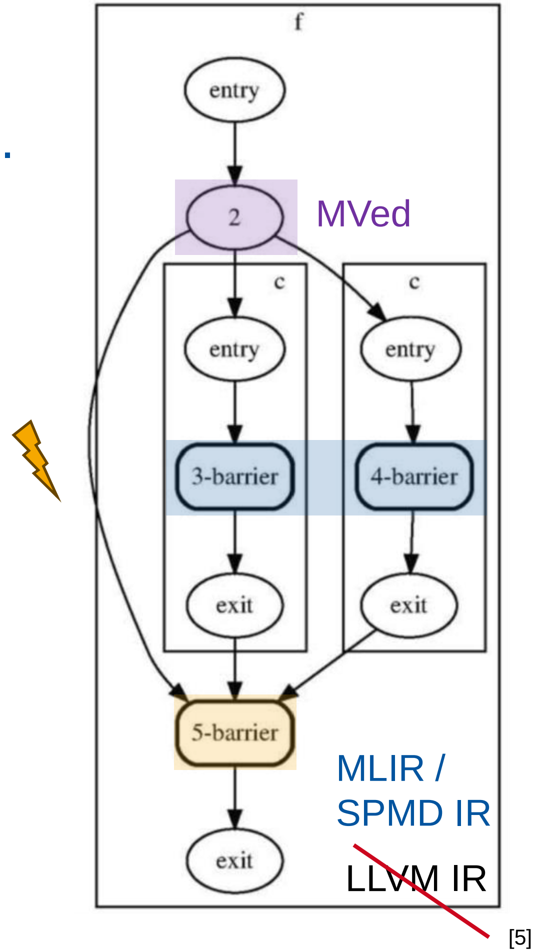
[5]

Evaluation

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

- Port the approach of PARCOACH[4] to the SPMD IR with two extensions
  - Considering execution counts of collectives induced by loops

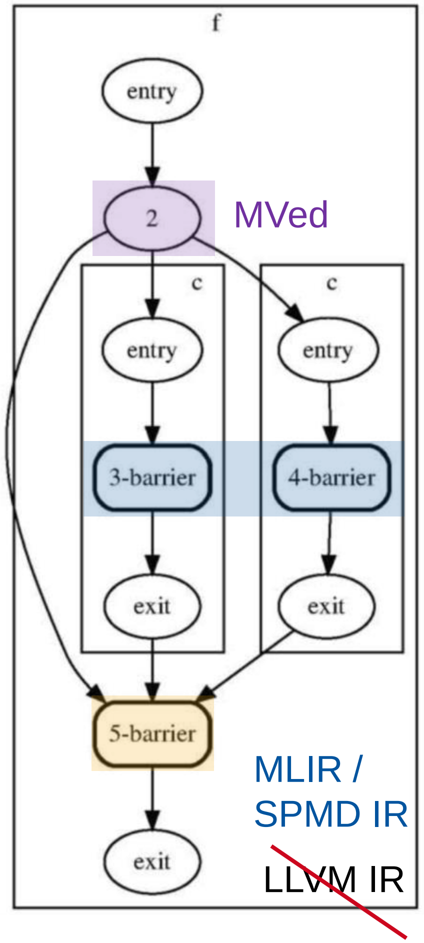


[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness ‘18

Evaluation

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

- Port the approach of PARCOACH[4] to the SPMD IR with two extensions
  - Considering execution counts of collectives induced by loops
  - Static cases where the total number and executing processes of operations are known



MVed

MLIR /
SPMD IR

~~LLVM IR~~

[5]

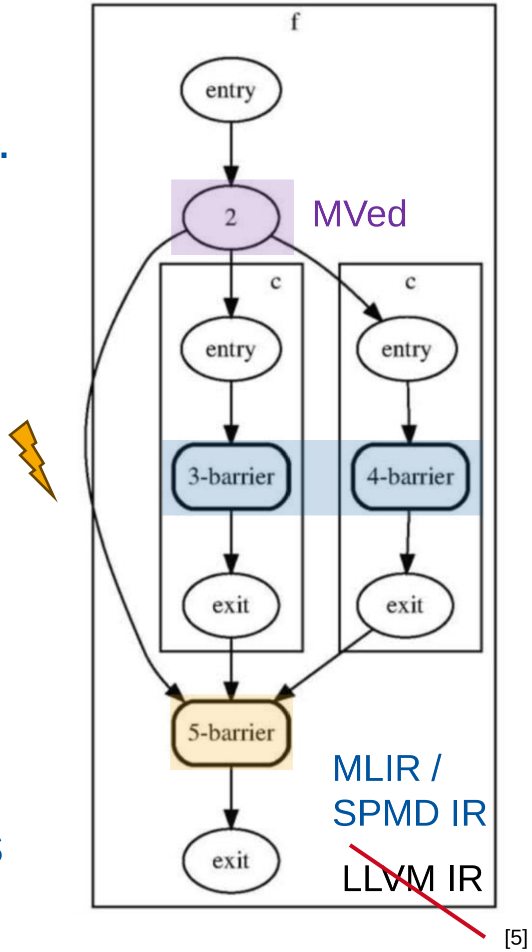
Evaluation

[4] Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19
[5] Huchant et al: PARCOACH Extension for a Full-Interprocedural Collectives Verification. Correctness '18

- Use case: static verification of collective communiation
  - The same kind of collectives has to be called in order by all processes of the same comm.

- Port the approach of PARCOACH[4] to the SPMD IR with two extensions
  - Considering execution counts of collectives induced by loops
  - Static cases where the total number and executing processes of operations are known

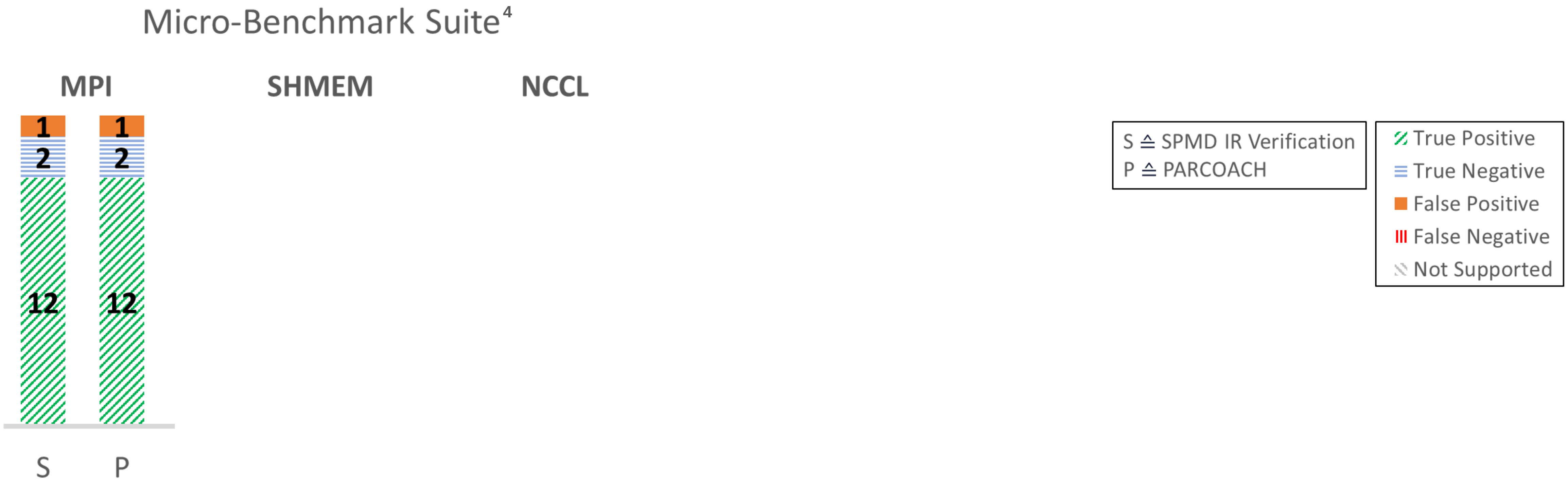
- Assessment based on PARCOACH‘s micro-benchmark suite and custom test cases
  - Port original MPI codes to SHMEM and NCCL and provide hybrid cases



[5]

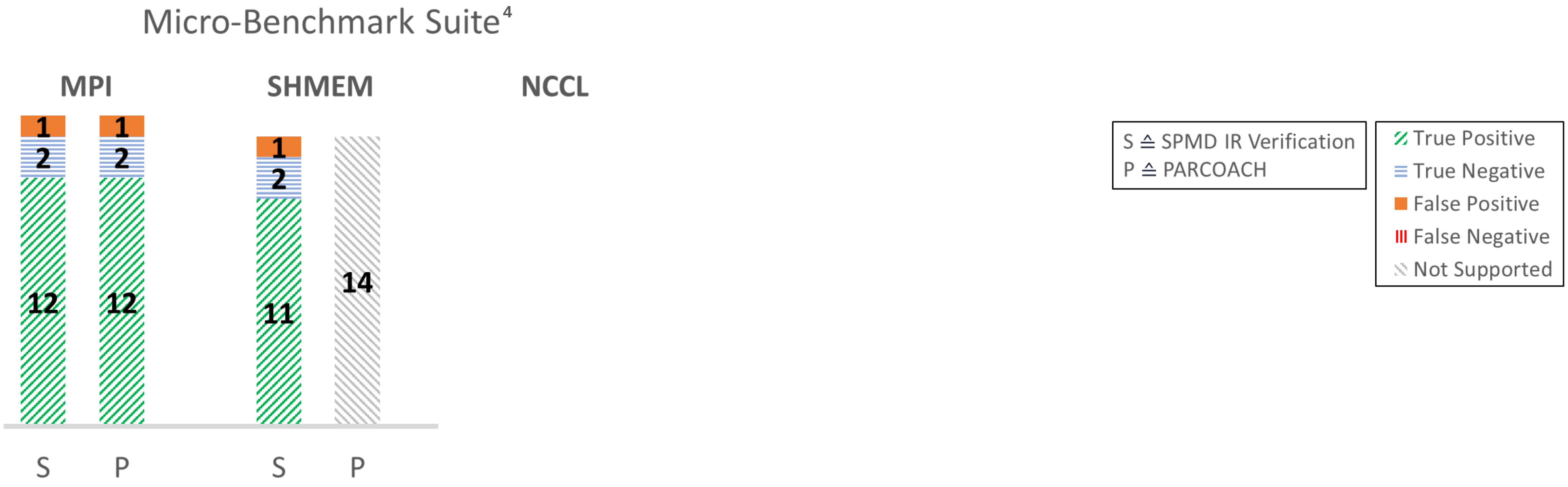
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



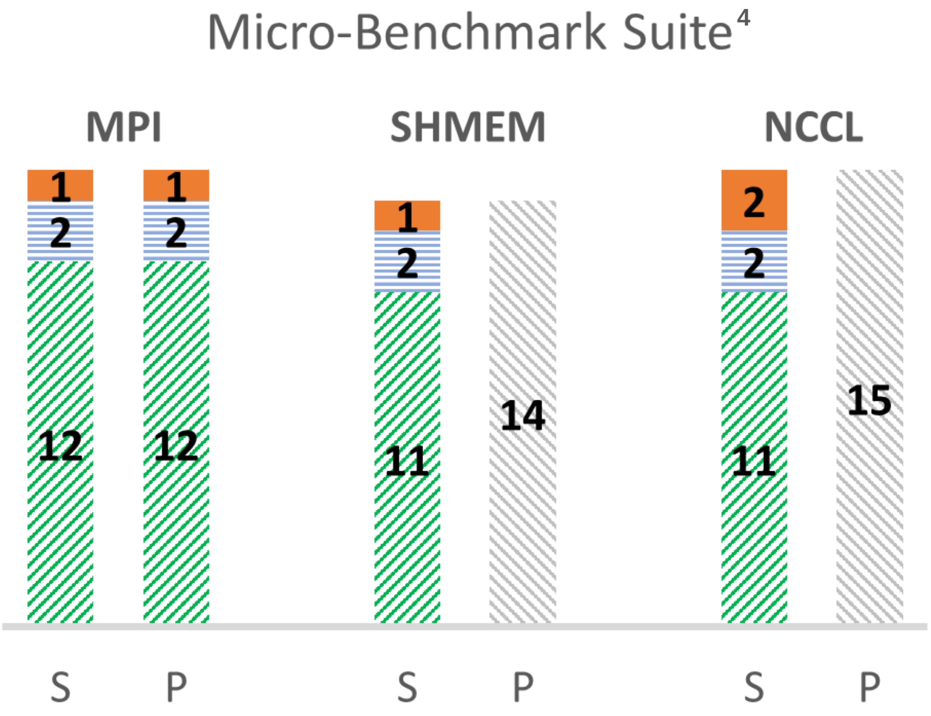
Evaluation: Results

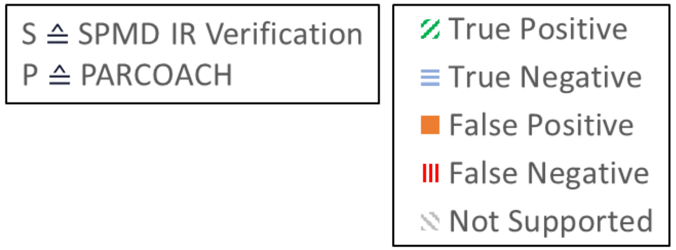
[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



Evaluation: Results

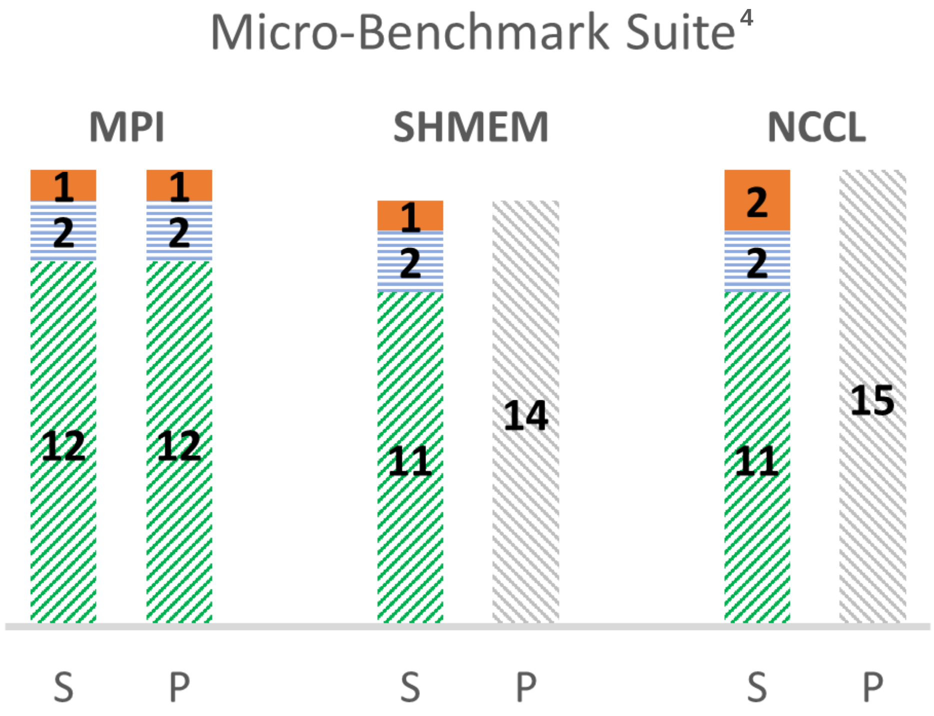
[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19





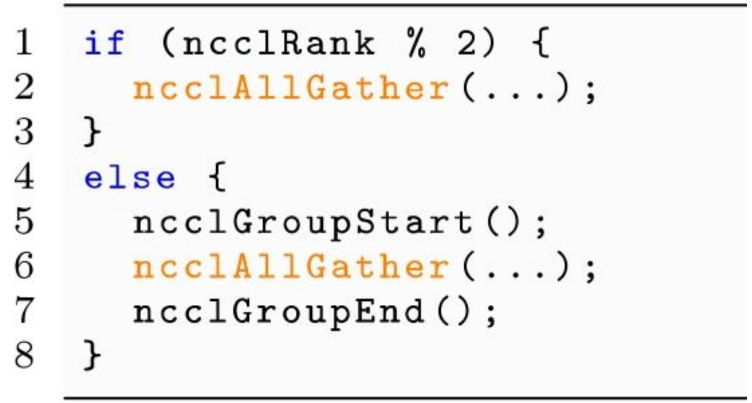
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19



S ≙ SPMD IR Verification
P ≙ PARCOACH

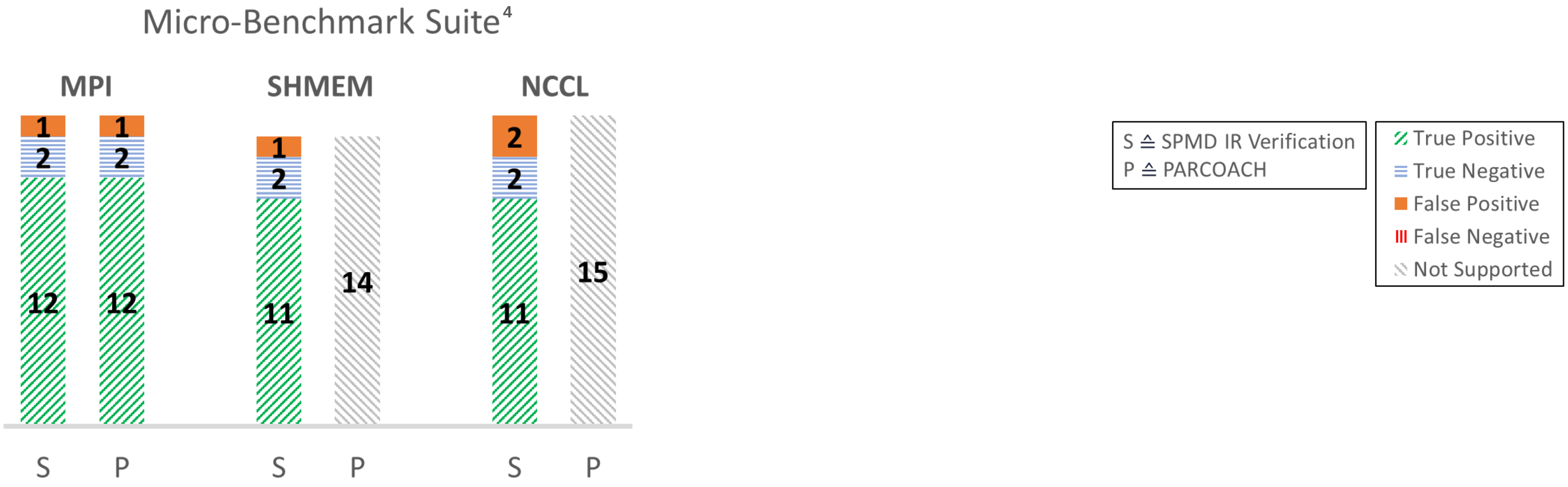
▨ True Positive
≡ True Negative
■ False Positive
Ⅲ False Negative
▨ Not Supported



```
if (ncclRank % 2) {
  ncclAllGather(...);
}
else {
  ncclGroupStart();
  ncclAllGather(...);
  ncclGroupEnd();
}
```

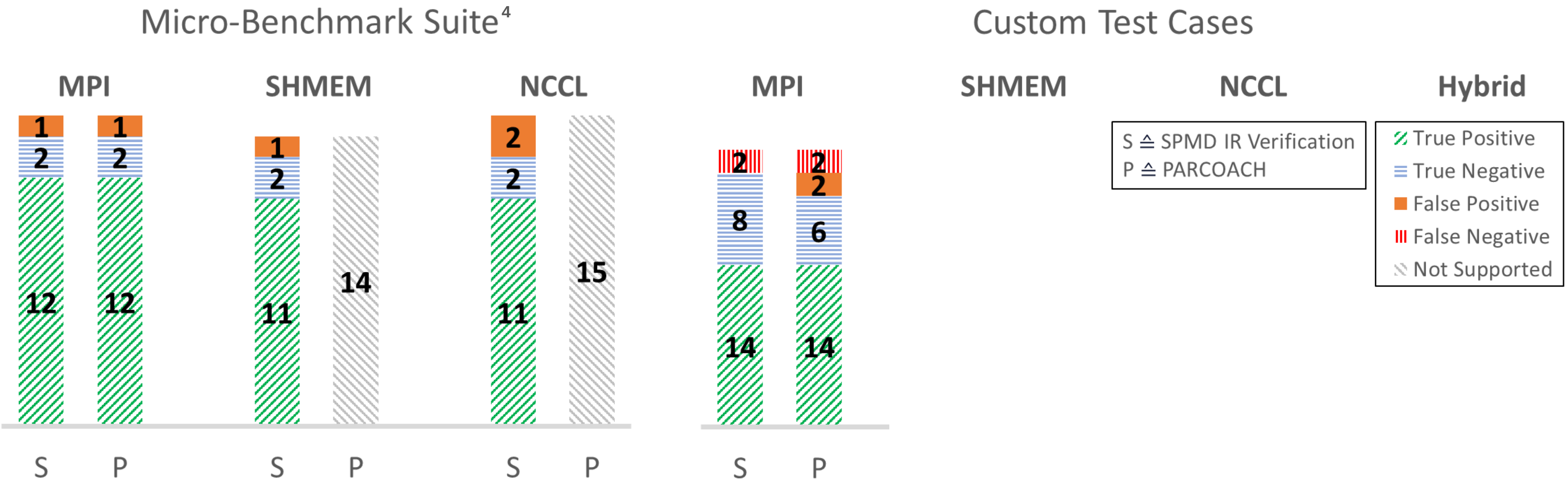
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19


Micro-Benchmark Suite[4]

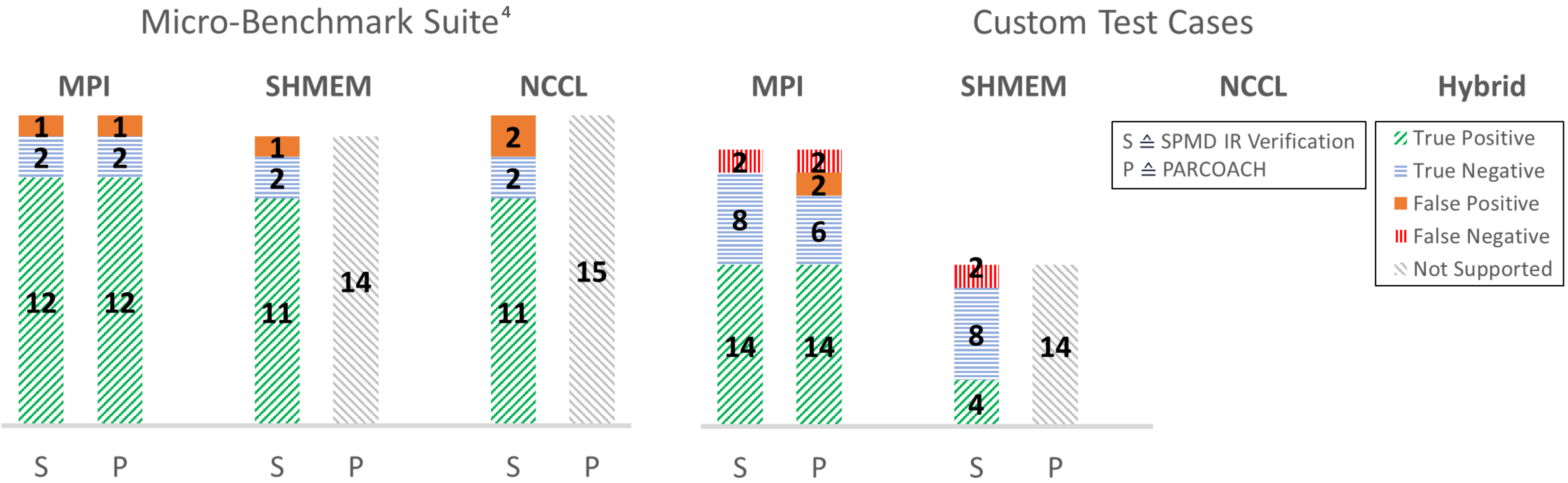
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



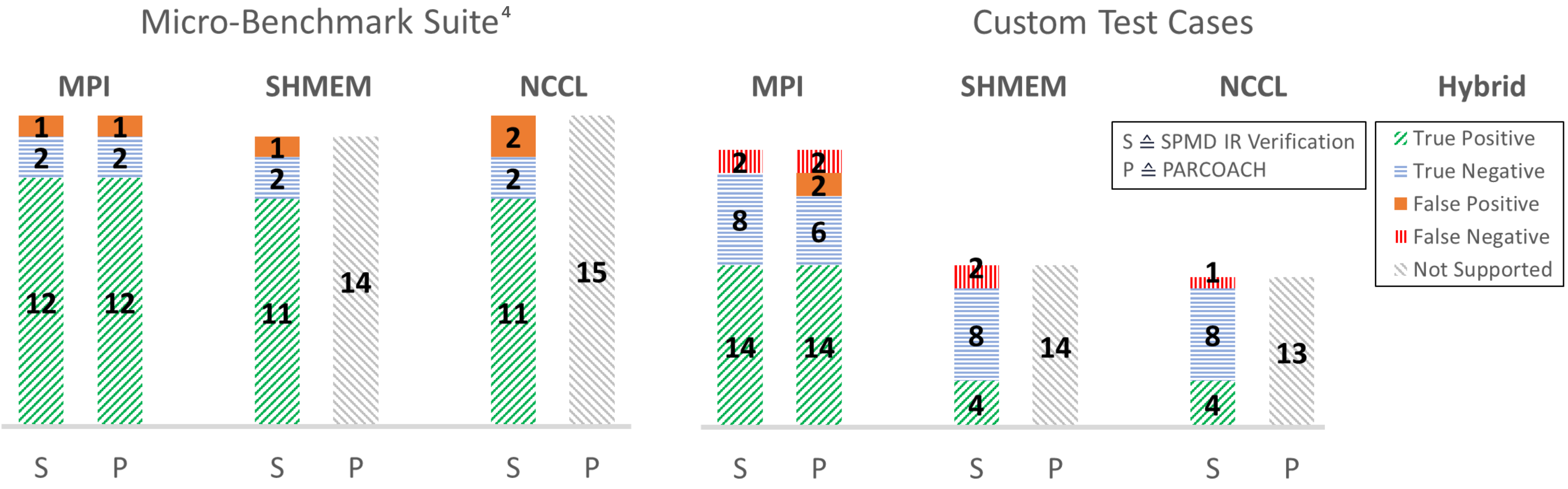
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



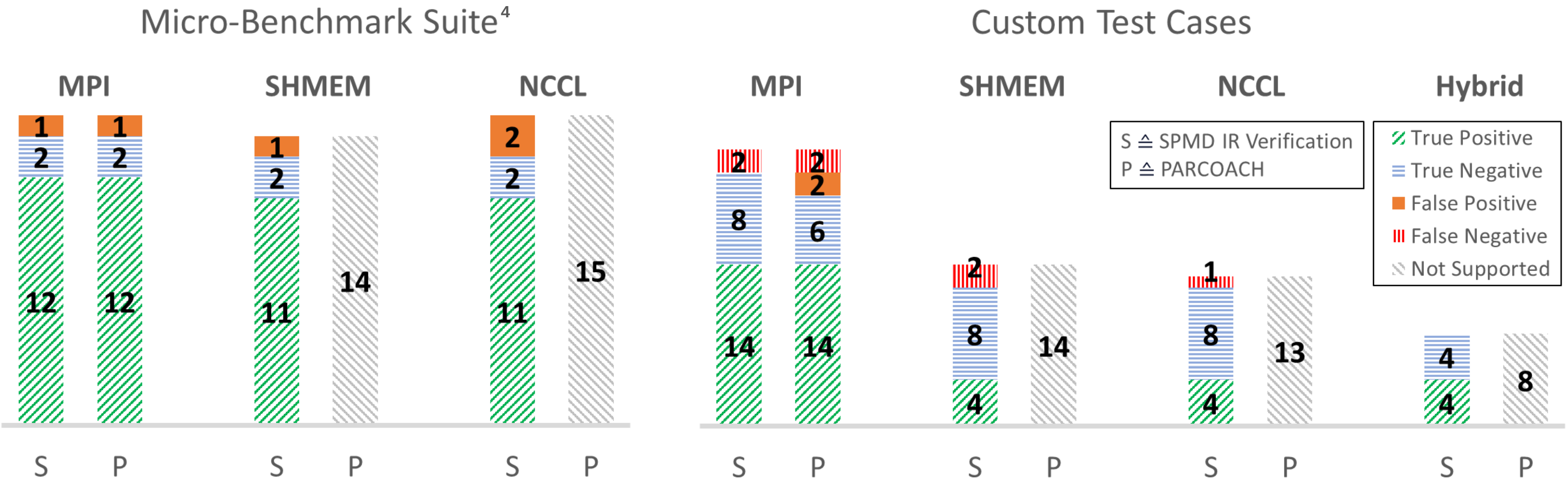
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par '19



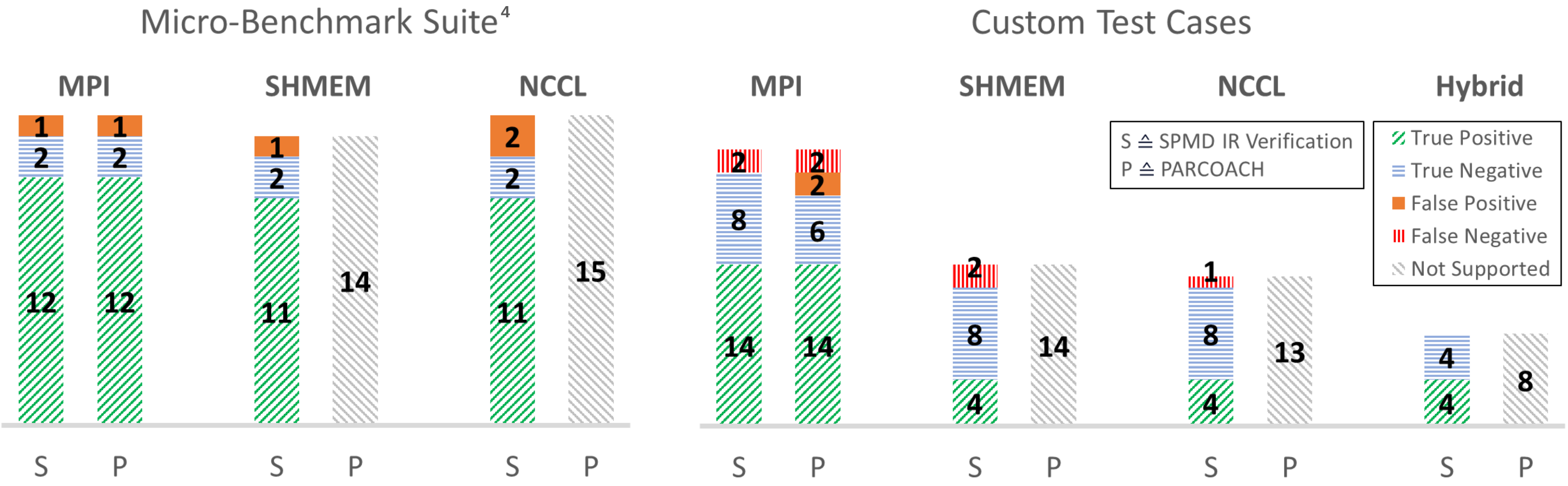
Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



Evaluation: Results

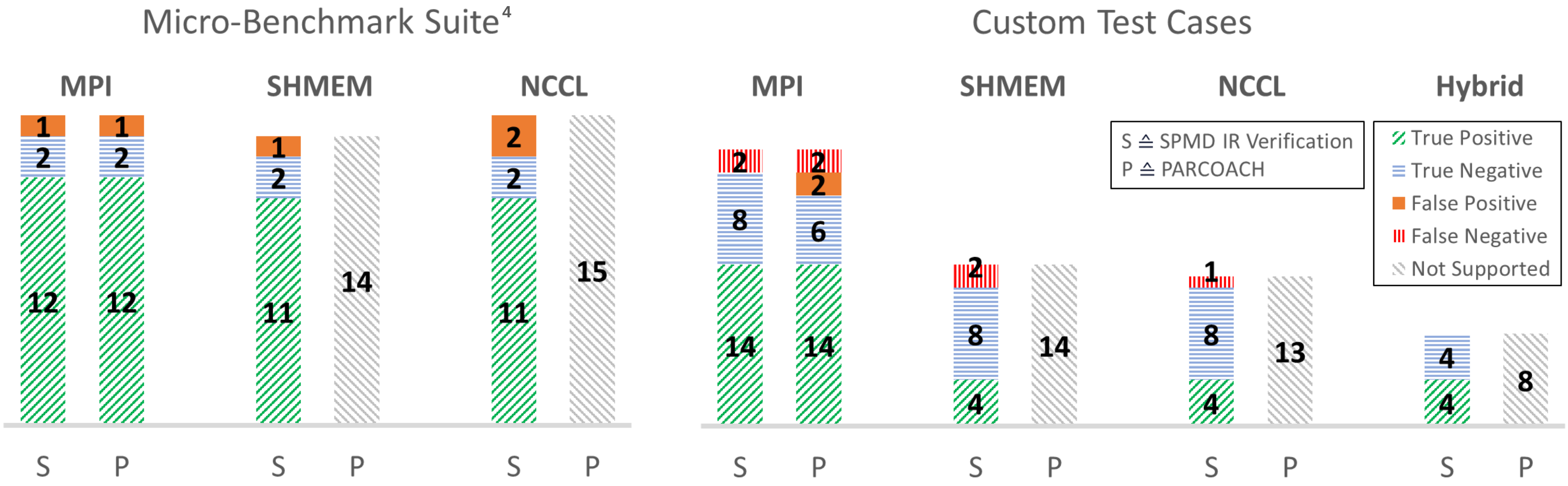
[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



- SPMD IR
  - covers two test cases correctly, where PARCOACH fails

Evaluation: Results

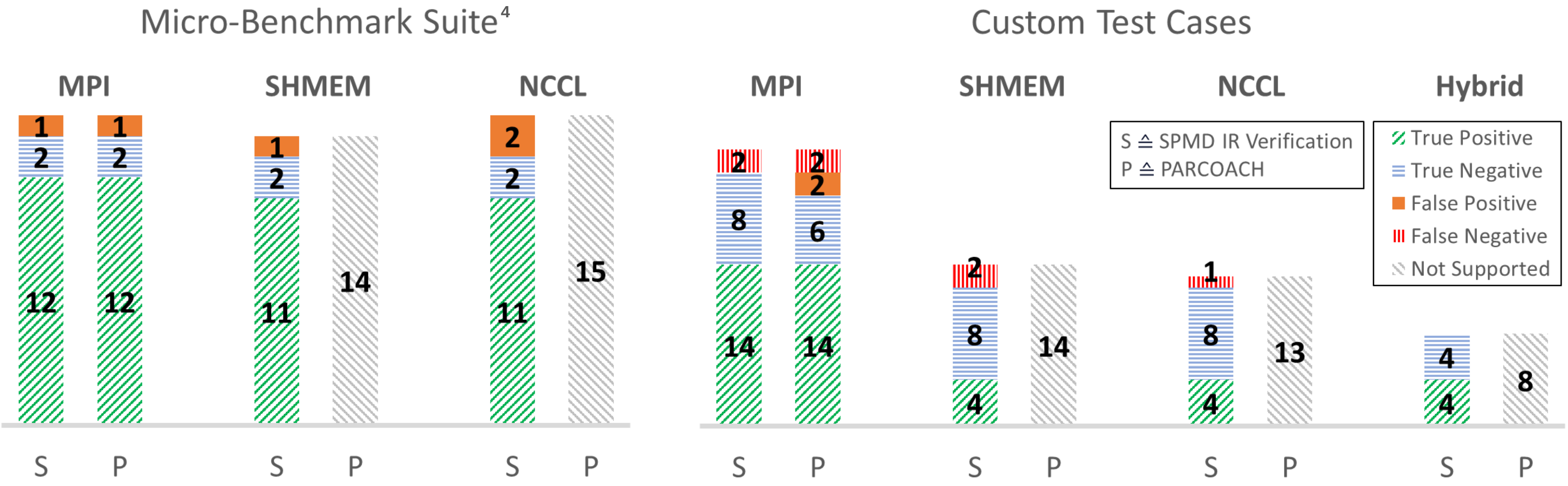
[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19



- SPMD IR
  - covers two test cases correctly, where PARCOACH fails
  - has one false positive for one NCCL port

Evaluation: Results

[4] MPI Codes from Huchant et al: Multi-Valued Expression Analysis for Collective Checking. Euro-Par ’19

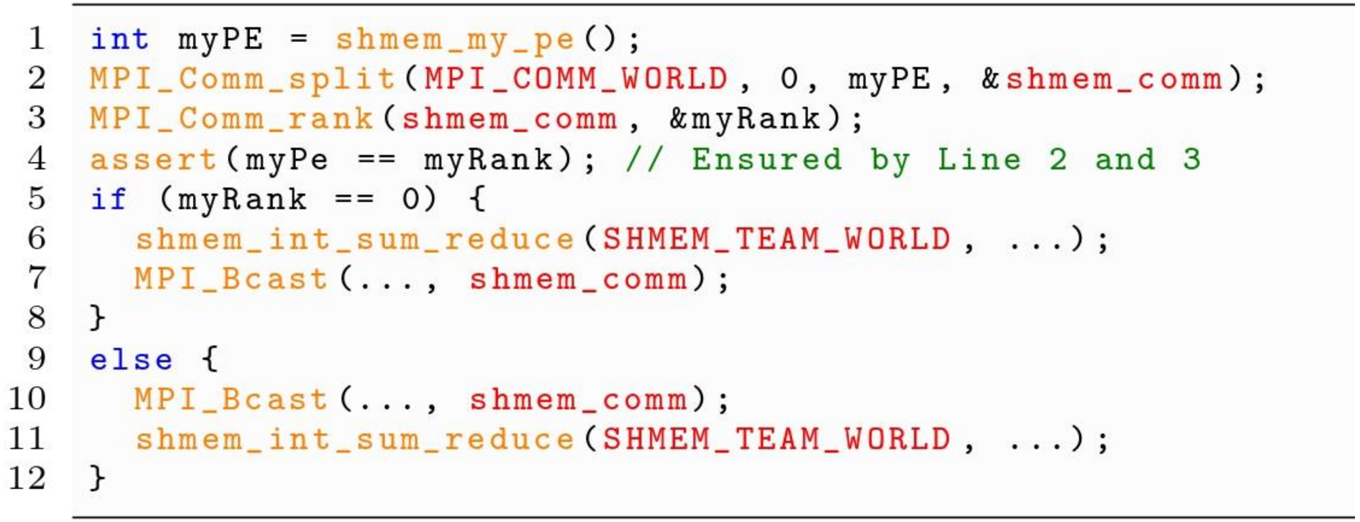


- SPMD IR
  - covers two test cases correctly, where PARCOACH fails
  - has one false positive for one NCCL port
  - besides MPI, also supports SHMEM, NCCL, and their hybrid combinations

Evaluation: Hybrid Example

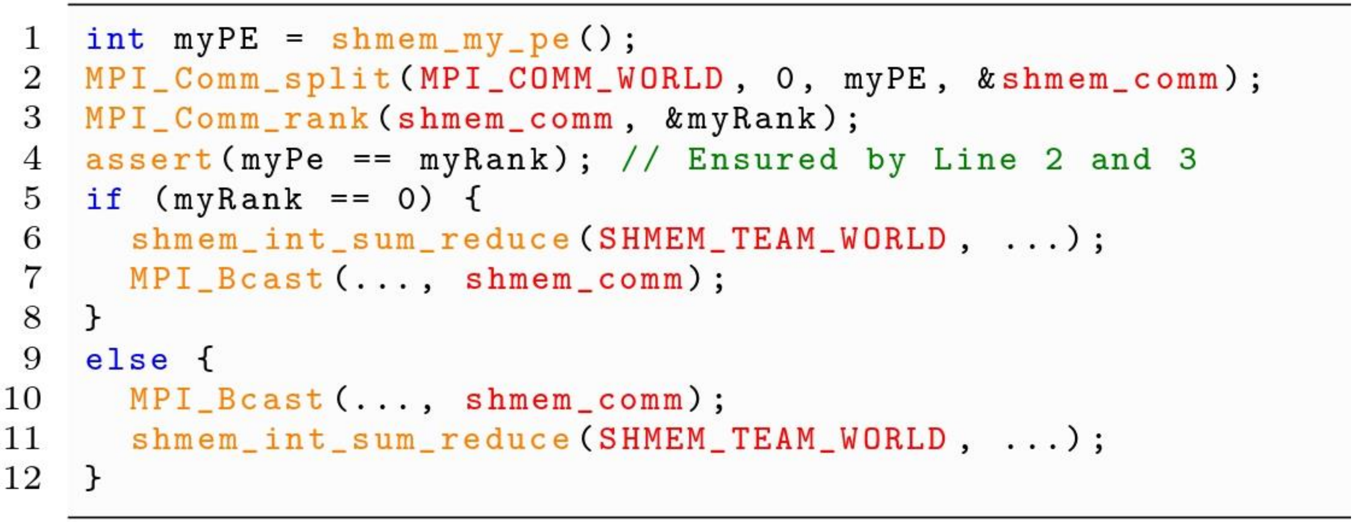
```
1  int myPE = shmem_my_pe();
2  MPI_Comm_split(MPI_COMM_WORLD, 0, myPE, &shmem_comm);
3  MPI_Comm_rank(shmem_comm, &myRank);
4  assert(myPe == myRank); // Ensured by Line 2 and 3
5
6
7
8
9
10
11
12
```

Evaluation: Hybrid Example



```
 1  int myPE = shmem_my_pe();
 2  MPI_Comm_split(MPI_COMM_WORLD, 0, myPE, &shmem_comm);
 3  MPI_Comm_rank(shmem_comm, &myRank);
 4  assert(myPe == myRank); // Ensured by Line 2 and 3
 5  if (myRank == 0) {
 6    shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
 7    MPI_Bcast(..., shmem_comm);
 8  }
 9  else {
10    MPI_Bcast(..., shmem_comm);
11    shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
12  }
```

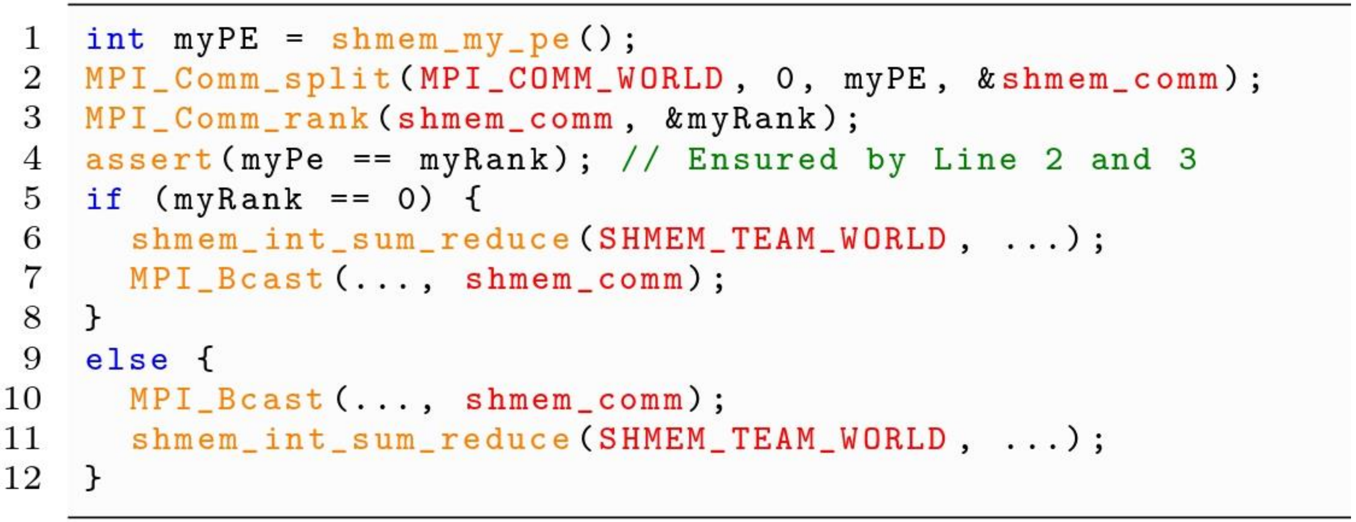
Evaluation: Hybrid Example



```
 1 int myPE = shmem_my_pe();
 2 MPI_Comm_split(MPI_COMM_WORLD, 0, myPE, &shmem_comm);
 3 MPI_Comm_rank(shmem_comm, &myRank);
 4 assert(myPe == myRank); // Ensured by Line 2 and 3
 5 if (myRank == 0) {
 6   shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
 7   MPI_Bcast(..., shmem_comm);
 8 }
 9 else {
10   MPI_Bcast(..., shmem_comm);
11   shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
12 }
```

- Understanding both program. models separately is not sufficient
- A tool needs to understand also the interaction / their combination

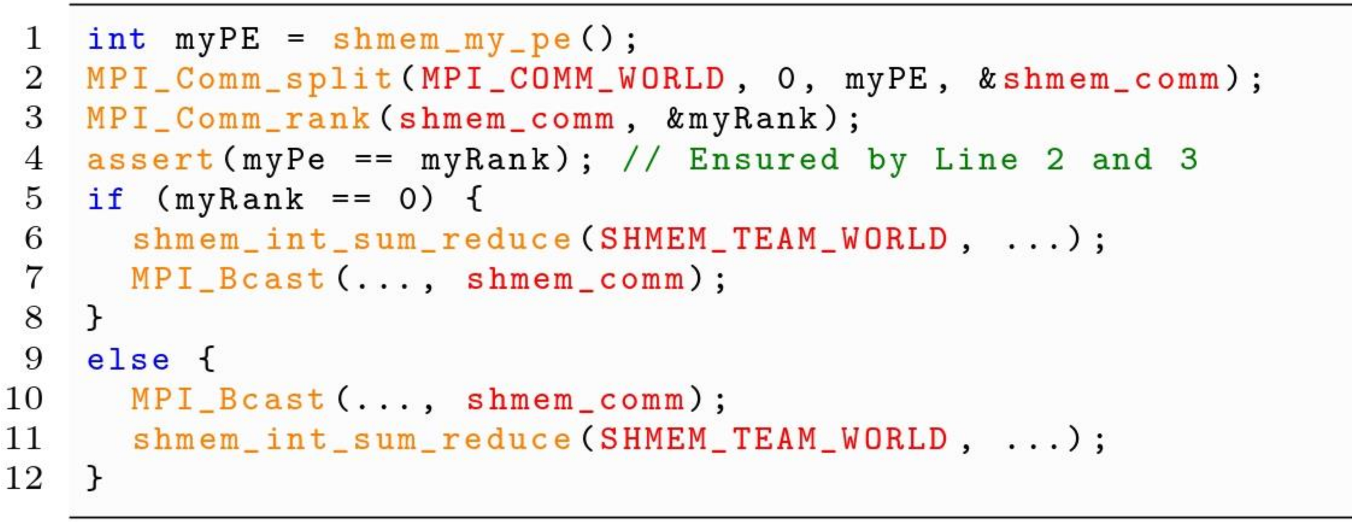
Evaluation: Hybrid Example



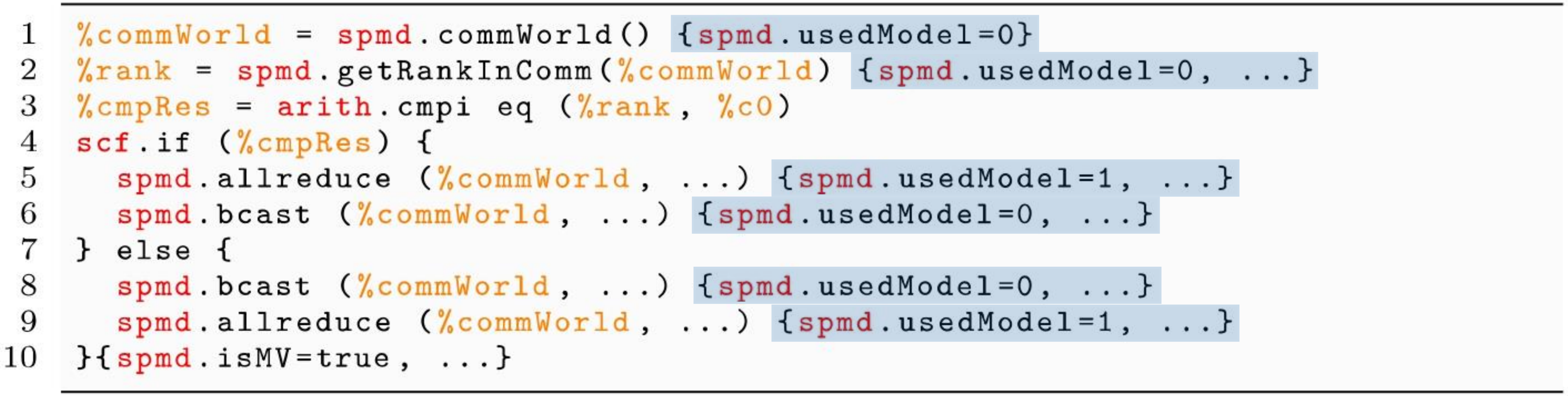
```
 1 int myPE = shmem_my_pe();
 2 MPI_Comm_split(MPI_COMM_WORLD, 0, myPE, &shmem_comm);
 3 MPI_Comm_rank(shmem_comm, &myRank);
 4 assert(myPe == myRank); // Ensured by Line 2 and 3
 5 if (myRank == 0) {
 6   shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
 7   MPI_Bcast(..., shmem_comm);
 8 }
 9 else {
10   MPI_Bcast(..., shmem_comm);
11   shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
12 }
```

- Understanding both program. models separately is not sufficient
- A tool needs to understand also the interaction / their combination
- The conversion to and representation in the unified SPMD IR addresses this issue

Evaluation: Hybrid Example



```
1  int myPE = shmem_my_pe();
2  MPI_Comm_split(MPI_COMM_WORLD, 0, myPE, &shmem_comm);
3  MPI_Comm_rank(shmem_comm, &myRank);
4  assert(myPe == myRank); // Ensured by Line 2 and 3
5  if (myRank == 0) {
6    shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
7    MPI_Bcast(..., shmem_comm);
8  }
9  else {
10   MPI_Bcast(..., shmem_comm);
11   shmem_int_sum_reduce(SHMEM_TEAM_WORLD, ...);
12 }
```



```
%commWorld = spmd.commWorld() {spmd.usedModel=0}
%rank = spmd.getRankInComm(%commWorld) {spmd.usedModel=0, ...}
%cmpRes = arith.cmpi eq (%rank, %c0)
scf.if (%cmpRes) {
  spmd.allreduce (%commWorld, ...) {spmd.usedModel=1, ...}
  spmd.bcast (%commWorld, ...) {spmd.usedModel=0, ...}
} else {
  spmd.bcast (%commWorld, ...) {spmd.usedModel=0, ...}
  spmd.allreduce (%commWorld, ...) {spmd.usedModel=1, ...}
}{spmd.isMV=true, ...}
```

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information
  - Unifying PGAS memory management calls with non-PGAS ones

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information
  - Unifying PGAS memory management calls with non-PGAS ones

- Loss of information due to abstraction
  - E.g., multiple calls could be mapped to a single operation

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information
  - Unifying PGAS memory management calls with non-PGAS ones

- Loss of information due to abstraction
  - E.g., multiple calls could be mapped to a single operation
  - Similar case led to one false positive in the evaluation for a NCCL port

Discussion

- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information
  - Unifying PGAS memory management calls with non-PGAS ones

- Loss of information due to abstraction
  - E.g., multiple calls could be mapped to a single operation
  - Similar case led to one false positive in the evaluation for a NCCL port
    - Could be approached by introducing an ID that distinguishes originally separate calls
    - but might be contrary to a unifying IR

Discussion

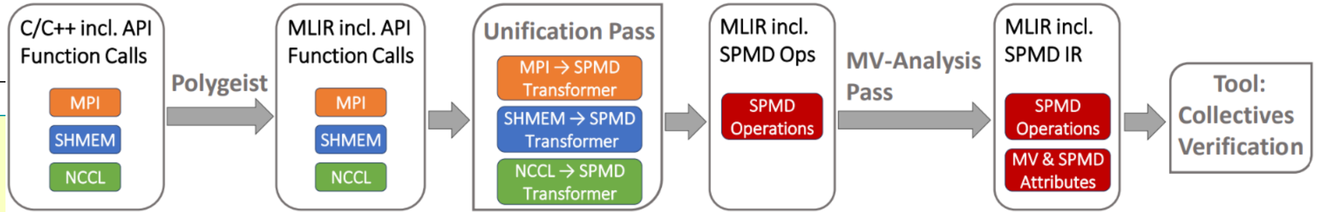
- MPI-centric design
  - MPI is an extensive standard and programming model
  - Large community support
  - De facto standard for distributed-memory programming

- Mostly, the API calls could be converted to SPMD IR operations
  - Non-blocking communication differences between MPI and NCCL
    - Possibly non-static incorporation of information
  - Unifying PGAS memory management calls with non-PGAS ones

- Loss of information due to abstraction
  - E.g., multiple calls could be mapped to a single operation
  - Similar case led to one false positive in the evaluation for a NCCL port
    - Could be approached by introducing an ID that distinguishes originally separate calls
    - but might be contrary to a unifying IR

- So far, executing kind and processes analyses limited to a few patterns for static cases

Conclusion

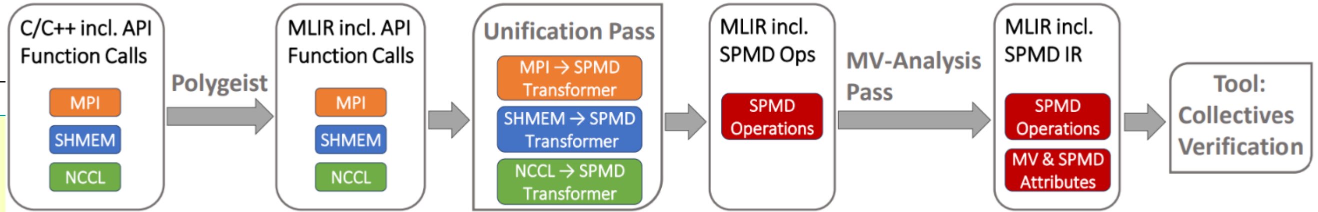

C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification

Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization

Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

Conclusion

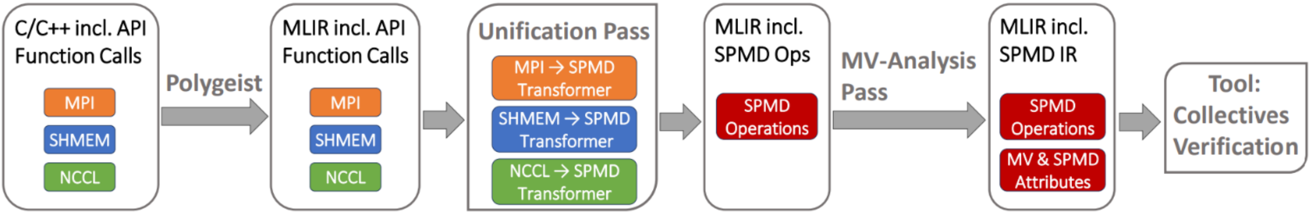

C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification

Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs

Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics
(P2P and Collectives)

Conclusion



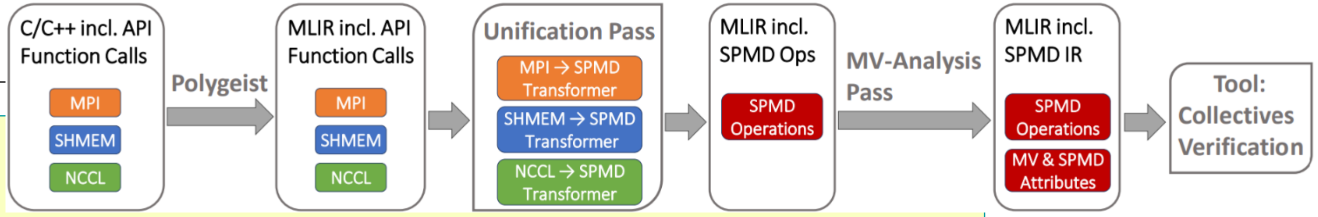
C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification

Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties

Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

Conclusion

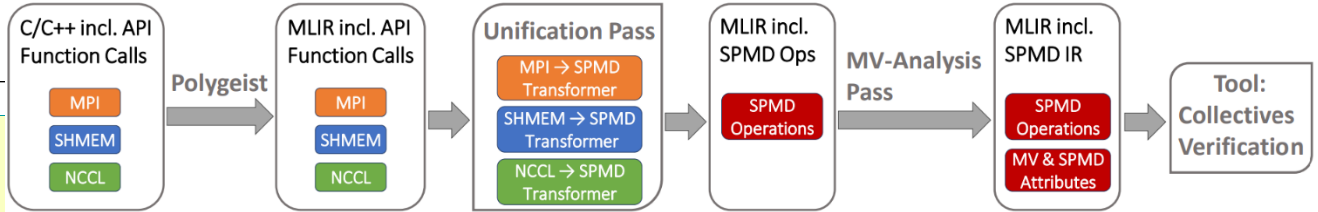


Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)

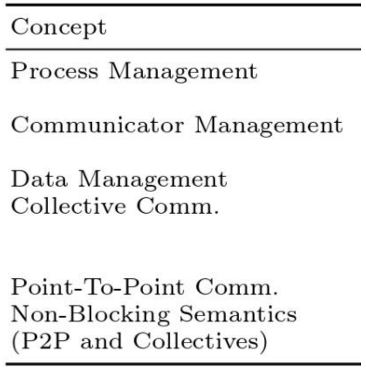
Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

Conclusion



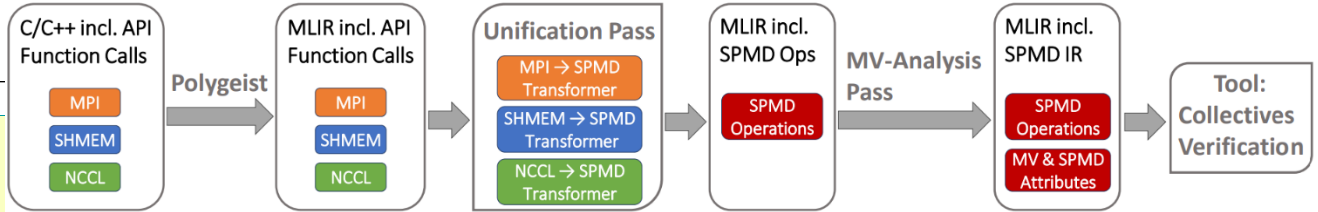
Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI



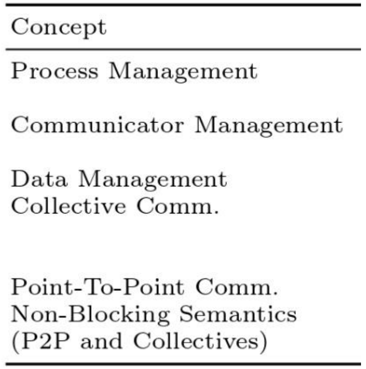
Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics (P2P and Collectives)

Conclusion


C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification

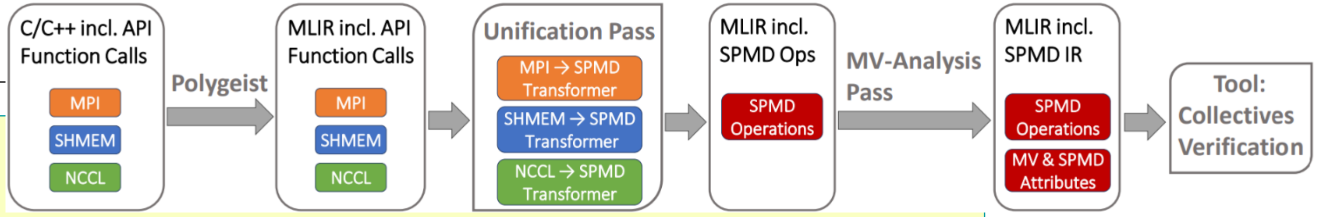
Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI
- Extended micro-benchmark suite and provide ports



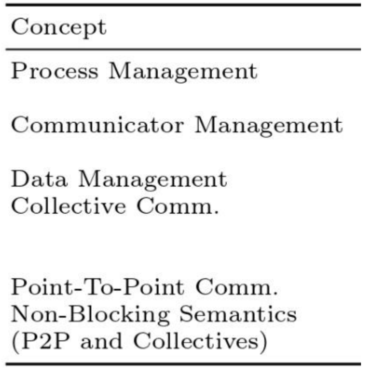
Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

Conclusion



Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI
- Extended micro-benchmark suite and provide ports

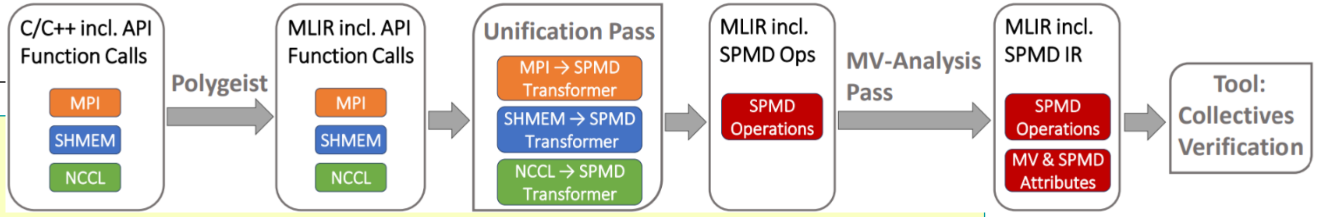


Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

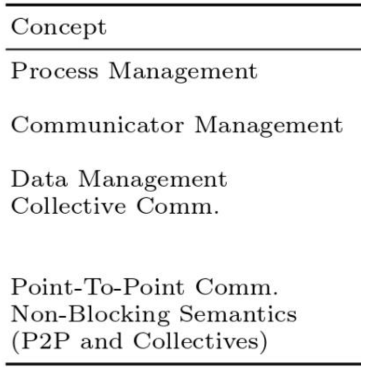
Future Work:

- Extend the execution kind and executing processes analysis, explore new use cases

Conclusion



C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification



Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics (P2P and Collectives)

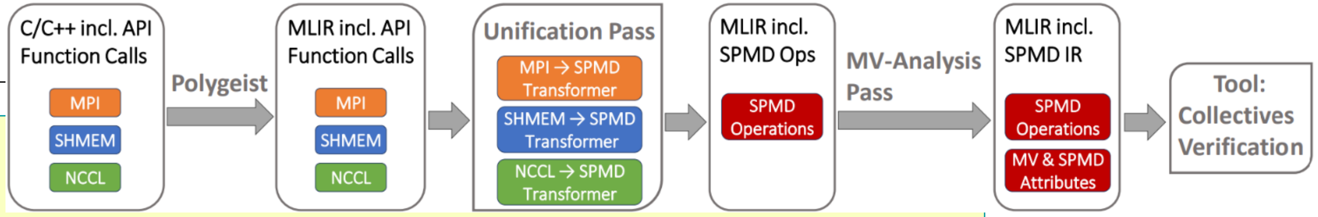
Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI
- Extended micro-benchmark suite and provide ports

Future Work:

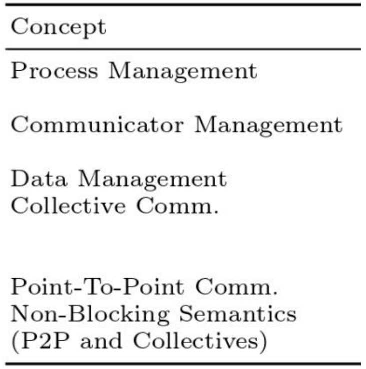
- Extend the execution kind and executing processes analysis, explore new use cases
- Apply the SPMD IR to other program. models such as NVSHMEM, GASPI, and UPC++

Conclusion


C/C++ incl. API Function Calls
MPI
SHMEM
NCCL
Polygeist
MLIR incl. API Function Calls
MPI
SHMEM
NCCL
Unification Pass
MPI → SPMD Transformer
SHMEM → SPMD Transformer
NCCL → SPMD Transformer
MLIR incl. SPMD Ops
SPMD Operations
MV-Analysis Pass
MLIR incl. SPMD IR
SPMD Operations
MV & SPMD Attributes
Tool: Collectives Verification

Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI
- Extended micro-benchmark suite and provide ports

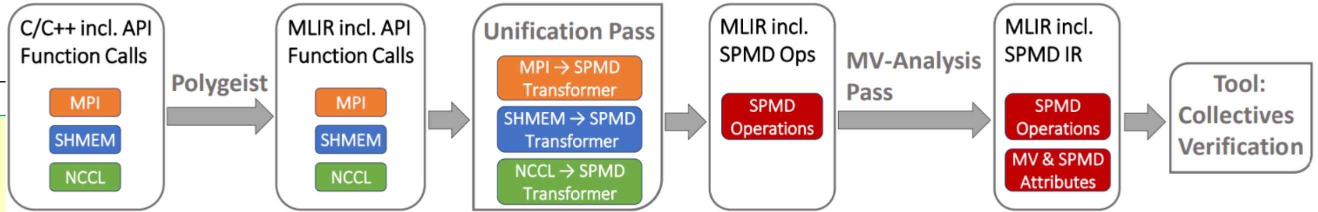


Concept
Process Management
Communicator Management
Data Management
Collective Comm.
Point-To-Point Comm.
Non-Blocking Semantics (P2P and Collectives)

Future Work:

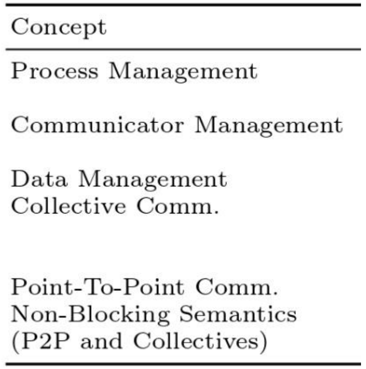
- Extend the execution kind and executing processes analysis, explore new use cases
- Apply the SPMD IR to other program. models such as NVSHMEM, GASPI, and UPC++
- Integrate other SPMD features / paradigms such as RMA or OpenMP

Conclusion



Summary:

- Established unifying and abstracting SPMD IR
  - Allowing programming-model-independent analysis and optimization
  - For SPMD-like programming models, incl. distributed-memory, PGAS, and GPU programs
- Introduced a multi-value IR and extended analysis for important SPMD properties
- Evaluated on collectives verification, outperforming PARCOACH (programming model support, hybrid codes, two extensions)
  - First collectives verification for NCCL and SHMEM, and their hybrid combinations with MPI
- Extended micro-benchmark suite and provide ports



Concept
Process Management
Communicator Management
Data Management Collective Comm.
Point-To-Point Comm. Non-Blocking Semantics (P2P and Collectives)

Future Work:

- Extend the execution kind and executing processes analysis, explore new use cases
- Apply the SPMD IR to other program. models such as NVSHMEM, GASPI, and UPC++
- Integrate other SPMD features / paradigms such as RMA or OpenMP
- Assess approach and workflow on more realistic codes such as proxy apps