llvm/flang : OpenMP update

Kiran Chandramohan

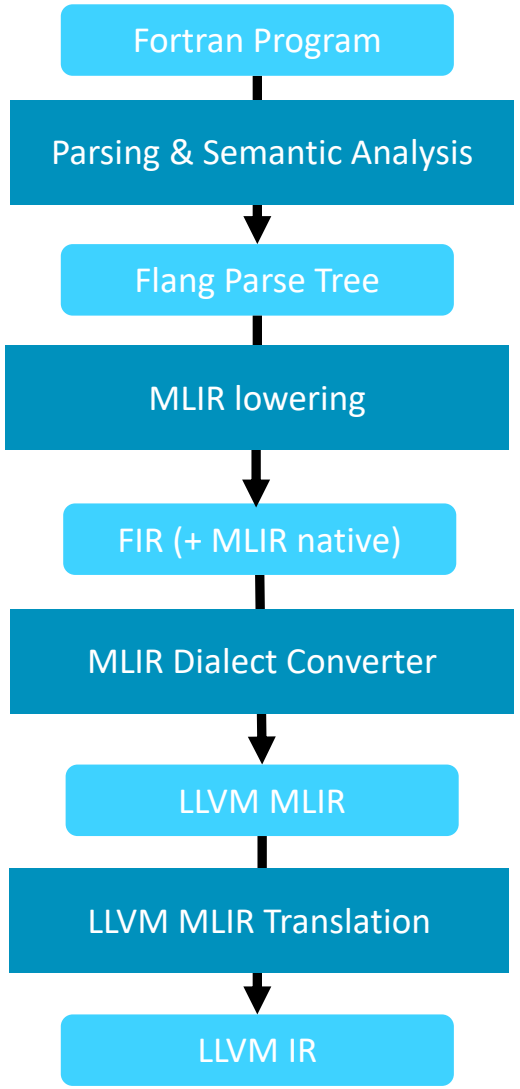
13 Sep 2021

Contents

Introduction

- The Flang Fortran frontend became part of LLVM on April 9, 2020
  - Flang started off as the F18 project at Nvidia in collaboration with US DoE
  - Arm, AMD, Huawei, Linaro, US DoE labs and a few individuals are contributing
  - Intends to replace the Classic Flang project (github.com/flang-compiler/flang)
    - Classic Flang is derived from pgfortran/nvfortran
    - AMD, Arm, Huawei Fortran frontends based on Classic Flang
      - All are expected to switch to llvm/flang
- Built using modern technologies
  - Written in C++17
  - Uses MLIR
- Support for OpenMP is important in HPC
  - Plan to support the latest standards
  - Classic Flang has partial support for OpenMP 4.5

Flang High Level Flow



- Flang frontend takes in Fortran program and generates LLVM IR
- Has a high-level IR : FIR
  - For ease of lowering and transformations
  - Uses MLIR infrastructure for Fortran IR (FIR)
  - IR has representation for Fortran types and constructs like loops, array expressions etc
- Parse Tree is lowered to a mix of FIR and other native MLIR dialects
  - These dialects are lowered to LLVM dialect
  - LLVM dialect is translated to LLVM IR (mostly one-to-one)

arm

Example : Flang High Level Flow

Fortran source

```
  c = a + b
end
```

Flang parse tree

```
Program -> ProgramUnit -> MainProgram
| SpecificationPart
| | ImplicitPart ->
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> ActionStmt -> AssignmentStmt = 'c=a+b'
| | | Variable = 'c'
| | | | Designator -> DataRef -> Name = 'c'
| | | Expr = 'a+b'
| | | | Add
| | | | | Expr = 'a'
| | | | | | Designator -> DataRef -> Name = 'a'
| | | | | Expr = 'b'
| | | | | | Designator -> DataRef -> Name = 'b'
| EndProgramStmt ->
```

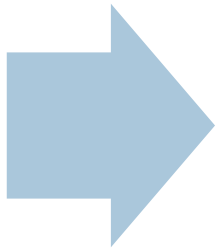
MLIR: FIR + std

```
func @_QQmain() {
  %0 = fir.alloca f32 {bindc_name = "a", uniq_name = "_QEa"}
  %1 = fir.alloca f32 {bindc_name = "b", uniq_name = "_QEb"}
  %2 = fir.alloca f32 {bindc_name = "c", uniq_name = "_QEc"}
  %3 = fir.load %0 : !fir.ref<f32>
  %4 = fir.load %1 : !fir.ref<f32>
  %5 = addf %3, %4 : f32
  fir.store %5 to %2 : !fir.ref<f32>
  return
}
```

Example : Flang High Level Flow

MLIR: LLVM dialect

```
llvm.func @_QQmain() {
  %0 = llvm.mlir.constant(1 : i64) : i64
  %1 = llvm.alloca %0 x f32 {bindc_name = "a", in_type = f32, uniq_name = "_QEa"} : (i64) -> !llvm.ptr<f32>
  %2 = llvm.mlir.constant(1 : i64) : i64
  %3 = llvm.alloca %2 x f32 {bindc_name = "b", in_type = f32, uniq_name = "_QEb"} : (i64) -> !llvm.ptr<f32>
  %4 = llvm.mlir.constant(1 : i64) : i64
  %5 = llvm.alloca %4 x f32 {bindc_name = "c", in_type = f32, uniq_name = "_QEc"} : (i64) -> !llvm.ptr<f32>
  %6 = llvm.load %1 : !llvm.ptr<f32>
  %7 = llvm.load %3 : !llvm.ptr<f32>
  %8 = llvm.fadd %6, %7  : f32
  llvm.store %8, %5 : !llvm.ptr<f32>
  llvm.return
 }
```



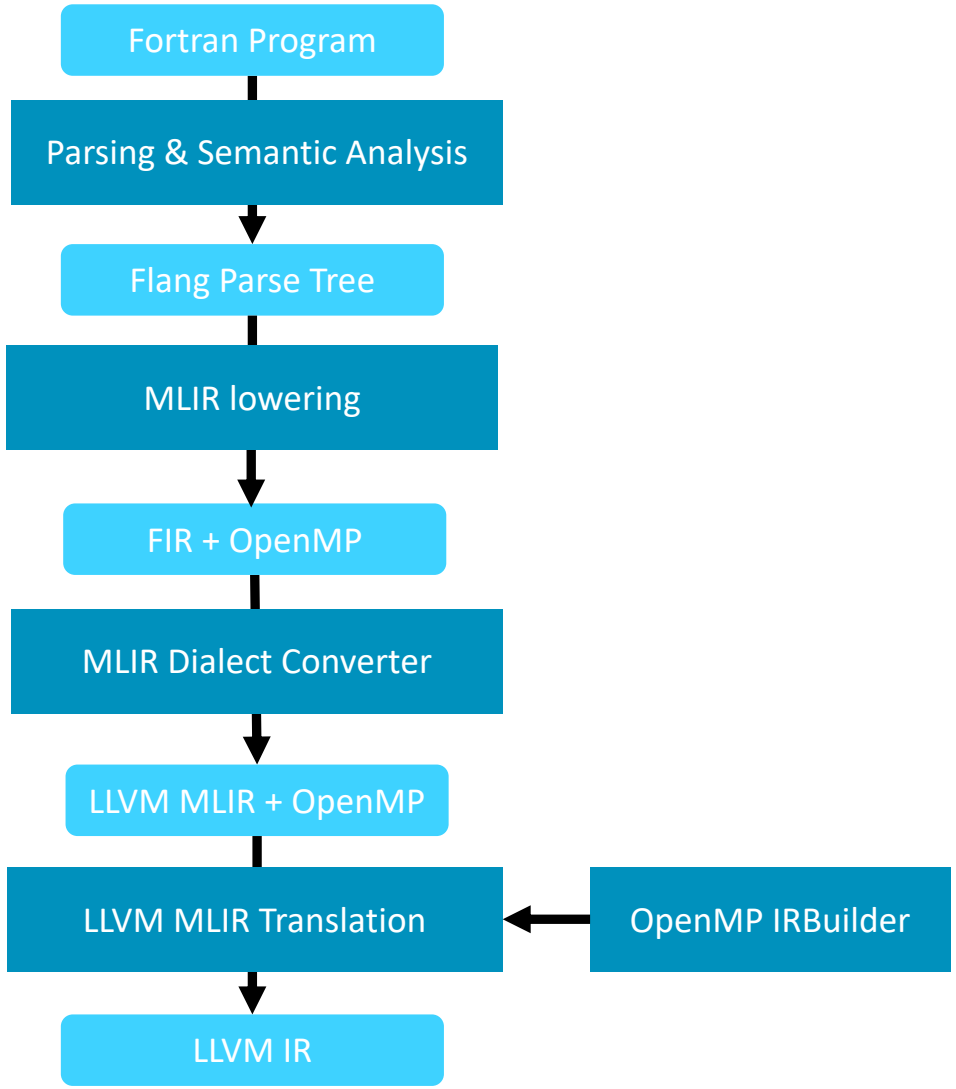
LLVM IR

```
define void @_QQmain() {
  %1 = alloca float, i64 1, align 4
  %2 = alloca float, i64 1, align 4
  %3 = alloca float, i64 1, align 4
  %4 = load float, float* %1, align 4
  %5 = load float, float* %2, align 4
  %6 = fadd float %4, %5
  store float %6, float* %3, align 4
  ret void
}
```

OpenMP support for Flang

- Parse-tree extended to represent OpenMP constructs
- FIR represents the Fortran constructs
  - Need a similar representation for OpenMP
  - The OpenMP dialect (https://mlir.llvm.org/docs/Dialects/OpenMPDialect/)
- OpenMP LLVM IR Codegen already present in Clang
  - Inserting calls to OpenMP runtime, Outlining OpenMP regions
  - Rewriting all this code for Flang seems un-necessary
  - OpenMP IRBuilder project
    - Refactor LLVM IR Codegen from Clang into LLVM
    - Use the OpenMP IRBuilder in the Flang flow
    - Traditional MLIR flow LLVM dialect is the composition layer
      - All dialects are converted to LLVM dialect
      - Cannot reuse code with OpenMP IRBuilder
    - Call the OpenMP IRBuilder while translating from LLVM Dialect to LLVM IR
- Moving in the direction of a single source of OpenMP info in llvm-project
  - `llvm/include/llvm/Frontend/OpenMP/OMP.td`

Flang OpenMP High Level Flow



- Flang parse-tree augmented to represent OpenMP
- Parse-tree is lowered to a mix of FIR + OpenMP + other native MLIR dialects
- Converted to LLVM + OpenMP MLIR dialects
- LLVM IR generated from this mix using the OpenMP IRBuilder

Example : OpenMP High Level Flow

Fortran source with OpenMP

```
!$omp parallel
  c = a + b
!$omp end parallel
end
```

Flang parse tree

```
| …
| | ExecutionPartConstruct -> ExecutableConstruct -> OpenMPConstruct -> OpenMPBlockConstruct
| | | OmpBeginBlockDirective
| | | | OmpBlockDirective -> llvm::omp::Directive = parallel
| | | | OmpClauseList ->
| | | Block
| | | | ExecutionPartConstruct -> ExecutableConstruct -> ActionStmt -> AssignmentStmt = 'c=a+b'
……
| | | OmpEndBlockDirective
| | | | OmpBlockDirective -> llvm::omp::Directive = parallel
| | | | OmpClauseList ->
| EndProgramStmt ->
```

MLIR: FIR + std + OpenMP

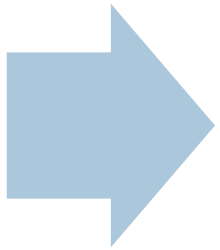
```
func @_QQmain() {
  %0 = fir.alloca f32 {bindc_name = "a", uniq_name = "_QEa"}
  %1 = fir.alloca f32 {bindc_name = "b", uniq_name = "_QEb"}
  %2 = fir.alloca f32 {bindc_name = "c", uniq_name = "_QEc"}
  omp.parallel {
    %3 = fir.load %0 : !fir.ref<f32>
    %4 = fir.load %1 : !fir.ref<f32>
    %5 = addf %3, %4 : f32
    fir.store %5 to %2 : !fir.ref<f32>
    omp.terminator
  }
  return
}
```

Example : OpenMP High Level Flow

MLIR: LLVM + OpenMP dialect

```
llvm.func @_QQmain() {
  %0 = llvm.mlir.constant(1 : i64) : i64
  %1 = llvm.alloca %0 x f32 {bindc_name = "a", in_type = f32, uniq_name = "_QEa"} : (i64) -> !llvm.ptr<f32>
  %2 = llvm.mlir.constant(1 : i64) : i64
  %3 = llvm.alloca %2 x f32 {bindc_name = "b", in_type = f32, uniq_name = "_QEb"} : (i64) -> !llvm.ptr<f32>
  %4 = llvm.mlir.constant(1 : i64) : i64
  %5 = llvm.alloca %4 x f32 {bindc_name = "c", in_type = f32, uniq_name = "_QEc"} : (i64) -> !llvm.ptr<f32>
  omp.parallel {
    %6 = llvm.load %1 : !llvm.ptr<f32>
    %7 = llvm.load %3 : !llvm.ptr<f32>
    %8 = llvm.fadd %6, %7  : f32
    llvm.store %8, %5 : !llvm.ptr<f32>
    omp.terminator
  }
  llvm.return
 }
```

Use OpenMP IRBuilder



LLVM IR

```
omp_parallel:
  call void (%struct.ident_t*, i32, void (i32*, i32*, ...)*, ...) @__kmpc_fork_call(...)@_QQmain..omp_par to void (i32*, i32*, ...)*), float* %1, float* %2, float* %3)



; Function Attrs: norecurse nounwind
define internal void @_QQmain..omp_par(i32* noalias %tid.addr, i32* noalias %zero.addr, float* %0, float* %1, float* %2) #0 {
...
omp.par.region:
  %4 = load float, float* %0, align 4
  %5 = load float, float* %1, align 4
  %6 = fadd float %4, %5
  store float %6, float* %2, align 4
...
}
```

OpenMP Parsing

- OpenMP constructs are represented in the parse tree as:
  - Executable Constructs: OpenMPConstruct
  - Declarative Constructs: OpenMPDeclarativeConstruct
  - Flang uses C++17 variants in the parse tree representation

```
struct OpenMPConstruct {
  UNION_CLASS_BOILERPLATE(OpenMPConstruct);
  std::variant<OpenMPStandaloneConstruct, OpenMPSectionsConstruct,
      OpenMPLoopConstruct, OpenMPBlockConstruct, OpenMPAtomicConstruct,
      OpenMPCriticalConstruct>
      u;
};
```

- Parser is declaratively written

```
TYPE_PARSER(sourced(construct<OmpCriticalDirective>(verbatim("CRITICAL"_tok),
                maybe(parenthesized(name)), maybe(Parser<OmpClause>{}))) /
    endOmpLine)

TYPE_PARSER(construct<OpenMPCriticalConstruct>(
    Parser<OmpCriticalDirective>{}, block, Parser<OmpEndCriticalDirective>{}))
```

OpenMP Semantic Checks

- Checks to ensure that Constructs and Clauses conform to the standard.
  - Some checks are autogenerated : Permitted clauses in a construct
  - Some are manually written : Checks specific to Fortran
  - Snippet from `llvm/include/llvm/Frontend/OpenMP/OMP.td`

```
def OMP_TargetData : Directive<"target data"> {
  let allowedClauses = [
    VersionedClause<OMPC_UseDevicePtr>,
    VersionedClause<OMPC_UseDeviceAddr, 50>
  ];
  let allowedOnceClauses = [
    VersionedClause<OMPC_Device>,
    VersionedClause<OMPC_If>
  ];
  let requiredClauses = [
    VersionedClause<OMPC_Map>
  ];
}
```

OpenMP Dialect in MLIR

- MLIR is a generic framework for building IRs
  - Can declaratively write definition of operations
  - Generates parsers, printers, builder functions
  - Can override with custom functions for these
- OpenMP dialect is a readable high-level IR
  - Models the standard
- Operations corresponding to constructs
  - Clauses represented as operands
    - Sometimes can be operations (reduction)
- Different kinds of operations
  - Region
    - With : Parallel, Master, Worksharing loop etc
    - Without : Barrier
  - Like directives : Enclose source code : Parallel
  - Loop like : Includes the loop : Worksharing loop

OpenMP Barrier: Definition of a basic operation

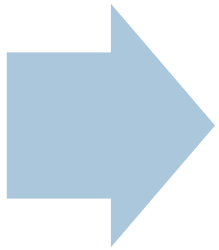
- Operation corresponding to barrier (`omp.barrier`)
- Declaratively defined

```
def OpenMP_Dialect : Dialect {
  let name = "omp";
}

class OpenMP_Op<string mnemonic, list<OpTrait> traits = []> :
      Op<OpenMP_Dialect, mnemonic, traits>;

def BarrierOp : OpenMP_Op<"barrier"> {
  let summary = "barrier construct";
  let description = [{
    The barrier construct specifies an explicit barrier at the point at which
    the construct appears.
  }];

  let assemblyFormat = “attr-dict”;
}
```

OpenMP Worksharing Loop : Definition

Fortran + OpenMP source

```
!$omp parallel do
do i = 1, a
  x = x + I
end do
```



OpenMP MLIR

```
omp.parallel {
 omp.wsloop (%i) : i32 = (%c1) to (%a)
 step (%c1) inclusive {
 }
}
```

OpenMP Worksharing Loop: Definition of a complex operation

```
def WsLoopOp : OpenMP_Op<"wsloop", [AttrSizedOperandSegments,
                                    AllTypesMatch<["lowerBound", "upperBound", "step"]>]> {
  let summary = "workshare loop construct";
  let arguments = (ins Variadic<IntLikeType>:$lowerBound,
              Variadic<IntLikeType>:$upperBound,
              Variadic<IntLikeType>:$step,
              …);
  let builders = [
    OpBuilder<(ins "ValueRange":$lowerBound, "ValueRange":$upperBound,
                "ValueRange":$step,
                CArg<"ArrayRef<NamedAttribute>", "{}">:$attributes)>,
    …
  ];
  let regions = (region AnyRegion:$region);
  …
  let parser = [{ return parseWsLoopOp(parser, result); }];
  let printer = [{ return printWsLoopOp(p, *this); }];
}
```

OpenMP Collapse

Fortran + OpenMP source

```
!$omp parallel do collapse(3)
do i = 1, a
 do j = 1, b
  do k = 1, c
   x = x + i + j + k
  end do
 end do
end do
```

MLIR without collapse

```
…
omp.parallel {
 omp.wsloop (%i) : i32 = (%c1) to (%a)
step (%c1) collapse(3) inclusive {
  fir.do_loop %arg5 = %c1 to %6 step
%c1 {
   fir.do_loop %arg6 = %c1 to %7 step
%c1  {
    ….
   }
  }
 }
}
```

MLIR with collapse

```
…
omp.parallel {
 omp.wsloop (%i, %j, %k) : i32 = (%c1,
%c1, %c1) to (%a, %b,%c) step (%c1,
%c1, %c1) inclusive {
  …
 }
}
```

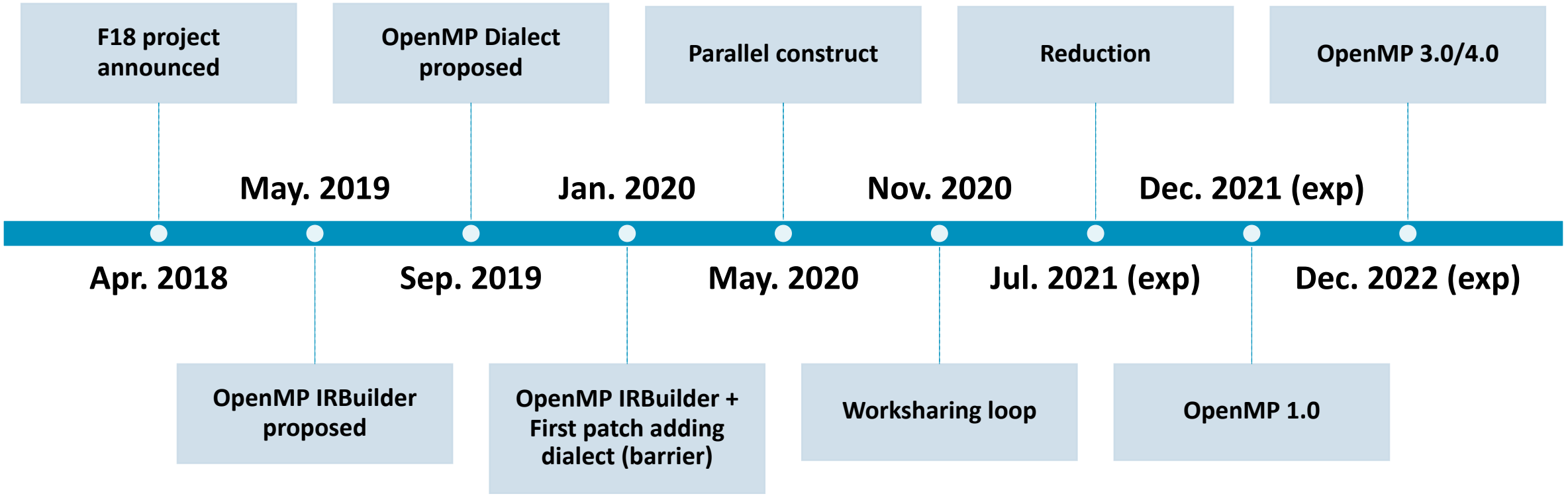
LLVM IR generation

- FIR + OpenMP + native MLIR dialects are converted to OpenMP + LLVM MLIR dialect
  - Skipping the details
- OpenMP + LLVM MLIR dialect translated to LLVM IR using the OpenMP IRBuilder
  - For each OpenMP MLIR Op the function in OpenMPIRBuilder is called

LLVM IR generation

```
LogicalResult
ModuleTranslation::convertOmpOperation(Operation &opInst,
                                       llvm::IRBuilder<> &builder) {
  if (!ompBuilder) {
    ompBuilder = std::make_unique<llvm::OpenMPIRBuilder>(*llvmModule);
    ompBuilder->initialize();
  }
  return llvm::TypeSwitch<Operation *, LogicalResult>(&opInst)
      .Case([&](omp::BarrierOp) {
        ompBuilder->CreateBarrier(builder.saveIP(), llvm::omp::OMPD_barrier);
        return success();
      })
      .Case([&](omp::TaskwaitOp) {
        ompBuilder->CreateTaskwait(builder.saveIP());
        return success();
      })
      …
```

Flang OpenMP: Events/Schedule



arm

Status

- Implementation proceeding vertically
  - Pick a construct : implement parsing, sema, mlir, lowering
  - Initially picked some widely used constructs : barrier, parallel, worksharing-loop
  - Now dual prioritization
    - OpenMP version
    - Applications
- Next target is OpenMP 1.0
  - Mostly Complete: Parallel, Do, Master, Critical, Barrier, Flush, Ordered
  - In progress: Privatisation, Sections, Single, Atomic, Reduction
  - Not Started: Threadprivate, Copyin

How to contribute?

- Open-source: Welcome to contribute
  - Contributors
    - AMD, Arm, BSC, Nvidia, Huawei, US Labs (ANL, BNL, LANL, ORNL)
- Project Management via google docs spreadsheet
  - Separate sheets for Parsing, Semantics, OpenMP MLIR, lowerings, OpenMP IRBuilder
  - Currently has entries as per OpenMP 5.0
  - https://docs.google.com/spreadsheets/d/1FvHPuSkGbl4mQZRAwCIndvQx9dQboffiD-xD0oqxgU0/edit#gid=0
- Bi-weekly meeting on Thursday (4pm UK time)
  - https://docs.google.com/document/d/1yA-MeJf6RYY-ZXpdol0t7YoDoqtwAyBhFLr5thu5pFI/edit

Thank You
Danke
Gracias
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

MLIR

- Customizable Framework for creating SSA based IRs
- Progressive Lowering
  - Maintain higher level of abstractions
  - Framework to support lowering
- Common Utilities for IR
  - Printer
  - Parser
  - Verifier
- Interfaces for defining transformations so that all dialects benefit
- Location tracking

Flang parse tree with OpenMP

Fortran source

```
program mn
...
!$omp flush(arr)
...
end
```

Flang Parse tree

Program -> ProgramUnit -> MainProgram
| ProgramStmt -> Name = 'mn'
| SpecificationPart
| | ...
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> OpenMPConstruct -> OpenMPStandaloneConstruct -> OpenMPFlushConstruct
| | | Verbatim
| | | OmpObjectList -> OmpObject -> Designator -> DataRef -> Name = 'arr'
| | ...
| EndProgramStmt ->

MLIR: Operation Definition

- Declaratively define OpenMP operations
  - Uses tablegen
- Can define the input and output operands
- Whether operations have regions inside them
- Provides generic printers and parsers for operations
- Simple example of barrier operation in the next slide

MLIR: Customized Op Definition

- Sometimes custom printers and parsers are required
- This helps to define operations in a domain specific way
- OpenMP clauses are best defined as in a directive
- Clauses can have a variable number of arguments
- Definition of parallel operation in the next slide
  - Clauses are modeled as arguments
  - Arguments are operands or attributes (constants)
  - Most OpenMP clauses are optional
  - OpenMP clauses can have a variable number of elements (like variables)

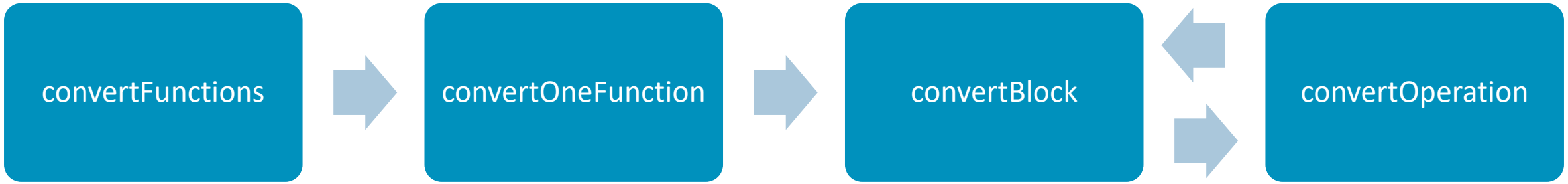
Lowering to OpenMP dialect

- Happens along with FIR lowering
- Lowering code in flang/lib/Lower/Bridge.cpp
  - Calls code in flang/lib/Lower/OpenMP.cpp

```
void Fortran::lower::genOpenMPConstruct(
                        Fortran::lower::AbstractConverter &,
                        Fortran::lower::pft::Evaluation &,
                        const Fortran::parser::OpenMPConstruct &)
```

Lowering to LLVM IR

- LLVM dialect in MLIR contains a list of functions
- Each function has a list of blocks
- Blocks have a list of operations
- OpenMP operations can have blocks inside



Lowering to LLVM IR

```
LogicalResult
ModuleTranslation::convertOmpOperation(Operation &opInst,
                                       llvm::IRBuilder<> &builder) {
  if (!ompBuilder) {
    ompBuilder = std::make_unique<llvm::OpenMPIRBuilder>(*llvmModule);
    ompBuilder->initialize();
  }
  return llvm::TypeSwitch<Operation *, LogicalResult>(&opInst)
      .Case([&](omp::BarrierOp) {
        ompBuilder->CreateBarrier(builder.saveIP(), llvm::omp::OMPD_barrier);
        return success();
      })
      .Case([&](omp::TaskwaitOp) {
        ompBuilder->CreateTaskwait(builder.saveIP());
        return success();
      })
      …
```