arm

OpenMP in Flang : An Intro

OpenMP Users Monthly Telecon

Kiran Chandramohan

24 Jun 2022

Contents

- Introduction
- Highlevel Flow of the Compiler
- OpenMP Dialect
- Representation of Worksharing loops, Collapse clause, Privatisation
- Schedule/Status
- Command Guide
- How to contribute?

Introduction

- Flang is the Fortran frontend of LLVM
  - Flang started off as the F18 project at Nvidia in collaboration with US DoE
  - It became part of LLVM on April 9, 2020
  - 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 new frameworks like MLIR defining Compiler Intermediate Representations
- Available from the llvm github repository
  - https://github.com/llvm/llvm-project/tree/main/flang

Introduction - OpenMP

- Support for OpenMP is important in HPC
- Plan to support the latest standards
  - Latest is OpenMP 5.2
- Classic Flang has partial support for OpenMP 4.5
  - Priority to get to this point of support
- Started with non-target constructs
- OpenACC is also important
  - Both OpenMP and OpenACC in Flang have similar flow
- Also shares code with Clang

Introduction - Sharing Code - Clang

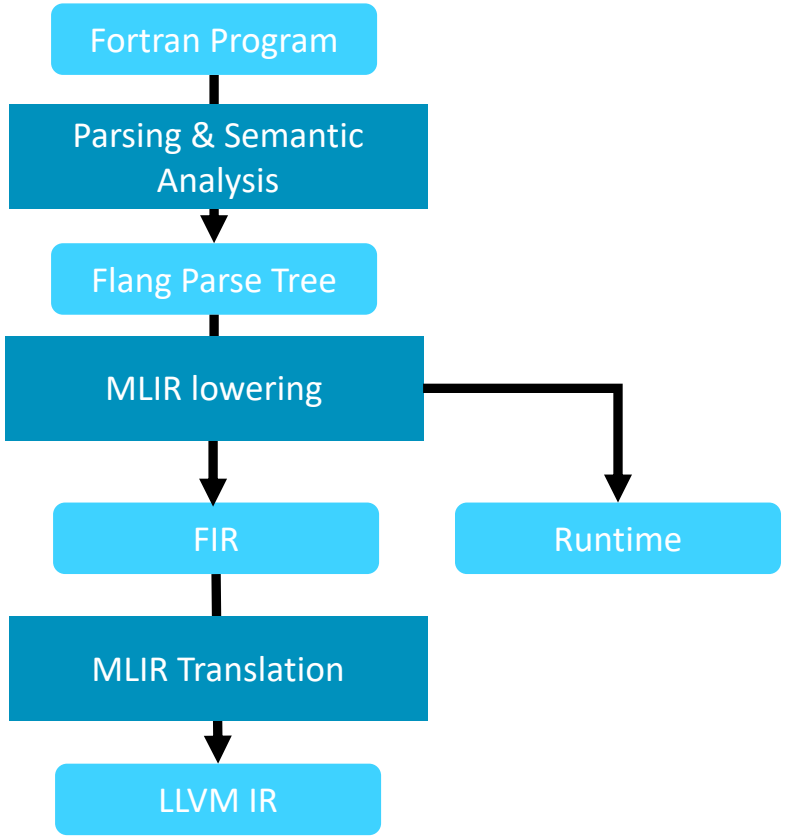
- LLVM has the Clang frontend for C/C++
- Clang has support for OpenMP
- Avoid redundant code and information about OpenMP standard
  - A single file captures information about the clauses in Constructs
    - E.g Usage: For performing semantic checks
  - Share code that generates LLVM code for OpenMP constructs, calls to the OpenMP runtime, Outlining etc
    - OpenMP IRBuilder project

Introduction Sharing code - Clang

```
def OMP_Task : Directive<"task"> {
  let allowedClauses = [
    VersionedClause<OMPC_Private>,
    VersionedClause<OMPC_FirstPrivate>,
    VersionedClause<OMPC_Shared>,
    VersionedClause<OMPC_Untied>,
    VersionedClause<OMPC_Mergeable>,
    VersionedClause<OMPC_Depend>,
    VersionedClause<OMPC_InReduction>,
    VersionedClause<OMPC_Allocate>,
    VersionedClause<OMPC_Detach, 50>,
    VersionedClause<OMPC_Affinity, 50>
  ];
  let allowedOnceClauses = [
    VersionedClause<OMPC_Default>,
    VersionedClause<OMPC_If>,
    VersionedClause<OMPC_Final>,
    VersionedClause<OMPC_Priority>
  ];
}
```

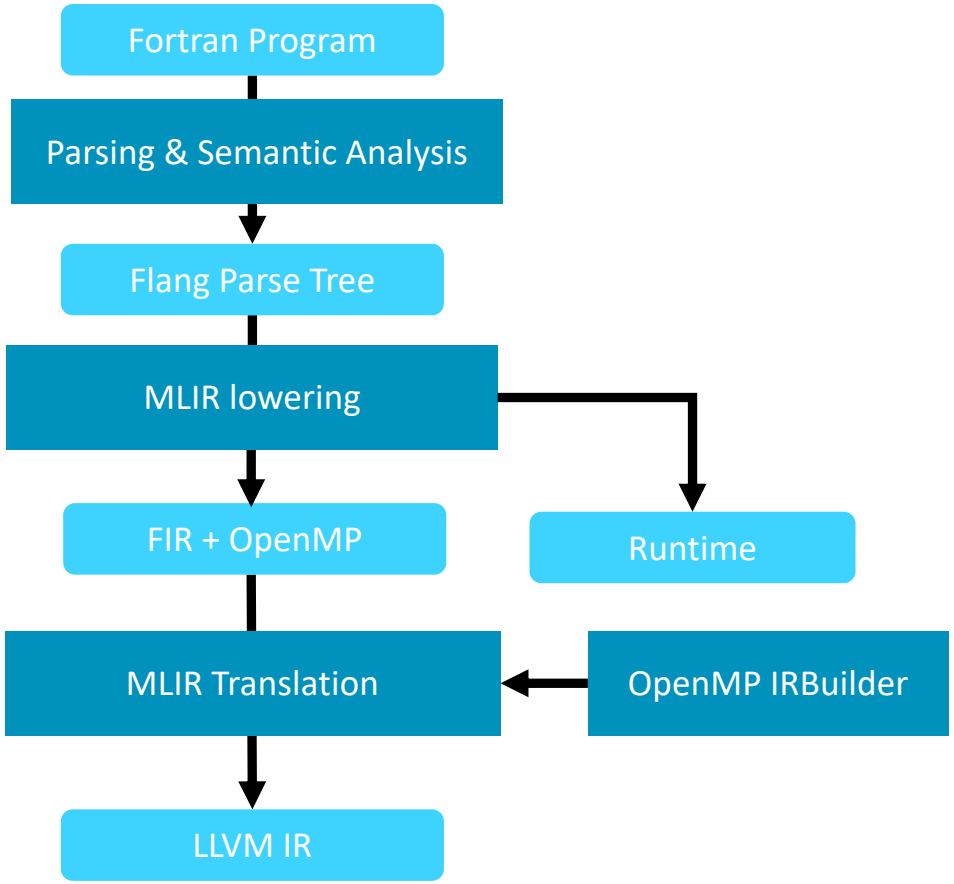
- Entry from the file (OMP.td) llvm/include/llvm/Frontend/OpenMP/OMP.td
- What clauses are allowed
- What clauses are allowed starting from version N (5.0 here)
- What clauses are only allowed once?

Flang: High Level Flow



- Traditional Compiler Flow
  - Takes in source program (Fortran)
  - Generates LLVM IR
- Difference with Clang
  - Clang lowers from AST to LLVM IR
  - Has a high-level IR : Fortran IR (FIR)
- Uses MLIR infrastructure for FIR

Flang: OpenMP High Level Flow



- Flang parse-tree augmented to represent OpenMP
- Semantic checks augmented to check OpenMP standard spec
- Parse-tree is lowered to a mix of FIR + OpenMP + other native MLIR dialects
- LLVM IR generated from this mix using the OpenMP IRBuilder
- Two major components
  - OpenMP Dialect : Spend some time
  - OpenMP IRBuilder : Opaque Box

arm

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 + OpenMP

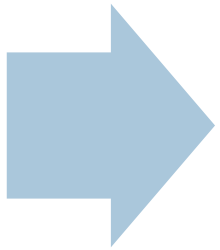
```
func @_QQmain() {
  %0 = fir.alloca f32 {bindc_name = "a", uniq_name = "_QEa"}
  %1 = fir.alloca f32 {bindc_name = "b", uniq_name = "_QEb"}
  %2 = fir.alloca f32 {bindc_name = "c", uniq_name = "_QEc"}
  omp.parallel {
    %3 = fir.load %0 : !fir.ref<f32>
    %4 = fir.load %1 : !fir.ref<f32>
    %5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

Example : OpenMP High Level Flow

MLIR: FIR + OpenMP dialect

```
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
}
```

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 Dialect in MLIR

- MLIR is a generic framework for building IRs
  - Can declaratively write definition of operations
  - Generates parsers, printers, builder functions
- OpenMP dialect is a readable high-level IR
  - Models the standard
  - Not specific for Fortran
- Operations corresponding to constructs
  - Clauses represented as operands and can be specified in any order (oilist)
    - Sometimes can be operations (reduction)
- Different kinds of operations
  - Region
    - With : Parallel, Master, Worksharing loop etc
    - Without : Barrier
  - Like containers : Enclose source code : Parallel
  - Loop like : Includes the Fortran loop in the operation : Worksharing loop

OpenMP Barrier: Definition of a simple 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”;
}
```

- Definition of OpenMP Dialect, OpenMP_Op
- Definition of barrier operation instantiates an OpenMP_Op that includes the name/mnemonic (barrier)
- A summary and description for generating documentation
- An assembly format that is used to construct the printer, parser and builder for this Operation
  - Simple Operation: No inputs/outputs
  - Format just includes the name
  - A dictionary of opaque attributes can also be added

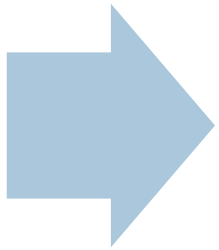
Representation of OpenMP Worksharing Loop

```
def WsLoopOp : OpenMP_Op<"wsloop", […,
                         AllTypesMatch<["lowerBound", "upperBound", "step"]>]> {
 …
 let arguments = (ins Variadic<IntLikeType>:$lowerBound,
            Variadic<IntLikeType>:$upperBound,
            Variadic<IntLikeType>:$step,
            UnitAttr:$nowait,
            …
            UnitAttr:$inclusive);
 let regions = (region AnyRegion:$region);

 …
 let assemblyFormat = [{
   oilist(…
         |`collapse` `(` $collapse_val `)`
         |`nowait` $nowait
         | …
   ) `for` custom<WsLoopControl>($region, $lowerBound, $upperBound, $step,
                                 type($step), $inclusive) attr-dict

 }];
}
```

Representation of OpenMP Worksharing Loop

Fortran + OpenMP source

```
!$omp do
do i = 1, a

  …

end do
!$omp end do nowait
```



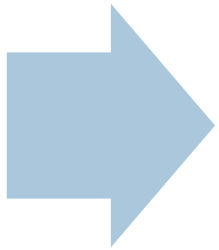
OpenMP MLIR

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

…

 }
```

Representation of Collapse

Fortran + OpenMP

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



FIR + OpenMP

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

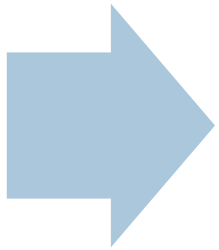
Representation of Privatisation

- Not all OpenMP details are represented in the dialect
- Privatisation is handled while lowering from parse-tree to MLIR
  - Longer term plan is to cover this in the OpenMP dialect
- Privatisation creates copies of the variables
- Copies can be allocated on the stack
- Examples of Private and Firstprivate in the next two slides

Privatisation – Private Clause

Fortran + OpenMP

```
integer :: x
!$omp parallel private(x)
!$omp end parallel
```



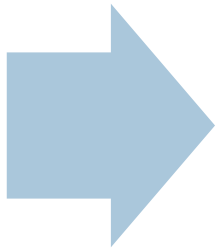
FIR + OpenMP

```
%0 = fir.alloca i32 {bindc_name = "x", uniq_name = "_QFEx"}
omp.parallel   {
  %1 = fir.alloca i32 {bindc_name = "x", pinned, uniq_name =
"_QFEx"}
  omp.terminator
}
```

Privatisation – Firstprivate Clause

Fortran + OpenMP

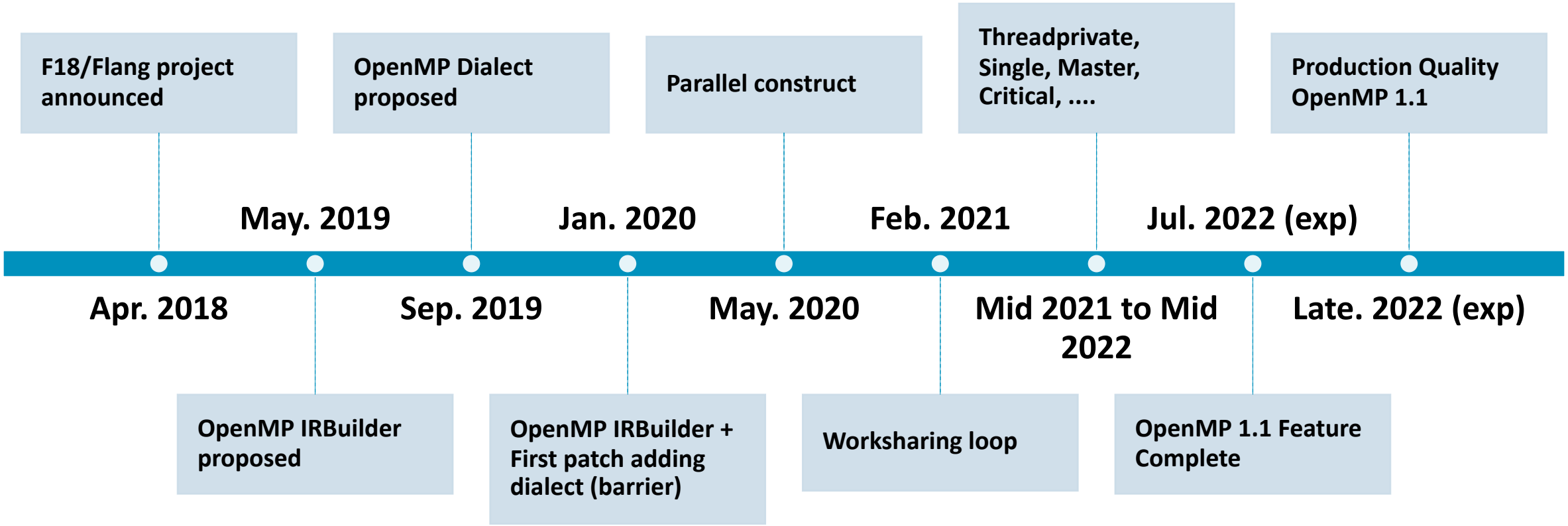
```
integer :: x
!$omp parallel firstprivate(x)
!$omp end parallel
```



FIR + OpenMP

```
%0 = fir.alloca i32 {bindc_name = "x", uniq_name = "_QFEx"}
omp.parallel   {
  %1 = fir.alloca i32 {bindc_name = "x", pinned, uniq_name =
"_QFEx"}
  %2 = fir.load %0 : !fir.ref<i32>
  fir.store %2 to %1 : !fir.ref<i32>
  omp.barrier
  omp.terminator
}
```

Events/Schedule



arm

Status - Standards

- Reaching close to OpenMP 1.1 completion
  - Includes a lot of the basic constructs
- A few non OpenMP 1.1 constructs are also in progress
  - OpenMP 2.5 has the workshare construct specifically for Fortran : Not started
  - OpenMP 3.0
    - Task construct : In progress
  - OpenMP 4.0
    - Simd, Taskgroup, Target : In progress

	Completed	Mostly complete	In Progress
OpenMP 1.1	Parallel, Do, Single, Critical, Sections, Master, Barrier, Flush	Atomic, Copyin, Privatisation	Reduction, lastprivate
OpenMP 3.0/4.0		Task, Taskgroup	Task, Taskgroup, Simd, Target, Target Data Map, Cancel, Cancellation

Status - Applications

- Tested with a proxy application – SNAP
  - https://github.com/lanl/SNAP
  - Around 60 OpenMP directives
  - Only uses Fortran 95 and OpenMP 1.1
  - Exposed a few issues with
    - OpenMP regions containing unstructured code (cycle, goto)
    - Privatising index variables
    - Reprivatising variables
- More testing ongoing with Spec OMP 2012 and Spec 2017 speed

Command Guide

- Shown some intermediate representations of the compiler
  - This slide gives the commands needed to generate these
- flang-new is the name of the driver
  - Use -fopenmp flag to enable OpenMP processing
  - Use -fc1 for generating intermediate representations
- Emit parse-tree
  - ./bin/flang-new -fc1 -fdebug-dump-parse-tree -fopenmp file.f90
- Perform parsing and semantic checks
  - ./bin/flang-new -fsyntax-only -fopenmp file.f90
- Generate FIR + OpenMP
  - ./bin/flang-new -fc1 -emit-fir -fopenmp file.f90
- Generate LLVM IR
  - ./bin/flang-new -S -emit-llvm -fopenmp file.f90
- Flang compiler is not yet fully open for users
  - Use `-flang-experimental-exec` flag to generate executables
  - ./bin/flang-new -flang-experimental-exec -fopenmp file.f90

How to contribute?

- Open-source: Welcome to contribute
  - Contributors
    - AMD, Arm, BSC, Nvidia, Huawei, US Labs (ANL, BNL, LANL, ORNL), couple of hobby developers
  - LLVM contribution process
    - https://llvm.org/docs/Contributing.html#how-to-submit-a-patch
- 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