OpenMP Dialect

FC + MLIR

Overview

- Dialect to represent OpenMP constructs
- FC AST nodes are converted to OpenMP dialect operations.
- `omp.parallel`, `omp.single`, `omp.do`, `omp.master` and `omp.parallel_do` are currently supported
- Data elements to be mapped are represented as arguments to the operations.
- Nesting of constructs are supported for parallel do, etc.
- Parallel regions are outlined to functions at LLVM Dialect level

Implementation Details

- AST Nodes Definition : **include/AST/StmtOpenMP.h**
- Parser support : **`lib/parse/ParseOpenMP.cpp`**
- MLIR code generation: **lib/codegen/CGOpenMP.cpp**
- OpenMP dialect operations: **include/dialect/OpenMP/OpenMPOps.td**
- LLVM Dialect lowering: **lib/transforms/OpenMPLowering.cpp**
- OpenMP unit tests**: test/openmp/**

OpenMP Dialect Operations

(include/dialect/OpenMP/)

Parallel construct

```
a = 10
b = 20
!$omp parallel
print *, "hellofrom", a, b
!$omp end parallel
```

```
omp.parallel(%2, %3)  {
   %5 = fc.constant_string("hellofrom")
   %6 = fc.load %2 {name = "a"} : i32
   %7 = fc.load %3 {name = "b"} : i32
   fc.print %5, %6, %7
}
```

Parallel Do Construct

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

```
omp.parallel_do(%5 = %6, %7, %8)(%4, %2, %3) {
  %9 = fc.load %5 {name = "i"} : i32
  %10 = index_cast %9 : i32 to index
  %11 = fc.load %2[%10] {name = "a", range_info =
                         #fc.subscript_range< 0 >} : i32
  %12 = fc.load %5 {name = "i"} : i32
  %13 = index_cast %12 : i32 to index
  %14 = fc.load %3[%13] {name = "b", range_info =
                          #fc.subscript_range< 0 >} : i32
  %15 = addi %11, %14 : i32
  %16 = fc.load %5 {name = "i"} : i32
  %17 = index_cast %16 : i32 to index
  fc.store %15, %4[%17] {name = "c", range_info =
          #fc.subscript_range< 0 >} : !fc.ref<!fc.array<1:10 x
    i32>>

} enddo
```

Do construct

- omp do construct is represented using OMP::OmpDoOp.
- Upper bound, lower bound and step are represented as arguments to OmpDoOp operation.

```
!$omp do
 do i = 1, 10
   print *,  "Hello omp do "
 enddo
!$omp end do
```

```
omp.do %arg0 = %c1_i32, %c10_i32, %c1_i32
                                  {construct_name = ""}  {
  %2 = fc.constant_string("Hello omp do ")
  fc.print %2 {arg_info = #fc.is_string< 0 >}
} enddo
```

Single construct

```
integer  :: n

n = 10

!$omp single

 print *, "Hello from omp", n

!$omp end single
```

```
%c10_i32 = constant 10 : i32

omp.single()  {

  %3 = fc.constant_string("Hello from omp")

  fc.print %3, %c10_i32 {arg_info = #fc.is_string< 0 >}

}
```

Master construct

```
!$omp master
print *, "Hello from master"
!$omp end master
print *, "From outsitde"
```

```
omp.master()  {
  %3 = fc.constant_string("Hello from master")
  fc.print %3 {arg_info = #fc.is_string< 0 >}
}
%2 = fc.constant_string("From ou tsitde")
fc.print %2 {arg_info = #fc.is_string< 0 >}
```

Nesting of constructs (1) : Parallel do

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

```
omp.parallel_do(%5 = %11, %12, %13)(%6, %7, %4, %2, %3) {
  ...
  omp.parallel_do(%6 = %15, %16, %17)(%7, %4, %5, %2, %3) {
    ...
    fc.store %28, %4[%30, %32] {name = "c"}
  } enddo
} endo
```

Nesting of constructs(2)

```
!$omp parallel
  print *, "Hello from omp"
  $omp single
  print *, "Hello from single thread"
  !$omp end single
!$omp end parallel
```

```
omp.parallel()  {
 %3 = fc.constant_string("Hello from omp"):fc.array<0:14 x i8>
 fc.print %3 {arg_info = #fc.is_string< 0 >}
 omp.single()  {
   %4 = fc.constant_string("Hello from single thread")
   fc.print %4 {arg_info = #fc.is_string< 0 >}
 }
}
```

Matmul Example

```
!$omp parallel do
 do i = 1,n
   do j = 1,n
     c(i,j) = 0
     do k = 1,n
       c(i,j) = c(i,j) + a(i,k) * b(k,j)
     enddo
   enddo
 enddo
!$omp end parallel do
```

```
omp.parallel_do(%3 = %9, %10, %11)(%4, %2, %5, %0, %1) {
  ...
  fc.do %arg0 = %c1_i32_9, %c300_i32_10, %c1_i32_11   {
    ...
    fc.do %arg1 = %c1_i32_13, %c300_i32_14, %c1_i32_15 {
       ...
       fc.store %26, %2[%28, %arg0] {name = "c"}
    } enddo
  } enddo
} enddo
```

OpenMP Dialect Lowering

(lib/transforms/OpenMPLowering.cpp)

Dialect Lowering

- OpenMP dialect lowering is scheduled after all the other dialects are lowered to LLVM.
- Operations are lowered directly to LLVM IR.
- Outlines OpenMP parallel regions to `LLVM::FuncOp`
- Outlined function is called using OpenMP runtime library function

  `__kmpc_fork_call(..)`

ParallelOp Lowering

```
omp.parallel(%2)  {
  %2 = fc.load %2 {name = "a"} : i32
  fc.print %2  {arg_info =
              #fc.is_string< >}
 }
```

```
define void @outlined.(i32* noalias nocapture readnone %0, i32* noalias
       nocapture readnone %1, i32* noalias nocapture readonly %2) {
  %4 = load i32, i32* %2, align 4, !alias.scope !0, !noalias !2
  tail call void (i32, ...) @__fc_runtime_print(i32 2, ,i32 3, i32 %4)
  ret void
}
define i32 @foo() local_unnamed_addr {
  %1 = alloca i32, align 4
  store i32 10, i32* %1, align 4
  call void ({ i32, i32, i32, i32, i8* }*, i32, i8*, ...)
              @__kmpc_fork_call({ i32, i32, i32, i32, i8* }* nonnull
             @ident.global, i32 1, i8* bitcast (void (i32*, i32*, i32*)*
              @outlined. to i8*), i32* nonnull %1)
  ret i32 0
}
```

SingleOp Lowering

```
%c10_i32 = constant 10 : i32
omp.single()  {
  %3 = fc.constant_string("Hello from omp")
  fc.print %3, %c10_i32
}
```

```
%1 = call i32 @__kmpc_global_thread_num({ i32, i32, i32, i32, i8* }* nonnull @ident.global)
%2 = call i32 @__kmpc_single({ i32, i32, i32, i32, i8* }* nonnull @ident.global, i32 %1)
 %3 = icmp eq i32 %2, 0
 br i1 %3, label %5, label %4
4:                                                ; preds = %0
 call void (i32, ...) @__fc_runtime_print(i32 4, i32 9, i8* getelementptr inbounds ([16 x i8], [16 x i8]* @str_const_2, i64 0, i64 0), i32 3, i32 10)
 call void @__kmpc_end_single({ i32, i32, i32, i32, i8* }* nonnull @ident.global, i32 %1)
 br label %5
5:                                                ; preds = %0, %4
 call void @__kmpc_barrier({ i32, i32, i32, i32, i8* }* nonnull @ident.global, i32 %1)
```

MasterOp Lowering

```
omp.master()  {
 %3 = fc.constant_string("Hello from
    master"):
 fc.print %3
}
```

```
  %1 = call i32 @__kmpc_global_thread_num({ i32, i32, i32,
                      i32, i8* }* nonnull @ident.global)
  %2 = call i32 @__kmpc_master({ i32, i32, i32, i32, i8* }*
                              nonnull @ident.global, i32 %1)
  %3 = icmp eq i32 %2, 0
  br i1 %3, label %5, label %4
  4:        ; preds = %0
  call void (i32, ...) @__fc_runtime_print(i32 2, i32 9,
i8*
 getelementptr inbounds ([19 x i8], [19 x i8]*
@str_const_2,
                                                  i64 0, i64 0))
 call void @__kmpc_end_master({ i32, i32, i32, i32, i8* }*
                              nonnull @ident.global, i32 %1)
```

OmpDoOp Lowering

```
omp.do %arg0 = %c1_i32, %c10_i32, %c1_i32 {
  %2 = fc.constant_string("Hello omp do ")
  fc.print %2
} enddo
```

```
%5 = call i32 @__kmpc_global_thread_num({ i32, i32, i32, i32, i8* }*
        nonnull @ident.global)
call void ({ i32, i32, i32, i32, i8* }*, i32, i32, i32*, i32*, i32*, i32*, i32,
        i32, ...) @__kmpc_for_static_init_4({ i32, i32, i32, i32, i8* }*
        nonnull @ident.global, i32 %5, i32 34, i32* nonnull %4, i32*
        nonnull %1, i32* nonnull %2, i32* nonnull %3, i32 1, i32 1)
.....
 br i1 %9, label %._crit_edge, label %.lr.ph
.lr.ph:                                   ; preds = %0, %.lr.ph
.... ; Loop body
._crit_edge:                              ; preds = %.lr.ph, %0
call void @__kmpc_for_static_fini({ i32, i32, i32, i32, i8* }* nonnull
        @ident.global, i32 %5)
call void @__kmpc_barrier({ i32, i32, i32, i32, i8* }* nonnull
        @ident.global, i32 %
```

Thank You