Advanced

T⚙pics

in

C⚙mpilers

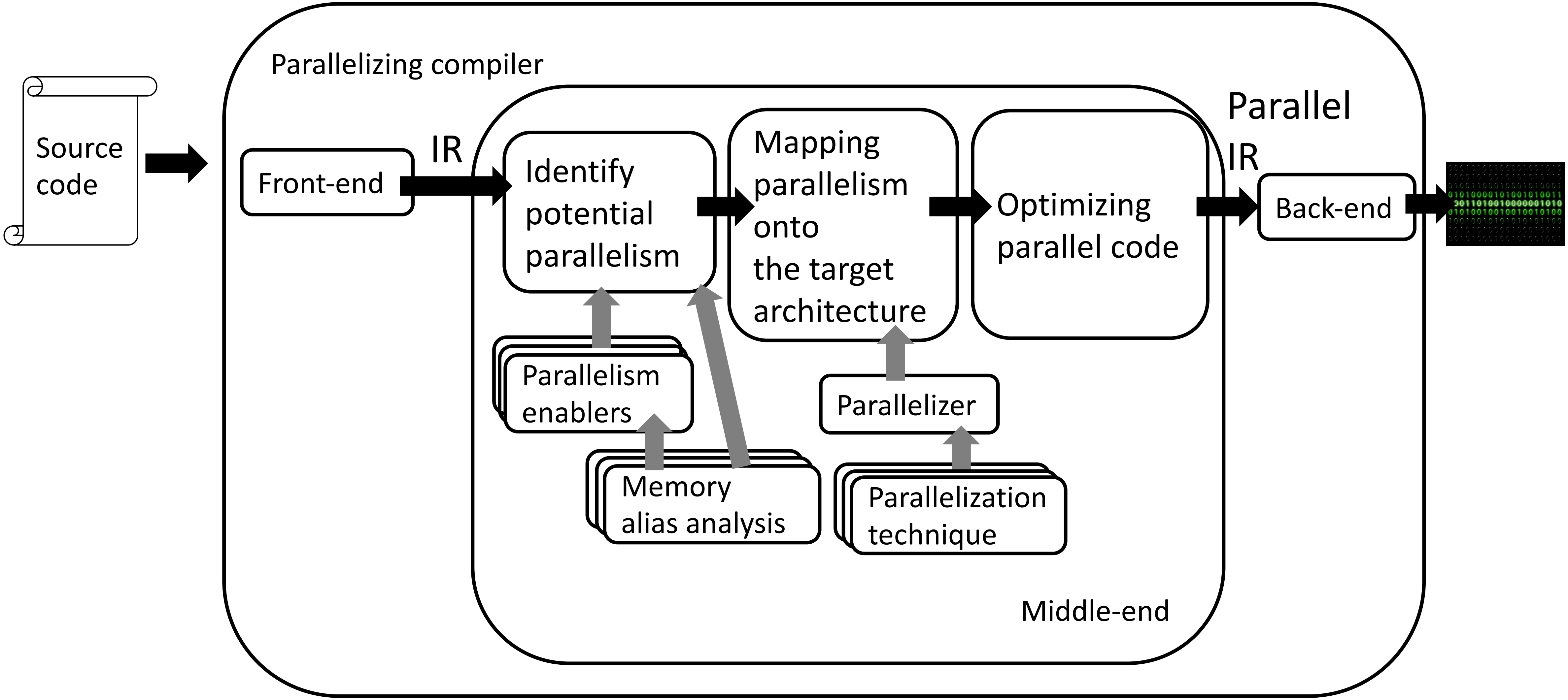
Parallelizer

Simone Campanoni
simone.campanoni@northwestern.edu

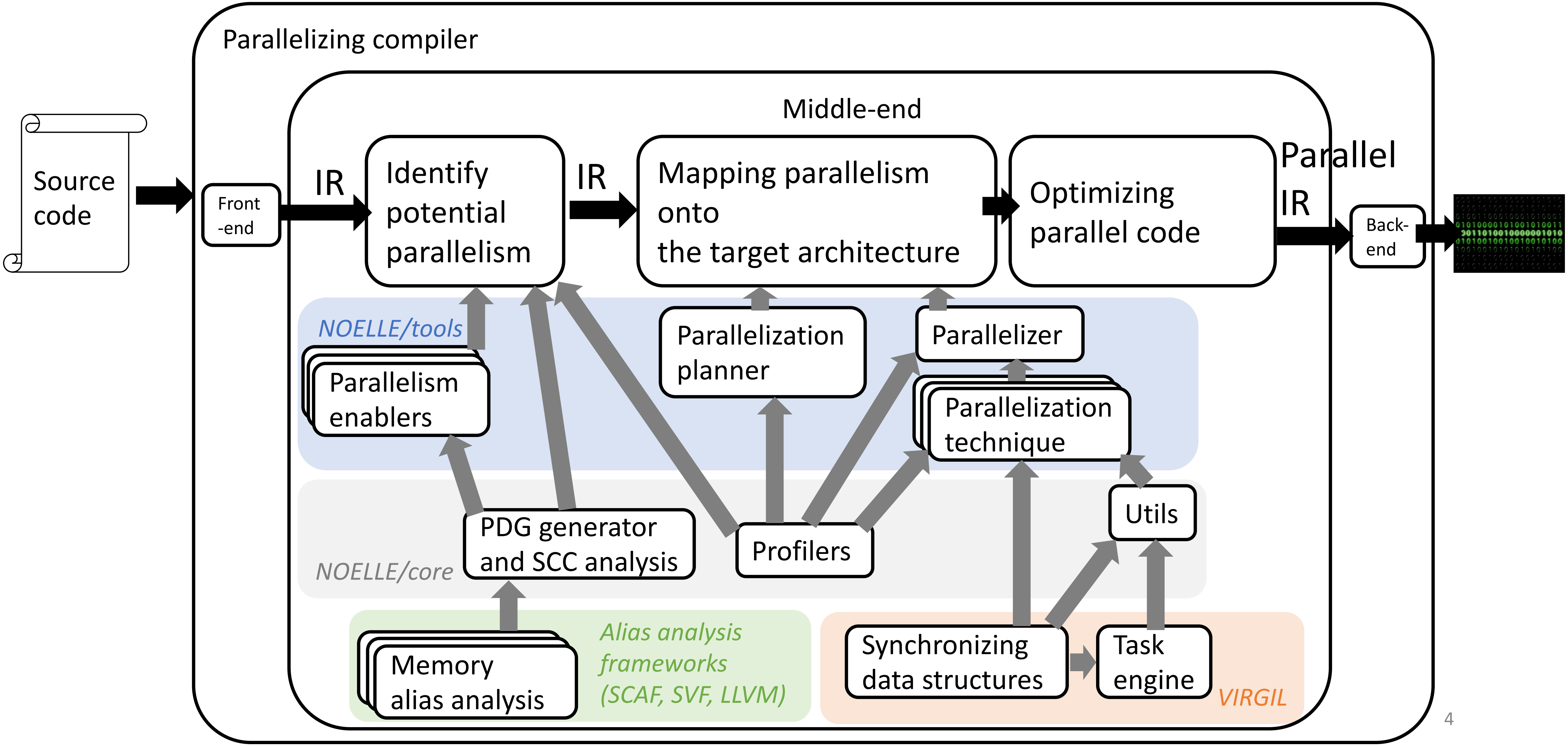
Outline

- The parallelizing compiler built upon NOELLE
- Compilation pipeline
- Debugging

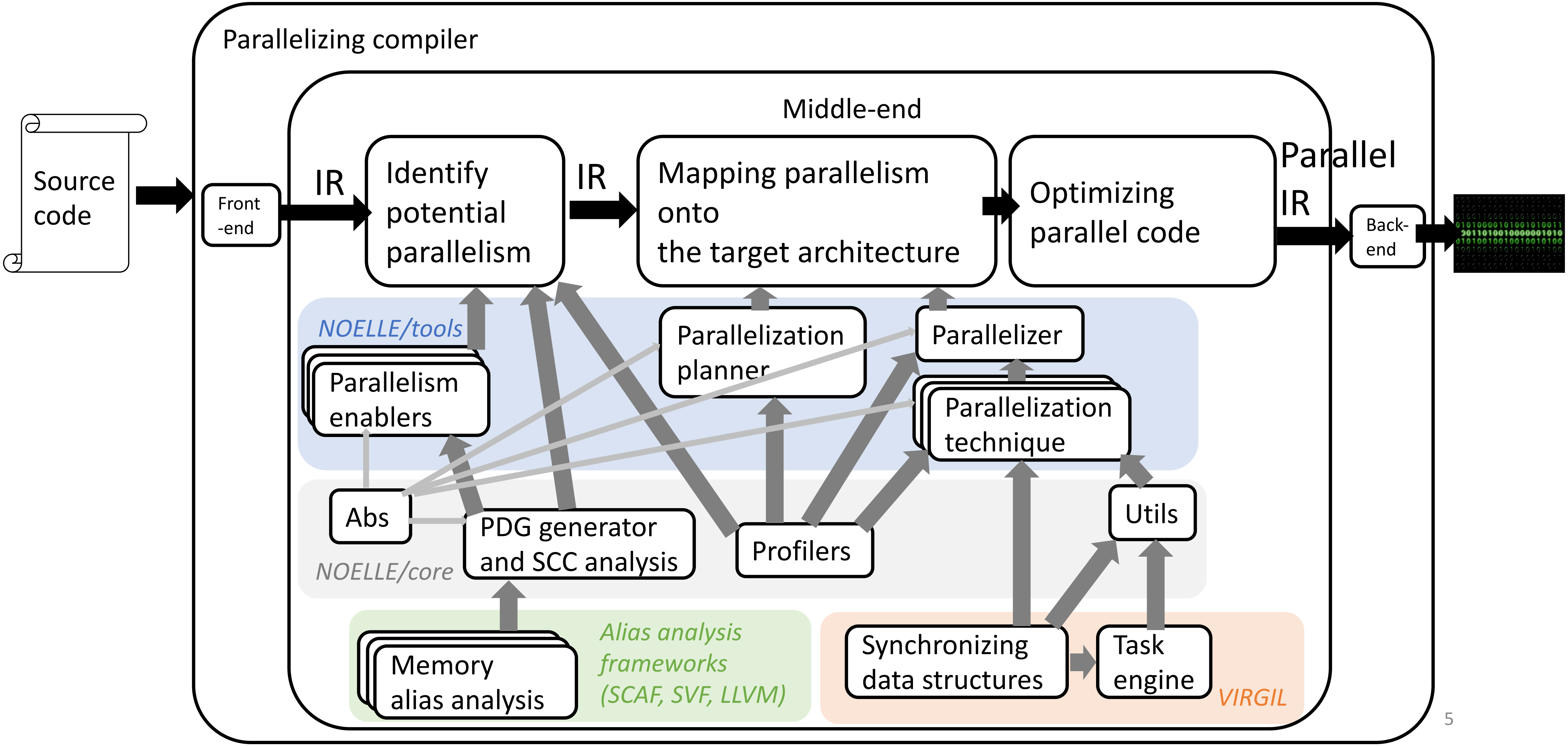
A typical parallelizing compiler



The parallelizing compiler built upon NOELLE



The parallelizing compiler built upon NOELLE



Outline

- The parallelizing compiler built upon NOELLE

- Compilation pipeline

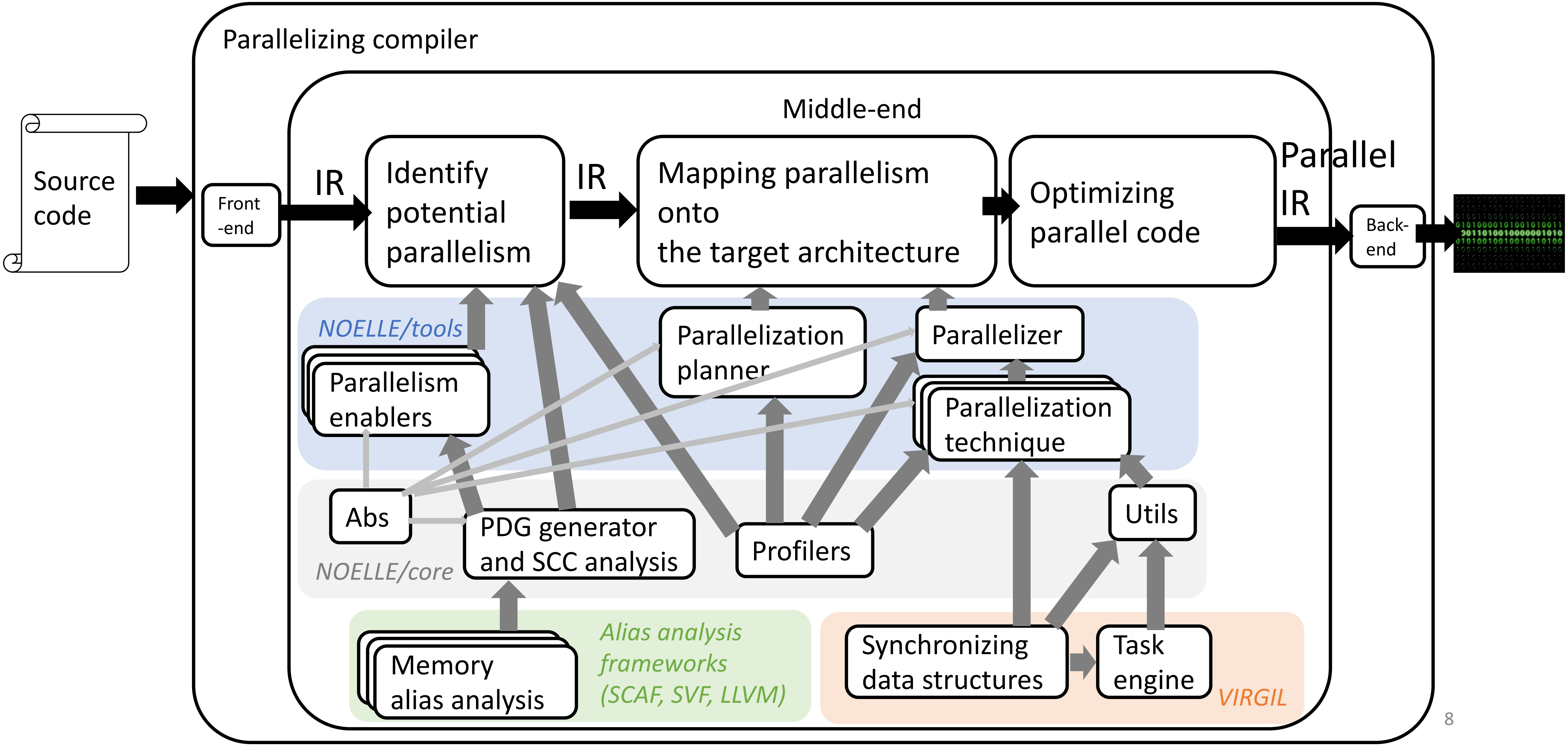
- Debugging

Compilation pipeline

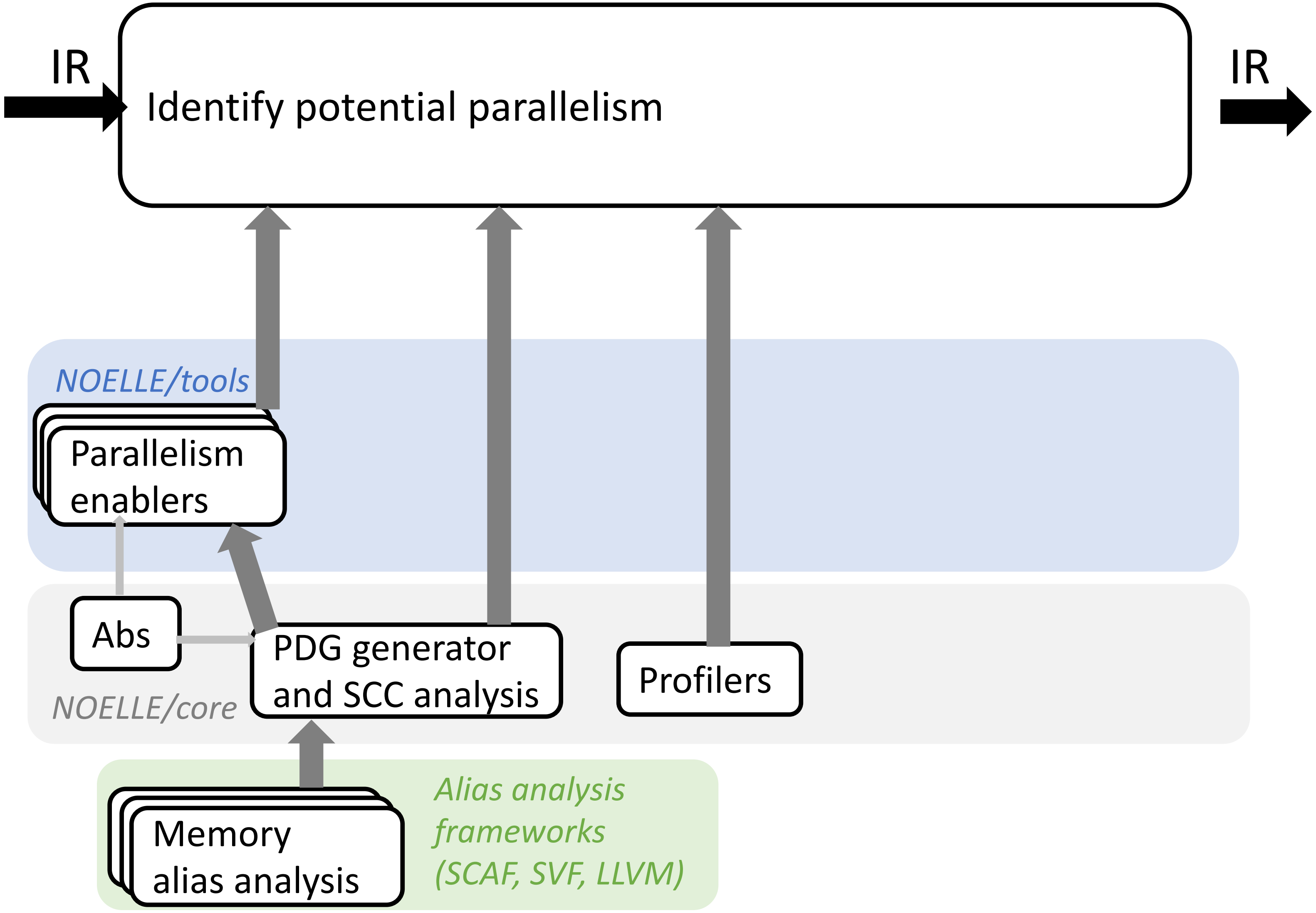
- Let’s assume test.cpp is the whole program

  (otherwise, if multiple source files exist, then
  use gclang if you run commands manually
  or use NOELLEGym to automate everything)

The parallelizing compiler built upon NOELLE

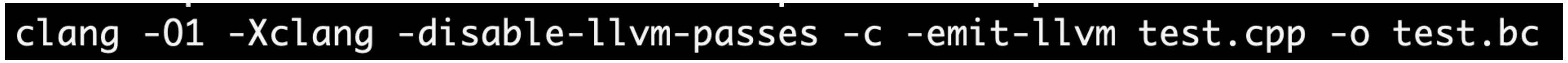


The parallelizing compiler built upon NOELLE


IR
Identify potential parallelism
IR
NOELLE/tools
Parallelism enablers
Abs
PDG generator and SCC analysis
Profilers
NOELLE/core
Memory alias analysis
Alias analysis frameworks (SCAF, SVF, LLVM)

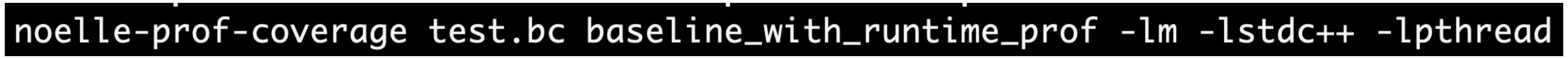
Compilation pipeline

- Let’s assume test.cpp is the whole program

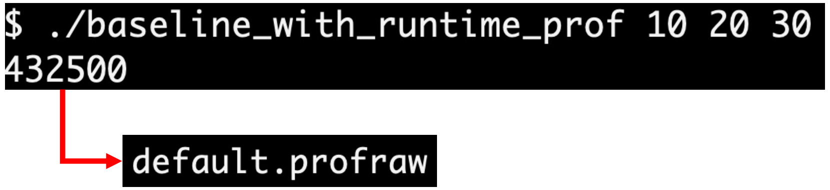


```
clang -O1 -Xclang -disable-llvm-passes -c -emit-llvm test.cpp -o test.bc
```

- Now we need to profile the code to identify hot code

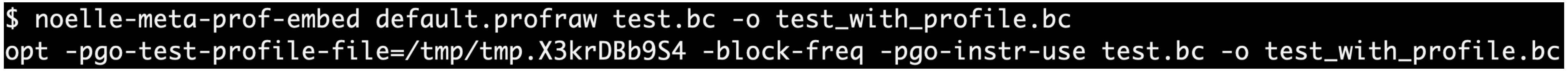


```
noelle-prof-coverage test.bc baseline_with_runtime_prof -lm -lstdc++ -lpthread
```



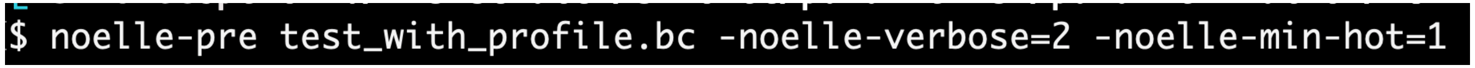
```
$ ./baseline_with_runtime_prof 10 20 30
432500
```

```
default.profraw
```



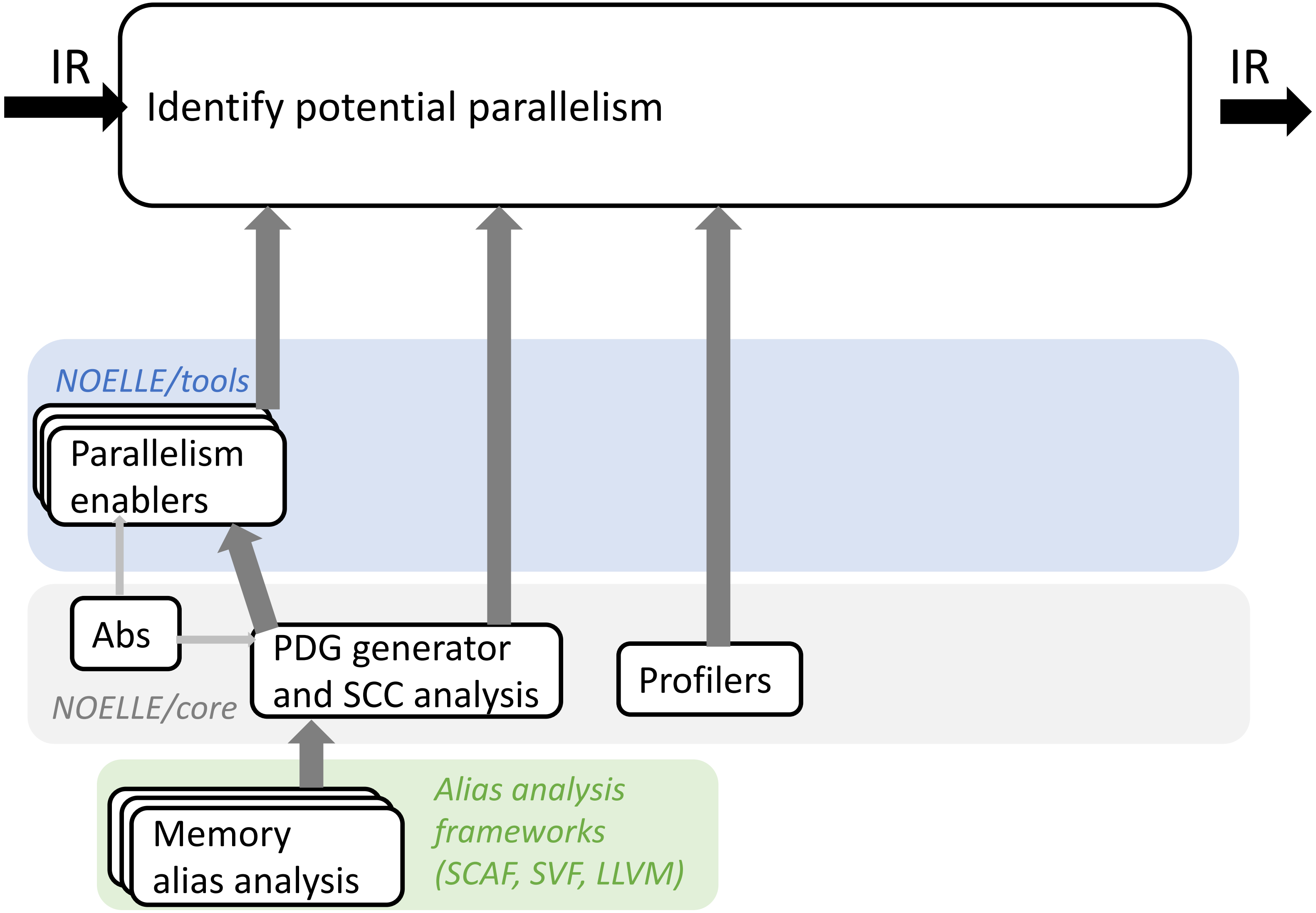
```
$ noelle-meta-prof-embed default.profraw test.bc -o test_with_profile.bc
opt -pgo-test-profile-file=/tmp/tmp.X3krDBb9S4 -block-freq -pgo-instr-use test.bc -o test_with_profile.bc
```

- Now we need to make the IR more amenable for parallelization

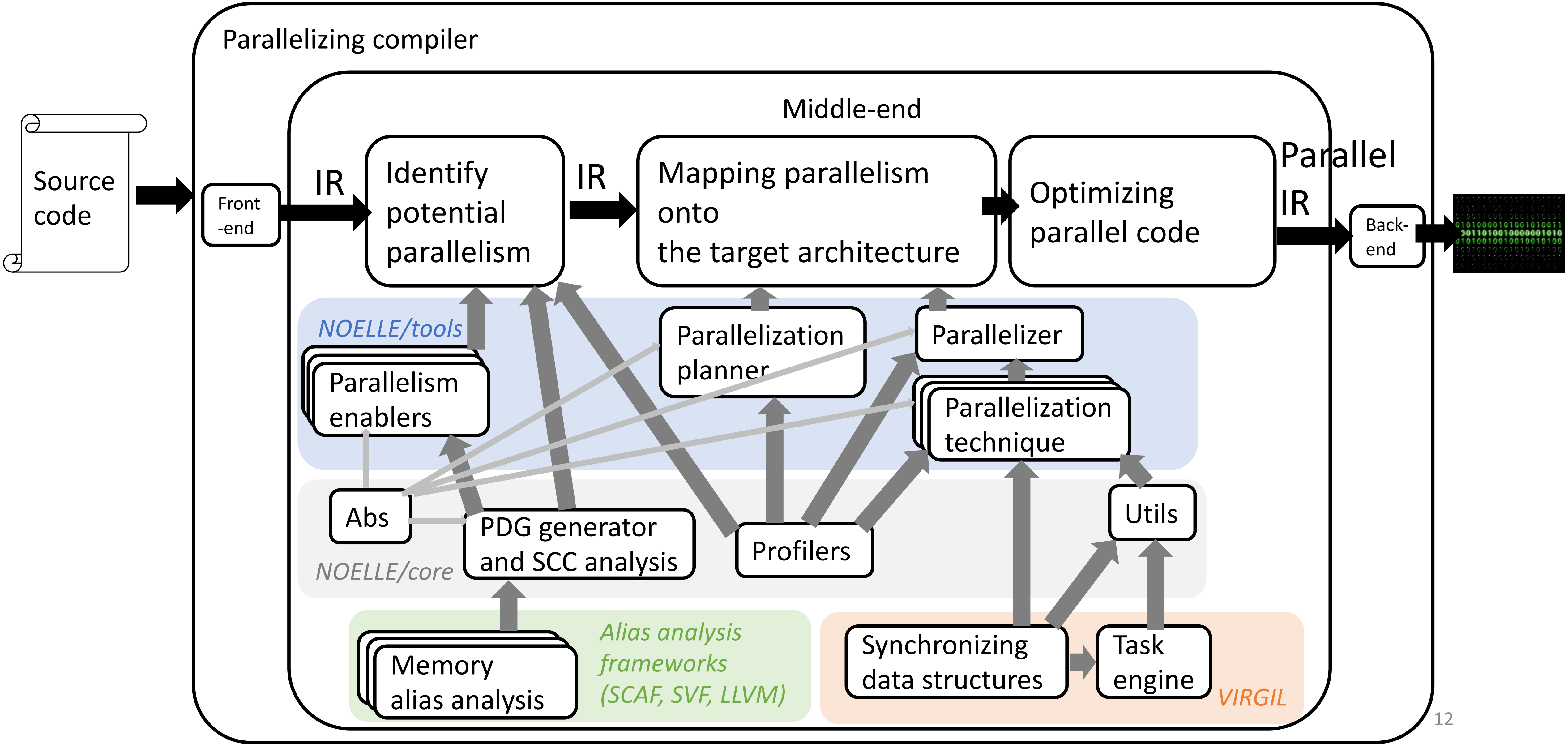


```
$ noelle-pre test_with_profile.bc -noelle-verbose=2 -noelle-min-hot=1
```

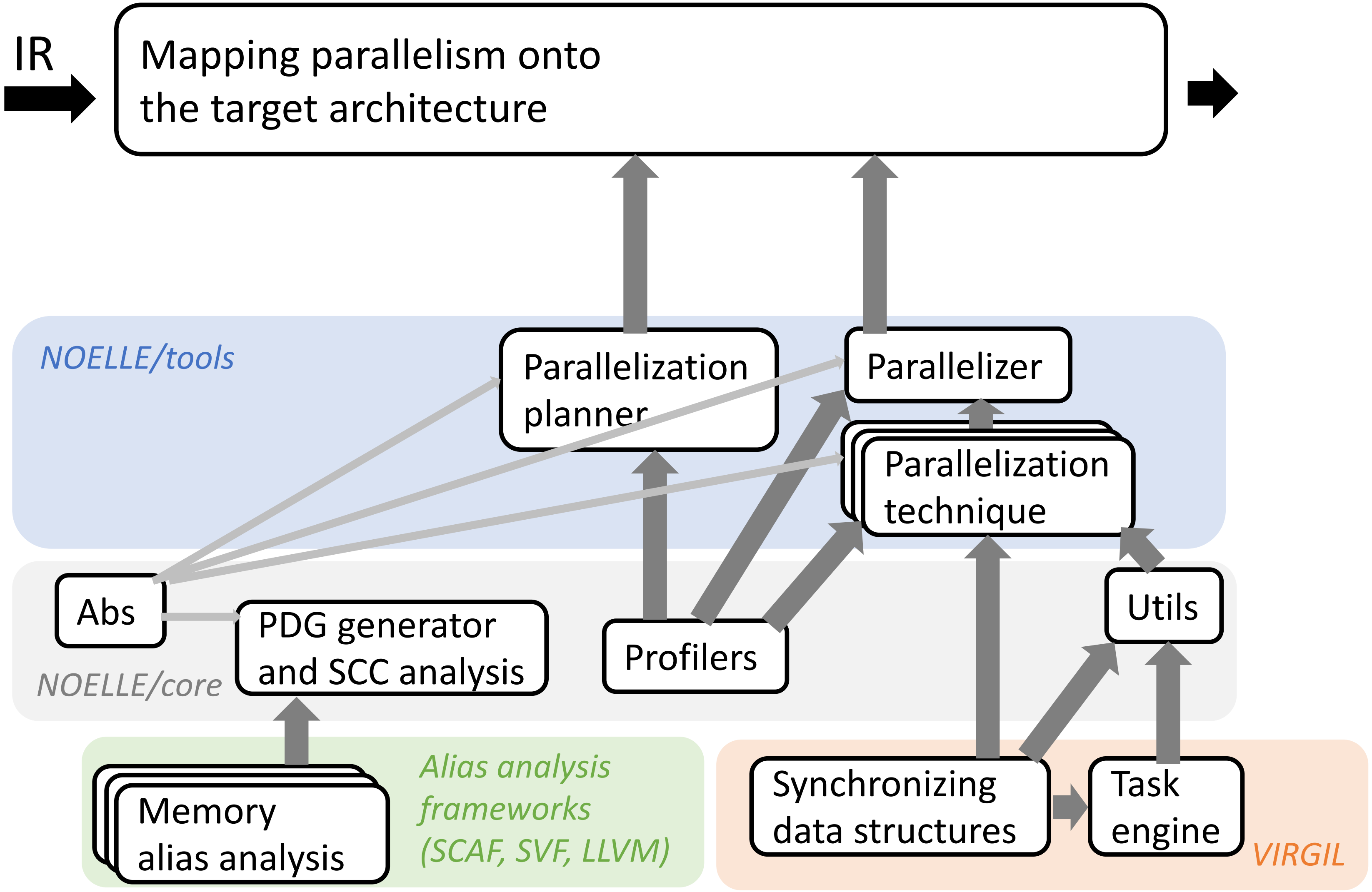
The parallelizing compiler built upon NOELLE


IR
Identify potential parallelism
IR
NOELLE/tools
Parallelism enablers
Abs
PDG generator and SCC analysis
Profilers
NOELLE/core
Memory alias analysis
Alias analysis frameworks (SCAF, SVF, LLVM)

The parallelizing compiler built upon NOELLE

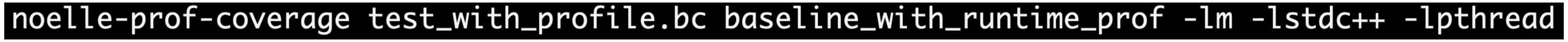


The parallelizing compiler built upon NOELLE

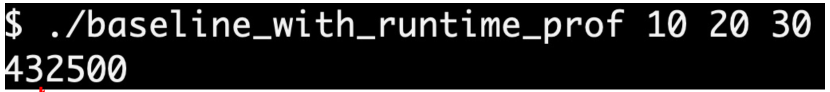

IR
Mapping parallelism onto the target architecture
NOELLE/tools
Parallelization planner
Parallelizer
Parallelization technique
Abs
PDG generator and SCC analysis
Profilers
Utils
NOELLE/core
Memory alias analysis
Alias analysis frameworks (SCAF, SVF, LLVM)
Synchronizing data structures
Task engine
VIRGIL

Compilation pipeline

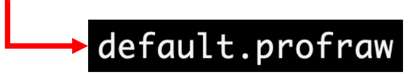
- We need to profile the code



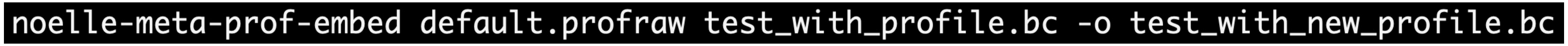
```
noelle-prof-coverage test_with_profile.bc baseline_with_runtime_prof -lm -lstdc++ -lpthread
```



```
$ ./baseline_with_runtime_prof 10 20 30
432500
```

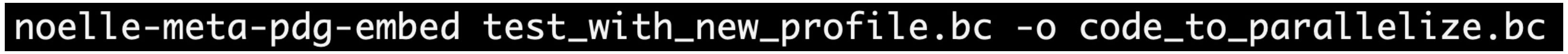


```
default.profraw
```



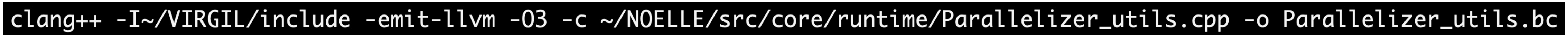
```
noelle-meta-prof-embed default.profraw test_with_profile.bc -o test_with_new_profile.bc
```

- Now we need to compute the PDG and embed it into the IR

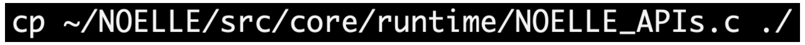


```
noelle-meta-pdg-embed test_with_new_profile.bc -o code_to_parallelize.bc
```

- Now we need to compile utilities written in C/C++ that the parallelizer will use to parallelize the code (e.g., synchronization data structures)



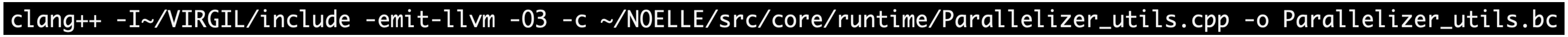
```
clang++ -I~/VIRGIL/include -emit-llvm -O3 -c ~/NOELLE/src/core/runtime/Parallelizer_utils.cpp -o Parallelizer_utils.bc
```



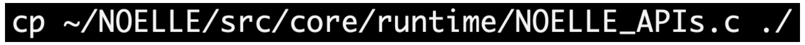
```
cp ~/NOELLE/src/core/runtime/NOELLE_APIs.c ./
```

Compilation pipeline

- Now we need to compile utilities written in C/C++ that the parallelizer will use to parallelize the code (e.g., synchronization data structures)

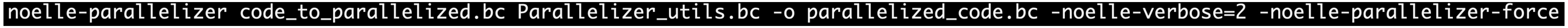


```
clang++ -I~/VIRGIL/include -emit-llvm -O3 -c ~/NOELLE/src/core/runtime/Parallelizer_utils.cpp -o Parallelizer_utils.bc
```



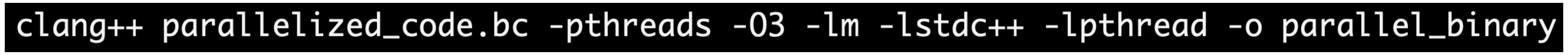
```
cp ~/NOELLE/src/core/runtime/NOELLE_APIs.c ./
```

- Now we can parallelize the IR



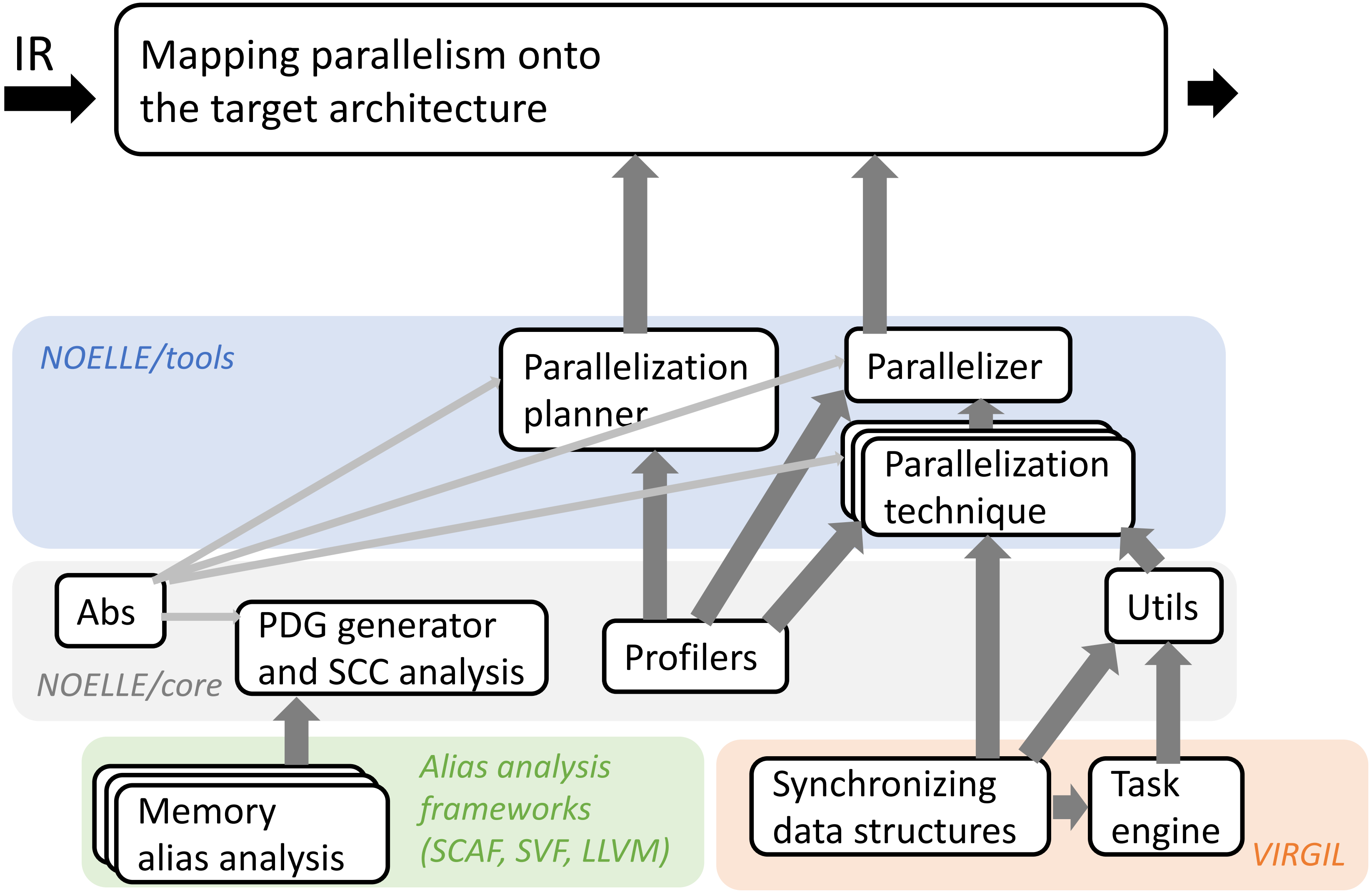
```
noelle-parallelizer code_to_parallelized.bc Parallelizer_utils.bc -o parallelized_code.bc -noelle-verbose=2 -noelle-parallelizer-force
```

- Now we can generate the parallelized binary

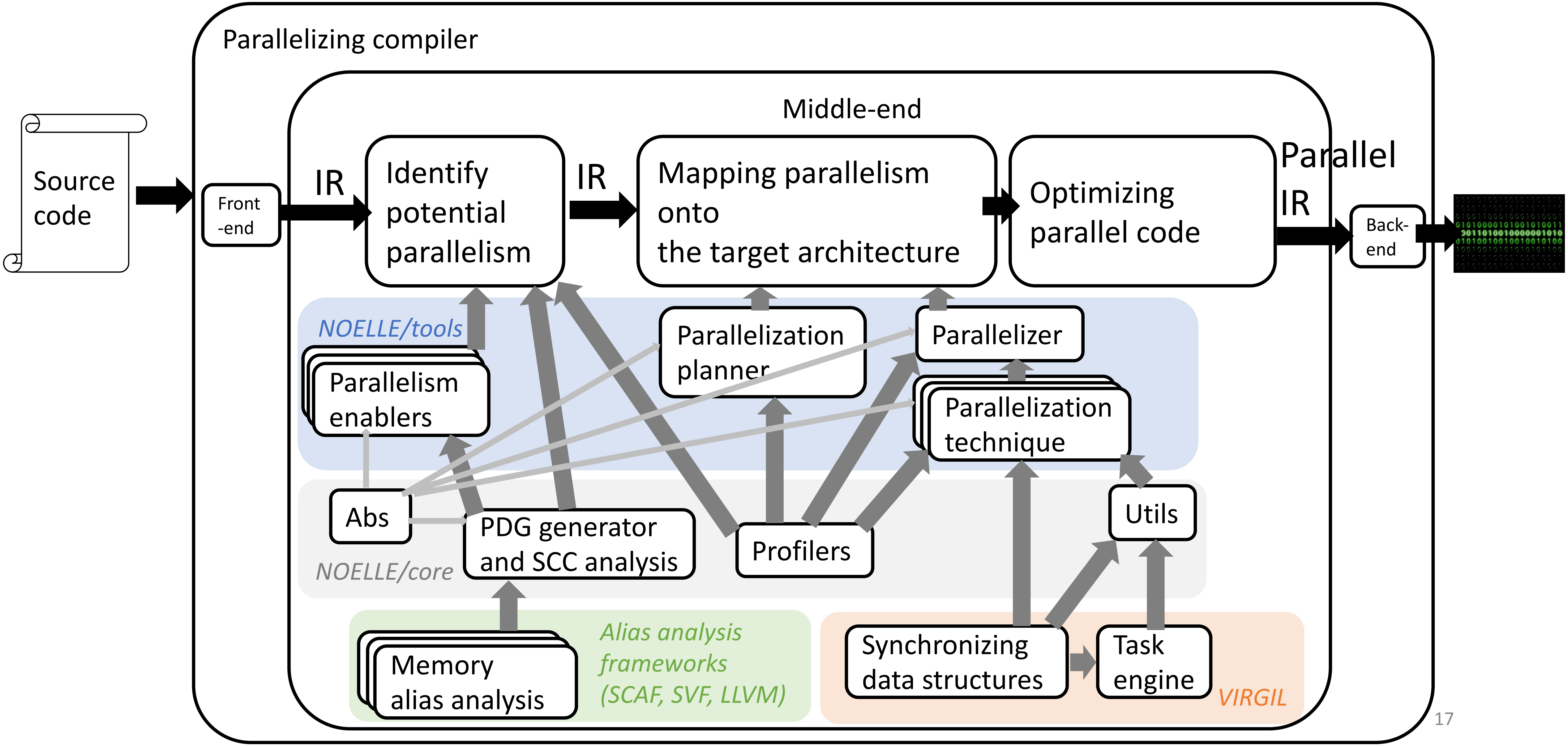


```
clang++ parallelized_code.bc -pthreads -O3 -lm -lstdc++ -lpthread -o parallel_binary
```

The parallelizing compiler built upon NOELLE


IR
Mapping parallelism onto the target architecture
NOELLE/tools
Parallelization planner
Parallelizer
Parallelization technique
Abs
PDG generator and SCC analysis
Profilers
Utils
NOELLE/core
Memory alias analysis
Alias analysis frameworks (SCAF, SVF, LLVM)
Synchronizing data structures
Task engine
VIRGIL

The parallelizing compiler built upon NOELLE



Outline

- The parallelizing compiler built upon NOELLE
- Compilation pipeline
- Debugging

Typical flow

1. The parallelizer in the master branch works,
   but you want to improve the speedup obtained by it for a given benchmark
   - Let’s assume you are using NOELLEGym

2. You extend/modify a code analysis/transformation in the parallelizing pipeline described in these slides
   - To do so, you modify something in NOELLEGym/NOELLE/src,
     and then you recompile and install NOELLE

3. You re-run the parallelizer and the new parallel binary generated doesn’t work (e.g., seg fault)

How should you debug it?

An approach to debug a loop-based parallelizing compiler

Assumption: the bug fit the common case, which is about parallelizing a given loop (independent on what other loops are parallelized)

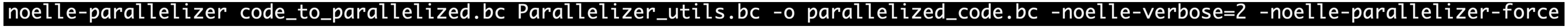
1. **Shrinking**:
   Identify a single loop that its parallelization
   (when using the new changes) leads to the bug

2. **Comparing**:
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR)
   of the parallelization between master and the changes.

3. **Correctness checking**:
   Deep analysis on the difference in parallelization that is incorrect
   (by manually checking why that parallelization aspect that differ is incorrect)

An approach to debug
a loop-based parallelizing compiler

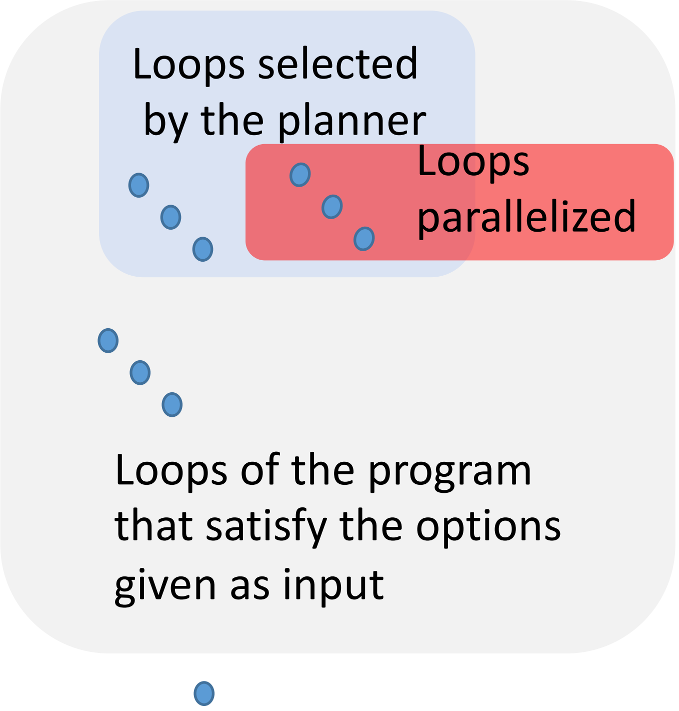
1. Shrinking

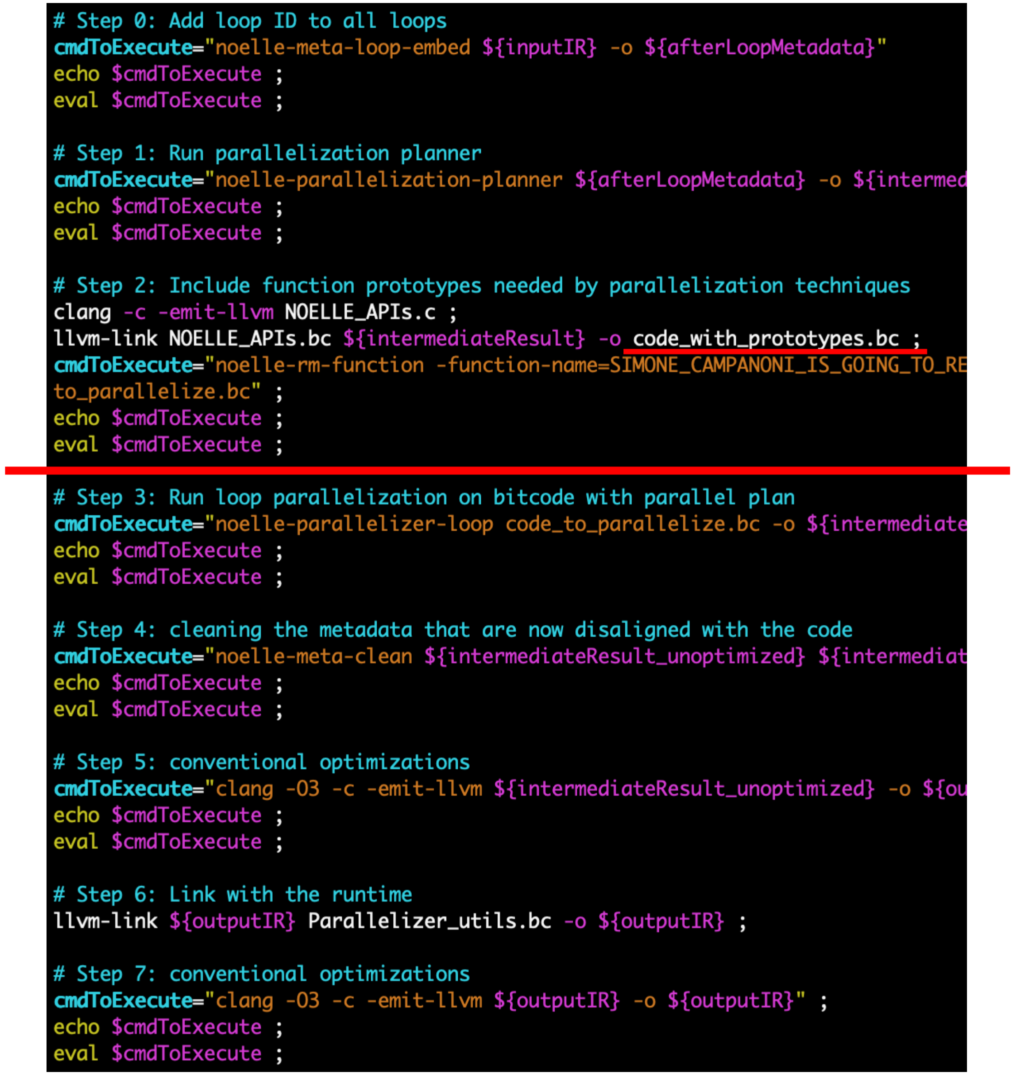


```
noelle-parallelizer code_to_parallelized.bc Parallelizer_utils.bc -o parallelized_code.bc -noelle-verbose=2 -noelle-parallelizer-force
```

An approach to debug a loop-based parallelizing compiler

1. Shrinking

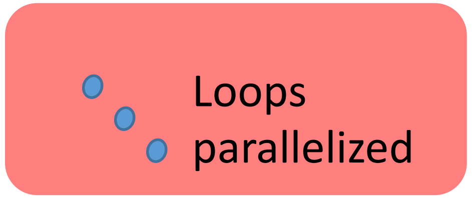

Loops selected by the planner
Loops parallelized
Loops of the program that satisfy the options given as input



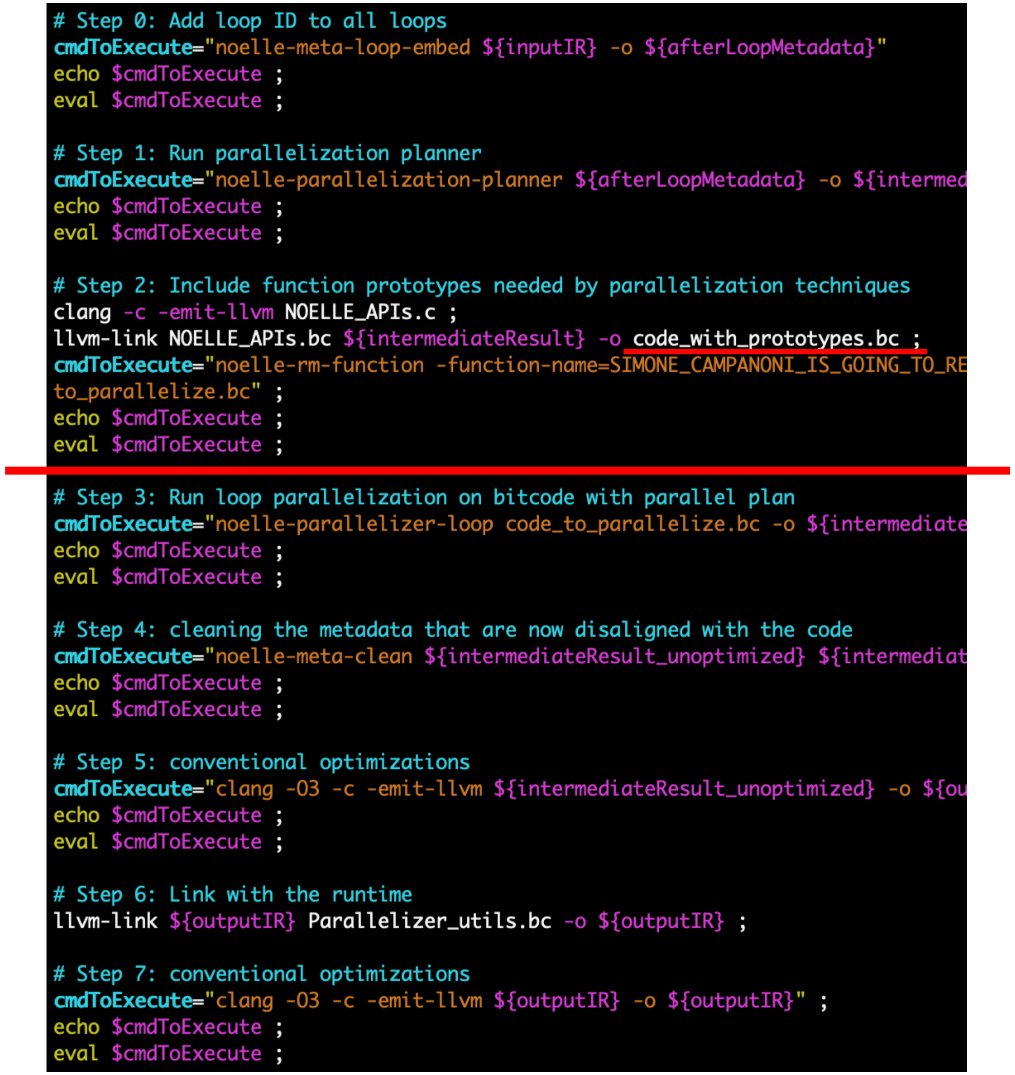
```
# Step 0: Add loop ID to all loops
cmdToExecute="noelle-meta-loop-embed ${inputIR} -o ${afterLoopMetadata}"
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 1: Run parallelization planner
cmdToExecute="noelle-parallelization-planner ${afterLoopMetadata} -o ${intermed
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 2: Include function prototypes needed by parallelization techniques
clang -c -emit-llvm NOELLE_APIs.c ;
llvm-link NOELLE_APIs.bc ${intermediateResult} -o code_with_prototypes.bc ;
cmdToExecute="noelle-rm-function -function-name=SIMONE_CAMPANONI_IS_GOING_TO_RE
to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 3: Run loop parallelization on bitcode with parallel plan
cmdToExecute="noelle-parallelizer-loop code_to_parallelize.bc -o ${intermediate
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
cmdToExecute="noelle-meta-clean ${intermediateResult_unoptimized} ${intermediat
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${ou
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link ${outputIR} Parallelizer_utils.bc -o ${outputIR} ;

# Step 7: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${outputIR} -o ${outputIR}" ;
echo $cmdToExecute ;
eval $cmdToExecute ;
```

An approach to debug a loop-based parallelizing compiler

1. **Shrinking**

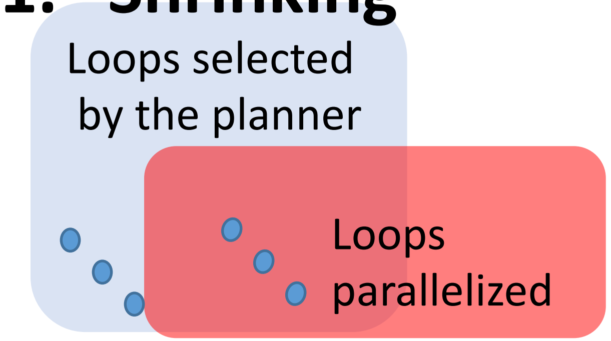

Loops parallelized

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

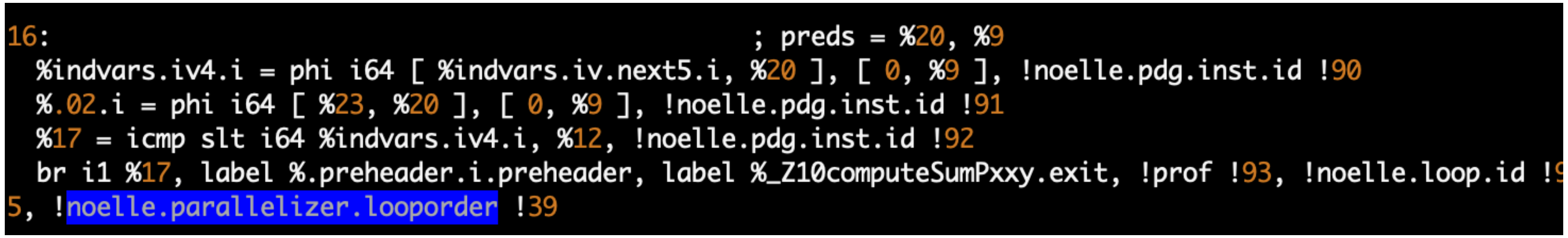

```
# Step 0: Add loop ID to all loops
cmdToExecute="noelle-meta-loop-embed ${inputIR} -o ${afterLoopMetadata}"
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 1: Run parallelization planner
cmdToExecute="noelle-parallelization-planner ${afterLoopMetadata} -o ${intermed
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 2: Include function prototypes needed by parallelization techniques
clang -c -emit-llvm NOELLE_APIs.c ;
llvm-link NOELLE_APIs.bc ${intermediateResult} -o code_with_prototypes.bc ;
cmdToExecute="noelle-rm-function -function-name=SIMONE_CAMPANONI_IS_GOING_TO_RE
to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 3: Run loop parallelization on bitcode with parallel plan
cmdToExecute="noelle-parallelizer-loop code_to_parallelize.bc -o ${intermediate
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
cmdToExecute="noelle-meta-clean ${intermediateResult_unoptimized} ${intermediat
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${ou
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link ${outputIR} Parallelizer_utils.bc -o ${outputIR} ;

# Step 7: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${outputIR} -o ${outputIR}" ;
echo $cmdToExecute ;
eval $cmdToExecute ;
```

An approach to debug a loop-based parallelizing compiler

1. **Shrinking**



$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

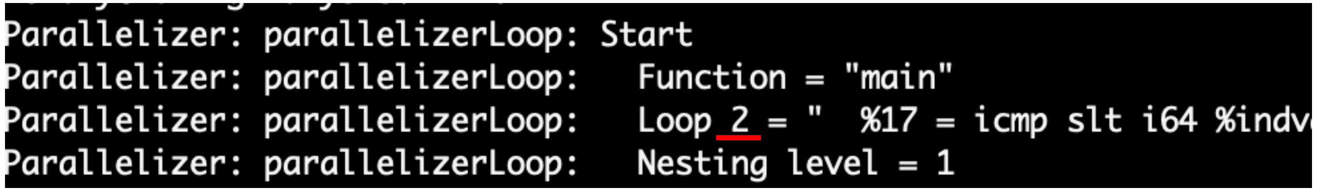


```
16:                                               ; preds = %20, %9
  %indvars.iv4.i = phi i64 [ %indvars.iv.next5.i, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !90
  %.02.i = phi i64 [ %23, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !91
  %17 = icmp slt i64 %indvars.iv4.i, %12, !noelle.pdg.inst.id !92
  br i1 %17, label %.preheader.i.preheader, label %_Z10computeSumPxxy.exit, !prof !93, !noelle.loop.id !9
5, !noelle.parallelizer.looporder !39
```

An approach to debug a loop-based parallelizing compiler

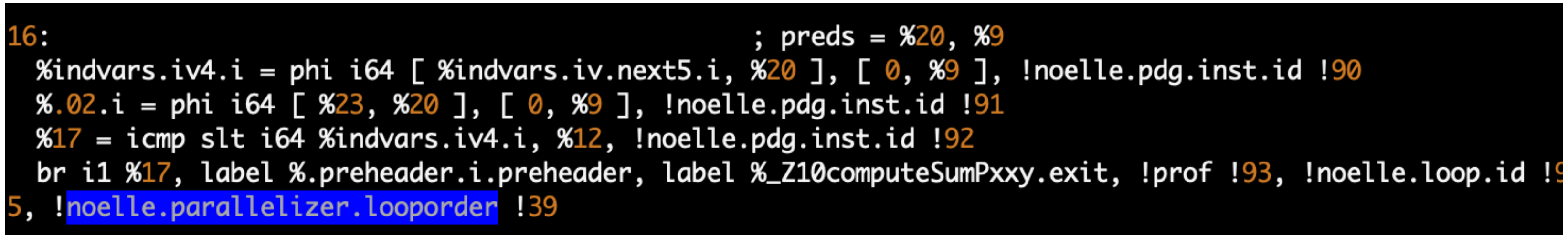
1. Shrinking

Loops parallelized



```
Parallelizer: parallelizerLoop: Start
Parallelizer: parallelizerLoop:   Function = "main"
Parallelizer: parallelizerLoop:   Loop 2 = "   %17 = icmp slt i64 %indv
Parallelizer: parallelizerLoop:   Nesting level = 1
```

```
$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
```

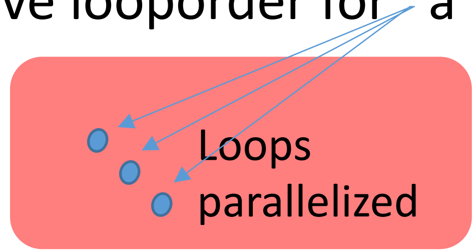


```
16:                                               ; preds = %20, %9
  %indvars.iv4.i = phi i64 [ %indvars.iv.next5.i, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !90
  %.02.i = phi i64 [ %23, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !91
  %17 = icmp slt i64 %indvars.iv4.i, %12, !noelle.pdg.inst.id !92
  br i1 %17, label %.preheader.i.preheader, label %_Z10computeSumPxxy.exit, !prof !93, !noelle.loop.id !9
5, !noelle.parallelizer.looporder !39
```

An approach to debug
a loop-based parallelizing compiler

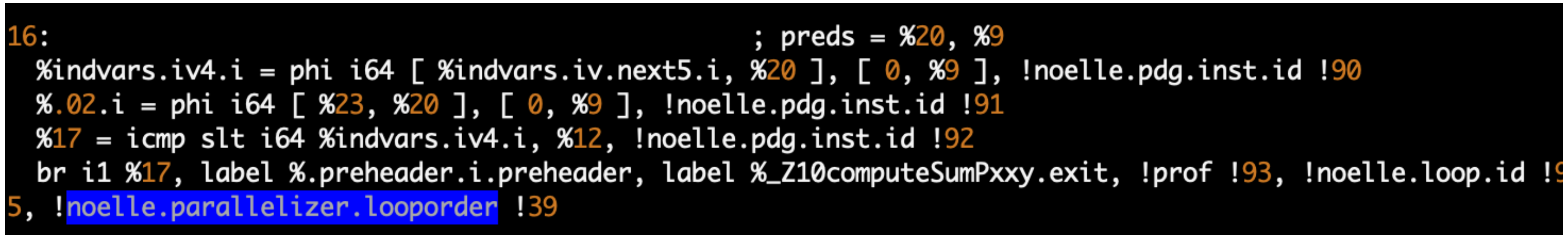
1. Shrinking

Remove looporder for a few at a times (e.g., binary search)



Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata

```
$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
```



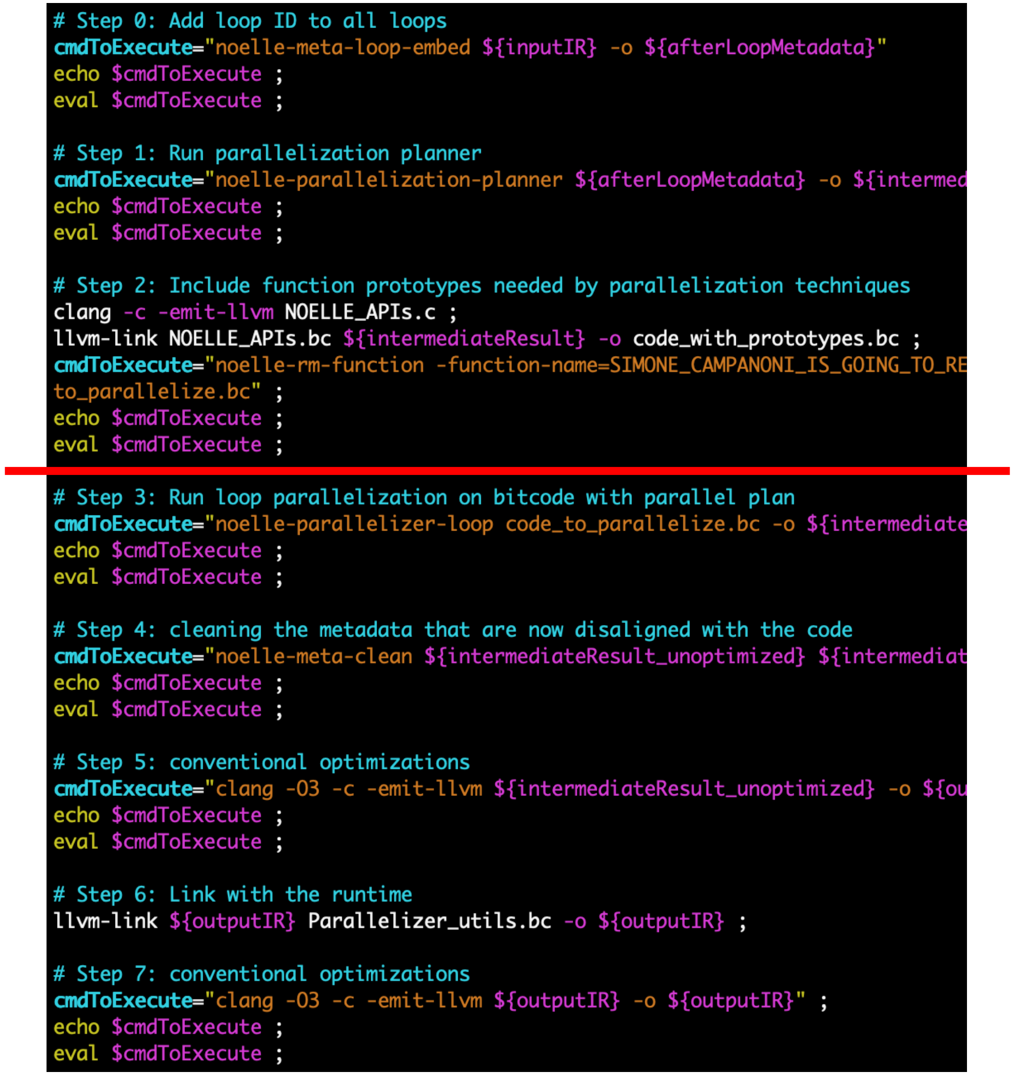
```
16:                                               ; preds = %20, %9
  %indvars.iv4.i = phi i64 [ %indvars.iv.next5.i, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !90
  %.02.i = phi i64 [ %23, %20 ], [ 0, %9 ], !noelle.pdg.inst.id !91
  %17 = icmp slt i64 %indvars.iv4.i, %12, !noelle.pdg.inst.id !92
  br i1 %17, label %.preheader.i.preheader, label %_Z10computeSumPxxy.exit, !prof !93, !noelle.loop.id !9
5, !noelle.parallelizer.looporder !39
```

An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata



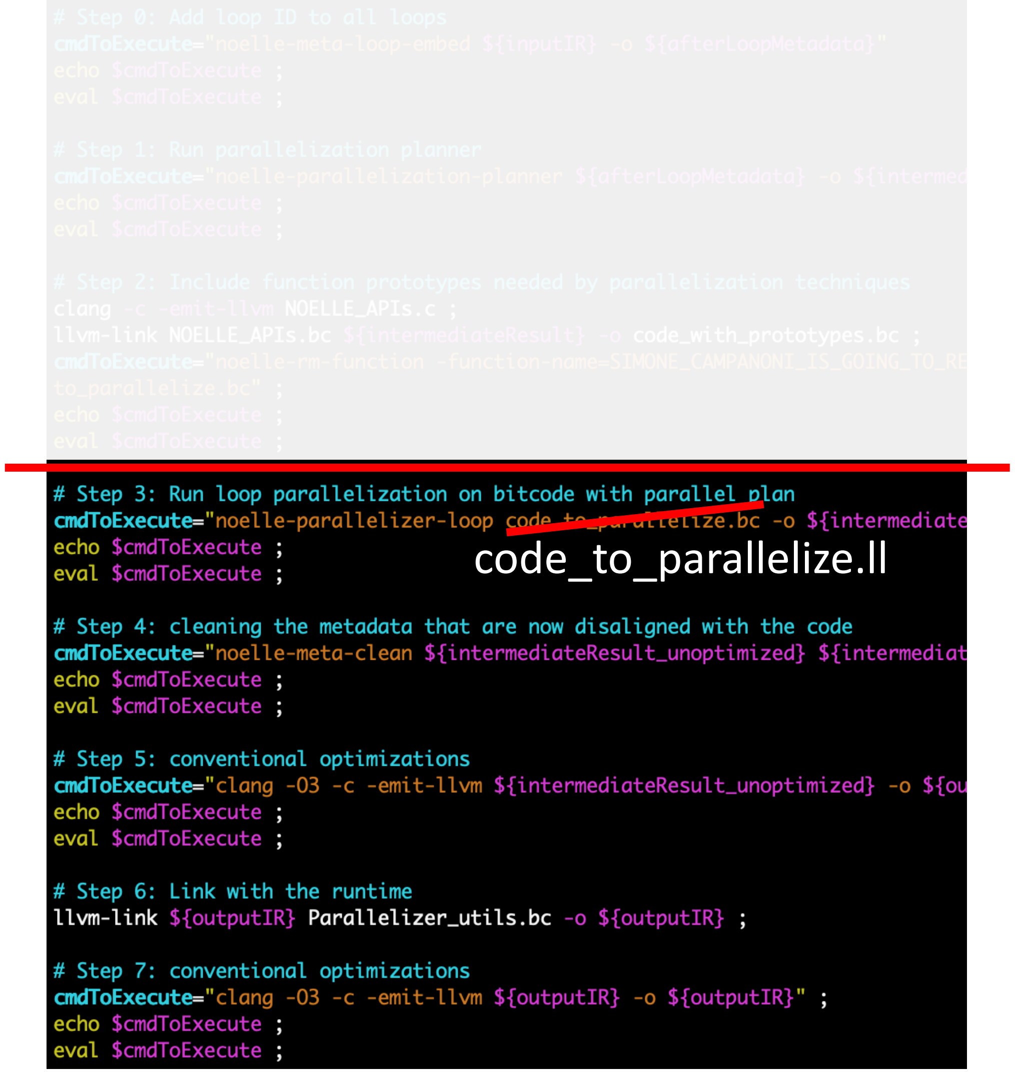
```
# Step 0: Add loop ID to all loops
cmdToExecute="noelle-meta-loop-embed ${inputIR} -o ${afterLoopMetadata}"
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 1: Run parallelization planner
cmdToExecute="noelle-parallelization-planner ${afterLoopMetadata} -o ${intermed
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 2: Include function prototypes needed by parallelization techniques
clang -c -emit-llvm NOELLE_APIs.c ;
llvm-link NOELLE_APIs.bc ${intermediateResult} -o code_with_prototypes.bc ;
cmdToExecute="noelle-rm-function -function-name=SIMONE_CAMPANONI_IS_GOING_TO_RE
to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 3: Run loop parallelization on bitcode with parallel plan
cmdToExecute="noelle-parallelizer-loop code_to_parallelize.bc -o ${intermediate
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
cmdToExecute="noelle-meta-clean ${intermediateResult_unoptimized} ${intermediat
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${ou
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link ${outputIR} Parallelizer_utils.bc -o ${outputIR} ;

# Step 7: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${outputIR} -o ${outputIR}" ;
echo $cmdToExecute ;
eval $cmdToExecute ;
```

An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata



```
# Step 0: Add loop ID to all loops
cmdToExecute="noelle-meta-loop-embed ${inputIR} -o ${afterLoopMetadata}"
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 1: Run parallelization planner
cmdToExecute="noelle-parallelization-planner ${afterLoopMetadata} -o ${intermed
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 2: Include function prototypes needed by parallelization techniques
clang -c -emit-llvm NOELLE_APIs.c ;
llvm-link NOELLE_APIs.bc ${intermediateResult} -o code_with_prototypes.bc ;
cmdToExecute="noelle-rm-function -function-name=SIMONE_CAMPANONI_IS_GOING_TO_RE
to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 3: Run loop parallelization on bitcode with parallel plan
cmdToExecute="noelle-parallelizer-loop code_to_parallelize.bc -o ${intermediate
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
cmdToExecute="noelle-meta-clean ${intermediateResult_unoptimized} ${intermediat
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${ou
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link ${outputIR} Parallelizer_utils.bc -o ${outputIR} ;

# Step 7: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${outputIR} -o ${outputIR}" ;
echo $cmdToExecute ;
eval $cmdToExecute ;
```

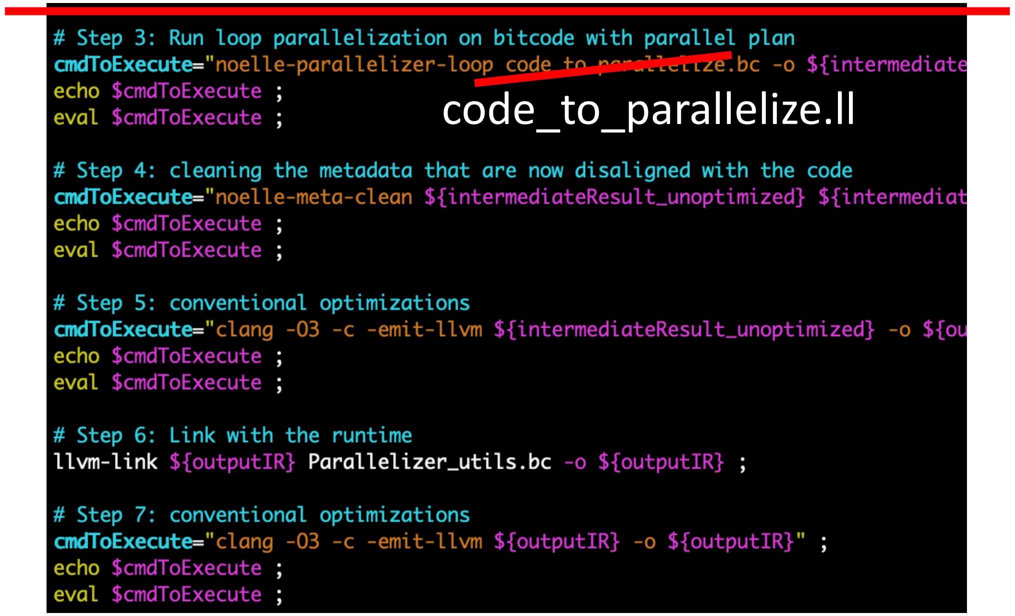
code_to_parallelize.ll

An approach to debug
a loop-based parallelizing compiler

1. Shrinking

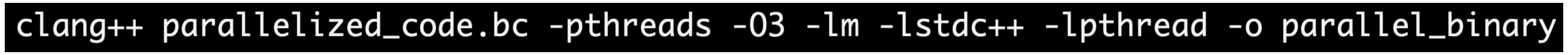
Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata



```
# Step 3: Run loop parallelization on bitcode with parallel plan
cmdToExecute="noelle-parallelizer-loop ~~code_to_parallelize~~.bc -o ${intermediate
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
cmdToExecute="noelle-meta-clean ${intermediateResult_unoptimized} ${intermediat
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${ou
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link ${outputIR} Parallelizer_utils.bc -o ${outputIR} ;

# Step 7: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${outputIR} -o ${outputIR}" ;
echo $cmdToExecute ;
eval $cmdToExecute ;
```

code_to_parallelize.ll



```
clang++ parallelized_code.bc -pthreads -O3 -lm -lstdc++ -lpthread -o parallel_binary
```

An approach to debug a loop-based parallelizing compiler

1. **Shrinking**
   As soon as you found the bad loop, go to step 2

An approach to debug a loop-based parallelizing compiler

1. **Shrinking**:
   Identify a single loop that its parallelization
   (when using the new changes) leads to the bug

2. **Comparing**:
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR)
   of the parallelization between master and the changes.

An approach to debug a loop-based parallelizing compiler

Assumption: the bug fit the common case, which is about parallelizing a given loop (independent on what other loops are parallelized)

1. **Shrinking**:
   Identify a single loop that its parallelization
   (when using the new changes) leads to the bug

2. **Comparing**:
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR)
   of the parallelization between master and the changes.

3. **Correctness checking**:
   Deep analysis on the difference in parallelization that is incorrect
   (by manually checking why that parallelization aspect that differ is incorrect)

Always have faith in your ability

Success will come your way eventually

Best of luck!