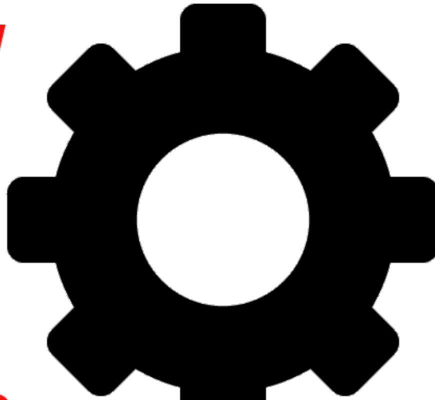
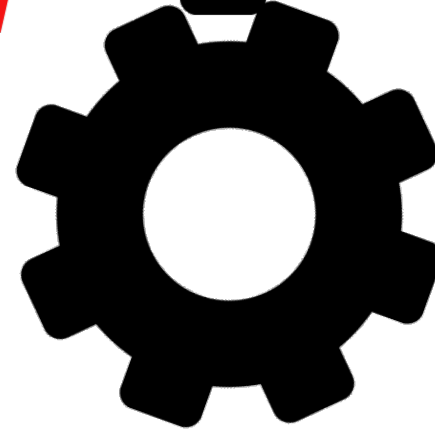
Advanced T  pics *in* C  mpilers

Gino

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- Introducing Gino

- Gino’s compilation pipeline
- Debugging Gino

Gino

- Gino is a parallelizing compiler for LLVM IR

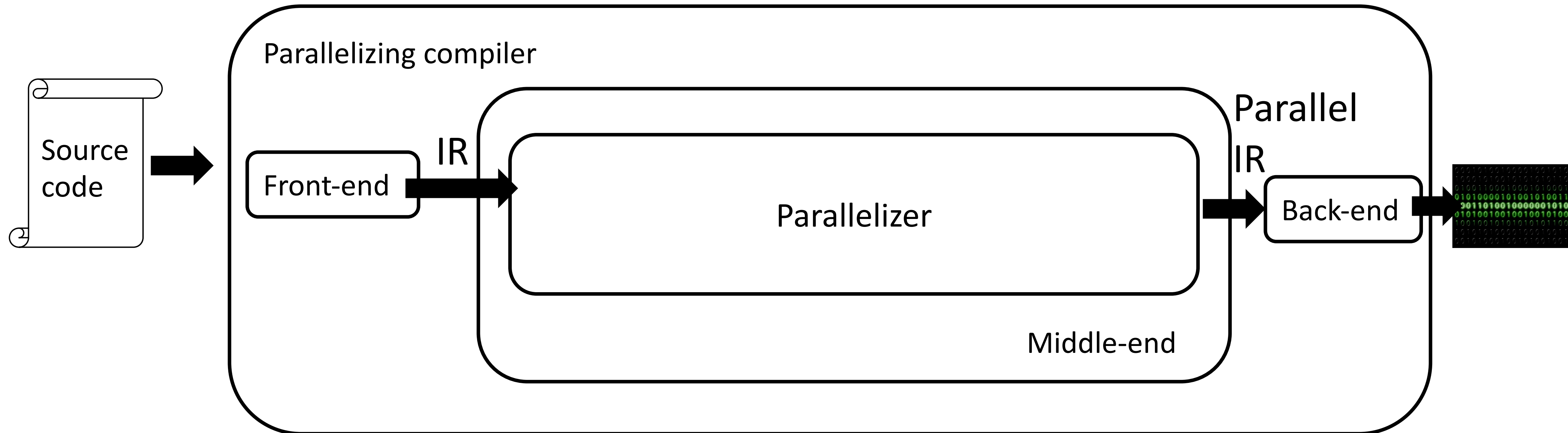
- Standalone codebase
  https://github.com/arcana-lab/gino

```
doc
LICENSE
Makefile
README.md
src
tests
```

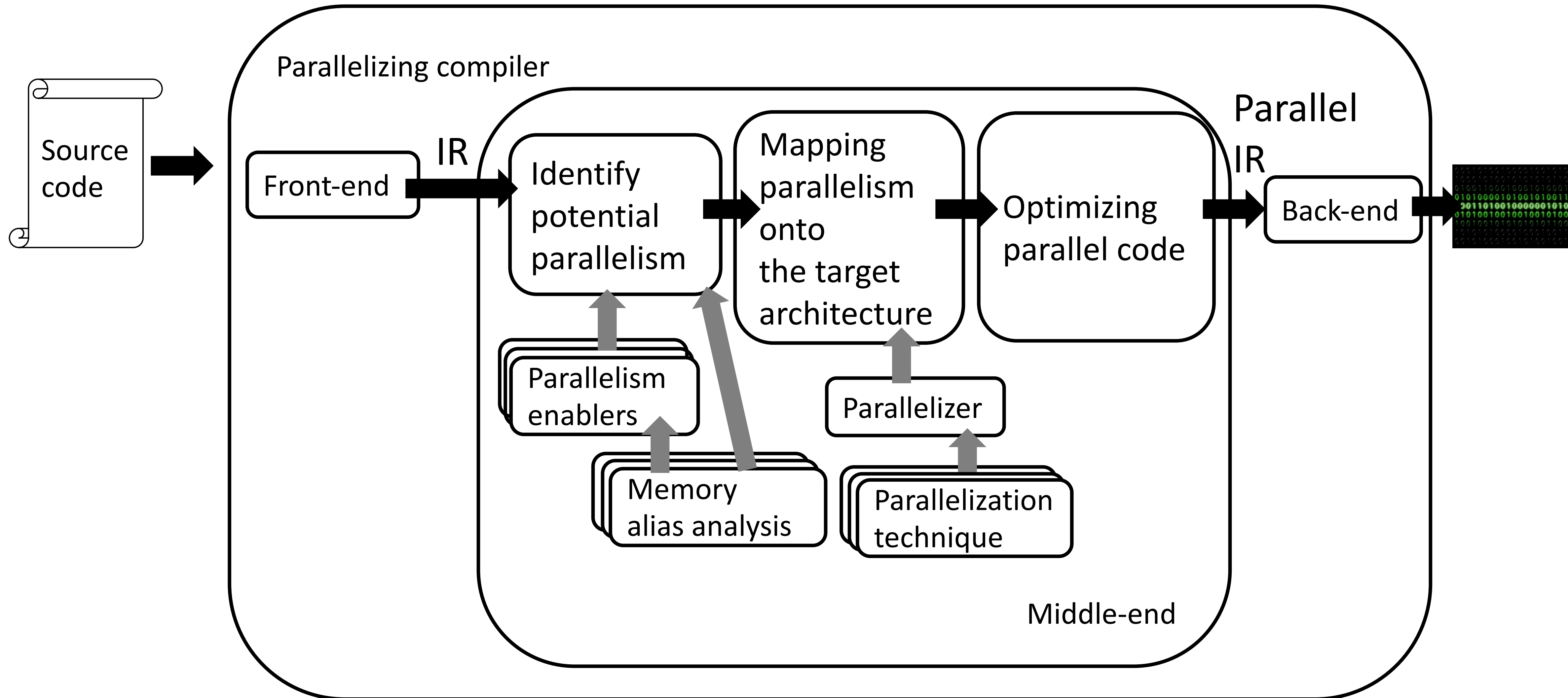
- To compile it
  1. Compile and install NOELLE
  2. Source NOELLE/enable
  3. Go to Gino's codebase and compile Gino
     cd gino ; make

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

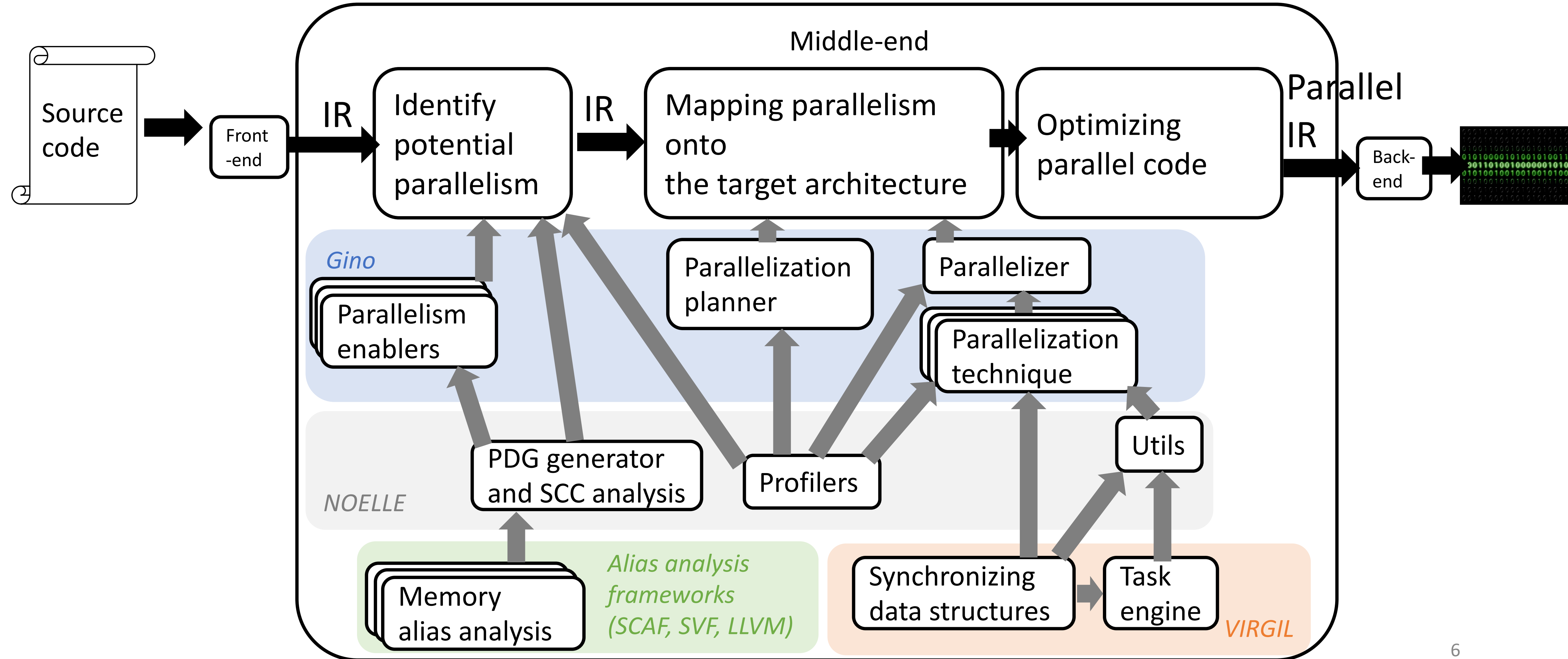
A typical parallelizing compiler



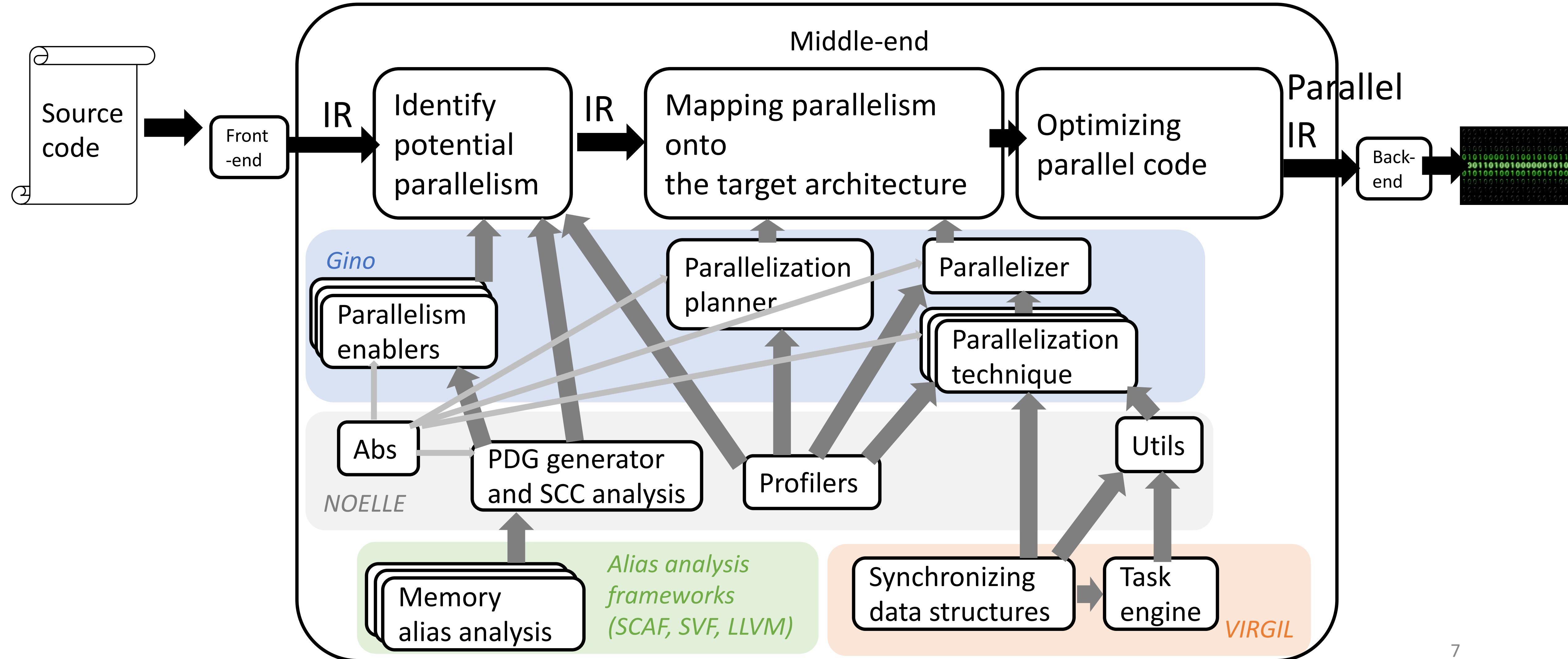
A typical parallelizing compiler



Gino: the parallelizing compiler upon NOELLE



Gino: the parallelizing compiler upon NOELLE



Outline

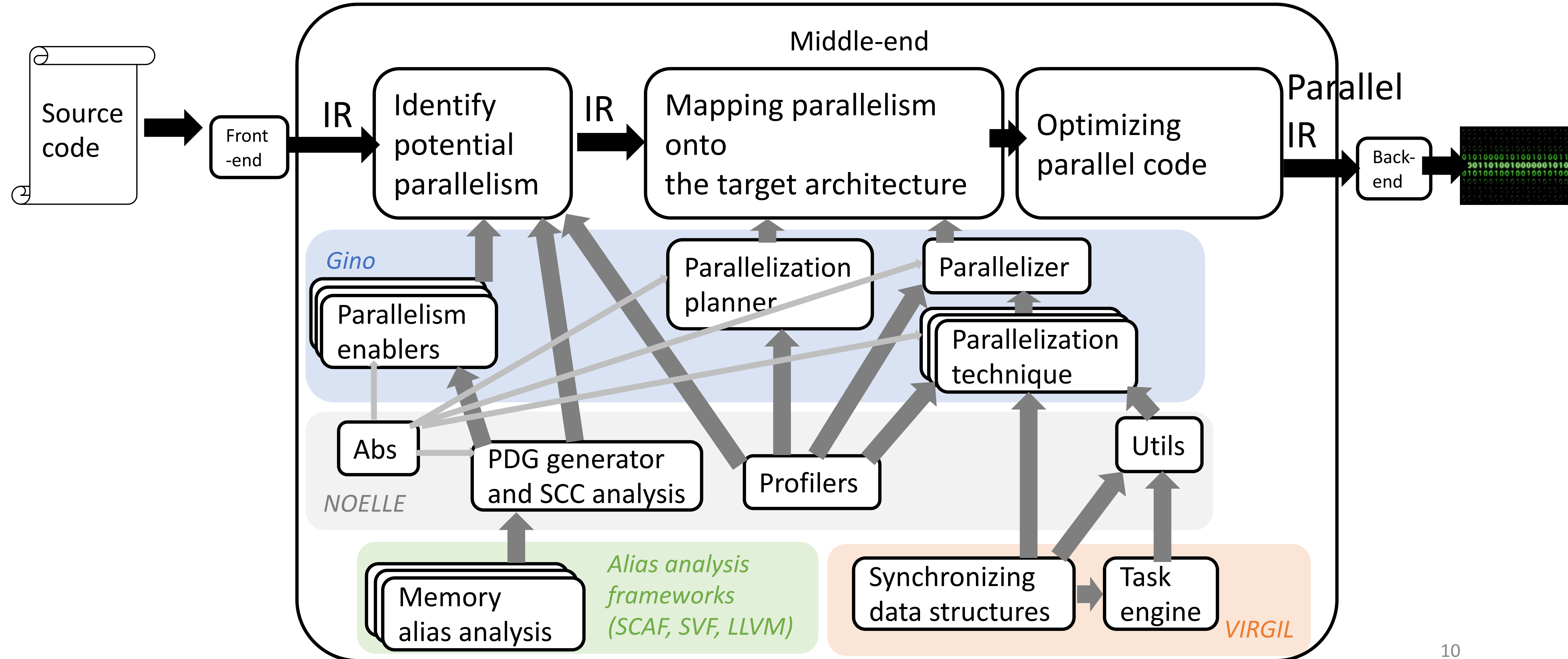
- Introducing Gino
- Gino’s compilation pipeline
- Debugging Gino

Compilation pipeline

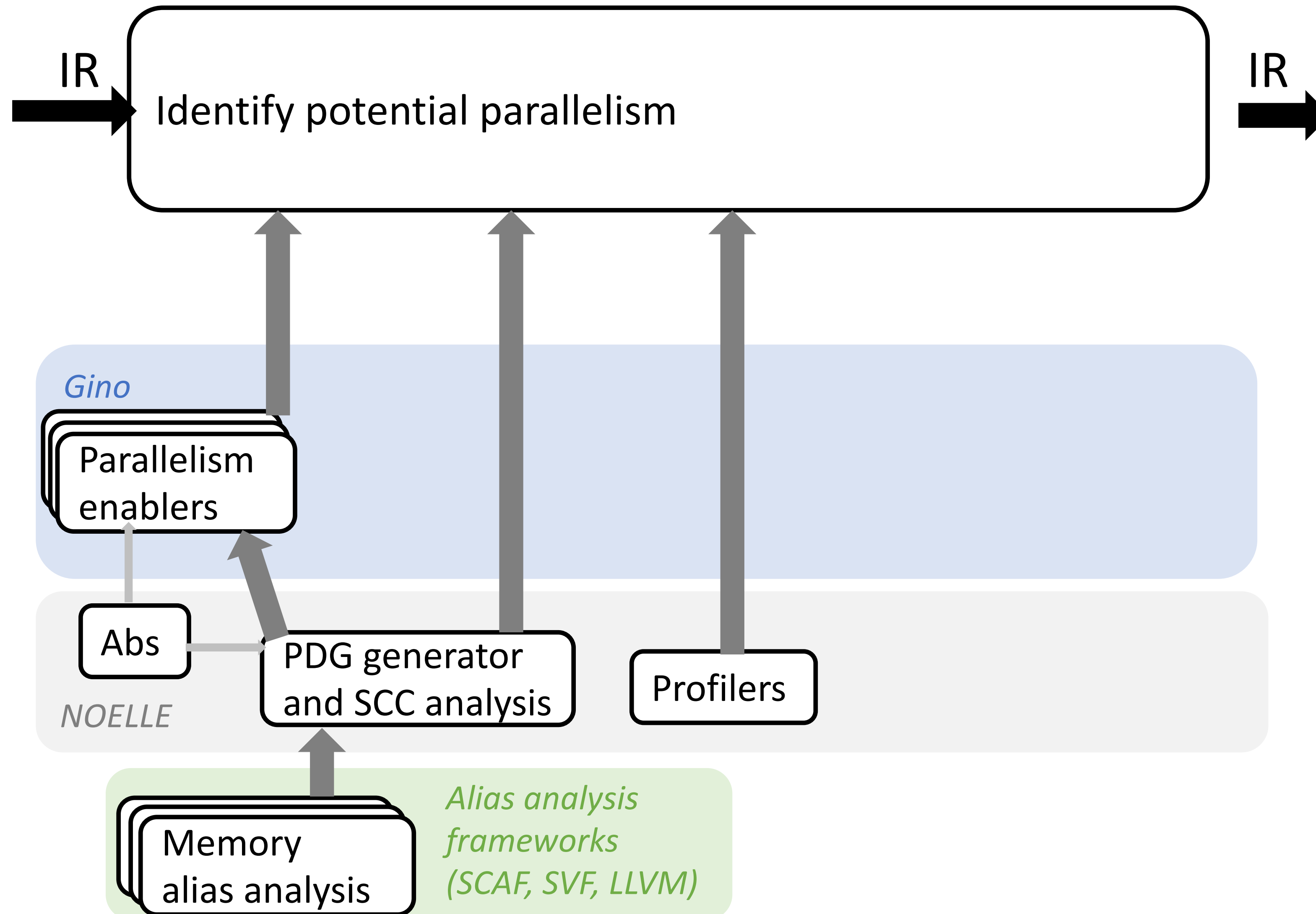
- Let's assume test.cpp is the whole program

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

Gino: the parallelizing compiler upon NOELLE



Gino: the parallelizing compiler upon NOELLE



Compilation pipeline

1. Let's assume test.cpp is the whole program

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

```
noelle-simplification test.bc -o test.bc
```

2. Now we need to profile the code to identify hot code

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

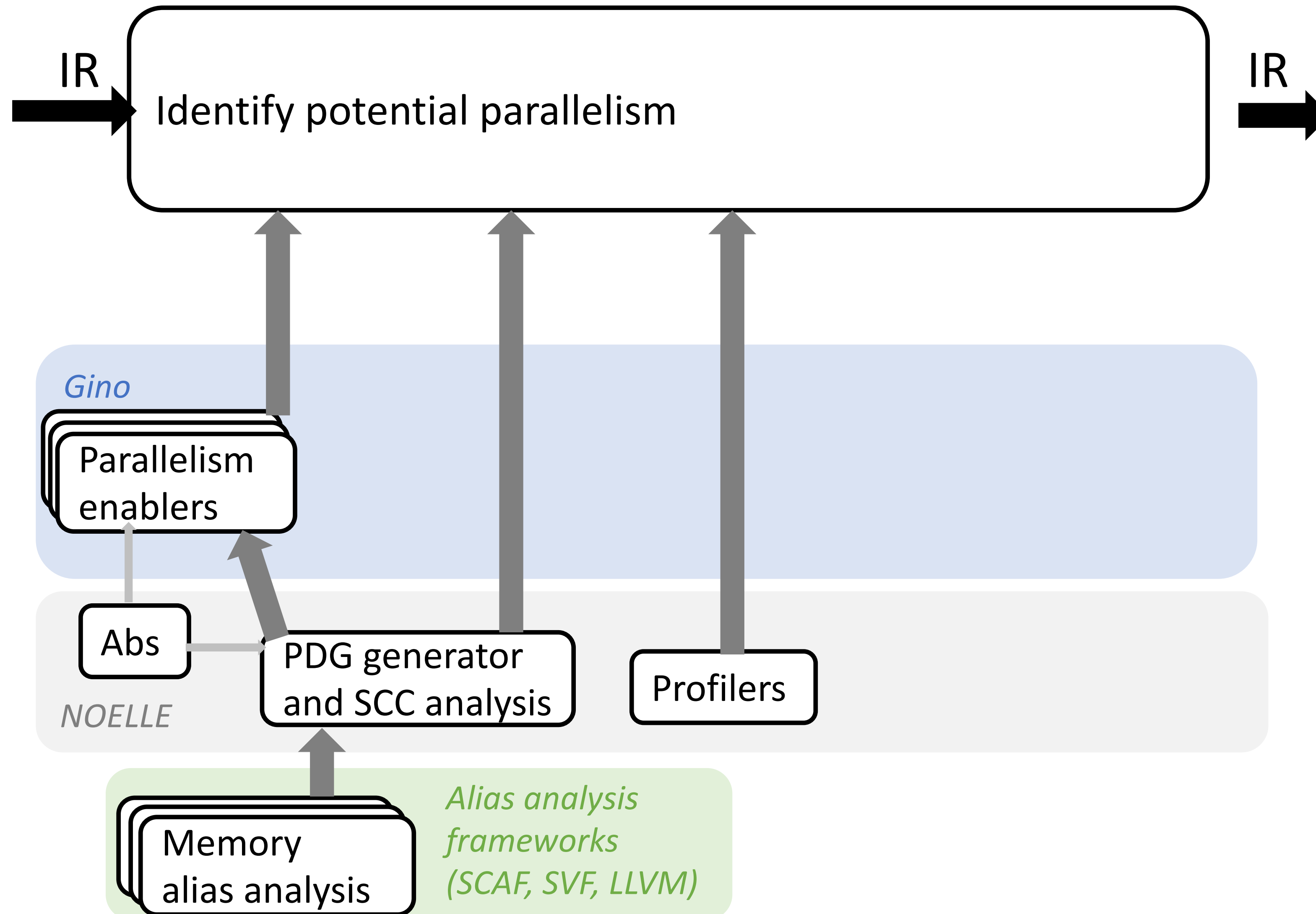
```
$ ./baseline_with_runtime_prof 10 20 30
432500
```

```
$ 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
```

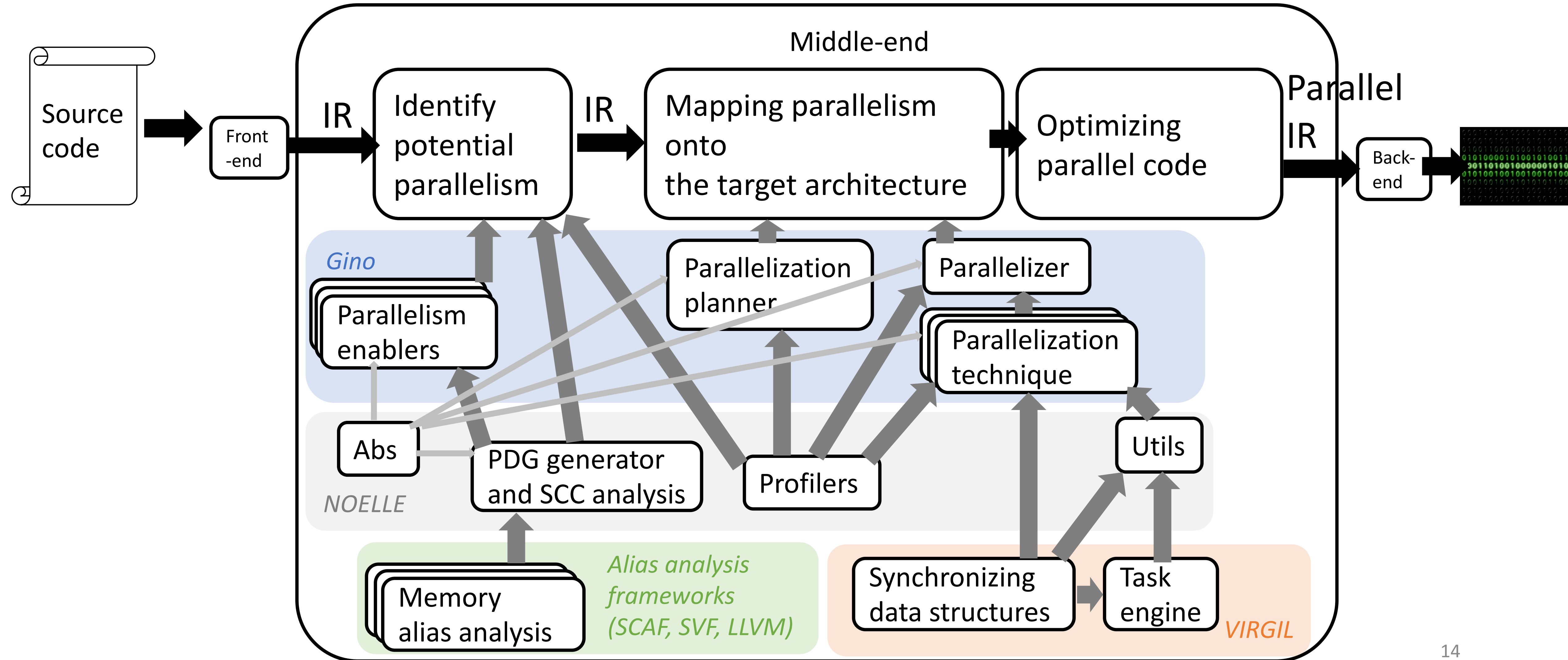
3. Now we need to make the IR more amenable for parallelization

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

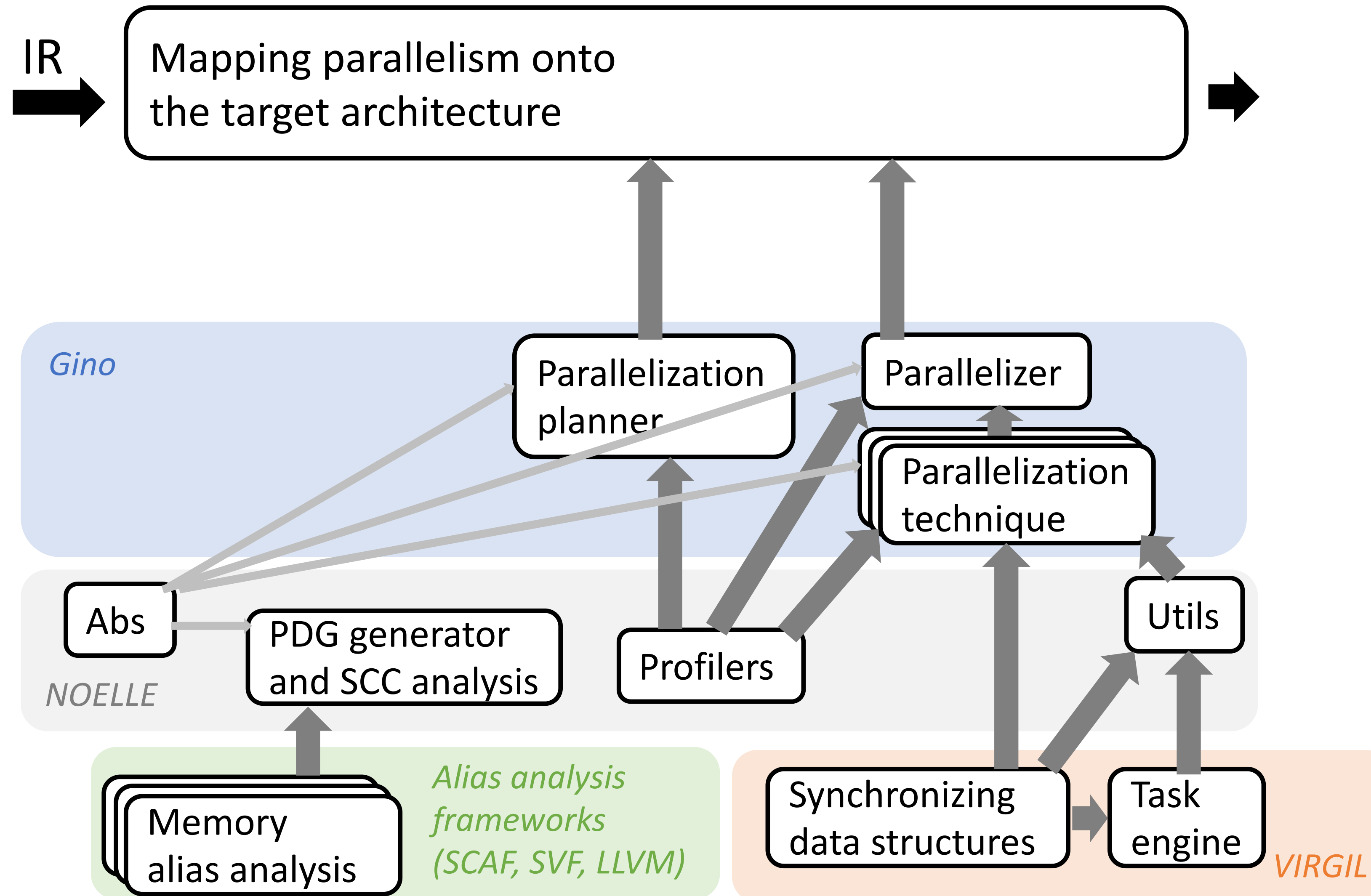
Gino: the parallelizing compiler upon NOELLE



Gino: the parallelizing compiler upon NOELLE



Gino: the parallelizing compiler upon NOELLE



Compilation pipeline

4. 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
```

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

5. 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
```

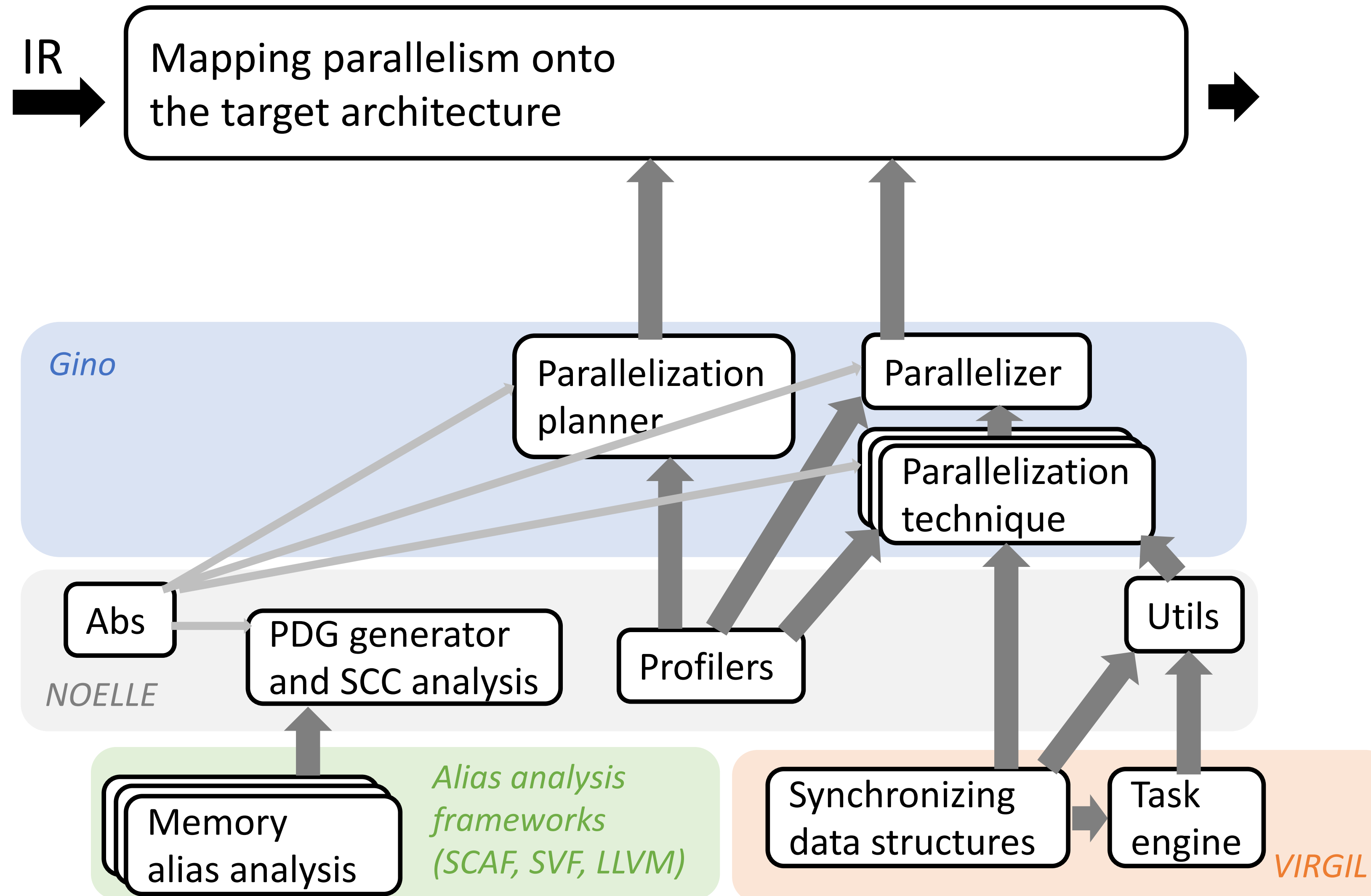
6. Now we can parallelize the IR

```
gino code_to_parallelize.bc -o parallelized_code.bc
```

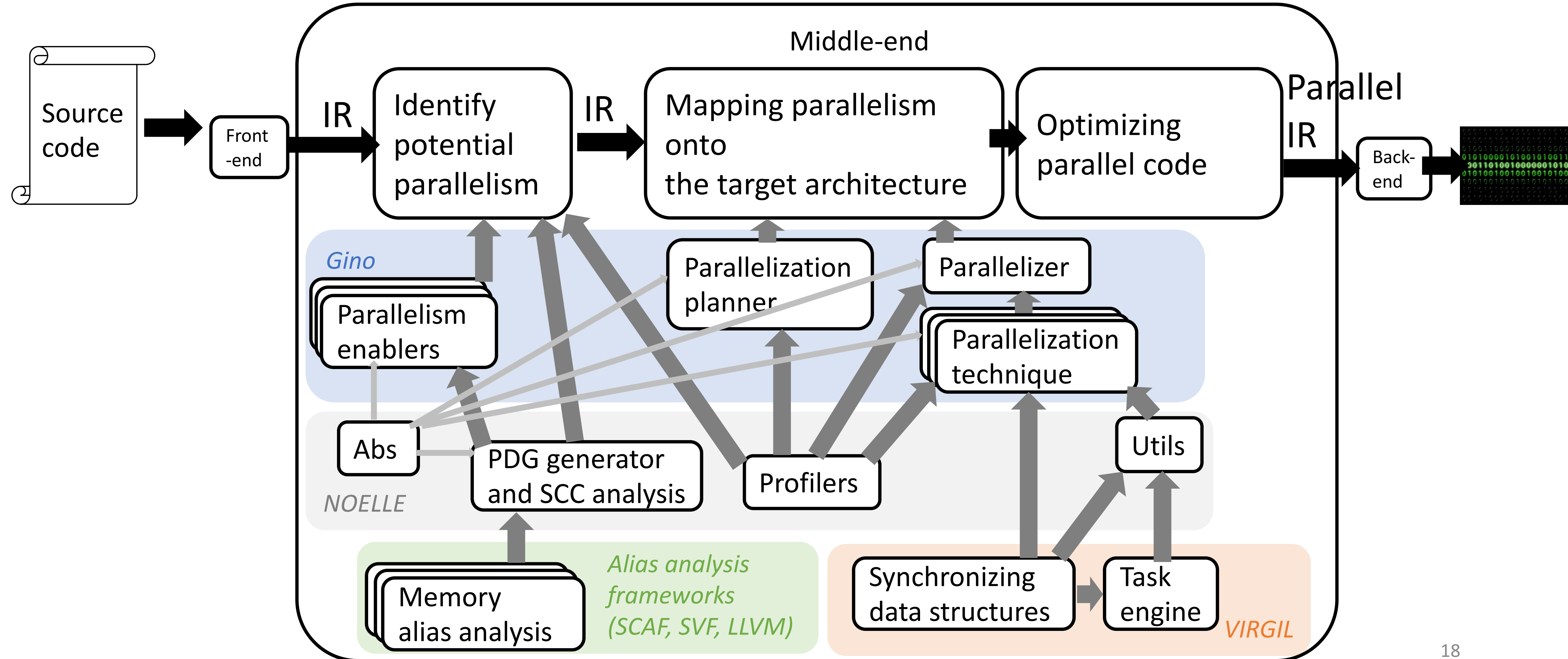
7. Now we can generate the parallelized binary

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

Gino: the parallelizing compiler upon NOELLE



Gino: the parallelizing compiler upon NOELLE



Outline

- Introducing Gino
- Gino’s compilation pipeline
- Debugging Gino

Developing and testing

- Let's say you are working to improve Gino or NOELLE (e.g., induction variable detection algorithm)

```
Contributing.md
LICENSE.md
Makefile
README.md
doc
examples
external
src
tests
```

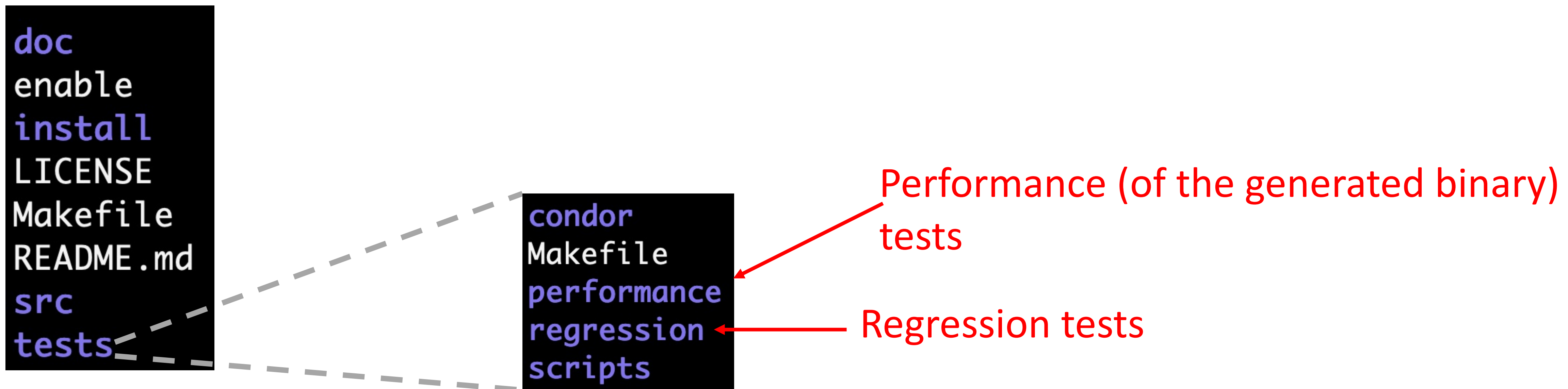
```
Makefile
core
scripts
tools
```

```
CMakeLists.txt
Makefile
alloc_aa
architecture
basic_utilities
callgraph
clean_metadata
dataflow
hotprofiler
induction_variables
invariants
loop_distribution
loop_structure
loop_transformer
loop_unroll
loop_whilifier
loops
metadata_manager
noelle
outliner
pdg
runtime
scheduler
scripts
talkdown
task
transformations
unique_ir_marker
...
```

- You need to test the correctness and impact of your work.
  - Gino can help you to do it

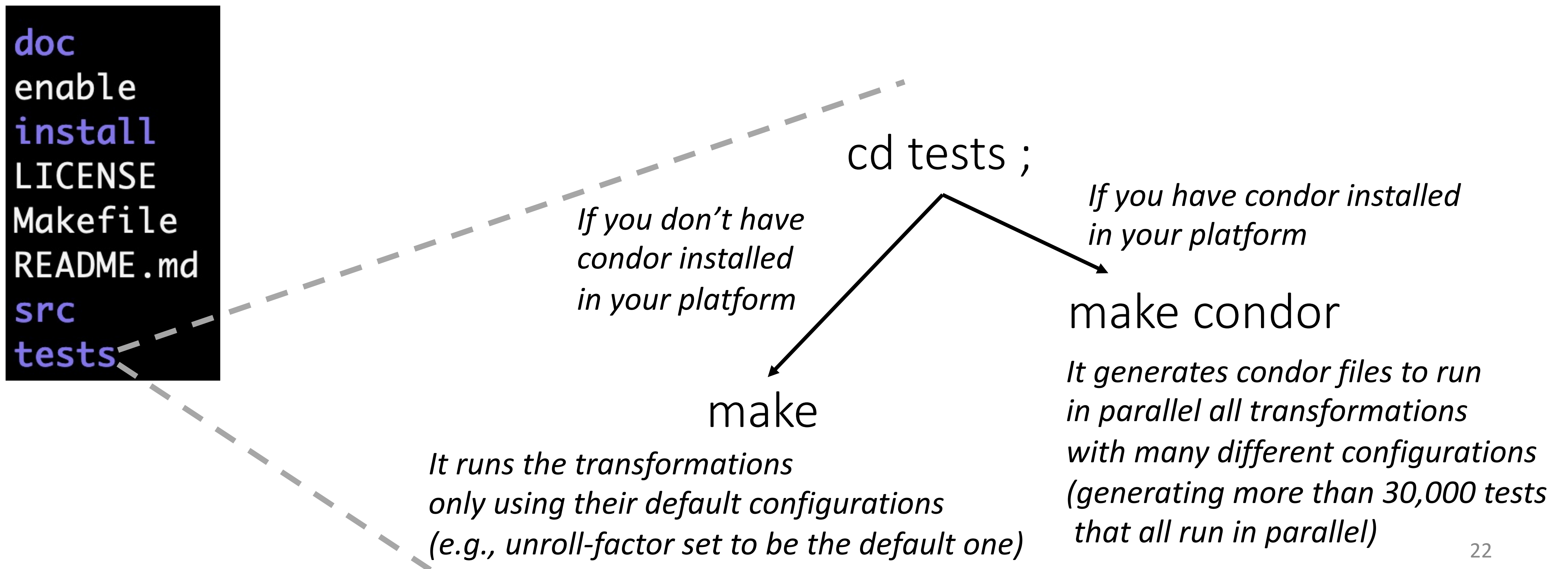
Testing

- NOELLE includes tests for its code transformations (e.g., code parallelization, loop-invariant code motion, etc…)



Testing

- NOELLE includes tests for its code transformations (e.g., code parallelization, loop-invariant code motion, etc…)



Testing with condor

cd tests ; make condor

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

```
condor
Makefile
performance
regression
scripts
```

...

```
regression_65
regression_66
regression_67
regression_68
regression_69
regression_7
regression_70
regression_71
regression_72
regression_73
regression_74
regression_75
regression_76
regression_77
```

...

copy of the original regression dir
one directory per configuration for the code transformations

All these tests
(~30,000 at the moment)
run in parallel!

Testing with condor

cd tests ; make condor

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

```
condor
Makefile
performance
regression
scripts
```

…

```
regression_65
regression_66
regression_67
regression_68
regression_69
regression_7
regression_70
regression_71
regression_72
regression_73
regression_74
regression_75
regression_76
regression_77
```

…

cd tests ; make condor_check

```
$ make condor_check
./scripts/condor_check.sh ;
################################### REGRESSION TESTS:
  Checking the regression test results
    There are 21204 jobs that are still running
    No new tests failed so far
    There are new tests that now pass for all configurations. They are the next ones:

        Chunking
        DSWPIterations_RemovableIntraIterMemEdge
        Exit_call2
        Exit_call3
        IndependentIterations11
        IndependentIterations5
        LICM
        LICM_2
        Multiloops
        Multiloops_list
        ReductionIterationsAnd
        ReductionIterationsOr


################################### UNIT TESTS:
  They are still running


################################### PERFORMANCE TESTS:
  They are still running
```

Testing with condor

cd tests ; make condor

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

```
condor
Makefile
performance
regression
scripts
```

…

```
regression_65
regression_66
regression_67
regression_68
regression_69
regression_7
regression_70
regression_71
regression_72
regression_73
regression_74
regression_75
regression_76
regression_77
```

…

cd tests ; make condor_check

```
$ make condor_check
./scripts/condor_check.sh ;
################################### REGRESSION TESTS:
  Checking the regression test results
    There are 11237 jobs that are still running
    31 new tests failed:
        regression_107/Memory_interprocedural_dependence -noelle-pdg-check -noelle-verbose=3 -noelle-max-cores=2 -noelle-disable-enabler
s -noelle-disable-inliner -noelle-disable-dead -noelle-parallelizer-force -O1 -Xclang -disable-llvm-passes -O0
```

- Tests that completed successfully get automatically deleted
- Directory of a test that failed is kept (so you can debug it; check compiler_output.txt) and a script to reproduce the fail is automatically generated
- To reproduce the fail:
  - Go to the directory of the test (e.g., cd regression_4/Simple)
  - Run ./run_me.sh

Re-run the tests using condor

cd tests ;

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

1. Make sure no tests are still running
   condor_q `whoami`

2. Clean the tests directory
   make clean

3. Run the tests
   make condor

Running a single test without condor

cd tests ; make download

```
doc
enable
install
LICENSE
Makefile
README.md
src
tests
```

1. Go to the test directory
   (e.g., cd regression/Simple)
2. Clean the directory
   make clean
3. Enable NOELLE and Gino binaries in your environment
   source ../../../enable
   source WHERE_NOELLE_IS/enable
4. Run the test
   make test_correctness
5. Check the output
   (look at the makefile to understand the scripts)

Notes about improving Gino or NOELLE

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**

```
gino code_to_parallelize.bc -o parallelized_code.bc
```

An approach to debug a loop-based parallelizing compiler

1. Shrinking

Loops selected by the planner

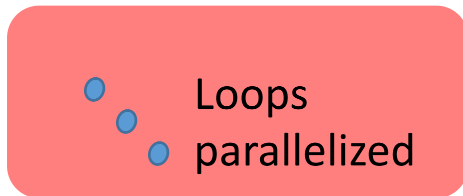
Loops parallelized

Loops of the program that satisfies the options given as input to NOELLE

```
# 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="gino-planner ${afterLoopMetadata} -o ${intermediateResult} ${@:4}" ;
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_REMOVE
code_to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

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

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

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${output
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**

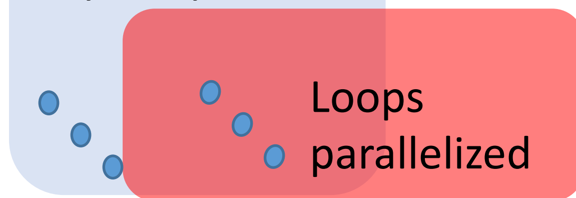


$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

```
# 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="gino-planner ${afterLoopMetadata} -o ${intermediateResult} ${@:4}" ;
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_REMOVE
code_to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

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

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

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

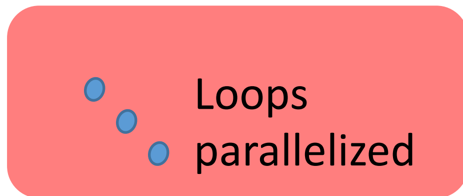


$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

```
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




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

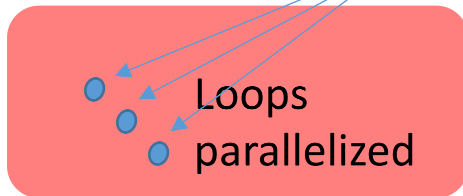
$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

```
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

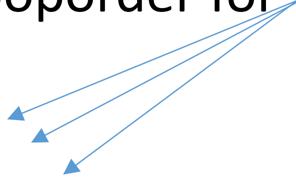
$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll

```
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



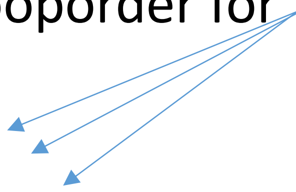
Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata

```
# 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="gino-planner ${afterLoopMetadata} -o ${intermediateResult} ${@:4}" ;
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_REMOVE
code_to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

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

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

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

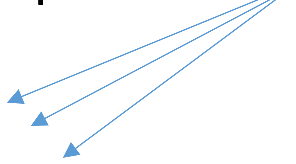
```
# Step 0: Add loop ID to all loops
cmdToExecute="noelle-meta-loop-embed ${inputIR} -o ${afterLoopMetadata}"
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 1: Run parallelization planner
cmdToExecute="gino-planner ${afterLoopMetadata} -o ${intermediateResult} ${@:4}" ;
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_REMOVE
code_to_parallelize.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

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

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

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${output
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="gino-loops code_to_parallelize.bc -o ${intermediateResult_unoptimized
echo $cmdToExecute ;
eval $cmdToExecute ;

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

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm ${intermediateResult_unoptimized} -o ${output
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!