Outline

• The parallelizing compiler built upon NOELLE

• Compilation pipeline

• Debugging
A typical parallelizing compiler

Source code → Front-end → IR → Identify potential parallelism → Mapping parallelism onto the target architecture → Optimizing parallel code → Back-end

Parallelizing compiler

- Parallelism enablers
- Memory alias analysis
- Parallelizer
- Parallelization technique

Front-end → IR → Middle-end → Back-end
The parallelizing compiler built upon NOELLE

Source code → Front-end

Parallelizing compiler

Identify potential parallelism → Middle-end

Mapping parallelism onto the target architecture

Optimizing parallel code → Parallel IR

Parallel IR → Back-end

Front-end

IR

Middle-end

IR

Parallel IR

Parallelizing compiler

NOELLE/tools

Parallelism enablers

NOELLE/core

Memory alias analysis

Alias analysis frameworks (SCAF, SVF, LLVM)

PDG generator and SCC analysis

Parallelization planner

Profilers

Parallelization technique

Parallelizer

Optimizing parallel code

Parallel IR

Synchronizing data structures

Task engine

VIRGIL

Partitioning

Front-end

IR

Middle-end

IR

Parallel IR

Back-end

Source code
The parallelizing compiler built upon NOELLE

Parallelizing compiler

- Source code
- Front-end
- IR
  - Identify potential parallelism
- Parallelism enablers
  - Abs
  - PDG generator and SCC analysis
  - Memory alias analysis
- NOELLE/core

- Parallelism enablers
- NOELLE/tools

Middle-end

- IR
  - Mapping parallelism onto the target architecture
- Parallelization planner
- Profilers

Parallelizer

- Parallelization technique
- Optimezing parallel code
- Back-end

- Utilities
- Task engine

Parallelizer

- Synchronizing data structures
- Alias analysis frameworks (SCAF, SVF, LLVM)
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

Parallelizing compiler

Source code

Front-end

IR

Identify potential parallelism

Parallelism enablers

NOELLE/tools

NOELLE/core

IR

Mapping parallelism onto the target architecture

Parallelization planner

Parallelizer

Parallelization technique

Optimizing parallel code

IR

Middle-end

Back-end

Parallel IR

IR

Back-end

Optimizing parallel code

Parallelizer

Parallelization technique

Profilers

NOELLE/tools

Parallelism enablers

NOELLE/core

IA

Parallel IR

Back-end

Source code

Front-end

IR

Identify potential parallelism

Parallelism enablers

NOELLE/tools

NOELLE/core

IR

Mapping parallelism onto the target architecture

Parallelization planner

Parallelizer

Parallelization technique

Optimizing parallel code

IR

Middle-end

Back-end

Parallel IR

IR

Back-end

Optimizing parallel code

Parallelizer

Parallelization technique

Profilers

NOELLE/tools

Parallelism enablers

NOELLE/core

Alias analysis frameworks (SCAF, SVF, LLVM)

Synchronizing data structures

Task engine

VIRGIL

Mem
The parallelizing compiler built upon NOELLE

- Identify potential parallelism
- NOELLE/tools
- Parallelism enablers
- NOELLE/core
- Abs
- PDG generator and SCC analysis
- Profilers
- 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
- 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

Parallelizing compiler

- Source code
- Front-end
  - IR
  - Identify potential parallelism
- Middle-end
  - IR
  - Mapping parallelism onto the target architecture
- Back-end
  - IR
  - Optimizing parallel code

Front-end:
- NOELLE/tools
- Parallelism enablers
  - Abs
  - PDG generator and SCC analysis
- NOELLE/core
  - Memory alias analysis
  - Alias analysis frameworks (SCAF, SVF, LLVM)

Middle-end:
- Parallelization planner
- Parallelizer
- Parallelization technique
- Profilers
- Utilities
- Task engine
- Synchronizing data structures

Back-end:
- Task engine
- Synchronizing data structures
- Utilities
- Profilers
- Parallelizer
- Parallelization technique
- Parallelization planner
- PDG generator and SCC analysis
- Abs
- Memory alias analysis
- Alias analysis frameworks (SCAF, SVF, LLVM)
- NOELLE/core
- NOELLE/tools

- Source code
- Front-end
The parallelizing compiler built upon NOELLE

- **NOELLE/core**: Abs, Memory alias analysis, Parallelization planner, PDG generator and SCC analysis, Profilers, Synchronizing data structures, Task engine
- **NOELLE/tools**: Parallelizer, Parallelization technique, Utilities
- **VIRGIL**: Alias analysis frameworks (SCAF, SVF, LLVM)

Mapping parallelism onto the target architecture
Compilation pipeline

• We need to profile the code

```sh	noelle-prof-coverage test_with_profile.bc baseline_with_runtime_prof -lm -lstdc++ -lpthread
$ ./baseline_with_runtime_prof 10 20 30
432500
```

```sh
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

```sh
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)

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

```sh
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)

```bash
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

```bash
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

```bash
clang++ parallelized_code.bc -lpthreads -O3 -lm -lstdc++ -lpthread -o parallel_binary
```
The parallelizing compiler built upon NOELLE

Mapping parallelism onto the target architecture

NOELLE/tools

Parallelization planner

PDG generator and SCC analysis

NOELLE/core

Memory alias analysis

Abs

Profilers

Parallelizer

Parallelization technique

Utils

Synchronizing data structures

Task engine

VIRGIL

Alias analysis frameworks (SCAF, SVF, LLVM)

16
The parallelizing compiler built upon NOELLE

Parallelizing compiler

Source code

Front-end

IR

Identify potential parallelism

NOELLE/tools

Parallelism enablers

Abs

PDG generator and SCC analysis

NOELLE/core

Parallelism enablers

Memory alias analysis

Alias analysis frameworks (SCAF, SVF, LLVM)

middle-end

IR

Mapping parallelism onto the target architecture

Parallelization planner

Parallelizer

Parallelization technique

Profilers

Optimizing parallel code

Parallel IR

Back-end

Parallelization technique

Synchronizing data structures

Task engine

VIRGIL
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

```bash
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

```bash
# Step 0: Add loop ID to all loops
cmdToExecute="moelle-meta-loop-embed $[inputIR] -o $(afterLoopMetadata)"
echo $cmdToExecute;
eval $cmdToExecute;

# Step 1: Run parallelization planner
cmdToExecute="moelle-parallelization-planner $(afterLoopMetadata) -o $(intermediate)"
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="moelle-rm-function -Function-name=SIMONE_CAMPAIGNONI_IS_GOING_TO_REALIZE_parallelize.bc" ;
echo $cmdToExecute;
eval $cmdToExecute;

# Step 3: Run loop parallelization on bytecode with parallel plan
cmdToExecute="moelle-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="moelle-meta-clean $(intermediateResult_unoptimized) $(intermediate)"
echo $cmdToExecute;
eval $cmdToExecute;

# Step 5: conventional optimizations
cmdToExecute="clang -O3 -c -emit-llvm $(intermediateResult_unoptimized) -o $(outputIR)"
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

```bash
$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
```
An approach to debug a loop-based parallelizing compiler

1. **Shrinking**

Loops selected by the planner

Loops parallelized

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
An approach to debug a loop-based parallelizing compiler

1. Shrinking

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
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
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove loop order 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 loop order metadata
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

```bash
# Step 2: Include function prototypes needed by parallelization techniques
clang -c -I include -I MELLIE_APIs.h -I $(dynamicLibPath) -o code_with_prototypes.bc ;
cmdToExecute="moelle-re-function -function-name=COORDIINATE_IS_GOING_TO_NEW_IR -o code_with_prototypes.bc" ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 3: Run loop parallelization on bytecode with parallel plan
clang -c -I include -I MELLIE_APIs.h -I $(dynamicLibPath) -o $(parallelizationPlan) ;
cmdToExecute="moelle-parallelizer-loop code_to_parallelize.ll -o $(intermediate) ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 4: cleaning the metadata that are now disaligned with the code
clang -c -I include -I MELLIE_APIs.h -I $(dynamicLibPath) -o $(intermediateResult_unoptimized) ;
cmdToExecute="moelle-meta-clean $(intermediateResult_unoptimized) $(intermediate) ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 5: conventional optimizations
clang -c -I include -I MELLIE_APIs.h -I $(dynamicLibPath) -o $(intermediateResult_unoptimized) ;
cmdToExecute="clang -c -I include -I MELLIE_APIs.h -I $(dynamicLibPath) -o $(intermediateResult_unoptimized) ;
echo $cmdToExecute ;
eval $cmdToExecute ;

# Step 6: Link with the runtime
llvm-link $(outputIR) Parallelizer_utils.bc -o $(outputIR) ;
```

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

```bash
clang++ parallelized_code.bc -pthread -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!