Tapir: Embedding Fork-Join Parallelism into LLVM’s Intermediate Representation

William S. Moses  Tao B. Schardl  Charles E. Leiserson

6.S898 Lecture
April 13, 2017
What is a compiler?
A Good Compiler Does Wonders

Minutes vs Hours!
Compilers Don’t Understand Parallel Code

What’s that?

cilk_for (int i = 0; i < n; ++i) {
    do_work(i);
}

#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    do_work(i);
}
Example: Normalizing a Vector

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Test: random vector, n = 64M.
Example: Normalizing a Vector

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Test: random vector, n = 64M.

Running time: 0.312 s
Idea: Run in Parallel!
Example: Normalizing a Vector in Parallel

__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    cilk_for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}

Test: random vector, n = 64M.

Original serial running time: $T_s = 0.312$ s
Example: Normalizing a Vector in Parallel

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    cilk_for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

A parallel loop replaces the original serial loop.

Test: random vector, n = 64M.

*Original serial running time: $T_S = 0.312$ s*

*18-core running time: 180.657s*
Example: Normalizing a Vector in Parallel

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    cilk_for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

A parallel loop replaces the original serial loop.

Test: random vector, n = 64M.

*Original serial running time:* $T_S = 0.312$ s

*18-core running time:* $180.657$ s

*1-core running time:* $2600.287$ s
What happened?
The LLVM Compilation Pipeline

C code → Clang → LLVM → -O3 → LLVM → CodeGen → EXE

Front end

Middle-end optimizer

Back end
Effect of Compiling Serial Code

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    double tmp = norm(in, n);
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / tmp;
}
```
Compiling Parallel Code Today

LLVM pipeline

C → Clang → LLVM → -O3 → LLVM → CodeGen → EXE

Cilk Plus/LLVM pipeline

Cilk → PClang → LLVM → -O3 → LLVM → CodeGen → EXE

The front end translates all parallel language constructs.
Effect of Compiling Parallel Code

```
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    cilk_for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Call into runtime to execute parallel loop.

Helper function encodes the loop body.

Existing optimizations cannot move call to norm out of the loop.
A More Complex Example

Cilk Fibonacci code

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```

Optimization passes struggle to optimize around these opaque runtime calls.
**Idea: A Parallel IR**

💡 Let’s embed parallelism directly into the compiler’s intermediate representation (IR)!

**LLVM pipeline**

```
C → Clang → LLVM → -O3 → LLVM → CodeGen → EXE
```

**Cilk Plus/LLVM pipeline**

```
Cilk → PClang → LLVM → -O3 → LLVM → CodeGen → EXE
```

**A better Cilk compilation pipeline**

```
Cilk → PClang → PLLVM → -O3 → PLLVM → CodeGen → EXE
```

New IR that encodes parallelism for optimization.
Previous Attempts at Parallel IR’s

- Parallel precedence graphs [SW91, SHW93]
- Parallel flow graphs [SG91, GS93]
- Concurrent SSA [LMP97, NUS98]
- Parallel program graphs [SS94, S98]
- **HIR** [ZS11, BZS13]
- **SPIRE** [KJAI12]
- **INSPIRE** [JPTKF13]
- LLVM’s parallel loop metadata
- “[LLVMdev] [RFC] Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.)” [http://lists.llvm.org/pipermail/llvm-dev/2012-August/052477.html](http://lists.llvm.org/pipermail/llvm-dev/2012-August/052477.html)
- “[LLVMdev] [RFC] Progress towards OpenMP support” [http://lists.llvm.org/pipermail/llvm-dev/2012-September/053326.html](http://lists.llvm.org/pipermail/llvm-dev/2012-September/053326.html)
- LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications [KJIAC15]
- LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading [TSSGMGZ16]
Parallel IR: A Bad Idea?

From “[LLVMdev] LLVM Parallel IR,” 2015:

❖ “[I]ntroducing [parallelism] into a so far ‘sequential’ IR will cause severe breakage and headaches.”

❖ “[P]arallelism is invasive by nature and would have to influence most optimizations.”

❖ “[I]t is not an easy problem.”

❖ “[D]efining a parallel IR (with first class parallelism) is a research topic…”

Other communications, 2016–2017:

❖ “There are a lot of information needs to be represented in IR for [back end] transformations for OpenMP.” [Private communication]

❖ “If you support all [parallel programming features] in the IR, a *lot* [of LOC]… would probably have to be modified in LLVM.” [[RFC] IR-level Region Annotations]
Tapir: Task-based Asymmetric Parallel IR

Tapir adds three instructions to LLVM IR that encode fork-join parallelism.

With few changes, LLVM’s existing optimizations and analyses work on parallel code.
Normalizing a Vector in Parallel with Tapir

Cilk code for `normalize()`

```c
__attribute__((const)) double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n) {
    cilk_for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Test: random vector, \( n = 64M \). Machine: Amazon AWS c4.8xlarge, 18 cores.

Running time of original serial code: \( T_S = 0.312 \) s

Compiled with Tapir/LLVM, running time on 1 core: \( T_1 = 0.321 \) s

Compiled with Tapir/LLVM, running time on 18 cores: \( T_{18} = 0.081 \) s

Great work efficiency: \( T_S / T_1 = 97\% \)
Work-Efficiency Improvement

Same as Tapir/LLVM, but the front end handles parallel language constructs the traditional way.

Decreasing difference between Tapir/LLVM and Reference

Test machine: Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM
## Implementing Tapir/LLVM

<table>
<thead>
<tr>
<th>Compiler component</th>
<th>LLVM 4.0svn (lines)</th>
<th>Tapir/LLVM (lines)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>105,995</td>
<td>943</td>
<td>1,768</td>
</tr>
<tr>
<td>Memory behavior</td>
<td>21,788</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td>Optimizations</td>
<td>152,229</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>Parallelism lowering</td>
<td>0</td>
<td>3,782</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>3,803,831</td>
<td>460</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4,083,843</strong></td>
<td><strong>6,010</strong></td>
<td></td>
</tr>
</tbody>
</table>
Compiler Analyses and Optimizations

What did we do to adapt existing analyses and optimizations?

❖ Dominator analysis: no change
❖ Common-subexpression elimination: no change
❖ Loop-invariant-code motion: 25-line change
❖ Tail-recursion elimination: 68-line change

Tapir also enables new parallel optimizations, such as unnecessary-synchronization elimination and puny-task elimination, which were implemented in 52 lines total.
Why does it work?
LLVM IR

LLVM represents each function as a control-flow graph (CFG).

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n - 1);
    y = fib(n - 2);
    return x + y;
}
```

For serial code a basic block sees values from just one predecessor at runtime.
Example Previous Parallel IR

Previous parallel IR’s based on CFG’s model parallel tasks symmetrically.

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```

**Problem:** The join block breaks implicit assumptions made by the compiler. **Example:** Values from all predecessors of a join must be available at runtime [LMP97].
Tapir vs. Previous Approaches

Tapir’s instructions model parallel tasks **asymmetrically**.

A control-flow edge connects one parallel task to another task, **not to a join**.
Tapir Semantics

- Tapir introduces three new opcodes into LLVM’s IR: `detach`, `reattach`, and `sync`.
- The successors of a detach terminator are the detached block and continuation and may run in parallel.
- Execution after a `sync` ensures that all detached CFG’s in scope have completed execution.
Tapir Semantics

- When run serially, programs first execute the detached CFG and then the continuation
- Registers computed in the detached CFG are not available in the continuation
- Tapir simultaneously represents the serial and parallel semantics of the program
Serial Elision of a Tapir Program

Tapir models the **serial elision** of the parallel program.

![Tapir CFG](image)

### Tapir CFG

- **entry**
  - x = alloca()
  - br (n < 2), exit, if.else

- **if.else**
  - detach det, cont

  - det
    - x0 = fib(n - 1)
    - store x0, x
    - reattach cont

  - cont
    - y = fib(n - 2)
    - sync
    - x1 = load x
    - add = x1 + y
    - br exit

- **det**
  - x0 = fib(n - 1)
  - store x0, x

- **cont**
  - br det

- **exit**
  - rv = \( \phi([n,entry],[add,cont]) \)
  - return rv

### Serial elision

- **entry**
  - x = alloca()
  - br (n < 2), exit, if.else

- **if.else**
  - br det

- **det**
  - x0 = fib(n - 1)
  - store x0, x

- **cont**
  - y = fib(n - 2)
  - nop
  - x1 = load x
  - add = x1 + y
  - br exit

- **exit**
  - rv = \( \phi([n,entry],[add,cont]) \)
  - return rv
Parallel Loops in Tapir

void normalize(double *restrict out,  
               const double *restrict in,  
               int n) {
    cilk_for (int i = 0; i < n; ++i)  
        out[i] = in[i] / norm(in, n);
}

Parallel loop resembles a serial loop with a detached body.

The sync waits on a dynamic set of detached sub-CFG’s.
Intuitively, much of the compiler can reason about a Tapir CFG as a minor change to that CFG’s serial elision.

Many parts of the compiler can apply standard implicit assumptions of the CFG to this block.
Maintaining Correctness

**Problem:** How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?
**Problem:** How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- It suffices to consider moving memory operations around each new instruction.
**Problem:** How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- It suffices to consider moving memory operations around each new instruction.
- Moving code above a **detach** or below a **sync** serializes it and is always valid.
Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- It suffices to consider moving memory operations around each new instruction.
- Moving code above a detach or below a sync serializes it and is always valid.
- Other potential races are handled by giving detach, reattach, and sync appropriate attributes and by slight modifications to mem2reg.
Valid serial passes cannot create race bugs.

Most of LLVM’s existing serial passes “just work” on parallel code.
Case Study: Common Subexpression Elimination

- CSE “just works.”
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for detach / reattach.

```c
void query(int n) {
    int x = cilk_spawn
    { search(0,n/2); }
    int y = search(n/2,n);
    cilk_sync;
    return x + y;
}
```
Case Study: Common Subexpression Elimination

- CSE “just works.”
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for detach/reattach.

```c
void query(int n) {
    int x = cilk_spawn
        { search(0,n/2); }
    int y = search(n/2,n);
    cilk_sync;
    return x + y;
}
```
Case Study: Parallel Tail-Recursion Elimination

- A minor modification allows TRE to run on parallel code.
- Ignore `sync`'s before a recursive call and add `sync`'s before intermediate returns.

```c
void qsort(int* begin, int* end) {
    if (begin == end) return;
    int* mid = partition(start, end);
    swap(end, mid);
    cilk_spawn qsort(begin, mid);
    qsort(mid, end);
    cilk_sync;
}
```
Case Study: Parallel Tail-Recursion Elimination

**entry**
- \( \text{br (begin == end), end, part} \)

**part**
- \( \text{mid = partition(start,end)} \)
- \( \text{swap(end,mid)} \)
- \( \text{detach \ det, cont} \)

**det**
- \( \text{qsort(begin,mid)} \)
- \( \text{reattach \ cont} \)

**cont**
- \( \text{qsort(mid,end)} \)
- \( \text{sync} \)

**end**
- \( \text{return} \)

**entry**
- \( \text{br (begin == end), end, part} \)

**part**
- \( \text{mid = partition(start,end)} \)
- \( \text{swap(end,mid)} \)
- \( \text{detach \ det, cont} \)

**det**
- \( \text{qsort(begin,mid)} \)
- \( \text{reattach \ cont} \)

**cont**
- \( \text{return} \)

**end**
- \( \text{sync \ return} \)
Conclusion

- Tapir enables existing serial optimizations to operate on fork-join parallel code
- Tapir requires minimal compiler modifications
- Tapir opens the door for parallel optimizations
- Ongoing development (bug fixes, new optimizations, etc).

- Available on GitHub!
  https://github.com/wsmoses/Parallel-IR.git