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

William S. Moses
IBM PL Day 2016
December 5, 2016

Joint work with
Tao B. Schardl and Charles E. Leiserson
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. Machine: Amazon AWS c4.8xlarge.

Running time: 0.396 s
Example: Normalizing a Vector in Parallel

OpenMP code for `normalize()`

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

void normalize(double *restrict out, const double *restrict in, int n) {
    #pragma omp parallel for
    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: \( T_S = 0.396 \text{s} \)

Running time on 18 cores: \( T_{18} = 167.731 \text{s} \)

Running time on 1 core: \( T_1 = 2316.063 \text{s} \)

Terrible work efficiency: \( \frac{T_S}{T_1} = \frac{0.396}{2316} \approx 1/5800 \)
Example: Normalizing a Vector in Parallel

Affects Cilk and other frameworks too!

```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);
}
```
Tapir: Task-Based Parallel IR

- Tapir is an extension to LLVM that embeds fork-join parallelism in the Intermediate Representation (IR).
- Tapir allows standard compiler optimizations to operate across parallel control constructs.
- Tapir/LLVM required only about 5000 lines of code compared with the 3-million lines in the LLVM codebase.
Outline

• Why Compilers Optimize Parallel Constructs Poorly
• Old Idea: Parallel IR
• Tapir: A New Twist on an Old Idea
• Evaluation of Tapir
• Conclusion
Outline

• Why Compilers Optimize Parallel Constructs Poorly
• Old Idea: Parallel IR
• Tapir: A New Twist on an Old Idea
• Evaluation of Tapir
• Conclusion
The Compilation Pipeline

C code → Clang → LLVM IR → -O3 → Optimized LLVM IR → CodeGen → EXE
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) {
    double tmp = norm(in, n);
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / tmp;
}
```

__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

**LLVM pipeline**

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

**CilkPlus/LLVM pipeline**

Cilk → Clang → LLVM → -O3 → LLVM → CodeGen → EXE

Front-end translates parallel language constructs.
Effect of Compiling Parallel Code

```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);
}
```

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;
}
```

Clang

```c
int fib(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    if (n < 2) return n;
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_fib(&x, n-1);
    y = fib(n-2);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
    return result;
}
```

```c
void spawn_fib(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = fib(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```

Optimization passes struggle to optimize around these opaque runtime calls.
Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir: A New Twist on an Old Idea
- Evaluation of Tapir
- Conclusion
Let’s embed parallelism directly into the compiler’s intermediate representation (IR)!

### LLVM pipeline

1. **C**
2. **Clang**
3. **LLVM**
4. **-O3**
5. **LLVM**
6. **CodeGen**
7. **EXE**

### CilkPlus/LLVM pipeline

1. **Cilk**
2. **Clang**
3. **LLVM**
4. **-O3**
5. **LLVM**
6. **CodeGen**
7. **EXE**

### A better compilation pipeline

1. **Cilk**
2. **PClang**
3. **PLLVM**
4. **-O3**
5. **PLLVM**
6. **CodeGen**
7. **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]
- HPIR [ZS11, BZS13]
- SPIRE [KJAI12]
- INSPIRE [JPTKF13]

“[LLVMdev] [RFC] Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.)” 
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
A Hard Problem

From the llvm-dev mailing list:

❖ “[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…”

Background: 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;
}
```
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;
}
```
Typical Issues with Parallel IR’s

- Parallel IR is incompatible with existing optimizations or analyses for serial code.
- Parallel IR requires many changes to the compiler.
- Parallel IR offers minimal benefits to optimization.
- Parallel IR is language specific.
- For LLVM, symmetric modeling violates the Linear Assumption: that each block is entered by one predecessor.
Outline

• Why Compilers Optimize Parallel Constructs Poorly
• Old Idea: Parallel IR
• Tapir: A New Twist On an Old Idea
• Evaluation of Tapir
• Conclusion
Tapir extends LLVM IR with **three instructions** that model parallel tasks **asymmetrically**.

**Symmetric CFG**

- **entry**: `br (n < 2), exit, if.else`  
- **if.else**: `parbegin`  
  - `x = fib(n - 1)`  
  - `br join`  
- **join**: `parend`  
  - `add = x + y`  
  - `br exit`  
- **exit**:  
  - `rv = \varphi([n,entry],[add,cont])`  
  - `return rv`

**Tapir CFG**

- **entry**:  
  - `x = alloca()`  
  - `br (n < 2), exit, if.else`  
- **if.else**: `detach det, cont`  
- **det**:  
  - `x0 = fib(n - 1)`  
  - `store x0, x`  
- **cont**:  
  - `reattach cont`  
  - `y = fib(n - 2)`  
  - `sync`  
  - `x1 = load x`  
  - `add = x1 + y`  
  - `br exit`  
- **exit**:  
  - `rv = \varphi([n,entry],[add,cont])`  
  - `return rv`
Tapir extends LLVM IR with **three instructions** that model parallel tasks **asymmetrically**.

**Symmetric CFG**

- **entry** \(\text{br (n < 2), exit, if.else}\)
- **if.else** \(\text{parbegin}\)
  - \(x = \text{fib}(n - 1)\)
  - \(y = \text{fib}(n - 2)\)
- **join** \(\text{parend}\)
  - \(\text{add = x + y}\)
  - \(\text{br exit}\)
- **exit** \(\text{rv = } \varphi([n,\text{entry}],[\text{add,cont}])\)
  - \(\text{return rv}\)

**Tapir CFG**

- **entry** \(\text{x = alloca()}\)
  - \(\text{br (n < 2), exit, if.else}\)
- **if.else** \(\text{detach det, cont}\)
  - \(x0 = \text{fib}(n - 1)\)
  - \(\text{store } x0, x\)
- **det** \(\text{reattach cont}\)
  - \(\text{y = fib(n - 2)}\)
  - \(\text{sync}\)
  - \(\text{x1 = load } x\)
  - \(\text{add = x1 + y}\)
  - \(\text{br exit}\)
- **cont** \(\text{rv = } \varphi([n,\text{entry}],[\text{add,cont}])\)
  - \(\text{return rv}\)

**Violates Lineage Assumption!**
Tapir's Pipeline

With few changes, LLVM’s existing optimization passes can optimize across parallel control flow.

Tapir adds three instructions to LLVM IR that express fork-join parallelism.
What does Tapir do to adapt existing optimizations?

- Common-subexpression elimination: no change
- Loop-invariant-code motion: minor change
- Tail-recursion elimination: minor change

Tapir also enables new parallel optimizations, such as:

- Unnecessary-synchronization elimination
- Puny-task elimination
- Parallel-loop scheduling (new pass combined with existing unrolling and vectorization passes)
Outline

• Why Compilers Optimize Parallel Constructs Poorly
• Old Idea: Parallel IR
• Tapir and Why It Works
• Evaluation of Tapir
• Conclusion
## Code Complexity of Tapir/LLVM

<table>
<thead>
<tr>
<th>Compiler component</th>
<th>LLVM 3.8 (lines)</th>
<th>Tapir/LLVM (lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>148,588</td>
<td>900</td>
</tr>
<tr>
<td>Memory behavior</td>
<td>10,549</td>
<td>588</td>
</tr>
<tr>
<td>Optimizations</td>
<td>140,842</td>
<td>255</td>
</tr>
<tr>
<td>Code generation</td>
<td>205,378</td>
<td>145</td>
</tr>
<tr>
<td>Parallelism lowering</td>
<td>0</td>
<td>1,903</td>
</tr>
<tr>
<td>New parallel optimizations</td>
<td>0</td>
<td>1,332</td>
</tr>
<tr>
<td>Other</td>
<td>2,854,566</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,359,893</strong></td>
<td><strong>5,123</strong></td>
</tr>
</tbody>
</table>

1,888

25
Work-Efficiency Improvement

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

Preliminary results
Speedup Improvement

Test machine: Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM
Preliminary results
Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir and Why It Works
- Evaluation of Tapir
- Conclusion
Normalizing 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 compiled with LLVM: T_S = 0.397 s

Compiled with Tapir, running time on 1 core: T_1 = 0.400 s

Compiled with Tapir, running time on 18 cores: T_{18} = 0.157 s

Great work efficiency: T_S / T_1 ~ 0.990
Status of Tapir

- We implemented Tapir in LLVM, along with a prototype Cilk front-end and a pass for lowering Tapir to Cilk runtime calls.
- Tapir is currently in use by over 120 MIT students.
- Out Tapir implementation appears to exhibit fewer bugs than GCC, ICC, or CilkPlus/LLVM when compiling Cilk codes.
- We have a provably good determinacy-race detector for Tapir programs, which we used to debug code transformations.
- We’re continuing to explore new optimizations.
- Try Tapir yourself!
  Email me at wmoses@mit.edu