Tapir: Embedding Fork-Join Parallelism into LLVM IR

William S. Moses  |  Tao B. Schardl  |  Charles E. Leiserson
MIT Computer Science and Artificial Intelligence Laboratory
July 6, 2016
Writing Fast Code

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);
}
Writing Fast Code

Declaring the return value to be `const` allows the compiler to move the call to `norm` out of the loop.

```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);
}
```
__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);
}

Serial running time: 0.340s

Wow! This is fun! What else can we do?

Declaring the return value to be const allows the compiler to move the call to norm out of the loop.
We have multiple cores!
Let’s make it run in parallel!

Intel Core i7-5960X
Writing Fast 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);
}
```

Serial running time: 0.340s
Writing Half-Fast 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);
}
```

Serial running time: 0.340s
1-core running time: 8532.316s
Writing Half-Fast 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);
}
```

Serial running time: 0.340s
1-core running time: 8532.316s
2-core running time: 4539.138s
Writing Half-Fast 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);
}
```

Serial running time: 0.340s
1-core running time: 8532.316s
2-core running time: 4539.138s

~25,000 cores needed to get parallel speedup!
What happened?
LLVM/Clang pipeline

`norm.c` → LLVM IR → Optimized IR → Executable

- Clang/ Frontend
- Optimization
- CodeGen
__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);
}
__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);
}
__attribute__((const))
double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in, int n)
int tmp = norm(in, n);
for(int i = 0; i < n; ++i)
    out[i] = in[i] / tmp;
```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);
```
void normalize(double *restrict out, const double *restrict in, int n)
    int* close[3] = { n, in, out };
    __cilk_rts_for(cilk_for_helper, close, 0, n);
}
void cilk_for_helper (int start, int end) {
    int n = *(int*)close;
    int* in = *(int**)(close+1);
    int* out = *(int**)(close+2);
    out[i] = in[i] / norm(in, n);
}
`void normalize(double *restrict out, const double *restrict in, int n)`

`int* close[3] = { n, in, out };`

`__cilk_rts_for(cilk_for_helper, close, 0, n);`

`void cilk_for_helper (int start, int end) {`

`int n = *(int*)close;`
`int* in = *(int**)(close+1);`
`int* out = *(int**)(close+2);`

`out[i] = in[i] / norm(in, n),`
void normalize(double *restrict out, const double *restrict in, int n)
    int* close[3] = { n, in, out };__cilk_rts_for(cilk_for_helper, close, 0, n);
}
void cilk_for_helper (int start, int end) {
    int n = *(int*)close;
    int* in = *(int**)(close+1);
    int* out = *(int**)(close+2);
    for (int i = start; i < end; i++)
        out[i] = in[i] / norm(in, n);
Another example

```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;
}
```

```
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;
}
```

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);
}

How is an optimization pass supposed to figure its way around this mess of opaque system calls?
Parallelism Should Not Be “Lowered” in the Front End

- **Opaque runtime calls** do not allow optimization passes on the IR to do their jobs on **parallel constructs**.

- Unoptimized parallel performance can be **much worse** than optimized serial performance.

- Optimizing parallel constructs in the front end **duplicates work** that the IR optimization passes are already doing for serial constructs.

- Optimizing parallel constructs in the front end is **error prone**.
Let’s put parallelism in the IR!
Comments on the idea from the llvm-dev mailing list [1]

- “[D]efining a parallel IR (with first class parallelism) is a research topic....”
- “[I]t is not an easy problem.”
- “[P]arallelism is a very invasive concept and introducing it 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.”

Prior Work

SPIRE [2, 3], INSPIRE [4], HPIR [5].


Typical Issues

- Parallel IR is language specific.
- Parallel IR offers minimal benefits to optimization.
- Parallel IR is incompatible with existing serial optimizations.
- Parallel IR requires many changes to the compiler.
Tapir: Task-based Asymmetric Parallel IR

- Tapir is an IR that exposes logical parallelism.
- Tapir enables existing serial optimization passes to operate across parallel control with few or no changes.
- Each of Tapir’s “asymmetric” parallelism constructs can also be viewed as a serial construct having ordinary serial semantics.
- Only 5123 LOC were needed to implement Tapir/LLVM.
- Tapir/LLVM includes a provably good determinacy race detector both to verify code transformations and to debug buggy source code.
Overhead Comparison on 1 Worker

Cilk and Intel Sample Benchmarks
Minimum of 5 runs on AWS c4.8xlarge
Overhead Comparison on 8 Workers

Parallel Overhead (Smaller is Better)

- Cholesky
- FFT
- Mandelbrot
- NQueens
- QSort
- RectMul

Cilk and Intel Sample Benchmarks
Minimum of 5 runs on AWS c4.8xlarge
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
LLVM Overview

Control flow graph (CFG)

- Basic blocks contain instructions.
- Values are stored in registers.
- Edges create control flow.

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

Invariants

- **Lineage assumption**: Only one predecessor of a basic block actually executes.

- A basic block can only access a register if the register *dominates* the basic block.

```
entry
   br (n < 2), exit, if.else

if.else
   x = fib(n - 1)
   y = fib(n - 2)
   add = x + y
   br join

exit
   rv = φ([n,entry],[add,join])
   return rv
```
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
Naive Parallel IR

**Idea**
Modify the program’s control-flow graph to represent logically parallel tasks symmetrically.

```c
int fib(int n) {
    if (n < 2) return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```
Initial Attempt: Naive Representation

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

**Major issue**

- LLVM’s lineage assumption breaks.
- Values from **all** predecessors must be available at the join.
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
```c
int fib(int n) {
    if (n < 2) return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```
Preservation of LLVM's lineage assumption

- The continuation has no access to parallel registers (as at the low level).
- LLVM's invariants are maintained.

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

- Tapir introduces three new opcodes into LLVM’s IR: detach, reattach, and sync.
- Tapir simultaneously represents the serial and parallel semantics of the program.
Semantics of Tapir

- The successors of a detach terminator are the *detached block* and *continuation* and may run in parallel.
- A *detached CFG* contains all blocks between a detached block and its corresponding reattach.

```
x = alloca()
br (n < 2), exit, if.else
```

```
x0 = fib(n - 1)
*x = x0
reattach cont
```

```
y = fib(n - 2)
sync
add = *x + y
br exit
```

```
rv = ϕ([n,entry],[add,join])
return rv
```
Semantics of Tapir (continued)

- 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.
- Execution after a `sync` ensures that all detached CFG’s in scope have completed execution.
Parallel Loops in Tapir

- Identical to serial loops, except with the body detached.

```c
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);
}
```
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
Maintaining Correctness

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

```
x = alloca()
br (n < 2), exit, if.else

if.else

detach det, cont

x0 = fib(n - 1)
*x = x0
reattach cont

y = fib(n - 2)
    sync
    add = *x + y
    br exit

cont

everything

exit

rv = φ([n,entry],[add,join])
return rv
```
Maintaining Correctness

**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.
Maintaining Correctness

**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.

```c
x = alloca()
br (n < 2), exit, if.else
detach det, cont
x0 = fib(n - 1)
*x = x0
reattach cont
y = fib(n - 2)
sync
add = *x + y
br exit
rv = \phi([n,entry],[add,join])
return rv
```
**Maintaining Correctness**

**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* start, 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

```
(b == end), part, end
mid = partition(start, end)
swap(end, mid)
det
qsort(mid, end)
cont
sync
return
```

```
b.phi = φ([begin, entry], [mid, cont])
(b == end), part, end
mid = partition(start, end)
swap(end, mid)
det
qsort(mid, end)
cont
sync
return
```
### Lines of Code for LLVM-3.8

<table>
<thead>
<tr>
<th>Compiler component</th>
<th>LLVM-3.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>148,558</td>
</tr>
<tr>
<td>Memory</td>
<td>10,549</td>
</tr>
<tr>
<td>Optimizations</td>
<td>140,843</td>
</tr>
<tr>
<td>CodeGen</td>
<td>205,378</td>
</tr>
<tr>
<td>Cilk ABI Lowering</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,359,893</strong></td>
</tr>
</tbody>
</table>
## Lines of Code for LLVM-3.8 versus Tapir

<table>
<thead>
<tr>
<th>Compiler component</th>
<th>LLVM-3.8</th>
<th>Tapir</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>148,558</td>
<td>900</td>
</tr>
<tr>
<td>Memory</td>
<td>10,549</td>
<td>588</td>
</tr>
<tr>
<td>Optimizations</td>
<td>140,843</td>
<td>306</td>
</tr>
<tr>
<td>CodeGen</td>
<td>205,378</td>
<td>145</td>
</tr>
<tr>
<td>Cilk ABI Lowering</td>
<td>0</td>
<td>3,184</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>3,359,893</td>
<td>5,123</td>
</tr>
</tbody>
</table>
Lines of Code for LLVM-3.8 versus Tapir

<table>
<thead>
<tr>
<th>Compiler component</th>
<th>LLVM-3.8</th>
<th>Tapir</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>148,558</td>
<td>900</td>
</tr>
<tr>
<td>Memory</td>
<td>10,549</td>
<td>588</td>
</tr>
<tr>
<td>Optimizations</td>
<td>140,843</td>
<td>306</td>
</tr>
<tr>
<td>CodeGen</td>
<td>205,378</td>
<td>145</td>
</tr>
<tr>
<td>Cilk ABI Lowering</td>
<td>0</td>
<td>3,184</td>
</tr>
<tr>
<td>Total</td>
<td>3,359,893</td>
<td>5,123</td>
</tr>
</tbody>
</table>

1,939
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
Tapir Speedups — 1 Worker (higher is better)

Cilk and Intel Sample Benchmarks
Min of 5 runs on AWS c4.8xlarge
Tapir Speedups — 8 Workers (higher is better)

Cilk and Intel Sample Benchmarks
Min of 5 runs on AWS c4.8xlarge
Outline

- LLVM Overview
- Naive Parallel IR
- Tapir Syntax and Semantics
- Optimization Passes
- Evaluation
- Conclusion
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.
Thank You!
Questions?