Leveraging LLVM to Optimize Parallel Programs

William S. Moses

2017 US LLVM Developers Meeting
October 18, 2017
Running LLVM Optimizations on Parallel Code

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

```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
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
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

1-core running time: 2600.287s
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);
}
```

```
__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
- 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

```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);
}

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

void normalize(double *restrict out, const double *restrict in, int n) {
    struct args_t args = { out, in, n };
    __cilkrts_cilk_for(normalize_helper, args, 0, n);
}

void normalize_helper(struct args_t args, int i) {
    double *out = args.out;
    double *in = args.in;
    int n = args.n;
    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.

```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;
}

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

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: $T_S / T_1 = 97\%$
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.
Reasoning About a Tapir CFG

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.
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 = detach
        { search(0,n/2); }
    int y = search(n/2,n);
    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 = detach
        { search(0,n/2); }
    int y = search(n/2,n);
    sync;
    return x + y;
}
```
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
<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>Cholesky</td>
<td>Cholesky decomposition</td>
</tr>
<tr>
<td></td>
<td>FFT</td>
<td>Fast Fourier transform</td>
</tr>
<tr>
<td></td>
<td>NQueens</td>
<td>n-Queens solver</td>
</tr>
<tr>
<td></td>
<td>QSort</td>
<td>Hoare quicksort</td>
</tr>
<tr>
<td></td>
<td>RectMul</td>
<td>Rectangular matrix multiplication</td>
</tr>
<tr>
<td></td>
<td>Strassen</td>
<td>Strassen matrix multiplication</td>
</tr>
<tr>
<td>Intel</td>
<td>AvgFilter</td>
<td>Averaging filter on an image</td>
</tr>
<tr>
<td></td>
<td>Mandel</td>
<td>Mandelbrot set computation</td>
</tr>
<tr>
<td>PBBS</td>
<td>CHull</td>
<td>Convex hull</td>
</tr>
<tr>
<td></td>
<td>detBFS</td>
<td>BFS, deterministic algorithm</td>
</tr>
<tr>
<td></td>
<td>incMIS</td>
<td>MIS, incremental algorithm</td>
</tr>
<tr>
<td></td>
<td>incST</td>
<td>Spanning tree, incremental algorithm</td>
</tr>
<tr>
<td></td>
<td>kdTree</td>
<td>Performance test of a parallel k-d tree</td>
</tr>
<tr>
<td></td>
<td>ndBFS</td>
<td>BFS, nondeterministic algorithm</td>
</tr>
<tr>
<td></td>
<td>ndMIS</td>
<td>MIS, nondeterministic algorithm</td>
</tr>
<tr>
<td></td>
<td>ndST</td>
<td>Spanning tree, nondeterministic algorithm</td>
</tr>
<tr>
<td></td>
<td>parallelSF</td>
<td>Spanning-forest computation</td>
</tr>
<tr>
<td></td>
<td>pRange</td>
<td>Compute ranges on a parallel suffix array</td>
</tr>
<tr>
<td></td>
<td>radixSort</td>
<td>Radix sort</td>
</tr>
<tr>
<td></td>
<td>SpMV</td>
<td>Sparse matrix-vector multiplication</td>
</tr>
</tbody>
</table>
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, 2.9 GHz, 60 GiB DRAM
What Else Does A Parallel IR Buy Us?

William S. Moses
George Stelle
Jiahao Li

Dougie Kogut
Bojan Serafimov
Parallel-Specific Optimizations

To ensure reasonable performance, parallel frameworks implement parallel-specific optimizations.
Example Opt: Coarsening

- Combine detached statements to overcome the overhead of running in parallel

```c
void scale(double *restrict A, double s, int n) {
    parallel_for (int i = 0; i < n; i++) {
        A[i] *= s;
    }
}

void scale(double *restrict A, double s, int n) {
    parallel_for (int i = 0; i < n; i+=4) {
        for (int i2 = 0; i2 < 4; i2++) {
            A[i+i2] *= s;
        }
    }
}
```
Example Opt: Coarsening

- Combine detached statements to overcome the overhead of running in parallel

```c
void scale(double *restrict A, double s, int n) {
    parallel_for (int i = 0; i < n; i++) {
        A[i] *= s;
    }
}
```

```c
void scale(double *restrict A, double s, int n) {
    parallel_for (int i = 0; i < n; i+=4) {
        for (int i2 = 0; i2 < 4; i2++) {
            A[i+i2] *= s;
        }
    }
}
```
Example Opt: Task Elimination

- If you have a detached task immediately followed by a sync, remove the detach.

```c
void foo() {
    detach bar();
    detach baz();
    sync;
}
```

```c
void foo() {
    detach bar();
    baz();
    sync;
}
```

Sounds trivial, but especially useful for OpenMP!
Example Opt: Task Elimination

```c
void fib(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

Linguistically OpenMP tasks encourages users to write code that needs this optimization!
Case Study: Task Elimination

Fib Runtime

<table>
<thead>
<tr>
<th></th>
<th>Time (s), less is better</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Opt</td>
<td>1.2</td>
</tr>
<tr>
<td>Task Elim</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Time (s), less is better
Case Study: Task Elimination

Fib Runtime

- **No Opt**: ~3x longer runtime compared to **Task Elim**
- **Task Elim**: More efficient, shorter runtime

Time (s), less is better

~3x faster in Task Elimination
Parallel Optimizations Today

- Every parallel framework today is independent, requiring large amounts of code duplication.

- Duplication from framework to framework

- Duplication from low level (i.e. LICM in LLVM) to high level
Parallel Pipeline Today

<table>
<thead>
<tr>
<th>Cilk Frontend</th>
<th>OpenMP Frontend</th>
<th>Halide Frontend</th>
<th>Weld Frontend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk Parallel Optimizations (shrink wrap)</td>
<td>OMP Parallel Optimizations (strip mine)</td>
<td>Halide Parallel Optimizations (scheduling)</td>
<td>Weld Parallel Optimizations, LICM</td>
</tr>
<tr>
<td>LLVM w/ Cilk Runtime Calls</td>
<td>LLVM w/ OpenMP Runtime Calls</td>
<td>LLVM w/ Halide Runtime Calls</td>
<td>LLVM w/ Weld Runtime Calls</td>
</tr>
<tr>
<td>Cilk Runtime</td>
<td>OpenMP Runtime</td>
<td>Halide Runtime</td>
<td>Weld Runtime</td>
</tr>
</tbody>
</table>
Rhino: The Parallel Compiler Dream

- Tapir is a nice way of representing and working with parallel programs
- Use Tapir as a common parallel intermediate representation for various parallel frontends and backends

Benefits

- Enable cross-framework compilation
- Have one set of common parallel optimizations that can be shared by all
Rhino: The Parallel Compiler Dream

- Cilk
- OpenMP
- CUDA
- Halide
- Weld
- Tapir/LLVM
- Common Parallel Optimizations
- Cilk Runtime
- OpenMP Runtime
- PTX ISA
- Polly
Rhino: The Parallel Compiler Dream

- Cilk
- OpenMP
- CUDA
- Halide
- Weld

Tapir/LLVM
Common Parallel Optimizations

- Cilk Runtime
- OpenMP Runtime
- PTX ISA

Completely Done
Partially Done

Polly
Parallel Runtime Choice

Examples from Barcelona OpenMP benchmark suite
Parallel Runtime Choice

Examples from Barcelona OpenMP benchmark suite
Takeaways

❖ With little modification, the compiler can do a lot of things to make your parallel programs faster
  ❖ Run (serial) optimizations on parallel code
  ❖ Build and share parallel optimizations
  ❖ Mix-and-match parallel runtimes
❖ Ongoing development (bug fixes, new optimizations, etc).
❖ Available on GitHub!
  https://github.com/wsmoses/Parallel-IR.git
Backup Slides!
When designing parallel optimization passes, we ran into the issue where we couldn’t represent the optimized code inside Tapir!

```c
void B() {
    detach B1();
    B2();
    sync;
}

void main() {
    detach A();
    detach B1();
    B2();
    sync;
    C();
    sync;
}
```

A is parallel to C

A must execute before C
Obstacle

- Tapir assumes detaches/syncs (or specifically detaches/syncs) are scoped to a function, whereas we need something more precise.

- How much more precise?
  - Provide a sync to individual detaches?
  - Provide a sync to groups of detaches?
Idea 1: Individualized Sync

- Permit synchronization of specific parallel statements
- Most general model

```c
void main() {
    a = detach A();
    b = detach B1();
        B2();
    sync a;
        C();
    sync b;
}
```
Idea 1: Individualized Sync

- Representing arbitrary sets to sync dramatically increases complexity
- Generality of model restricts possible runtimes
- Harder to optimize! (Previously could assume that a detached statement no longer can alias after a sync)

```c
f = detach foo();
\phi = {};
for (int i = \emptyset; i < n; ++i) {
    \gamma_0 = phi [(\emptyset, entry), (\gamma_1, loop)];
    a = detach A(i);
    \gamma_1 = union [ \gamma_0, a ];
}
\gamma_2 = phi [(\emptyset, entry), (\gamma_1, loop)];
sync \gamma_2;
bar();
```
Idea 2: Scoped Sync

- Represent parallelism in nested parallel regions
- A sync now acts on all detaches in that region
- Doesn’t change runtime compatibility
- Maintain guarantee that no detaches (now in the region) continue after a sync

This implies that all parallel optimizations developed for vanilla Tapir work, except using a parallel region scope instead of function scope
Idea 2: Individualized Sync

```c
void main() {
    detach A();
    parallel_region {
        detach B1();
        B2();
        sync;
    }
    C();
    sync;
}
```
**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.
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.
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`. 

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

if.else
  detach det, cont

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

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

exit
  rv = \Phi([n,entry],[add,cont])
  return rv
```
Valid serial passes cannot create race bugs.

Most of LLVM’s existing serial passes “just work” on parallel code.
Example Optimization: Fusion

- Existing code written in parallel frameworks can leverage polyhedral optimizations such as loop fusion or tiling with no extra effort.

```c
void add(double * A, double * B, double * C, int n) {
    parallel_for (int i = 0; i < n; i++) {
        A[i] += B[i];
    }
    parallel_for (int i = 0; i < n; i++) {
        A[i] += C[i];
    }
}
```
Example Optimization: Fusion

- Existing code written in parallel frameworks can leverage polyhedral optimizations such as loop fusion or tiling with no extra effort

```c
void add(double * A, double * B, double * C, int n) {
    parallel_for (int i = 0; i < n; i++) {
        A[i] += B[i];
    }
    parallel_for (int i = 0; i < n; i++) {
        A[i] += C[i];
    }
}
```

```c
void add(double * A, double * B, double * C, int n) {
    parallel_for (int i = 0; i < n; i++) {
        A[i] += B[i];
        A[i] += C[i];
    }
}
```

~2x