Leveraging LLVM to optimize parallel programs

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

Parlay Meeting
September 29, 2017
Modern Compilers Don’t Understand Parallel Code
Modern Compilers Don’t Understand Parallel Code

Intel® Threading Building Blocks
C and C++ template library for creating high performance, scalable parallel applications

DESIGNED FOR NVIDIA CUDA™
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
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, \( n = 64 \text{M} \).

*Original serial running time: \( T_S = 0.312 \text{ s} \)
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: 180.657s

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);
}
```

The code shown above is compiled with the `-O3` flag. Without optimization, the code is already efficient. However, when compiled with `-O3`, the `normalize` function becomes even more efficient due to the optimization of the `norm` function and the loop. The optimized code eliminates the redundant calculation of the norm inside the loop.
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

```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;
}
```

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;
}
```

```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);
}
```
Tapir: Task-based Asymmetric Parallel IR

Cilk Plus/LLVM pipeline

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

Tapir/LLVM pipeline

Cilk → PClang → Tapir → -O3 → Tapir → CodeGen → EXE

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.
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
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 Loops in Tapir

Parallel loop resembles a serial loop with a detached body.

The sync waits on a dynamic set of detached sub-CFG's.
Most Serial Optimizations
“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;
}
```
Normalizing a Vector in Parallel with Tapir

Cilk code for normalize()

```
__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: 
$T_S / T_1 = 97\%$
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>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>105,995</td>
<td>943</td>
</tr>
<tr>
<td>Memory behavior</td>
<td>21,788</td>
<td>445</td>
</tr>
<tr>
<td>Optimizations</td>
<td>152,229</td>
<td>380</td>
</tr>
<tr>
<td>Parallelism lowering</td>
<td>0</td>
<td>3,782</td>
</tr>
<tr>
<td>Other</td>
<td>3,803,831</td>
<td>460</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>4,083,843</td>
<td>6,010</td>
</tr>
</tbody>
</table>
Rhino: The Parallel Compiler Dream

- There is ample opportunity for parallel-specific optimizations

- Scheduling has potential for big boosts!
In order to maintain some level of parallel performance, each high-level framework has its own set of parallel optimizations.

Done at a high-level, this requires both duplicating low-level code (i.e. LICM) and duplicating from framework to framework.
Rhino: The Parallel Compiler Dream

TC -> Tapir/LLVM
TVM -> Tapir/LLVM
CUDA -> Tapir/LLVM
Halide -> Tapir/LLVM
Weld -> Tapir/LLVM

Common Parallel Optimizations

Cilk Runtime
OpenMP Runtime
PTX ISA

PPCG
Rhino: The Parallel Compiler Dream

- Cilk
- OpenMP
- CUDA
- Halide
- Weld
- PPCG

Common Parallel Optimizations

- Tapir/LLVM

- Cilk Runtime
- OpenMP Runtime
- PTX ISA
Conclusion

- Tapir/LLVM enables existing serial optimizations to operate on fork-join parallel code, with minimal modification
- Using Tapir as a common parallel framework, you allow high-level parallel frameworks to share optimizations and allow library transformation
- Ongoing development (bug fixes, new optimizations, etc).
- Available on GitHub!
  https://github.com/wsmoses/Parallel-IR.git