Leveraging LLVM to optimize parallel programs



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

Parlay Meeting
September 29, 2017

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Modern Compilers Don’t Understand Parallel Code

OpenMP

CILK ARTS

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

```
__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

```
__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

```
__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

```
__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

```
__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.

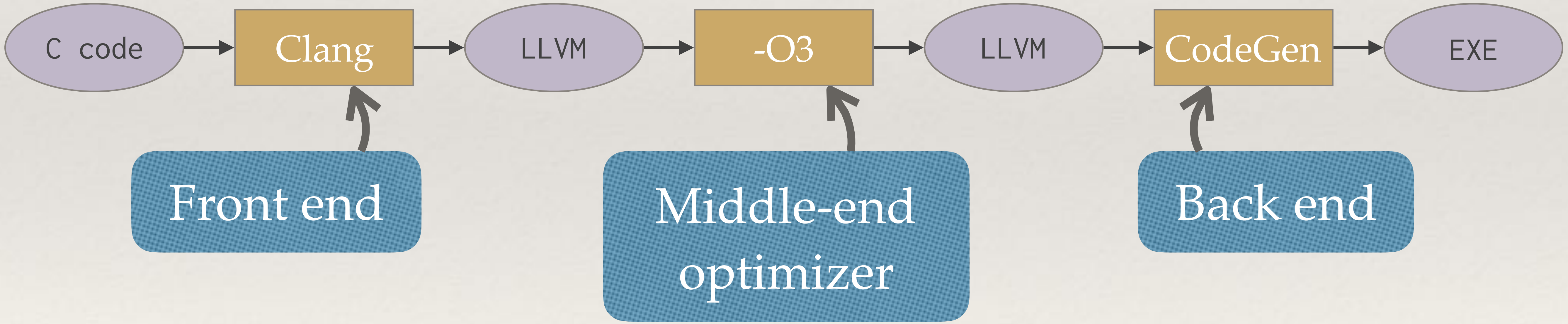
Original serial running time: $T_S = 0.312$ s

18-core running time: 180.657s

1-core running time: 2600.287s

What Happened?

The LLVM Compilation Pipeline



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);
}
```

-O3

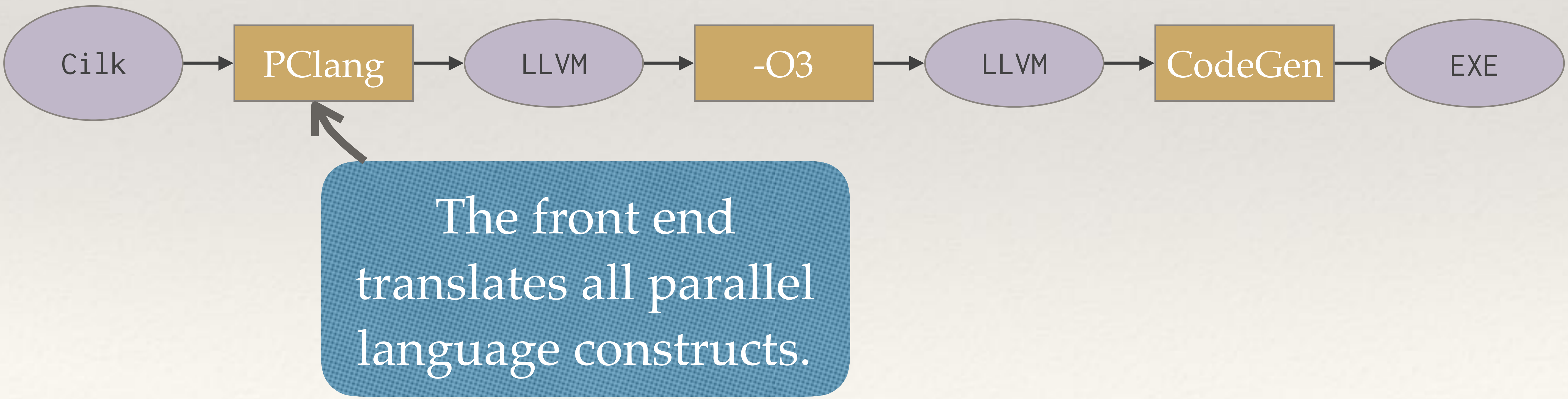
```c
__attribute__((const)) double norm(const double *A, int n);

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

Cilk Plus/LLVM pipeline



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);
}
```

PClang

```
__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

```
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;
}
```

PClang

```
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);
}
```

Optimization passes struggle to optimize around these opaque runtime calls.

Tapir: Task-based Asymmetric Parallel IR

Cilk Plus/LLVM pipeline

Tapir/LLVM pipeline

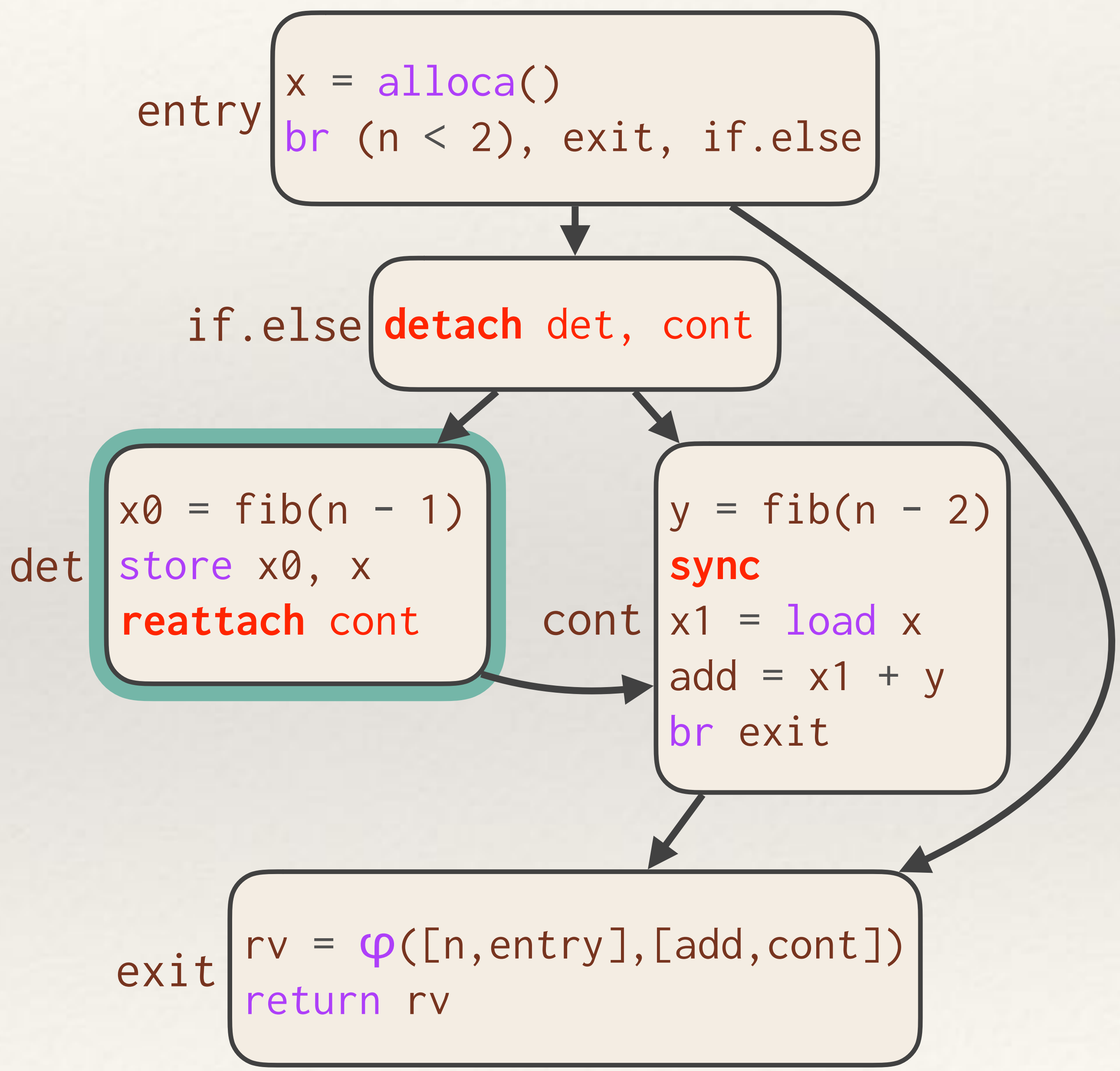
Tapir adds **three instructions** to LLVM IR that encode **fork-join parallelism**.

With **few changes**, LLVM's **existing optimizations and analyses** work on **parallel code**.

Tapir 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

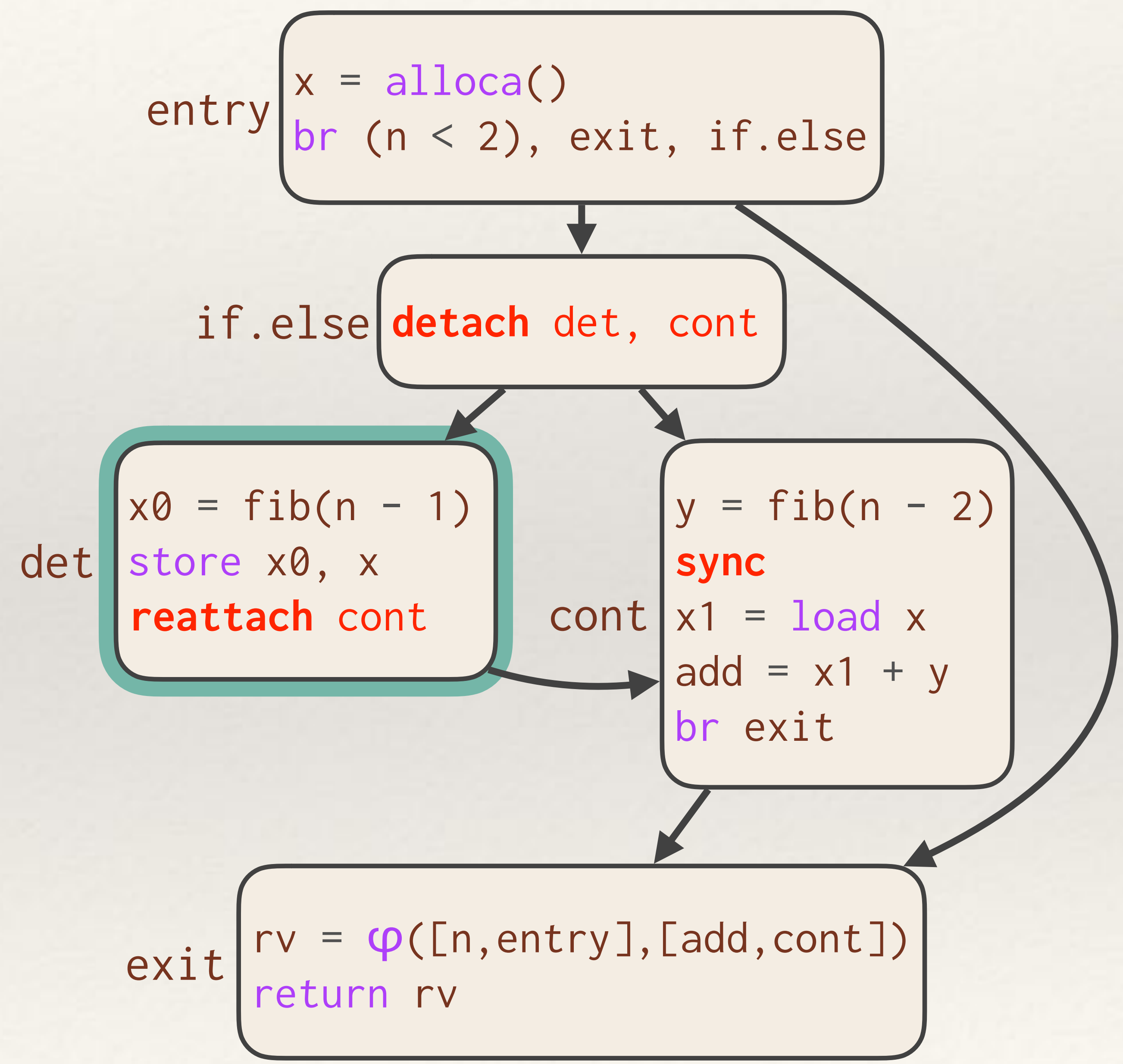
Tapir CFG



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

Tapir CFG

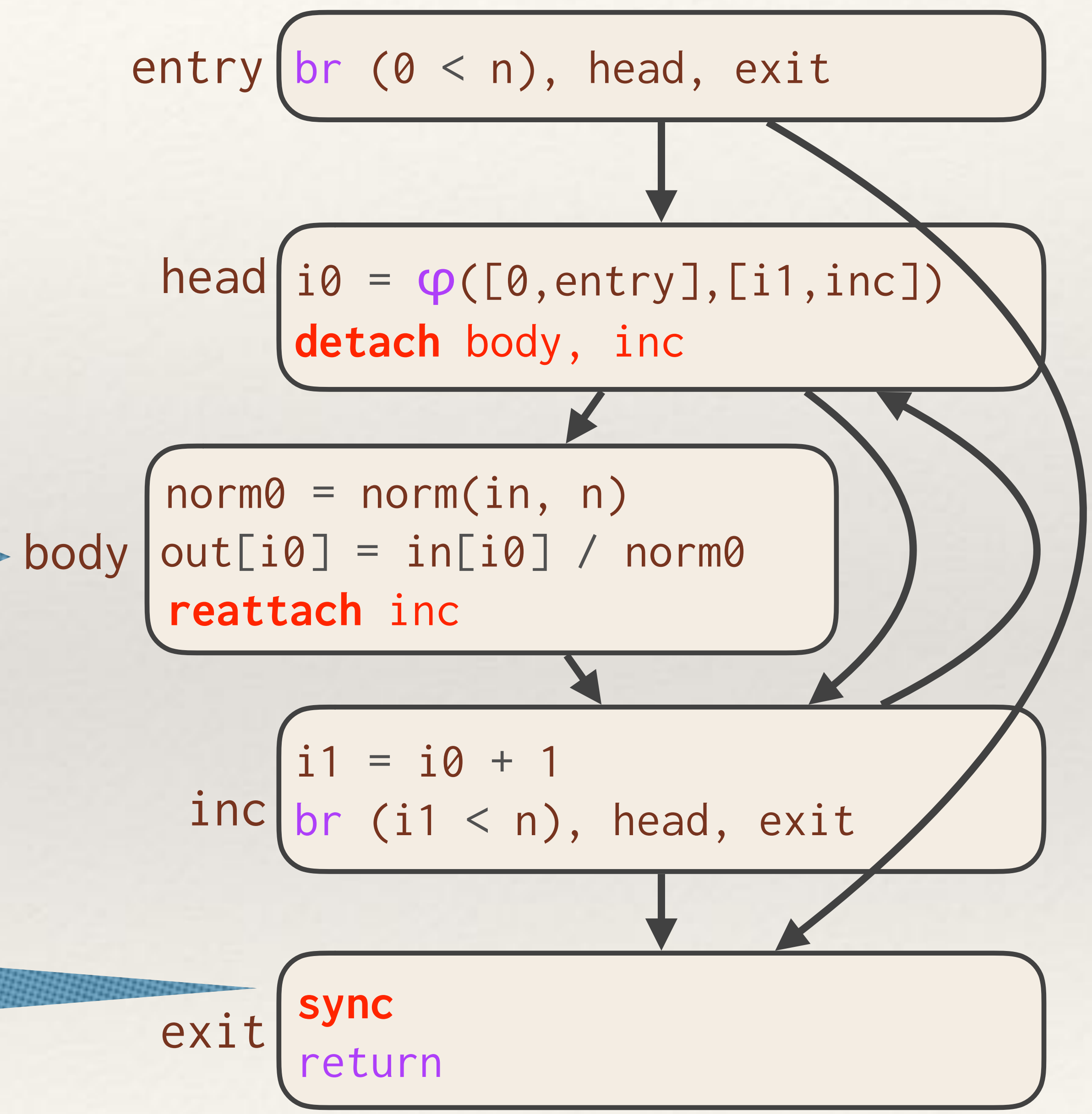


Parallel Loops in Tapir

```
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.

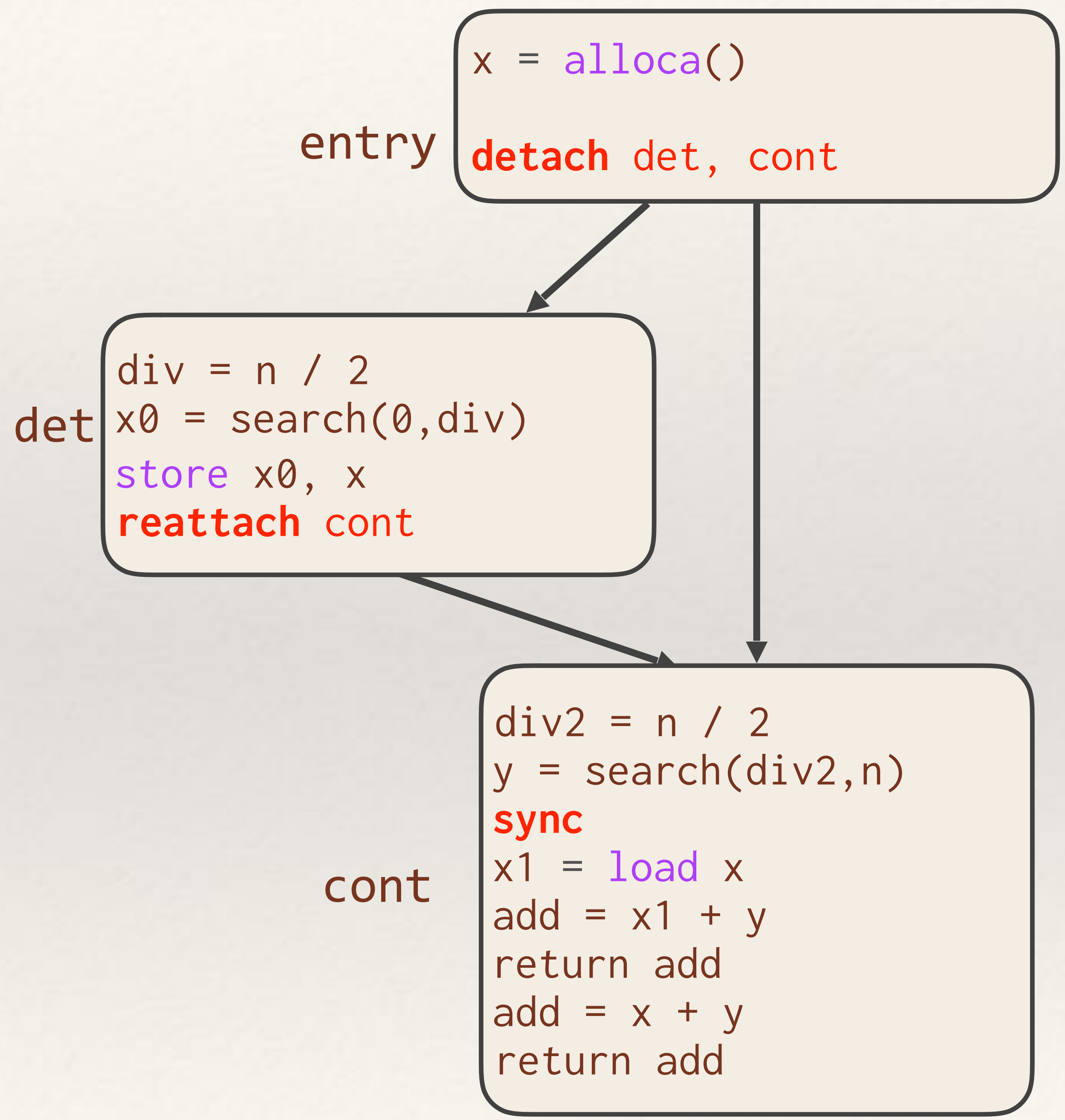


Most Serial Optimizations “Just Work” on Parallel Code!

Case Study: Common Subexpression Elimination

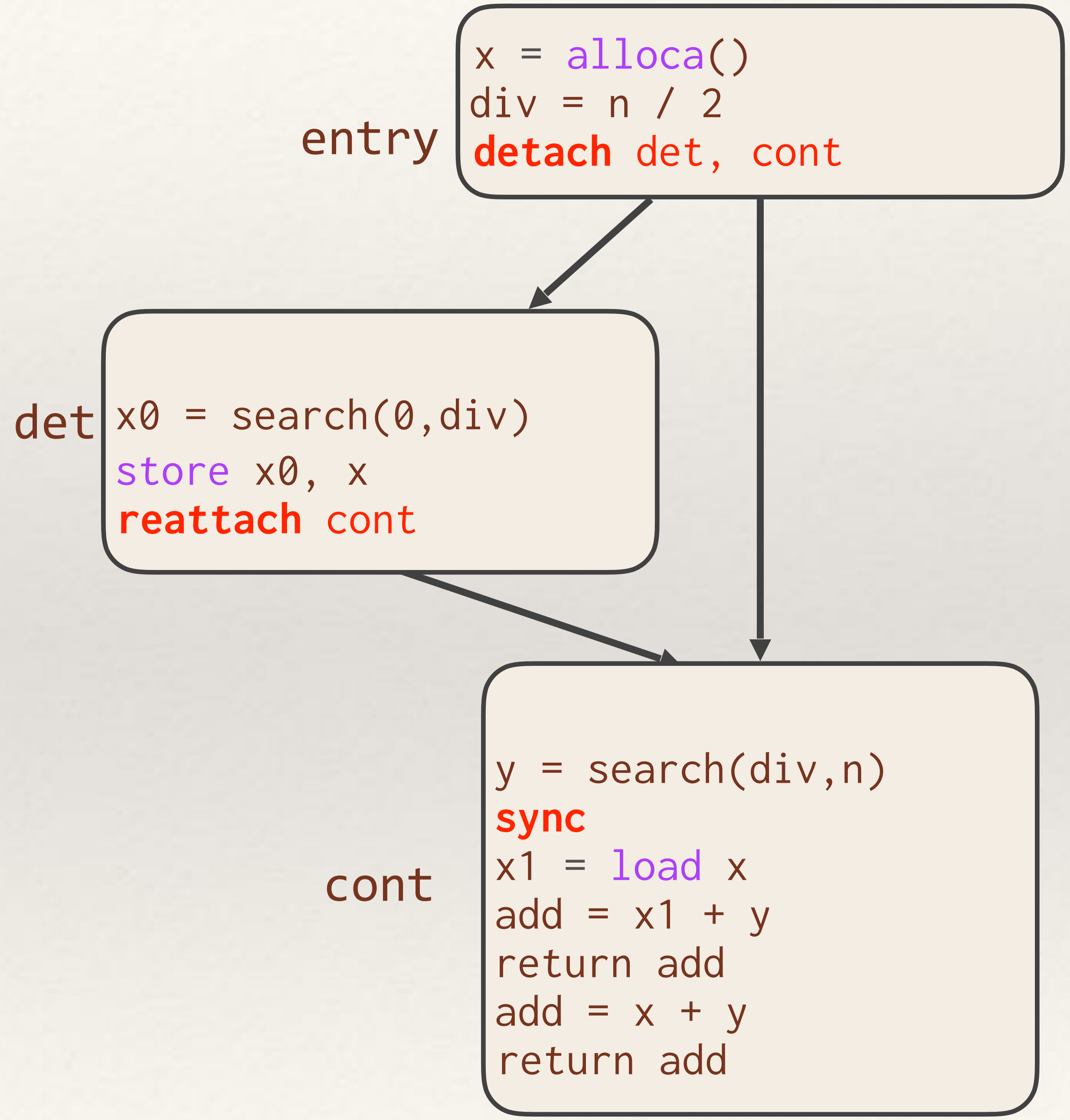
- CSE "just works."
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for **detach**/ **reattach**.

```
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`.




```
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);
}
```

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

Running time of original serial code: $T_S = 0.312$ s

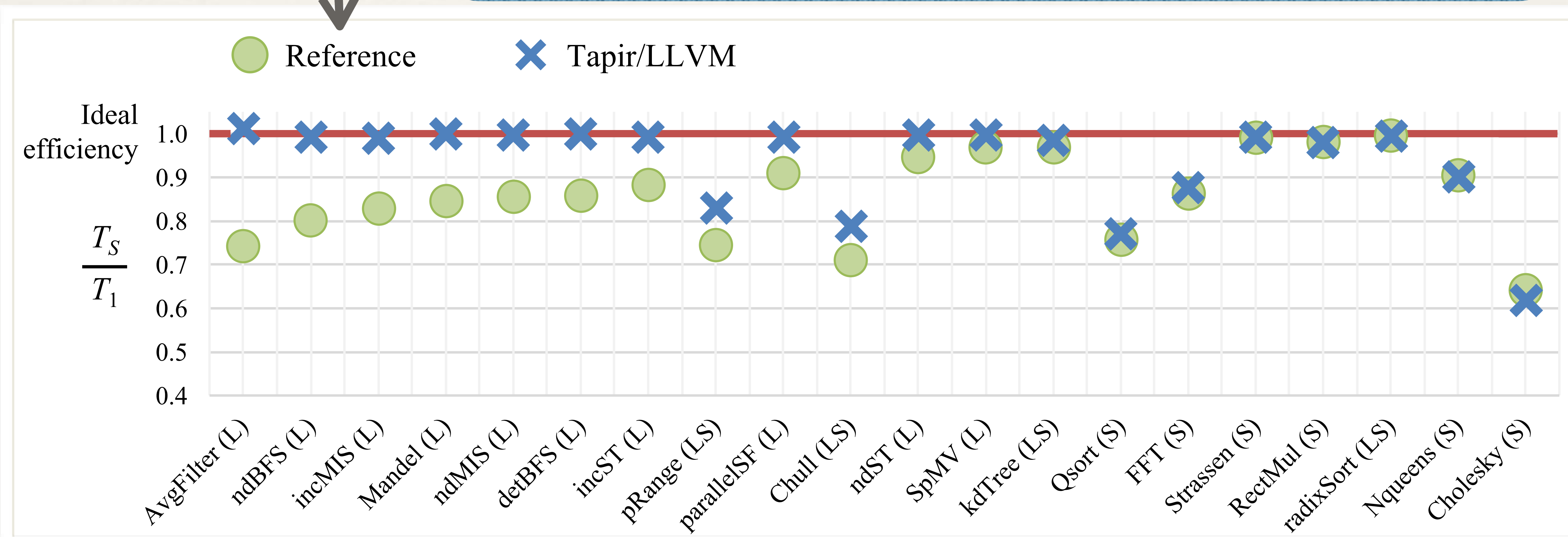
Compiled with Tapir/LLVM, running time on 1 core: $T_1 = 0.321$ s

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

Compiler component	*LLVM 4.0svn (lines)*	*Tapir/LLVM (lines)*
Instructions	105,995	943
Memory behavior	21,788	445
Optimizations	152,229	380
Parallelism lowering	0	3,782
Other	3,803,831	460
Total	4,083,843	6,010

1,768

Rhino: The Parallel Compiler Dream

Parallel Optimizations

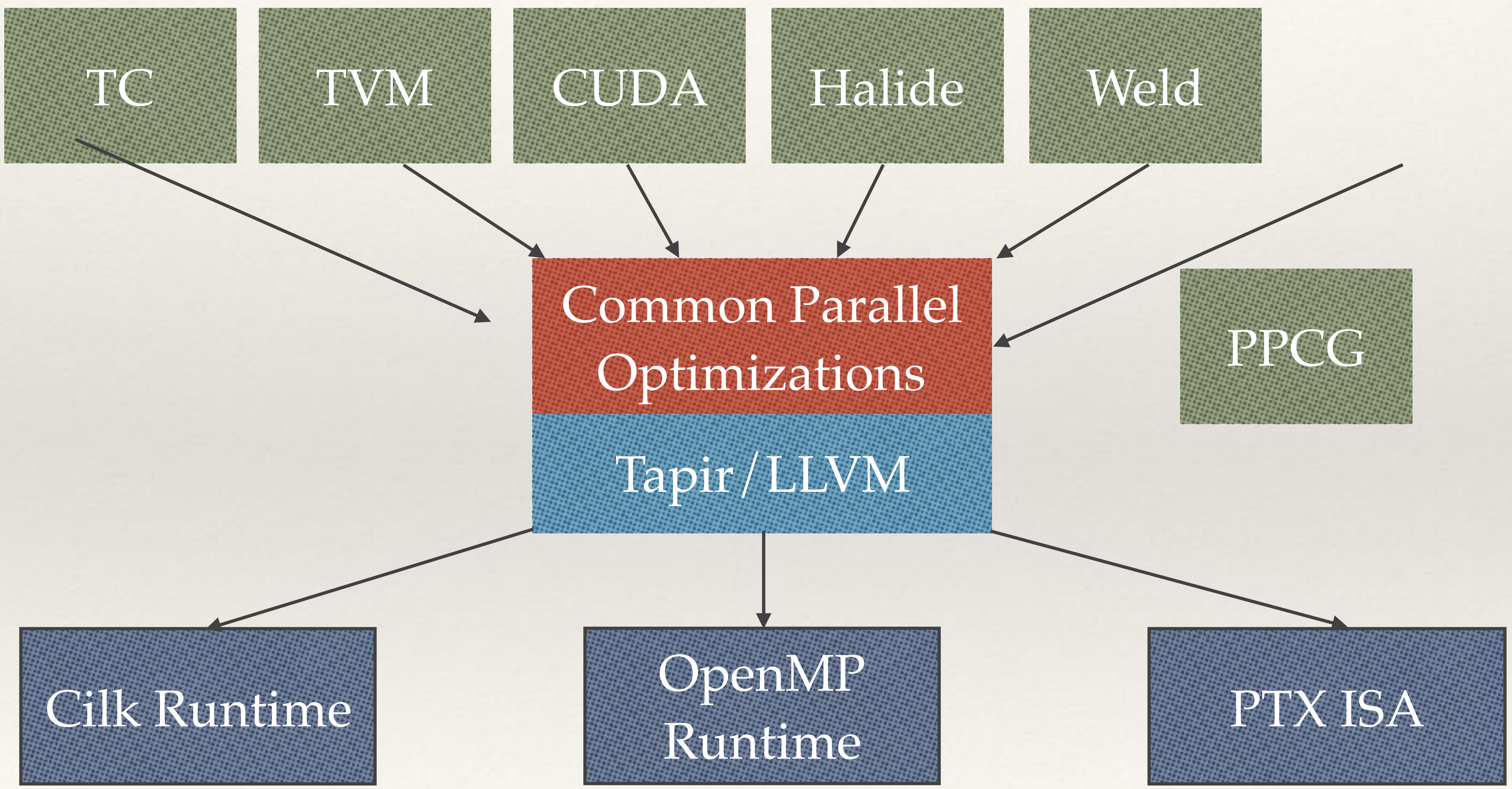
- There is ample opportunity for parallel-specific optimizations
  - Scheduling has potential for big boosts!

Rhino: The Parallel Compiler Dream

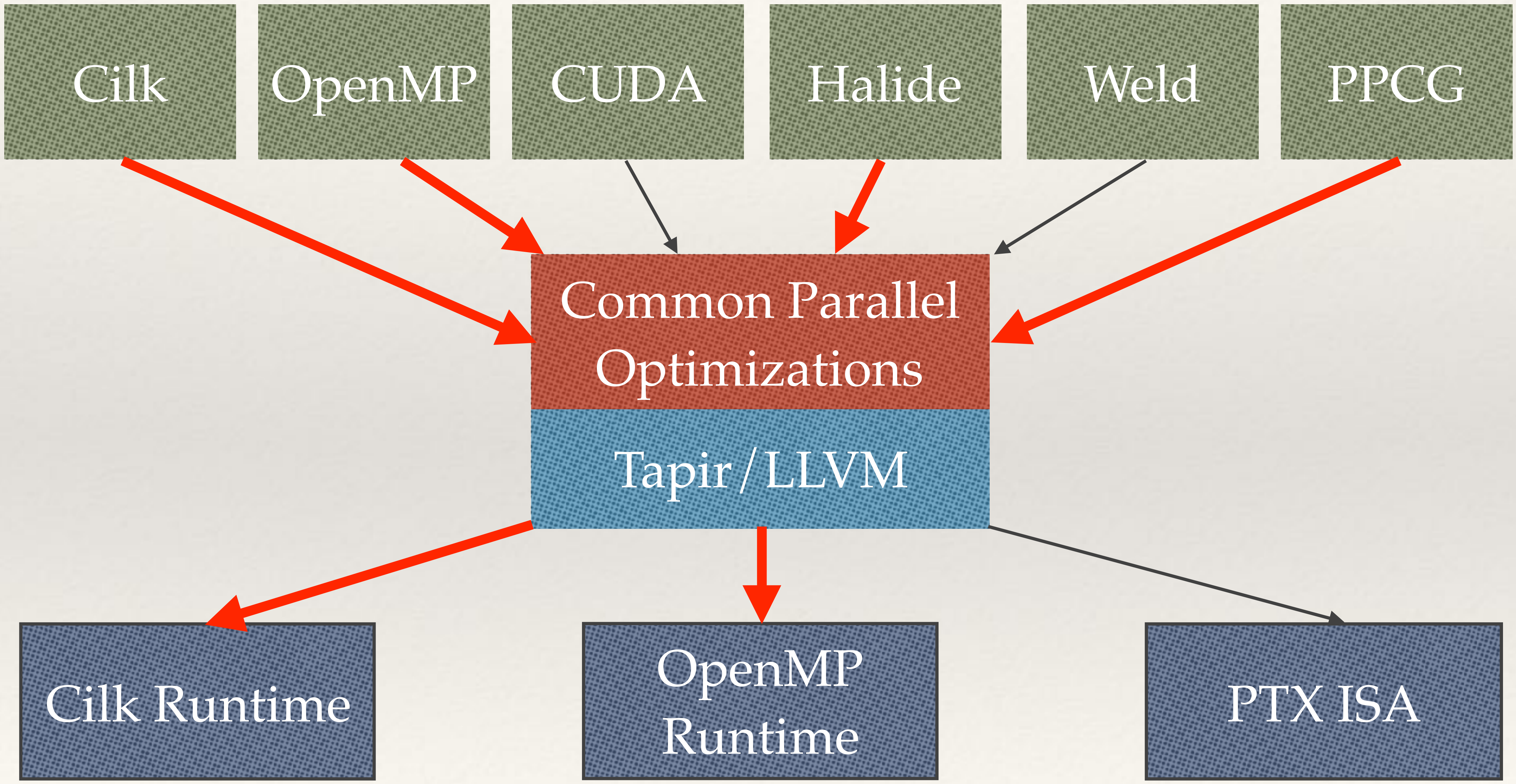
Parallel Optimizations

- 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



Rhino: The Parallel Compiler Dream



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