Principles of Tapir

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

LLVM Performance Workshop
February 4, 2017
Outline

❖ What is Tapir?
❖ Tapir’s debugging methodology
❖ Tapir’s optimization strategy
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: 0.312 s
__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;
}
Example: Normalizing a Vector in Parallel

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, \(n = 64\text{M}\). Machine: Amazon AWS c4.8xlarge, 18 cores.

Running time of original serial code: \(T_S = 0.312\text{ s}\)

Running time on 18 cores: \(T_{18} = 180.657\text{ s}\)

Running time on 1 core: \(T_1 = 2600.287\text{ s}\)

The story for OpenMP is similar, but more complicated.
Parallel code compiled using GCC 6.2. Cilk Plus/LLVM produces worse results.
Compiling Parallel Code Today

LLVM pipeline

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

LLVM pipeline for parallel code

Parallel code → 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.

Call into runtime to execute parallel loop.

Helper function encodes the loop body.

Existing optimizations cannot move call to `norm` out of the loop.
Tapir: Task-based Asymmetric Parallel IR

Tapir embeds parallelism into LLVM 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 optimization passes work on parallel code.
Parallel IR: An Old Idea

Previous work on parallel IR’s:

- Parallel precedence graphs [SW91, SHW93]
- Parallel flow graphs [SG91, GS93]
- Concurrent SSA [LMP97, NUS98]
- Parallel program graphs [SS94, S98]
- HIR [ZS11, BZS13]
- SPIRE [KJAI12]
- INSPIRE [JPTKF13]
- LLVM’s parallel loop metadata
- “[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]
- LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications [KJIAC15]
- LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading [TSSGMGZ16]
Parallel IR: A Bad Idea?

From “[LLVMdev] LLVM Parallel IR,” 2015:

❖ “[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…”

Other communications, 2016–2017:

❖ “There are a lot of information needs to be represented in IR for [back end] transformations for OpenMP.” [Private communication]
❖ “If you support all [parallel programming features] in the IR, a *lot* [of LOC]… would probably have to be modified in LLVM.” [[RFC] IR-level Region Annotations]
## 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>
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);
}
```

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: \(T_S / T_1 = 97\%\)
Summary of Performance Results

Compared to handling parallel constructs in the front end:

❖ Tapir/LLVM produces executables with higher work-efficiency for 17 of the 20 benchmarks — as much as 10–25% higher on a third of these benchmarks.

❖ Tapir/LLVM produces executables with at least 99% work-efficiency on 12 of the benchmarks, whereas the competition does so on 2.

❖ Tapir/LLVM produces executables with comparable or better parallel speedup.
For More on Tapir...

Come to the PPoPP talk!
Tuesday, February 7
Room 400/402

Or ask me and Billy after this talk.
Principles of Tapir

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

LLVM Performance Workshop
February 4, 2017
What Is Parallel Programming?

- Pthreads
- Message passing
- Vectorization
- Task parallelism
- Data parallelism
- Dataflow
- Multicore
- HPC
- GPU’s
- Heterogeneous computing
- Shared memory
- Distributed memory
- Clients and servers
- Races and locks
- Scheduling and load balancing
- Work efficiency
- Parallel speedup
- Etc.

Tapir does NOT directly address ALL of these.
Focus of Tapir

Tapir strives to make it easy for average programmers to write efficient programs that achieve parallel speedup.

- Multicores
- Task parallelism
- Simple and extensible
- Deterministic debugging
- Serial semantics
- Simple execution model
- Work efficiency
- Parallel speedup
- Composable performance
- Parallelism, not concurrency
Focus of Tapir

Tapir strives to make it easy for average programmers to write efficient programs that achieve parallel speedup.

- Multicores
- Task parallelism
- Simple and extensible
- Deterministic debugging
- Serial semantics
- Simple execution model
- Work efficiency
- Parallel speedup
- Composable performance
- Parallelism, not concurrency
Outline

- What is Tapir?
- Tapir’s debugging methodology
- Tapir’s optimization strategy
Parallel programming is strictly harder than serial programming because of race bugs.

**Example:** A buggy `norm()` function

```c
__attribute__((const))
double norm(const double *A, int n) {
    double sum = 0.0;
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
        sum += A[i] * A[i];
    return sqrt(sum);
}
```

Concurrent updates to `sum` can nondeterministically produce different results.

How do I spot these bugs in my million-line codebase?

How do I find a race if I’m “lucky” enough to never see different results?

What if the compiler creates the race?
A Compiler Writer’s Nightmare

Bug 55555 - Transformation puts race into race-free code

Attachments

Parallel test case  (text/plain)
2017-02-04, Angry Hacker

Add an attachment

Angry Hacker 2017-02-04

Created attachment 12345
Parallel test case

My parallel code is race free, but the compiler put a race in it!! >:(

1 run ✔
10 runs ✔
1000 runs ✔

Compiled program

Despite the programmer’s assertion, multiple runs indicate no problem.

❖ Is the compiler buggy?
❖ Is the programmer wrong?
Debugging Tapir/LLVM

Tapir/LLVM contains a provably good race detector for verifying the existence of race bugs deterministically.

❖ Given a program and an input — e.g., a regression test — the race-detection algorithm guarantees to find a race if one exists or certify that no races exist [FL99, UAFL16].

❖ The race-detection algorithm introduces approximately constant overhead.

❖ We used the race detector together with opt to pinpoint optimization passes that incorrectly introduce races.
What about Thread Sanitizer?

Efficient race detectors have been developed, including FastTrack [FF09] and Thread Sanitizer [KPIV11].

❖ These detectors are best effort: they are not guaranteed to find a race if one exists.

❖ These detectors are designed to handle a few parallel threads, comparable to the number of processors.

❖ Task-parallel languages are designed to get parallel speedup by exposing orders of magnitude more parallel tasks than processors.
Outline

- What is Tapir?
- Tapir’s debugging methodology
- Tapir’s optimization strategy
Example: Normalizing a Vector with OpenMP

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:* $T_S = 0.312$ s

Compiled with LLVM 4.0, running time on 1 core: $T_1 = 0.329$ s

Compiled with LLVM 4.0, running time on 18 cores: $T_{18} = 0.205$ s
Work Analysis of Serial Normalize

\[ T(\text{norm}) = O(n) \]

\[ T(\text{normalize}) = n \cdot T(\text{norm}) + O(n) = O(n^2) \]
Work Analysis After LICM

```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;
}
```

\[ T(\text{normalize}) = T(\text{norm}) + O(n) = O(n) \]
Compiling OpenMP 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);
}

void omp_outlined(int n, double *restrict out,
                  const double *restrict in) {
    int local_n = n; double *local_out = out, *local_in = in;
    __kmpc_for_static_init(&local_n, &local_out, &local_in);
    double tmp = norm(in, n);
    for (int i = 0; i < local_n; ++i)
        local_out[i] = local_in[i] / tmp;
    __kmpc_for_static_fini();
}
```

Each processor runs the helper function once.

Helper function contains a serial copy of the original loop.
Work Analysis of OpenMP Normalize

How much work (total computation outside of scheduling) does this code do?

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

void normalize(double *restrict out, const double *restrict in, int n)
{
  __kmpc_fork_call(omp_outlined, n, out, in);
}

void omp_outlined(int n, double *restrict out,
                   const double *restrict in)
{
  int local_n = n; double *local_out = out, *local_in = in;
  __kmpc_for_static_init(&local_n, &local_out, &local_in);
  double tmp = norm(in, n);
  for (int i = 0; i < local_n; ++i)
  {
    local_out[i] = local_in[i] / tmp;
  }
  __kmpc_for_static_fini();
}
```

Let $P$ be the number of processors.

$T_1(\text{norm}) = O(n)$

$T_1(\text{omp_outlined}) = T_1(\text{norm}) + O(\text{local}_n) = O(n)$

$T(\text{normalize}) = P \times T_1(\text{omp_outlined}) = O(n \times P)$
What Does This Analysis Mean?

- This code is only work-efficient on one processor.
- Only minimal parallel speedup is possible.
- The problem persists whether norm is serial or parallel.
- This code slows down when not all processors are available.

\[ T(normalize) = O(n \times P) \]

```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);
}
```

Original serial running time: \( T_S = 0.312 \) s

1-core running time: \( T_1 = 0.329 \) s

18-core running time: \( T_{18} = 0.205 \) s
Tapir strives to optimize parallel code according the work-first principle:

- **First** optimize the work, not the parallel execution.
- **Sacrifice** minimal work to support parallel execution.

The work-first principle helps to ensure that parallel codes can achieve speedup in all runtime environments.
Status of Tapir

- Try Tapir/LLVM yourself!
  `git clone --recursive https://github.com/wsmoses/Tapir-Meta.git`

- We have a **prototype front end** for Tapir/LLVM that is substantially compliant with the Intel Cilk Plus language specification.

- Tapir/LLVM achieves **comparable or better performance** with GCC, ICC, and Cilk Plus/LLVM, and is becoming **comparably robust**.

- Last fall, a **software performance-engineering class** at MIT with ~100 undergrads used Tapir/LLVM as their compiler.

- Tapir’s **race detector** is available for debugging parallel programs.

- We’re continuing to enhance Tapir/LLVM with bug fixes, new compiler optimizations, and other new features.
Question?

Recap: Foci of Tapir

- Multicores
- Task parallelism
- Simple and extensible
- Deterministic debugging
- Serial semantics

- Simple execution model
- Work efficiency
- Parallel speedup
- Composable performance
- Parallelism, not concurrency
Backup Slides
Multicores Are The Bargain Component

Multicores are pervasive in today’s computing ecosystem.

Average programmers are writing code for multicores.
Multicores Are Powerful

GraphChi: How a Mac Mini outperformed a 1,636 node Hadoop cluster

Last year GraphChi, a spin-off of GraphLab, a distributed graph-based high performance computation framework, did something remarkable. GraphChi Subsequent algorithms by not dealing with distributed processing. Armed with this knowledge and understanding of the general benefits and disadvantages of single machine performance the processing steps can be designed. A single computer has usually two characteristics, firstly a large graph problem does not fit into the fast RAM (Random Access Memory), and secondly it has a large disk, which is large enough to hold the data. The traditional disks are only performant on sequential and not on random reads. Modern computers may come with solid state disks for faster random read and write though they are still significantly slower than RAM. Consequently, any algorithm aiming to solve graph problems on single machine commodity hardware has to utilise the disk and minimise the random access to data.

Divide and Conquer
Task Parallelism

Task parallelism provides simple linguistics for average programmers to write parallel code.

Example: parallel quicksort

```c
void pqsort(int64_t array[], size_t l, size_t h) {
  if (h - l < COARSENING)
    return qsort_base(array, l, h);
  size_t part = partition(array, l, h);
  cilk_spawn pqsort(array, l, part);
  pqsort(array, part, h);
  cilk_sync;
}
```

The child function is allowed (but not required) to execute in parallel with the parent caller.

Control cannot pass this point until all spawned children have returned.
Advantages of Fork-Join

Fork-join parallelism provides a simple, low-level model of parallelism.

Many higher-level parallel constructs can be encoded in the fork-join model [MRR12].

- Map
- Reduction
- Stencil
- Recursion
- Scan

Efficient parallel runtimes exist that support fork-join parallelism.
What About Everything Else?

Fork-join doesn’t cover all parallel patterns, but it can be extended (e.g., [LLSSZ15]).

Tapir itself is meant to be extensible.

❖ For now, a front-end can still insert runtime calls to handle parallel constructs that don’t have explicit IR support.

❖ New extensions for additional parallel constructs should be able to interoperate with Tapir.
How Can We Be Sure?

We are currently developing **formal semantics** for Tapir.

- An early draft of Tapir’s semantics can be found in Tao B. Schardl’s Ph.D. thesis.

- **Current Goal:** Prove that all serial code transformations are **safe** to perform on both race-free and racy Tapir programs.
Three Traits of Parallel Programs

❖ A parallel program has **serial semantics** if a 1-processor execution of the program is valid.

❖ A parallel program is a **faithful extension** of a serial program if eliding the parallel constructs yields a serial program with the original program’s serial semantics.

❖ A parallel program is **deterministic** if all (serial and parallel) executions have the same semantics.
Serial Semantics vs. Faithful Extension

Example: OpenMP code with private construct

```c
int main(void)
{
    int i = 0;
    #pragma omp parallel for private(i)
    for (int j = 1; j < 10; ++j) {
        i += j * j;
        printf("i = %d\n", i);
    }
    printf("i = %d\n", i);
    return 0;
}
```

Some languages, such as OpenMP, allow programs that are not faithful extensions.

- Tapir doesn’t care about whether the program source is a faithful extension.
Serial Semantics vs. Determinism

❖ A Tapir program can contain races, even though it has serial semantics.

❖ If the execution of a Tapir program contains no determinacy races [FL99], then it is deterministic.

❖ When optimizing a Tapir program, the compiler strives to preserve the program’s serial semantics and avoid introducing new races.
The logical control structure of a Tapir program execution can be modeled as a **directed acyclic graph**, or **dag**.

**Strand**: Serial chain of executed instructions.

**Fork**

**Dependency**: Strand B cannot execute until after strand A.

**Join**
Using the dag model, **parallel performance** can be understood in terms of **work** and **span**.

**Work**, \( T_1 \), is the serial running time of the program.

Example: \( T_1 = 18 \)

**Span**, \( T_\infty \), is the length of a longest path in the dag.

Example: \( T_\infty = 9 \)
Parallel Speedup

Parallel speedup on $P$ processors can be understood in terms of work and span.

- **Work Law:** $T_P \geq T_1/P$
- **Span Law:** $T_P \geq T_\infty$
- Modern parallel runtime systems are guaranteed to execute a parallel program on $P$ processors in time $T_P \leq T_1/P + O(T_\infty)$ [BL99].
Concurrency Is Complicated

Interactions between threads can **confound** traditional compiler optimizations.

Thread 1

\[ a = 1; \]

Thread 2

\[
\text{if } (x.\text{load}(RLX)) \\
\text{if } (a) \\
\text{y.store(1, RLX);}
\]

Thread 3

\[
\text{if } (y.\text{load}(RLX)) \\
x.\text{store}(1, RLX);
\]

This program produces different results under the C11 memory model if Threads 1 and 2 are sequentialized [VBCMN15].
Parallelism Sans Concurrency

Conceptually, Tapir introduces task-parallelism for speeding up a single thread of control.

C code for normalize()

```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);
}
```

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);
}
```

Same control, but cilk_for indicates an opportunity to speed up execution using parallel processors.
Weak Memory Models and Tapir

Tapir’s task-parallel model, with a serial elision, helps ensure that standard optimizations are legal.

C11 optimization example, written in Cilk pseudocode

```cilk
    cilk_spawn { a = 1; }

    cilk_spawn {
        if (x.load(RLX))
            if (a)
                y.store(1, RLX);
    }

    cilk_spawn {
        if (y.load(RLX))
            x.store(1, RLX);
    }
```

The serial semantics of `cilk_spawn` ensures that sequentialization is always allowed.
A Sweet Spot for Compiler Optimizations

- When optimizing across threads, standard compiler optimizations are not always legal.
- By enabling parallelism for a single thread of control, Tapir’s model is amenable to standard compiler optimizations.
- Vectorization is another example of where compilers use parallelism to speed up a single thread of control.