Exploring Tradeoffs in Parallel Implementations of C++ using Futures

Jonathan Sharman

July 14, 2017
Table of Contents

1. Introduction
2. Background
3. Our Approach: Fibertures
4. Micro-benchmark Results
5. Case Study: Local Sequence Alignment using Smith-Waterman
6. Related Work
7. Conclusions and Future Work
Introduction
Motivation

Need for standardized parallelism in C++

1. Hardware concurrency constantly increasing
2. C++ is a high-level language for writing efficient code

Standard solution: futures and async tasks

Benefits of Futures:

1. Synchronization operations cannot introduce data races on future objects
2. Support an easily maintained and composable functional style
3. Support object-oriented programming
4. Can express other parallel constructs with futures
Desired properties:

1. Safe, maintainable, programmable, portable
2. Low-overhead, scalable

Current implementations satisfy first set of criteria but not second
A combination of compile-time and run-time approaches is the most effective means of implementing parallel futures in C++. 
Contributions

1. Parallel C++ futures implementation: Fibertures
2. Source-to-source compiler transformations to facilitate code migration
3. A quantitative comparison of several implementations of parallel futures in C++
Background
Async Tasks and Futures in the C++ Standard Template Library

Contained in header <future>

- `std::promise<T>`: placeholder for a value of type `T`
- `std::future<T>`: represents a future value of type `T`
- `std::async()`: executes a task asynchronously, returns future return value
A future references the shared state of a promise object

Wait for a future to be ready using

- `get()`
- `wait()`
- `wait_for()`
- `wait_until()`

Can only call `get()` once unless future is converted to a shared future with `share()`. 
std::promise<int> int_promise;
std::future<int> int_future = int_promise.get_future();
//int_future.get();
int_promise.set_value(14);
std::cout << int_future.get();
```cpp
int square(int n) { return n * n; }

int main() {
    std::future<int> square_future = std::async(square, 3);
    std::future<int> cube_future = std::async([](int n) {
        return n * n * n;
    }, 2);
    std::cout << (square_future.get() + cube_future.get());
}
```
Async Task Launch Policy

User may specify a launch policy for an async task:
- `std::launch::async`
- `std::launch::deferred`
- `std::launch::async | std::launch::deferred`

A task marked async is invoked in a new thread.

A task marked deferred is invoked the first time its value is used (lazy evaluation).

We are interested only in the parallel overloads.
Pitfalls of \texttt{std::async()} and \texttt{std::future}

- Threads have high overhead for creation and context-switching
- Synchronization is blocking
HClib (https://github.com/habanero-rice/hclib)

- A high-level, lightweight, task-based programming model for intra-node parallelism in C and C++
- Uses a cooperative work-stealing strategy, implemented using Boost Context
- API includes a variety of parallel constructs, including async tasks with futures
- Supports integration of task parallelism with multiple distributed runtimes, including MPI, UPC++, and OpenSHMEM
- Supports data-driven futures (DDFs) and data-driven tasks (DDTs)
```cpp
uint64_t fibonacci(uint64_t n) {
    if (n < 2) return n;
    hclib::future_t<uint64_t> n1 = hclib::async_future([] {
        return fibonacci(n - 1);
    });
    hclib::future_t<uint64_t> n2 = hclib::async_future([] {
        return fibonacci(n - 2);
    });
    return n1.get() + n2.get();
}

int main(int argc, char* argv[]) {
    hclib::launch([]() {
        std::cout << fib(10) << '
';
    });
}
```
Our Approach: Fibertures
Problem Statement

- Pthreads are inefficient for applications using many tasks with possibly varying run time
- Want to utilize the programmability and portability of std::future while enabling scalable parallel performance
- Several implementations of C++ futures exist, using a variety of compiler- and library-based approaches
- We implemented Fibertures on top of the libfib runtime library
- We compare these approaches to C++ futures by programmability, portability, and efficiency
Swapstack Calling Convention

Swapstack:

- A calling convention used for switching between continuations
- Calls exchange the current stack for that of the invoked continuation
- Saves the address where execution should continue when the calling continuation resumes
libfib (https://github.com/stedolan/libfib)

A cooperative work-stealing runtime scheduler for C++ built using Swapstack

- Spawn lightweight tasks (fibers)
- Rapidly context-switch between fibers
- Cooperatively yield a fiber’s worker thread for another fiber to use
Our Approach: Fibertures

Our Extension to libfib: Fibertures

- fibertures defines an async task function that returns std::future, uses fibers
- modifies libfib scheduler to support improved future synchronization
- libfib supports fibers but not futures
- same safety and programmability downsides of using STL threads
// in namespace fibertures

struct Task {
    // The callback function.
    std::function<result_t()> f;
    // The promise used to create and set the future.
    std::promise<result_t> p;
    // Constructor.
    Task(std::function<function_t>&& f, args_t... args)
        : f{std::bind(f, args...)}
    {}
};
Our Approach: Fibertures

Replacement for `std::async()`: `fibertures::async()`

```cpp
// in namespace fibertures

std::future<result_t> async(function_t&& f, args_t... args)
{
    auto task = new Task{move(f), forward<args_t>(args)...};
    std::size_t task_address = (std::size_t)task;
    auto fiber_lambda = [](std::size_t task_address) {
        auto task = (Task<function_t, args_t>*)task_address;
        auto value = task->f();
        task->p.set_value(std::move(value));
        delete task;
    };
    auto future_result = task->p.get_future();
    worker::current().new_fiber(fiber_lambda, task_address);
    return future_result;
}
```

Jonathan Sharman

Exploring Tradeoffs in Parallel Implementations of C++ using Futures

July 14, 2017 23 / 57
Capturing and Passing Parameters with Fibertures

Cannot pass lambdas and multiple arguments directly to libfib
- Supports a limited number of parameters (just one, in our port)
- Does not support lambdas with captures

Solution
- Capture parameters and store callback in a new fibertures::Task
- Convert task address to an integer type
- Pass task address into a capture-less lambda inside fibertures::async()
- During task execution, convert task address back into a task pointer and invoke the original callback
Our Approach: Fibertures

Source-to-source Transformations

- Designed to make using Fibertures with existing standard C++ trivial
- Currently applied manually but could be automated
Include the header "fibertures.h" in each source file containing a parallel async task.

In `main()`, initialize runtime with `worker::spawn_workers(nworkers)`.

`nworkers` could be determined based on hardware concurrency or chosen by the user.
Asynchronous Call Transformations

- Transform calls to the parallel overloads of `std::async()` to calls to `fibertures::async()`.
- Same function signature and return type as `std::async()`.
- `fibertures::async()` does not support deferred evaluation.
- Must not apply if tasks uses thread-local data.
Our Approach: Fibertures

Asynchronous Call Transformations

Source Code

```cpp
std::future<T> fut = std::async(f, ...);
std::future<T> fut = std::async(std::launch::async, f, ...);
std::future<T> fut = std::async
    (std::launch::async | std::launch::deferred, f, ...);
```

Transformed Code

```cpp
std::future<T> fut = fibertures::async(f, ...);
```
Synchronization Transformations

- Need to prevent synchronization operations from blocking
- While future is not ready, yield current fiber and look for more work
- No worker thread is ever blocked unless there is no work to be done
Task Scheduling with Yield-loops

- Worker threads should perform useful work when available
- Want to avoid staying in the yield-loop waiting for a future to become unblocked
- libfib scheduler prioritizes doing local work over stealing
- Problem: A worker thread that generates a yield-loop never steals
- Solution: Modified libfib scheduler to prioritize stealing work
get() Transformation

Source Code

T val = fut.get();

Transformed Code

while (fut.wait_for(std::chrono::seconds(0))
    != std::future_status::ready) {
    worker::current().yield();
}
T val = fut.get();
wait() transformation.

Source Code

```cpp
fut.wait();
```

Transformed Code

```cpp
while (fut.wait_for(std::chrono::seconds(0)) != std::future_status::ready) {
    worker::current().yield();
}
```
wait_for() transformation.

Source Code

```cpp
future_status status = fut.wait_for(duration);
```

Transformed Code

```cpp
future_status status = future_status::ready;
{
    auto start = high_resolution_clock::now();
    while (fut.wait_for(std::chrono::seconds(0)) != future_status::ready) {
        if (high_resolution_clock::now() - start > duration) {
            status = future_status::timeout; break;
        }
        worker::current().yield();
    }
}
```
Our Approach: Fibertures

wait_until() Transformation

Source Code

```cpp
future_status status = fut.wait_until(time_point);
```

Transformed Code

```cpp
future_status status = std::future_status::ready;
while (fut.wait_for(std::chrono::seconds(0)) != future_status::ready) {
    if (time_point::clock::now() >= time_point) {
        status = future_status::timeout;
        break;
    }
    worker::current().yield();
}
```
Handling Return Values

Three cases:

- If the return value is assigned to a variable in the source code, do the same in the transformed code.
- If the return value of a synchronization operation is unused, omit the assignment.
- If the return value is used in a temporary expression, introduce a new variable with an unused name.
#include <future>

int f(); // Some lengthy computation

int main() {
    // Spawn a new thread to compute f().
    std::future<int> fut = std::async(std::launch::async, f);
    // Wait for the spawned thread to finish.
    fut.wait();
    return 0;
}
```cpp
#include <future>
#include "fibertures.h"

int f(); // Some lengthy computation

int main() {
    // Spawn 8 worker threads.
    worker::spawn_workers(8);
    // Spawn a new fiber to compute f().
    std::future<int> fut = fibertures::async(f);
    // Wait for the spawned fiber to finish.
    while (fut.wait_for(std::chrono::seconds(0)) != std::future_status::ready) {
        worker::current().yield();
    }
    return 0;
}
```
Measure the cost of \texttt{std::thread} and \texttt{libfib} fiber creation

```cpp
constexpr int nthreads = 100000;
for (int i = 0; i < nthreads; ++i) {
    std::thread{[] {}}.detach();
}
```

```cpp
constexpr int nthreads = 100000;
for (int i = 0; i < nthreads; ++i) {
    worker::current().new_fiber([](size_t) {}, 0);
}
```
Experimental Setup

All performance results obtained on Intel Ivy Bridge architecture
- 8 GB of memory
- Intel Core i7-3770K processor
- Four cores and eight hardware threads
- Default optimization level
## Micro-benchmark Results

### Task Creation Time for STL Threads and libfib Fibers

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Mean Task Creation Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL thread</td>
<td>3,230</td>
</tr>
<tr>
<td>libfib fiber</td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>185</td>
</tr>
<tr>
<td>4 threads</td>
<td>262</td>
</tr>
<tr>
<td>8 threads</td>
<td>332</td>
</tr>
<tr>
<td>16 threads</td>
<td>655</td>
</tr>
</tbody>
</table>
Micro-benchmark: Parallel Recursive Fibonacci Function

- Creates an extremely large number of asynchronous tasks
- Not a realistic problem
- Good stress test for parallel runtime systems
Micro-benchmark Results

Time to Compute the $N^{th}$ Fibonacci Number using STL and Fibertures, Log Scale

- Fibertures
- STL with Parallel Futures

Run Time (ms, log scale) vs. $N$

Jonathan Sharman  Exploring Tradeoffs in Parallel Implementation  July 14, 2017  43 / 57
Case Study: Local Sequence Alignment using Smith-Waterman
Identifies the maximally homologous subsequences between two input gene sequences

Constructs a rectangular scoring matrix

Non-border cells computed as a function of left, upper-left, and upper neighbors

High degree of ideal parallelism
Case Study: Local Sequence Alignment using Smith-Waterman

Smith-Waterman Inter-cell Data Dependence Graph

Jonathan Sharman
Exploring Tradeoffs in Parallel Implementation
July 14, 2017 46 / 57
### Execution Times of Smith-Waterman by Runtime System

<table>
<thead>
<tr>
<th>Runtime System</th>
<th>Execution Time (ms)</th>
<th>Number of Tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Sequential STL</td>
<td>26,335</td>
<td>34</td>
</tr>
<tr>
<td>STL with Parallel Futures</td>
<td>6,349</td>
<td>18</td>
</tr>
<tr>
<td>1 thread</td>
<td>26,650</td>
<td>17</td>
</tr>
<tr>
<td>2 threads</td>
<td>13,540</td>
<td>40</td>
</tr>
<tr>
<td>HClib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,113</td>
<td>83</td>
</tr>
<tr>
<td>8 threads</td>
<td>6,463</td>
<td>58</td>
</tr>
<tr>
<td>16 threads</td>
<td>7,126</td>
<td>221</td>
</tr>
<tr>
<td>1 thread</td>
<td>25,510</td>
<td>11</td>
</tr>
<tr>
<td>2 threads</td>
<td>13,153</td>
<td>64</td>
</tr>
<tr>
<td>Fibertures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,284</td>
<td>34</td>
</tr>
<tr>
<td>8 threads</td>
<td>6,218</td>
<td>26</td>
</tr>
<tr>
<td>16 threads</td>
<td>6,780</td>
<td>74</td>
</tr>
</tbody>
</table>
## Execution times of Smith-Waterman by Runtime System, 2,208 tiles

<table>
<thead>
<tr>
<th>Runtime System</th>
<th>Execution Time (ms)</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential STL</td>
<td></td>
<td>26,335</td>
<td>34</td>
</tr>
<tr>
<td>Parallel STL</td>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1 thread</td>
<td>26,650</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>13,540</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,113</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td>6,463</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td>7,126</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>1 thread</td>
<td>25,534</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>13,461</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,615</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td>6,343</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td>6,319</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>HClib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 thread</td>
<td>26,650</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>13,540</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,113</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td>6,463</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td>7,126</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>1 thread</td>
<td>25,534</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>13,461</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td>7,615</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td>6,343</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td>6,319</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

Jonathan Sharman  
Exploring Tradeoffs in Parallel Implementations  
July 14, 2017  
48 / 57
Qthreads (http://www.cs.sandia.gov/qthreads/)

- Provides support for lightweight tasks in C, comparable to fibers
- Support the use of full/empty bits (FEBs) for synchronization
- Does not support futures but could be used as a building block for futures
Folly Futures (https://github.com/facebook/folly/tree/master/folly/futures)

- Promise and future library for C++11
- Supports callback chaining with `then()` and `onError()`
- The C++ standard does not currently support callback chaining; however, the Concurrency TS does support chaining
- Different API from `<future>`
- Requires user-specified Executor for async tasks
- Offers more control over execution policy but requires more manual effort
Boost Fiber (http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/fiber/overview.html)

- A fiber runtime that supports futures
- Future synchronization does not block the worker thread
- Different future type from std::future, but APIs are similar
- User can specify scheduler (defaults to round-robin)
- Provides moderate level of compatibility with existing STL-only code
HPX (https://github.com/STEllAR-GROUP/hpx)

- Distributed and intra-node parallel runtime library for C++
- High level of compatibility with STL and Boost
- Extends futures with continuation chaining and locality awareness
- Relatively easy to translate existing code to HPX
Conclusions and Future Work
Conclusions

- C++ must support expressive and efficient means of expressing parallelism
- C++ futures confer programmability and safety benefits but can have significant performance drawbacks
- Drawbacks particularly apparent in applications requiring large numbers of async tasks
- Third-party libraries using compiler transformations and/or a runtime scheduler can effectively solve these problems
- We implemented Fibertures to improve the performance of parallel futures in C++ through use of fibers
  - Performs well compared to the STL and other futures libraries in some benchmarks
  - Easy transition from STL to Fibertures due to matching APIs and source-to-source transformations
Future Work

- Fix libfib to support compilation under higher optimization levels and reevaluate performance of libraries
- Integrate dependences directly into fibers
- Automate source-to-source transformations using a compiler tool such as LibTooling
- Explore ways to infer the best stack size automatically, perhaps per call rather than per program
Acknowledgments

- Vivek Sarkar
- Dan Wallach and Corky Cartwright
- The Habanero Extreme Scale Software Research Group
  - Special thanks to Max Grossman, Akihiro Hayashi, Jun Shirako, Nick Vrvilo, and Jisheng Zhao