RICE UNIVERSITY

Exploring Tradeoffs in Parallel Implementations of Futures in C++

by

Jonathan Sharman

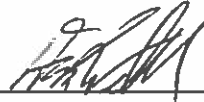
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master

Approved, Thesis Committee:

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

Robert "Corky" Cartwright
Professor of Computer Science



Dan Wallach
Professor of Computer Science and of
Electrical and Computer Engineering

Houston, Texas

August, 2017

ABSTRACT

Exploring Tradeoffs in Parallel Implementations of Futures in C++

by

Jonathan Sharman

As the degree of hardware concurrency continues to rise, multi-core programming becomes increasingly important for the development of high-performance code. Parallel futures are a safe, maintainable, and portable mechanism for expressing parallelism in modern C++, but the efficiency of the futures library implementation is critical for its usefulness in high-performance computing. In this work we introduce the Fibertures library and accompanying compiler support, as an alternate implementation for parallel futures in C++. We measure the performance of Fibertures against standard C++ and other runtime libraries. We find that a combination of compiler transformations and a runtime library, as exemplified by Fibertures, promises to be an effective means of implementing C++ futures.

Acknowledgements

Soli Deo gloria.

I would like to thank my MS advisor, Vivek Sarkar, for his indispensable guidance and support throughout this project. I would also like to thank the other members of my thesis committee, Dan Wallach and Corky Cartwright, both for their feedback on this work and for their mentorship throughout my graduate studies. Finally, I want to acknowledge the members of the Habanero Extreme Scale Software Research Group for their advice and assistance, with special thanks to Max Grossman, Akihiro Hayashi, Jun Shirako, Nick Vrvilo, and Jisheng Zhao.

Contents

Abstract ii

List of Illustrations vi

List of Tables viii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Statement 2

1.3 Contributions 2

2 Background 4

2.1 Async and Future in the C++ Standard Template Library 4

2.1.1 Basic Types 4

2.1.2 Pitfalls 9

2.2 Async Tasks and Futures in HClib 11

2.3 Async Tasks and Futures in HPX 13

3 Our Approach: Compiler Transformations and Fibertures Runtime 15

3.1 Problem Statement 15

3.2 Compiler Transformations and Run-time Library 16

3.2.1 `Swapstack` Primitive 16

3.3 libfib 17

3.4 Our Extension to libfib: Fibertures 18

3.5 Source-to-source Transformations 21

3.5.1 Run-time Library Inclusion and Initialization 21
3.5.2 Asynchronous Call Transformations 21
3.5.3 Synchronization Transformations 22
3.6 Transformation Example 26
3.7 Micro-benchmark Results 28
3.7.1 Fiber Creation Overhead 28
3.7.2 Parallel Recursive Fibonacci Function 30

4 Case Studies 32
4.1 Local Sequence Alignment using Smith-Waterman 32
4.1.1 Smith-Waterman with STL 33
4.1.2 Fibertures 37
4.1.3 HClib 37
4.1.4 Timing Results 39
4.2 Fast Multipole Method 40

5 Related Work 44
5.1 Polling and Spin-waiting 44
5.2 Qthreads 45
5.3 Folly Futures 45
5.4 Boost Fiber 47

6 Conclusions and Future Work 49
6.1 Conclusions 49
6.2 Future Work 50
6.2.1 Fibertures Runtime and Transformation Improvements 50
6.2.2 Fibertures as a Foundation for Other High-level Parallel Features 51

Bibliography 52

Illustrations

2.1 Example of using promises and futures in C++. Uncommenting line 3 would result in a deadlock. 6

2.2 Async task example that computes $3^2 + 2^3$. 7

2.3 Packaged task example that computes $3^2 + 2^3$. 8

2.4 Sequential `std::async()` call due to destructor of temporary object. 10

2.5 Parallel recursive Fibonacci function using HClib. 12

2.6 Parallel recursive Fibonacci function using HPX V1.0. 14

3.1 Fibertures `Task` type. 19

3.2 Definition of `fibertures::async()` as a replacement for `std::async()`. 20

3.3 Asynchronous call transformations. 23

3.4 `get()` transformation. 24

3.5 `wait()` transformation. 25

3.6 `wait_for()` transformation. 25

3.7 `wait_until()` transformation. 26

3.8 Complete STL-to-Fibertures source transformation. 27

3.9 Thread-creation micro-benchmark. 28

3.10 Fiber-creation micro-benchmark. 29

3.11 Task creation time for STL threads and libfib fibers. 29

3.12 Time to compute the Nth Fibonacci number using STL and Fibertures, using a log scale. 31

4.1 Inter-cell data dependence graph for the Smith-Waterman local sequence alignment algorithm. 34
4.2 STL-only parallel Smith-Waterman algorithm: outer loop. 35
4.3 STL-only parallel Smith-Waterman algorithm: task body. 36
4.4 Modifications to parallel Smith-Waterman algorithm to use Fibertures. 38

5.1 "Hello World" program using Qthreads, adapted from [1]. 46
5.2 Callback chaining using Folly Futures [2]. 47
5.3 Parallel recursive Fibonacci function using Boost Fiber futures. . . . 48

Tables

4.1 Execution times of Smith-Waterman by runtime system. 40

4.2 Execution times of Smith-Waterman by runtime system. In this case, all runtimes use the same number of tiles: 2,208. 41

4.3 Execution times of FMM benchmark by runtime system, problem size 20,000. 42

4.4 Execution times of FMM benchmark by runtime system, problem size 100,000. 43

Chapter 1

Introduction

1.1 Motivation

As processor clock speeds have plateaued, designers of high-performance hardware have compensated by increasing the number of hardware threads per machine. Utilization of multi-core parallel programming techniques is thus increasingly essential to producing performant code. The C++ programming language aims to provide high-level language and library features to facilitate writing portable, efficient code. To achieve these goals, C++ needs a standard way to express parallelism. In the C++11 language standard [3], the standards comittee adopted *async tasks* and *futures* as the primary tool for expressing high-level multi-core parallelism in C++. The choice of futures as the basic construct for parallelism has several advantages over lower-level constructs such as threads and locks.

First, future synchronization operations cannot introduce data races on future objects, making it easier to guarantee data-race freedom with futures relative to other parallel programming models. It is still possible to introduce a data race by reading and writing to a shared memory location from within the body of a function passed to `std::async()`. However, as long as the programmer only spawns parallel tasks via `std::async()` (as opposed to, for example, `std::thread`) and only communicates between parallel tasks using `std::future` objects, data-race freedom is guaranteed. Furthermore, deadlock freedom is also guaranteed when there are no races on refer-

ences to future objects [4].

Futures support a composable, functional style of parallel programming that is easy to reason about and maintain. In particular, futures are excellent for encoding task and data dependences that can be expressed using directed acyclic graphs. C++ is one of the oldest mainstream languages for object-oriented programming, and futures are also well suited for OOP, due to their encapsulation of async tasks and synchronization into objects.

The parallel programming model of asyncs and futures is also general enough to act as a basis for other high-level parallel constructs. This implies that no expressive power is lost in the choice of asyncs and futures over threads.

A quality parallel futures library in C++ would provide all these benefits to safety, maintainability, and programmability while ensuring performance through low overhead and good scalability, compared to more manual, lower-level parallel models. Unfortunately, we find that C++11's `<future>` library does not as yet meet this performance criterion. A solution that improves `<future>`'s performance while maintaining compatibility with existing standard C++ code would be enormously beneficial to the development of parallel applications in C++.

1.2 Thesis Statement

A combination of compile-time and run-time approaches is the most effective means of implementing parallel futures in C++.

1.3 Contributions

This thesis makes the following contributions:

1. An implementation of parallel futures in C++ called Fibertures, utilizing a custom LLVM-based compiler and a fibers-based lightweight threading library.

2. A set of source-to-source compiler transformations to facilitate migration from standard C++ futures alone to Fibertures.

3. A quantitative comparison of several implementations of parallel futures in C++ to study their tradeoffs.

Chapter 2

Background

In this chapter we present background information on parallel futures and their current status in C++, including the Standard Template Library (STL) and third-party libraries for C++.

2.1 Async and Future in the C++ Standard Template Library

The `<future>` header contains several basic tools for creating parallel tasks and synchronizing on them, including `std::promise`, `std::async()`, and `std::future`. In this section we will describe each of these components in more detail, explain their application to writing portable parallel C++ programs, and highlight some efficiency concerns with using the types and functions in the `<future>` header in programs that create large numbers of parallel tasks.

2.1.1 Basic Types

The `std::promise<T>` class template serves as a container for an object of a type `T` to be placed into the promise at a later time. Calling `get_future()` on a promise returns a corresponding `std::future<T>`, which references the promise's contained value. Calling `set_value()` on a promise sets its internal `T` value and notifies the corresponding future that the value is now available. Calling `set_value()` more than

once on the same promise object is an error and throws an exception.

The std::future<T> class template represents the result of a future computation. As stated above, a future holds a reference to the shared state of a promise object. The user may retrieve a future's value using its get() method, which blocks the current thread of execution until the future's value is available, i.e. until the corresponding promise has been set with set_value(), and returns the value using a move operation. The user may also wait for the value to become available without actually retrieving it using one of the methods wait(), wait_for(), or wait_until(). Calling get() on a future more than once results in an exception; this behavior prevents unknowingly reading a moved-from value. When the user needs to retrieve the future's value more than once, he or she may obtain a std::shared_future using std::future<T>::share(). Shared futures do not move the value upon retrieval.

Figure 2.1 exemplifies the basic functionality of std::promise and std::future. In line 1, we default-construct a promise object that holds an int value. In line 2, we obtain the future object corresponding to the promise. Uncommenting the statement on line 3 would result in a deadlock because int_future would block waiting for a promise that would never be satisfied. On line 4 we call set_value(14) on the promise object, storing 14 within the state shared by int_promise and int_future, thus satisfying the future. Now we can call get() on the future to obtain the value 14 from it.

The std::async<T>() function template is used to invoke a callback function asynchronously. It takes a callable object (a function pointer, pointer to member function, function object, or lambda) and a variadic list of arguments and returns a future object containing the future result of invoking the callable object on the given arguments. Internally, std::async() creates a std::promise object to store

Figure 2.1 : Example of using promises and futures in C++. Uncommenting line 3 would result in a deadlock.

```
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() << '\n';
```

the result of calling the passed callable object. An implementation may choose to use a `std::packaged::task<T>` (described below) instead of separate promise and function objects; the semantics are equivalent.

Figure 2.2 demonstrates the use of async tasks to compute multiple values in parallel. Line 3 creates an async task using a function pointer, while line 4 creates an async task using a lambda. Both spawn a new task to execute their respective functions with the passed arguments, each returning a future containing the result of its call. The choice between function pointer and stateless lambda is largely a matter of preference; the performance should be equivalent. Line 7 blocks first on `square_future` and then on `cube_future`, completing only when both tasks have finished.

The user may also specify a launch policy using `std::launch::async`, `std::launch::deferred`, or `std::launch::async | std::launch::deferred` as the first argument, before the callback function. If the user passes `std::launch::async`, the task is executed as though using a `std::thread`, which in all major implementations (e.g. GCC, Clang, Microsoft C/C++) executes on an ordinary operating system-level thread. Note that the requirements for `std::thread` are usually fewer than for system-level threads, e.g., Pthreads; how-

Figure 2.2 : Async task example that computes $3^2 + 2^3$.

```
1 int square(int n) { return n * n; }
2 int main() {
3   std::future<int> square_future = std::async(square, 3);
4   std::future<int> cube_future = std::async([](int n) {
5     return n * n * n;
6   }, 2);
7   std::cout square_future.get() + cube_future.get();
8 }
```

ever, each `std::thread` does require initialization of thread-local variables. If the user passes `std::launch::deferred`, then the task is invoked the first time its result is requested via a synchronization operation such as `std::future<T>::get()` or `std::future<T>::wait()`. This launch mode is useful for supporting lazy evaluation. If the user passes the bitwise disjunction of the two options, then it is left to the implementation to choose which mode to use. Specifying no launch policy results in implementation-defined behavior. In practice, implementations default to parallel execution for all calls except those explicitly specifying only `std::launch::deferred`. For the purpose of this paper, we are only concerned with async calls using the `std::launch::async` policy, explicitly or implicitly.

The `<future>` header also contains the `std::packaged_task<T>` class template. A packaged task acts like a combination of a `std::function` and `std::promise`. A user may create a packaged task from a callable object, obtain a `std::future` from the task containing the future value of invoking the callable object, and manually invoke the packaged task at a later time, possibly asynchronously.

Figure 2.3 demonstrates the use of packaged tasks and `std::thread` to achieve the same result as in Figure 2.2. Lines 3 and 4 create packaged tasks from a function

Figure 2.3 : Packaged task example that computes $3^2 + 2^3$.

```
1  int square(int n) { return n * n; }
2  int main() {
3    std::packaged_task<int()> square_task{square};
4    std::packaged_task<int()> cube_task{[](int n) {
5      return n * n * n;
6    }};
7    std::future<int> square_future = square_task.get_future();
8    std::future<int> cube_future = cube_task.get_future();
9    std::thread{[&] {
10     square_task(3);
11   }}.detach();
12   std::thread{[&] {
13     cube_task(2);
14   }}.detach();
15   std::cout << square_future.get() + cube_future.get();
16 }
```

pointer and lambda respectively. Lines 7 and 8 obtain futures from the packaged tasks. Unlike before, to execute the packaged tasks asynchronously, we must manually spawn them on separate threads, as in lines 9-14. Note that we may detach the threads since `std::future` provides the necessary synchronization. Finally, we get the futures, blocking until both threads have finished and the future results are available.

As the preceding example shows, packaged tasks can be used to achieve similar behavior to that of the async function. However, they have the disadvantage of introducing additional names into the local scope and require more lines of code. Therefore, `std::async()` is a better choice when the user does not require the flexibility of packaged tasks.

2.1.2 Pitfalls

While convenient for the programmer, there are several important drawbacks to using the standard parallel future facilities in C++. The first is that the overhead of spawning new threads and context-switching between them is relatively high, in both run time and memory usage. On a given platform, there are typically ways to reduce the overhead of thread creation somewhat, but these are inherently non-portable. When using the tools in `<future>` to create large numbers of tasks, thread creation overhead can dominate the computation and, in extreme cases, even exhaust virtual memory and crash the program. Especially in applications with unstructured workloads, it is often helpful to create a significantly larger number of async tasks than the number of available cores. This improves load balancing among the cores since one that runs out of work can switch to a new task rather than wait idly, but it also exacerbates the thread creation and context-switch overheads. See Section 3.7 for an empirical analysis of the cost of task creation using `<future>`.

A second pitfall is that `std::future` synchronization operations block the current thread of execution while waiting for data. This can result in poor utilization of available hardware threads. Ideally, a thread executing a task that is waiting for data should switch to another task that is ready to run, but this is not possible using `std::async()` and `std::future`.

Finally, a call to `std::async()` executes sequentially when the user does not store the returned future. This is a consequence of the blocking nature of `std::future` synchronization in combination with the the fact that `std::future` synchronizes in its destructor. If the user does not store the resulting future, then the return value of the call to `std::async()` is a temporary object, and its destructor is called at the end of the statement.

Figure 2.4 : Sequential `std::async()` call due to destructor of temporary object.

```
{
  auto start = steady_clock::now();
  std::async([] { // Return value not stored.
    std::this_thread::sleep_for(seconds{1});
  });
  auto end = steady_clock::now();
  auto elapsed = duration_cast<milliseconds>(end - start);
  std::cout << elapsed.count() << " ms\n"; // Prints ~1000 ms.
}
{
  auto start = steady_clock::now();
  std::future<void> f = std::async([] { // Return value stored in f.
    std::this_thread::sleep_for(seconds{1});
  });
  auto end = steady_clock::now();
  auto elapsed = duration_cast<milliseconds>(end - start);
  std::cout << elapsed.count() << " ms\n"; // Prints ~0 ms.
}
```

Figure 2.4 demonstrates this surprising behavior. The program contains two async tasks, each of which sleeps for one second. In the first call to `std::async()`, the result of the call is discarded after statement execution because it is not bound to a variable. The elapsed time shows that this task executes sequentially (though still in a separate thread), taking approximately 1,000 ms from before the call to after. In the second call, the result is assigned to a local `std::future` variable `f`. Since `f` remains in scope until the end of `main`, its destructor is not invoked at the end of the statement, and the calling thread continues almost immediately, resulting in an elapsed time of approximately 0 ms, as expected when spawning an asynchronous task.

As the example shows, this problem can be overcome by always storing the resulting future. However, it is easy to forget to do so when the future is not

otherwise required, and this behavior can be a subtle source of performance bugs. There is a proposal [5] to the C++ standard to change the behavior of `std::future` and `std::shared_future` such that neither blocks in the destructor. It would add `std::waiting_future` and `std::shared_waiting_future`, which behave as the current blocking versions, and change `std::async()` to return `std::waiting_future`. The user could convert a blocking future to a non-blocking future by calling `detach()`. However, this proposal has not yet been accepted, and furthermore, a user may still introduce a performance bug by forgetting to detach a waiting future.

2.2 Async Tasks and Futures in HClib

HClib is a high-level, lightweight, task-based programming model for intra-node parallelism in C and C++, developed in the Habanero Extreme Scale Software Research Group at Rice University [6]. The HClib runtime uses a cooperative work-stealing strategy, implemented using Boost Context [7]. Its API includes a variety of parallel constructs, including async tasks with futures. HClib futures behave similarly to `std::shared_future` in that they permit multiple `get()` operations by default. Figure 2.5 shows a complete HClib program that prints the 11th Fibonacci number, computed recursively in parallel, with no cutoff. The program is very similar to the corresponding STL-only program. The first difference is the use of `hclib::future_t` instead of `std::future` and `hclib::async_future` instead of `std::async()` (lines 3, 6). The second difference is that calls to HClib functions must occur within an `hclib::launch()` dynamic scope, as shown in lines 12-14. The `hclib::launch()` call handles runtime initialization and finalization, automatically allocating and deallocating worker threads and ensuring any outstanding parallel computations have completed after the call returns. Finally HClib supports integration of task parallelism

Figure 2.5 : Parallel recursive Fibonacci function using HClib.

```
1  uint64_t fibonacci(uint64_t n) {
2    if (n < 2) return n;
3    hclib::future_t<uint64_t> n1 = hclib::async_future([=] {
4      return fibonacci(n - 1);
5    });
6    hclib::future_t<uint64_t> n2 = hclib::async_future([=] {
7      return fibonacci(n - 2);
8    });
9    return n1.get() + n2.get();
10 }
11 int main(int argc, char* argv[]) {
12   hclib::launch([]() {
13     std::cout << fib(10) << '\n';
14   };
15 }
```

with multiple distributed runtimes, including MPI, UPC++, and OpenSHMEM.

In addition to C++11-style futures synchronized with `put()` and `get()` operations, HClib provides data-driven futures (DDFs) and data-driven tasks (DDTs) [8]. DDTs differ from `std::async()` in that they are registered with one or more futures and execute when some subset of those futures (e.g., all or any) are ready, rather than blocking the current thread while waiting for the necessary data. Though they use a different syntax and are explicitly non-blocking, DDFs are conceptually similar to callback chaining, which the STL does not support as of C++17 but does support in the Concurrency TS, using `std::future::then`, `std::when_all`, and `std::when_any` [9].

2.3 Async Tasks and Futures in HPX

HPX (High Performance ParalleX) is a distributed and intra-node parallel library and runtime system for C++, developed by the STE||AR Group [10]. HPX is distinguished by its high degree of conformance to the C++ language and Boost library. HPX defines `hpx::future`, `hpx::async()`, and related functions, which closely conform to the corresponding types and functions in `<future>`, with some extensions such as continuation chaining and locality awareness. It is therefore relatively easy to translate existing modern parallel C++ code written only with the STL to use HPX as the runtime instead.

Figure 2.6, adapted from [11], shows the single-node HPX equivalent to the program in Figure 2.5. HPX is locality-aware, and therefore, unlike in the corresponding standard C++ program, the user must pass action and locality arguments to the `async` calls. Actions are the means HPX uses to transfer work between localities. The `HPX_PLAIN_ACTION` macro on line 2 handles the boilerplate code for defining an action, in this case `fibonacci_action`. Since this is not a distributed program, this program uses the current locality (determined by a call to `hpx::find_here` on line 5) for the asynchronous calls on lines 8 and 10.

Note also that HPX programs require initialization and finalization of the HPX runtime, via calls to `hpx::init()` and `hpx::finalize()`, as opposed to the implicit specification of initialization and finalization calls in HClib's `hclib::launch()` method. The `init()` function processes command-line parameters recognized by the HPX runtime and invokes the user-defined `hpx_main()` function, while the `finalize()` function handles shut-down. When using HPX for distributed programming, `hpx::finalize()` gracefully terminates all other localities.

Figure 2.6 : Parallel recursive Fibonacci function using HPX V1.0.

```
1  boost::uint64_t fibonacci(boost::uint64_t n);
2  HPX_PLAIN_ACTION(fibonacci, fibonacci_action);
3  boost::uint64_t fibonacci(boost::uint64_t n) {
4    if (n < 2) return n;
5    hpx::naming::id_type const locality_id = hpx::find_here();
6    fibonacci_action fib;
7    hpx::unique_future<boost::uint64_t> n1 =
8      hpx::async(fib, locality_id, n - 1);
9    hpx::unique_future<boost::uint64_t> n2 =
10     hpx::async(fib, locality_id, n - 2);
11   return n1.get() + n2.get();
12 }
13 int hpx_main() {
14   {
15     fibonacci_action fib;
16     std::cout << fib(hpx::find_here(), 10);
17   }
18   return hpx::finalize();
19 }
20 int main(int argc, char* argv[]) {
21   return hpx::init(argc, argv);
22 }
```

Chapter 3

Our Approach: Compiler Transformations and Fibertures Runtime

3.1 Problem Statement

Despite the fact that performance is one of the the primary goals of the C++ language, standard C++ parallel futures are inefficient. Due in part to the requirements the C++ standard imposes on the behavior of `std::async()`, vendors currently implement async and future using Pthreads or a comparable operating system-level threading construct. However, Pthreads scale poorly when used to implement futures. For common use cases of futures, involving the creation of many tasks, some of which may be short-running, Pthreads require too much time and memory to create and too much time to context switch. Our goal is to utilize the programmability and portability of `std::future` while enabling scalable parallel performance. Several groups have written custom implementations of C++ futures, using a variety of compiler- and library-based approaches. We have implemented futures on top of the libfib [12][13] runtime library, a lightweight threading library supported by a custom compiler, along with a set of source-to-source transformations, allowing existing C++ code to leverage the efficiency of fibers. In this work, we compare these various approaches to C++ futures in terms of programmability, portability, and performance.

3.2 Compiler Transformations and Run-time Library

Our approach utilizes the `Swapstack` LLVM compiler primitive and fiber library described in [12] to enable the creation of lightweight tasks instead of heavyweight threads.

3.2.1 `Swapstack` Primitive

`Swapstack` is a calling convention used for switching between continuations. A call to a function marked with the `Swapstack` attribute not only yields control to the invoked continuation but also swaps its stack for the current stack, saving the address where execution should continue when the calling continuation resumes. There are three performance advantages to this approach to context switching [12]:

1. `Swapstack` does not use callee-save registers, so the LLVM compiler only saves live registers, avoiding saving registers that are not live.

2. The context switch is implemented using the existing LLVM call/return mechanism, allowing the unmodified optimizer to work on `Swapstack` functions.

3. An arbitrary number of parameters may be passed via registers, allowing very efficient communication between continuations.

The original implementation of `Swapstack` is a modification of Clang/LLVM 3.0. For use in this work, we have ported `Swapstack` to Clang/LLVM 3.7. This port supports only a single argument, via a register. Support for additional arguments may be added later.

3.3 libfib

libfib is a cooperative work-stealing runtime scheduler for C++ built using `Swapstack` to provide the necessary support for rapid context-switching, suspension of execution, and resumption of suspended fibers. It provides utilities for spawning new lightweight fibers and for yielding a fiber's worker thread for another fiber to use. The spawn operation takes a function pointer and allocates a new stack on which to execute that function using a new fiber. The yield operation allows a fiber to suspend its execution and place itself back on the running queue and invoke another fiber on the running queue while it waits for its data to become ready.

Because of the advantages of `Swapstack`, the spawn and yield operations can be orders of magnitude faster than thread creation and context-switching at the OS level, allowing for the creation of thousands or even millions of fibers [12]. Since the worker threads can always pick up more work (if available) when a fiber suspends itself, the threads never need to block while waiting for pending results.

Use of libfib is straightforward. The user initializes the libfib runtime using a call to the static function `worker::spawn_workers(nworkers)`, which allocates `nworkers` worker threads to handle execution of fibers. The user can execute a task on a new fiber spawned from the current worker thread using `worker::current().new_fiber(func, arg)`, where `func` and `arg` are the task function and argument, respectively. Finally, the user can await the completion of all outstanding fibers by calling `worker::await_completion()`, if the program's synchronization does not otherwise guarantee fiber completion.

3.4 Our Extension to libfib: Fibertures

libfib is a useful tool for creating large numbers of lightweight tasks. However, it only supports thread-like fibers; i.e., fibers cannot return a value asynchronously with a future, so communication between fibers requires manual synchronization. This has all the safety and programmability downsides of using STL threads and locks.

We have developed an extension to libfib called Fibertures, which adds the ability to create async tasks that return `std::future`s, just like the STL's `std::async()`, except that internally, it spawns a lightweight fiber instead of a heavyweight OS-level thread.

Note that for the sake of clarity and brevity, we do not list the full template parameter types in the code listings in this section. Instead, we use `function_t` and `result_t` to represent a passed function's decayed type and return type, respectively, and `args_t` to represent the variadic types of passed arguments. Furthermore, we do not show template specializations for functions returning `void` since these specializations are structurally very similar to the unspecialized code shown.

First we define a utility type called `fibertures::Task` to encapsulate a callback function along with a promise object to store the future result of invoking the callback. The definition of the `Task` type is shown in Figure 3.1. The `fibertures::Task` constructor takes a function and variadic argument list, binds them, and stores the result for later execution.

Next, to support the creation of future-based lightweight tasks, we define a new async task creation function: `fibertures::async()`. This function utilizes libfib to spawn tasks on fibers instead of OS-level threads. We will use this function in our source transformations as a replacement for `std::async()`. Figure 3.2 shows the definition of `fibertures::async()`.

Figure 3.1 : Fibertures `Task` type.

```
1  // in namespace fibertures
2  struct Task
3  {
4    // The callback function.
5    std::function<result_t()> f;
6
7    // The promise used to create and set the future.
8    std::promise<result_t> p;
9
10   // Constructor.
11   Task(std::function<function_t>&& f, args_t... args)
12     : f{std::bind(f, args...)}
13   {}
14 };
```

We now summarize our approach for using fibers to implement async tasks. Because libfib only supports a limited number of parameters for tasks spawned on fibers (just one, in our port), and because it does not support lambdas with captures, we cannot simply forward the function and its parameters to `worker::current()::new_fiber`. Instead, we store the callback in a new `fibertures::Task`, converting its address to an integer type and passing the task address into a lambda inside `fibertures::async()`. Because this lambda is captureless, it is implicitly convertible to `void (*)(std::size_t)`, which is the type we need for a fiber callback. When the fiber callback begins executing, it converts the task address back into a task pointer, invokes the original callback, and resolves the future with its return value.

The extra indirection of passing a task object to the new fiber rather than passing the user's callback directly introduces a small amount of additional overhead. There

Figure 3.2 : Definition of `fibertures::async()` as a replacement for `std::async()`.

```
// in namespace fibertures
std::future<result_t> async(function_t&& f, args_t... args)
{
  // Create a new task to hold f and its promise.
  auto task = new Task{move(f), forward<args_t>(args)...};
  std::size_t task_address = reinterpret_cast<std::size_t>(task);

  auto fiber_lambda = [](std::size_t task_address) {
    // Restore the task pointer from the address argument.
    auto task = reinterpret_cast<Task<function_t, args_t>*>
      (task_address);

    // Invoke the callback and store its return value.
    auto value = task->f();

    // Set the promise value, readying futures awaiting this value.
    task->p.set_value(value);

    // Done with the task; delete it.
    delete task;
  };

  // Retrieve the future from the promise.
  auto future_result = task->p.get_future();

  // Spawn a fiber to invoke the callback and set
  // the promise's value.
  worker::current().new_fiber(fiber_lambda, task_address);

  // Return the future result.
  return future_result;
}
```

is thus some opportunity for future work to optimize the case where the callback function does not capture and has sufficiently few parameters that they can fit into the available registers for a fiber.

3.5 Source-to-source Transformations

In addition to Fibertures, we have implemented a set of source-to-source transformations designed to simplify the use of Fibertures in standard C++ programs.

3.5.1 Run-time Library Inclusion and Initialization

First we include the header `"fibertures.h"` in each source file that contains any calls to the parallel overloads of `std::async()`. This allows subsequent transformation steps to use features of the Fibertures library. Next we add a call to the libfib function `worker::spawn_workers(nworkers)`, where `nworkers` is the desired number of worker threads, at the top of the `main()` function. The choice for the value of `nworkers` could be determined automatically based on hardware concurrency or set by the user via an environment variable, as in other multithreaded runtimes including OpenMP.

3.5.2 Asynchronous Call Transformations

We transform calls to the asynchronous overload of `std::async()` to calls to `fibertures::async()`. The `fibertures::async()` function takes a `Callable` object with a variadic argument list and returns a `std::future`, just as `std::async()` does. However, unlike `std::async()`, `fibertures::async()` does not have an overload that takes a `std::launch` object as the first argument because it always executes the function on a new fiber, which is analogous to calling `std::async()` using the

`std::launch::async` flag. In other words, `fibertures::async()` does not support the synchronous deferred evaluation launch policy.

The behavior of `std::async()` when a launch policy argument is not present or when both the `std::launch::async` and `std::launch::deferred` flags are set is implementation-defined. For simplicity, we will treat all `std::async()` calls that do not specify `std::launch::deferred` explicitly and exclusively as asynchronous calls and transform them accordingly. With this assumption, the transformation for a call to `std::async()` consists of replacing occurrences of asynchronous calls to `std::async()` with calls to `fibertures::async()` and removing the launch policy argument if present. There are three forms such a call to `std::async()` can take, and each maps to the same transformed call, as shown in Figure 3.3.

3.5.3 Synchronization Transformations

Spawning fibers instead of threads addresses the scalability issue with `std::async()`; however, it does not address the problem of blocking synchronization operations. To prevent these operations from blocking the worker thread, it is necessary to transform every call to one of the four standard `wait` or `get` operations to a non-blocking form. Our transformations introduce while-loops utilizing the yield operation to suspend the execution of the current fiber as long as the future object is not yet ready. Recall that each call to `worker::yield` also resumes a fiber waiting on the running queue so that no worker thread is ever blocked while awaiting future data, unless there are no more pending tasks to be executed. The synchronization transformations are given in Figures 3.4, 3.5, 3.6, and 3.7.

We want worker threads to perform useful work if there is any available, rather than stay in the yield-loop waiting for a future to become unblocked. The original

Figure 3.3 : Asynchronous call transformations.

(a) Source code. Each of these three variants is a valid candidate for transformation.

```
1 std::future<T> fut = std::async(f, arg1, arg2, ...);
```

```
1 std::future<T> fut = std::async
2   ( std::launch::async
3   , f
4   , arg1, arg2, ...
5   );
```

```
1 std::future<T> fut = std::async
2   ( std::launch::async | std::launch::deferred
3   , f
4   , arg1, arg2, ...
5   );
```

(b) Transformed code.

```
1 std::future<T> fut = fibertures::async(f, arg1, arg2, ...);
```

Figure 3.4 : `get()` transformation.

(a) Source code.

```
1 T val = fut.get();
```

(b) Transformed code.

```
1 while (fut.wait_for(std::chrono::seconds(0))
2        != std::future_status::ready) {
3   worker::current().yield();
4 }
5 T val = fut.get();
```

implementation of libfib prioritizes doing local work before stealing work from other worker threads. This scheduling algorithm has the side effect that a worker thread that has generated a yield-loop never steals because the yield-looping task remains in its local run queue. To mitigate this problem, we modified the libfib scheduler loop to prioritize stealing work. Under this new scheme, when a task is blocked awaiting availability of a future value, the worker thread executing that task steals from another worker's run queue. This reduces the expected overall amount of time workers spend repeatedly checking whether futures are ready and improves load balancing. There are more advanced strategies for handling blocking synchronization operations without using a yield-loop. We discuss some of these alternatives in Section 6.2.

In cases where the return value of `get()`, `wait_for()`, or `wait_until()` is not used, the assignment in the transformed code may simply be omitted. Cases where the return value is used as a temporary object in an expression, e.g. `if (fut.get() == 0) { ... }`, the transformed code may need to introduce new local variables to store the temporary results. In these cases, we must employ the techniques described

Figure 3.5 : `wait()` transformation.

(a) Source code.

```
1 fut.wait();
```

(b) Transformed code.

```
1 while (fut.wait_for(std::chrono::seconds(0))
2        != std::future_status::ready) {
3   worker::current().yield();
4 }
```

Figure 3.6 : `wait_for()` transformation.

(a) Source code.

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

(b) Transformed code.

```
1  future_status status = future_status::ready;
2  {
3    auto start = std::chrono::high_resolution_clock::now();
4    while (fut.wait_for(std::chrono::seconds(0))
5           != std::future_status::ready) {
6      auto elapsed_time = std::chrono::high_resolution_clock::now() -
         ↪ start;
7      if (elapsed_time > duration) {
8        status = future_status::timeout;
9        break;
10     }
11     worker::current().yield();
12   }
13 }
```

Figure 3.7 : `wait_until()` transformation.

(a) Source code.

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

(b) Transformed code.

```
1 std::future_status status = std::future_status::ready;
2 while (fut.wait_for(std::chrono::seconds(0))
3        != std::future_status::ready) {
4   if (time_point::clock::now() >= time_point) {
5     status = future_status::timeout;
6     break;
7   }
8   worker::current().yield();
9 }
```

in [14] to hoist the statement out of the expression while preserving the semantics of the original program.

3.6 Transformation Example

The example in Figure 3.8 demonstrates the complete set of transformations needed for a simple program using `std::async()` and `std::future`. The source transformation adds an `include` directive to support the use of the fiber library and `fibertures::async()` and adds a statement at the start of `main()` to spawn worker threads (line 6). It also replaces the call to `std::async()` with `fibertures::async()` (line 8). Finally, it eliminates the blocking `wait` operation using a while-loop that yields the current fiber if the future's value is not yet available (lines 10-13).

The transformed code is valid standard C++ with the exception of the fiber

Figure 3.8 : Complete STL-to-Fibertures source transformation.

(a) Source program.

```
1 #include <future>
2 int f(); // Some lengthy or I/O-based computation
3 int main() {
4   // Spawn a new OS-level thread to compute f().
5   std::future<int> fut = std::async(std::launch::async, f);
6   // Wait for the thread spawned with async to finish.
7   fut.wait();
8   return 0;
9 }
```

(b) Transformed program.

```
1  #include <future>
2  #include "fibertures.h"
3  int f(); // Some lengthy or I/O-based computation
4  int main() {
5    // Spawn 8 worker threads.
6    worker::spawn_workers(8);
7    // Spawn a new fiber to compute f().
8    std::future<int> fut = fibertures::async(f);
9    // Wait for the fiber spawned with fibertures::async to finish.
10   while (fut.wait_for(std::chrono::seconds(0))
11          != std::future_status::ready) {
12     worker::current().yield();
13   }
14   return 0;
15 }
```

Figure 3.9 : Thread-creation micro-benchmark.

```
1  constexpr int nthreads = 100000;
2  for (int i = 0; i < nthreads; ++i) {
3    std::thread{[] {}}.detach();
4  }
```

library itself, which requires a modified LLVM compiler that supports the `Swapstack` primitive.

3.7 Micro-benchmark Results

Here we compare the performance of libfib or Fibertures against the STL `<future>` library using two micro-benchmarks. The task-creation micro-benchmark measures the cost of creating a task in each system. The Fibonacci benchmark measures overall performance when the number of tasks is very high and the amount of work very low. We performed these tests on the Intel Ivy Bridge architecture, with 8 GB of memory and an Itel Core i7-3770K processor, which has four cores and eight hardware threads.

3.7.1 Fiber Creation Overhead

Figure 3.9 is a micro-benchmark for measuring the cost of `std::thread` creation. It spawns and detaches a large number of threads using `std::thread` that do nothing and then computes the average time of creation. Figure 3.10 performs a similar computation for fiber creation using libfib with 2-16 worker threads.

Figure 3.11 gives the timing results for `std::thread` and libfib fibers. The cost of fiber creation in libfib increases with the number of worker threads. The results show that thread creation is around an order of magnitude slower than fiber creation,

Figure 3.10 : Fiber-creation micro-benchmark.

```
constexpr size_t stacksize = 8;
constexpr int nthreads = 100000;
for (int i = 0; i < nthreads; ++i) {
  worker::current().new_fiber([](long long) {}, 0, stacksize);
}
```

Figure 3.11 : Task creation time for STL threads and libfib fibers.

Task Type		Mean Task Creation Time (ns)
STL thread		3,230
libfib fiber	2 threads	185
	4 threads	262
	8 threads	332
	16 threads	655

highlighting the benefit of fibers when spawning large numbers of tasks with light work.

3.7.2 Parallel Recursive Fibonacci Function

The naive parallel implementation of the Fibonacci function, using recursive calls with no depth cutoff, creates an extremely large number of asynchronous tasks, each of which do very little work. While this is not a practical algorithm, the stress it places on the runtime system's task creation and context switching mechanisms makes it a good tool for examining the overhead of those mechanisms. Therefore, we have implemented the Fibonacci function using both `std::async()` and `fibertures::async()`. Figure 3.12 shows the run time of each implementation for $N = 9$ to 29*. The STL-only version fails after $N = 15$ due to excessive creation of Pthreads, surpassing the system limit. The Fibertures version fails after $N = 29$, due to stack overflow. This micro-benchmark demonstrates that for problems requiring a large number of async tasks, Fibertures has the potential to outperform the STL both in run time and in the problem size it is able to compute without running into system resource limitations.

* $N = 9$ is the smallest input for which the time is non-zero for both systems.

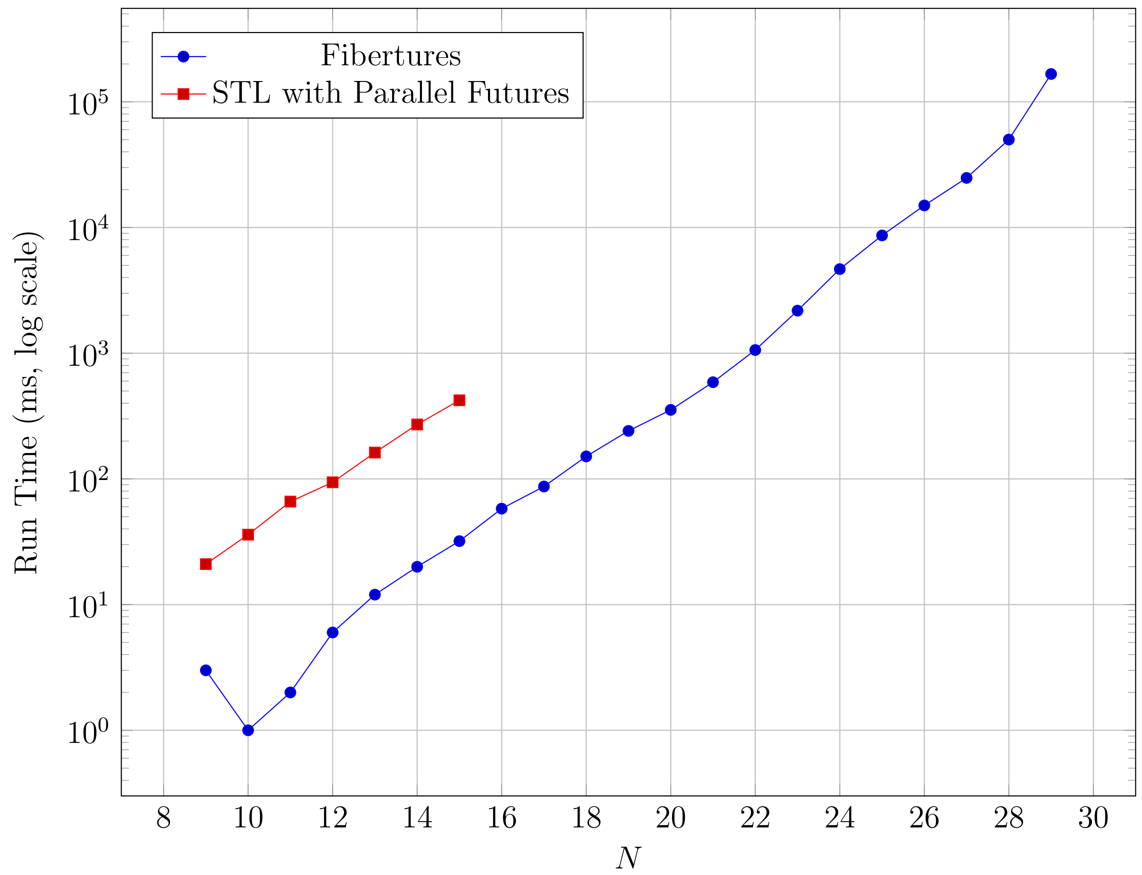


Figure 3.12 : Time to compute the Nth Fibonacci number using STL and Fibertures, using a log scale.

Chapter 4

Case Studies

In this chapter, we evaluate the performance of Fibertures relative to other futures libraries through two case studies: local sequence alignment and the fast multipole method. These benchmarks provide some insight into the comparative performance of these libraries in practical applications.

4.1 Local Sequence Alignment using Smith-Waterman

The Smith-Waterman algorithm for local sequence alignment identifies the maximally homologous subsequences between two input gene sequences [15]. The algorithm first constructs an integral scoring matrix in which every cell is a function of its left, upper-left, and upper neighbors, with the exception of the left-most and upper-most border cells, which are initialized to zero. The algorithm then obtains the two subsequences by starting at the maximal cell of the scoring matrix and tracing back towards the the upper-left, following the maximal neighboring cell at each step.

The subproblem of generating the scoring matrix exhibits a high degree of ideal parallelism. We are free to compute, in parallel, all cells whose neighbors have already been computed. For input sequences of size n and m, and for $0 < i < n$ and $0 < j < m$, cell $c_{(i,j)} = f(c_{(i-1,j)}, c_{(i-1,j-1)}, c_{(i,j-1)})$, where f is the scoring function. As stated, the top and left border cells ($i = 0$ or $j = 0$) are simply initialized to zero and thus have no dependences. Figure 4.1 illustrates the data dependence graph for

Smith-Waterman.

We can encode these data dependencies very conveniently using futures by changing the scoring matrix from integers to future integers. Then for each cell, we generate an async task to compute its value from its neighbors, getting their values only when needed. Once all cells have been assigned a future value, we force complete evaluation of the future matrix by performing a get on the lower-right cell, $c_{(n-1,m-1)}$. Note that while this parallelization scheme captures the ideal logical parallelism, in practice we use tiling to amortize the overhead of creating an async task over multiple iterations.

4.1.1 Smith-Waterman with STL

We begin with an implementation using only the STL with `std::async()` and `std::future`. The salient lines of code are shown in Figures 4.2 and 4.3.

To increase the granularity of the problem, our implementation uses tiling to generate one future per chunk of cells rather than one per cell. Lines 1-3 break the scoring matrix into a logical `tile_row_count` × `tile_column_count` array of tiles. Instead of breaking the scoring matrix into non-contiguous chunks of memory and storing those chunks in the futures themselves, we allocate a single large scoring matrix and use a separate two-dimensional array of `std::future<void>` objects (declared on line 4) to synchronize between tiles of indices. Despite these differences, the structure of the inter-cell data dependence graph for this algorithm is the same as that of the naive fine-grained approach. Since the parallel tasks now share a mutable reference to the scoring matrix, this solution is no longer strictly functional. However, the solution is still data-race-free because only one task writes to a given chunk, and no chunk is written until the chunks on which it depends have been written.

Outer for-loops iterate over tile rows and tile columns (lines 5 and 7, respectively).

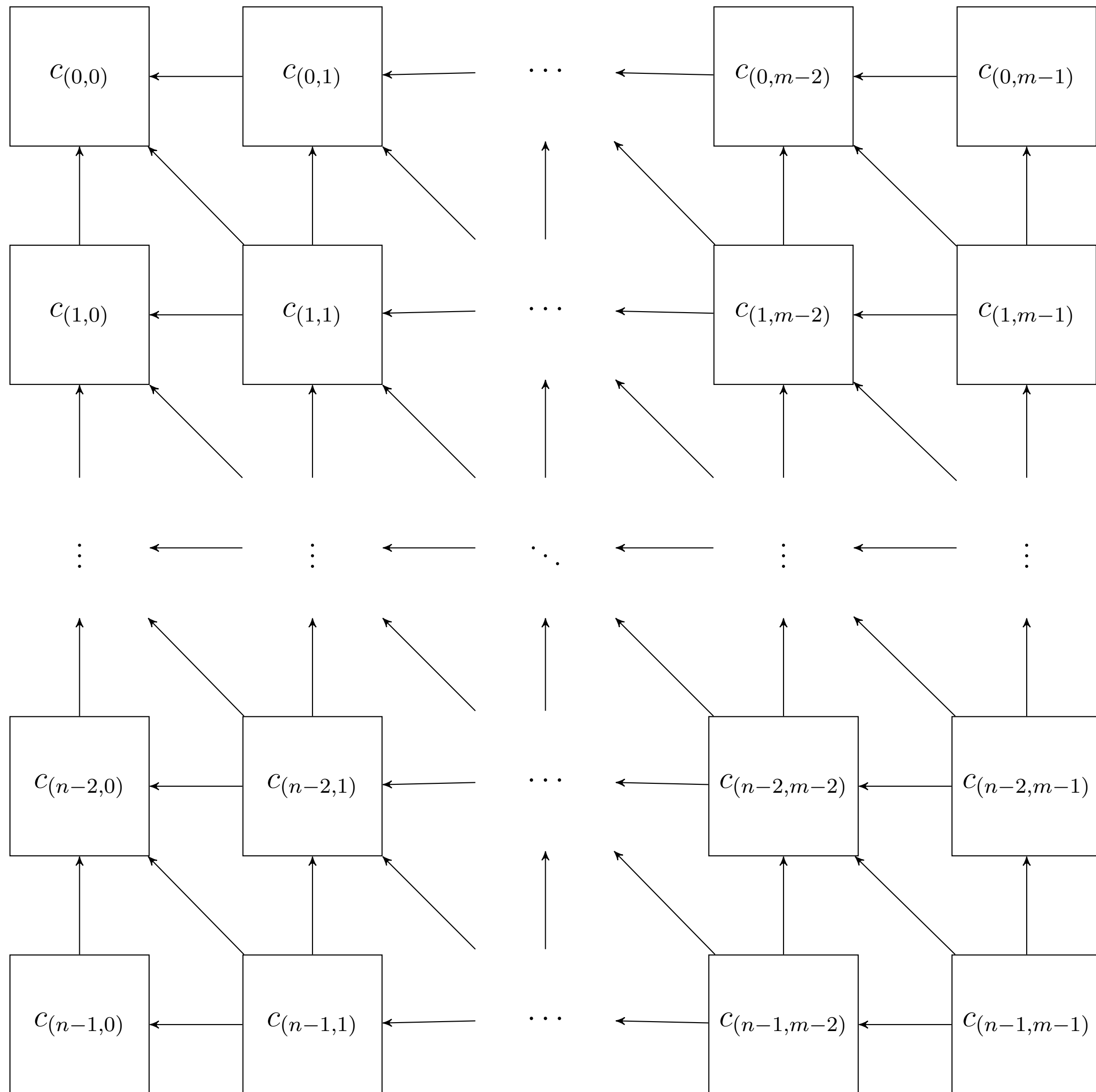


Figure 4.1 : Inter-cell data dependence graph for the Smith-Waterman local sequence alignment algorithm.

Figure 4.2 : STL-only parallel Smith-Waterman algorithm: outer loop.

```
1  constexpr int desired_tile_size = 800;
2  int const tile_row_count = max(1, m.height / desired_tile_size);
3  int const tile_column_count = max(1, m.width / desired_tile_size);
4  auto futures = vector<vector<shared_future<void>>>(tile_row_count);
5  for (int tile_i = 0; tile_i < tile_row_count; ++tile_i) {
6    futures[tile_i].resize(tile_column_count);
7    for (int tile_j = 0; tile_j < tile_column_count; ++tile_j) {
8      futures[tile_i][tile_j] = async([=, &m, &score_matrix, &futures]
9      {
10       /* See Figure 4.3 for task body. */
11     });
12   }
13 }
14 futures.back().back().wait();
15 int const score = score_matrix(m.height - 1, m.width - 1);
```

For each of these logical tiles, we spawn a task to compute all the scoring values within that tile, using `std::async()` (line 8). The task body is shown in Figure 4.3. Lines 1-6 ensure proper synchronization among tiles by waiting on the top-left, top, and left neighbor tile futures, as necessary, before computing the current tile. The actual work occurs in lines 11-31, where for each cell in the tile, we compute the three scores and write the maximum value to the scoring matrix. Once the lambda passed to `std::async()` returns, all the cells for that tile have their values, and the future associated with the tile is ready.

Once the outer loops have spawned all the tasks, we wait on the lower-right tile in line 14. Since the lower-right tile transitively depends on all other tiles, this guarantees the entire scoring matrix has been computed after that statement. Finally, we read the score from the lower-right scoring matrix cell.

Figure 4.3 : STL-only parallel Smith-Waterman algorithm: task body.

```
if (tile_i != 0 && tile_j != 0)
  futures[tile_i - 1][tile_j - 1].wait();
if (tile_i != 0)
  futures[tile_i - 1][tile_j].wait();
if (tile_j != 0)
  futures[tile_i][tile_j - 1].wait();
int const i_min = m.height * tile_i / tile_row_count;
int const i_max = m.height * (tile_i + 1) / tile_row_count;
int const j_min = m.width * tile_j / tile_column_count;
int const j_max = m.width * (tile_j + 1) / tile_column_count;
for (int i = i_min; i < i_max; ++i) {
  for (int j = j_min; j < j_max; ++j) {
    if (i == 0 || j == 0) {
      score_matrix(i, j) = 0;
    } else {
      int const diagScore = score_matrix(i - 1, j - 1);
      int const aboveScore = score_matrix(i - 1, j);
      int const leftScore = score_matrix(i, j - 1);
      int const scoreFirstWithGap = AlignmentMatrix::alignmentScore
        (m.sequence1[i], AlignmentMatrix::gap);
      int const scoreCurrentAlign = AlignmentMatrix::alignmentScore
        (m.sequence1[i], m.sequence2[j]);
      int const scoreSecondWithGap = AlignmentMatrix::alignmentScore
        (AlignmentMatrix::gap, m.sequence2[j]);
      int const above = aboveScore + scoreFirstWithGap;
      int const diag = diagScore + scoreCurrentAlign;
      int const left = leftScore + scoreSecondWithGap;
      score_matrix(i, j) = max(above, max(diag, left));
    }
  }
}
```

4.1.2 Fibertures

We apply the transformations from Chapter 3 to the STL source code to obtain an implementation using libfib with the Fibertures extension. Figure 4.4 shows code that has been added or modified from the STL version. We have omitted the additional header inclusion and the worker initialization statement.

There are two notable modifications. First, the call to `std::async()` inside the outer for-loops has been replaced by a call to `fibertures::async()`, resulting in the use of fibers instead of threads to execute async tasks. Second, the blocking calls to `future<void>::wait()` in the task lambda and just before the statement retrieving the score have been replaced with a check-and-yield-loop, allowing the current worker thread to steal work rather than blocking until the future value is available.

4.1.3 HClib

We used a conversion process similar to that of the STL-to-Fibertures conversion to obtain an HClib implementation. First we must include the HClib header and initialize the runtime. In HClib, this is accomplished by wrapping all the parallel code in a call to `hclib::launch()`. We also changed the element type of the futures array from `std::future<void>` to `hclib::future_t<void>*` and use the arrow operator in place of the dot operator when performing operations on the futures. Lastly, we substitute a call to `hclib::async_future()` for the call to `std::async()`, spawning a lightweight task instead of a system thread. As in the Fibertures version, the HClib version aims to reduce task creation overhead and eliminate blocking synchronization.

Figure 4.4 : Modifications to parallel Smith-Waterman algorithm to use Fibertures.

```
futures[tile_i][tile_j] = fibertures::async
    ([=, &m, &score_matrix, &futures] {
  if (tile_i != 0 && tile_j != 0) {
    auto status = futures[tile_i - 1][tile_j - 1].wait_for(0ms);
    while (status != future_status::ready) {
      worker::current().yield();
      status = futures[tile_i - 1][tile_j - 1].wait_for(0ms);
    }
  }
  if (tile_i != 0) {
    auto status = futures[tile_i - 1][tile_j].wait_for(0ms);
    while (status != future_status::ready) {
      worker::current().yield();
      status = futures[tile_i - 1][tile_j].wait_for(0ms);
    }
  }
  if (tile_j != 0) {
    auto status = futures[tile_i][tile_j - 1].wait_for(0ms);
    while (status != future_status::ready) {
      worker::current().yield();
      status = futures[tile_i][tile_j - 1].wait_for(0ms);
    }
  }
  ...
});
...
auto status = futures.back().back().wait_for(0ms);
while (status != future_status::ready) {
  worker::current().yield();
  status = futures.back().back().wait_for(0ms);
}
```

4.1.4 Timing Results

We evaluate the performance of each implementation of parallel Smith-Waterman using the same hardware as in Section 3.7, i.e. the Intel Ivy Bridge architecture, with 8 GB of memory and an Itel Core i7-3770K processor with four cores and eight hardware threads. Thus, for compute-bound (rather than I/O-bound) applications such as the examples studied in this chapter, we would expect to achieve the best performance with four or eight worker threads.

Table 4.1 shows the timing results for input sequences with 18,560 and 19,200 nucleotides. Each row lists the sample mean and standard deviation over ten runs. For the Fibertures and HClib implementations, we give results for thread pool sizes of 1, 2, 4, 8, and 16. Since `std::async()` does not use thread-pooling, we give only one set of results for the pure STL implementation. For each runtime system, we used the tile size that produced the shortest run time. The STL and Fibertures implementations performed best with tiles approximately 800 cells by 800 cells, for a total of 552 tiles. HClib performed best with tiles approximately 400 cells by 400 cells, totalling 2,208 tiles. Recall that each tile corresponds to one async task. The results show that Fibertures and HClib performed best using eight worker threads, as expected. The performance of the STL with `std::async()` and `std::future` was comparable to that of both HClib and Fibertures, although Fibertures was slightly faster.

Table 4.2 shows results for the same experiment as Table 4.1 except that we used the same tile size for each runtime system in this case. Note that the STL-only implementation fails in this case (throwing a `std::system_error`) due to the large number of Pthreads it attempts to spawn. This highlights one of the weaknesses of current `std::async()` implementations, which is that applications that spawn

Table 4.1 : Execution times of Smith-Waterman by runtime system.

Runtime System		Execution Time (ms)		Number of Tiles
		Mean	Std. Dev.	
Sequential STL		26,335	34	N/A
STL with Parallel Futures		6,349	18	552
HClib	1 thread	26,650	17	2,208
	2 threads	13,540	40	
	4 threads	7,113	83	
	8 threads	6,463	58	
	16 threads	7,126	221	
Fibertures	1 thread	25,510	11	552
	2 threads	13,153	64	
	4 threads	7,284	34	
	8 threads	6,218	26	
	16 threads	6,780	74	

very large numbers of async tasks may quickly run into platform-dependent system resource limitations.

4.2 Fast Multipole Method

The fast multipole method (FMM) is an important algorithm originally designed to find approximate solutions to N-body problems, which has since found widespread application in other fields. In [16], the performance of standard C++, OpenMP, HPX, and Cilk were compared using the FMM benchmark. For this case study, we ported

Table 4.2 : Execution times of Smith-Waterman by runtime system. In this case, all runtimes use the same number of tiles: 2,208.

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

Table 4.3 : Execution times of FMM benchmark by runtime system, problem size 20,000.

Runtime System		Execution Time (s)	
		Mean	Std. Dev.
STL with Parallel Futures		1.2	0.044
Fibertures	1 thread	8.9	0.089
	2 threads	8.0	0.057
	4 threads	8.0	0.089
	8 threads	9.3	0.061

the FMM benchmark to use the Fibertures library and obtained results comparing Fibertures to the STL with parallel futures.

We have adjusted the problem sizes to account for differences in hardware and optimization levels* between this study and [16]. We used data type II (points distributed over the surface of the unit sphere) and calculated the solution to six decimal places. Tables 4.3 and 4.4 show results from the FMM benchmark by runtime system, for problem sizes of 20,000 and 100,000, respectively. Each row gives the sample mean and standard deviation over ten successful runs.

There are no timing results for the STL implementation in Table 4.4 because it always fails due to excessive thread creation. The STL implementation often fails even for the size 20,000 problem. The STL entry in Table 4.3 represents the first ten successful runs, with seven out of twenty runs aborting. However, when the STL implementation did terminate successfully, it was faster than the best Fibertures

*We use the default optimization level due to a bug in libfib preventing the use of higher levels.

Table 4.4 : Execution times of FMM benchmark by runtime system, problem size 100,000.

Runtime System		Execution Time (s)	
		Mean	Std. Dev.
STL with Parallel Futures		N/A	N/A
Fibertures	1 thread	78	0.69
	2 threads	73	1.2
	4 threads	73	1.3
	8 threads	81	0.62

runs by a factor of 6.7 on average. Furthermore, the Fibertures implementation exhibits poor scaling as the number of worker threads is increased. These results are inconsistent with the other benchmarks and require further investigation.

Chapter 5

Related Work

In this chapter we discuss prior work related to the Fibertures library and runtime system.

5.1 Polling and Spin-waiting

The Fibertures strategy for awaiting a future value, i.e. executing a yield-loop in the current task until the value is available, is reminiscent of two existing software techniques in asynchronous programming: polling and spin-waiting.

Polling is common in event-driven programming as a means for detecting some condition and responding by invoking the appropriate event handler. The interaction between async tasks and futures can be described in terms of event-driven programming, where the event is the act of setting a promise's value, and the handler invokes any task continuations that were awaiting the corresponding future. The key difference between polling and Fibertures yield-loops is that a yield-loop only checks for an "event", i.e. a satisfied future, when the task that is executing it resumes. There is no fixed, centralized polling loop. This means that the higher the number of ready tasks compared to the number of blocked tasks, the less time the Fibertures scheduler spends checking future statuses.

Spin-waiting is a technique used in a number of scenarios, including mutexes and resolution of I/O contention. The critical difference between a spin-waiting loop and

a yield-loop is that spin-waiting is a blocking operation. A spin-waiting thread will simply sleep until its requested resource is available (or until it times out), rather than attempting to find useful work in the meantime. On the other hand, a worker thread in the Fibertures runtime is capable of context-switching to a new task when its current task becomes blocked while awaiting a future.

5.2 Qthreads

The Qthread Library from Sandia National Laboratories (not to be confused with QT's QThreads) provides support for light-weight tasks in C, comparable to fibers [17]. Qthreads support the use of full/empty bits (FEBs) to perform a kind of data-driven synchronization between qthreads. However, unlike `std::futures`, this pattern lacks the structure and strong safety guarantees that STL futures provide, and the Qthread API is not compatible with `<future>`. Therefore, Qthreads by themselves are not a suitable replacement for `std::async()`. They could, however, serve as a building block for a higher-level futures interface, similar to how libfib serves as the basis for Fibertures. Note that while the Qthread Library also has a construct called "future", this construct is unrelated to C++ STL futures. Figure 5.1 is an example of a basic parallel "Hello World" program using Qthreads. Note the use of FEBs for synchronization on line 12.

5.3 Folly Futures

Facebook has written their own promise-and-future library for C++11, independent of `<future>`, called Folly Futures [2]. The purpose of this library is to make composition of futures easier and cleaner. Folly Futures have similar functionality to C++11 `std::futures` but also support callback chaining via the `then()` and `onError()`

Figure 5.1 : "Hello World" program using Qthreads, adapted from [1].

```
aligned_t greeter(void *arg) {
    printf
    ( "Hello World! My input argument was %lu\n"
    , (unsigned long)(uintptr_t)arg
    );
    return 0;
}
int main() {
    aligned_t return_value = 0;
    qthread_initialize();
    qthread_fork(greeter, nullptr, &return_value);
    qthread_readFF(NULL, &return_value);
    printf("greeter returned %lu\n", (unsigned long)return_value);
    return EXIT_SUCCESS;
}
```

methods. The C++ standard does not currently support callback chaining; however, the Concurrency TS does support chaining via `std::future::then`, `std::when_all`, and `std::when_any`. Folly Futures are incompatible with `<future>` due to significant API differences, meaning existing code written using STL futures cannot benefit from Folly Futures without some refactoring. Furthermore, the Folly Futures library does not itself provide a mechanism for asynchronous tasks, such as `std::async()` or `fiber_async()`. Rather, it allows the user to optionally specify an `Executor` to schedule or execute a `Future` upon creation [18]. `Executor` is Facebook's abstract base class representing a task execution policy. Facebook provides several choices of executor, including `ThreadPoolExecutor`, which is a simple thread pooling implementation. Unlike with `<future>` or Fibertures, the choice of execution policy is left to the user. Folly Futures is a good alternative to Fibertures if the user requires more

Figure 5.2 : Callback chaining using Folly Futures [2].

```
Future<OutputA> futureA(Output);
Future<OutputB> futureB(OutputA);
Future<OutputC> futureC(OutputB);
// then() automatically lifts values (and exceptions) into a Future.
OutputD d(OutputC) {
  if (somethingExceptional) throw anException;
  return OutputD();
}
Future<double> fut =
  fooFuture(input)
  .then(futureA)
  .then(futureB)
  .then(futureC)
  .then(d)
  .then([](OutputD outputD) { // lambdas are ok too
    return outputD * M_PI;
  });
```

control over the execution policy. However, it requires more manual effort on the user's part when porting existing code using `std::future` to benefit from lightweight threading.

The Folly Futures example in Figure 5.2 demonstrates the expressiveness of callback chaining with the `then()` member function.

5.4 Boost Fiber

Boost also recently released a lightweight cooperative threading library called Boost Fiber, which includes futures among its synchronization constructs [19]. Similarly to Fibertures, blocking future operations do not block the underlying worker thread but merely suspend the current fiber, allowing the worker thread to continue doing

Figure 5.3 : Parallel recursive Fibonacci function using Boost Fiber futures.

```
1 int fib(int n) {
2   if (n <= 1) return n;
3   boost::fibers::future<int> left = boost::fibers::async([n] {
4     return fib(n - 2);
5   })
6   int right = fib(n - 1);
7   return left.get() + right;
8 }
```

useful work. Boost Fiber futures are a different type (`boost::fibers::future`) than `std::future` and even `boost::future`; however, the interfaces are all very similar, allowing easy refactoring from one type to another. Boost Fiber allows the use of user-specified schedulers, defaulting to a round-robin scheduler. Due to the similarity between `std::future` and `boost::fibers::future`, the Boost Fiber library provides a similar level of backwards compatibility with existing STL-only code to Fibertures, with the usual drawback that the user must be aware of thread-local storage when performing the conversion from `<future>` to Boost Fiber.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

To remain at the forefront of high-performance high-level programming languages, C++ must support expressive and efficient means of expressing parallelism. We have seen that while C++ futures confer programmability and safety benefits, current standard (STL) implementations have significant drawbacks in terms of performance. These drawbacks are particularly apparent in applications that require large numbers of async tasks, where the overhead of task creation, context-switching, and blocking synchronization can overwhelm the computation time and system resources. As an alternative, we see that for this class of problems, third-party libraries that support parallel futures can effectively solve these problems through the use of thread-pooling and work-stealing, reducing overhead and more efficiently utilizing system resources.

We implemented the Fibertures extension to libfib to improve the performance of parallel futures in C++ by utilizing fibers. We have shown that Fibertures performs well compared to the STL and other futures libraries in some benchmarks. Fibertures' close adherence to standard C++ APIs in conjunction with our set of source-to-source transformations from pure standard C++ to Fibertures facilitate conversion of existing parallel C++ code, allowing application programmers to leverage the improved performance of Fibertures in existing code with less manual effort.

6.2 Future Work

6.2.1 Fibertures Runtime and Transformation Improvements

Our case studies show that Fibertures has the potential to compete with other C++ future runtime systems. However, the validity of these results is limited to compilation at the default optimization level, due to limitations of libfib. An important next step in the evaluation and use of Fibertures is to fix libfib to support compilation under higher optimization levels and reevaluate the performance of each runtime system in that context.

The Fibertures runtime currently relies on a modified scheduler loop that prioritizes stealing tasks over popping tasks from the local run queue. This prevents a worker thread that has yielded a blocked task from repeatedly executing and suspending that task, improving load balancing. However, the scheduler may still resume blocked tasks more frequently than is necessary, and this scheduling algorithm may reduce data locality as a side effect. It could be worthwhile to explore alternative techniques for handing task dependences, such as keeping a list of dependences for each task and only scheduling a task to run when all of its dependences are satisfied.

Currently, we implement the source-to-source transformations and analysis in this work by hand per application. To facilitate the use of Fibertures for development, it would be useful to automate these transformations using a compiler tool such as LibTooling [20]. Such a tool would appreciably reduce programmer burden when converting large bodies of existing code to use Fibertures. Furthermore, it would be valuable to extend the transformations in this work to support other runtime systems, such as HClib and HPX.

Lastly, the current Fibertures implementation requires manual specification per

application of the fiber stack size. The choice of stack size has a large impact on performance. If the stack size is too low, the program will experience a segmentation fault; if it is too high, performance suffers from excess memory usage and allocation time. Therefore, it could be valuable to explore ways to infer the best stack size automatically. Furthermore, within a given program, it is not necessarily the case that all fiber stacks have the same ideal size. We could investigate compile-time and run-time analysis techniques to determine the best stack size for each instance of fiber stack creation, rather than using the same value across the entire program. There is prior work on this problem, such as SLAW, a runtime scheduler that can bound the stack and heap space required by tasks[21]. Some of these techniques may be applicable to Fibertures.

6.2.2 Fibertures as a Foundation for Other High-level Parallel Features

In Chapter 1, we noted that async tasks and futures are a general parallel programming construct, meaning we can use them to implement other models of parallel programming. A natural next step would be to use the Fibertures library to build a full framework of parallel tools for C++, beyond those found in the STL. This extended library could include constructs found in other C++ libraries such as HPX and HClib as well as features from other languages, such as Java's parallel streams.

Besides the implementation of the additional library features themselves, there could be useful work to be done in developing extensions to libfib and Fibertures to facilitate more efficient implementation of some of those features. For instance, the ability to set thread affinity hints for async tasks could be useful for the development of a parallel producer-consumer API by improving cache performance through improved data locality.

Bibliography

[1] D. Stark, "Qthreads." https://github.com/Qthreads/qthreads/, 2012. Accessed: 2017-06-24.

[2] H. Fugal, "Futures for C++11 at Facebook." https://code.facebook.com/posts/1661982097368498/futures-for-c-11-at-facebook/, 2015. Accessed: 2017-06-24.

[3] Standard C++ Foundation, "C++11 Standard Library Extensions Concurrency." https://isocpp.org/wiki/faq/cpp11-library-concurrency, 2017. Accessed: 2017-06-24.

[4] R. Surendran and V. Sarkar, "Automatic parallelization of pure method calls via conditional future synthesis," *OOPSLA*, vol. 51, 2016.

[5] H. Sutter, C. Curruth, and N. Gustafsson, "async and future (Revision 4)." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3773.pdf, 2013. Accessed: 2017-06-24.

[6] The Habanero Extreme Scale Software Research Group, Rice University, "HClib." https://github.com/habanero-rice/hclib, 2017. Accessed: 2017-06-24.

[7] O. Kowalke, "Boost Context." http://www.boost.org/doc/libs/1_64_0/libs/context/doc/html/index.html, 2017. Accessed: 2017-06-24.

[8] S. Tasirlar and V. Sarkar, “Data-driven tasks and their implementation,” *ICPP*, 2011.

[9] C++ reference, “Extensions for concurrency.” `http://en.cppreference.com/w/cpp/experimental/concurrency`, 2016. Accessed: 2017-06-24.

[10] The STE||AR Group, “HPX.” `http://stellar-group.org/libraries/hpx/`, 2017. Accessed: 2017-06-24.

[11] The STE||AR Group, “HPX.” `https://github.com/STEllAR-GROUP/hpx`, 2017. Accessed: 2017-06-24.

[12] S. Dolan, S. Muralidharanet, and D. Gregg, “Compiler support for lightweight context switching,” *TACO*, vol. 9, 2013.

[13] S. Dolan, “libfib.” `https://github.com/stedolan/libfib`, 2011. Accessed: 2017-06-24.

[14] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, “The essence of compiling with continuations,” in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI ’93, (New York, NY, USA), pp. 237–247, ACM, 1993.

[15] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[16] B. Zhang, “Asynchronous task scheduling of the fast multipole method using various runtime systems,” in *Data-Flow Execution Models for Extreme Scale Computing, 2014 Fourth Workshop on*, DFM ’14, IEEE, 2014.

[17] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, IPDPS '08, IEEE, 2008.

[18] Facebook, "Folly Futures." `https://github.com/facebook/folly/tree/master/folly/futures`, 2016. Accessed: 2017-06-24.

[19] O. Kowalke, "Fiber." `http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/index.html`, 2016. Accessed: 2017-06-24.

[20] The Clang Team, "LibTooling." `https://clang.llvm.org/docs/LibTooling.html`, 2017. Accessed: 2017-06-24.

[21] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," *PPoPP*, vol. 45, 2010.