Journal of
Computer Science and Software Development

Research Article Open Access

An Experience Report on Integrating JavaScript Style Promises into the C++Runtime

Nyalia Lui* **and James H Hill**

Indiana University-Purdue University Indianapolis, Department of Computer and Information Science, Indianapolis, IN, United States

***Corresponding author:** Nyalia Lui, Indiana University-Purdue University Indianapolis, Department of Computer and Information Science, Indianapolis, IN, United States, Tel: 7858450800, E-mail: nlui@iupui.edu

Received Date: March 28, 2022 **Accepted Date:** April 28, 2022 **Published Date:** April 30, 2022

Citation: Nyalia Lui (2022) An Experience Report on Integrating JavaScript Style Promises into the C++ Runtime. J Comput Sci Software Dev 1: 1-14.

Abstract

This article presents our experience designing and implementing a Promise library for C++ named Promise++. Our Promise library is implemented using template meta-programming to pass system state between asynchronous tasks. Wealso introduce other C++ Promise libraries and present our experience using each in comparison to Promise++. Unlike existing C++ promise libraries, our implementation includes features to resolve a map of Promises well as fetching theresult from a Promise. Based on our experience implementing and using C++ promise libraries, we learned it is possibleto reduce cyclomatic complexity by ~25% at the cost of performance.

Keywords: Promises; A+; C++; JavaScript

Introduction

Promises were first mentioned as a concept in 1976 by Daniel P. Friedman and David Wise, which was later updated in their 1978 revision [1, 2]. Friedman et al. intent was to provide high-level tools for expressing parallel code. A piece of their work showed that a placeholder for the result, called a Promise, can reference the value eventu- ally returned. Other researchers have also derived their own implementations of similar objects, such as the Even- tual [3], Future [4], and Defer [5]. These objects make par- allel and asynchronous programming easier by using *con- tinuation style programming (CSP)* where instances can be passed as arguments or returned from functions [6, 7]. In the present day, Promises supported in popular pro- gramming languages, such as JavaScript, are based on the Promises/A+ standard [8]. Prior to Promises/A+, asyn- chronous programming in JavaScript was done with many nested callback functions, which increase code complex- ity [9]. The Promises/A+ standard was therefore cre- ated to reduce the number of callback functions needed for asynchronous programming [10].

Other languages, such as C++, do not have a standard for Promises as there are many libraries with different ap- proaches to CSP. For example, there is Promise-cpp [11] offers fast Promise consumption by taking advantage of Defer design methodology and in-place allocation. Like- wise, PoolQueue[12] extends the Boost Future [13] design to return a Promise from then() methods. The C++11 Standard Template Library (STL) [14] has Promises used together with Futures. The C++ community has also ex- tended the STL with CSP by attaching then() methods to the C++ Future. This is seen in frameworks such as High- Performance ParallelX (HPX)[15] and Software Technol- ogy Lab (STLab) [16].

Although existing C++ libraries allow developers to leverage Promises or CSP in their design, existing ap- proaches require additional constructs that increase a pro- gram's complexity. To address this issue, we created *Promise++*. Promise++ is a library that implements Promises adhering to the Promises/A+ specification. The goal of this work is to decrease the complexity in applica- tions using Promises and CSP.

In this article, we describe our experience using different approaches to Promises and CSP in the C++ runtime. We used each library on three asynchronous programming ex- amples and discuss the design trade-offs between libraries in the context of cyclomatic complexity and performance. Based on this understanding, the main contributions of this article are as follows:

·It demonstrates how to use existing Promise libraries in C++;

·It discusses, in detail, the design and implementation of Promise++, and how it realizes the Promises/A+ specification; and It discusses design trade-offs between different C++ Promise libraries.

Article organization: The remainder of this article is organized as follows: Section 2 describes JavaScript Promises and provides background information on exist- ing Promise libraries in C++; Section 3 introduces the design behind Promise++; Section 4 presents our exam- ple applications and results on cyclomatic complexity and performance; Section 5 discusses lessons learned; Section 6 discusses related work; and, Section 7 provides concluding remarks.

Background

Promises & Existing C++ Li- braries

In this section, we discuss Promises in JavaScript and the C++ libraries introduced in Section 1. We also illus- trate how to create and use Promises for each library with a simple program.

JavaScript and Promise/A+ Specification

Listing 1 provides a simple example that illustrates Promise code in JavaScript, which uses the Promise/A+ specification. Lines 1-3 show a Promise created with a pairwise tuple of `resolve()` and `reject()` callback func- tions. The user may call either function to set the con- structed Promise to a value with `resolve()`, or to an ex- ception with `reject()`. This action is called *settling* the Promise. In the listing, we resolve the Promise to the string "Hello World!".

```
let prom = new Promise((resolve, reject) => {
  resolve ("Hello World!");
});

prom.then (value => {
  console.log (value);
  return 51;
}).then (value => {
  console.log (value);
});

prom.then (value => {
  console.log (`The value is still ${value}`);
});
```

Listing 1: A simple example illustrating the structure and behaviorof Promise code in JavaScript

To enable continuous asynchronous execution, every Promise has a then () method that takes two function arguments. The first is a resolution function that de- fines code to execute after the Promise resolves to a value. The second argument is an optional rejection function (not shown in Listing 1) that defines code to run after a Promise is rejected. Additionally, there exists a catch () method that is a wrapper around then () for passing a rejection function only.

In the A+ specification, the resolve () and reject (function take an argument. The resolve () function is passed the value that the calling Promise will hold. The reject () function is passed the reason (*e.g.* an exception) for execution failure. The then () and catch () methods must return a value or a Promise. This allows developers to chain asynchronous tasks that immediately execute af- ter the calling Promise is settled (*i.e.*, either resolved or rejected).

```
auto deferred = promise::newPromise();

deferred.then([](std::string value) {
    std::cout << value << std::endl;
    return 51;
}).then([](int value) {
    std::cout << value << std::endl;
});

deferred.resolve("Hello World!");

auto new_chain = promise::newPromise();

new_chain.then([](std::string value) {
    std::cout << "Value is still " << value << std::endl;
});

new_chain.resolve("Hello World!");
```

Listing 2: The Promise-cpp version of the JavaScript code sample where two separate Defers are created. The first starts a then()chain and the second begins another using the same resolved value

Promise chaining is illustrated in lines 5-10 of Listing 1. The parameter value holds the string "Hello World!" and is printed to the console before the function returns value 51. When control exits the function a Promise is created that contains value 51. In lines 8-10, value is 51 is received as an argument and printed to the console. Chaining can occur any number of times. Lines 12-14 of Listing 1 show a new chain that simply prints "The value is still Hello World!" to the console.

Existing C++ Libraries

Promise-cpp

Promise-cpp [11] is a library that allows developers to write Promise code in C++. In Promise-cpp, a Promise is represented using the Defer object that is constructed via a factory method [17] and Listing 2 illustrates the example from Listing 1 written with the Promise-cpp. In line 1, a Defer is created with the default constructor and a then() chain is formed in lines 3-8 where lambda expressions are the function handles for then() methods. The Defer is resolved to "Hello World!" in line 10 which will initiate asynchronous execution. Lines 12-18, a new Defer is cre- ated to start a new chain for asynchronous execution. A new Defer is required because Defer state is cleared after a chain completes.

PoolQueue

PoolQueue [12] is another library that enables Promises in C++. Listing 3 illustrates the code from Listing 1 writ- ten using PoolQueue. In this listing, a Promise is created using the default constructor in line 1. The then() chain is also formed with lambda expressions in lines 3-9. In PoolQueue, all then() methods must return a value, so the developer suggests nullptr is used for the last call- back in a chain. The created Promise is settled to "Hello World!" in line 16.

C++ Standard Template Library

The Standard Template Library (STL) has support for Promises. Listing 4 illustrates the code from Listing 1 written with the STL library. In lines 1-6 of Listing 4 two Promises are created along with two Futures that will be used to retrieve a string and integer. These values will be stored into two variables "out1" and "out2." Lines 8- 14 show a Thread passed a Promise where its value is set to "V12 Engine!" One of the Futures is used to suspend execution until the value is ready before a call to join the Thread.

Lines 16-29 show the short then () chain formed using a combination of Threads, Promises, and Futures.

```
1  poolqueue::Promise prom;
2
3  prom.then([](const std::string &value) {
4    std::cout << value << std::endl;
5    return 51;
6  }).then([](const int value) {
7    std::cout << value << std::endl;
8    return nullptr;
9  });
10
11 prom.then([](const std::string &value) {
12   std::cout << "Value is still " << value << std::endl;
13   return nullptr;
14 });
15
16 prom.settle(std::string("Hello World!"));
```

Listing 3: The PoolQueue version of the JavaScript code sample where two then () chains are began using the same Promise. Note the call to settle a Promise at the bottom

The first thread must be passed the previous value from "out1" and the next Promise to be settled. This Promise is resolved to value 51. Again a Future is used to retrieve the resolved value and assigned to another variable, "out2," which is passed into the following thread for the last step in our chain. No Promise is passed into this Thread since the chain terminates. A third Thread for starting a new chain uses the first Promise's value from "out1" in lines 31-33.

the execution strategy for asynchronous tasks. The return value from the lambda expression, the second parameter, will be held by the Future. Lines 5-10 & 12-14 show the then () chains; however, separate Futures are generated from each method and must be captured. The user must explicitly wait on those new Futures, shown in Lines 16-18 & 20-22, to ensure safe termination.

```
1  std::promise<std::string> prom1;
2  std::future<std::string> result1 = prom1.get_future();
3  std::string out1;
4  std::promise<int> prom2;
5  std::future<int> result2 = prom2.get_future();
6  int out2;
7
8  std::thread creator([](std::promise<std::string> p) {
9    p.set_value("Hello World!");
10 }, std::move(prom1));
11
12 result1.wait();
13 out1 = result1.get();
14 creator.join();
15
16 std::thread then1([](std::string &value, std::promise<int> p) {
17    std::cout << value << std::endl;
18    p.set_value(51);
19 }, std::ref(out1), std::move(prom2));
20
21 result2.wait();
22 out2 = result2.get();
23 then1.join();
24
25 std::thread then2([](int &value) {
26   std::cout << value << std::endl;
27 }, std::ref(out2));
28
29 then2.join();
30
31 std::thread then3([](std::string &value) {
32   std::cout << "The value is still " << value << std::endl;
33 }, std::ref(out1));
34
35 then3.join();
```

Listing 4: The STL version of the JavaScript code sample where Promises are managed using a combition of STL futures Promises and threads

High-Performance ParallelX

High-Performance ParallelX (HPX) [15] extends the STL Future to handle asynchronous tasks with CSP. List- ing 5 shows the JavaScript example from Listing 1 written with HPX.In lines 1-3, async() executes a lambda expres- sion asynchronously where the return value is held within the produced Future. In lines 5-10, the first then() chain is shown where Futures are the output and input parame- ters for asynchronous tasks. To start a new then() chain in HPX, a new Future must be created as illustrated in lines 12-18.

Software Technology Lab

Software Technology Lab (STLab) [16] also re-defines Futures in C++ by enabling then() methods with their design. Listing 6 shows the JavaScript example from List- ing 1 written with STLab. In lines 1-3, a Future is created using a factory method where the first parameter specifies

Design & Implementation

In this section we introduce the design and implemen- tation to Promise++ grown on top of the C++ paradigm. Promise++ features several reusable classes and interfaces to allow easy configuration of a Promise enabled system. These modules were written following the C++11 stan- dard.

Listing 7 shows code from Listing 1 written with the Promise++ library. As illustrated on lines 1-3, a Promise is constructed by calling a factory method [17]. An Settlement object from Figure 2 is used to resolve the Promise to “Hello World!”. Similar to other frameworks, the then() chain is formed on lines 5-10 with lambda expressions. The number 51 is passed down stream by creating a resolved Promise. The library will extract and send 51 to the callback on line 8. Lastly, a new chain is formed one lines 12-14.

```
1  auto future1 = hpx::async([]() {
2    return std::string("Hello World!");
3  });
4
5  future1.then([](hpx::future<std::string> value) {
6    hpx::cout << value.get() << hpx::endl;
7    return 51;
8  }).then([](hpx::future<int> value) {
9    hpx::cout << value.get() << hpx::endl;
10 });
11
12 auto future2 = hpx::async([]() {
13   return std::string("Hello World!");
14 });
15
16 future2.then([](hpx::future<std::string> value) {
17   hpx::cout << "The value is still " << value.get() << hpx::
         endl;
18 });
```

Listing 5: The HPX version of the JavaScript code sample where Futures are used instead of Promises

```
auto future = stlab::async(stlab::default_executor, []() {
  return std::string("Hello World!");
});

auto temp1 = future.then([](std::string value) {
  std::cout << value << std::endl;
  return 51;
}).then([](int value) {
  std::cout << value << std::endl;
});

auto temp2 = future.then([](std::string value) {
  std::cout << "The value is still " << value << std::endl;
});

while (!temp1.get_try()) {
  std::this_thread::sleep_for(std::chrono::milliseconds(1));
}

while (!temp2.get_try()) {
  std::this_thread::sleep_for(std::chrono::milliseconds(1));
}
```

Listing 6: The STLab version of the JavaScript code sample whereFutures are used instead of Promises

```
auto prom = promise([](Promises::Settlement settle) {
    settle.resolve<std::string>("Hello World!");
});

prom->then([](std::string value) {
    std::cout << value << std::endl;
    return Promises::Resolve(51);
})->then([](int value) {
    std::cout << value << std::endl;
});

prom->then([](std::string value) {
    std::cout << "The value is still " << value << std::endl;
});
```

Listing 7: The Promise++ version of the JavaScript code sample

To represent a Promise's state, we used the State Soft- ware Pattern [17]. As seen in Figure 1, there are two concrete objects which refer to a resolved Promise and a rejected Promise. Respectively they are named the Re solved State and Rejected State. These objects are where type information for the resolved value and rejected exception are captured. Pending State refers to a Promise that has not settled yet. Once a Promise is Resolved or Rejected, it can never go back to a Pending state.

When a user creates their own Promise in JavaScript, a resolve() and reject() function is used to settle a Promise to a value or error. This is implemented in many JavaScript as a pair-wise tuple such as line 1 in Listing 1. As seen in Line 1 of Listing 7, the Settlement object rep- resents that pair-wise tuple and is passed into the call- back function. A Settlement takes a Promise at creation time, shown in Figure 2, and holds for later use. The user has access to two methods, Settlement.resolve() and Settlement.reject(), that will resolve or reject the held Promise.

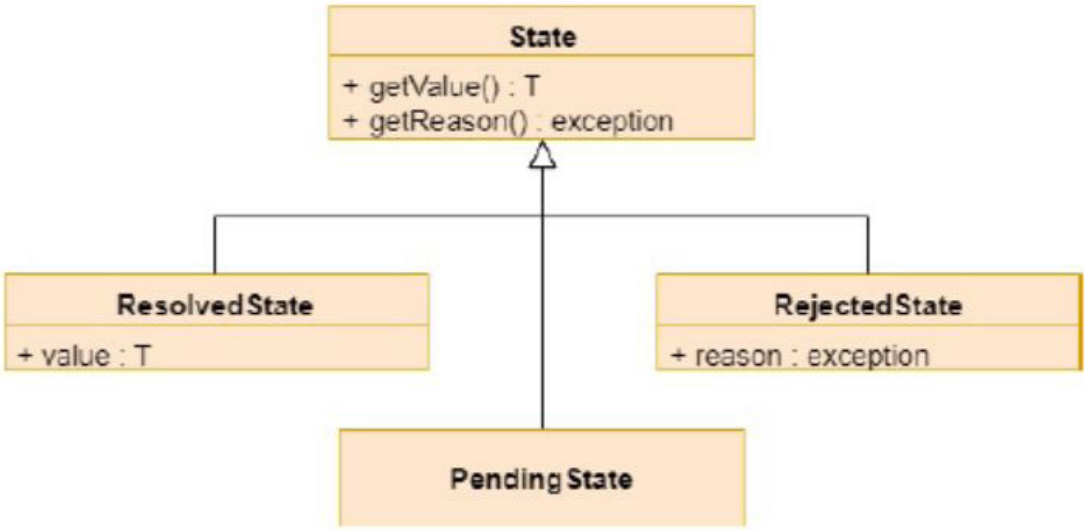

Figure 1: Defines three classes for representing Promise resolved, rejected, and pending state. Those classes are Resolved State, Rejected State, and Pending State

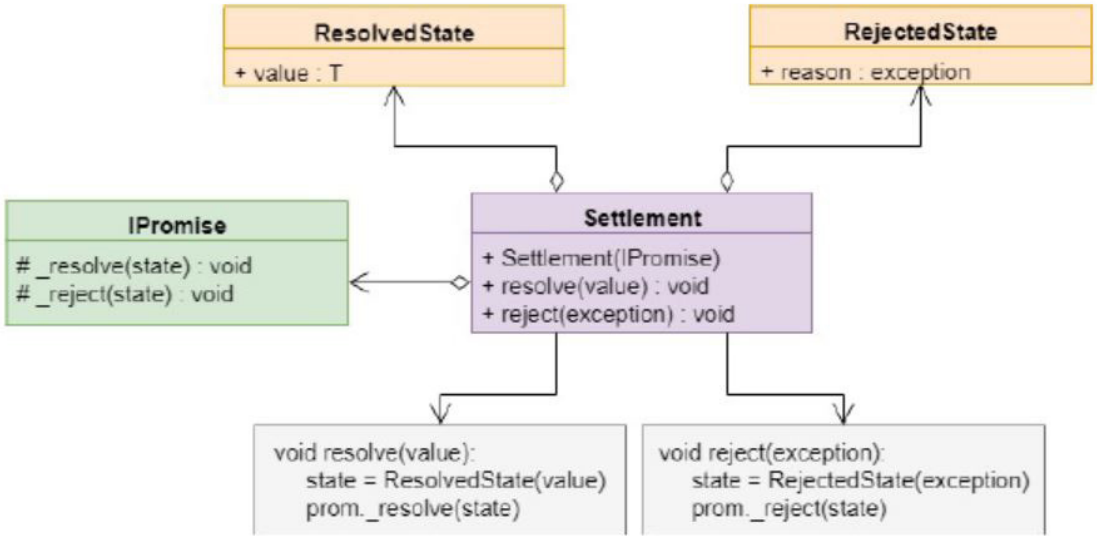

Figure 2: Shows the design for a settlement object which resolves orrejects a Promise

Handling many Promises & Fetching Results

Promise++ features two procedures for processing many asynchronous tasks. The first is Promises::all() which takes a C++ Vector of Promises and returns a Promise resolved to a Vector with the values at the same indexes. If any of the Promises fail, the returned Promise will take the first rejected Promise.

The second way users can process many asynchronous tasks is to use Promises::hash() which takes a C++ Map instead of a Vector. The input domain to the Map can be any data type and the output domain are Promises. The returned Promise is resolved to a separate Map with the same input domain but the output domain will be the values. Again, the returned Promise will take the first rejected Promise if any of them fail.

```
1  Promises::PROM_TYPE dummy(int value) {
2    return promise([&value](Promises::Settlement settle) {
3      settle.resolve<int>(value);
4    }));
5  }
6  typedef std::pair<std::string, Promises::PROM_TYPE>
       prom_pair;
7  std::map<std::string, Promises::PROM_TYPE> promises;
8
9  promises.insert(prom_pair("promise1", dummy(10)));
10 promises.insert(prom_pair("promise2", dummy(20));
11 promises.insert(prom_pair("promise3", dummy(30));
12
13 //returns a Promise resolved to a Map
14 auto prom = Promises::hash<std::string, int>(promises);
15
16 //values is the Map
17 auto values = Promises::await<std::map<std::string, int>>(
       prom);
18 assert(values["promise1"] == 10);
19 assert(values["promise2"] == 20);
20 assert(values["promise3"] == 30);
```

Listing 8: Handle many Promises in a C++ Map and reference theresults with the same keys

Listing 8 shows an example using Promises::hash(). Lines 1-5 is a function for creating a Promise. Lines 6-11 define the key-value pair type, the C++ Map, and insert pairs into the Map. Line 14 calls Promises::hash() and returns a Promise settled to a Map. As far as the au- thors know, no other C++ Promise library has function- ality similar to Promises::hash().

Promise++ also features a function to fetch the com- puted result from a Promise. Users can pass a Promise to Promises::await() which will suspend execution un- til the Promise is settled. If settled to ResolvedState, then the value is returned where as an exception is thrown if the Promise is settled to RejectedState. Promises::await() only suspends execution of the thread owner. An exam- ple is shown on lines 17-20 of Listing 8 where the Map is returned and results are accessed with string literals.

Experimental Results

This section evaluates each library from Section 2 based on cyclocmatic complexity and performance. We mea- sure cyclocmatic complexity using the C and C++ Code Counter (CCCC) open-source metrics tool [18] and we measure performance by taking the average execution time across 40 runs with an additional aux-

iliary run conducted beforehand. The auxiliary run is used so process start-up time does not affect execution speeds.

Three sample applications were used. The first applica- tion is the simple code from Listing 1 converted to each framework. The second application, called "Future Re- duce", randomly returns pass or fail from a routine X number of times and reduces the results to a single integer. The third application, called "Pipeline", trims leading and trailing white space from strings in a pipe. The "Fu- ture Reduce" and "Pipeline" applications were modified from High Performance ParallelX [15] quick start exam- ples. To server as the benchmark, we use Promises from the Standard Template Library (STL) [14] because it is automatically shipped with the modern C++ runtime.

Experimental Setup

Code was executed on a 64-bit 8GB RAM machine us- ing the Ubuntu 18.04 operating system and an Intel i5- 5250U 2-core processor with 1.6 GHz. The source code was compiled using GNU C++ Compiler (g++) version 7.4.0. The workspace was setup using The Makefile, Project, and Workspace Creator [19], which runs a Perl script to cre- ate the necessary Makefiles which auto configured for your local machine's environment.

Experimental Results

From Figure 3, the STL has the lowest cyclomatic com- plexity in the "Simple" application because the main() function is the only linear independent path. All other libraries increase the number of paths with each lambda expression in a then() chain. Promise++ reduces com- plexity in the "Pipeline" and "Future Reduce" examples by 25% and 20% respectively. Factors contributing to complexity are:

1. *~The number of functions and lambda expressions.* Each function or lambda expression contributes to the number of linear independent paths. The STL version of the "Simple" program has a complexity of one be- cause the main function is the only procedure present.All other libraries use several lambda expressions to represent asynchronous tasks and thus add to an ap- plication's complexity metric.

2. *The number of Promises needed to start new chains.* Promise creation is separate from asynchronous task execution in libraries such as Promise-cpp and PoolQueue. The lambda expressions used for ini- tialization are run after a Promise is settled and, therefore, are written as new linear paths in then() methods. Promise++ combines Promise creation and asynchronous tasks in the same methods. In HPX, a new Future is required to form new chains which also contributes to complexity.

3. *System cleanup after asynchronous tasks complete.* Some libraries, such as STL and STLab, require cleanup procedures after a task has completed exe- cution. Every function is another linear independent path and adds to a programs complexity.

4. *The number of control statements needed within callback functions.* Sometimes more control state- ments are necessary for fault-tolerant code. With Promise++, if/then statements may be necessary to ensure the call to Settlement.resolve() or Settlement.reject() is invoked. Each new control statements is another linear independent path.

Figure 4 shows our experimental performance results with the fastest run-time from STLab which was 93% faster than the benchmark. Promise-cpp was 92% faster, PoolQueue was 88% faster, HPX was 67% ~ faster, and Promise++ was 10% faster. The primary factor contributing to performance slowdowns is memory and thread management. In the STL, PoolQueue and Promise++, the system allocators and deallocators are called each time a new Promise or value is created which impacts performance as the input size increases. The STL, Promise-cpp, and Promise++ all use system threads with- out any pool mechanism, so performance is affected when the maximum limit is reached and tasks are suspended. In HPX, the call to texttttFuture.get() outside of lambda ex- pressions suspends execution when the value is not ready and thus contributes to system slowdowns. STLab has several variations of thread pools, including an abstrac- tion around system threads. Tasks are executed based on a priority which the user can specify. For these reasons, STLab had the fastest runtime.

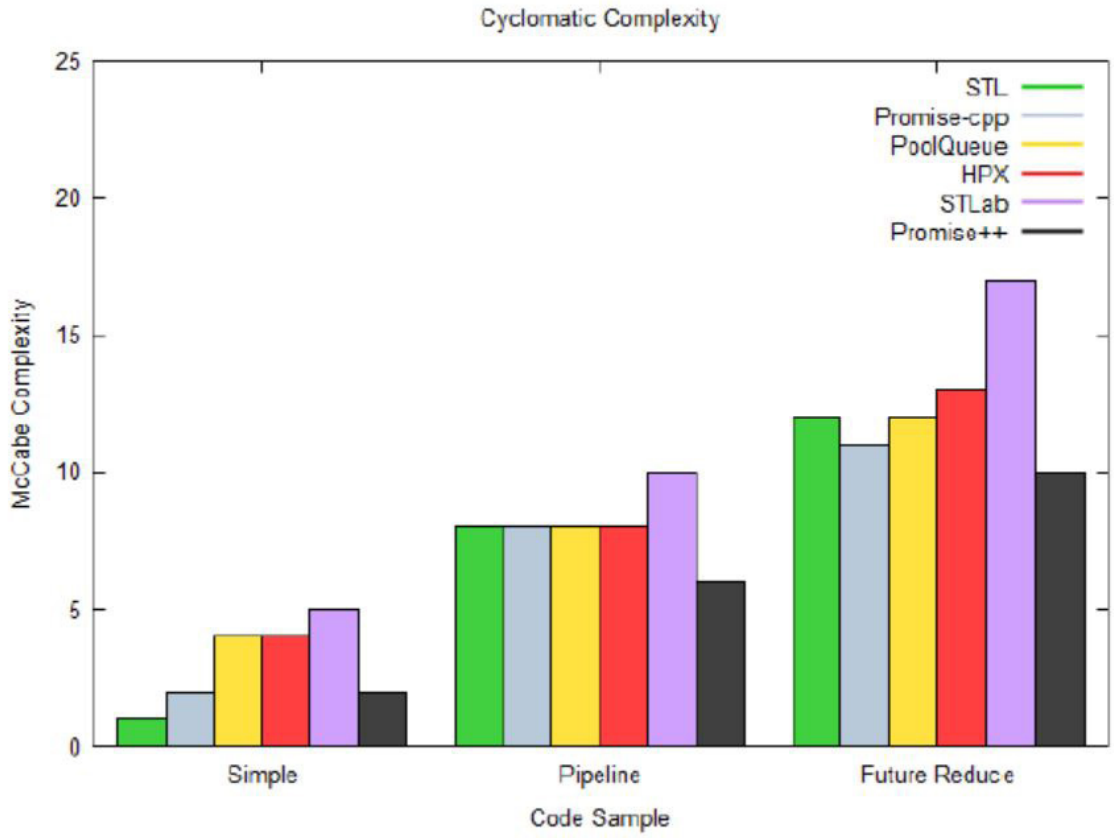

Figure 3: Illustrates the change in cyclomatic complexity acrossPromise designs

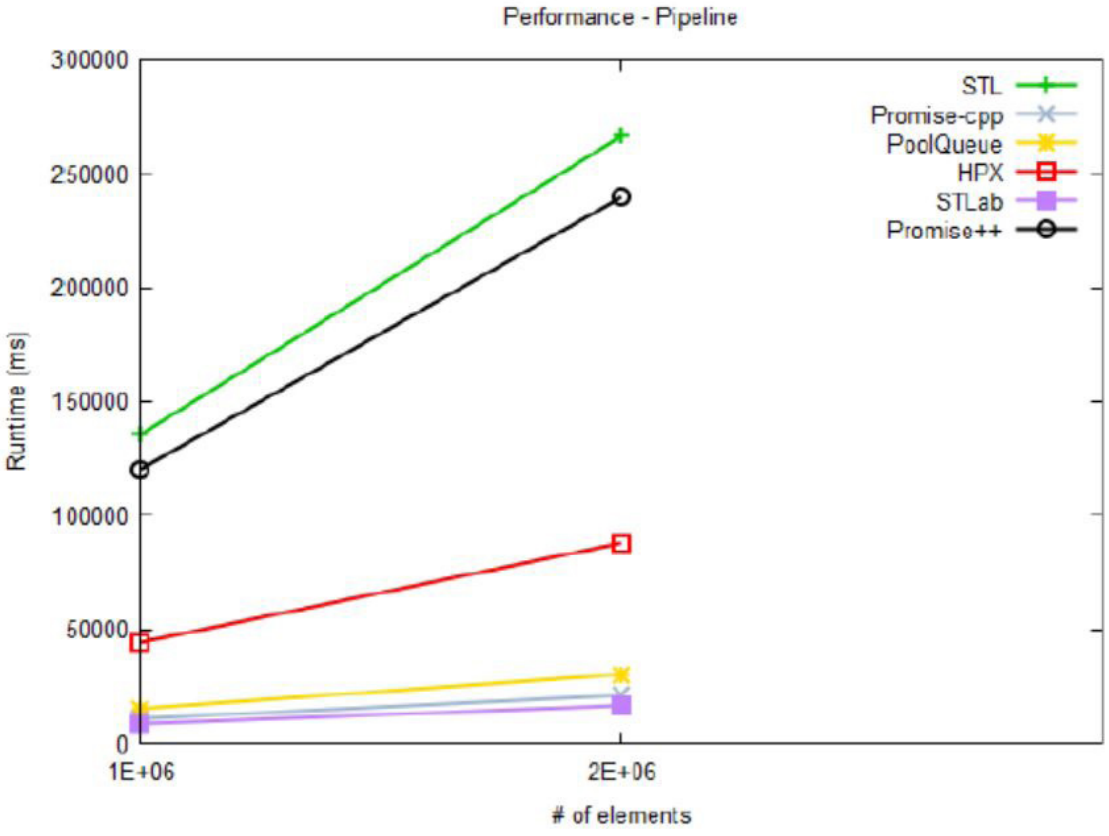

Figure 4: Shows run-times for each library

Threats to Validity

Our results in this study are based on two assumptions, representing threats to validity. These limitations are as follows:

5. *Promise creation*. Our performance experiments are based on Promise creation and destruction. New Promises or Futures are created with each call to then(). This can affect an applications performance since memory allocation and deallocation are expen- sive operations. Moreover, all sample code uses the system allocator behind the scenes which also affects execution speed because there is multiple threads con- tending for the allocator.

6. *CCCC and cyclomatic complexity*. When measuring cyclomatic complexity, the C and C++ Code Counter (CCCC) only counts functions or lambda expressions with return statements. Therefore, those callbacks with return type void do not contribute to the number of linear independent paths.

Lessons Learned

After using many C++ Promise & Future libraries in- cluding our own, we learned some lessons, which may be helpful to individuals creating their own implementations.

Reducing the cyclomatic complexity may cost you speed

Existing C++ Promise and CSP libraries introduce sev- eral accidental complexities that make asynchronous appli- cations difficult to understand and hard to program. We, therefore, created a library that reduces cyclomatic com- plexity. However, as shown in Section 4, our experience suggests that applications may run slower.

Correctly handling Promise state is vital

Managing memory and threads is important with CSP libraries because they are the first and last entities with access into Promises before the state changes. When a task completes and returns a value, that value is contained within the Promise forever. This essentially means that some entity, usually an allocator, must be robust enough to create a value that can live until destruction time which is undefined. Moreover, that value must travel between asynchronous tasks and thus it is the thread manager's responsibility to pass the value to the appropriate desti- nation.

Related Work

Tamino Dauth and Martin Sulzmann [20] compare two C++ libraries, Boost.Thread [21] and Facebook's Folly li- brary [22]. Both libraries extend the STL Future with their own versions of the then() method and the authors dis- cuss the design differences between the two. The authors also developed their on C++ framework for CSP called the Advanced Futures & Promises in C++ (ADV). ADV combines ideas from Boost.Thread and Folly, specifically the use of executors such as thread pools and a method to filter Futures that do not meet a predicate condition. We did not include Boost.Thread, Folly, or ADV in our evaluation of C++ CSP libraries because Dauth and Sulz- mann already provide an in-depth comparison of the de- signs. Our experience report, however, differs from the authors in that we also consider cyclomatic complexity as a factor to CSP design.

Promise/A+ [8] lists known Promise implementations in several languages, such as Python and Perl, that are compliant with the A+ specification. The Promises/A+ organization does not give a comparison between imple- mentation. They do define a suite of JavaScript test cases that, when passed, indicate a library successfully imple- ments A+ Promises.

Another C++ implementation for CSP is Libq [23]. Libq differs from other implementations by dispatching tasks to Queue data structures which are assigned to spe- cific threads. This allows the user to decide which thread a function is called. We did not include Libq in our study be- cause its design for Promise creation is similar to Promise- cpp and PoolQueue. With Libq, a separate Promise must be created to re-start a then() chain and this design is already well represented in our study.

Conclusion

This article describes our experience using C++ Promise libraries; including, our own implementation of the Promises/A+ specification named Promise++. We discuss the design trade-offs between frameworks in the context of cyclomatic complexity and performance. The key contribution of this article is represented by the lessons learned from using and writing C++ Promise libraries. Lessons learned about cyclomatic complexity and resource management have potential to affect a libraries usability in the context of performance.

Promise++ is freely available in open-source format for download from the following location: https:// github. com/SEDS/promisepp.

References

1. D Friedman, D Wise (1976) The Impact of Applicative Program- ming on Multiprocessing, Technical report (Indiana University, Bloomington. Computer Science Department), Indiana Univer- sity, Computer Science Department, 1976.

2. DP Friedman, DS Wise (1978) Aspects of pplicative programming for parallel processing, IEEE Transactions on Computers 27: 289–96.

3. P Hibbard (1976) Parallel processing facilities, New Directions in Al- gorithmic Languages: 1–7.

4. HC Baker, C Hewitt (1977) The incremental garbage collec- tion of processes, in: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, ACM, New York, NY, USA, 1977: 55–9.

5. Module deferred (2017).

6. H Kaiser, T Heller, D Bourgeois, D Fey (2015) Higher-level par- allelization for local and distributed asynchronous task-based programming, in: Proceedings of the First International Work- shop on Extreme Scale Programming Models and Middleware, ESPM '15, ACM, New York, NY, USA 2015: 29–37.

7. D Denicola, You're Missing the Point of Promises (2012).

8. K Zyp, K Kowal (2017) Promises/a+.

9. A Watson, D Wallace, T McCabe (1996) Associates, N. I. of Standards, T. (U.S.), Structured Testing: A Testing Method- ology Using the Cyclomatic Complexity Metric, NIST special publication, U.S. Department of Commerce, Technology Ad- ministration, National Institute of Standards and Technology, 1996.

10. B Bibeault, Y Katz, AD Rosa (2017) jQuery in Action, Third Edition, Manning Publications Co, 2015.

11. xhawk, Promise-cpp: C++ promise/a+ library in javascript styles, https://github.com/xhawk18/promise-cpp (2017).

12. R Hashimoto, Poolqueue: C++ asynchronous promises, inspired by promises/a+, https://github.com/rhashimoto/ poolqueue (2017).

13. O Kowalke Future (2020)

14. C++: promise (2018).

15. ea Hartmut Kaiser (2020) Hpx v1.4.1: The c++ stan- dards library for parallelism and concurrency.

16. Software technology lab: Future (2019).

17. E Gamma, R Helm, R Johnson, J Vlissides (1995) Design Pat- terns: Elements of Reusable Object-oriented Software, Addison- Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

18. T Littlefair (2017) C and c++ code counter.

19. J Michel, C Elliott (2017) The makefile, project, and workspace cre- ator.

20. T Dauth, M Sulzmann (2018) Advanced futures and promises in c++ .

21. A Williams, VJB Escriba, Boost. Thread.

22. Folly: Facebook open-source library, https://github.com/ facebook/folly (2020).

23. G R¨antil¨a, Libq: A platform-independent promise library for c++ (2018).