Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016
© 2016 The authors and Open Channel Publishing Ltd. All rights reserved.

Development and Evaluation of a Modern C++CSP Library

Kevin CHALMERS [1]

School of Computing, Edinburgh Napier University

Abstract. Although many CSP inspired libraries exist, none yet have targeted modern C++ (C++11 onwards). The work presented has a main objective of providing a new C++CSP library which adheres to modern C++ design principles and standards. A secondary objective is to develop a library that provides simple message passing concurrency in C++ using only the standard library. The library is evaluated in comparison to JCSP using microbenchmarks. CommsTime and StressedAlt are used to determine the properties of coordination time, selection time, and maximum process count. Further macrobenchmarks, Monte Carlo π and Mandelbrot, are gathered to measure potential speedup with C++CSP. From the microbenchmarks, it is shown that C++CSP performs better than JCSP in communication and selection operations, and due to using the same threading model as JCSP can create an equal number of processes. From the macrobenchmarks, it is shown that C++CSP can provide an almost six times speedup for computation based workloads, and a four times speedup for memory based workloads. The implementation of move semantics in channels have provided suitable enhancements to overcome data copy costs in channels. Therefore, C++CSP is considered a useful addition to the range of CSP libraries available. Future work will investigate other benchmarks within C++CSP as well as development of networking and skeleton based frameworks.

Keywords. CSP library, microbenchmarking, macrobenchmarking, C++

Introduction

This paper describes work undertaken in the development and evaluation of a new C++ library supporting CSP semantics that uses modern C++ standards and design principles. The aim of the developed library is to provide an API familiar to both the JCSP programmer and the C++ programmer, with the latest C++ standard exploited to provide some additional syntactic sugar. The aim of the work is not to build an optimised runtime at present, but rather build upon the concurrency model exposed in modern C++, that is, the C++11 standard onwards.

The rest of the paper is divided as follows. In Section 1 background on related CSP libraries is presented alongside an overview of modern C++ standards and design principles, and how these are used in the new C++CSP library is discussed in Section 2. In Section 3 an experimental framework is described, the defined benchmarks being used to evaluate the library. Results of the experimental work are provided in Section 4. Finally, Section 5 draws conclusions and highlights future work directions with the library.

[1]Corresponding Author: *Kevin Chalmers, School of Computing, Edinburgh Napier University, Edinburgh, EH10 5DT*. Tel.: +44 131 455 2484; E-mail: `k.chalmers@napier.ac.uk`.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

1. Background

A number of libraries supporting Hoare's CSP semantics [1] exist. Languages such as Java (JCSP [2], CTJ [3]), Python (PyCSP[4]), Haskell (CHP [5]), JavaScript [6] and C++ (C++CSP [7]) have all had a library developed and evaluated, either using the language's inbuilt threading support or through external library support. Of these libraries, JCSP is the most well known.

Developed libraries are almost always implemented within the language's provided multi-threading mechanisms. For example, JCSP utilises operating system threads as exposed through Java's runtime. Other approaches exist for implementing CSP semantics. A recent C# approach has investigated the use of asynchronous operations [8]. Python approaches to CSP semantics have explored stackless and non-concurrent methods. Java approaches have also aimed at millions of processes via bytecode rewriting [9]. Also, previous C++ work evaluated both thread and fiber support as provided by external libraries and the native operating system.

Most CSP libraries provide primitives in an API that would be familiar to the standard programmer in the target language. However, Communicating Scala Objects [10] aims to provide familiarity to the CSP user.

1.1. Why Another C++CSP Library?

C++CSP2 launched in 2007, the last update occurring in 2013. C++CSP2 requires the Boost set of C++ libraries, requires the library be pre-built, and the library is not easily extensible.

An advantage of aiming at a low level language such as C or C++ is the ability to integrate with other languages. Indeed, the CCSP library has been integrated into Java [11]. Although CCSP would be an ideal target library for many concurrent runtimes, the library requires a Unix based operating system for building, and therefore some areas of the market are excluded.

In addition, modern C++ has threading support provided within the standard library. Therefore, a further aim of this work is to build and evaluate a C++CSP library using modern C++ standards.

1.2. Modern C++ and C++ Design

C++CSP utilises the various design principles encouraged in modern C++. C++CSP is a header only library and thus requires no pre-built code, and is therefore easy to incorporate within existing code. A compliant C++11 compiler is all that is required to use C++CSP.

One of the design decisions made for C++CSP was to allow expression of parallelism as simply as possible. Therefore, a number of approaches in C++ have been utilised as detailed in Section 2.

The rest of this section discusses the features and design principles utilised in C++CSP. The C++11 standard onwards provides a number of new features to the language and standard library, however only those relevant to the development of C++CSP are discussed in this section. The topics covered are:

- new C++ language features.
- threading support in C++.
- modern C++ design features.

1.2.1. New Language Features

C++ as added a number of new language features to simplify and optimise application development. Here, four new features are described: move semantics; initializer lists; variadic

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

templates, and lambda expressions. Smart pointers are also described, although they are part of the standard library supported by the language features discussed.

Move Semantics The C++11 standard introduced a new parameter passing concept in the form of move semantics. Move semantics (*rvalue* references) enable parameters to be *moved* into new scopes rather than being copied or referenced, and are enabled via new constructors and assignment overrides, as well as declaring parameters as *rvalues* using the && type declaration. Move semantics provide advantages over both copy and reference passing:

1. there is no reference held in the caller's scope, reducing side-effects.
2. there is no copy created, reducing memory overhead.

A key use of move semantics is in collections. All C++ standard collections implement move constructors and assignment operators, and therefore as data is moved into the new scope rather than explicit copies being made, temporary memory allocation and copying is removed.

Initializer Lists Initilializer lists provide a method to construct an object via a list of values. A list of values is defined which can be iterated across to initialise an object, or passed into a function for a similar purpose. For example, C++ collections are automatically constructed from an initializer list of the correct type.

As an example, consider Listing 1.

```
1 vector<int> v{1, 2, 3, 4, 5};
```

Listing 1: Object Creation via an Initializer List.

Here, the `vector v` is provided elements in a bracketed list. Elements of the list become elements of the `vector` in the order defined, and as the list is also a *rvalue*, the elements are moved into the `vector` constructor.

Initializer lists are useful for the declaration of small lists of data in code, or for a list of elements that share a common base type. They are not useful as a data store themselves, but are rather syntactic sugar for object creation with some optimisation.

Variadic Templates C++11 also introduced variadic template concepts, which enhance the power of C++ templating with a richer set of possibilities. Variadic templates are powered by the parameter pack, a variable length type definition denoted by `...`. A parameter pack is expanded during compilation to provide access to the types held within.

As an example, consider Listing 2.

```
1 template<typename T, typename... args>
2 void foo(T value, args... rest)
3 {
4     cout << value;
5     if (sizeof...(args) > 0)
6         foo(rest);
7 }
```

Listing 2: Variadic Template Example.

Listing 2 declares a templated function `foo` with a standard type `T` and a variadic type `args`. `foo` also takes parameters of type `T` and `args`. `foo` prints the parameter of type `T`, and then recursively calls itself if the size of the remaining parameter pack is one or greater. The recursive call ensures that the head of the parameter pack is allocated to `T` in the recursive call, and the remainder of the parameter pack is allocated to `args`.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

In comparison to standard C++ templating and the generic capabilities of Java and C#, the power of variadic templating is the enabling of new compile time types. Variadic templates permit `function` objects of various type lengths, and allow `thread` creation with multiple parameters to be type safe. Listing 3 illustrates this concept.

```
1  void work(int x, float y, string str)
2  {
3      // ... do some work
4  }
5
6  int main(int argc, char **argv)
7  {
8      // Create thread from work function
9      thread t(work, 5, 2.0f, string("test"));
10
11     // ...
12 }
```

Listing 3: Thread Creation with Multiple Parameters.

The C++ standard library uses variadic templates throughout. The standard advises that C-style variadic functions be no longer used, with variadic templates replacing the capability in a type safe manner. Variadic templates also enable and support other new features of C++ including tuples and lambda expressions.

Lambda Expressions Lambda expressions are another addition to C++. The general principles follow those of other languages which have added lambda expressions; functions can now be treated as first class objects and anonymous functions can be defined. An example lambda expression is provided in Listing 4.

```
1 int a = 5;
2 int b = 10;
3 auto fun = [=](int x, int y) { return x / a + y / b; };
```

Listing 4: Lambda Expression in C++.

`auto` allows the compiler to determine the type at compile time, negating the need to explicitly define the type.

Lambda expressions in C++ have three parts: the capture method; the parameter list; and the function body. The capture method is defined using square brackets (`[ ]`). The method allows the programmer to explicitly define how used variables in the lambda expression are captured in the constructed object. `=` means values are copied, whereas `&` means a value is referenced, leading to potential mutation of the created object. Capture methods can be mixed, and others exist to enable capture of the `this` pointer.

The parameter list is a basic list of parameter types and associated names. The lambda body is a standard set of instructions. The return type of the lambda can be defined or the compiler can determine it based on the `return` call made.

`function` objects provide the expressiveness required to utilise lambda expressions. `function` is a templated type; the type template being the same as the type definition for the lambda expression. For example, we can have:

- `void(int, int)` a function that takes two `int` parameters and returns nothing.
- `int()` a function that returns an `int` and takes no parameters.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

`function` objects can be created from existing functions. The `function` object can then have values bound to it, allowing further currying of the function. Once curryed, the function can be called with the new set of required parameters. Listing 5 illustrates this concept.

```
1  void add(int x, int y)
2  {
3      return x + y;
4  }
5
6  int main(int argc, char **argv)
7  {
8      auto add_three = bind(add, 3, _1);
9      int x = add_three(5); // x is now equal to 8
10 }
```

Listing 5: Currying a C++ Function.

The `_1` denotes a placeholder, stating the first parameter passed to add_three will be used for this parameter. Other numbered placeholders exist.

Finally, as C++ has operator overloading capabilities, functor objects can be used to create function objects. Consider Listing 6.

```
1  struct functor
2  {
3      void operator(int x, int y) { /* ... */ }
4  };
5
6  int main(int argc, char **argv)
7  {
8      functor f;
9      function<void(int, int)> fun(f);
10 }
```

Listing 6: Using Functor Objects.

In Listing 6, `struct functor` declares a call operator overload, and therefore we can convert the object into a `function` object.

With lambda expressions, C++ provides functional programming mechanisms. Templates provide the strong typing at compile time, but flexibility at development time. Variadic templates in particular can be useful here.

Smart Pointers Smart pointers overcome the need for keeping track of allocated resources without the use of garbage collection. Resources are tracked through the use of copy construction, move semantics, and object destruction to determine when a resource should be freed. The C++11 standard formalised three smart pointer types:

- `unique_ptr` is a resource owned by one, and only one, scope.
- `shared_ptr` is a resource owned by multiple scopes and controlled via reference counting.
- `weak_ptr` is a non-owning (i.e., non-counted) reference to a `shared_ptr` controlled resource.

`unique_ptr` has zero overhead beyond object construction and destruction and enables automatic cleanup of allocated resource. `shared_ptr` has an overhead due to reference counting, but does enable simple management of shared resources that need to be cleaned up when no longer needed. `weak_ptr` is used in circumstances where temporary referencing is required, and has some overhead associated with copying the `shared_ptr` object.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

1.2.2. Threading Support in Modern C++

Threading support in the standard library is also a feature of modern C++. Support is provided in four features:

- Threads and the associated locking mechanisms.
- Futures.
- Atomics.
- A defined C++ memory model.

Thread Creation Basic multi-threading support is provided via instantiation of the `thread` class. Listing 3 provided an example of how to create a `thread` object.

The `thread` class is similar in most regards to the Java and C# threading classes. However, C++ threads have subtle differences.

Firstly, C++ threads are not necessarily operating system threads, although it is likely that they are, and major compilers have adopted this model. Therefore, a C++ thread could be represented via a lightweight threading library under the hood.

Secondly, as a C++ thread may not be an operating system thread, certain features common in threading libraries are not present. For example, a C++ thread cannot be interrupted externally. The CPA community will find this concept intuitive as events replace interrupts in most regards.

Finally, a C++ thread is owned by the creating context/scope. C++ threads cannot be copied, and can only be shared using pointers or move semantics.

Mutex Basic C++ thread coordination is provided by the `mutex` object. Unlike Java and C#, where every object can be used for coordination, C++ requires explicit creation of individual `mutex` objects to control access to shared resources.

A `mutex` provides only three methods: `lock`; `unlock`; and `try_lock`. `lock` and `unlock` are obvious, and `try_lock` returns `true` or `false` dependent on the success of attempting to lock the `mutex`. In this case, no waiting will occur.

As with threads, mutexes are typically provided by the OS, although may be replaced with other backend dependent on the compiler used.

Lock guards Automated `mutex` locking is supported in C++ by the `lock_guard` type. The `lock_guard` will automatically lock a `mutex` on the guard's creation, and will automatically unlock the `mutex` on the guard's destruction. `lock_guard` utilises RAII concepts (see Section 1.2.3) and an example is provided in Listing 7.

C++ provides two guard techniques. `lock_guard` operates as described, and `unique_lock` ensures the lock is accessible in only one scope. `unique_lock` is used within the `condition_variable` discussed next.

Condition variables Mutexes and lock guards provide only mechanisms to lock resources, and provide no mechanisms for signaling between threads. The `condition_variable` is the C++ class providing such functionality, via suitable methods for both waiting and notifying.

A `condition_variable` is a separate object to `mutex` and `lock_guard`, and a call to wait on a condition variable must provide a `unique_lock` object as a parameter.

1.2.3. C++ Design Principles

Two main design principles have been used to ensure a well designed library for C++CSP - *PIMPL* and *RAII*.

PIMPL PIMPL is short for private implementation or pointer to implementation. PIMPL is a design principle in C++ where the inner workings of a class are hidden from global scope by using a privately declared, internal class, with a pointer to an instance of the internal class

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

contained within the external class. PIMPL allows an object to be copied easily - the overhead being a pointer copy, and therefore provides a mechanism similar to Java referencing.

PIMPL is a useful paradigm for object-orientated C++, as C++ polymorphism requires pointers or references to function. As a C++ reference must be associated with an instantiated object, and cannot change which object it is associated with, pointers become the only reasonable design choice for C++ polymorphism. PIMPL permits base class instantiation when working with class hierarchy with an abstract base class. As the external class provides an interface to the internal class, there are no pure virtual methods in global scope, and hence no abstract classes externally. A specialization class implements its own internal specialized class, providing an actual implementable definition.

RAII Resource Acquisition is Initialisation (RAII) is a design principle used to control resource lifetime. RAII states that an object should create/acquire its resources on instantiation, and destroy/free its resources on destruction. The principle requires that a resource exists and is held by an object between the end of initialisation (construction) and the start of finalisation (destruction), and thus correct use of RAII ensures resources do not leak as long as the object does not leak (i.e., is cleaned up correctly).

RAII is typically associated with object lifetim, however the general principle of RAII is also used in C++ for scope based resource management. In particular, C++ threading relies on RAII principles, an example of which is provided in Listing 7.

```
1 mutex mut;
2 resource res;
3
4 void work()
5 {
6     lock_guard<mutex> lock(mut);
7     // ... work with locked resource.
8     // End of scope.  lock_guard automatically released.
9 }
```

Listing 7: `lock_guard` with RAII.

A `lock_guard` object is created at the start of the function scope. For initialisation of the `lock_guard` to complete, it must acquire (lock) control of the `mutex`, and at this point the scope has exclusive access to the resource managed by the `mutex`. At the end of the function scope, the `lock_guard` object is automatically destroyed as it is a stack based object, and on object destruction, the `lock_guard` automatically calls unlock on the `mutex`, freeing the resource.

1.3. Objectives

The aim of this work is to develop and evaluate a modern C++CSP library. The library should be easy to use by any C++ programmer, utilise C++ threading support, and adhere to modern C++ design principles.

Library evaluation is undertaken using micro and macro level benchmarks. Microbenchmarks allow base properties of the C++CSP library such as communication time, selection time, and memory usage, to be examined. JCSP will be used as a comparator to C++CSP as JCSP is the maturest library taking CSP as an inspiration, and therefore provides a good baseline. Ideally the existing C++CSP library would also be used as a comparator. However, it was determined that the existing version of C++CSP no longer builds with a modern C++ compiler and Boost installation.

The objective in undertaking microbenchmarking is the comparison of the basic properties in C++CSP to a standard. The aim is therefore to determine if a modern C++CSP is a

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

useful addition. C++CSP itself is a stepping stone towards further work outlined in the Future Work section of this paper.

Macrobenchmarking will analyse the potential speedup available using C++CSP. It is expected using C++CSP will incur a minimal overhead, and therefore a good overall speedup should be evidenced. The objective of the macrobenchmarks therefore lie in illustrating good speedup potential.

2. Design of C++CSP

In this section, the design aspects of C++CSP are discussed within the context of the standards and principles described in Section 1.2. The goal of the library is to make C++CSP as simple to use as possible, and the concepts used are as follows:

move semantics and *rvalues* used throughout the library; channel communication exploits data movement when possible.
smart pointers used throughout the library; C++CSP has a pointer free API.
initializer lists used throughout; especially useful in `par` construction.
variadic templates used in process creation.
lambda expressions used in process creation and helper patterns.
PIMPL used throughout to provide pointer free API.
RAII used throughout; resource management hidden from API user as much as possible.

Operator overloading has been used to provide callable objects and automatic conversion of channel types to channel end types.

The remainder of the section is divided into six subsections as follows:

1. an example of the high level design of the library.
2. library features.
3. primitive implementation (channels and barriers).
4. process and parallel usage.
5. selection and timers.
6. exemplar applications.

2.1. High Level Design

Figure 1 presents a high level view of class associations in part of the library. The `one2one_chan` type is shown as it provides enough for discussion on the use of PIMPL and RAII.

For each row in Figure 1, only the leftmost type is user accessible and usable. The other types are internal support classes for C++CSP hidden from the C++CSP library user. There are six types of note: `one2one_chan`; `chan`; `alting_chan_in`; `chan_in`; `chan_out`; and `guard`. Of these types, only `one2one_chan` is user creatable, the others being accessible from the this core instantiated channel.

`one2one_chan` is a core channel type in C++CSP, with one2any, any2one, and any2any channel types also supported by C++CSP. `one2one_chan` itself is only a shell definition, containing a created `chan` object (RAII), and methods to construct input and output channel ends from the instantiated `chan` object.

The channel end types - `alting_chan_in`, `chan_in`, and `chan_out` in Figure 1 - provide read and write functionality exposure from the `chan` object. `alting_chan_in` extends both the `chan_in` and `guard` classes to expose correct external behaviour. Each of the channel end types also use the PIMPL idiom, and therefore have private internal implementation classes, the outer classes being shells to interact with these internal class implemen-

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

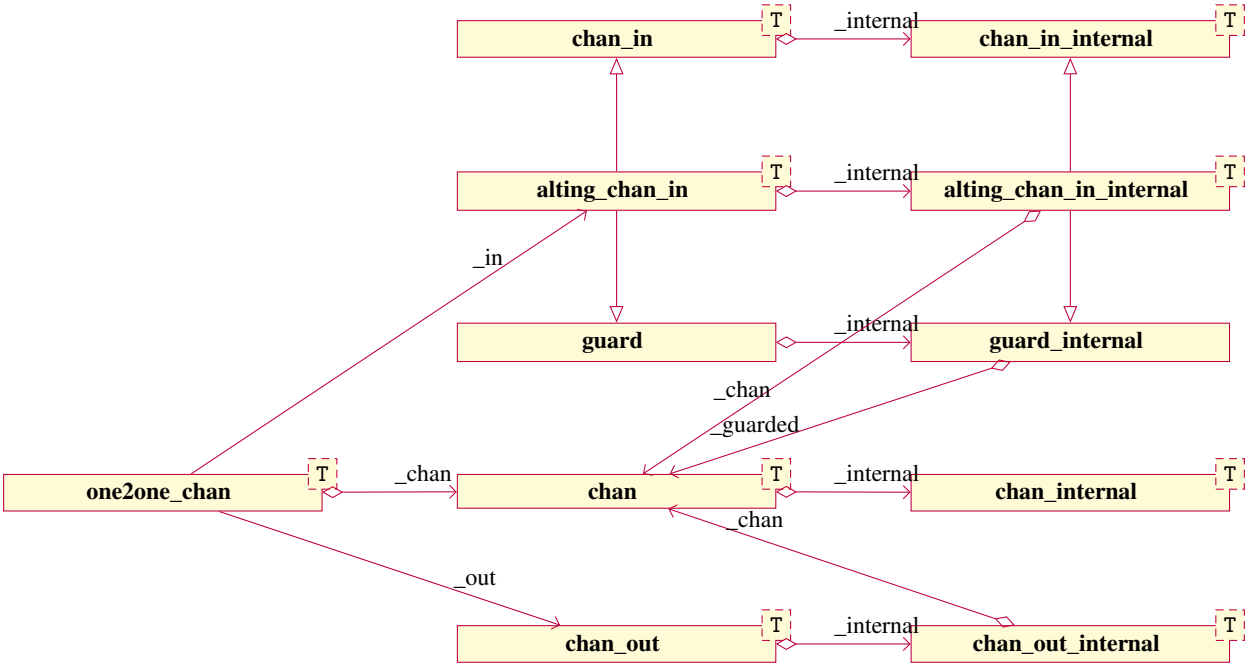

Figure 1. High Level Diagram Example of C++CSP Architecture.

tations. Because of the PIMPL approach, `alting_chan_in_internal` also specialises the `chan_in_internal` and `guard_internal` classes.

The internal class instantiations each have a copy of the `chan` object, itself a shell to the private `chan_internal` class. However, `chan_internal` itself is abstract, with `basic_chan_internal` and `buffered_chan_internal` types provided in C++CSP for core channel functionality.

C++CSP therefore operates on a cenralised channel model. Other classes are effectively interfaces to the inner `chan` instantiation, providing access to the `chan_internal` class for other parts of the C++CSP library. Such an approach enables simple updating of the channel functionality as only `basic_chan_internal` and `buffered_chan_internal` contain actual channel operation code. Thus, simple extensibility is possible in future, such as planned network channel functionality.

2.1.1. Pointer Free API

One criticism often levelled at C and C++ is the requirement to work with memory resources via pointers. Although C++11 provides smart pointers to automate resource management, an understanding of when to use and define pointer types is still often required.

The PIMPL idiom was developed to overcome the requirement of working with pointers directly, and best practice in C++ now dictates that pointers be avoided in all but exceptional circumstances. Indeed, the C++ standard library does not expose pointers itself, utilising the PIMPL idiom throughout.

The RAII idiom ensures resource lifetime is managed by object lifetime. Therefore, combining RAII and PIMPL, and augmented by smart pointers, provides a C++CSP API that exposes no pointers, and requires no usage of pointers.

2.2. Library Features

Two features of note are provided with the C++CSP library: the ability to treat certain objects as functions (via operator overloads); and the provision of helper patterns to support simple, common behaviours found in CSP libraries.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.2.1. Operator Overload for Common Behaviour

An often overlooked feature of C++ is the ability to overload operator behaviour on an object. One such operator is the *call* or *invoke* operator `()`. This operator allows the programmer to treat the object as a callable function.

C++CSP uses operator overloading of the invoke operator to call common behaviour of the object in question. As an example, consider Listing 8.

```
1 void successor(chan_in<int> in, chan_out<int> out)
2 {
3   while (true)
4   {
5     auto val = in();
6     out(++val);
7   }
8 }
```

Listing 8: Callable Channel Objects.

Here, a value is read from the the `in` channel by invoking it using the call operator. A value is also written to the `out` channel in a similar manner, but with a parameter passed into the call.

C++CSP provides invoke operator overloads on most basic types:

- channels
- barriers
- parallels
- processes
- alts

The overloads provide syntactic sugar when working with these types, and the override simply calls an existing method on the object. For example, calling an `alt` object will invoke the fair select method.

2.2.2. Helper Patterns

One common requirement when developing in a CPA style is the need to perform certain common operations in parallel. For example, reading or writing with a collection of channel ends in parallel is a common pattern of behaviour. Replicated parallel operations (parallel loops) are also features of both CSP and occam, and therefore it is sensible to implement such behavioural patterns directly within a CSP inspired library.

`par_for` allows a function to be executed in parallel based on a collection of parameter. Listing 9 provides an example of how `par_for` can be used.

```
1 void delta(chan_in<int> in, vector<chan_out<int>> out)
2 {
3   while (true)
4   {
5     auto value = in();
6     par_for(out.begin(), out.end(), [&](chan_out<int> chan){chan(
          value);});
7   }
8 }
```

Listing 9: `par_for` Helper Pattern.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

`par_read` and `par_write` are similar patterns that perform parallel reads and parallel writes on a collection of channels. Their definitions are provided in Listing 10 with other definitions for working with `vector` types and ranges part of the C++CSP library.

```
1 template<typename T>
2 std::vector<T> par_read(std::initializer_list<chan_in<T>> &&chans);
3
4 template<typename T>
5 void par_write(std::initializer_list<chan_out<T>> &&chans, const
     std::vector<T> &values);
```

Listing 10: `par_read` and `par_write` Definitions.

The helper patterns provide further syntactic sugar to the programmer, but at the cost of threads being created each time the helper patter function is called. That being said, the plug and play library provided with C++CSP uses the helper patterns whenever possible.

2.3. Primitives Implementation

C++CSP provides two core event primitives: channels; and barriers. A discussion of the model used to implement channels was provided in Section 2.1, and `barrier` follows the same principles. A selectable `alting_barrier` that extends the `barrier` type has also been provided in the C++CSP library.

2.3.1. Channels

Figure 1 provided an extract of the high-level architectural view of a C++CSP channels. In this section, the behavioural implementation of C++CSP channels is provided, presenting both the write and read operations.

Listing 11 presents a simplified version of the C++CSP channel write operation. Stepping through the pertinent lines:

```
1  void write(T value)
2  {
3    std::unique_lock<std::mutex> lock(_mut);
4    _hold.push_back(std::move(value));
5    if (_empty)
6    {
7      _empty = false;
8      if (_alting)
9        guard::guard_internal::schedule(_alt);
10   }
11   else
12   {
13     _empty = true;
14     _cond.notify_one();
15   }
16   _cond.wait(lock);
17 }
```

Listing 11: Channel Write Operation.

3 lock the channel for exclusive use.

4 store the written value. Note that C++CSP uses a `vector` to store values to enable simple movement communication.

5-10 *if the channel is empty*, set empty to false, and notify any waiting `alt`.
11-15 *otherwise* the reader is waiting, so set empty to true and notify the reader.
16 wait for the read to finish.

Listing 12 presents a simplified version of the C++CSP channel read operation. Stepping through the pertinent lines:

```
1  T read()
2  {
3    std::unique_lock<std::mutex> lock(_mut);
4    if (_empty)
5    {
6      _empty = false;
7      _cond.wait(lock);
8    }
9    else
10     _empty = true;
11   auto to_return = std::move(_hold[0]);
12   _hold.pop_back();
13   _cond.notify_one();
14   return std::move(to_return);
15 }
```

Listing 12: Channel Read Operation.

3 lock the channel for exclusive use.
4-8 *if the channel is empty*, set empty to false and wait for the writer.
9-10 *otherwise* set empty to true.
11 retrieve the value from the store.
12 empty the store.
13 notify the writer.
14 return retrieved value.

Combined together, we have behaviour defined for either the read committing first, or the write committing first. The different behaviours are illustrated in Figure 2.

Tick	Reader	Writer	Reader	Writer
1	*lock*			*lock*
2	*set empty false*			*store*
3	*wait*			*set empty false*
4		*lock*		*schedule*
5		*store*		*wait*
6		*set empty true*	*lock*	
7		*notify*	*set empty true*	
8		*wait*	*retrieve*	
9	*retrieve*		*notify*	
10	*notify*		*return*	
11	*return*			*return*
12		*return*		

Figure 2. Overview of Channel Behaviour. Read commit first is on the left, and Write commit first is on the right.

Channel ends provide the correct interface to the internal channel, with mutexes, etc., created utilised as required.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.3.2. Move Semantic Channels

Channels in C++CSP exploit move semantics, and when a channel is written to, the C++CSP user can choose to pass the parameter directly as a copy, or they can move it into the channel. Listing 13 illustrates how pure movement can be achieved.

```
chan_out<mandelbrot_packet> out;
// Value is copied into channel, then moved out.
out(packet);
// Value is moved into channel, then moved out.
out(move(packet));
```

Listing 13: Using Channels.

Within the channel, the written value is moved into the local store, and then moved out during the subsequent read operation. The channel retains no copy, nor creates any additional copies unless the programmer chooses to when calling the write operation. The aim in providing move functionality in channels is to reduce copying, while ensuring values are not referenced between process scopes.

2.3.3. Auto Conversion of Types

Channels can be converted to their respective input and output ends automatically. This functionality is enabled via operator overloads that perform implicit conversion of channel types to their respective channel end types. As a further example, consider Listing 14.

```
template<typename T>
class prefix : public process
{
  // ... rest of definition
public:
  prefix(T value, chan_in<T> in, chan_out<T> out)
  {
    // ... construct object
  }
  // ... rest of definition
};

int main(int argc, char **argv)
{
  one2one_chan<int> a;
  one2one_chan<int> b;

  par
  {
    prefix<int>(0, a, b),
    // ... other processes
  }();
}
```

Listing 14: Conversion from `one2one_chan` to `chan_in` and `chan_out`.

`prefix` has a defined constructor that takes a value, an input channel end, and an output channel end as parameters. However, in the instantiation of `prefix` within `main`, the parameters passed are `0` and two `one2one_chan` typed objects. The implicit conversion operator overload ensures the respective input and output ends of the channel are captured accordingly.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.3.4. Barriers

The C++CSP barrier implementation follows that of the channel implementation, with PIMPL, RAII, and operation overloads used to simplify the API provided. The `barrier` class is implemented in a similar manner to JCSP, with alting barriers also provided, and likewise implemented in a similar manner to JCSP. Listing 15 provides an example of C++CSP barrier usage.

```
1  barrier bar(3);  // Three processes to register.
2  par
3  {
4    [=]()
5    {
6      while (true)
7      {
8        // ... do some work
9        bar();
10     }
11   },
12 }();
```

Listing 15: Barrier Usage Example

2.4. Processes and Parallels

The `par` type of C++CSP internally operates with the same thread pool approach as JCSP, and therefore the inner workings of `par` will not be presented. Rather, this section will look at how processes are represented, and how `par` can be used simply.

2.4.1. Lambda Expression Processes

Although C++CSP does provide a `process` type, the `par` construct operates on objects of type `function<void()>`. This decision was made to provide flexibility within the API, as `process` objects can be automatically converted to `function<void()>` due to their invoke operator being overloaded.

`par` working with `function<void()>` has some advantages. In particular, any void function can be converted to `function<void()>` by binding all the required parameters as was discussed in Section 1.2.1. However, as such a call is common in C++CSP code, a helper function has been provided called `make_proc`.

`make_proc` is variadically templated, and therefore any function returning void can be used. `make_proc` only requires the function that is to be called and the parameters to bind to the function. Listing 16 illustrates how `make_proc` is used.

`make_proc` is simply a helper function, and underneath calls bind on the function with the parameters provided to create an object of type `function<void()>` which the `par` can use.

An offshoot of having `par` use function objects is that lambda expressions can be used directly in a `par` definition, as shown in Listing 17.

The syntax here is not ideal, and this feature is just an outcome of using `function` objects within the `par`. A possibility is to define `seq` as `[=]()` to make the code somewhat cleaner if required, as will be shown in the exemplar applications.

2.4.2. Parallel with Lists

C++CSP provides initializer list constructors for a number of types, one of which is the `par`. With an initializer list, it becomes possible to define a parallel construct as a list of

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

```
1  void prefix(int value, chan_in<int> in, chan_out<int> out)
2  {
3    out(value);
4    while (true)
5      out(in());
6  }
7
8  int main(int argc, char **argv)
9  {
10   one2one_chan<int> a;
11   one2one_chan<int> b;
12
13   par
14   {
15     make_proc(prefix, 0, a, b),
16     // ... other processes
17   }();
18 }
```

Listing 16: Using `make_proc` to Create a Process.

```
1  int main(int argc, char **argv)
2  {
3    one2one_chan<int> a;
4    one2one_chan<int> b;
5
6    par
7    {
8      [=]() // Prefix
9      {
10       a(0);
11       while (true)
12         a(b());
13     },
14     // ... other processes
15   }();
16 }
```

Listing 17: Using Lamda Expressions to Create a Process.

processes to execute. As an example, Listing 18 defines a *CommsTime* process network using an initializer list construct.

The advantage of this definition is purely syntactic sugar, and it means a `par` can be defined as a braced list of processes. It allows a syntax similar to occam in many regards. Also, note the use of another initializer list to define the output channels for the `delta` process.

2.5. Selection and Timers

The final key feature of any CSP inspired library is the ability to select between events. C++CSP provides an `alt` type that can be used to select between a set of `guard` objects. Operator overloads and initializer lists are again used to simplify syntax. Listing 19 illustrates the use of `alt` in this manner.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

```
int main(int argc, char **argv)
{
  one2one_chan<int> a;
  one2one_chan<int> b;
  one2one_chan<int> c;
  one2one_chan<int> d;

  par
  {
    prefix<int>(0, c, a),
    delta<int>(a, {b, d}),
    successor<int>(b, c),
    consumer(d)
  }();
}
```

Listing 18: Parallel with Lists.

```
alt a{a, b, c, d};
// Perform fair select
auto selected = a();
// Perform fair select with guards
selected = a({true, true, false, true});
```

Listing 19: Example of C++CSP `alt` Use.

C++11 also introduced a `chrono` namespace to the standard library, which enables simpler interaction with clocks and temporal primitives. The C++CSP `timer` incorporates these temporal capabilities, with the ability to capture time points, and wait until either a time point or for a duration as available functions. `timer` extends `guard`, and has operator overloads to simplify use. Listing 20 illustrates the creation and use of a `timer` in C++CSP.

```
timer t;
// Get current time
auto now = t();
// Add five seconds
auto future = now + seconds(5);
// Wait until this time
t(future);
// Could just wait 5 seconds
t(seconds(5));
```

Listing 20: Example of C++CSP `timer` Use.

2.6. Exemplar Applications

To demonstrate how to use C++CSP, two exemplar applications are presented. The first, a *CommsTime* example, uses lambda expressions captured within a `par` block. The second, an implementation of the Dining Philosophers problem, uses defined function objects that are bound with parameters.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.6.1. CommsTime

Listing 21 provides an example of CommsTime where the individual processes are defined inline within the `par`.

```
1  #define seq [=]()
2
3  int main(int argc, char **argv)
4  {
5    one2one_chan<int> a;
6    one2one_chan<int> b;
7    one2one_chan<int> c;
8    one2one_chan<int> d;
9
10   par
11   {
12     seq // prefix
13     {
14       a(0);
15       while (true)
16         a(c());
17     },
18     seq // delta
19     {
20       while (true)
21       {
22         auto value = a();
23         par_write({b, d}, {value, value});
24       }
25     },
26     seq // successor
27     {
28       while (true)
29       {
30         auto value = b();
31         c(++value);
32       }
33     },
34     seq // consumer
35     {
36       while (true)
37         cout << d() << endl;
38     }
39   }();
40 }
```

Listing 21: CommsTime using Inline Lambda Expressions.

There are a few items of note in Listing 21. Firstly, line 1 defines `seq` as `[=]()`, and thus we can use `seq` at the start of our lambda expressions on lines 12, 18, 26, and 34. Secondly, note that channels are not converted to the respective ends, but written to and read from explicitly. Each channel type has operator overloads to enable use of the channel in this manner. Finally, line 23 demonstrates the use of the `par_write` helper function to write to a list of channels (b and d), also building the values to write to the channels as a list. Thereby, we can write a CommsTime application in 18 lines of C++CSP code (excluding curly brackets).

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.6.2. Dining Philosophers

For the Dining Philosophers example, N fork processes and N philosopher processes are controlled by a security process. Each process definition is declared as a lambda expression, assigned to a function object for binding in the network definition.

Fork Definition `FORK` uses an `alt` to select between the left and right owners. Listing 22 provides the definition of `FORK` in C++CSP.

```
1  auto FORK = [=](alting_chan_in<int> left,
2                  alting_chan_in<int> right)
3  {
4    alt a{left, right};
5    while (true)
6    {
7      switch (a())
8      {
9        case 0: left(); left(); break;
10       case 1: right(); right(); break;
11     }
12   }
13 };
```

Listing 22: `FORK` Definition.

Philosopher Definition The `PHIL` process uses a `timer` (declared line 5), and writes status updates to a printer process via a report channel. Between each state change, `PHIL` waits i seconds (e.g. line 8), i being the index of the philosopher.

```
1  auto PHIL = [=](int i,
2                  chan_out<int> left, chan_out<int> right,
3                  chan_out<int> down, chan_out<int> up)
4  {
5    timer t;
6    while (true)
7    {
8      report(to_string(i) + " thinking");
9      t(seconds(i));
10     report(to_string(i) + " hungary");
11     down(i);
12     report(to_string(i) + " sitting");
13     par_write({left, right}, {i, i});
14     report(to_string(i) + " eating");
15     t(seconds(i));
16     report(to_string(i) + " leaving");
17     par_write({left, right}, {i, i});
18     up(i);
19   }
20 };
```

Listing 23: `PHIL` Definition.

Security Definition `SECURITY` (Listing 24) ensures that a maximum of $N - 1$ philosophers are sitting at one time, through the use of pre-conditions during the select (line 8). Note the use of a list for supplying the pre-conditions to simplify the use of the select.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

```
auto SECURITY = [=](alting_chan_in<int> down,
                    alting_chan_in<int> up)
{
  alt a{down, up};
  int sitting = 0;
  while (true)
  {
    switch (a({sitting < N - 1, true}))
    {
      case 0:
        down();
        ++sitting;
        break;
      case 1:
        up();
        --sitting;
        break;
    }
  }
};
```

Listing 24: SECURITY Definition.

Process Network Definition Finally, the definition for the Dining Philosophers process network is provided in Listing 25.

```
1  using proc = function<void()>;
2
3  one2one_chan<int> left[N];
4  one2one_chan<int> right[N];
5  any2one_chan<int> down;
6  any2one_chan<int> up;
7
8  vector<proc> fork(N);
9  for (int i = 0; i < N; ++i)
10   fork[i] = make_proc(FORK, left[i], right[(i +1)%N]);
11
12 vector<proc> phil(N);
13 for (int i = 0; i < N; ++i)
14   phil[i] = make_proc(PHIL, i, left[i], right[i], down, up);
15
16 par
17 {
18   par(phil),
19   par(fork),
20   make_proc(SECURITY, down, up),
21   printer<string>(report, "", "")
22 }();
```

Listing 25: Dining Philosophers Process Network Definition.

Listing 25 is quite a typical manner for defining a set of parallel functions. For a shortcut, we declare `proc` to stand in place of `function<void()>`. We declare arrays of channels (lines 3 and 4), and then create `vectors` of processes for both the `FORK` (8 to 10) and `PHIL` (12 to 14) definitions. We can run these individual process `vectors` within inline `par` declarations (lines 18 and 19).

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.7. Summary

This section has provided a run down of some of the key features and implementations within C++CSP. As stated, the library itself provides an API similar to JCSP if required, but the use of various techniques enables some simpler definitions if required. The examples provided for CommsTime and Dining Philosophers illustrate these potential approaches to using C++CSP.

3. Methodology

The benchmarks used in this work incorporates two levels. To understand low level overheads of C++CSP a series of microbenchmarks have been undertaken, and potential speedup of applications developed in C++CSP are examined using macrobenchmarks.

The rest of this section describes the benchmarks used to evaluate the C++CSP library. The benchmarks are divided into two sections. Firstly, a description is given of the microbenchmarks used to compare to JCSP, allowing measurement of properties such as channel communication time and event selection time. Secondly, a description is given of the macrobenchmarks used to measure potential speedup when using C++CSP. The benchmarks chosen allow analysis of computation based work and memory based work to evaluate how well C++CSP supports both these factors.

3.1. Microbenchmarks

Microbenchmarks allow examination of three properties:

1. channel communication time (cost of coordination).
2. event selection time (cost of choice).
3. total number of processes (cost of process).

For property one a CommsTime benchmark is used. For properties two and three a StressedAlt benchmark is used.

3.1.1. CommsTime

The CommsTime microbenchmark includes four processes: *prefix*, *delta*, *successor*, and *consumer*. These four processes are connected together to produce the natural numbers. The single iteration around the network requires four channel communications, which means that the time of a single channel communication can be determined via recording the time to produce one number and dividing by four.

The *delta* process outputs on two channels, and may do so sequentially or in parallel. The CommsTime microbenchmark will examine both approaches as parallel communication incurs a process creation overhead. Also, as C++CSP can define a process in three separate manners (class, function, lambda expression), each of these methods will also be measured using CommsTime.

3.1.2. StressedAlt

The StressedAlt microbenchmark requires two types of process: a *reader*, and a *writer*. The *reader* process will perform a select operation on the set of incoming channels, then perform a read on the selected channel. The set of *writer* processes put the *reader* process under stress by communicating using these channels. The number of *writer* processes can be scaled to determine the number of processes the C++CSP library can support.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

3.2. Macrobenchmarks

Two macrobenchmarks have been developed to understand the speedup when using the C++CSP library:

1. Monte Carlo π Simulation.
2. Mandelbrot Fractal Generation.

Monte Carlo π provides a benchmark that is purely computational. Mandelbrot requires data to be generated by a worker and then sent to a gathering process, and therefore, memory communication overheads are possible.

3.2.1. Monte Carlo π

The Monte Carlo π simulation benchmark evaluates speedup within a purely computational setting. The benchmark uses random number generation to estimate π, with the greater number of iterations providing a better estimation. By dividing iterations across a number of workers, we can calculate potential speedup. No memory is allocated during the iteration, with only a few local variables required.

To evaluate C++CSP, the Monte Carlo π benchmark evaluates π within a collection of *worker* processes. The result of the local evaluation is then communicated to a *calculate* process for final evaluation. Listing 26 provides the process network definition of the Monte Carlo π benchmark within C++CSP.

```
1 any2one_chan<double> chan;
2 vector<function<void()>> workers;
3 for (int count = 0; count < NUM_WORKERS; ++count)
4   workers.push_back(make_proc(monte_carlo_pi, chan, iter_worker));
5 par
6 {
7   par(workers),
8   make_proc(calculate, chan, NUM_WORKERS)
9 }();
```

Listing 26: Monte Carlo π Parallel.

For benchmarking, 2^{30} iterations are performed and split between 2^0 to 2^5 worker processes.

3.2.2. Mandelbrot

The Mandelbrot fractal generation benchmark allows evaluation of speedup when working with some memory constraints. The fractal is a $n \times n$ matrix of 64bit floating point values, generated with an escape time algorithm. The data used has to be allocated and filled during the algorithm. When dividing the work between a collection of workers, the subsequent sub-matrices are sent to a gatherer process, and thus communicated between two concurrent scopes.

To evaluate C++CSP, the developed Mandelbrot benchmark uses a *producer* process to communicate line indices to a collection of `worker` processes. Each *worker* generates the data for said line and communicates it to a *consumer* process which stores the data in the correct order. Listing 27 provides the process network definition for the Mandelbrot benchmark in C++CSP.

For evaluation, fractals of dimension 2^8 to 2^{13} are generated with 2^0 to 2^3 worker processes. Each worker therefore generates 2^{11} to 2^{16} bytes of data and communicates it to the consumer process.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

```
one2any_chan<int> lines;
any2one_chan<mandelbrot_packet> data;
vector<function<void()>> workers;
for (int i = 0; i < NUM_WORKERS; ++i)
    workers.push_back(make_proc(mandelbrot, lines, data));
par
{
    make_proc(producer, lines, DIM, NUM_WORKERS),
    par(workers),
    make_proc(consumer, data, DIM)
}();
```

Listing 27: Mandelbrot.

The aim with the Mandelbrot benchmark is to determine if there are any copy overheads in the use of the channels. Therefore, three separate measures will be taken: copying into a channel; moving into a channel; and using a pointer to the data.

3.3. Approach

All experiments are undertaken on the following machine:

CPU Intel(R) Core(TM) i7-4770K running at 3.5GHz. Four cores with hyperthreading (eight hardware threads).
Memory 8GB DDR3 1333 MHz.
OS CentOS 7.2. Linux Kernel 3.10. 64bit.

Applications are compiled using the clang 3.4.2 C++ compiler with the posix thread model. Programs are compiled using full optimisation using the following command line:

```
clang++ -std=c++11 -O3 ...
```

As the clang compiler used for the experiments uses the posix threading model, the underlying thread model of both JCSP and C++CSP under test will be the same, operating system supported, method. JCSP benchmarks use JCSP-1.1rc4 built with the OpenJDK 1.8 64bit compiler. Each version of a benchmark is run 1000 times, with the mean time calculated from these 1000 iterations and used as the final reported value. Speedup is calculated using the standard measure:

$$S = \frac{t_{seq}}{t_{par}}$$

4. Results and Discussion

In this section, the results gathered from the benchmarks are presented. Firstly, the microbenchmark results and comparisons to JCSP are provided. Secondly, the macrobenchmark results and subsequent speedup calculations are provided. Finally, a discussion of the results is given. For all results gathered, the standard deviation was within 0.1% of the results provided.

4.1. Microbenchmarks

The microbenchmarks are used to calculate basic C++CSP properties: channel communication time, event selection time, and process load. These properties are compared to those of

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

JCSP. The first section presents the CommsTime benchmark results, and the second section the StressedAlt benchmark results.

4.1.1. CommsTime

Table 1 presents the results for the CommsTime benchmark.

Table 1. CommsTime Benchmark Results. Times in nanoseconds.

Approach	Channel Time	Estimated Context Switch
JCSP	2,649	1,325
JCSP Seq	3,476	1,738
C++CSP	4,435	2,218
C++CSP Seq	1,994	997
C++CSP `make_proc`	4,532	2,266
C++CSP `make_proc` Seq	1,997	999
C++CSP lambda	4,481	2,241
C++CSP lambda Seq	2,092	1,046

It can be seen that C++CSP performs better than JCSP for the sequential benchmark, but poorer when analysing the parallel benchmark. This is due to the extra process creation, scheduling, and destruction when using the parallel *delta* process. Therefore, we can ascertain some measures from Table 1:

- C++CSP channel communication on the test machine is approximately 2μs.
- C++CSP context switch time on the test machine is approximately 1μs.
- C++CSP has an approximate 25% faster channel communication time in comparison to JCSP.
- C++CSP takes approximately 1.2μs to create, schedule, and destroy a process (C++CSP Seq result subtracted from C++CSP result and divided by two for the two processes created by *delta*).
- C++CSP there is no consistent difference when using the different process creation methods.

4.1.2. StressedAlt

Table 2 presents the StressedAlt benchmark results.

Table 2. StressedAlt Benchmark Results. Times in nanoseconds.

Channels	JCSP Select	C++CSP Select
64	990	750
128	890	845
256	965	825
512	975	787
1,024	1,139	880
2,048	1,386	958
4,096	FAIL	FAIL

As shown, C++CSP has a faster select time, ranging from 45 to 428 nanoseconds faster depending on the channel count. The more interesting number is when the benchmark fails, which is after the 2048 channel count. In this scenario, 2049 threads will have been created. Both JCSP and C++CSP fail at the same point as they are both using the same threading model. On testing, approximately 3000 threads can be created in the test.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

The low process count is a result of the OS being used. When the similar benchmark is performed on a lower powered laptop with a newer Linux kernel, approximately 10000 threads can be created. Therefore, C++CSP suffers from the same process count limitations as JCSP when compiled with most modern C++ compilers, and it is really the OS that is bounding the process limit.

4.2. Macrobenchmarks

The macrobenchmarks are used to evaluate speedup when using C++CSP in certain application scenarios. The two chosen benchmarks allow analysis of speedup in a purely computational scenario as well as a scenario where memory copying s used. The first section presents the Monte Carlo π benchmark results and the second section the Mandelbrot fractal results.

4.2.1. Monte Carlo π

The results for the Monte Carlo π simulation benchmark results are provided in Table 3.

Table 3. Results for Monte Carlo π Benchmark.

Number of Workers	ms	speedup
1	193.84	-
2	96.95	2.0
4	51.09	3.79
8	32.87	5.90
16	32.92	5.89
32	32.87	5.90

When dealing with a purely computational load, C++CSP performs well. For two workers, a two times speedup is recorded. For four workers, a 3.79 speedup is recorded. The slight drop in performance is likely due to some cross core context switching. Finally, speedup levels out at 5.9 times for eight workers and above.

4.2.2. Mandelbrot

The results for the Mandelbrot fractal benchmark are split across three tables: copy communication; move communication; and mobile communication. Table 4 presents the results when calling write on the channel in the standard manner.

Table 4. Results for Mandelbrot Benchmark.

Dimension	1 Worker		2 Workers		4 Workers		8 Workers	
	ms	*speedup*	*ms*	*speedup*	*ms*	*speedup*	*ms*	*speedup*
256	18.04	-	9.33	1.93	5.05	3.57	4.44	4.06
512	21.79	-	11.11	1.96	6.84	3.19	6.07	3.59
1,024	33.74	-	17.01	1.98	11.69	2.88	10.15	3.32
2,048	73.73	-	40.02	1.84	25.53	2.89	20.14	3.66
4,096	230.24	-	124.94	1.84	80.99	2.84	63.73	3.61
8,192	837.94	-	446.74	1.88	252.89	3.31	210.72	3.98

As shown, speedup is a little less than in the Monte Carlo π benchmark, however there is now some memory allocation and cleanup happening. The speedups are fairly consistent, although there is a slight slowdown when working with four threads on data sizes 1024, 2048, and 4096.

In comparison, Table 5 presents the results when data is moved into the channel upon the write operation.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

Table 5. Results for Mandelbrot Benchmark using Move Semantics.

Dimension	1 Worker		2 Workers		4 Workers		8 Workers	
	ms	*speedup*	*ms*	*speedup*	*ms*	*speedup*	*ms*	*speedup*
256	18.22	-	9.32	1.95	4.99	3.65	4.41	4.13
512	21.96	-	11.18	1.96	6.67	3.29	6.11	3.59
1,024	32.81	-	17.31	1.90	10.26	3.20	9.87	3.32
2,048	73.58	-	39.02	1.89	25.32	2.91	23.19	3.17
4,096	227.81	-	119.08	1.91	70.08	3.25	57.31	3.98
8,192	826.95	-	440.54	1.88	260.58	3.17	207.94	3.98

When using move there is an occasional improvement in performance, but it is not consistent. It can be observed that in general the time overall time is lower, particularly for larger data sizes. Therefore, it can be speculated that there is a slight improvement from avoiding the extra copy because of the move.

To fully test the communication copy overhead, a `mobile` data type is defined as given in Listing 28. By defining and using this type, it is now assured that the communication is only copying the pointer within the `mobile` type (an 8 byte transaction). Table 6 presents the results for this version of the Mandelbrot benchmark.

```
1 template<typename T>
2 using mobile = unique_ptr<T>;
```

Listing 28: Defining a `mobile` Type.

Table 6. Results for Mandelbrot Benchmark using Mobile (`unique_ptr`) Semantics.

Dimension	1 Worker		2 Workers		4 Workers		8 Workers	
	ms	*speedup*	*ms*	*speedup*	*ms*	*speedup*	*ms*	*speedup*
256	18.46	-	9.31	1.98	5.00	3.69	4.40	4.20
512	21.85	-	11.43	1.91	7.28	3.00	6.05	3.61
1,024	33.37	-	17.11	1.95	10.99	3.04	9.58	3.48
2,048	73.95	-	39.59	1.87	25.93	2.85	20.40	3.63
4,096	229.91	-	121.98	1.88	70.47	3.26	60.02	3.83
8,192	809.95	-	444.48	1.82	259.45	3.12	206.21	3.93

The results in Table 6 show a similar pattern to those between the previous two implementations of the benchmark - inconsistency. Although occasional improvements are seen, there are occasions when this is not so.

From analysing the three tables of results for the Mandelbrot benchmark, it can be determined that there appears to be little if any overhead when copying data into the channel. This is promising as it means the channel itself does not occur much of an overhead due to the data being moved through. Further testing with larger data types is required to truly determine if this is the case, but it appears likely.

4.3. Discussion

The results presented in this section have shown that C++CSP performs well in comparison to JCSP, and also performs well when looking at potential speedup.

A number of points are worth considering from the benchmark results. Firstly, although the parallel helper methods are useful to the programmer, they are not necessarily useful for performance having more than doubled the CommsTime result for C++CSP. Therefore, it

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

could be worth considering other techniques for building such functionality such as through a thread pooling mechanism. This would allow threads to be reused more readily and overcome the limitations of the helper functions.

The limitation on process numbers that also impacts JCSP is an issue that needs further investigating. A possible avenue is moving towards a coroutine model similar to that developed for C# [8]. The C++ standard committee is currently reviewing a proposal for coroutine support in the standard library that would enable such functionality.

The speedup measures are promising, but do require further evaluation. Computational loads appear good, and memory loads seem to be supported well. However, further evaluation around the size of the data used will help the argument.

In all, it can be determined that C++CSP is useful, and provides performance expected for a CSP inspired library. The additional syntactic sugar in the API will therefore hopefully make it appealing to CSP library users in the future.

5. Conclusions

This paper presented work in the development and evaluation of a new C++CSP library built using modern C++ standards and design principles. The library was evaluated against JCSP using microbenchmarks, and for speedup using macrobenchmarks. From these benchmarks the following conclusions can be made:

1. C++CSP performs better than JCSP in regards to channel communication time and event selection time.
2. C++CSP will create as many processes as JCSP when built with a compiler using the same threading model. There is no additional overhead for C++CSP processes.
3. In computational loads, C++CSP provides an almost six times speedup when working with a suitable quad-core processor supporting hyperthreading.
4. In conditions where memory copying is used, a potential four times speedup is possible.
5. C++CSP channels effectively support move semantics to limit memory copying.

5.1. Future Work

Three pieces of further work are expected for C++CSP. Firstly, the development of further benchmarks will be undertaken to further evaluate C++CSP's suitability for undertaking parallel application development. Secondly, the development of a supporting network stack will be undertaken. This will allow C++CSP to be used in distributed parallel environments. The aim is to develop an MPI backend as part of this work. Finally, the development of a skeleton library using C++CSP is planned. This library will use the same design principles as C++CSP itself to provide a simple skeleton API for developers.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[2] Peter H. Welch, Neil C.C. Brown, James Moores, Kevin Chalmers, and Bernhard H.C. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, jul 2007.

[3] Jan F. Broenink, Andrè W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, mar 1999.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

[4] Brian Vinter, John Markus BjÃČÂÿrndalen, and Rune MÃČÂÿllegard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, nov 2009.

[5] Neil C.C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83, sep 2008.

[6] Kurt Micallef and Kevin Vella. Communicating Generators in JavaScript. In *Communicating Process Architectures 2016*, 2016.

[7] Neil C.C. Brown. C++CSP2: A Many-to-Many Threading. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–206, jul 2007.

[8] Kenneth Skovhede and Brian Vinter. CoCoL: Concurrent Communications Library. In *Communicating Process Architectures 2015*, 2015.

[9] Cabel Shrestha and Jan BÃękgaard Pedersen. JVMCSP - Approaching Billions of Processes on a Single-Core jvm. In *Communicating Process Architectures 2016*, 2016.

[10] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, sep 2008.

[11] James Moores. Native JCSP - the CSP for Java library with a Low-Overhead CSP Kernel. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 263–274, sep 2000.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.