Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018
© 2018 The authors and IOS Press. All rights reserved.

A Transparent Thread and Fiber Framework in C++CSP

Kevin CHALMERS

School of Computing, Edinburgh Napier University, Edinburgh, EH10 5DT, Scotland.

k.chalmers@napier.ac.uk

Abstract. There are multiple low-level concurrency primitives supported today, but these often require the programmer to be explicit in their implementation decisions at design time. This work illustrates how a process-oriented model written in C++CSP can hide the underlying primitives from the programmer to allow M:N style thread support. The objective is to provide integrated kernel-level and user-level thread support in C++CSP without major changes to the process interface. To illustrate that kernel-level and user-level threads are working together two experiments have been undertaken. The first executes a stressed select to determine the cost of process-count scaling. The second experiment executes a scaling number of processes syncing across hardware to illustrate the sync-time cost as the number of user-level threads increases in each context. The results illustrate that it is possible to build M:N style thread support transparently for process design, and that doing so provides a significant increase in the number of active processes in library supported CSP while still taking advantage of multicore hardware.

Keywords. threads, fibers, M:N threading, C++CSP

Introduction

Today, there are numerous concurrency primitives available to the programmer. A concurrency primitive is a language or library construct that allows expression of or control over concurrent behaviour. Threads, actors, mutexes, and channels are all such primitives. Each primitive attempts to overcome a limitation of previous concurrency mechanisms and therefore the scope for potential primitives is unbounded.

Hardware support for concurrency also comes in several forms. Such support allows the actual running of a computation in parallel. Multicore processors, atomic instructions, SIMD instructions, and networked hardware each provide different parallel support to the programmer. Without parallel hardware the key benefit of performance in concurrency is not possible.

Matching concurrency primitives to hardware is where the challenge lies. System processes (e.g. application instances) map to a computer and threads to a core, but the majority of programmers do not think beyond this level. Operating systems typically use a coarse-grained thread and system process model with mutex support, and therefore the default is to use the operating system model as the concurrency model. Atomic and SIMD instructions may be used, but this is typically determined by the compiler or operating system developer rather than the programmer.

A concurrent process designer is a software engineer. As such, they do not want to consider hardware unless entirely necessary. This leads to problems when the process designer

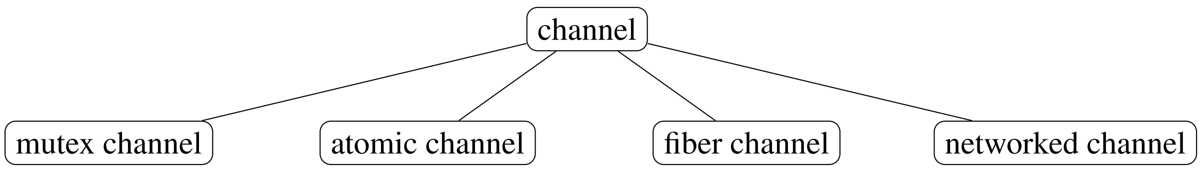

Figure 1. Channel types increase in number as different implementation models are supported. The different models also require input and output ends, as well as shared equivalents.

wishes to express further concurrency within their process. When executing a parallel, should kernel-level threads or user-level threads be used? Should a choice occur using user-level or kernel-level thread controls? The process designer does not know the answer to these questions at design time, and providing multiple versions of the same process is wasteful. It also leads to an explosion of primitive types, as illustrated in Figure 1.

Therefore, we need to support primitives transparent to the underlying operating system and hardware execution mechanisms. A process should not be designed to be either kernel-level or user-level executed, but the system implementer may choose to use either or both. A process should be able create a channel, parallel, or choice without explicitly knowing at design time if it is executing on a kernel-level or user-level thread. This is not just a problem of inheritance but encapsulation. Overcoming this issue and supporting both kernel-level and user-level threads transparently in single application is the goal of this work.

Kernel-level and user-level threads highlight the challenge of transparent concurrency support. A process when designed does not know if it will be working in a kernel-level or user-level environment. Therefore, the process may have channels which have a kernel-level mutex, a user-level mutex, or some other coordination primitive internally. When performing a select the process has to decide which type to use. A kernel-level mutex select can lead to undefined behaviour if it attempts to enable a user-level channel because multiple kernel-level threads will be accessing a single-kernel-thread resource. Conversely, a user-level select with kernel-level channels will not lead to undefined behaviour, but will be slow in comparison to a pure kernel-level thread select as it will lock the user-level scheduler. Thus, we need to transparently support the right type of concurrency behaviour at runtime which is undefined at design-time.

Libraries developed for CPA have poor co-support for multicore user-level threading. The original C++CSP [1] supported both kernel-level and user-level threads, although the latter was Windows only. The limitation of the original C++CSP was this lack of cross-platform support and that parallel execution defaulted to user-level threading in this case unless the programmer specified that a kernel-level thread was required. PyCSP has also been examined using different underlying concurrency approaches [2]. However, the decision on which support mechanism was based on an import statement at the start of the program. The different approaches were distinct, and kernel-user thread co-support was not provided. Other libraries such as JCSP [3] are based on threading of the JVM which is typically kernel-level.

Languages have been more successful in supporting multicore user-level threading. occam-π, Go and Erlang each support multicore scheduling of user-level threads [4]. This is done transparently to the programmer as the languages each provide a runtime that takes care of scheduling. Languages have an advantage here as any library has to work with the host language to provide the support.

Other approaches to concurrency have also been explored in CPA libraries. New C++CSP [5] has examined atomic operations to support channel communication. Haskell CSP [6] utilises Software Transactional Memory (STM) for its underlying concurrency. Both of these libraries point to successes in implementing channel-based models on different underpinning concurrency models, but it adds to the complexity of transparent implementation at the process design stage.

The long-term goal in C++CSP is to support as many underlying concurrency models as feasible, including networked hardware. The objectives of this work are threefold:

1. Support user-level threading within C++CSP.
2. Update C++CSP to support both kernel-level and user-level threading as transparently as possible.
3. Measure the effectiveness of using kernel-level and user-level thread support in a single application.

The first objective is trivial as cross-platform user-level threading is provided by the Boost.Fiber Library[1]. Boost.Fiber provides an interface similar to the C++ threading library so translation is straightforward for the majority of C++CSP. The only change was in fiber-management which required the use of `thread local` storage for fibers for each thread. A brief discussion on Boost.Fiber is provided in Section 1.

The second objective required significant modification of the implementation of C++CSP while trying to maintain the same external interface. This work is detailed in Section 2. Objective three is met with some standard benchmarks provided in Section 3. Final conclusions and future work are detailed in Section 4.

1. Boost.Fiber User Level Threads

C++ is scheduled to support coroutines in the C++20 standard. At this point, C++ will have an official supported mechanism to provide user-level concurrency. Boost aims to overcome the limitations of the C++ standard library by providing behaviour not yet added to the standard. It is often seen as an incubator for future C++ standards.

Boost provides user-level threading in two coroutine libraries (stack-based and stackless) and a fiber library. These libraries use the the Boost.Context[2] library which allows capturing and switching of execution states, such as the stack and registers. It does so using low-level continuation capture mechanism which does not require an OS-level interrupt; therefore a context-switch can be undertaken in a few clock-cycles rather than a few thousand.

Boost fibers provide an interface similar to a standard C++ thread. A function can be used to create the fiber and on creation the scheduler will manage its execution. User-level mutexes and condition variables are provided which look like C++ kernel-level mutexes and condition variables. The programmer uses these types with fibers and thereby the scheduler can manage context switching as required. As such, creating C++CSP user-level thread code is a simple case of changing from using `std::thread` and `std::mutex` to `boost::fibers::fiber` and `boost::fibers::mutex` as the used types.

2. Providing a Transparent Primitive Interface

C++CSP originally aimed to support CSP primitives in a trivial interface to the programmer. Importantly, pointers were hidden from the API user. This was considered important as maintaining resources between concurrent threads is a problem and using pointers would require some thought from the C++CSP user which was not about the design of processes. API design is a key-feature for any library adoption, and Listing 1 illustrates a simple program in original C++CSP.

```
#include <iostream>
#include <csp/csp.h>
```

[1]https://www.boost.org/doc/libs/1_67_0/libs/fiber/doc/html/fiber/overview.html
[2]https://www.boost.org/doc/libs/1_67_0/libs/context/doc/html/index.html

```

using namespace std;
using namespace csp;

void producer(chan_out<int> out)
{
    out(10);
}
void consumer(chan_in<int> in)
{
    cout << in() << endl;
}
int main(int argc, char **argv)
{
    one2one_chan<int> c;
    par
    {
        make_proc(producer, c),
        make_proc(consumer, c)
    }();
    return 0;
}
```

Listing 1: Trivial C++CSP Program.

A recent extension focused on performance improvements via atomic operations. This required the implementation of a `busy_one2one_chan` and similar types. The aim of the extension was the exploration of atomic operations for implementation of a channel and the required extensions were simple yet added eight new types to the library, and would have required a further eight for full-implementation. This illustrated that a single extension alone has cascading effects throughout the library which needed to be addressed.

Next stage C++CSP development was user-level thread support. Atomic operations had provided fast inter-core communication, being under 150ns for a channel communication between processes on different cores. User-level threading was seen as providing the best support for intra-core communication and concurrency. However, this would lead to further types such as `fiber_one2one_chan`, etc.

Much of these problems came from the approach of implementing C++CSP from JCSP. JCSP was designed for Java and had to work within Java typing constraints. C++CSP followed the model as it made sense from an object-oriented point-of-view. However, like JCSP, adding new underlying concurrency support led to work across the code-base. As such, some thought was required in how to reduce type proliferation while still enabling the application implementor the flexibility in choosing their concurrency model.

Providing a simplified interface to C++CSP primitives became the key problem. The implementation requires a hierarchy of types (such as `guarded_chan_in` inheriting from `chan_in`), while trying not to expose various implementation approaches (`busy_chan_in`, `fiber_chan_in`, etc.). Such considerations were required across all CSP primitive types. The rest of this section examines the original C++CSP implementation and the new C++CSP implementation with consideration of how types are categorised to ensure concurrency implementation conflicts do not occur.

2.1. Original C++CSP Implementation

C++CSP provides a modern C++ interface. That is, there are types that manage internal data hidden from the programmer so there are no concerns around memory management. The code in Listing 1 illustrates that channels can be typed and automatically convert to their relevant ends with programmer intervention. Some data types (e.g. `par`) allow construction via initializer lists to simplify development. These mechanisms allow a cleaner API to the programmer.

Such development required use of certain C++ idioms. PIMPL (Pointer to IMPLementation) and RAII (Resource Acquisition Is Initialisation) are used throughout. PIMPL is a technique where a lightweight outer class contains a pointer to an implementation class that has the actual data and behaviour defined. PIMPL means that a programmer does not worry about resource lifetime as they do not explicitly create objects on the freestore. RAII is a technique where an object is created where needed (e.g. passed in as a newly created object as a parameter). The RAII idiom also negates the need for programmers to keep track of resources. These idioms are common practice in C++ today; however, they can lead to code bloat, which adds to the concerns around type explosion already indicated.

To illustrate the problem, a common approach to developing a CSP inspired library is to develop different channel types. JCSP, the template for many CSP libraries, documents six channel services:

- One-to-one.
- One-to-any.
- Any-to-one.
- Any-to-any.
- One-to-all.
- Symmetric (guarded at both ends) one-to-one.

Each of these services would require a different implementation type (e.g. fiber, atomic, etc.). This leads to six new types being generated for each new concurrency primitive. With those, three versions (normal, guarded, shared) of two end types (input and output) must be created. That is a further six types for each implementation type. Finally, it is likely there is a channel implementation type to actually support the distinct behaviour. Coupled with barriers, alternatives and parallels, we end up with close to twenty new types for each concurrency support mechanism.

This led to the underlying architecture of C++CSP to be updated. The aim is to provide a template to other CSP library designers. As inheritance is intrinsic (normal, guarded, and shared input ends) while implementation specifics need hidden to the process designer, the template should support further library developments in the area.

2.2. New C++CSP Implementation

The new implementation aims to keep as much of the original interface as possible. Doing so ensures existing code will still work, and will keep API looking simple. Some changes were required to meet the requirements of reducing type bloat, hiding implementation details, and ensuring that the implementor can make decisions. Listing 2 is Listing 1 in the new C++CSP interface.

```
#include <iostream>
#include <csp/csp.hpp>

using namespace std;
using namespace csp;

void producer(chan_out<int> out)
{
    out(10);
}

```

```
12  void consumer(chan_in<int> in)
13  {
14      cout << in() << endl;
15  }
16
17  int main(int argc, char **argv)
18  {
19      using model = thread_model;
20      auto c = model::make_one2one<int>();
21      parallel<model>
22      {
23          make_proc<process_function>(bind(producer, c)),
24          make_proc<process_function>(bind(consumer, c))
25      }();
26      return 0;
27  }
```

Listing 2: Trivial New C++CSP Program.

Process definitions as functions still operates as before. The change comes in `main` where implementation details are specified. Line 19 defines a model that will be used (`thread_model`) which is used to create a channel. The `using` keyword is just defining an alias for a type so we do not have to type `thread_model` all the time. The parallel definition requires the type to be specified at this level. The use of `make_proc` now requires a type, which is used to build a `proc_t` type. This acts as a container to a process to enable subsequent building methods.

A key requirement is that a process now needs to know its type to use correct primitive implementation. Therefore, a `process` type becomes more important. In the original C++CSP implementation, processes were functions. This is not possible as a function knows nothing about its underpinning concurrency type. So, lines 23-24 show that functions are usable but not necessary simple.

```
#include <iostream>
#include <csp/csp.hpp>

using namespace std;
using namespace csp;

class producer final : public process
{
private:
    chan_out<int> out;
public:
    producer(chan_out<int> out)
    : out(out)
    {
    }

    void run() noexcept
    {
        out(10);
    }
};

class consumer final : public process
{
private:
    chan_in<int> in;
public:
    consumer(chan_in<int> in)
    : in(in)
    {
    }

    void run() noexcept
    {
        cout << in() << endl;
    }
};

int main(int argc, char **argv)
{
    using model = thread_model;
    auto c = model::make_one2one<int>();
    parallel<model>
    {
        make_proc<producer>(c),
        make_proc<consumer>(c)
    }();
    return 0;
}
```

Listing 3: Using `process` Types.

When the `process` type is extended (see Listing 3) the process designer gains access to a collection of helper functions inherited from `process`. These include:

`par` to launch child processes in parallel using the correct concurrency model.
`make_one2one` to create a one2one channel using the correct concurrency model.
`make_alt` to create an alternative from a collection of channels using the correct concurrency model.
`par_write` to write to a collection of channels in parallel using the correct concurrency model.

Other helper functions for different types (e.g. one2any) and behaviours (e.g. parallel read). The outcome is a process that has a richer interface due to its need to provide access to the correct concurrency model, while hiding the internal model from the designer.

2.2.1. Classification of Types

A consideration had to be made on how the different underlying concurrency types should be classified. This is because some types cannot be used together, such as a kernel-level thread not using a user-level channel, but the inverse being allowed. A hierarchy of types was defined (Figure 2) to understand the models to be provided.

Inter-machine	Network		
Inter-core	Kernel-mutex	Atomic-mutex	
Intra-core	Fiber-mutex	*Coroutine*	*STM*

Figure 2. Classification of concurrency types for CSP libraries. Emphasised types currently not supported in C++CSP. Lower-types can use primitives defined in upper-types, but upper-types cannot use lower-types. Types at the same level may have problems if a different scheduler is used.

Three type-levels exist for current channel implementations seen in CSP-inspired libraries:

inter-machine allows communication between different machine instances. At present, only networked channels exist at this level, although in theory anything that provides I/O with a system could be provided with a channel like interface.

inter-core allows communication between processor cores. At present, only OS-thread level approaches provide this functionality in CSP libraries, although occam-π and Go can do so as standard.

intra-core allows communication between processes on the same core. User-level threads are the main type here. It can be argued that a user-level thread may switch core, or that mechanisms can be used to enable user-level threads to communicate across core, and this is correct. The reason for the intra-core classification is that a collection of user-level threads are scheduled within a thread, which is itself scheduled on a core.

The key problem is ensuring correct use of primitives. Concurrency at the top is usable down the hierarchy, but at a cost of blocking schedulers. Concurrency at the bottom is not usable higher-up. The models must abstract potential underlying concurrency support from the process designer, but allow a system implementor to specify exactly what is required. So, a process can have a set of input and output channels defined, but the implementor must decide if these channels are user-level, kernel-level, or other.

2.2.2. Model Types

The system implementor specifies application execution by choosing a model. A model is currently a machine specific definition, and thus consists of either an inter-core approach, intra-core approach, or a combination thereof. Five models are currently provided in C++CSP:

`thread_model` uses kernel-level threads and mutexes.

`atomic_model` uses kernel-level threads and atomic channels and barriers.

`fiber_model` uses user-level threads and mutexes.

`thread_fiber_model` initial parent parallel creates kernel-level threads, child processes create user-level threads if further concurrency is required.

`atomic_fiber_model` as `thread_fiber_model` except channels and barriers provided are atomic based.

A model is defined as a C++ `struct` with type definitions. A model provides five types:

`par_type` internal parallel representation used by the model.

`chan_type` internal channel representation used by the model.

`chan_end_mutex` the mutex type used to protect a channel end if it is shared.

`bar_type` internal barrier representation used by the model.

`alt_type` internal alternative representation used by the model.

These types are implemented in a separate namespace (e.g. `csp::thread_implmentation`). It is in these types that implementation functionality is provided. A thread model `par_type` will use standard C++ threads and mutexes, whereas a fiber model `par_type` will use fibers and fiber mutexes. Channel types are simplified by a single channel type pointing to a `chan_type` implementation using the PIMPL idiom. The use of abstracted mutex types (`chan_end_mutex`) means the input and output ends can also be abstracted in this manner.

A model also provides factory methods to create channels and barriers:

- `make_one2one`.
- `make_any2one`.
- `make_one2any`.

- `make_any2any`.
- `make_bar`.

As further features are added, future requirements for alting barrier, symmetric and broadcast channels will be met. The overhead is minimal the internal representations are hidden, leading to these types only requiring implementation types for the relevant models.

3. Analysing Thread and Fiber Approaches

Three metrics have been gathered to evaluate combined kernel-level and user-level threading in C++CSP. Communication time represents the time taken for a single channel communication. It provides a reasonable estimate of context-switch time as a channel invokes two context-switches during value hand-over. Stressed select has also been undertaken using user-level threads. This is to examine the limits of thread creation which is a weakness in kernel-thread based libraries. Finally, a stressed barrier application is developed which examines multiple user-threads synchronising within multiple kernel-threads.

Experiments are run on a single machine as specified below:

CPU	Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
Cores	4 physical, 8 logical
Memory	8GB DDR3 1600MHz
Operating System	Linux Kernel 4.4.0
Compiler Toolchain	GCC 8.1.1

3.1. Communication Time

The Communication Time Benchmark measures the time it takes for a message to send from one process to another. It does so by running a process network as illustrated in Figure 3. Four channels are in use, thus producing a number via four channel-communications. The Delta process can output values on its two outputs either sequentially or in parallel, leading to analysis of the thread start-up time. Figure 4 provides the results for single-channel communication time for fiber, atomic and thread mutex channel models.

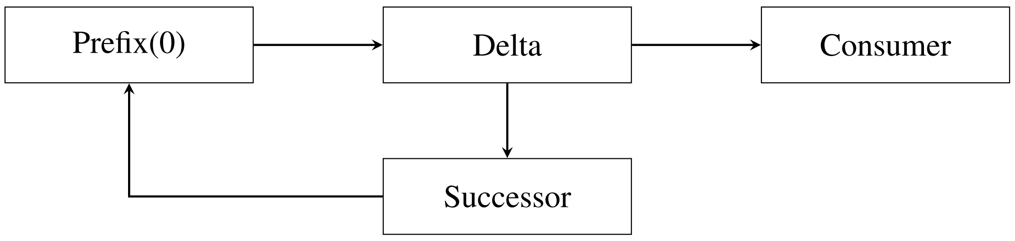

Figure 3. Communication Time Benchmark Process Network. Will produce the natural numbers starting from 0. As only computation is in successor to increment input by one the benchmark is synchronisation based.

Sequentially, atomic communication is still the fastest at 148ns. This is a cross-core communication relying on a memory sync. Fibers communicate at 272ns which will not require the same cross-core communication and memory sync, while kernel-thread mutexes take 3041ns. Atomics are faster sequentially as the processes are constantly trying to complete the communication – no process is descheduled. As such, the atomic measure will be close to the memory sync time between the separate processes. Fiber communication will require processes to be rescheduled to allow communication and progress.

For parallel Delta communication fibers are the fastest at 579ns. This provides a rough estimate of fiber creation and scheduling time of approximately 300ns. Kernel-threads take

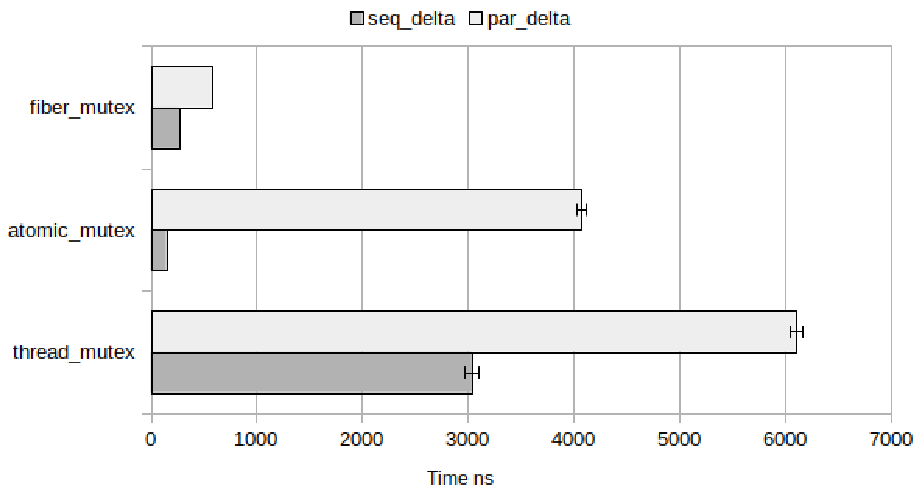

Figure 4. Communication Time Benchmark results for thread, fiber and atomic mutexes both sequentially and in parallel. Atomic channels are fastest when hardware supports the concurrency but have a major drop in performance when more threads are created as shown in the parallel delta results.

longer as illustrated in the `thread_mutex` timings where a communication is 6104ns, approximately 3060ns greater. Atomic communication is 4074ns which is approximately 3930ns greater than the sequential version. The reason atomic communication invokes the greatest increase is the additional kernel-threads for the parallel delta causing concurrency to no longer match the hardware, thus causing some scheduling overhead. See [5] for more details of this cost.

In previous work [4], Go and Erlang have been measured for communication time. Both of these languages provide a multicore runtime with user-level threads. On the same machine, Erlang had a communication time of 500ns and Go 900ns. Therefore, fibers in C++CSP have a lower communication overhead than these two languages, although it may be an unfair comparison if there is cross-core communication occurring.

The data provides three results of interest:

1. Atomic communication is faster than fiber communication when we can match the concurrency to the hardware. Thus, to best exploit hardware, atomic communication between cores should be the aim.
2. Fiber communication is reasonably close to atomic communication, and greater than an order-of-magnitude faster than kernel-thread approaches. Fiber communication is therefore good for intra-core communication.
3. Fiber start-up time is an order-of-magnitude faster than kernel-thread start-up time. Therefore, fibers also have low overheads when considering applications which have concurrency creation – a weakness of the kernel-thread approach.

3.2. Stressed Select

The Stressed Select Benchmark provides information on two areas: the time it takes to schedule an increasing number of processes; and the limit of the number of processes that can be created by the concurrency runtime being used. The process network (Figure 5) has a number of writers sending to a single reader which must select a ready channel. The number of writers can be increased to stress the writer and determine a limit to the number of processes that can be created. C++CSP already has results around kernel-threading (see [5]) so Figure 6 only presents fiber-based results. The timings represent how long it takes to perform a single select on the number of incoming channels, which scales from 2^1 to 2^{20}.

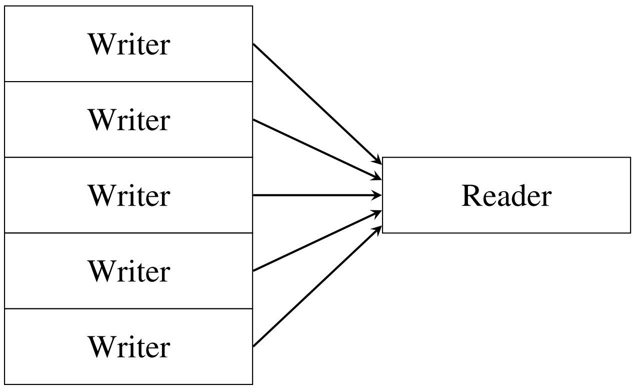

Figure 5. Stressed Select Benchmark Process Network. N writers send to a single reader which must select a ready channel. As a writer must be ready to be selected the benchmark gives some indication to scheduling cost.

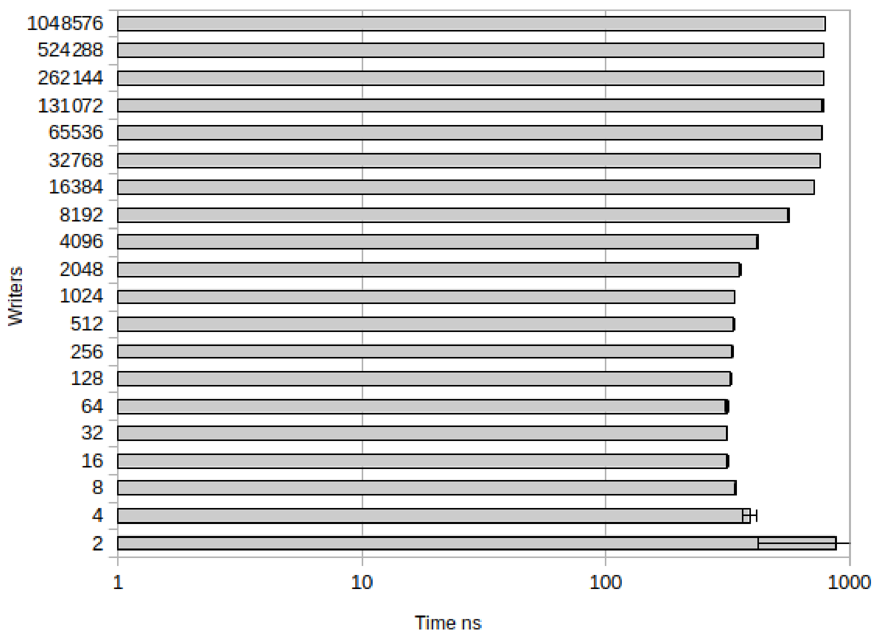

Figure 6. Stressed Select Benchmark results for the fiber implementation. The application fails when 2^{21} fibers are created.

Select time ranges from 313ns (32 writers) to 884ns (2 writers). The later result is likely high due to the speed of the application, and hence the error visible. 2^{20} writers takes 788ns for a single select. Not easily visible is the slight increase in select time as the number of writers is doubled. Therefore, Table 1 provides the raw results.

When 2^{21} fibers are created the application fails. This appears to be a memory resource problem, and perhaps using a machine with more memory will allow more fibers. This is left for future work. What has been shown is the normal limit on a kernel-thread application (approximately 9000 writers on the machine used) is no longer a limit, and the overhead for scheduling and managing large numbers of fibers appears small. Therefore, implementing user-level threads in this manner provides C++CSP with a millions of concurrent processes, and with a small scheduling overhead.

3.3. *Stressed Barrier*

The Stressed Barrier Benchmark measures the synchronisation time for multiple groups of user-level threads across different kernel-level threads. It does this by synchronising M user-

Table 1. Stressed Select Results.

Number of Writers	Select Time ns
2	884.52
4	388.76
8	341.61
16	316.20
32	313.65
64	314.30
128	325.28
256	329.91
512	334.44
1024	338.34
2048	355.32
4096	418.80
8192	561.14
16384	714.74
32768	758.02
65536	771.01
131072	775.81
262144	777.77
524288	781.81
1048576	788.24

level threads on a kernel-thread. When the user-level threads have synchronised, the kernel-level thread synchronises with other kernel-level thread controllers and a timer process via an atomic barrier. Therefore, $M \cdot N + 1$ processes are synchronising. The process network is provided in Figure 7. The aim of the benchmark is to determine if multicore hardware is being used, and the overhead of synchronising large process groupings across cores. Figure 8 provides the result timings for a single synchronisation of all the processes.

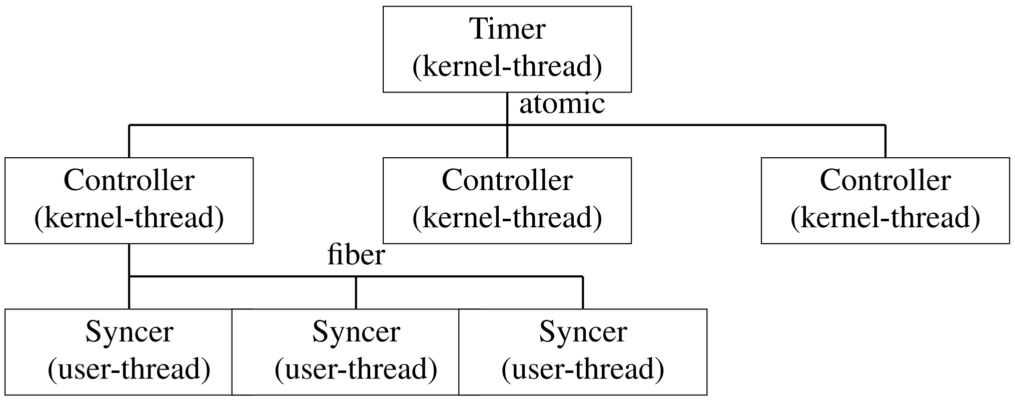

Figure 7. Stressed Barrier Process Network. N controllers on kernel-threads each synchronise with M child processes using user-level threads. The controllers then synchronise together with a timer via an atomic barrier. This allows examination of the overhead of increasing N to determine if parallel hardware is truly being used.

As the results indicate, increasing the number of fiber groups does not increase synchronisation time proportionately. For example, to synchronise 65536 user-level threads on various controllers are as follows:

one controller – 65,537 total processes 30.4ms.
two controllers – 131,073 total processes 37.8ms.

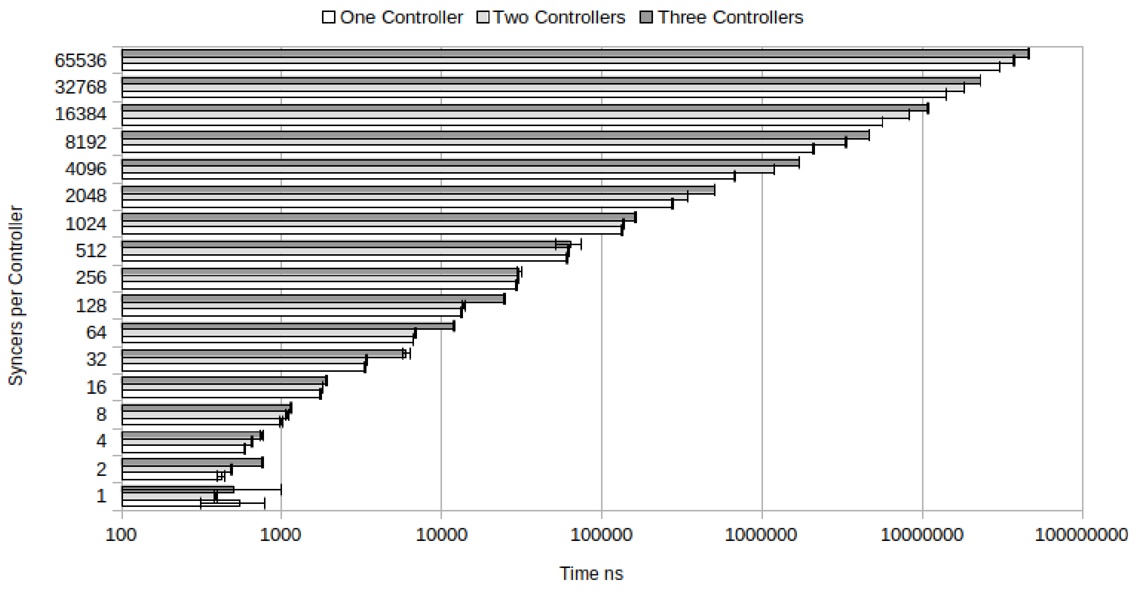

Figure 8. Stressed Barrier Benchmark results for combined thread-fiber application. Each controller runs on a separate kernel-thread.

three controllers – 196,609 total processes 46.3ms.

There is a cost as the number of controllers increases, but it does not increase at the same rate as the total number of processes. Therefore, scaling the using fibers across cores does provide a benefit. It does lead to an application which can synchronise approximately 200,000 processes in 50ms.

4. Conclusions and Future Work

This paper has examined a transparent framework to support kernel-level and user-level threads in C++CSP. The work has shown that not only is a combined model possible, but the performance benefits give reasonable reasons to use such an approach. The key findings are:

1. To best exploit hardware, atomic communication between cores is still best if the kernel-threads can be matched to hardware.
2. Fiber communication is good for intra-core communication, and in C++CSP is still lower than common message-passing languages such as Go and Erlang by approximately 50%.
3. Fibers have low overheads when considering concurrency creation.
4. The overhead for scheduling large numbers of fibers is small.
5. Millions of fibers can be created and scheduled given the correct resources.
6. Kernel-level threads can support different groups of user-level threads to provide multicore support for M:N threading in C++CSP.

Better benchmarking of combined kernel-thread and user-thread applications is required. Furthermore, an understanding of memory requirements to support fibers should be explored. Finally, the next stage with C++CSP will be to add networked support, which could potentially support 100s of millions of processes.

References

[1] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007.

[2] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three unique implementations of processes for pycsp. In *Communicating Process Architectures 2009*, pages 277–292, 2009.

[3] 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.

[4] Kevin Chalmers. What are Communicating Process Architectures? Towards a Framework for Evaluating Message-passing Concurrency Languages. In Kevin Chalmers, Jan B. Pedersen, Jan Broenink, Kevin Vella, Brian Vinter, and Peter Welch, editors, *Communicating Process Architectures 2017*.

[5] Kevin Chalmers. Building a C++CSP Channel Using C++ Atomics: a Busy Channel Performance Analysis. In Kevin Chalmers, Jan B. Pedersen, Jan Broenink, Kevin Vella, Brian Vinter, and Peter Welch, editors, *Communicating Process Architectures 2017*.

[6] 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.