Development and Evaluation of a Modern C++CSP Library

Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

- DISCLAIMER - The real reason I've been working on this is to build an MPI layer and an algorithmic skeleton framework.
- However ...
    - Original C++CSP is a little dated, and currently does not build with a modern C++ and Boost installation.
    - C++11 provided major updates to the C++ standard, which included thread support.
    - C++ is callable from a number of languages.
    - I want a cleaner API. I don't like Java code, and JCSP suffers from Java code.

Outline

Existing CSP Inspired Libraries

- JCSP [Welch et al., 2007]
- CTJ [Broenink et al., 1999]
- JVMCSP [Shrestha and Pedersen, 2016]
- PyCSP[Vinter et al., 2009]
- CHP (Haskell) [Brown, 2008]
- JavaScript [Micallef and Vella, 2016]
- C++CSP [Brown, 2007]
- C# [Skovhede and Vinter, 2015]
- CSP (Scala)[Sufrin, 2008]

Modern C++ Standards and Design - Language Features

- Move semantics (*rvalue* references - denoted with &&)
  1. there is no reference held in the caller's scope, reducing side-effects.
  2. there is no copy created, reducing memory overhead.
- Initializer list construction
  - `vector<int> v = {1, 2, 3, 4, 5};`
- Variadic Templates

Variadic Template Example

```
template<typename T, typename... args>
void foo(T value, args... rest)
{
  cout << value;
  if (sizeof...(args) > 0)
    foo(rest);
}
```

Modern C++ Standards and Design - Language Features

- Lambda Expressions
  - `auto add = [=](int a, int b){ return a + b; };`
- Smart pointers
  - `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.

Smart Pointer Example

```
int main(int argc, char **argv)
{
  // ptr has type shared_ptr<vector<int>>.
  // Parameters captured as variadic
  auto ptr = make_shared<vector<int>>();
}
```

Modern C++ Standards and Design - Thread Support

- Thread support features
  - Threads and the associated locking mechanisms.
  - Futures.
  - Atomics.
  - A defined C++ memory model.
- Thread creation just requires the void procedure to run.

Thread Creation Example

```
void work(int x, float y, string str)
{
  // ... do some work
}
int main(int argc, char **argv)
{
  // Create thread from work function
  thread t(work, 5, 2.0f, string("test"));
  // ...
  t.join();
}
```

Locking and Communicating Between Threads

```
mutex mut;
condition_variable cv;
resource res;

void work()
{
  unique_lock<mutex> lock(mut);
  // ... work with locked resource.
  cv.wait(mut);
  // .. carry on working
  // Notify next waiting thread
  cv.notify();
  // Automatic freeing of lock on stack cleanup
}
```

Modern C++ Standards and Design - Design Principles

- PIMPL
  - Private IMPLementation or Pointer to IMPLementation
  - Class contains a private class containing actual implementation code
  - Class contains pointer to instance of the internal object
  - Reduces need for external pointers and simplifies copies
- RAII
  - Resource Acquisition Is Initialisation
  - Ties resource lifetime to object lifetime
  - If no leaks of top level objects, created inner resources will not leak

Outline

Goals

- Pointer free API (C++CSP user does not need to create objects on the free store)
- Header only library (simple drop into existing code - no pre-built libraries)
- API similar to JCSP
- API familiar to C++ programmer
- Exploit C++ features to simplify code further

Operator Overloads and Helper Patterns

- Primitives have overloads on call operator for basic behaviour.
  - `auto read = c();`
  - `c(5);`
- Channels have implicit copy constructors to grab ends.
- Common patterns are provided to simplify code (currently with an overhead)

C++CSP Helper Pattern Usage

```
par_write({a, b}, {5, 3});
auto vals = par_read({c, d, e});
vector<chan_out<int>> chans = {a, b, e};
par_for(chans.begin(), chans.end(),
        [=](chan_out<int> chan){ chan(5); });
```

Move Semantic Channels

- Channels exploit move semantics as far as possible.
- C++CSP users have the choice of copying or moving values into the channel.

Copying and Moving into Channels

```
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));
```

- Processes are functions / lambda expressions.
- An extendible `process` type exists but clunky

Process Creation with `make_proc`

```
void prefix(int value, chan_in<int> in, chan_out<int>
    out)
{
  out(value);
  while (true) out(in());
}
int main(int argc, char **argv)
{
  one2one_chan<int> a;
  one2one_chan<int> b;
  par
  {
    make_proc(prefix, 0, a, b),
    // ... other processes
  }();
}
```

Parallel List

```
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)
  }();
}
```

```
#define seq [=]()
int main(int argc, char **argv) {
  one2one_chan<int> a, b, c, d;
  par {
    seq { // prefix
      a(0);
      while (true) a(c());
    },
    seq { // delta
      while (true) {
        auto value = a();
        par_write({b, d}, {value, value});
      }
    },
    seq { // successor
      while (true) {
        auto value = b();
        c(++value);
      }
    },
    seq { // consumer
      while (true) cout << d() << endl;
    }
  }();
}
```

PHIL Definition

```
auto PHIL = [=](int i, chan_out<int> left,
chan_out<int> right, chan_out<int> down, chan_out<int>
     up)
{
  timer t;
  while (true)
  {
    report(to_string(i) + " thinking");
    t(seconds(i));
    report(to_string(i) + " hungry");
    down(i);
    report(to_string(i) + " sitting");
    par_write({left, right}, {i, i});
    report(to_string(i) + " eating");
    t(seconds(i));
    report(to_string(i) + " leaving");
    par_write({left, right}, {i, i});
    up(i);
  }
```

SECURITY Definition

```
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;
    }
  }
}
```

Process Network Definition

```
using proc = function<void()>;
one2one_chan<int> left[N], right[N];
any2one_chan<int> down, up;
vector<proc> fork(N);
for (int i = 0; i < N; ++i)
  fork[i] = make_proc(FORK, left[i], right[(i +1)%N]);
vector<proc> phil(N);
for (int i = 0; i < N; ++i)
  phil[i] = make_proc(PHIL, i, left[i], right[i], down
      , up);
par
{
  par(phil),
  par(fork),
  make_proc(SECURITY, down, up),
  printer<string>(report, "", "")
}();
```

- To evaluate the library, two benchmark approaches are taken.
  - Microbenchmarking (properties of the library)
  - Macrobenchmarking (speedup)
- Microbenchmarks compare to JCSP
  - CommsTime (channel communication time)
  - StressedAlt (selection time and process count)
- Macrobenchmarks
  - Monte Carlo π - purely computational
  - Mandelbrot - some memory communication

Microbenchmark Results - CommsTime

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

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

Macrobenchmark Results - Monte Carlo π

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

Macrobenchmark Results - Mandelbrot with Copy and Move

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

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

Outline

Conclusions

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.

Future Work

- Further benchmarking
- Investigate some other optimisations (e.g. atomics)
- Network stack development with MPI backend
- Skeletal programming support

Broenink, J. F., Bakkers, A. W. P., and Hilderink, G. H. (1999).
Communicating Threads for Java.
In Cook, B. M., editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262.

Brown, N. C. (2007).
C++CSP2: A Many-to-Many Threading.
In McEwan, A. A., Schneider, S., Ifill, W., and Welch, P. H., editors, *Communicating Process Architectures 2007*, pages 183–206.

Brown, N. C. (2008).
Communicating Haskell Processes: Composable Explicit Concurrency Using Monads.
In Welch, P. H., Stepney, S., Polack, F., Barnes, F. R. M., McEwan, A. A., Stiles, G. S., Broenink, J. F., and Sampson, A. T., editors, *Communicating Process Architectures 2008*, pages 67–83.

Micallef, K. and Vella, K. (2016).
Communicating Generators in JavaScript.
In *Communicating Process Architectures 2016*.

Shrestha, C. and Pedersen, J. B. (2016).
JVMCSP - Approaching Billions of Processes on a Single-Core jvm.
In *Communicating Process Architectures 2016*.

Skovhede, K. and Vinter, B. (2015).
CoCoL: Concurrent Communications Library.
In *Communicating Process Architectures 2015*.

Sufrin, B. (2008).
Communicating Scala Objects.
In Welch, P. H., Stepney, S., Polack, F., Barnes, F. R. M., McEwan, A. A., Stiles, G. S., Broenink, J. F., and Sampson, A. T., editors, *Communicating Process Architectures 2008*, pages 35–54.

Vinter, B., Bjrndalen, J. M., and Friborg, R. M. (2009).
PyCSP Revisited.
In Welch, P. H., Roebbers, H., Broenink, J. F., Barnes, F. R. M., Ritson, C. G., Sampson, A. T., Stiles