Development and Evaluation of a Modern C++ CSP Library

Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk
1. Background

2. Design of C++CSP

3. Experimental Results

4. Conclusions
• 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

1. Background
2. Design of C++CSP
3. Experimental Results
4. Conclusions
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

```cpp
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

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

```cpp
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();
}
```
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 waiting thread
    cv.notify();
    // Automatic freeing of lock on stack cleanup
}
• 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
• 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

```cpp
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); });
```
• 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

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

**Process Creation with make_proc**

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

```cpp
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, "", "")
}();
```
Outline

1 Background

2 Design of C++CSP

3 Experimental Results

4 Conclusions
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 $\pi$ - purely computational
- Mandelbrot - some memory communication
## Microbenchmark Results - CommsTime

<table>
<thead>
<tr>
<th>Approach</th>
<th>Channel Time</th>
<th>Estimated Context Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCSP</td>
<td>2,649</td>
<td>1,325</td>
</tr>
<tr>
<td>JCSP Seq</td>
<td>3,476</td>
<td>1,738</td>
</tr>
<tr>
<td>C++CSP</td>
<td>4,435</td>
<td>2,218</td>
</tr>
<tr>
<td>C++CSP Seq</td>
<td>1,994</td>
<td>997</td>
</tr>
<tr>
<td>C++CSP make_proc</td>
<td>4,532</td>
<td>2,266</td>
</tr>
<tr>
<td>C++CSP make_proc Seq</td>
<td>1,997</td>
<td>999</td>
</tr>
<tr>
<td>C++CSP lambda</td>
<td>4,481</td>
<td>2,241</td>
</tr>
<tr>
<td>C++CSP lambda Seq</td>
<td>2,092</td>
<td>1,046</td>
</tr>
<tr>
<td>Channels</td>
<td>JCSP Select</td>
<td>C++CSP Select</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>64</td>
<td>990</td>
<td>750</td>
</tr>
<tr>
<td>128</td>
<td>890</td>
<td>845</td>
</tr>
<tr>
<td>256</td>
<td>965</td>
<td>825</td>
</tr>
<tr>
<td>512</td>
<td>975</td>
<td>787</td>
</tr>
<tr>
<td>1,024</td>
<td>1,139</td>
<td>880</td>
</tr>
<tr>
<td>2,048</td>
<td>1,386</td>
<td>958</td>
</tr>
<tr>
<td>4,096</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
</tbody>
</table>
## Macrobenchmark Results - Monte Carlo $\pi$

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>ms</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>193.84</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>96.95</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>51.09</td>
<td>3.79</td>
</tr>
<tr>
<td>8</td>
<td>32.87</td>
<td>5.90</td>
</tr>
<tr>
<td>16</td>
<td>32.92</td>
<td>5.89</td>
</tr>
<tr>
<td>32</td>
<td>32.87</td>
<td>5.90</td>
</tr>
</tbody>
</table>
Macrobenchmark Results - Mandelbrot with Copy and Move

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

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

2 Design of C++CSP

3 Experimental Results

4 Conclusions
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
Communicating Threads for Java.

C++CSP2: A Many-to-Many Threading.

Communicating Haskell Processes: Composable Explicit Concurrency Using Monads.

Communicating Generators in JavaScript.

JVMCSP - Approaching Billions of Processes on a Single-Core jvm.

CoCoL: Concurrent Communications Library.
In Communicating Process Architectures 2015.

Communicating Scala Objects.

PyCSP Revisited.