Channels Are Useful, Not Only For Water

Prepared for Italian C++ 2018

Felix Petriconi

2018-06-23

Thanks to the sponsors!

Bloomberg®

JFrog | CONAN C/C++ package manager

JET BRAINS

aresys

AIV ACCADEMIA ITALIANA VIDEOGIOCHI

KDAB

RECOGNITION ROBOTICS
The Visual Guidance Company

sigeo
Servizi Informativi Geografici

Italian C++ Conference 2018 – June 23, Milan #itcppcon18

Felix Petriconi

- Started with C++ 1994
- Programmer and development manager since 2003 at MeVis Medical Solutions AG, Bremen, Germany
  - Development of medical devices in the area of mammography and breast cancer therapy (C++, Ruby)
- Programming activities:
  - Blog editor of ISO C++ website
  - Active member of C++ User Group Bremen
  - Contributor to stlab's concurrency library
  - Member of ACCU conference committee
- Married with Nicole, having three children, living near Bremen, Germany
- Other interests: Classic film scores, composition

The [C++] language is too large for *anyone* to master

The [C++] language is too large for *anyone* to master
So *everyone* lives within a subset

The [C++] language is too large for *anyone* to master
So *everyone* lives within a subset

Sean Parent, C++Now, 2012

Why I am here?

Why I am here?

I saw how we used different ways to delegate work to different CPU cores

I saw how easy it is to make mistakes

I saw and still see the difficulties in maintaining the code

Why I am here?

- I saw how we used different ways to delegate work to different CPU cores
- I saw how easy it is to make mistakes
- I saw and still see the difficulties in maintaining the code

- I listened 2015 to the CppCast with Sean Parent about Concurrency
- I was impressed
- I wanted to learn more

Why I am here?

I saw how we used different ways to delegate work to different CPU cores
I saw how easy it is to make mistakes
I saw and still see the difficulties in maintaining the code

I listened 2015 to the CppCast with Sean Parent about Concurrency
I was impressed
I wanted to learn more

I started collaborating in his open source project for a new concurrency library
I'm having fun in learning there a lot
I care about sharing my knowledge

Why are you here?

Why do we have to talk about concurrency?

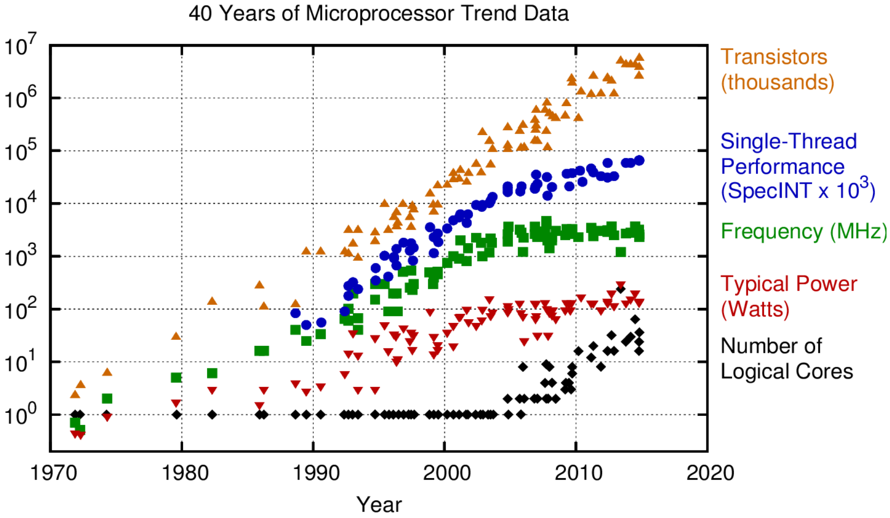
The free lunch is over!

The free lunch is over!

Herb Sutter, 2005[1]

[1] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
`http://www.gotw.ca/publications/concurrency-ddj.htm`

The free lunch is over



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Amdahl's Law

Amdahl's Law[2]

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

S : Speed up

P : Synchronization $[0-1]$

N : Number of Cores

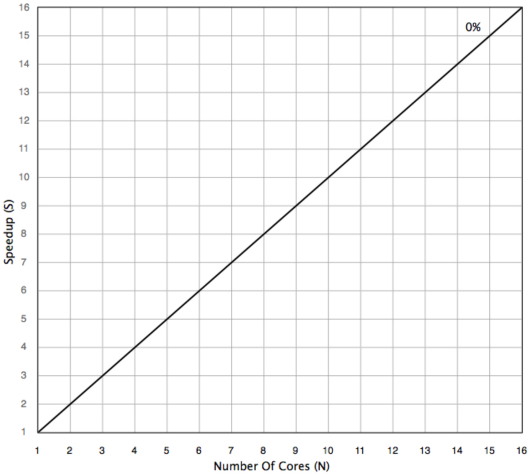
[2] Presented 1967 by Gene Myron Amdahl (1922-2015)

Amdahl's Law

0% Synchronization

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

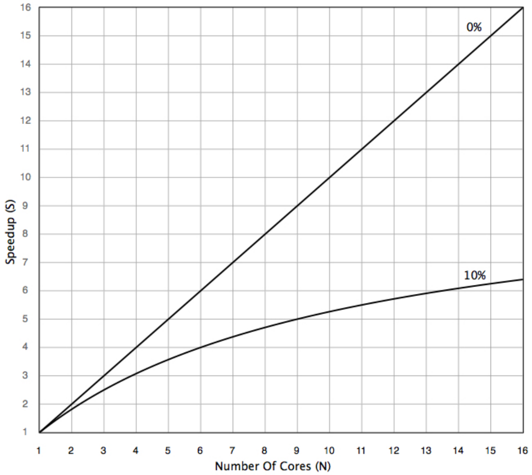
$$P = 0$$



Amdahl's Law

10% Synchronization

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0.1$$

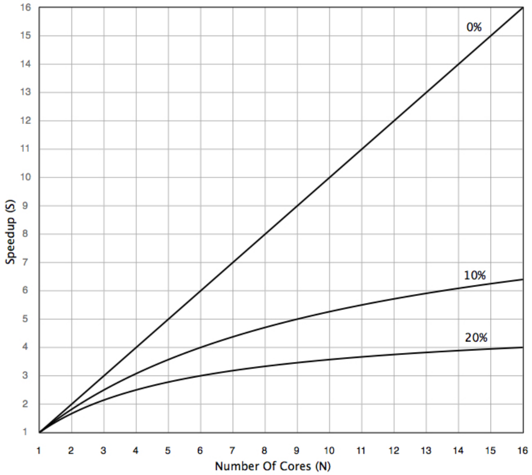


Amdahl's Law

20% Synchronization

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

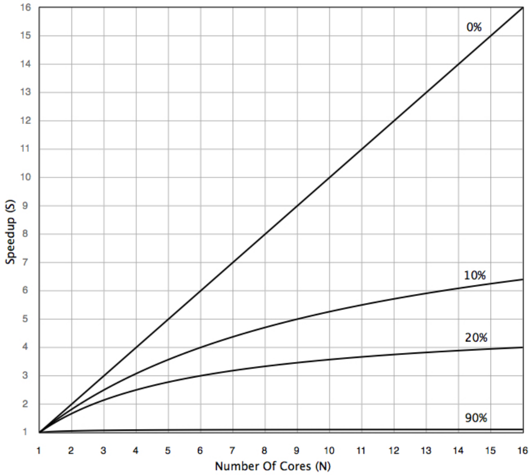
$$P = 0.2$$



Amdahl's Law

90% Synchronization

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$
$$P = 0.9$$



How to use multiple cores?

How to use multiple cores?

- Individual single threaded processes

How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization

How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization
- Multi threaded process with synchronization

How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization
- Multi threaded process with synchronization

$$\left.\begin{array}{l}\blacktriangleright\ \text{Mutex}\\ \blacktriangleright\ \text{Atomic}\\ \blacktriangleright\ \text{Semaphore}\\ \blacktriangleright\ \text{Memory Fence}\\ \blacktriangleright\ \text{Transactional Memory}\end{array}\right\}\ \text{Low level synchronization primitives}$$

How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization
- Multi threaded process with synchronization
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Transactional Memory

  Low level synchronization primitives
- Multi threaded process with higher level abstractions

How to use multiple cores?

- Individual single threaded processes
- Multi threaded process without synchronization
- Multi threaded process with synchronization
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Transactional Memory

  } Low level synchronization primitives
- Multi threaded process with higher level abstractions
  - Future
  - Channel
  - Actor
  - ...

Future Introduction

Future Introduction

- Futures provide a mechanism to separate a function $f(\ldots)$ from its result r

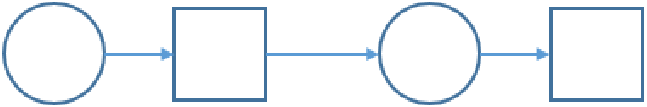
Future Introduction

- Futures provide a mechanism to separate a function $f(...)$ from its result r
- After the function is called the result appears "magically" later in the future

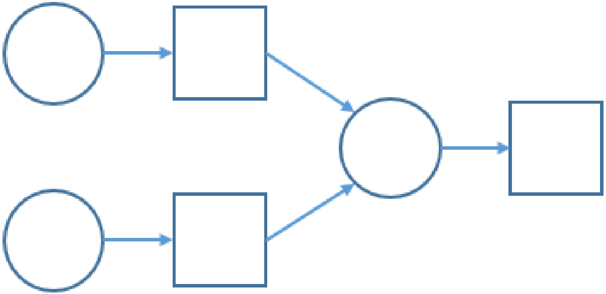
Future Introduction

- Futures provide a mechanism to separate a function $f(...)$ from its result r
- After the function is called the result appears "magically" later in the future
- Futures, resp. promises where invented 1977/1978 by Daniel P. Friedman, David Wise, Henry Baker and Carl Hewitt

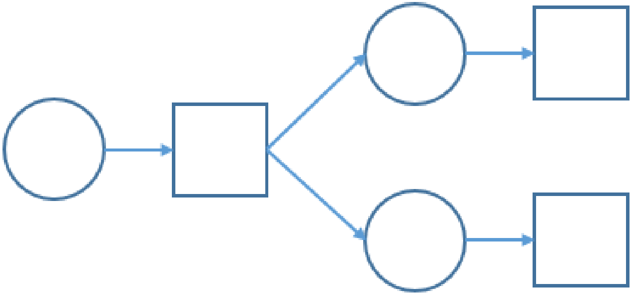
Future Introduction - Continuation



Future Introduction - When All / When Any



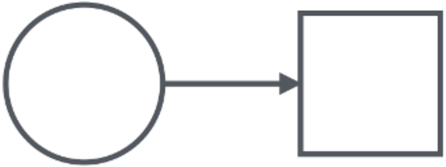
Future Introduction - Split



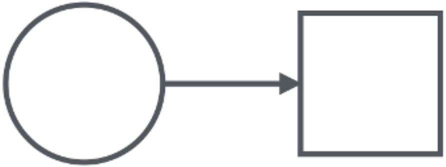
See my talks at Meeting C++ 2017 and ACCU 2018:

- `https://www.youtube.com/watch?v=L63XGqiNuhI`
- `https://www.youtube.com/watch?v=vDmQlIeY4z0`

Channel Introduction

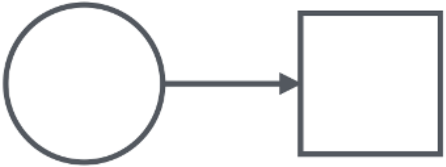


Channel Introduction



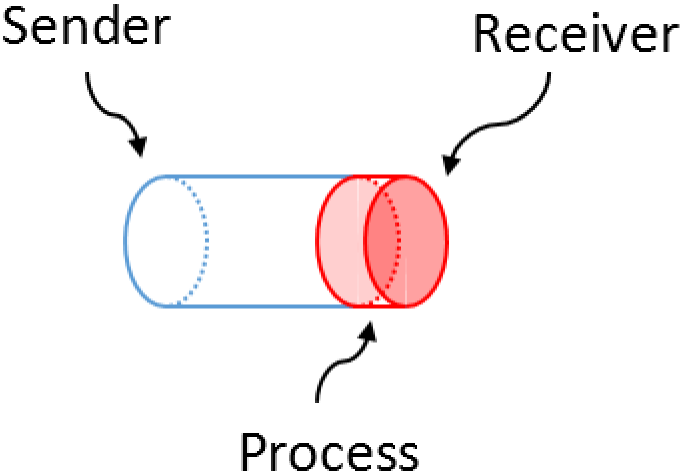
- Channels allow the creation of persistent execution graphs

Channel Introduction



- Channels allow the creation of persistent execution graphs
- First published by Tony Hoare 1978

Channel Introduction



Channel - Stateless Process

```cpp
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;         // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =       // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;         // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;        // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```cpp
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;         // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =       // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;         // sending part of the channel
  stlab::receiver<int> receiver;   // receiving part of the channel
  std::tie(send, receiver) =       // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;          // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =      // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;           // attaching process to the receiving
                                  // part
  receiver.set_ready();           // no more processes will be attached
                                  // process starts to work
  send(1); send(2); send(3);      // start sending into the channel

  int end; std::cin >> end;       // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;          // sending part of the channel
  stlab::receiver<int> receiver;    // receiving part of the channel
  std::tie(send, receiver) =        // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;             // attaching process to the receiving
                                    // part
  receiver.set_ready();             // no more processes will be attached
                                    // process starts to work
  send(1); send(2); send(3);        // start sending into the channel

  int end; std::cin >> end;         // simply wait to end application
}
```

Channel - Stateless Process

```
#include <iostream>
#include <stlab/concurrency/channel.hpp>
#include <stlab/concurrency/default_executor.hpp>

int main() {
  stlab::sender<int> send;         // sending part of the channel
  stlab::receiver<int> receiver; // receiving part of the channel
  std::tie(send, receiver) =       // combining both to a channel
    stlab::channel<int>(stlab::default_executor);

  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Channel - Stateless Process cont.

```
int main() {
  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;              // attaching process to the receiving
                                     // part
  receiver.set_ready();              // no more processes will be attached
                                     // process starts to work
  send(1); send(2); send(3);         // start sending into the channel

  int end; std::cin >> end;          // simply wait to end application
}
```

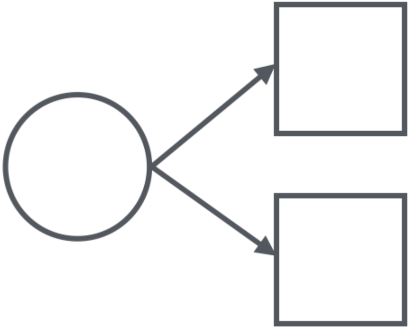
Channel - Stateless Process cont.

```cpp
int main() {
  auto printer =
    [](int x){ std::cout << x << '\n'; }; // stateless process

  auto printer_process =
    receiver | printer;            // attaching process to the receiving
                                   // part
  receiver.set_ready();            // no more processes will be attached
                                   // process starts to work
  send(1); send(2); send(3);       // start sending into the channel

  int end; std::cin >> end;        // simply wait to end application
}
```

Output

```
1
2
3
```

Channel - Split



New edges are concatenated with the `operator|()` on the same receiver

Channel - Split Process

```
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();           // no more processes will be attached
                                  // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```cpp
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```cpp
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Split Process

```
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();            // no more processes will be attached
                                  // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

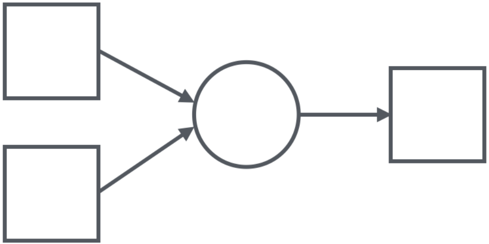
Channel - Split Process

```cpp
int main() {
  auto [send, receive] = channel<int>(default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receive | printerA;
  auto printer_processB = receive | printerB;

  receive.set_ready();           // no more processes will be attached
                                  // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Output

```
Process A 1
Process B 1
Process A 2
Process B 2
Process B 3
```

Channel - Merge



- `join()` The downstream process is invoked when all arguments are ready.
- `zip()` The downstream process is invoked in round robin manner with the incoming values.
- `merge()` The downstream process is invoked unordered with the next value that comes from upstream.

Channel - Merged Processes

```
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Channel - Merged Processes

```
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Channel - Merged Processes

```cpp
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Channel - Merged Processes

```
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Channel - Merged Processes

```cpp
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Channel - Merged Processes

```
using namespace stlab;

int main() {
  auto [sendA, receiverA] = channel<int>(default_executor);
  auto [sendB, receiverB] = channel<int>(default_executor);

  auto printer = [](int x, int y){ printf("Process %d %d\n", x, y); };

  auto printProcess = join(default_executor, printer,
    receiverA, receiverB);

  receiverA.set_ready(); receiverB.set_ready();

  sendA(1); sendA(2); sendB(3); sendA(4); sendB(5); sendB(6);
  int end; std::cin >> end;
}
```

Output

```
Process 1 3
Process 2 5
Process 4 6
```

Additional channel options

Additional channel options

- With `buffer_size`$\{n\}$ within the concatenation it is possible to limit the incoming queue to size `n`

Additional channel options

- With `buffer_size`$\{n\}$ within the concatenation it is possible to limit the incoming queue to size `n`
- With `executor`$\{ex\}$ within the concatenation it is possible to specify a different executor for the following process(es)

Channel - With Options

```cpp
int main() {
  auto [send, receiver] = channel<int>(stlab::default_executor); // C
    ++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver |
    (stlab::executor{ stlab::immediate_executor } & printerB);

  receiver.set_ready();            // no more processes will be attached
                                   // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - With Options

```cpp
int main() {
  auto [send, receiver] = channel<int>(stlab::default_executor); // C++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver |
    (stlab::executor{ stlab::immediate_executor } & printerB);

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - With Options

```
int main() {
  auto [send, receiver] = channel<int>(stlab::default_executor); // C
    ++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver |
    (stlab::executor{ stlab::immediate_executor } & printerB);

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - With Options

```
int main() {
  auto [send, receiver] = channel<int>(stlab::default_executor); // C
    ++17

  auto printerA = [](int x){ printf("Process A %d\n", x); };
  auto printerB = [](int x){ printf("Process B %d\n", x); };

  auto printer_processA = receiver | printerA;
  auto printer_processB = receiver |
    (stlab::executor{ stlab::immediate_executor } & printerB);

  receiver.set_ready();          // no more processes will be attached
                                 // process may start to work
  send(1); send(2); send(3);
  int end; std::cin >> end;
}
```

Channel - Stateful Process - Motivation

Channel - Stateful Process - Motivation

- Some problems need a processor with state

Channel - Stateful Process - Motivation

- Some problems need a processor with state
- Some problems have an $n : m$ relationship from input to output

Channel - Stateful Process - Motivation

- Some problems need a processor with state
- Some problems have an $n : m$ relationship from input to output
- The picture becomes more complicated with states:
  - When to proceed?
  - How to handle situations when less than expected values come from upstream?

Channel - Stateful Process Signature

```cpp
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                        // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - await

```cpp
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                          // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - yield

```cpp
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                      // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - state

```cpp
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                        // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - close

```
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                         // optional

    void set_error(std::exception_ptr); // optional
};
```

Stateful Process Signature - set_error

```
#include <stlab/concurrency/channel.hpp>

enum class process_state { await, yield };

using process_state_scheduled =
  std::pair<process_state, std::chrono::steady_clock::time_point>;

struct process_signature
{
    void await(T... val);

    U yield();

    process_state_scheduled state() const;

    void close();                    // optional

    void set_error(std::exception_ptr); // optional
};
```

Channel - Stateful Process Example

```
struct adder
{
};

int main() {
  auto [send, receiver] = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
struct adder
{
};

int main() {
  auto [send, receiver] = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
struct adder
{
};

int main() {
  auto [send, receiver] = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
struct adder
{
};

int main() {
  auto [send, receiver] = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example

```cpp
struct adder
{
};

int main() {
  auto [send, receiver] = channel<int>(default_executor);

  auto calculator = receiver | adder{} |
    [](int x) { std::cout << x << '\n'; };

  receiver.set_ready();

  while (true) {
    int x;
    std::cin >> x;
    send(x);
  }
}
```

Channel - Stateful Process Example cont.

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

  auto state() const { return _state; }
};
```

Channel - Stateful Process Example cont.

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

  auto state() const { return _state; }
};
```

Channel - Stateful Process Example cont.

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

  auto state() const { return _state; }
};
```

Channel - Stateful Process Example cont.

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

  auto state() const { return _state; }
};
```

Channel - Stateful Process Example cont.

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

  auto state() const { return _state; }
};
```

Channel - Timed Stateful Process

Channel - Timed Stateful Process

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    } else {
      _state.first = process_state::await;
      _state.second = std::chrono::steady_clock::now() +
        std::chrono::seconds(15);
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

```

Channel - Timed Stateful Process


```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    } else {
      _state.first = process_state::await;
      _state.second = std::chrono::steady_clock::now() +
        std::chrono::seconds(15);
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

```

Channel - Timed Stateful Process

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    } else {
      _state.first = process_state::await;
      _state.second = std::chrono::steady_clock::now() +
        std::chrono::seconds(15);
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

```

Channel - Timed Stateful Process

```
struct adder
{
  int _sum = 0;
  process_state_scheduled _state = await_forever;

  void await(int x) {
    _sum += x;
    if (x == 0) {
      _state = yield_immediate;
    } else {
      _state.first = process_state::await;
      _state.second = std::chrono::steady_clock::now() +
        std::chrono::seconds(15);
    }
  }

  int yield() {
    int result = _sum;
    _sum = 0;
    _state = await_forever;
    return result;
  }

```

Channel - Conclusion

Channels close the gap of multiple invocations where futures allow just one.

With splits and the different kind of joins it is possible to build graphs of execution.

Take Away

Use high level abstractions like futures, channels or others (actors, etc.) to distribute work on available CPU cores.

Take Away

Use high level abstractions like futures, channels or others (actors, etc.) to distribute work on available CPU cores.

Use thread pools from your operating system! Use highly optimized task stealing custom thread pools in case that the operating system does not provide one!

Take Away

Use high level abstractions like futures, channels or others (actors, etc.) to distribute work on available CPU cores.

Use thread pools from your operating system! Use highly optimized task stealing custom thread pools in case that the operating system does not provide one!

Design your application with the mindset that it can run dead-lock free on an 1-n core hardware!

Take Away

Use high level abstractions like futures, channels or others (actors, etc.) to distribute work on available CPU cores.

Use thread pools from your operating system! Use highly optimized task stealing custom thread pools in case that the operating system does not provide one!

Design your application with the mindset that it can run dead-lock free on an 1-n core hardware!

Don't let your application code be soaked with threads, mutex' and atomics.

Acknowledgement

- My family, who supports me in my work on the concurrency library and this conference.
- Sean Parent, who taught me over time lots about concurrency and abstraction. He gave me the permission to use whatever I needed from his presentations for my own.
- All contributors to the stlab library.

Reference

- Concurrency library `https://github.com/stlab/libraries`
- Documentation `http://stlab.cc/libraries`
- Communicating Sequential Processes by C. A. R. Hoare `http://usingcsp.com/cspbook.pdf`
- The Theory and Practice of Concurrency by A.W. Roscoe `http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`
- Towards a Good Future, C++ Standard Proposal by Felix Petriconi, David Sankel and Sean Parent `http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0676r0.pdf`
- A Unified Futures Proposal for C++ by Bryce Adelstein Lelbach et.al. `http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1054r0.html`

Further reading I

Software Principles and Algorithms

- Elements of Programming by Alexander Stepanov, Paul McJones, Addison Wesley
- From Mathematics to Generic Programming by Alexander Stepanov, Daniel Rose, Addison Wesley

Further reading II

Concurrency and Parallelism

- HPX `http://stellar-group.org/libraries/hpx/`
- C++CSP `https://www.cs.kent.ac.uk/projects/ofa/c++csp`
- CAF_C++ Actor Framework `http://actor-framework.org/`
- C++ Concurrency In Action by Anthony Williams, Manning

Further listening and viewing

- Goals for better code by Sean Parent:
  http://sean-parent.stlab.cc/papers-and-presentations
- Goals for better code by Sean Parent: Concurrency:
  https://youtu.be/au0xX4h8SCI?t=16354
- Future Ruminations by Sean Parent http://sean-parent.stlab.cc/2017/07/10/future-ruminations.html
- CppCast with Sean Parent http://cppcast.com/2015/06/sean-parent/
- Thinking Outside the Synchronization Quadrant by Kevlin Henney:
  https://vimeo.com/205806162

stlab Futures

stlab::future

Source: `https://github.com/stlab/libraries`

Documentation: `http://www.stlab.cc/libraries`

Thank's for your attention!

- Mail: felix@petriconi.net
- GitHub: `https://github.com/FelixPetriconi`
- Web: `https://petriconi.net`
- Twitter: @FelixPetriconi

Q & A

- Mail: felix@petriconi.net
- GitHub: `https://github.com/FelixPetriconi`
- Web: `https://petriconi.net`
- Twitter: @FelixPetriconi

Q & A

- Mail: felix@petriconi.net
- GitHub: `https://github.com/FelixPetriconi`
- Web: `https://petriconi.net`
- Twitter: @FelixPetriconi

Feedback is always welcome!