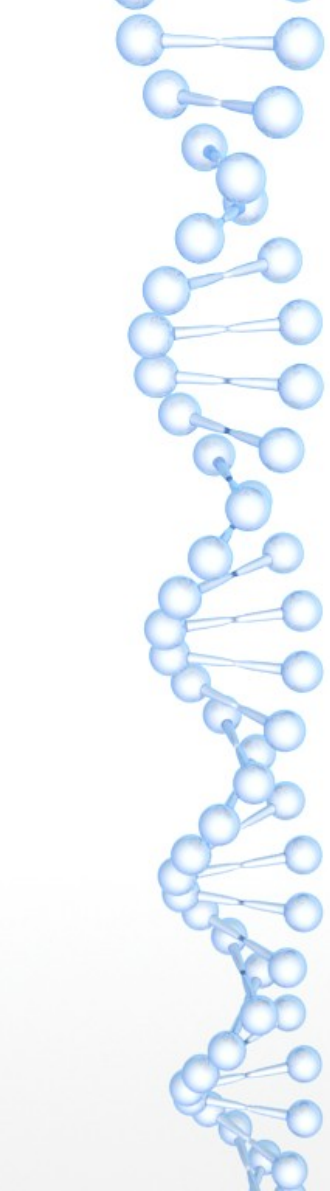
From Problem to Coroutine: Reducing I/O Latency

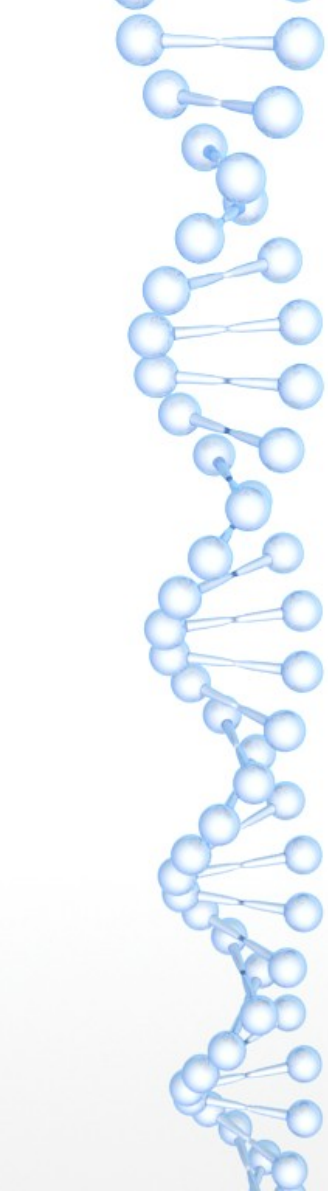
CHEINAN MARKS



About This Talk

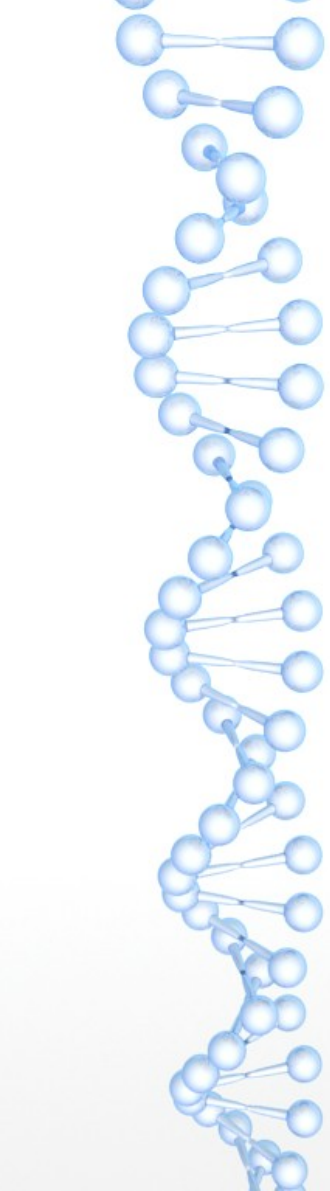
- Francesco Zoffoli: https://github.com/MakersF/cppcon-2021-corobatch
- 2016-2019 CppCon Talks by Gor Nishanov and James McNellis
- Lewis Baker’s Blogs: https://lewissbaker.github.io/
- N4868 C++ 20 Draft: https://github.com/cplusplus/draft/releases/download/n4868/n4867.html

- cheinan@pobox.com cyliber.org



I Had a Simple Program to Write

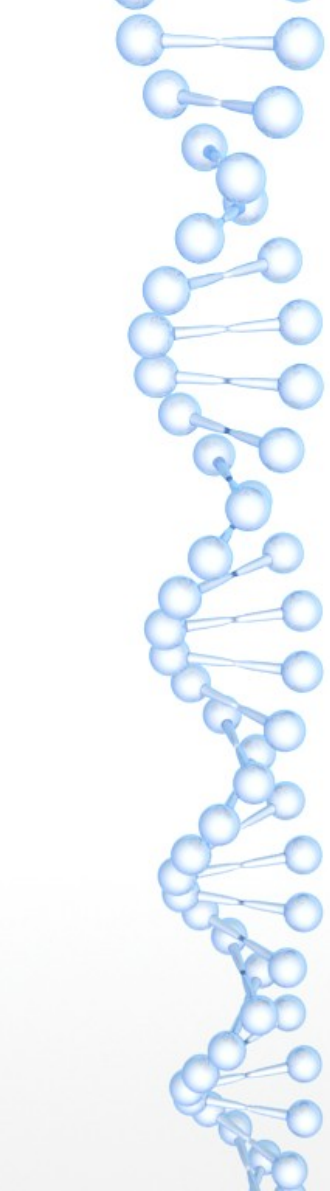
- Gather DNA Reads from FASTQ
- Count Instances of kmers



I Had a Simple Program to Write

- Gather DNA reads from FASTQ
- Count instances of kmers

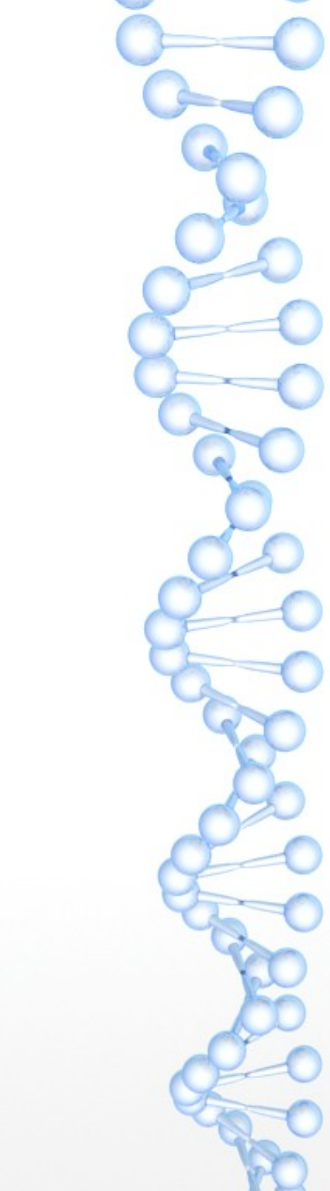
- Simple(?) Line-Based ASCII File in 4 line records
- DNA Read is second line
- Read is ASCII string made of A, C, G, T, N



I Had a Simple Program to Write

- Gather DNA reads from FASTQ
- Count instances of kmers

- Polymer – multiple repetition of single molecule
- E.g. polyethylene, our chromosomes
- Monomer – polymer building block
- Dimer – pair of monomers
- Kmer – k (arbitrary integer) number of monomers
- E.g. AATGGGC for k = 7



I Had a Simple Program to Write

- Gather DNA Reads from FASTQ
- Count Instances of kmers

- Substring search
- Count instances of “AT” in “GATGATC”

I Had a Simple Program to Write

- Gather DNA Reads from FASTQ
- Count Instances of kmers

- Substring search
- Count instances of “AT” in “GATGATC”

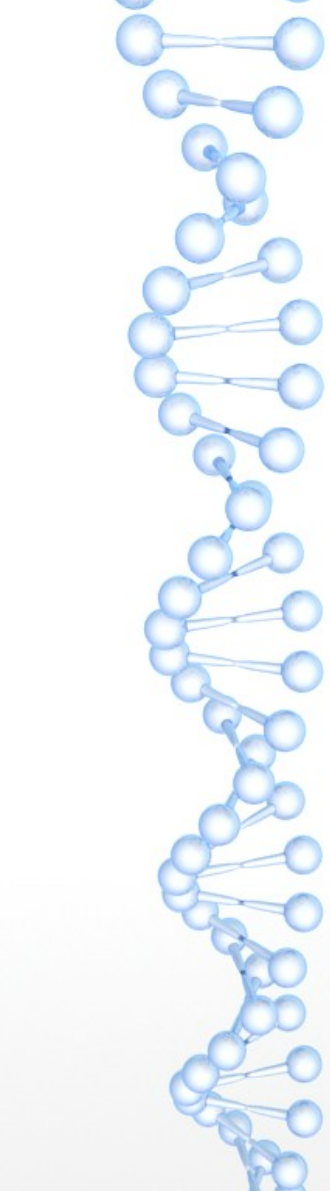
- Answer: 2

I Had a Simple Program to Write

```
while(stream.ignore(maxSize, '\n')) {
  std::getline(stream, line);
  kmerCount += CountKmer(line, kmer);
  stream.ignore(maxSize, '\n');
  stream.ignore(maxSize, '\n');
}

return kmerCount;
```

I Had a Simple Program to Write

```
std::string_view::size_type kmerPos = 0;
While ((
   kmerPos =
     line.find(kmer, kmerPos)) != line.npos) {
   kmerCount++;
   kmerPos++;
}

return kmerCount;
```

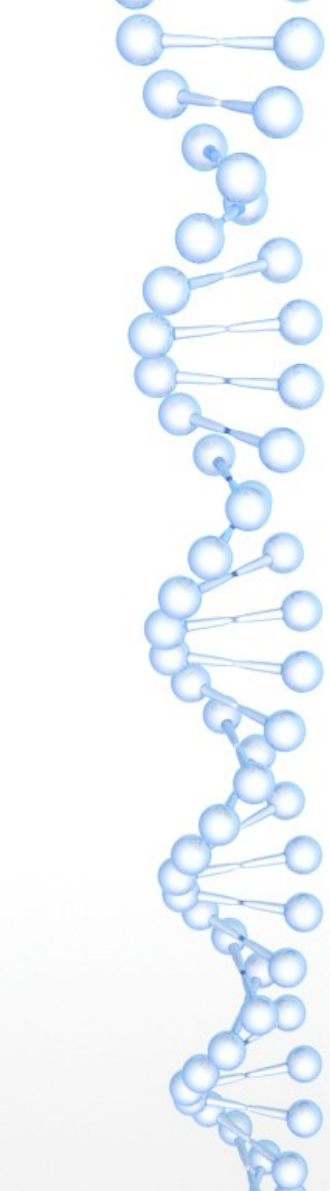


I Had a Simple Program to Write

- Gather DNA Reads from FASTQ
  - Read lines from a file
  - Extract a portion of a record
  - I/O
- Count Instances of kmers
  - Compute on the record

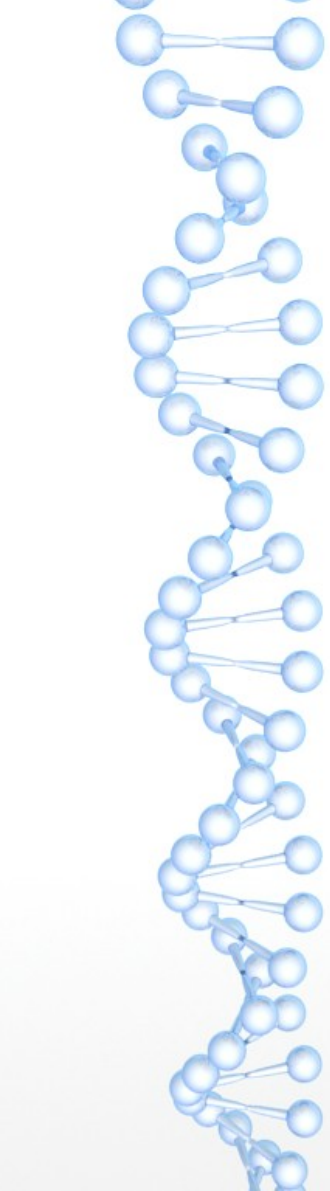
I Had a Simple Program to Write

- Gather DNA Reads from FASTQ
  - Read lines from a file
  - Extract a portion of a record
  - I/O
- Count Instances of kmers
  - Compute on the extracted data
- Producer – Consumer threads



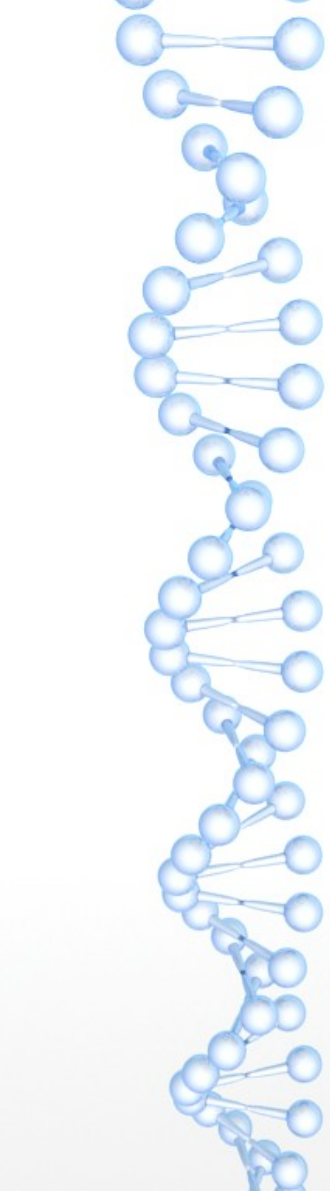
I Had a Simple Program to Write

- Gather DNA Reads from FASTQ
  - Read lines from a file
  - Extract a portion of a record
  - I/O
- Count Instances of kmers
  - Compute on the extracted data
- Producer – Consumer threads
- Zzzzzz



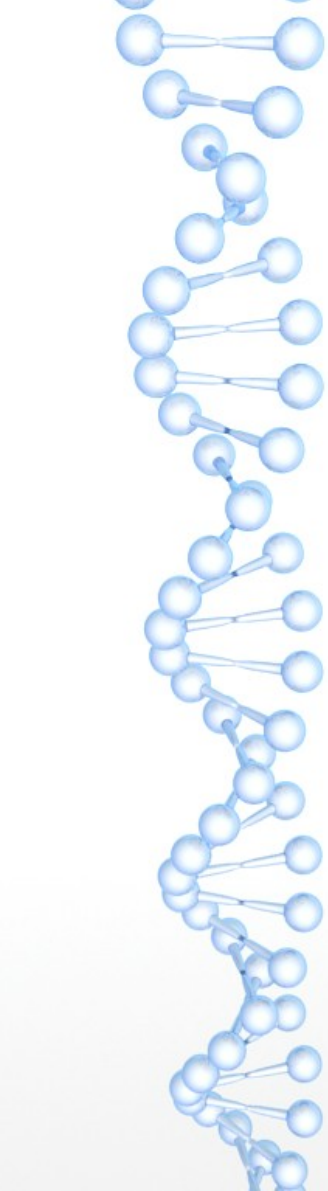
I Had a Great Idea

- Use coroutines to reduce latency



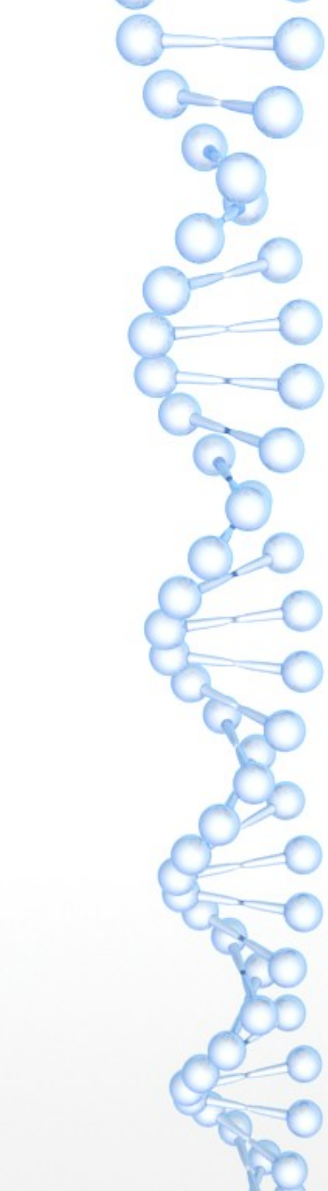
I Had a Great Idea

- Use coroutines to reduce latency
- It’s easy, if convoluted
  - Add `co_await` keyword to async parts
  - Add a bunch of boilerplate code
- Enjoy Speedup
  - And get practice with a new C++20 feature



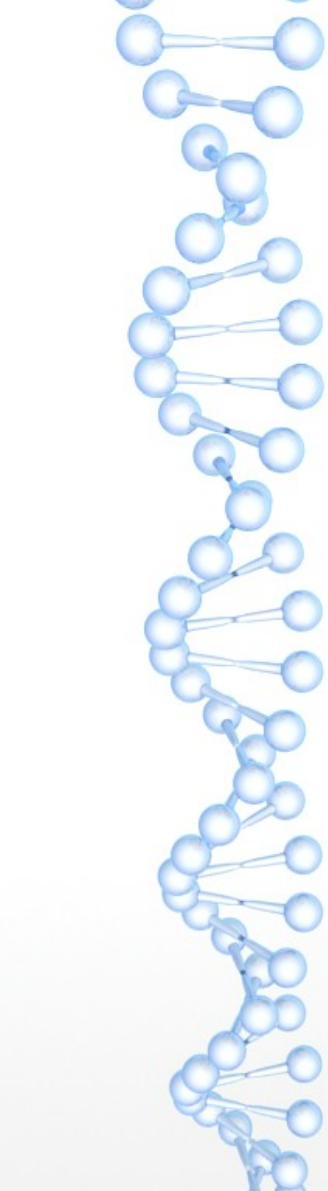
Expectation

```
std::string read = GetRead(stream);
int kmerCount = CountKmers(read);
```



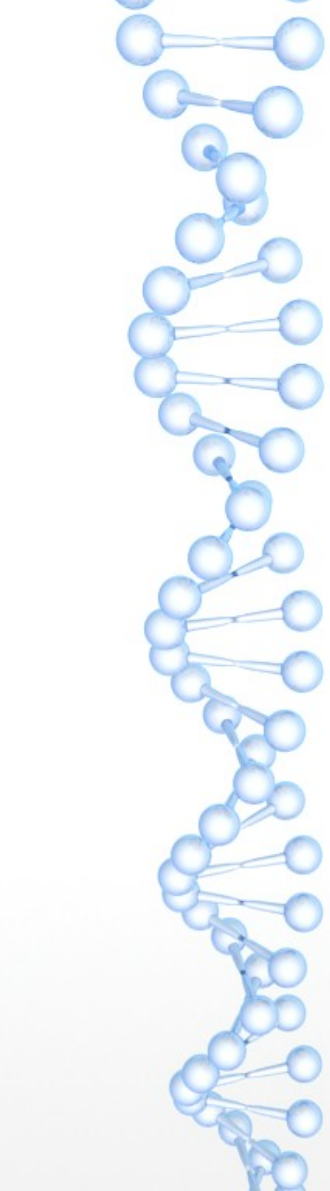
Expectation

```
std::string read = co_await GetRead(stream);
int kmerCount = co_await CountKmers(read);
```



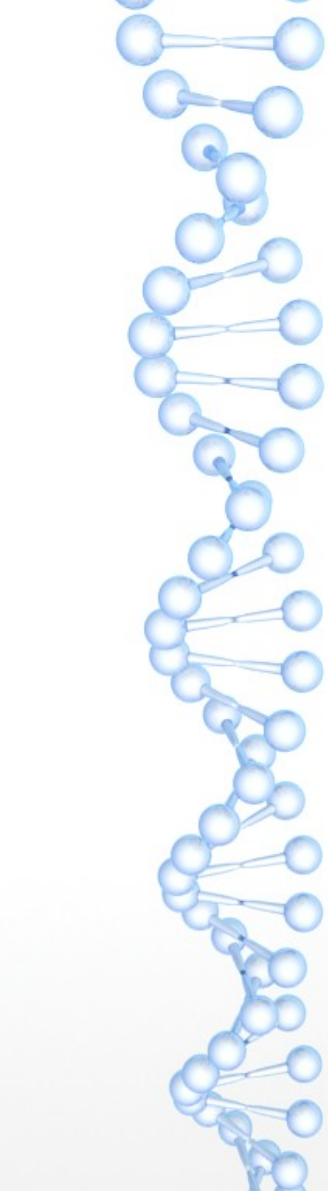
Expectation

```
std::string read = co_await GetRead(stream);
int kmerCount = co_await CountKmers(read);

// Everything magically runs faster!
```



Learnings

- Consider not using coroutines
- Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end

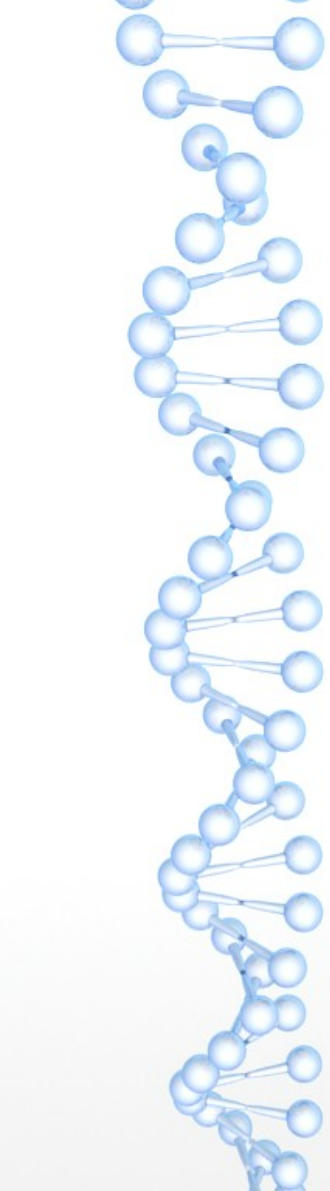


Consider Not Using Coroutines

The facilities the C++ Coroutines TS provides in the language can be thought of as a low-level assembly-language for coroutines. These facilities can be difficult to use directly in a safe way and are mainly intended to be used by library-writers to build higher-level abstractions that application developers can work with safely.

– Lewis Baker

https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await

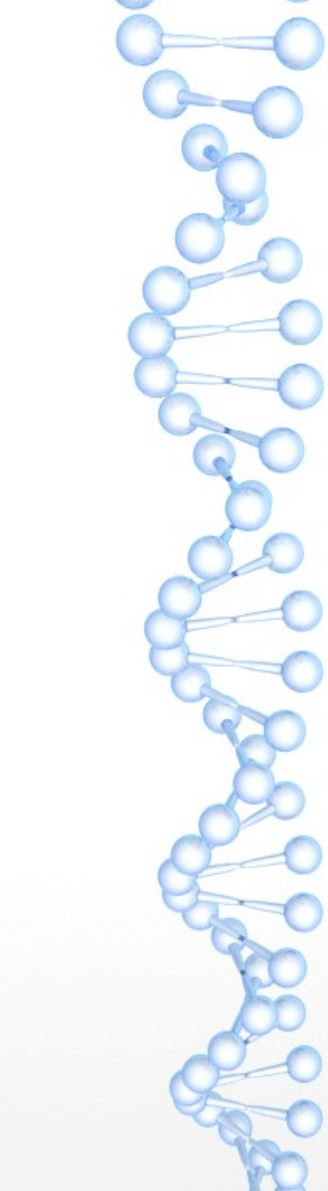


Learnings

- Consider not using coroutines
- Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end

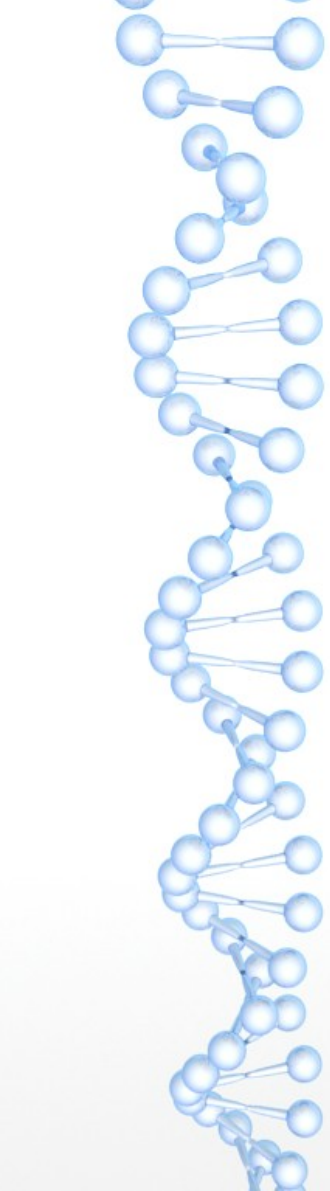
Learnings

- Consider not using coroutines
- Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end



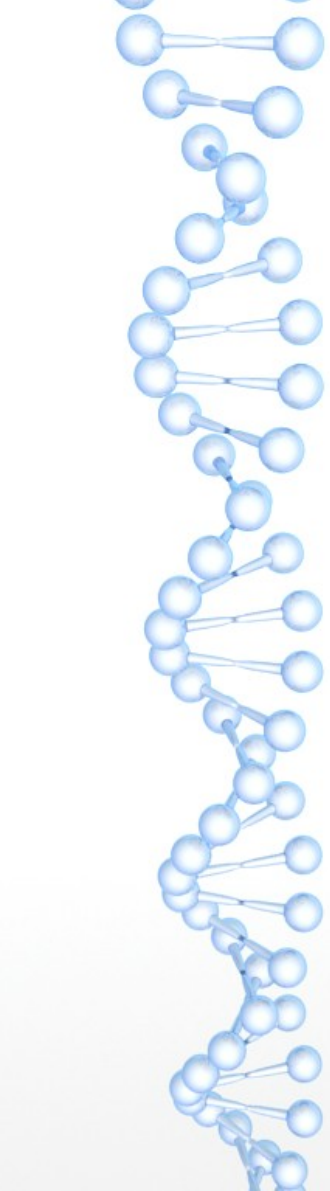
Plan out the use of the co_* keywords first

- What am I doing?
- Where can I use coroutines?
  - Should I use coroutines?



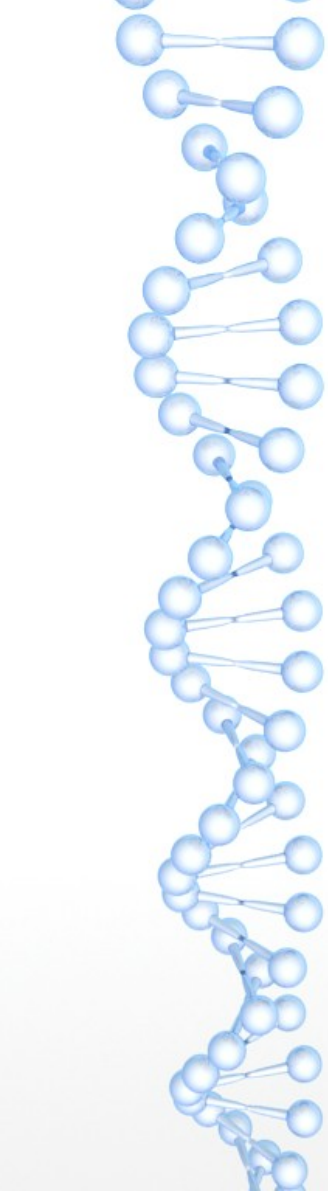
Plan out the use of the co_* keywords first

- What am I doing?
- Where can I use coroutines?
  - Should I use coroutines?
- Reading I/O in unstructured records
  - Can’t calculate record location
- Compute on data received from I/O



Plan out the use of the co_* keywords first

- What am I doing?
- Where can I use coroutines?
  - What do the coroutines do?
  - Should I use coroutines?
- Reading I/O in unstructured records
  - Can’t calculate record location
- Compute on data received from I/O



Where Can I Use Coroutines

- Offload compute to a worker thread

Where Can I Use Coroutines

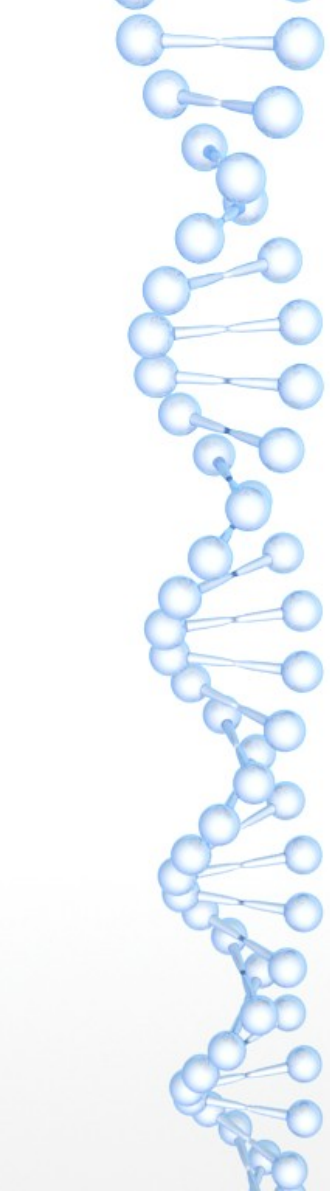
- Offload compute to a worker thread

```
CountKmersTask GetReadAndCount(…) {
  auto read = GetRead();
  co_await ResumeInComputePool();
  kmerCount = CountKmersInRead(read, kmer);

  co_return kmerCount;
}
```

Where Can I Use Coroutines

```
CountKmersTask GetReadAndCount(…) {
  auto read = GetRead();
  co_await ResumeInComputePool();
  kmerCount = CountKmersInRead(read, kmer);

  co_return kmerCount;
}
```

- Transfer



Transfer

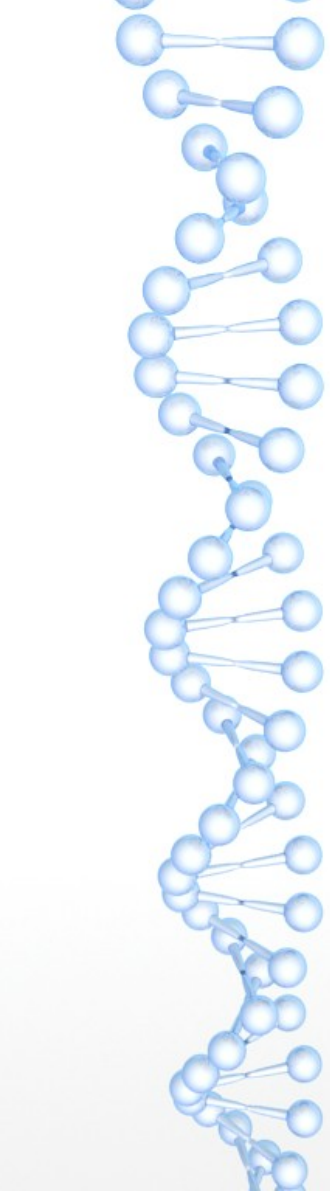
- Coroutines are *not* functions (subroutines)
- They exist independently of a thread or stack

- Can start one one thread and end on another
- No synchronization needed
- Can even transfer to another coroutine
  - State machines

```
co_await ResumeInComputePool();
```

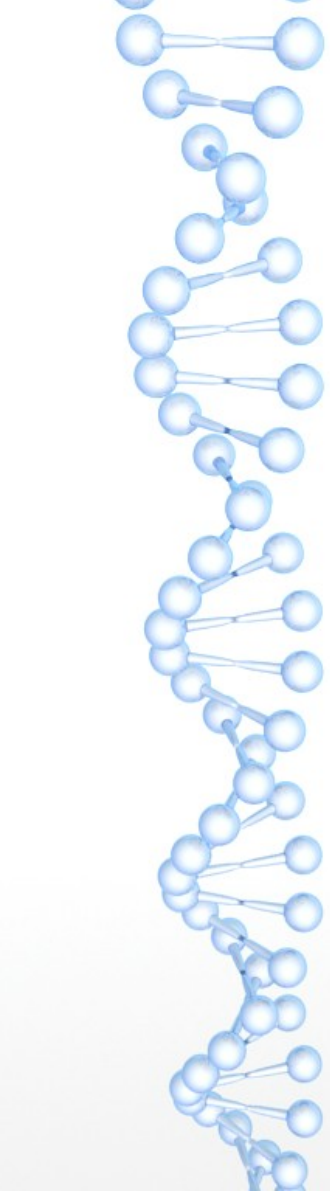
Transfer

```
struct ResumeInComputePool {
  bool await_ready() { return false; }
  void await_suspend(std::coroutine_handle<> h)
  {
    pool.execute([](auto h) {h.resume();}, h);
  }
  void await_resume() {}
};
```



Plan out the use of the co_* keywords first

- What am I doing?
- Where can I use coroutines?
  - Should I use coroutines?
- Reading I/O in unstructured records
  - Can’t calculate record location
- Compute on data received from I/O



Where Can I Use Coroutines?

- Transform unstructured file records into DNA reads
- Extract data from file records

- Generator

Generator

```
FastqGenerator GetARead(
  const std::filesystem::path &fastqPath) {
  std::ifstream fastqStream(fastqPath);
  std::string read;

  while (fastqStream) {
    fastqStream.ignore(…);
    std::getline(fastqStream, read);
    co_yield std::move(read);
    fastqStream.ignore(…);
    fastqStream.ignore(…);
  }
}
```

Generator Boilerplate

```
struct FastqGen {
  FastqGen(const FastqGen &) = delete;
  FastqGen &operator=(const FastqGen &) = delete;

  FastqGen(FastqGen &&rhs) noexcept : h_(rhs.h_) { rhs.h_ = {}; }
  FastqGen &operator=(FastqGen &&rhs) noexcept {
    if (this != &rhs) {
      if (h_) {
        h_.destroy();
      }
      h_ = rhs.h_;
      rhs.h_ = {};
    }
    return *this;
  }
```

Generator Boilerplate

```
struct FastqGenerator {
  ~FastqGenerator() {
    if (h_) {
      h_.destroy();
    }
  }
private:
  FastqGenerator(handle h) : h_(h) {}
  handle h_;
};
```

Generator Boilerplate

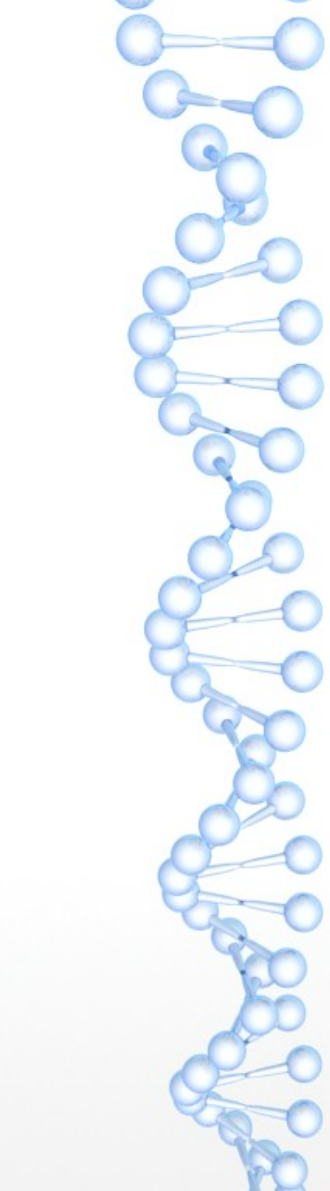
```
struct FastqGenerator {
  struct promise_type {
     auto yield_value(std::string read) {
      read_ = std::move(read);
      return std::suspend_always{};
    }

    std::string read_;
  };
};
```

Generator Boilerplate

```
struct FastqGenerator {
  bool move_next() {
    if (h_ && !h_.done()) {
      h_.resume();
      return !h_.done();
    }
    return false;
  }
  std::string current_value() {
    return std::move(h_.promise().read_);
  }
};
```

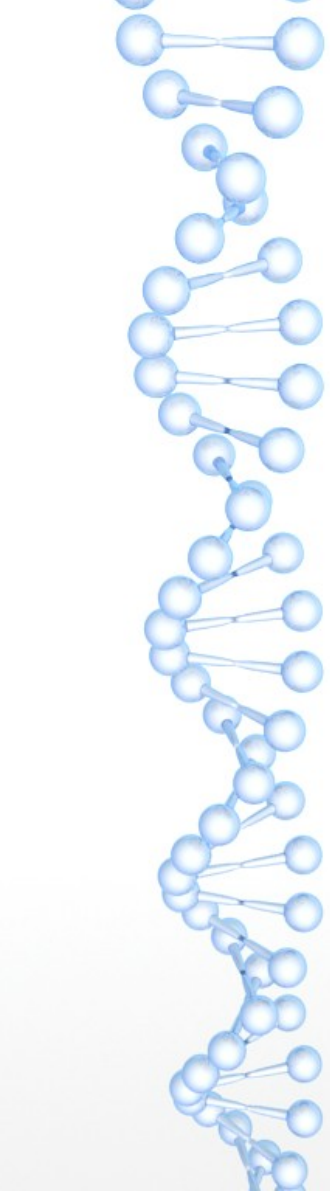
Generator Boilerplate Bonus

```
class FastqIter {
public:
  FastqIter &operator++() {
    coroutine_.resume();
    return *this;
  }

  const std::string &operator*() const { return coroutine_.promise().read_; }

  bool operator==(std::default_sentinel_t) const {
    return !coroutine_ || coroutine_.done();
  }

  explicit FastqIter(const handle coroutine) : coroutine_{coroutine} {}

private:
  handle coroutine_;
};

FastqIter begin() {
  if (h_) {
    h_.resume();
  }
  return FastqIter{h_};
}
std::default_sentinel_t end() { return {}; }
```



Learnings

- Consider not using coroutines
- Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end



The work goes in the coroutine body

- A coroutine returns a class called a task or a job
- Truly async work (that notifies) can go in awaiter
  - Work in await_suspend
  - Notification calls resume (possibly on other thread)
- Needed sync work goes in coroutine

The work goes in the coroutine body

```
CountKmersTask GetReadAndCount(…) {
  auto kmerCount = 0;

  if (!isCancelled && readGenerator.move_next()) {
    read = readGenerator.current_value();
    if (read.empty()) {
      co_return 0;
    }
    … // Compute transfer and work
  }
}
```

The work goes in the coroutine body

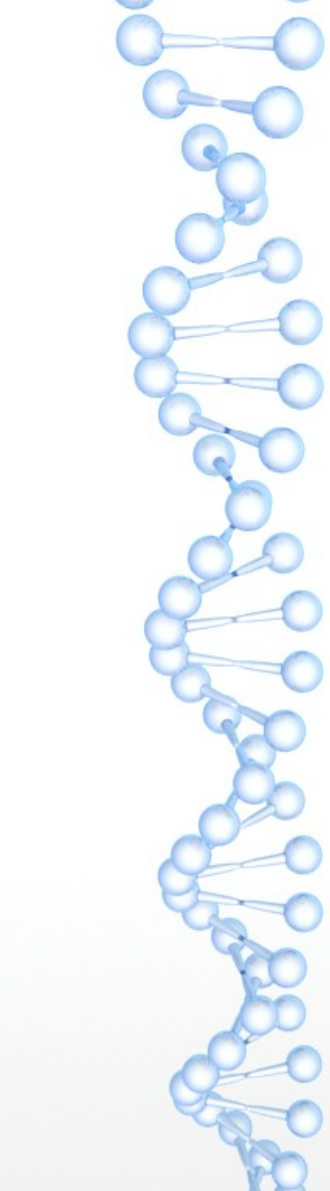
```
CountKmersTask GetReadAndCount(…) {
  std::string read;
  // co_await ResumeInIOThread{};
  if (!isCancelled && readGenerator.move_next()) {
    read = readGenerator.current_value();
    if (read.empty()) {
      co_return 0;
    }
    … // Compute transfer and work
  }
}
```

The work goes in the coroutine body

```
CountKmersTask GetReadAndCount(…, read) {
  co_await ResumeInComputeThread{};
  …
}
```

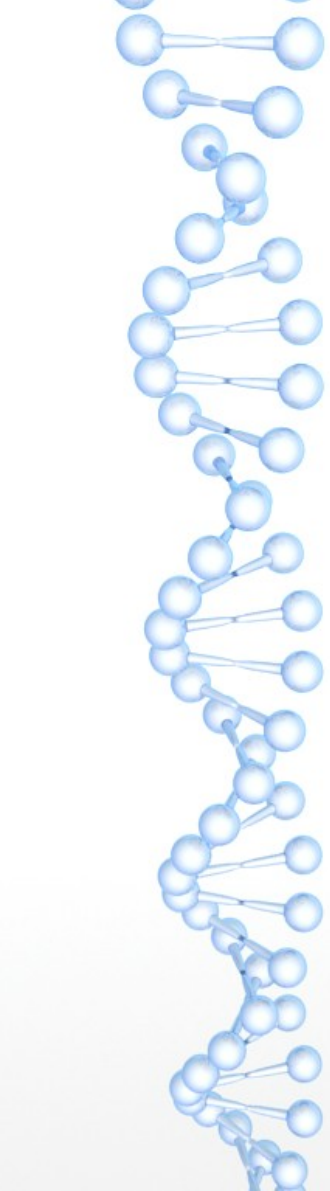
The work goes in the coroutine body

```
CountKmersTask GetReadAndCount(…) {
  std::string read;
  // co_await ResumeInIOThread{};
  …
}
```



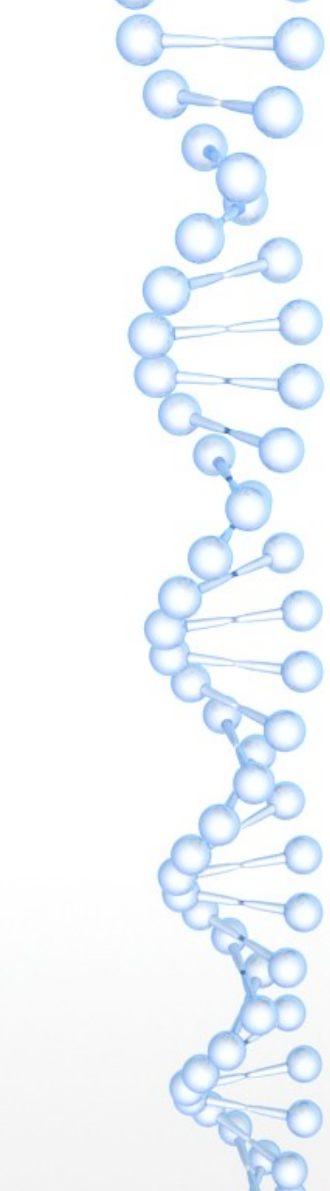
Learnings

- Consider not using coroutines
- Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end



Be aware of object lifetime and ownership

- Be sure to return suspend_always from final_suspend if you want to get the result from co_return
- References to coroutine internals are valid for the life of the coroutine
- Final_suspend is final. You can’t resume.



Conclusions

- Consider not using coroutines
  - Wait for standard library support in C++23 or C++26
  - Consider using a coroutine library if you can
- Coroutines are *not* subroutines – they're independent
- Plan out the use of the co_* keywords first in coro body
  - Then you can add the boilerplate
- The work goes in the coroutine body
- Be very careful with object lifetime and ownership
- You will have to write some sort of manager/queue
- Leave optimization for the very end

From Problem to Coroutine: Reducing I/O Latency

CHEINAN MARKS