FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Evaluation of thread pool implementations by resolution of webserver requests
Student:	Bc. Martin Mucha
Supervisor:	Ing. Daniel Langr, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Get familiar with the functionality of thread pools, i.e., systems that use a fixed number of threads for solving enqueued tasks. Study thread pool solutions available for the C++ programming language (such as OpenMP tasks, Intel TBB, CppCoro). Propose and implement your own thread pool in C++. Implement a simple webserver based on a thread pool system. Propose tests that use various kinds of requests. Use these tests to experimentally evaluate and compare different thread pool implementations.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 12 October 2022 in Prague.

Master's thesis

EVALUATION OF THREAD POOL IMPLEMENTATIONS BY RESOLUTION OF WEBSERVER REQUESTS

Bc. Martin Mucha

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Daniel Langr Ph.D.
May 6, 2024

Contents

List of Figures

List of Tables

List of code listings

I would like to extend my sincere gratitude to Ing. Daniel Langr, Ph.D. for his invaluable insights, patience, and guidance throughout the development of this thesis.
Additionally, I wish to express my heartfelt thanks to my family, partner, and friends for their unwavering moral support and encouragement.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Praze on May 6, 2024 .....

Abstract

Utilizing a pool of worker threads to scale web servers is a widely adopted approach employed by many server implementations. Individual thread pool implementations can significantly vary in scheduling algorithms and task encapsulation.

In this thesis, we design and implement our own thread pool utilizing C++20 coroutines and work-stealing scheduling. To ensure an efficient scheduling, we utilize lock-free structures. Our implementation not only achieves significantly lower scheduling overhead compared to OpenMP implementations but also performs comparably to, and in one test outperforms, the oneTBB thread pool implementation.

Finally, we conduct testing of various versions of our thread pool in a server environment to assess performance across different server requests types.

Keywords coroutines, parallel computing, scheduling, server requests, thread pool, web server, work-stealing

Abstrakt

Použitie thread pool, ako prostriedok na škálovanie webových serverov je zaužívaný v mnohých serverových implementáciach. Jednotlivé implementácie thread pool sa môžu výrazne líšiť v plánovacích algoritmoch a reprezentacií úloh.

V tejto práci, sme navrli a implementovali náš vlastný thread pool s využívam C++20 coroutines a work-stealing plánovacím algoritmom. Pre efektivné plánovanie úloh sme použili lock-free dátové štruktúry. Naša implementácia dosiahla výrazne nižšie oneskorenie pri planovaní v porovnaní s OpenMP implementáciami. Taktiež dosiahla porovnateľné výsledky v testoch s oneTBB knižnicou a v jednom z nich ju prekonala.

Nakoniec, sme otestovali rôzne verzie nášho thread pool v serverovom prostredí aby sme zmerali výkon naprieč rôznymi typmi požiadavkov na webový server.

Kľúčové slová coroutines, paralelné programovanie, plánovanie, serverové požiadavky, thread pool, webový server, work-stealing

Chapter 1
Introduction

With the increasing popularity of the internet, greater demands were placed on web servers, giving rise to the well-known C10K problem, which questioned whether a server could handle 10,000 concurrent connections. Advances in operating systems and hardware made it possible to overcome this problem. Moreover, the subsequent challenge of maintaining 10 million concurrent connections has also been overcome.

As hardware continues to evolve, so do strategies for scaling servers. One of the earliest approaches was adopted by Apache web servers, which spawned a separate process or thread for each connection, relying on the operating system's scheduler to manage these individual connections. This method, however, proved to be less effective. Over time, more complex approaches have emerged, such as worker pools and asynchronous event loops.

Worker pools in server architectures can be either dynamic or static. In a dynamic worker pool, a certain number of worker threads are actively waiting for requests; if the server load increases, additional threads are spawned to handle the demand. On the other hand, a static thread pool maintains a fixed number of worker threads throughout the server's lifetime, regardless of the load. Additionally, asynchronous event loops represent another approach for handling multiple requests, allowing for more efficient utilization of server resources, by deferring blocking operations and performing useful work in the meantime.

In this thesis we focus on designing and implementing our own static thread pool. To achieve this, we will explore multiple thread pool implementations to identify effective strategies. Two crucial design decisions must be addressed to effectively implement a thread pool.

The first component of a thread pool is task representation. This involves defining how to encapsulate tasks as units that thread pool workers can efficiently process. Good task representation can significantly enhance performance, ease of use, and simplicity of the pool's design.

The second crucial design decision involves the scheduling algorithm. C++ provides fine-grained control over program performance, which is essential for developing effective scheduling algorithms. Typically, these algorithms employ lock-free structures to achieve optimal performance, often utilizing relaxed atomics. Designing a new lock-free structure and an algorithm is beyond the scope of this thesis, will therefore explore various existing approaches that can be integrated into our thread pool.

Once we design and implement our own thread pool, we will conduct tests against various existing thread pool implementations to assess performance differences. Subsequently, we will integrate our thread pool into a simple server setup. This server will then be put under various types of load tests to evaluate performance of our thread pool implementation under various types of server requests.

Chapter 2
Background

2.1 Threads and processes

To leverage parallel execution and implement parallel algorithms, we need the ability to instruct each CPU core on which code to execute. Operating systems provide this capability through the use of threads. As programmers, we can create threads that are subsequently scheduled onto CPU cores. We will discuss the concepts of threads and processes in Linux and Windows, although similar concepts apply to macOS.

For example, on Linux, we would use the POSIX Threads library[1], while on Windows, we could utilize the ProcessThreadAPI[2]. A thread is the basic unit that is mapped onto a processor cores. This fundamental concept is consistent across Linux, Windows, and macOS.[3][4] [5]

The process of mapping threads onto real CPU cores is called scheduling. The scheduler is a system component that manages how processes share CPU time on individual cores. Typically, it operates in a preemptive manner[6], meaning that each thread is allocated a certain amount of CPU time; once this time is exhausted, another thread is scheduled on the core. Without preemptive scheduling, a process could block a CPU core indefinitely. In fact, it is possible to change the preemption model. For instance, in Linux, setting the `CONFIG_PREEMPT` in the kernel configuration enables different types of preemption. Another aspect that can be modified is the scheduling policy. In Linux, it is possible to choose from several different policies. We will show three of these policies[6]:

- `SCHED_NORMAL` - typical for regular tasks
- `SCHED_BATCH` - used with batch tasks, tasks are not preempted as often, allowing tasks to run longer and leverage caches
- `SCHED_IDLE` - for jobs with very low priority

As we've demonstrated, there are multiple ways to influence scheduling, each of which can impact performance[7].

2.1.1 Thread

We have deliberately not mentioned what exactly a thread is. Since threads are closely related to processes, we will describe both concepts in the following lines. Each thread consists of its own stack, registers, and thread-local storage[4][5]. This is common across different operating systems; although there may be slight deviations, the core concept remains the same. Each thread maintains its own execution context.

A process groups multiple resources together including an address space, threads, handles to files and sockets. However, the specifics can differ among operating systems. Each process begins with a single thread known as the main or primary thread, which must exist for the process to execute. As previously mentioned, a process itself cannot be scheduled directly; only the threads within the process.

The main advantage of processes is the grouping of multiple threads together, which allows for more effective communication among them through shared memory. This method of communication has much lower overhead compared to inter-process communication (IPC). Furthermore, creating new threads is faster than creating new processes, as threads within the same process share many resources.

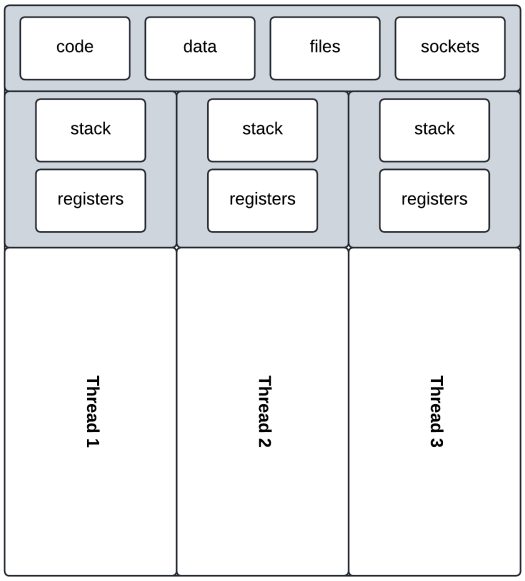

Figure 2.1 Representation of shared resources between threads in a process.

For example, in Linux, the scheduler represents individual processes and threads using the `task_struct`[8] structure. This structure contains all necessary information for scheduling, like policies and priorities, as well as details about which thread to schedule. This design allows the scheduler to abstract away from processes and threads.

2.1.2 Green threads

Green threads are an abstraction that work on top of actual threads. While OS-level threads are mapped onto CPU cores, green threads are mapped onto these OS-level threads. The OS scheduler does not schedule green threads directly, as they exist purely in the program. The mapping can be M:N, meaning multiple green threads may be mapped to a given number of OS-level threads.

Since green threads are not scheduled by the operating system's scheduler, they must be managed manually within the program. This added layer of abstraction can provide significant benefits, particularly in scenarios requiring fine-grained control, such as task-based parallelism and nested parallelism.

Many modern programming languages now support green threads, either through external libraries or as an feature of the language itself. Examples include Go's goroutines[9] and Rust's Tokio library[10], among others.

2.1.3 Thread pool

The concept of green threads is closely tied to the idea of a thread pool. A thread pool represents a set amount of OS-level threads that are available to execute tasks. Each green thread needs to be mapped onto one of these OS-level threads for execution. The way green threads are represented and managed can differ between implementations.

Thread pools can be categorized as either static or dynamic based on how they manage OS-level threads. A static thread pool maintains a fixed number of threads throughout its lifetime. Conversely, a dynamic thread pool adjusts the number of threads based on demands. For the purposes of this thesis, our focus will be on static thread pools.

As previously mentioned, the operating system is unaware of green threads; therefore, they must be scheduled at the application level. The responsibility of the scheduler is to effectively map these green threads onto OS-level threads. Different mapping strategies exist and are crucial for achieving optimal performance.

The main benefit of thread pools is the lower overhead compared to creating a new thread for each task. Creating a new OS-level thread involves overhead, whereas scheduling a task onto an existing thread in a pool can be simple as calling a function. However, this raises a question about resource utilization: Do thread pools waste resources when idle ? Typically, thread pools are designed to yield the OS-level threads if no task are being executed for a certain period. This allows the OS scheduler to schedule different threads and reschedule the thread pool's threads later in time.

2.2 Scheduling

Scheduling strategy plays a crucial role in parallel programming, where the effectiveness of a scheduler directly impacts the performance of the system. The responsibility of a scheduler is to effectively schedule individual tasks for best performance. An appropriate scheduling strategy can significantly reduce issues like false sharing and context switching. By minimizing these inefficiencies, the scheduler can improve performance.

There are two main scheduling strategies in parallel computing: work sharing and work stealing. Each has its distinct advantages and disadvantages.

2.2.1 Work sharing

Work sharing involves a global scheduler that can be accessed by all thread in the system. In this strategy, each thread may get additional tasks from the scheduler, either by actively requesting or through assignment by the scheduler. Workload distribution can be based on various schemes, such as priority, the current load of each thread or other factors.

We will demonstrate simple static scheduling on a problem of updating an array of numbers by incrementing each number by one. This example illustrates a basic form of work sharing, where each worker thread is assigned a fixed portion of the workload. For instance the split can be determined at compile time and each thread gets predetermined chunk of the array.

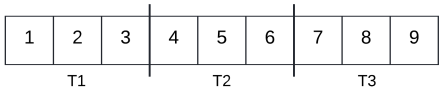

Figure 2.2 Workload distribution in simple work sharing implementation.

Figure 2.2 shows that each worker gets equal amount of work, equally sized chunks of the

array. This approach encourages cache consistency. If the chunks are big enough modification of the values does not invalidate cache in other threads.

Other modification can include dynamically assigning chunks to individual threads. If the workload is not distributed evenly some chunks can take longer to process. Dynamically assigning individual chunks to thread that has already finished can improve performance. With the approach shown in figure 2.2 some threads could be idly waiting for other threads to finish instead of doing useful work.

An alternative modification to the static scheduling approach is to dynamically assign chunks of the array to individual threads as they become available. This method addresses situations where the workload might not be evenly distributed across individual chunks, causing some chunks to take longer to process than others. By dynamically allocating tasks to threads that are free, performance can be improved. In contrast, static approach illustrated previously may lead for some thread to idly wait for other threads to finish.

Typically work sharing is used with loop based parallelism.

2.2.2 Work stealing

Unlike work sharing, where a global scheduler assigns tasks to individual threads, work stealing has a “decentralized” model. Each threads is responsible for generating its own tasks and managing its task queue. Additionally, when a thread completes its tasks and becomes idle, it can “steal” tasks from the queue of other threads. This implies that each thread must maintain its own set of tasks to process, which support cache coherence. Stealing other thread’s tasks potentially improves load balancing across the thread pool.

A basic implementation can be achieved using locks and queues. In this setup, each thread maintains its own queue for tasks. When a thread completes its tasks and needs more work, it can attempt to “steal” tasks from another thread’s queue. To do this, it must first acquire a lock on that queue. While this example serves to illustrate the concept, it’s important to note that locks can introduce significant overhead into the scheduling process.

Work stealing is primary used in task based programming. Each worker works on its own task and upon encountering forking point (new tasks needs to be created), they are put into the thread’s own queue, compared to work sharing where there exists a global queue.

Work stealing is primarily used in task-based multithreading. Each worker executes its own task and when a forking point is encountered, the new spawned tasks are put into the thread’s container. This is a key difference from work sharing strategy, where tasks are typically managed in a global queue accessible by all threads

2.3 Hardware memory model

In the previous chapter, we covered threads, how operating systems schedule processes and threads, and the advantages of thread-based communication over inter-process communication (IPC). Since threads can execute concurrently, there is a need for synchronization mechanisms to facilitate communication between threads.

Hardware memory models describe how CPUs, based on specific architectures, manage thread-to-thread communication. Since threads are executed on CPU cores, we use the term ’threads’ in this context. By understanding and following the rules defined by these models, we can effectively program against them.

2.3.1 Sequential consistency

Sequential consistency is a concept defined as follows: ”the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations

of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called sequentially consistent"[11]

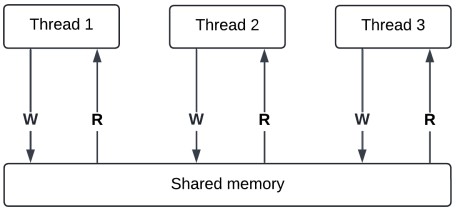

Figure 2.3 Representation of sequential consistent model.

Two important properties manifest from this model[12]:

- Each instruction is executed in order specified by the program, implying no reorderings.
- Each modification, writes, are visible to all thread at the same time.

This is the conventional understanding of program execution: on a single processor, instructions are executed sequentially, one following the other, giving the impression that the code is processed from top to bottom. However, in environments with multiple processors, instructions from different threads can interleave. The sequence in which these instructions are executed may vary due to timing. Nevertheless, once an instruction is executed, its effects become visible to all other threads.

In modern CPUs, memory is structured in multiple layers, which represent different types of memory. At the top is the fastest type of memory, CPU cache, and at the bottom slower, but shared across all CPU cores is the RAM.

For changes in memory to be visible across different threads, these modifications must propagate through all these layers. The sequential consistency model can be implemented in CPUs with such multi-level memory architectures. However, to ensure sequential consistency, it must guarantee behavior described above. One method to enforce this behavior is the use of memory barriers.

2.3.2 x86-TSO

The sequential consistency model is straightforward and easy to reason about. However, modern processors typically implement this model:

"Early multiprocessors maintained this fiction, but more modern ones usually do not. Instead, they provide special commands with which processes themselves can synchronize memory accesses. The programmer must determine, for each particular computer, what synchronization commands are needed to make his program correct".[13]

Another hardware model is x86-TSO(total store order)[14], which describes how x86 processors manage synchronization. Unlike sequential consistency, x86-TSO is a more relaxed model. It does not guarantee sequential consistency, allowing for certain types of reorderings to optimize performance

Obvious addition compared to sequential consistency model is the write buffer, as shown in Figure 2.4. This buffer operates as a simple FIFO (First In, First Out) queue, which ensures that the order of writes is preserved. To make synchronization possible, special instructions can be issued to flush this buffer. Write buffer is a convenient method to prevent thread blocking while waiting for writes to complete.

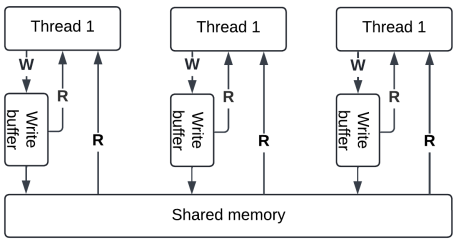

Figure 2.4 Representation of TSO memory model.

Another important characteristic is the ability to read its own write buffer. The thread is forced to read its own memory buffer first, before accessing the shared memory. This allows the thread to read value from the shared memory before publishing a write from the buffer, essentially doing a reordering.

Before accessing shared memory, a thread is required to read from its own memory buffer first. This also allows to threads to continue execution, before the writes propagate to main memory, basically facilitating reorderings.

Even on x86 sequential consistency can be guaranteed, however, this type of synchronization comes at a performance cost in form of special instructions and barriers.

2.3.3 ARM and POWER

The final hardware memory model we will discuss is for ARM and POWER processors. This model is even more relaxed than the x86-TSO model used in x86 processors. The primary reasons for this greater flexibility include considerations for performance enhancement, power efficiency, reduced hardware complexity, and historical design choices[12].

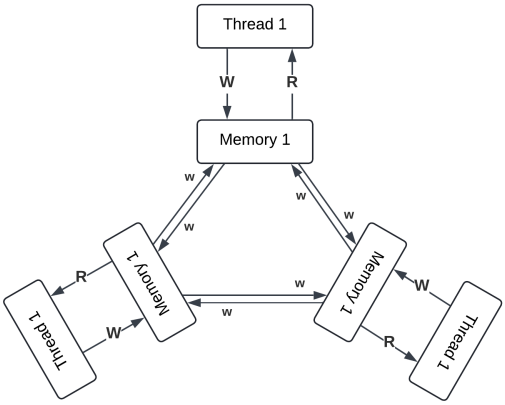

Figure 2.5 Representation of ARM/POWER memory model.

Two main characteristics are[12]:

- Reads and writes can be executed out of order or even speculatively (before preceding conditional branches have been resolved).
- No guarantee that the atomic writes will be visible to all threads at the same time, implying

no multiple copy atomic model. In comparison x86 is multiple copy atomic model, once a write is propagated to main memory it is propagated to all threads.

This model is the most relaxed out of these three. Each thread acts on its own "copy" of memory, and all modification can be propagated independently to different threads.

This model does not fully apply to the ARMv8 architecture, which introduces the multiple copy atomic model[15], thereby strengthening the memory model. We will not cover differences between these two models here. For a more comprehensive explanation, readers are encouraged to refer to the source material. In following chapters, we will use model demonstrated above.

Our main goal was to illustrate that different CPU architectures permit various optimizations, which can lead to confusion when code behaves differently on different CPU architectures. For instance, an error in our code might not manifest in its execution due to architectural constraints.

To further illustrate the differences between each architecture, we will conduct several litmus tests.

2.3.4 Store buffering

We will employ a uniform notation where 'r' signifies a local register, representing a local variable. The letters 'x', 'y', and 'z' will denote shared variables. All variables are initialized to zero.

```
// thread 1              // thread 2
x  = 1                   y = 1
r1 = y                   r2 = x
```

Code listing 2.1 Store buffering.

Can the program end with $r1 = 0 \wedge r2 = 0$?

Sequential consitent model No. The Sequential Consistency Model explicitly states that each instruction can only be executed after the preceding one has been completed, and all memory modifications are visible to all other threads. To demonstrate this, we could enumerate all possible interleavings and examine whether any configuration permits such an execution sequence.

x86-TSO Yes. This model allows for such an execution. In the x86-TSO model, stores can be buffered, enabling reads to be executed before the stores have propagated out of the buffer to other threads.

ARM/POWER Yes. Given that this model is even more relaxed than x86-TSO, it permits writes to propagate independently from local memory to other threads. This allows for situations where a read may be executed before other threads see the write.

2.3.5 Load buffering

```
// thread 1              thread 2
r1 = x                   r2 = y
y = 1                    x = 1
```

Code listing 2.2 Load buffering.

Is it possible for program to end with $r1 = 1 \wedge r2 = 1$?

Sequential consitent model No. There is no possible interleaving that would produce the desired result.

x86-TSO No. This model allows threads to store writes, but not loads. The load needs to happen before the write.

ARM/POWER Yes. ARM/POWER model allows for read buffering, by executing the instruction out of order.

2.3.6 IRIW

Independent reads of independent writes, is similar to store buffering but extended to multiple threads. Individual threads are reading the shared variables in opposite order.

```
// thread 1     // thread 2     // thread 3     // thread 4
x = 1           y = 1           r1 = x          r3 = y
                                r2 = y          r4 = x
```

Code listing 2.3 Independent reads of independent writes.

Can the program end with $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$?

Sequential consitent model No. Once a write operation is committed to shared memory, it must become immediately visible to all other threads. This implies that once thread 3 reads `x = 1`, then thread 4 must also read `x = 1`. Otherwise, if thread 4 were to execute before thread 3 and reads `y = 1`, thread 3 should also read `y = 1`. However, if thread 3 reads `y = 0`, this leads to a contradiction. It contradicts the desired result with thread 3 reading `y = 0`. Another approach to proving this would be to enumerate all possible interleavings.

x86-TSO No. Storing writes into the buffer does not make a difference to sequential consistent model in this case. Individual reads need to happen in order and writes need to be visible to all threads.

ARM/POWER Yes. ARM/POWER model allows for independent write propagation, allowing thread 3 and 4 to see writes in different order.

2.4 C++ memory model

In the chapter Hardware memory model, we explored various hardware memory models. Now, we will delve into the C++ memory model, which was first introduced in C++11. Over the years, this model has undergone several changes. In this discussion, we will focus on the C++20 memory model. C++ is not the only language using this model, Rust also adopts this model for its atomics[16].

In higher-level languages, programmers typically do not directly engage with the hardware memory model. They interact with the language's memory model. The task of accurately mapping the program's behavior to the underlying hardware memory model is handled by the compiler.

From a programmer's perspective, it does not matter whether optimizations are made by the compiler or the CPU; the key requirement is that these optimizations must not alter the intended behavior of the code. This principle is similar to the 'as-if'[17] rule in C+. Therefore, the memory model can be seen as a contract among the programmer, the compiler, and the hardware. This arrangement is practical, allowing for abstraction against which programmers can program.

Explaining the entire memory model is beyond the scope of this thesis due to its complexity. The primary goal of this thesis is to design a thread pool, and for this purpose, we need a foundational understanding of how atomics work in C++. We will therefore focus only on the essential details, excluding topics such as `memory_order::consume` and memory fences.

In the following sections, we will outline the necessary rules and relationships for understanding the basics of the C++ memory model. As previously mentioned, we will omit some rules that are not needed for our use cases. However, all rules discussed will be directly sourced from the C++20 standard[17], with any changes commented on.

Memory order

Each atomic operation in C++ can specify its memory order by passing an additional argument. The term 'atomic operation' might seem misleading because these operations do more than just ensure atomicity, they can also impose synchronization constraints. This means that atomic operations not only prevent data races when multiple threads modify a variable simultaneously but also synchronizes different threads.

Possible memory orders are:

- `memory_order::relaxed` does not impose any synchronization, only ensure that modifications are atomic.
- `memory_order::acquire` pairs with release memory order. All operations after this atomic operation can observe modifications done by the releasing thread, before the release operation.
- `memory_order::release` guarantees that all modifications before this atomic operation are visible to the acquiring thread.
- `memory_order::release` guarantees that all modifications before this atomic operation are visible to the acquiring thread.

We have provided a brief description of each type of memory order to hint their roles. To fully understand how these memory orders function, we must first define some rules and relationships. As stated previously, there are additional memory orders, but these are not necessary for us.

Sequenced before

This relationship determines the order of evaluation within the program. Following rules apply:

- If A is sequenced before B (or, equivalently, B is sequenced after A), then evaluation of A will be complete before evaluation of B begins.
- If A is not sequenced before B and B is sequenced before A, then evaluation of B will be complete before evaluation of A begins.

The full definition and all cases covered by the `sequenced before` relationship are too extensive to detail here. For our purposes, we can consider `sequenced before` as synonymous with program order. This means that each operation in the program is sequenced before the next operation that appears in the code. Consequently, all modifications and side effects of one operation are visible to any operation that follows it sequentially. This relation is defined as asymmetric, transitive, and pair-wise.

Synchronizes with

"If an atomic store in thread A is a release operation, an atomic load in thread B from the same variable is an acquire operation, and the load in thread B reads a value written by the store in thread A, then the store in thread A synchronizes-with the load in thread B.[18]

The crucial part of this relation is that it requires reading the value stored by another thread. Without this no synchronization happens. The release operation does not necessarily have to

be `memory_order::release` it can be a 'stronger' memory order, such as `seq_cst`. The same principle applies to the acquire operation.

Memory order `relaxed` does not form `synchronizes with` relationships.

Happens before

Evaluation `A` happens before evaluation `B`, when one of the following is true:

▶ **Definice 2.1** (HB-1). *`A` is sequenced before `B`*

▶ **Definice 2.2** (HB-2). *`A` synchronizes with `B`*

▶ **Definice 2.3** (HB-3). *`A` simply happens before `X` and `X` simply happens before `B`*

The C++ standard refers to this relationship as `simply happens before`. However, since the use of `memory_order::consume` is discouraged and not widely adopted, we have chosen to simplify by omitting this difference. Thus, we will refer to this relationship `happens before`.

This relationship is important especially with `memory_order::acq` and `memory_order::rel`, these two operations (or stronger version with `seq_cst`) can synchronize between threads. Once synchronization occurs, the thread performing the acquire operation will see all the side effects and modifications that occurred before the release operation in another thread. In other words, one thread can see all operations and side effects that `happens before` the release operation on the other thread.

Another interpretation can be that this relation extends `sequenced before` between threads, because sequenced before relation works only in context of a single thread.

Modification order

All modifications to any particular atomic variable occur in a total order that is specific to this one atomic variable.

The following four requirements are guaranteed for all atomic operations[18]:

▶ **Definice 2.4** (MO-1). *Write-write coherence: If evaluation A that modifies some atomic M (a write) happens-before evaluation B that modifies M, then A appears earlier than B in the modification order of M.*

▶ **Definice 2.5** (MO-2). *Read-read coherence: if a value computation A of some atomic M (a read) happens-before a value computation B on M, and if the value of A comes from a write X on M, then the value of B is either the value stored by X, or the value stored by a side effect Y on M that appears later than X in the modification order of M.*

▶ **Definice 2.6** (MO-3). *Read-write coherence: if a value computation A of some atomic M (a read) happens-before an operation B on M (a write), then the value of A comes from a side-effect (a write) X that appears earlier than B in the modification order of M.*

▶ **Definice 2.7** (MO-4). *Write-read coherence: if a side effect (a write) X on an atomic object M happens-before a value computation (a read) B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.*

Strongly happens before

Evaluation A strongly happens before an evaluation D if one of the rules applies:

▶ **Definice 2.8** (SH-1). *`A` is sequenced before `D`*

▶ **Definice 2.9** (SH-2). *`A` synchronizes with `D`, and both `A` and `D` are sequentially consistent atomic operations*

▶ **Definice 2.10** (SH-3). *there are evaluations `B` and `C` such that `A` is sequenced before `B`, `B` happens before `C`, and `C` is sequenced before `D`*

▶ **Definice 2.11** (SH-4). *there is an evaluation `B` such that `A` strongly happens before `B`, and `B` strongly happens before `D`*

This relationship is more constrained than `happens before` as we can see it also "extends" `sequenced before`, however form `SH` both operations need to be `seq_cst`.

Coherence ordered before

An atomic operation A on some atomic object M is `coherence ordered before` another atomic operation B on M if:

▶ **Definice 2.12** (CO-1). *`A` is a modification, and `B` reads the value stored by `A`*

▶ **Definice 2.13** (CO-2). *A precedes B in the modification order of M , or*

▶ **Definice 2.14** (CO-3). *A and B are not the same atomic read-modify-write operation, and there exists an atomic modification X of M such that A reads the value stored by X and X precedes B in the modification order of M*

▶ **Definice 2.15** (CO-4). *there exists an atomic modification X of M such that A is coherence-ordered before X and X is coherence-ordered before B*

Single total order

There is a single total order S on all `memory_order::seq_cst` operations that satisfies the following constraints:

▶ **Definice 2.16** (TO-1). *if A and B are `memory_order_seq_cst` operations, and A strongly happens-before B, then A precedes B in S,*

▶ **Definice 2.17** (TO-2). *if A and B are both `memory_order_seq_cst` operations, A is coherence ordered before B, then A precedes B in S.*

These two rules allow for construction of a single total order S that all threads observer. All threads agree on this order, implying all threads agree on the order of changes. This single total order has to be consistent with modification order of any atomic variable.

To construct this order we need mainly `strongly happens before` and `coherence order before` relations.

2.4.1 Store buffering

With relations defined above we can answer following litmus test.

```
// thread  1            // thread 2
A: x = 1 (SC)           C: y = 1  (SC)
B: r1 = y (SC)          D: r2 = x  (SC)
```

Code listing 2.4 Store buffering with C++ atomics.

Can the program end with $r1 = 0 \wedge r2 = 0$, if all operation are `memory_order::seq_cst` ?

In the Hardware memory model chapter, the sequential consistent model did not allow this. Do C++ `seq_cst` operations allow this ?

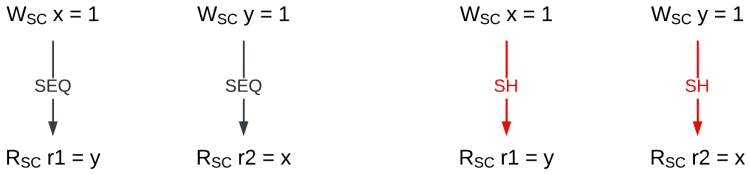

Figure 2.6 Operations form `strongly happens before` relationship according to SH-1

In Figure 2.6 we show `sequenced before` relation denoted (SEQ) and `strongly happens relation` denoted (SH). Sequenced before relation is formed from the program order, using the definition. Given all operation are sequentially consistent, they form `strongly happens before` relationship, SH-1 .

With sequential consistent operations we can try to build a total order S observed by all threads. According to TO-1 , it is clear that A precedes B and C precedes D. This implies that at least one load/read operations must be last in this order. Therefore, the last operation has to read the value stored by the modification according to MO-4. Moreover order S has to be consistent with modification order of the variable. Any interleaving of the operations must result in at least one thread reading the stored value. This forms following additional edges:

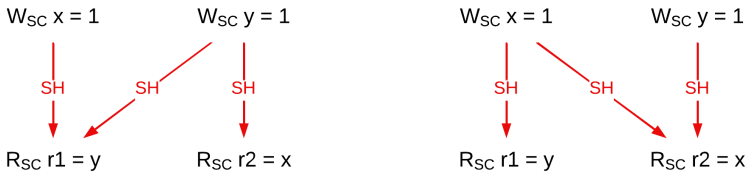

Figure 2.7 Illustration of possible executions of the program with sequential consistent atomic operations.

Another perspective on this problem is to consider the scenario where both B and D read 0. This would imply that they must have read these zeros before A and C had happened. Given CO-3 and TO-2, we would derive that A must precede B, C must precede D, B must precede C, and D must precede A. This sequence forms a cycle, which is not possible in total order, at it would violate the principel that total order must be acyclic[19]. The answer to the test is: No.

With acquire and release semantics. These operations form following relations shown in Figure 2.8.

There is no total order to construct TO-1 nor TO-2 are satisfied. Each tread can perceive the operations in different order. The answer to the test is: Yes.

Same argument follows for relaxed atomic operations.

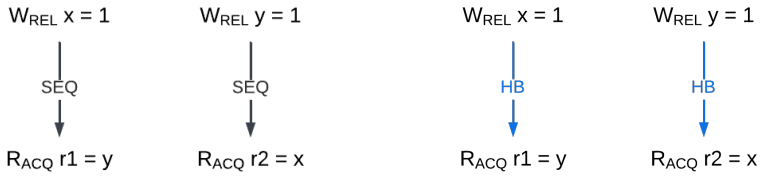

Figure 2.8 Relations formed by acquire and release atomics.

2.4.2 Load buffering

```
// thread  1            // thread 2
A: r1 = x (SC)          C: r2 = y  (SC)
B: y = 1 (SC)           D: x = 1  (SC)
```

Code listing 2.5 Load buffering with C++ atomics.

Is it possible for program to end with $r1 = 1 \wedge r2 = 1$?

From the `sequenced before` relationship and SH-1, we establish `A strongly happens before B` and `C strongly happens before D`. Assume `B` and `D` read one from shared variables `x` and `y`. According to SH-2, this would create a new `strongly happens before` relationship. Implying a cycle in the single total order, according to TO-2. For `memory_order::seq_cst` the answer to the test is No.

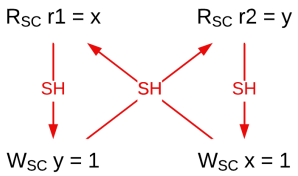

Figure 2.9 Cycle formed in total order `S` in load buffering example.

With acquire and release operations, the single total order is not formed. However, consider the scenario `A` and `C` read values stored by `B` and `D`. According to HB-2, this establishes new edges and thus creating a cycle. For acquire and release operations the answer is No.

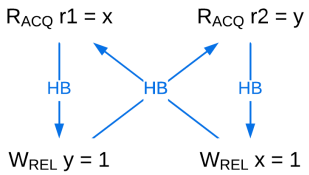

Figure 2.10 Acquire and release operations also form a cycle due to synchronization.

With `memory_order::relaxed` operations, `A` is `sequenced before B` and `C` is `sequenced before D`. Relaxed atomics do not provide any synchronization guarantees between threads. Only relation that form by reading the stored values is CO-1. Which can form a cycle, given they do not synchronize. The answer for relaxed atomics is Yes.

2.4.3 IRIW

```
// thread  1      // thread 2       // thread 3       // thread 4
A: x = 1 (SC)     B: y = 1 (SC)     C: r1 = x (SC)     E: r3 = y (SC)
                                    D: r2 = y (SC)     F: r4 = x (SC)
```

Code listing 2.6 Independent reads of independent writes with C++ atomics.

Can the program end with $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$?

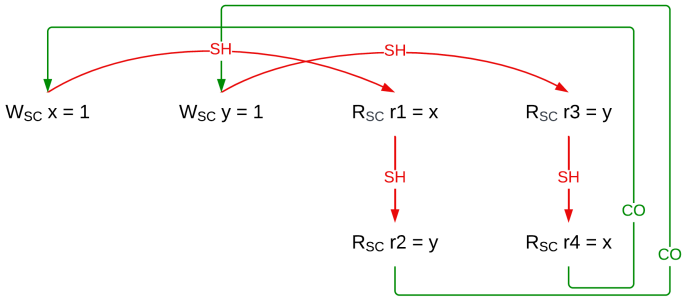

Figure 2.11 Illustration of relations formed with sequentially consistent atomics in IRIW.

Let's assume thread 3 reads values stored by thread 1, forming a `synchronize with` relationship, given that these are sequential consistent operations. According to SH-2, this sets up `strongly happens before` relationship. This means A must precede C in single total order S and `C` must precede `D`. Moreover, for thread 3 to read zero, `D` needs to precede `B`, according to CO-3 this forms a `coherence ordered before` relation. Next, lets assume, thread 4 reads `y=1` in E. This forms `B strongly happens before E` and `E strongly happens before F`. However, for F to read 0, new `coherence ordered before` relationship forms between `F` and `A`. According to TO-1 and TO-2, forming a total order would contain a cycle. Therefore, the answer for `memory_order::sec_cst` is No.

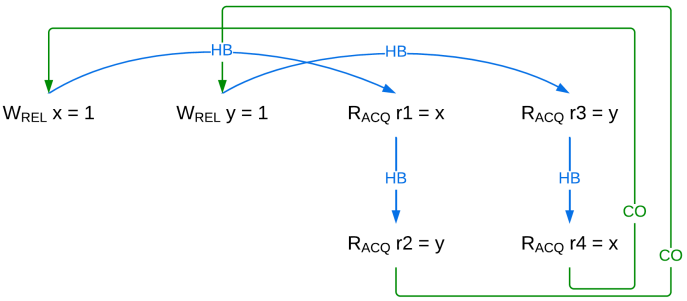

Figure 2.12 Relations formed by acquire and release atomics in IRIW litmus test.

With acquire and release atomics, there is no established single total order. As illustrated in Figure 2.12, thread 3 and thread 4 do not synchronize with each other, allowing each to maintain its own perspective on sequence of events. The answer for acquire and release atomics is Yes.

For relaxed atomics, the answer is also Yes, for apparent reasons.

2.4.4 Out of thin air values

```
// thread 1                    // thread 2
A: r1 = x (RLX)                C: r2 = y (RLX)
if (r1 == 42)                  if (r2 == 42)
  B: y = 42 (RLX)                D: x = 42 (RLX)
```

Code listing 2.7 Out of thin air values in C++.

Can this program produce $r1 = 42 \wedge r2 = 42$?

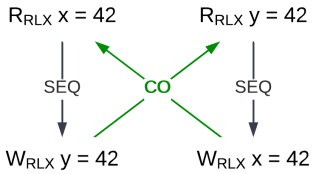

Figure 2.13 Relations formed by relaxed atomics, allowing for out of thin air values.

From `sequenced before` relationship we know that `A` is `sequenced before B` and `C` is `sequenced before D`. For the final result the threads need to read the stored values from each other. Since these are relaxed operations, they do not synchronize according to `synchronizes with` relation. Only relation that is formed is according to MO-1. Although these operations imply a cycle, there is no synchronization between threads, therefore the cycle is allowed. The rules stated above allow for such a behavior, called our of thin air values. However, C++ standard includes this statement : "Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation."[17], disallowing such a behavior. This statement is somewhat vague, but aims to address the gap in C++ memory model rules.

Chapter 3

Available tools

To effectively leverage parallel programming, we need constructs that allow us to program in parallel way. The C++ standard library offers several tools for this purpose, including `std::thread`, `std::atomic` and others. We will delve into these in greater detail in the next section.

Additionally, we will explore third-party libraries. This will range from simpler ones like the OpenMP specification and its implementations to more robust frameworks like oneTBB. Our focus will be on their functionality and capabilities.

Given the vast number of available libraries for parallel programming, covering them all is beyond our scope. Instead, we will examine a selection of these libraries in greater depth, highlighting their most important and distinctive features.

In the following sections, we will use two examples: calculating the sum of an array and computing the n-th Fibonacci number through recursion. The first problem will serve as a basis for demonstrating loop parallelism, requiring a division of the array into chunks that can be processed concurrently. The second problem will illustrate the need for task-based parallelism. Through these examples, we aim to demonstrate the syntax and practical application of various libraries. Below, we present straightforward sequential versions of these algorithms:

```
int array_sum() {
  int sum = 0;
  for (int i = 0; i < arr.size(); i++)
    sum += arr[i];
  return sum;
}
```

Code listing 3.1 Calculating sum of an array sequentially.

```
int fib(int n) {
  if (n <= 1) return 1;
  return fib(i-1) + fib(i-2);
}
```

Code listing 3.2 Calculating n-th Fibonacci number sequentially.

3.1 C++ standard library

C++ provides a variety of parallel constructs in its standard library, with some being more high-level than others. In this section, we aim to discuss the most significant ones.

For parallel programming, it's essential to be able to execute code on different threads. The C++ standard provides `std::thread`, a low-level object used to launch new threads of execution. The association of `std::thread` with an actual thread depends on the object's state. To synchronize the thread with the main thread, we can use the join method, which blocks until the spawned thread completes. Alternatively, the detach method allows the thread to run independently from the main thread.

```
int main() {
  std::thread my_thread([]() {
    // do something;
  });
  my_thread.join();
}
```

Code listing 3.3 Spawning a new thread and waiting for its completion.

C++20 introduced a new thread object named `std::jthread`, which closely resembles the original `std::thread`, with the key distinction being its automatic joining upon destruction, making the destructor a blocking call. Another notable difference is the ability to pass an object called `std::stop_token` to `std::jthread`. This token is commonly used with `std::jthread` to facilitate a cooperative stop request for the thread. For the stop request to take effect, the thread must periodically check for this request and stop its execution accordingly.

Both `std::thread` and `std::jthread` are low-level constructs designed to enable users to spawn individual threads.

Executing code in parallel also requires some form of synchronization. One of the most typical one is mutex[20]. In C++ we can use `std::mutex`. To ensure exclusive access, each thread must acquire the mutex by calling either the `lock` or `try_lock` methods, both of which are blocking operations. C++ provides several variations of mutexes, including:

- `std::timed_mutex` - `lock` method returns after certain amount of time.
- `std::recursive_mutex` - this mutex can be locked multiple times by the same thread, allowing for reentrant locking patterns.

To make the mutex usage more convenient, without the need to call `lock` and `unlock` methods, we can utilize the `std::lock` function. This function locks any number of mutexes without deadlock, eliminating the need for manual locking of each mutex.

Furthermore, we can use `std::scoped_lock`, which serves as a RAII wrapper around the `std::lock` function. This ensures that mutexes are automatically unlocked when the `scoped_lock` object goes out of scope.

```
{
  std::scoped_lock(mutex); // (1)
  // access shared memory
} // (2)
```

Code listing 3.4 Using scoped_lock to guard critical region.

When a thread attempts to enter the code block, it must first acquire the mutex specified on line (1). This gives the thread exclusive access to shared memory. Upon exiting the enclosing scope, the mutex is automatically released on line (2). Similar to mutexes, there are several variants of the `std::lock` wrapper.

It's important to note that the C++ standard[17] doesn't specify the exact behavior when a thread waits for a mutex, only that the thread will be blocked. For instance, in the GCC and Clang implementations on Linux, a blocked thread is typically put to sleep and yielded, allowing the OS scheduler to execute a different thread or process. The blocked thread is then woken up later. This behavior has been observed using the perf profiler[21].

Another synchronization mechanism is `std::condition_variable`, which is used together with `std::mutex`. A thread waiting on a condition variable is usually yielded and rescheduled once the condition variable is notified. Execution on the yielded thread continues once the condition is met, even in case of spurious wakeups.

There is also support for `std::barrier` which allows multiple threads wait on the barrier and start executing once a certain condition is met. Thread local storage allows users to store data separately for each thread.

3.1.1 Atomics

C++ provides support for atomic variables, which can be accessed and modified in a thread-safe manner. Atomic variables are typically implemented in a lock-free way, relying on hardware support for atomic operations. While the C++20 standard doesn't mandate that atomic variables must be implemented lock-free, it does include the method `std::is_lock_free`, which can be used to check the implementation of the atomic variable. The C++ standard[17] suggests, but doesn't require, that lock-free atomic operations should also be address-free. Lock-free atomics allows for additional optimizations and more effective communication via shared memory.

```
std::atomic<int> num; // (1)
num = 10; // (2)
num.store(10); // (3)
```

Code listing 3.5 Demonstration of C++ atomics.

To manipulate atomic variables, we can use methods such as `store` or `load`, among others. For some of these methods, operator overloads exist, allowing for more intuitive usage. In our example, we could use the `store` method as shown on line (3), or utilize the overloaded assignment operator as demonstrated on line (2).

Additionally, we can create an atomic reference to an object using `std::atomic_ref`, ensuring atomic modifications to the object. However, modifications must be made through `std::atomic_ref`. With `std::atomic_compare_exchange` there's also support for compare-And-Swap (CAS)[22] operations. C++20, even supports atomic `std::shared_pointer`.

Atomic variables provide low-level synchronization. For more fine-grained control, atomic operations can be used with different memory orders. By default, if no specific order is mentioned, `std::memory_order_seq_cst` (sequential consistency) is used. For instance, in the example shown in 3.5, on line (3), we can specify a different memory order, such as `memory_order_relaxed`, by using `num.store(10, std::memory_order_relaxed)`.

Using different memory orders can be quite challenging, particularly in complex programs where ensuring correctness becomes more difficult. However, in performance-critical applications, carefully chosen memory orders can bring significant performance improvements.

3.1.2 Future and promise

In the standard library C++ offers support for futures[23] and promises. Futures are used to retrieve the result of an asynchronous operation. Typically, we the call `get` method on the future object to wait for the result. If the result is not yet available, the call blocks until it becomes ready.

The `std::promise` object is used to store the value or an exception that the corresponding `std::future` object will retrieve. The promise object and future object are tied together and both share the same shared state. Values can be stored in the promise by calling `set_value`, and exceptions can be stored by calling `set_exception`.

It is crucial that only one thread can call the `std::future.get()` on the promise. If the result needs to be retrieved in multiple threads, `std::shared_future` should be used. To retrieve the

result asynchronously, the function must be executed on a different thread. This can be done manually by creating a future and promise pair and then spawning a new thread.

```
std::promise<int> promise;
std::future<int> future = promise.get_future();
auto lambda_fun = [promise = std::move(promise)]() mutable{
  int result = 42;
  promise.set_value(42);
};
std::thread thread(std::move(lambda_fun));
int result = future.get();
thread.join();
```

Code listing 3.6 Creating future and promise objects and executing a function on a different thread.

This code snippet demonstrates how to manually utilize the `std::future` and `std::promise` objects. However, C++ standard library offers more higher level constructs, such as: `std::async` and `std::packed_task`.

With `std::packaged_task` we can wrap a function, a lambda function or a callable object. This automatically encapsulates the return value into `std::future` object. Both values and exceptions are returned through this future object. This simplifies usage compared to manually calling `promise.set_value` inside the function.

```
auto task = std::packaged_task<int()>([]() {
  return 42;
});
auto future = task.get_future();
std::thread thread(std::move(task));
int result = future.get();
thread.join();
```

Code listing 3.7 Utilizing `std::packaged_task` to execute a function on another thread.

`std::async` provides an even higher level of abstraction. We can pass a callable object into `std::async`, as long as it is a movable object. The function call returns `std::future` which allows us to retrieve the value of the function. Another argument to `std::async` is a launch policy. The launch policy determines the type of execution of the function. There are two options:

- `std::deferred` wraps the function and causes it to be lazily evaluated on the thread that calls wait on the associated `std::future` object.
- `std::async` launches the execution on a different thread.

```
auto future = std::async(std::launch::async,[](){
  return 42;
});
int result = future.get();
```

Code listing 3.8 Using `std::async` to launch a function on another thread.

If neither of these policies is specified, the implementation chooses how to execute the function based on available resources[17].

3.1.3 Parallel algorithms

The C++ standard features a library known as the algorithm library, which contains implementations of commonly used algorithms designed to operate on ranges of elements. Examples include `std::find`, which is used to locate an element that meets specific criteria, and `std::count`, which counts the number of elements that satisfy a given condition, among many others.

As in the previous example with `async` we can specify the execution policy for the algorithms:

- `sequenced_policy`
- `unsequenced_policy`
- `parallel_policy`
- `parallel_unsequenced_policy`

The first policy is straightforward and indicates that the algorithm should be executed in the calling thread. The operations within the algorithm are executed sequentially and are not interleaved.

The second policy, `unsequenced_policy` specifies that the algorithm should run in the same thread as the calling thread, similar to the first policy. However, individual operations can be interleaved or leverage vectorization, meaning there is no deterministic order in which individual operations will be executed. Therefore, the algorithm applied with `unsequenced_policy` must be safe for vectorization

`parallel_policy`, tries to execute the algorithm in parallel across different threads. The term ”tries” is used because the standard permits a fallback to `sequenced_policy` if parallel execution isn't feasible, such as due to a lack of resources.

The last policy combines aspects of the two previous ones, allowing for parallel execution of the algorithm across multiple threads while also permitting individual operations within the same thread to be interleaved.

```
std::vector<int> v = {1, 2, 3, 4, 5};
int sum = std::reduce(std::execution::par, v.begin(), v.end(), 0);
```

Code listing 3.9 Calculating an array sum using algorithms library.

The parallel algorithms in the C++ standard library provide a convenient way to utilize parallel computing. Internally, these algorithms can be implemented using the openMP library as is the case with GCC[24].

3.1.4 Coroutines

A significant addition to the C++ standard is the support for coroutines. Coroutines are functions that can be suspended at certain points and resumed later. To make this work, each coroutine is associated with its own coroutine state, which may be allocated either on the heap or on the stack, depending on the compiler and optimizations. The C++ standard does not dictate the specific allocation location for this state. The coroutine state maintains the necessary data to resume the coroutine at the suspension points.

To transform a standard function into a coroutine, it must utilize one of the new keywords:

- `co_await` - Potentially suspends the coroutine.
- `co_yield` - Allows the coroutine to produce a value and suspends the coroutine.
- `co_return` - Completes the coroutine, may return a value.

In addition to using the necessary keywords, the return object of a coroutine must be associated with another object known as the promise object. This promise object must implement a specific interface to qualify as a promise object for a coroutine. The interface typically includes methods to handle the initialization of the coroutine, return or yield of values or destruction of coroutine state.

This very brief summary outlines the basics required to transform a function into a coroutine. However, it's important to note that coroutines, as introduced in the C++ standard, are not immediately ready for use "out of the box." To effectively utilize coroutines, we need to implement several things ourselves. The primary goal of incorporating coroutines into the standard is to provide library authors with built-in support for coroutines. Allowing them to leverage this feature directly rather than depending on custom implementations, such as those provided by libraries like Boost.Coroutine[25].

3.2 OpenMP

OpenMP is a specification designed for parallel programming in C++ and Fortran. Being a specification, its implementation can vary across different compilers. While all implementations must adhere to the specification, the underlying mechanism can differ.

There are several implementations of the openMP standard:

- libgomp is a library that implements the OpenMP standard for the GCC compiler.
- libomp is an implementation within the LLVM project, typically used with the Clang compiler.
- Visual C++ offers its own implementation for the Windows.

Each of these implementations can differ in terms of functionality, performance, and even specific bugs. For instance, OpenMP 6.0 is expected to be released in 2024. At the time of writing this, Visual C++ supports only version 3.1. Both GCC and Clang support at least standard version 4.0, along with some functionalities from later versions.

OpenMP is based on the fork-join model. The program executes sequentially until it encounters a parallel region. Within this region, a pool of workers is created to execute the code in parallel. Users can use directives to define the behavior of the code in the parallel region. It's important to note that these directives can be ignored by the compiler if the necessary flag to compile with OpenMP, such as `-fopenmp` for GCC, is not specified. In such cases, the code would execute as a sequential program. This approach allows for easy switching between parallel and sequential program with just one flag.

```
int sum_array(array<int, 100>& arr){
  int sum = 0;
  #pragma omp parallel // (1)
  {
    #pragma omp for // (2)
    for (int i = 0; i < arr.size();i++){
      #pragma omp atomic // (3)
      sum += arr[i];
    }
  } // (4)
}
```

Code listing 3.10 Demonstration of OpenMP directives to sum an array.

This code, while obviously inefficient, serves as an example to demonstrate how OpenMP directives are used. We create a parallel region with a directive on line (1). Upon entering the

scope of this parallel region, a pool of workers is created. We can also specify the number of workers to be created using the directive `#pragma omp parallel num_threads(num)`.

On line (2), we specify the parallelization of the for loop using an OpenMP directive. OpenMP also provides a directive for making an operation atomic. The choice between using C++ atomics or OpenMP directives for atomic operations depends on the specific use case.

On line (4), we exit the parallel region, and the individual threads are joined back to the main thread. This is the mentioned fork-join model in OpenMP.

```
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < arr.size(); i++) {
    sum += array[i];
}
```

Code listing 3.11 A concise way of doing parallel for in OpenMP.

This example illustrates a simpler way of expressing a parallel for loop using OpenMP directives. This approach is more efficient as each worker maintains a local variable for accumulating the intermediate sum. These sums are then combined by all the workers. This method significantly reduces the synchronization overhead[26].

Another important thing to note with OpenMP is scheduling. The standard allows us to specify how can be the work scheduled. "The schedule clause specifies how iterations of associated loops of a work sharing loop construct are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team"[26].

In the previous example, the default scheduling option, `auto`, was used. This setting allows the compiler to decide which type of scheduling to use. It's reasonable to guess that static scheduling was used in this case. To explicitly specify static scheduling, one would include `schedule(static)` in the OpenMP directive. Static scheduling is a type of work-sharing strategy, where each worker is allocated a fixed number of chunks to process. Size of individual chunks can also be defined by user. While this approach is quite straightforward, it may have limitations if the workload varies significantly between individual array entries. In such cases, some threads might end up idly waiting for others to complete their work.

Another type of scheduling available in OpenMP is a `dynamic` scheduling. The key difference is that, as soon as a thread completes its assigned chunk of work, the next chunk is assigned to it. This dynamic allocation allows for a more evenly distributed workload. A similar option is `guided` scheduling, which operates in the same manner as the `dynamic` scheduling, but with the added feature of decreasing the chunk size over time.

3.2.1 Task

OpenMP also offers a `task` construct.

```
int fib(int n) {
  if (n < 2) return 1;
  int a, b;
  # pragma omp task
  a = fib(n - 1);
  # pragma omp task
  b = fib(n - 2);
  # pragma omp taskwait // (4)
  return a + b;
}
```

Code listing 3.12 Calculating a Fibonacci number using OpenMP task.

```
int main() {
  # pragma omp parallel // (1)
  {
    # pragma omp single // (2)
    {
      # pragma omp task // (3)
      fib (10);
    }
  }
}
```

Code listing 3.13 Starting calculation of the n-th Fibonacci number in OpenMP.

Creating a task in OpenMP is straightforward: a standard function call is simply encapsulated within the `omp task` directive, as shown on line (3). To execute this task, we first need to create a parallel region (1), similar to the previous example. Then, using the directive on line (2), we ensure that only one thread executes the task. Without the directive on line (2), every thread would execute this top-level function call.

Wrapping a function in the `omp task` directive does more than just schedule the function as a task for workers. It also allows the task to be suspended (4), and then resumed later once conditions are met.

We looked at the implementation of `omp task` in the libgomp[24] library. While we won't go into extensive details, given the library's complexity, a simple review can be insightful.

```
void (*local_fn) (void *);
...
do {
  struct gomp_team *team = thr->ts.team;
  struct gomp_task *task = thr->task;
  local_fn (local_data);
  gomp_team_barrier_wait_final (&team->barrier);
  gomp_finish_task (task);
  gomp_simple_barrier_wait (&pool->threads_dock);
  local_fn = thr->fn; // (1)
  local_data = thr->data; // (2)
  thr->fn = NULL;
} while (local_fn); // (3)
```

Code listing 3.14 Task to task control transfer in libgomp.

On lines (1) and (2), we observe that the function is assigned to the `local_fn` variable, which is a function pointer, and the data are loaded into the `local_data` pointer. This data pointer is then passed into `local_fn`. The loop continues until there is no more work for the thread, indicated by `local_fn` being NULL (3).

Looking deeper into the implementation, each `omp task` is essentially a struct that contains data and a function to be executed on this data, along with some additional variables for proper functioning and synchronization. Given that `omp taskwait` are known at compile time, the function can be divided into multiple `local_fn` functions, each with its corresponding `local_data`. This function splitting allows for the suspension and subsequent resumption of tasks.

OpenMP is a widely-used standard that allows for simple parallelism, such as the loop parallelism, as well as more abstract forms like the task parallelism. A notable feature of OpenMP is that, when code is written correctly, it can function in both parallel and sequential version. The standard continues to evolve, with new versions being developed, that add even more functionality and robustness to the library.

3.3 OneTBB

oneTBB is a library for multi-threading developed by Intel and formerly known as Threading Building Blocks (TBB). Nowadays it is part of the oneAPI and referred to as oneTBB. Among all the libraries discussed in this chapter, oneTBB stands out as the most robust one.

For loop parallelism, oneTBB provides `parallel_for` construct, enabling programmers to apply a function to each element of an array efficiently.

```
parallel_for(blocked_range, function_to_apply)
```

Code listing 3.15 `parallel_for` usage in oneTBB.

To utilize oneTBB's `parallel_for` construct, we must define `blocked_range` and the function to be applied,is a straightforward process. The `blocked_range<T>` class determines how the iteration space is partitioned among workers. While it's possible to create a custom `blocked_range` by implementing an interface, we won't go into details and use the general ones. Essentially, this class should implement several methods, such as `is_divisible`, to decide whether the iteration space can be further divided, and splitting constructors to divide the iteration space.

For common use cases, oneTBB provides predefined `blocked_range` classes. For simple arrays it's `blocked_range<T>` and `blocked_range2d<T>` for two-dimensional arrays. For more complex iteration spaces, we can utilize a custom implementation of `blocked_range`.

The second argument to `parallel_for` is a function, a functor, or a lambda expression we want to apply to the individual array elements. A simple functor example can look as follows :

```
class ApplyFoo {
 public:
  void operator()( const blocked_range<size_t>& r ) const {
  for( size_t i=r.begin(); i!=r.end(); ++i )
    Foo(a[i]);
  }
  ApplyFoo( float a[] ) : my_a(a) {} // constructor
};
```

Code listing 3.16 Simple function object used with `parallel_for`. [27]

When the `parallel_for` function is executed, the functor's `operator()` is applied to each element of the array. It's important to note that this function must accept a `blocked_range` as its argument. This function is also declared as `const` since it does not modify the array.

In the OpenMP section, we discussed various schedulers that controlled the size of individual chunks and their distribution among workers. Similarly, oneTBB also allows for control over chunking. The size of individual chunks can be specified in the `blocked_range` parameter. To enable manual control of chunk sizes we need to pass an additional argument `simple_partitioner`. This is because other partitioners use heuristics to determine size of individual chunks.

```
parallel_for(blocked_range<size_t>(
  0, vec.size(), G),
  [=](const blocked_range<size_t>& r) {
    for (size_t i = r.begin(); i != r.end(); ++i) {
      Foo(vec[i]);
  }
  }, simple_partitioner());
```

Code listing 3.17 Utilizing simple partitioner for custom grain size, determined by the `G` parameter.

For instance, consider `affinity_partitioner`. This partitioner aims to leverage cache affinity by scheduling operations on the same data to be executed by the same workers. For example, if we iterate over an array twice, applying a transformation during each iteration, the

`affinity_partitioner` will attempt to schedule the processing of the same array elements in both the first and second iterations to the same worker.

There is also `auto_partitioner`, which serves as the default partitioner in oneTBB, similarly to the `schedule(auto)` directive in OpenMP. Additionally, oneTBB offers `static_partitioner`, which is similar to the static scheduler in OpenMP.

For parallel reduction, oneTBB provides the `parallel_reduce` function. This is similar to `parallel_for`, however, there are several key differences.

```
parallel_reduce(blocked_range, reduction_functor)
```

Code listing 3.18 Reduction in oneTBB.

As with the previous example involving `parallel_for`, we must define `reduction_functor` for `parallel_reduce`. In addition to the `operator()`, we need to define a couple more functions: a splitting constructor and a join method. The process begins with the first worker splitting the iteration space in half and processing the first chunk while the second chunk can be processed by another worker if it 'steals' the chunk. After both chunks are processed, the parent worker uses the join function to combine the results. oneTBB guarantees that this operations works correctly even for non-commutative operations[27].

```
class SumFoo {
 public:
  int my_sum;
  void operator()( const blocked_range<size_t>& r ) {
          for( size_t i=r.begin(); i!=r.end(); ++i ) my_sum += my_a[i];
  }
  SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0) {}
  void join( const SumFoo& y ) {my_sum+=y.my_sum;}
  SumFoo(const array<int, 100>& a) : my_a(a), my_sum(0) {}
 private:
  const array<int, 100>& my_a;
};
```

Code listing 3.19 Functor for `parallel_reduce` in oneTBB.

This approach provides customization flexibility with the `parallel_reduce` operation in oneTBB. Although the use of lambda functions can make it more concise, we find it still a bit verbose for simple parallel reduce compared to OpenMP.

Next, we will explore task parallelism in oneTBB, which can be achieved through two main approaches: `task_group` and `parallel_invoke`. `task_group` offers greater flexibility, allowing for addition of tasks to the group dynamically during execution. On the other hand, `parallel_invoke` is suited for executing a fixed number of tasks that are known at compile-time.

```
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int x, y; task_group tg;
    tg.run([&] { x = fib(n - 1); });
    tg.run([&] { y = fib(n - 2); });
    tg.wait();
    return x + y;
  }
}
```

Code listing 3.20 The Fibonacci example using `task_group` in oneTBB.

We create a `task_group` and add two tasks to it. Invoking `wait` on the group suspends the current function and returns only after both tasks have completed. This mechanism is analogous to OpenMP's `#pragma omp taskwait`.

OneTBB also provides a construct known as `task_arena`, this is similar concept to the parallel region in OpenMP. It's possible to specify the number of workers, set NUMA affinity, and task priority for the `task_arena`. In the example discussed previously, we didn't explicitly define a `task_arena`[28]; this is because oneTBB automatically creates an implicit one. This feature grants more control over the execution of individual tasks.

OneTBB uses a work stealing scheduler[29], which aims to:

- "Enable as many threads as possible, by creating enough job, to achieve actual parallelism"
- "Preserve data locality to make a single thread execution more efficient"
- "Minimize both memory demands and cross-thread communication to reduce an overhead" [27]

Each worker in the work-stealing scheduler maintains its own deque (double-ended queue) and has the option to either pop a task from its own deque or steal a task from another worker's deque. This is the same concept we discussed in the Scheduling section.

Having covered task based and loop parallelism, next we will explore data flow parallelism.

3.3.1 Data flow parallelism

oneTBB offers robust support for graph parallelism. Main constructs used are nodes and edges (communication channels) between them. This form of parallelism is suited for problems that are more intuitively represented as graphs rather than through the traditional task paradigm. It is most commonly applied in data processing.

There are various types of function nodes. For instance, `function_node` is a single-input, single-output node that broadcasts its output to all successors. Another example, which is also a single-input, single-output node is `continue_node`. However, it not only accepts an input but also a message. The node begins its execution once the required number of messages is received, functioning similarly to a barrier. To create a `function_node`, we would proceed as follows:

```
function_node<typename Body> function_node(
    graph& g, size_t concurrency, Body body)
```

Code listing 3.21 Creation of `function_node` in oneTBB.

In this setup, `Body` refers to the type of the `body` object, which can be a user-defined function object. The `concurrency` parameter specifies the maximum number of concurrent executions of this node that can occur at any given time. Lastly, the `graph` parameter represents the graph to which the node belongs to.

Creating an edge is very simple, we call `make_edge(node1, node2)`.

In 3.22 we compute sum of an array using graph parallelism. We need to create two `partial_sum_node` nodes, which calculate partial sums from range `[l,r)`, and `final_sum_node`, which adds the partial sums together. In case of `partial_sum_node` we can set parallelism to unlimited, since it neither modifies the array nor accesses any global variables. The `final_sum_node` must have its parallelism set to 1 or use some form of synchronization to safely modify a global variable. After computing a partial sum, the `partial_sum_node` broadcasts it to the `final_sum_node`. This is similar process to parallel reduction.

OneTBB provides a wide array of options for graph parallelism like: various node types, buffering within nodes, different message-passing protocols and othres. Additionally, the framework allows for the modification of the graph at runtime by adding edges or nodes. However, removing nodes during execution is not advised, as this can potentially lead to data races[27].

```
int sum = 0; int k = 5;
vector<int> vec; // size of k * N
graph g;
using node_input = tuple<vector<int>&, int, int>;
function_node<node_input, int> partial_sum_node( g, unlimited, [](
  node_input& inputs) {
    auto vec = get<0>(inputs);
    int l = get<1>(inputs); int r = get<2>(inputs);
    int local_sum = 0;
    for (int i = l; i < r;i++)
    local_sum += vec[i];
    return local_sum;
} );
function_node<int, continue_msg> final_sum_node(
    g, 1, [&sum](const int partial_sum ) {
  sum += partial_sum;
});
make_edge(partial_sum_node, final_sum_node);
for (int i = 0; i < vec.size(); i+=k) {
  partial_sum_node.try_put(make_tuple(ref(vec), i, i + k));
}
g.wait_for_all();
```

Code listing 3.22 Calculating an array sum with graph parallelism.

We have explored the oneTBB library, covering aspects from basic loop parallelism and task parallelism to data flow parallelism. It is an all purpose library that offers multiple parallel paradigms, deep customization, parallel data structures and great performance, especially when compiled with Intel's compiler. While writing parallel algorithms in oneTBB might present more complexity compared to OpenMP, the library is well-supported by documentation. This documentation includes implementations of numerous parallel design patterns, which can significantly ease the use of oneTBB.

3.4 TaskfLow

Taskflow is an open-source library that focuses on task-based programming. It is similar in design to the oneTBB's data flow feature, where tasks are represented as nodes and dependencies as edges between them. Taskflow aims to offer performance on par with oneTBB while simplifying the use[30]. It is a header-only library, requiring compilers that support C++17 features. A simple example of its usage is as follows:

```
Executor executor; // (1)
Taskflow taskflow; // (2)
Task A = taskflow.emplace ([](){
  cout << "taskA" << endl ;}). name("A");
Task B = taskflow.emplace ([](){
  cout << "taskB" << endl ;}). name("B");
A.precede(B); // (3)
executor.run(taskflow).wait(); // (4)
```

Code listing 3.23 Task creation in Taskflow.

On line (1), we instantiate an executor object, which is responsible for creating the specified number of workers to execute individual `Taskflows`. The executor can operate in multiple modes. For instance, the `run` mode executes the task flow once until completion, `run_n` repeats the

execution of the entire `Taskflow` 'n' times, and `run_until` continues execution until a specified condition is met.

Next, we initialize a `Taskflow` object, which serves as the representation of the dependency graph. Individual tasks can be emplaced into the `Taskflow` object. On line (3), we create a dependency, or an edge, between two tasks, indicating that the task `A` must be completed before the task `B` can begin its execution. Finally, on line (4), we start executing the `Taskflow` using the executor. This execution is synchronous, meaning the program will wait until the entire graph has finished processing. Taskflow library also offers an asynchronous type of execution.

Individual tasks in Taskflow act as wrappers around callable objects. A `Task` can be created using a lambda expression, a `std::function`, or by defining a custom functor. Taskflow supports several types of tasks, each serving a different purpose:

- static task
- dynamic task
- condition task
- multi-condition task
- module task

A static task is created by passing a callable object to `emplace(callable)`, which is of type `function<void()>`. In the earlier provided example, both tasks are the static type.

For creating a dynamic task, a callable of type `function<void(Subflow&)>` is required. The provided `Subflow` parameter allows tasks to dynamically introduce new tasks into the current `Taskflow`.

```
Task A = taskflow.emplace([](){}).name("A");
Task B = taskflow.emplace([] (Subflow& subflow) {
  Task B1 = subflow.emplace([](){}).name("B1");
  Task B2 = subflow.emplace([](){}).name("B2");
  Task B3 = subflow.emplace([](){}).name("B3");
  B3.succeed(B1, B2);
}).name("B");
Task C = taskflow.emplace([](){}).name("C");
C.succeed(A, B);
```

Code listing 3.24 Utilizing `subflow`, which allows dynamically adding tasks into itself.

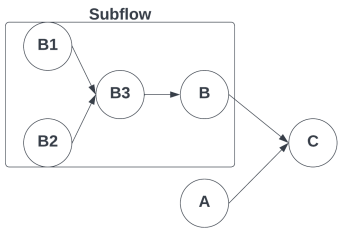

Figure 3.1 Visual representation of `subflow`.

Code listing 3.24 dynamically creates new tasks by constructing an additional task dependency graph in the node B. The node B is considered complete only after tasks `B1`, `B2`, and `B3` have finished their execution. Additionally, tasks `B1` and `B2` must be completed prior to starting execution of the node `B3`.

Condition task represent another task category within Taskflow, enabling conditional execution where the flow of the graph is determined at runtime based on specific conditions. To create a condition task, a callable that returns an integer, `function<int()>`, must be passed into the `emplace` function. The flow of the task graph is determined based on the returned value, allowing for dynamic path selection during runtime.

```
Taskflow taskflow;
auto [init, cond, A, B, C] = taskflow.emplace(
  [] () {},
  [] () {return rand % 3;},
  [] () {std << "A" << endl;},
  [] () {std << "B" << endl;},
  [] () {std << "C" << endl;}
);
A.precede(cond);
cond.precede(A, B, C);
```

Code listing 3.25 Demonstration of condition task in Taskflow.

By calling `cond.precede`, we create a conditional connection, meaning the subsequent path is chosen based on the condition defined within the `cond` task. Given three successor tasks, each node gets assigned a number from 0 to 3, based on the order they were passed to the `precede` function. The `cond` task selects one of these successors by generating a random number, thereby determining the flow dynamically. The `init` is a static task and is necessary because `Taskflow` cannot start a graph with a conditional task.

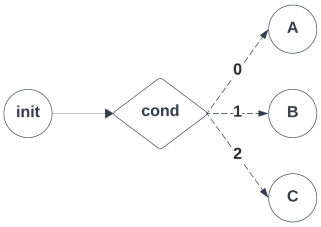

Figure 3.2 Visual representation of a conditional task in TaskFlow.

The multi-condition task extends the concept of the conditional task. Unlike a standard conditional task that chooses a single path, a multi-condition task can return a vector of integers, enabling the selection of multiple paths simultaneously. To define such a task, the callable should be of type `function<SmallVector<int>()>`, where `SmallVector` is Taskflow's implementation of the the small vector.

The final type of task we'll discuss is the module task. This task is created from an existing `Taskflow` by calling `Taskflow.composed_of(Taskflow)`. By transforming an entire `Taskflow` into a single `Task`, this approach significantly enhances reusability, allowing entire `Taskflow` structures to be integrated as modular components within larger `Taskflows`.

```
Taskflow t1;
auto [B1, B2, B3] = t1.emplace (
  [](){},
  [](){},
  [](){}
);
B3.succeed(B1, B2);
Taskflow t2;
Task A = t2.emplace([](){});
Task B = t2.composed_of(t1);
Task C = t2.emplace([](){});
C.succeed(A, B);
```

Code listing 3.26 Module task in Taskflow allows for clearer task composition, by transforming a whole `taskflow` into a single task.

We've covered basic building block of the Taskflow library. A different notable feature of Taskflow is its integration with CUDA, enabling the seamless combination of parallel computing on both CPU and GPU.

To create a task intended for GPU execution, we can use `cudaFlow` within the Taskflow library by defining a task in a `Taskflow` that takes a `cudaFlow` as an argument. This task will then be executed on the GPU. Within this GPU-specific task, we can define additional `cudaTasks` to specify the GPU operations.

For solving basic parallel computing problems, such as performing a parallel reduction to sum an array, Taskflow provides predefined constructs.

```
Task task2 = taskflow.reduce(
  first, last, init, [] (auto a, auto b) { return a + b; }
);
```

Code listing 3.27 Parallel reduction in Taskflow utilizing a built-in function.

Taskflow also includes other predefined parallel algorithms, such as `taskflow.for_each` and `taskflow.sort`. Similar to previous examples, these algorithms generate static tasks that can be incorporated into a `Taskflow`.

Calculating the n-th Fibonacci number with Taskflow introduces complexity due to the dynamic creation of tasks, this implies the need of `Subflow`. Each task spawns two additional tasks. The `Subflow` object is passed down this branching structure, allowing each child task to append further tasks to the flow.

```
int spawn(int n, Subflow& sbf) {
  if (n < 2) return n;
  int res1, res2;
  sbf.emplace([&res1, n] (Subflow& sbf_n_1) {res1 = spawn(n - 1, sbf_n_1); })
     .name(std::to_string(n-1));
  sbf.emplace([&res2, n] (Subflow& sbf_n_2) {res2 = spawn(n - 2, sbf_n_2); })
     .name(to_string(n-2));
  sbf.join();
  return res1 + res2;
}
```

Code listing 3.28 Calculating the n-th Fibonacci number in Taskflow. [31]

Taskflow's primary objective is to simplify the use of graph parallelism while delivering good performance. Constructing individual `Taskflows` is designed to be intuitive. The library also provides high-level constructs for common parallel patterns, such as loop parallelism. Additionally, its seamless integration with CUDA further extends its capabilities. From our perspective, Taskflow is promising tool for projects involving graph parallelism.

3.5 CppCoro

Another library is CppCoro, shorthand for cpp coroutines, which describes itself as follows: "This library is an experimental library that is exploring the space of high-performance, scalable asynchronous programming abstractions that can be built on top of the C++ coroutines proposal"[32].

From the citation is obvious, it was created while C++ coroutines were still in proposal stage. From the date on GitHub it was first published around year 2017. It is not finished in a sense that some features works only on the Windows platform[32].

As name suggests it is built upon C++ coroutines. The basic building block is `task<T>`. Which is lazily evaluated, meaning it is not executed upon creation, but after it is `co_await`-ed.

```
task<int> foo(){ co_return 42; }
```

Code listing 3.29 Simple task representation with `task<T>` in CppCoro.

It is similar to the ordinary function, except the returned value is of type `task<T>`, which is an object users can interact with. It can be `co_await`-ed or used to start the execution of the coroutine.

There are multiple flavors of individual tasks:

- `task<T>`
- `shared_task<T>`
- `generator<T>`
- `recursive_generator<T>`
- `async_generator<T>`

`shared_task<T>` can be awaited from multiple coroutines. Meaning one task has to start the execution of a `shared_task<T>` and wait for its resumption. Other tasks can also wait for completion of the `shared_task<T>` or continue synchronously if the `shared_task<T>` has already complete its execution.

Another type is `generator<T>` which as name suggests is a generator. Meaning it uses `co_yield` to produce values. It also has a method `generator<T>::begin()` that returns an iterator and allows for the values to be iterated through.

```
generator<int> foo(int start) {
  int a = start;
  while (a < 100)
    co_yield a++;
}
int main(){
  for (int i : foo(10)) {
    ...
  }
}
```

Code listing 3.30 Generator in CppCoro.

Next type of a coroutine is `recursive_generator<T>` which is used for getting values from the nested generators. By calling `operator++` on the iterator of `nested_generator<T>` it yields values from the most nested generator.

The last type is `async_generator<T>` it generates values as any other generator, however, it can also `co_await` other tasks inside its body.

Additionally to the different types of tasks, CppCoro offers different types of awaitables. Typically concepts that are common in parallel computing, but wrapped inside a coroutine, so they can be used with another coroutines by simply using key words like `co_await`, `co_return` and `co_yield`.

For example `async_mutex`, which can be `co_await`-ed from within a coroutine. It is a non blocking mutex, which allows thread to do another work and resume the coroutine later, once the lock has been acquired. Another interesting awaitable is `single_consumer_event` which allows a single coroutine wait for an event. Once the event is set, the coroutine can be resumed. There are many more simple synchronization primitives.

CppCoro does not support any constructs for loop parallelism. Doing an example with the parallel reduce, would be cumbersome. We would need to split the array, for each split create a task to calculate intermediate sums and then combine these sums together.

On contrary, calculating the n-th Fibonacci number is straightforward:

```
cppcoro::task<int> fibonacci(int n) {
  if (n <= 1) {
    co_return n;
  } else {
    auto a = co_await fibonacci(n - 1);
    auto b = co_await fibonacci(n - 2);
    co_return a + b;
  }
}
```

Code listing 3.31 Calculating the n-th Fibonacci number using CppCoro.

CppCoro also offers its own implementation of thread pool. It works with two queues. Each worker has its own queue for work it spawns. This queue is LIFO. The second queue is a global queue shared by all workers, and is used for tasks that are put into the thread pool from remote threads. This queue is FIFO. Once each worker finds its queue empty it can look into the global queue for more work.

Another notable feature CppCoro offers are I/O constructs for work with files in asynchronous manner, especially support for sockets and networking. However, this features are only supported on windows.

From the Github issues it is clear that his library has been abandoned since the year 2020.

3.6 HPX

High Performance ParalleX, abbreviated as HPX, is a C++ library with a clear goal: "The goal of HPX is to create a high-quality, freely available, open-source implementation of a new programming model tailored for conventional systems. This includes classic Linux-based Beowulf clusters or multi-socket, highly parallel SMP nodes."[33].

HPX introduces the concept of nodes, enabling programmers to develop applications for both single-node and multi-node environments. In multi-node configurations, HPX manages the communication between individual nodes, facilitating easy deployment on cluster architectures. It implements the ParalleX execution model[34], which provides a full runtime environment for the execution of individual tasks.

HPX is a complex library. It implements or extends its own versions of C++ parallel constructs such as barriers, mutexes, semaphores, and more. It also offers a support for GPU computations.

Given the extensive features of HPX, particularly its full runtime environment for scaling applications from single node to clusters, we will not cover it further in this context. Our focus has been on libraries that use single-node parallelism, which is a focus this thesis.

3.7 Summary

In this chapter, we've reviewed various libraries and parallel constructs provided by the C++ standard library. It's evident that the C++ standard library lacks higher-level constructs such as threads, tasks, or concurrent data structures. Furthermore, its support for asynchronous programming is quite limited. However, the C++ standard library does provide parallel algorithms within the `<algorithm>` header. In certain compilers, these are implemented using the OpenMP library.

The OpenMP library additionally provides support for simple tasking. For more complex parallel programming needs we can use oneTBB with its concurrent structures, graph parallelism, and task-based parallelism. For even more robust graph parallelism capabilities, the Taskflow library is an excellent alternative. Libraries such as HPX provide complete runtime environment, enabling building parallel programs from single node loop parallelism to cluster-wide computations.

A common trait among these libraries is their aim to offer a more abstract and user-friendly approach to parallel programming. At the same time, they provide customization options for parallel execution, such as the use of various types of schedulers.

With introduction of coroutines, many new libraries were created, including CppCoro, which we have covered in this chapter. Direct utilization of coroutines for parallel programming is not straightforward; users must either depend on third-party libraries or develop their own solutions using coroutines. Additionally, there is a proposal to incorporate more abstract constructs for asynchronous programming into C++, with seamless integration with coroutines[35]. This would allow for more cohesive interaction with parallel and asynchronous programming.

While the C++ standard library provides basic support for parallel algorithms, it does not support more complex parallel constructs. Fortunately, this limitation can be overcome utilizing one of many third party libraries.

Chapter 4
Design

In the previous chapter, we explored multiple libraries for parallel computing in C++ and discussed the support for parallel computing within the C++ standard. Given the support for loop based parallelism in the C++ algorithms library, we conclude that creating a new library specifically for loop parallelism might not be beneficial.

This leads us to the second option: developing a library dedicated to task based parallelism. C++ provides partial support for this concept in its standard library. By "partial support," we refer to the availability of constructs such as `std::packaged_task` and `std::async`, which enable launching a tasks on a new thread. Nonetheless, these mechanisms come with their own set of limitations:

- Scaling this approach requires spawning a new thread for each `std::packaged_task`. There isn't a standard thread pool implementation in C++. We must either implement one ourselves or use a third-party solution.
- Another issue is that `packaged_task` cannot be paused and resume later. For example in our Fibonacci example 3.12, the parent level task needs to wait for its child tasks to complete. This is not feasible with `std::packaged_task`.

To address these limitations, we could utilize coroutines, which support suspension and resumption at specific points. This would solve the problem of waiting for child tasks to complete. Coroutines also enable us to transfer tasks between threads without requiring synchronization, allowing a coroutine to be paused in one thread and resumed in another.

For our task implementation, we've chosen to use C++ coroutines. In a previous chapter, we discussed a similar library called CppCoro. However, CppCoro is no longer under active development and is considered experimental. We believe it would be interesting to explore the features introduced in C++20 to construct a library focused on task based parallelism.

With task based parallelism, we also have the benefit of being able to implement loop based parallel algorithms by dividing them into tasks.

4.1 Task

Our Task implementation should encapsulate all the necessary requirements to enable users to easily create coroutines.

The user's only responsibility would be to use one of the required keywords to transform a function into a coroutine. This means our Task implementation must also include its own promise object.

```
our_task foo() {
  co_return;
}
```

Code listing 4.1 Transformation of a regular function into a coroutine.

Another thing to consider, is the return value. It's important to note that the value cannot be returned directly from the coroutine itself; instead, it must be returned through our Task object, if the coroutine does return a value. Since we want `our_task` to be a generic type for all type of coroutines we need to use template metaprogramming.

Users will need to specify the type of value the task will return. The resulting task type would be defined as follows:

```
our_task<RETURN_TYPE> foo() {
  co_return;
}
```

Code listing 4.2 Task needs to be templated for a desired return value.

An essential feature of our library should be the capability to wait for other tasks to complete without resorting to busy waiting. To make this feasible we need to suspend the current execution with an new keyword `co_await` and resume it once child tasks are completed. The semantics can look as follows:

```
our_task<RETURN_TYPE> foo() {
  co_await wait_for_tasks(task1, task2 .... );
  co_return;
}
```

Code listing 4.3 Design of task waiting.

4.2 Thread pool

Now that we have an outline how to implement a task, we can proceed to the second part, which is the thread pool. As demonstrated, C++ enables the spawning of individual threads directly with the `std::thread` object, which will serve as the foundation for our thread pool. Each task will be executed on a thread, either on the main thread or on a thread spawned by `std::thread`.

We can create a class called `ThreadPool` that acts an abstraction over individual threads. This class would also have ownership of the threads, ensuring that when a ThreadPool object is destroyed, the individual threads are also terminated.

Encapsulating individual tasks with coroutines and spawning threads with `std::thread`, outlines a straightforward approach. Another important design consideration is the scheduling process, specifically how individual tasks are scheduled between threads.

We have two viable options for scheduling: a work-stealing scheduler or a work-sharing scheduler. Implementing these schedulers can be challenging. A thread must retrieve its tasks from the scheduler, suggesting that tasks need to be stored within the scheduler itself. However, C++ does not provide built-in support for concurrent data structures. The alternatives are to either develop such structures ourselves or to use `std::mutex` with `std::queue`, which could be inefficient. Our goal is to implement one or both scheduling strategies with minimal overhead, ideally through the use of atomic variables.

Using atomic variables can become challenging, particularly when we move away from memory ordering that guarantees sequential consistency. Our options are: using a third-party library or creating our own version of a known concurrent structure. Designing an efficient lock-free structure using relaxed atomics is not only difficult but also beyond the scope of this thesis.

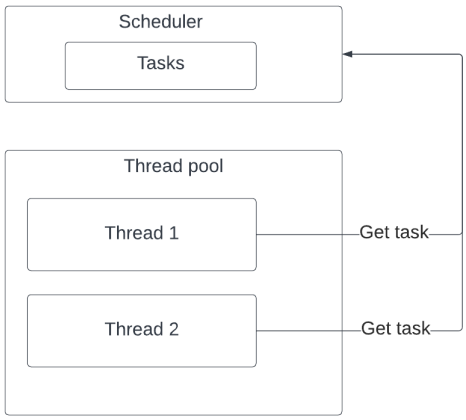

Figure 4.1 A possible implementation of work-sharing.

In the diagram above, we illustrate a work-sharing scenario in which each thread retrieves a task from the scheduler. Concurrent structures are necessary for both work-stealing and work-sharing types of scheduling for efficient scheduling.

4.3 Launching parallel execution

Another design consideration involves the method of initiating concurrent execution and retrieving results. The parallel execution must be initiated from the main thread and can be done in one of two ways: either by launching the parallel execution and waiting for it to complete before retrieving the result, or by adopting a pattern similar to `std::future`, where a future object is returned from the call. The latter approach does not block the calling thread, allowing it to continue execution while awaiting the result.

We have chosen to implement the blocking mechanism, as asynchronous operations are not essential for our implementation. A concern might be whether this approach wastes resources. However, it doesn't, since we won't be using busy blocking. `std::condition_variable` can be utilized, which allows the main thread to yield and is only awakened once the parallel execution is complete.

```
our_task<RETURN_VALUE> foo() {
  co_return;
}

int main() {
  our_task<RETURN_VALUE> task = foo(); // (1)
  start_parallel_execution(task); // (2)
  RETURN_VALUE value = task.get_value() // (3)
}
```

Code listing 4.4 Launching a task and retrieving a value from it.

In the example above, a task object is created on line (1). On line (2), a task is launched to execute in parallel. In case we implement multiple schedulers, this method would be a proper place for users to specify the desired type of scheduler. Finally, on line (3), the result is retrieved from the task object.

Even with an asynchronous approach, we would require a mechanism to wait for the parallel execution to finish if the main thread attempts to access the result before the parallel tasks are

completed.

Once the main thread is notified, the result can be retrieved. This raises the question of where the result should be stored. Storing the result within the task itself is one option, making the calling thread responsible for managing the task's lifetime. Each thread must ensure the task object remains valid beyond the completion of the parallel execution to successfully retrieve the result.

Chapter 5
Implementation

5.1 Coroutines

C++20 introduced a new feature called Coroutines. The main difference between standard functions and coroutines is that coroutines can be suspended and resumed at specific points. This functionality is achieved by storing the execution state of a coroutine at a suspension point. Typically, this state is stored on the heap, though compiler can optimize in certain cases by storing the state on the stack. C++20 standard does not specify where the coroutine state should be stored. The process of restoring involves loading the execution context from the coroutine state and resuming the function.

Each coroutine is associated with its own promise type, which should not be confused with `std::promise`. The promise of a coroutine determines its behavior and can be utilized to retrieve a return value. Promises will be described in detail later in this chapter. This is a high-level overview of coroutines. Before we delve into specifics of coroutines, it's essential to understand operator `co_await`.

5.1.1 `co_await`

Operator `co_await` is a new keyword and it can only be used inside a coroutine. Its purpose, as the name suggests, is to await an object. Why would we want to await an object ? As we mentioned earlier, coroutines can be suspended(paused) at suspension points. If we need to wait for another computation to finish, we can suspend the current coroutine and do some useful work. Then, we can resume this suspended coroutine once the computation is complete.

Awaited objects must implement an awaiter interface or support the `co_await` operator that returns such an object.

```
struct awaiter_object {
  bool await_ready();
  <return_type> await_suspend(coroutine_handle);
  <return_type> await_resume();
}
```

Code listing 5.1 Awaiter object interface.

Purpose of `await_ready` method is an optimization. If this method returns false, the current coroutine is suspended; if it returns true, the coroutine continues in its execution. We can make this method constant for certain type of awaiters, for example to always pause the current coroutine.

Once `awaiter_object` returns false from `await_ready`, the `await_suspend` method is called. In the body of `await_suspend` we can define what should happen, once the coroutine is suspended. It is important to note that once we are inside the `await_suspend` method, the coroutine is already suspended. The coroutine handle of the currently suspended coroutine is passed as an argument. We will discuss coroutine handles later in this chapter.

`await_suspend` can return multiple return types, as supported by C++20 standard[17]:

- `void`, control is returned to the caller of a coroutine, the coroutine remains suspended
- `bool`, if we return `true`, coroutine remains suspended and we resume the caller (same as `void`), if we return `false`, current coroutine is resumed
- `coroutine_handle` returning handle from `await_suspend`, resumes suspended coroutine that belongs to said handle

`await_resume` is invoked once the coroutine is either resumed from a suspended state, or was never suspended. The `return_type` is result of the whole `co_await` expression. For instance, we can return `void`, if we do not need any return value.

C++20 introduces two trivial awaitable types: `std::suspend_always` and `std::suspend_never`. The former always suspends a coroutine, while the latter never suspends. Neither of these types produce any value, as their `await_resume` function returns `void`.

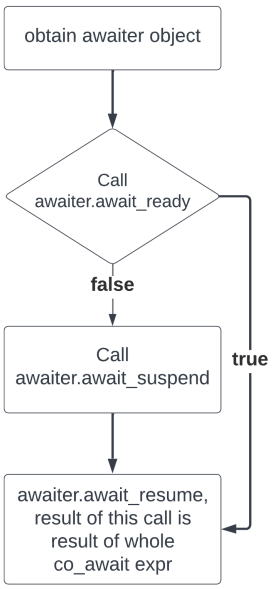

Figure 5.1 Awaiter flow diagram.

5.1.2 Coroutine handle

Each coroutine is associated with `std::coroutine_handle<promise_type>`. These handles enable us to manipulate coroutines and their states. For example resuming coroutines, destroying coroutines, manipulating promise object. The C++20 standard recognizes two types of coroutine handles with respect to promise types: `std::coroutine_handle<promise_type>` for coroutine handles with a specified promise type, and `std::coroutine_handle<void>` which is type erased version[17]. Handle `std::coroutine_handle<promise_type>` can be converted to `std::coroutine_handle<void>`.

```
template <PROMISE_TYPE>
struct coroutine_handle<PROMISE_TYPE>{
  constexpr void* address() const noexcept;
  static constexpr coroutine_handle from_address(void* addr);
  void operator()() const;
  void resume() const;
  void destroy() const;
  static coroutine_handle from_promise(Promise&);
}
```

Code listing 5.2 Coroutine handle interface.

In this pseudocode, we present only the essential methods needed to explain how coroutine handles function. For a complete specification, readers can refer to standard[17].

As mentioned earlier, each coroutine is associated with its own promise object and state. The promise object is stored within the coroutine's state, which is why we can get coroutine handle from promise by using `from_promise`. In case we do not need to work with the promise object, the void version can be used `std::coroutine_handle<>`.

Another notable functionality includes resuming a coroutine from its suspended state and destroying the coroutine state.

5.1.3 Promise object

By explaining how `co_await` operator and coroutine handle works, we can examine a coroutine promise object. The promise object is something that defines the behavior of a coroutine. We will look at the promise object method by method since each of this method is crucial to understand coroutine behavior.

```
struct promise_type {
  get_return_object () // mandatory
  awaiter_object initial_suspend () // mandatory
  awaiter_object final_suspend noexcept() // mandatory
  unhandled_exception() // mandatory
  return_value() // mandatory if returning value
  yield_value() // mandatory if we use yield keyword
  return_void() // only when we do not return value from coroutine
}
```

Code listing 5.3 Promise object interface.

The `get_return_object` method is called at most once. It is used to create a return object that the coroutine returns when it is suspended for the first time. The return object is an object that users, us programmers, use to interact with a coroutine.

`initial_suspend` is called immediately following `get_return_object`. It must return an awaiter object, so the result can be `co_await`-ed. For example, if `std::suspend_always` is returned, the coroutine will always suspend at this point. Alternatively, a custom awaiter can be defined.

The `final_suspend` method is called when a coroutine reaches the end of its execution, either through a return statement or by falling off the end of the coroutine. It is important to note that this method needs to be `noexcept`, a detail that will be made clearer once we discuss the flow of coroutines. The name of the function might be somewhat misleading. When this method is called, the coroutine is not immediately suspended. It is only suspended after the result of `final_suspend` is `co_await`-ed, and that awaiter can suspend the coroutine. This same principle applies to `initial_suspend`.

`unhandled_exception` is called once a coroutine throws an exception. This allows us to process the exception as needed, after which `final_suspend` is called.

Depending on how a coroutine returns a value, we need to implement one of the return methods: either `return_value`, `yield_value`, or `return_void`.

The coroutine state includes a promise object, a copy of the parameters passed to the coroutine, and the necessary data to resume the coroutine from a suspension point. The standard does not specify where the coroutine state should be allocated; this depends on the compiler and optimizations used. The most common approach is using dynamically-allocated storage. With a coroutine handle, we can access this state object.

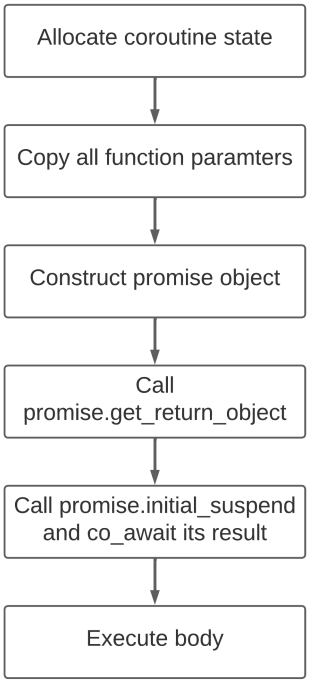

Figure 5.2 Coroutine initialization process.

5.1.4 Symmetric transfer

In our discussion about the awaiter object, we noted that a coroutine handle can be returned from the `await_suspend` method. This is called symmetric transfer, a concept proposed by Gor Nishanov[36]. It enables coroutines to transfer control amongst themselves without need for additional function calls.

Without symmetric transfer we would need to suspend a coroutine, obtain a coroutine handle, and then resume another coroutine by calling the `coroutine_handle<>.resume` method. Doing this within `await_suspend` would keep the `await_suspend` frame on the stack and allocate a new stack frame for a coroutine by using the resume method on the handle[37]. Symmetric transfer avoids this additional stack space reservation. Returning a coroutine handle from `await_suspend` resumes another coroutine without retaining the `await_suspend` stack frame. Consequently, the frames of the old coroutine and `await_suspend` are removed from the stack, and the new coroutine is resumed by 'loading' it onto the stack.

With this proposal, a new type of coroutine was introduced into standard : `noop_coroutine`, along with its associated `noop_coroutine_handle` and `noop_coroutine_promise`. The `noop_-`

`coroutine_handle` is template specialization `coroutine_handle<noop_coroutine_promise>`. A `noop_coroutine` is a coroutine that has no observable side effects, meaning operations like `handle.destroy` or `handle.resume` have no effect. Why is this useful ? Returning such a coroutine handle from `await_suspend` , indicates we do not want to resume another coroutine, but to return control to the caller.

With the implementation of symmetric transfer and `noop_coroutine`, we gain an efficient method to resume a coroutine from a different coroutine, eliminating the need for consuming additional stack space.

5.1.5 Coroutine flow

Up until this point, we have discussed the necessary aspects to understand the functionality of coroutines. Now we will examine coroutine flow and show a simple coroutine.

```
int fun(int a, int b) {
  return a + b;
}

return_object cor(int a, int b) {
  co_return a + b;
}
```

Code listing 5.4 Coroutine to function transformation.

This example illustrates the difference between a standard function and a coroutine. We must change the `return` keyword to `co_return`, and `int` to `return_object`. The `return_object` object is constructed as shown in the Coroutine initialization process. diagram. It is not a built-in type of C++ standard library, so we must either define one ourselves or use a library that provides such an object. We will be implementing our own return object in the Task section.

Having demonstrated a simple coroutine, let's now explore how it is executed:

```
promise-type promise promise-constructor-arguments ;
try {
  co_await promise .initial_suspend() ;
  function-body
} catch ( ... ) {
  if (!initial-await-resume-called )
    throw ;
  promise .unhandled_exception() ;
}
final-suspend :
  co_await promise .final_suspend() ;
```

Code listing 5.5 Coroutine execution context.

This code snippet from the C++20 standard illustrates how coroutines are executed. Initially, the promise object is constructed. If the construction of the promise type throws an exception, it must be caught within a try-catch block, or alternatively, the construction of the promise type can be made non-throwable.

Next, we enter a try-catch block where `promise.initial_suspend` is called, and we `co_await` the result. This call may either suspend the coroutine, requiring it to be resumed later, or allow the execution to continue with the function body.

If any exception is thrown within this try-catch block, which also includes coroutine's body, `promise.unhandled_exception` is called. Here, we can decide how we want to handle the exception, either by rethrowing it or processing it inside the function call. The only exception to

this is when `await_resume` has not yet been called. For details on when `await_resume` is called, refer to the Awaiter flow diagram. diagram.

After the coroutine's body finishes, the `promise.final_suspend` is called, and the result is `co_await`-ed. This follows the same principle as in `initial_suspend`, with the key exception being that the `final_suspend` method must be non-throwing.

We have outlined the core mechanisms of coroutines to clearly illustrate their functionality and our utilization of them in our implementation. For a more detailed reference, please consult the C++20 standard[17].

5.2 Task

In this section, we describe our task implementation that utilizes coroutines. Our goal is to create a task that can be easily swapped between threads and can be awaited synchronously and asynchronously. We need:

- Return an object from a coroutine that user can interact with
- Custom awaiters to define behavior at suspension points
- A promise object that defines behavior of a coroutine

First, let's start with the return object. This object should be returned to the user when a coroutine is created. It should enable starting the coroutine, destroying the coroutine state, and retrieving a return value from the coroutine. To achieve this, the return object must have access to the coroutine handle of the said coroutine.

We decided to make this return object an RAII object that is the sole owner of the coroutine. It cannot be copied, only moved. Furthermore, when this object is destroyed, the coroutine state is also destroyed.

Another important feature is the ability to `co_await` a coroutine. In an instance where one coroutine waits for another, it needs to call `co_await` on the return object. For that to be possible, the return object must implement methods `await_ready`, `await_suspend`, and `await_resume`, effectively making it an awaiter. Alternatively, the return object can support the `co_await` operator. We opted for the latter approach, as we find it to be a cleaner solution. Simple pseudo-code for our class can be:

```
class Task {
 public:
  using promise_type = OurPromiseObject;
  Task(coroutine_handle<promise_type> handle);
  OurAwaiter operator co_await();
  void resume();
  ~Task();
  ...
 private:
  coroutine_handle handle_;
}
```

Code listing 5.6 Task implementation, shortened for brevity.

What should occur when we `co_await` the return object? It should suspend the current coroutine and begin executing the one we `co_await`. Once the awaited coroutine finishes its execution, control should resume in the previously suspended coroutine. This behavior dictates how the awaiter returned from the `co_await` operator should behave. To achieve this, we set a continuation for the awaited coroutine, so it knows where to resume execution upon completion.

```
Task bar() {...}
Task foo() {co_await bar;}
```

Code listing 5.7 Example of awaiting another coroutine.

When we await `bar`, it returns an awaiter object that suspends the current coroutine `foo`, sets `bar`'s continuation to `foo`, and resumes `bar`. After `bar` finishes its execution, the suspended coroutine `foo` is resumed. In our implementation, we use `ContinuationSetAwaiter` to achieve this behavior. If a `Task` is `co_await`ed, it returns `ContinuationSetAwaiter` from its `operator co_await()` method.

```
class ContinuationSetAwaiter {
 public:
  ContinuationSetAwaiter(coroutine_handle to_resume)
  : to_resume_(to_resume){}
  coroutine_handle<> await_suspend(
      coroutine_handle<> currently_suspended) {
    to_resume_.promise().set_continuation(currently_suspended);
    return to_resume_;
  }
  ...
private:
  coroutine_handle<> to_resume_;
}
```

Code listing 5.8 ContinuationSetAwaiter implementation, shortened for brevity.

At first glance this behavior may seem overly complicated for simple control transfer. The same thing can be done with plain functions. However, the difference is obvious in multithreaded environment, where different `Task`s can be executed on different threads. For example, `foo` can be executed on one thread and resumed later on another thread after `bar` completes.

The final piece necessary for a functioning `Task` is the associated promise object. As mentioned in the previous section, the promise object defines what happens at the initial and final suspend points. We aim for `Task` to be a lazily evaluated coroutine, which means the `initial_suspend()` method should always suspend; therefore, we can return `suspend_always`. For the `final_suspend` method, we either resume a `Task` awaiting our completion or return control to the coroutine's caller. To resume a suspended coroutine, we need another awaiter and must pass it the handle of the suspended coroutine. This handle is stored in the current coroutine's state, set during the call to the `co_await` operator that created a `ContinuationSetAwaiter`. To return control to the caller, we can return an awaiter with `noop_coroutine_handle<>`; as the standard specifies[17], this has no side effects and effectively returns control to the caller.

Another key responsibility of the promise object is to store the return value of the coroutine. To make storage and retrieval of the return object work, we must implement additional methods. Specifically, to store the return value, we need to define `return_value()` as specified by the standard[17]. For retrieving the return value, we could either overload an operator or create a new method.

The promise object also manages exceptions. Essentially, it can either rethrow exceptions immediately or store them for later retrieval and handling by the user. In our design, we chose immediate rethrowing: any exception that occurs within the coroutine is immediately rethrown. This means users are responsible for handling exceptions that arise during coroutine execution.

Since coroutines can return different types of values or none, we need to template the promise object. In our implementation we associate `SimplePromise` with `Task` and `ReturnValue` of the promise object is determiend by the return value of `Task`. Following code demonstrates how `SimplePromise` can be implemented.

```
Template<typename ReturnValue>
class SimplePromise {
 public:
  suspend_always initial_suspend() { return {}; }
  ResumeAwaiter final_suspend() noexcept {
        if (continuation_ == nullptr}
      return ResumeAwaiter{noop_coroutine()};
    return ResumeAwaiter{continuation_};
  }
  std::suspend_always initial_suspend() { return {}; }
  template <typename U>
  void return_value(U&& v) {
    // construct return value for coroutine
    new (return_value_) ReturnValue(forward<U>(v));
  }
  ...
 private:
  coroutine_handle continuation_;
  alignas(alignof(ReturnValue)) char return_value_(sizeof(ReturnValue)];
}
```

Code listing 5.9 Simple promise implementaion, shortened for brewity.

5.3 WaitForTasksAwaiter

In this section, we will describe a new type of an awaiter called `WaitForTasksAwaiter`. In parallel computing, it's often necessary to wait for other tasks to complete before continuing the current task. With the simple `Task` that we introduced in the previous section, this is not doable. We require a mechanism that pauses our current task and resumes it once all child tasks have finished their execution. Importantly, we want this to happen in parallel, allowing all tasks to be executed on different threads. This implies a need for some form of synchronization.

```
Task<void> foo{...}
Task<void> bar{
  ...
  co_await wait_for_tasks(foo1, foo2)
  ...
}
```

Code listing 5.10 Awaiting another tasks for completion.

We want to pause the currently running coroutine and start executing other tasks on the same thread. This can be achieved using the `co_await` operator with an awaiter or some other awaitable object that supports the `co_await` operator. We have chosen the latter approach. We create an object that supports the `co_await` operator and returns an awaiter.

The returned awaiter should:

- suspend the current coroutine
- schedule tasks for execution

Suspending the current coroutine is straightforward: we return `false` from `await_ready`. This action moves the execution into the `await_suspend` method, where the current coroutine is suspended. Within the `await_suspend` method, we can schedule individual child tasks for execution. This scheduling is done by communicating with a scheduler. Therefore, our awaitable object requires a reference or a pointer to the scheduler.

```
struct WaitForTasksAwaiter{
 public:
  constexpr bool await_ready() const noexcept {return false;}
  void await_suspend(coroutine_handle){
      schedule_tasks()
  }
 ...
}
```

Code listing 5.11 WaitForTasksAwaiter needs to schedule awaiting tasks. Shortened for brevity.

Now that the individual tasks are scheduled, the question arises: how do we resume the suspended coroutine once those tasks have completed their execution? Our `Task` implementation is designed to transfer control back once it completes its execution. This approach is works in the case of a single task, but for multiple tasks, a different approach is required. Logically, we need to track the number of completed or pending tasks to determine when we can resume the suspended parent task. Utilizing a simple counter that decrements its value each time a child task is completed seems to be a logical solution

When the final child task completes, we resume the suspended coroutine. However, what should occur if we decrement the counter and it turns out that this task is not the last child task? In such cases, control should be transferred back to the worker so that it can execute a different task.

Resuming a coroutine from another coroutine can be achieved through a coroutine to coroutine transfer, as described in Symmetric transfer. We can return an awaiter that resumes the coroutine using a handle that it receives. This approach is similar to what we have implemented in our `Task`.

5.3.1 SharedBarrier

This object conveys the concept we've previously described: a counter that, once reduced to zero, returns the handle to the suspended coroutine. This handle is returned to the last task that decreases the counter, enabling it to resume the suspended coroutine with a coroutine to coroutine transfer.

```
class SharedBarrier{
 public:
  coroutine_handle decrement_and_resume() {
  int remaining = remaining_tasks_.fetch_sub(
    1, memory_order_acq_rel
  );
  if (remaining = 1)
    return continueation;
  else
    return noop_coroutine();
}
...
 private:
  coroutine_handle<> continuation_;
}
```

Code listing 5.12 Barrier implementaion, a handle for a suspended task is returned once all tasks are completed.

The main method is `decrement_and_resume`. It decrements the number of remaining tasks and checks if the current task is the last one to decrement the barrier. If it is, the handle of the suspended coroutine is returned; if not, the `noop_coroutine` handle is returned

The `fetch_sub` function returns the value that precedes the decrement. This means if `remaining` is equal to one, we can be certain that we are the task that decremented it to zero, thus making us the last task to decrement the counter.

Choosing the appropriate memory order for atomic operations is a complex task. By not using the sequential memory order, we are loosing sequential consistency. However, in the case of this counter, sequential consistency is not necessary, as the order of decrements is not crucial. Nevertheless, we cannot use a relaxed memory order due to potential reorderings. Therefore, we have used `memory_order_acq_rel`. This decision mirrors a similar scenario to `shared_ptr`.

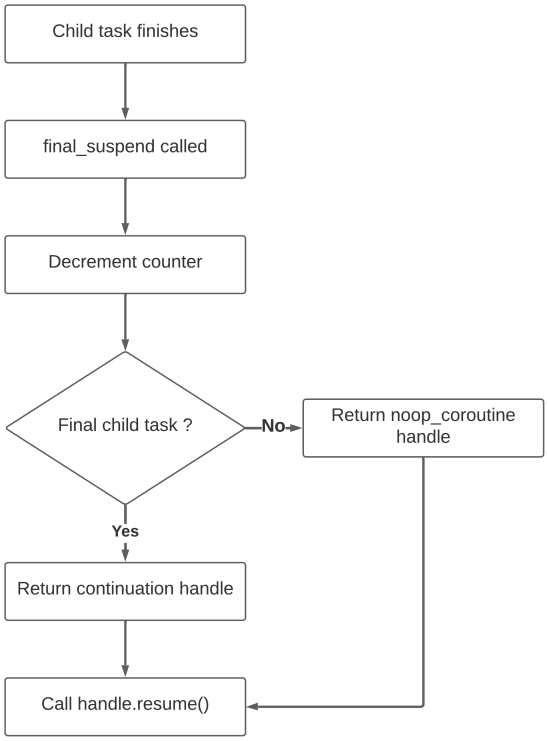

Figure 5.3 Task resumption process diagram.

This simplified diagram illustrates the process of resuming a suspended coroutine. When a child task completes its execution, the `final_suspend` method is called. The next step involves decrementing the counter and retrieving a handle from `SharedBarrier`. However, this cannot be performed within the `final_suspend` function, as the coroutine is not yet suspended at this point. The suspension occurs only after the awaiter from `final_suspend` has been `co_await`-ed

We need to return an awaiter from `final_suspend` that decrements the `SharedBarrier` counter and returns a handle from the `await_suspend` method. This implementation is straightforward because the `SharedBarrier` provides a coroutine handle in both cases: if we are the last task, we receive a handle to the suspended coroutine; if not, we receive `noop_coroutine_handle`.

```
struct DecreaseCounterAwaiter{
  constexpr bool await_ready() const noexcept {return false;}
  coroutine_handle<> await_suspend(coroutine_handle)
    return barrier.decrement_and_resume();
}
```

Code listing 5.13 DecreaseCounterAwaiter implementation, shortened for brevity.

There is one issue with this entire mechanism: the `DecreaseCounterAwaiter` behaves differently compared to the awaiter used in `Task` for the `final_suspend` method. Child tasks require a different approach at the final suspend point compared to parent tasks. To solve this, we have decided to create a wrapper for the `Task`, which we call `WaitTask`.

5.3.2 WaitTask

The idea behind `WaitTask` is to encapsulate individual child tasks within it and pass them to the `WaitForTasksAwaiter`. The `WaitForTasksAwaiter` is responsible for passing `WaitTasks` to the scheduler. We have also moved the `DecreaseCounterAwaiter` to the `WaitTask`'s `final_suspend` method. This ensures that each completed `WaitTask` decreases the counter and possibly resumes the suspended coroutine.

The actual body of the `WaitTask` coroutine is quite straightforward. Since the `Task` returns control to the caller upon completion, we can use `co_await` on `Task` within the `WaitTask`. This ensures that when the `Task` finishes, the `WaitTask` also reaches completion.

```
WaitTask fun{
  co_await Task<>;
}
```

Code listing 5.14 `WaitTask` body.

Since `WaitTask` is a coroutine, it must have an associated promise object where we can specify the `final_suspend` method. The promise object for `WaitTask` does not need to store a value, unlike in `Task`, because `WaitTask` will always return `void`.

Another difference involves storing a `SharedBarrier` pointer. This pointer then allows the `DecreaseCounterAwaiter` to decrement the counter on the barrier. The pointer is passed to the `WaitTask` by the `WaitForTasksAwaiter`, which is responsible for creating the `WaitTasks` and passing them to a scheduler.

```
class WaitTaskPromise{
 public:
  suspend_always initial_suspend();
  DecreaseCounterAwater final_suspend();
  void reutrn_void();
  void set_barrier(barrier);
  ...
 private:
  SharedBarrier barrier_;
}
```

Code listing 5.15 `WaitTaskPromise`.

To summarize the entire process, we have developed an additional task named `WaitTask`, associated with `WaitTaskPromise`. We also introduced two new awaiters: `DecreaseCounterAwaiter` and `WaitForTasksAwaiter`. Additionally, the `SharedBarrier` is a synchronization primitive that allows us to resume the parent task, once all the child tasks are finished.

In our implementation, we created a function named `wait_for_tasks`. This function generates an awaitable object that passes the `WaitForTasksAwaiter` to the `co_await` operator inside the parent task, simplifying the use.

The overall mechanism, with all its interactions, may seem somewhat complicated. To help with understanding, we have included a diagram below that captures all the significant steps, hopefully making the entire process clearer.

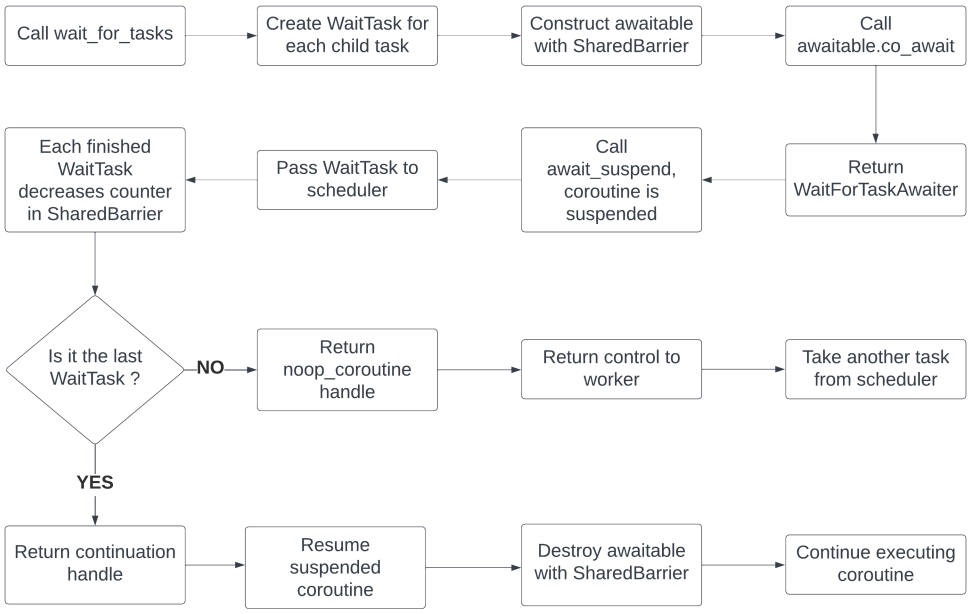

Figure 5.4 Waiting for child tasks process diagram.

5.3.3 Lifetime

The final thing to consider is the lifetime of the individual objects, particularly in multi threaded context.

We will start with the `SharedBarrier`. Since the barrier pointer is transferred to `WaitTask` from the `WaitForTasksAwaiter`, we opted to create it within the awaitable that returns this awaiter form its `co_await` operator. This awaitable is stored in the suspended coroutine's frame and is destroyed immediately after the `co_await expr` is evaluated. This means, the barrier is also destroyed once the parent coroutine resumes. By this point all child tasks finished execution and are suspended, making the barrier obsolete.

```
Task<int> fib(int n) {
  if (n < 2) return 1;
  Task<int> t1 = fib(n-1);
  Task<int> t2 = fib(n-2);
  co_await wait_for_tasks(t1, t2);
  // awaitable destroyed => barrier destroyed
  return t1() + t2() //retrieve value
}
```

Code listing 5.16 Life time of the barrier and the awaitble in process of suspending a task.

A scenario may arise where the last child task to decrement the counter resumes the suspended parent coroutine before some other child tasks exit the barrier. This is a perfectly valid scenario. The synchronization mechanism ensures that the counter is correctly decremented and that only one task can resume the suspended parent coroutine.

It does not guarantee that the last task to decrement the counter is also the last to exit this 'critical section'. A `std::mutex` could provide such a guarantee, but this is not applicable in our case. The question then arises: what happens when the coroutine is resumed while another child task is still within the `decrement_and_resume` method?

```
int remaining = remaining_tasks_.fetch_sub(
    1, memory_order_acq_rel
  ); //(1)
if (remaining = 1) // (2)
  return continueation; // (3)
else
  return noop_coroutine(); // (4)
```

Code listing 5.17 Barrier decrement process.

To resume the suspended coroutine, all child tasks must decrement the counter, as indicated by line (1). This process ensures that each task will eventually load the number of remaining tasks into their local `remaining` variable. Only one task, the one with `remaining = 1`, can resume the suspended coroutine. Suppose this task exits the function and resumes the suspended coroutine, which then destroys the `SharedBarrier` before any other tasks exit the function. In this case, all other tasks will execute lines (2) and (4). Neither line (2) nor (4) accesses member variables of the `SharedBarrier`, making their execution safe even after the `SharedBarrier` has been destroyed.

This simple proof makes it clear why the `SharedBarrier` is safe to use in a multithreaded context. An additional advantage is the memory optimization that occurs once the awaitable's memory is freed. The resumed coroutine can then reuse this space for other variables. Such optimizations are handled by the compiler.

Another crucial design decision concerns the lifetime of individual `Task`s and `WaitTask`s. In the section on `Task`, we explained that the returned `Task` object is an RAII object. When this object is destroyed, the associated coroutine state is also destroyed.

If we store individual child `Task`s inside the awaitable, it would be necessary to return the values of these individual `Task`s as a result of the entire `co_await` expression. This is because the coroutine states of the `Task`s would be destroyed along with the destruction of the awaitable object.

An alternative approach is to return `void` as the result of the `co_await awaitable` expression and accept child tasks as references. This way, users can access the values of individual tasks through the `Task` object itself.

We chose the latter approach. The `wait_for_tasks` function returns `void`. However, each child `Task` can be passed either as an l-value or an r-value. It is the user's responsibility to manage the retrieval of values from individual tasks. If the values are not needed, tasks can simply be passed as r-values.

```
template <typename T>
WaitTask create_wait_task(T&& tp) {
  T t = forward<T>(tp);
  co_await t;
}

template <typename... Args>
auto wait_for_tasks(Args&&... args) {
  vector<WaitTask> tasks;
  tasks.reserve(sizeof...(Args));
  (tasks.emplace_back(create_wait_task(forward<Args>(args))), ...);
  return WaitForTasksAwaitable(move(tasks));
}
```

Code listing 5.18 `WaitTask` construction.

In case an argument for `wait_for_tasks` is an r-value, it is binded to the `t` variable inside `WaitTask`. In case of l-value just a reference is passed. This allows for mixing r-values and

l-values in arguments of `wait_for_tasks`. In our Fibonacci example 5.16 both `t1` and `t2` go out of scope and their coroutine state is destroyed once the parent coroutine finishes its execution.

5.4 Scheduling

As we described in Scheduling, there are two popular methods for task scheduling: work stealing and work sharing. In our implementation we have implemented both approaches.

In our work sharing implementation, we utilize a global queue from which each worker thread can add or retrieve tasks. To ensure synchronization among workers, we considered two possibilities: using a standard queue and protecting it with mutex or using lock free queue. We implemented both versions, and their performance was evaluated in the Performance section.

For the mutex implementation, we used queue and mutex from standard library, `std::queue` and `std::mutex`. Implementing a lock free queue is complex. It requires careful usage of memory orderings and memory barriers, especially to implement it effectively. As implementing a lock free structures was not the main goal of this diploma thesis, we tried to find existing lock free implementation that would fit our needs and could be used in our project. Ultimately, we opted for an open source implementation by MoodyCamel called concurrentqueue[38]. It is a header only implementation of concurrent queue. It does not guarantee globa sequential consistency, since it leverages relaxed atomics. However, such guarantee is not required for our purposes, so this is not a limitation for us.

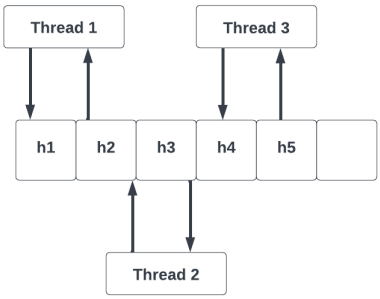

Figure 5.5 Global queue illustration.

The work sharing implementation is not complex. Each worker must have access to the global queue and mutex, if needed. This can be achieved by passing a pointer or a reference to each worker. Workers can lock the mutex, add or remove and elements, and release the lock.

In case of a lock free implementation, a mutex is not needed.

5.4.1 Work stealing

With the work stealing approach things are more complicated. Firstly, instead of a shared queue, each worker must have their own queue while also being able to steal from other workers' queues. Another difference is the need for double-ended queue, or deque. In this case we've also made two implementations: the first one uses locks and the standard deque, `std::deque` combined with `std::mutex`, and the second implementation is a lock-free one.

In the mutex implementation, each worker thread requires a mutex and a deque. One option would be to store the mutex and deque in thread-local variables. However, this is cumbersome since allowing other workers to steal from our deque, we would need to publish a pointer from within the worker thread to the others. The approach we chose is to create a `std::vector` of deques and mutexes, where each worker receives an index, and stores it in its thread local

variable. During worker creation, we simply pass an index and a pointer to the vector containing deques and mutexes. This allows the individual workers to access all other deques and lock mutexes if needed.

The lock free version requires a lock free deque. We didn't find a suitable lock free deque that would fit our needs, so we decided to implement our own. There exists a relatively simple algorithm described in Dynamic Circular Work-Stealing Deque[39]. We won't go into detail about the algorithm here, but will outline the necessary parts to understand our implementation. Detailed description can be found in the paper.

The underlying structure for this deque, as the name suggests, is a circular buffer. This allows for a more effective memory use. It is also dynamic, meaning it can increase its capacity as needed. In the deque we maintain two indices `top` and `bottom`, that are used for indexing. To find an element in underlying circular buffer we need to modulo both indices by the size of the circular buffer.

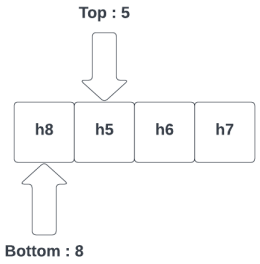

Figure 5.6 Illustration of circular buffer.

The lock-free deque supports these methods:

`push_bottom` puts a new element into the deque, increases bottom index by one, and enlarges the circular buffer if needed

`pop_bottom` pops the most recent element in the deque, decreases bottom index by one

`steal` steals the oldest element from the deque, and increments top index by one

Methods `push_bottom` and `pop_bottom` can only be used by the owner of the deque. The steal `method` is intended for workers to steal a task from different threads.

We also had to make a decision on how to manage memory with the deque. The Chase-Lev[39] algorithm suggests increasing buffer capacity as needed, while also keeping the previous buffers. This approach has two benefits: it avoids slowing down a computation by not destroying the buffers, and it saves time when shrinking the deque, by reusing the stored buffers. However, this method doubles the memory requirements compared to a strategy of growing and shrinking without keeping the smaller buffers. Another option would be to make the deque size static.

We chose to make the queue dynamic, it increases its size as needed, similar to `std::vector`. In our implementation, we double the size of buffer each time more space is required. The deque does not shrink its buffer when elements are popped.

Given that this is a concurrent data structure, a scenario may arise where one thread is reading values from the old buffer while the owner of the buffer is resizing the deque. This implies that we cannot immediately destroy the old buffer after resizing; we must keep it alive until no stealing thread has access to it.

In reference counting version we do not keep the smaller buffers alive, so we destroy the old buffers. We can use reference counting approach. Last thread that accesses the old buffer and decrements the reference count to zero is responsible for destroying it. As we mentioned earlier this does bring a runtime overhead, the extent of which will discuss in the Performance section.

For a reference counting we can use `std::shared_ptr`. Since this is a concurrent data structure, we need to use atomic operations. With C++20 we can wrap `std::shared_ptr` in `std::atomic` enabling atomic operations. However, this feature is not supported by libc++ which we use in combination with Clang. In such a case `std::atomic_load_explicit` and `std::atomic_store_explicit` can be used. Although these operations are deprecated in C++20, they are necessary when working with libc++.

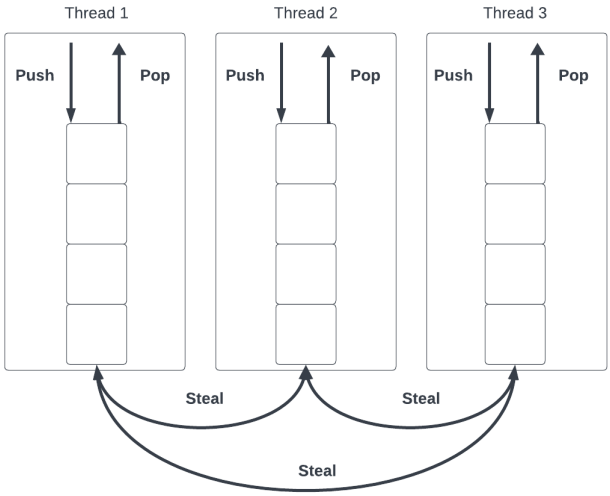

Figure 5.7 Work stealing scheduling diagram.

Another important aspects to consider are memory orders and memory barriers. In C++ we can specify `std::memory_order` for each atomic operation, or utilize `std::atomic_thread_fence`. To effectively implement Chase-Lev algorithm, we referred to Correct and Efficient Work-Stealing for Weak Memory Models[40]. This paper includes mathematical proofs for effective implementation, focusing on atomic operations and proper memory ordering.

The core idea of the approach is; once we have successfully managed to increment index we know no other thread could have done the same. This means it is safe for us to either store or remove an element from the deque. This safety is achieved using compare-and-swap operations and different memory ordering.

Chapter 6

Performance

In this section, we will examine the performance of our multithreaded library. We will execute a series of benchamarks, each designed to assess performance of our implementation from different perspective. Calculating the n-th Fibonacci number using a recursive approach serves as an effective benchmark to test the scheduling overhead associated with suspending and resuming individual tasks. Given that the function call involves only a single addition and the algorithm does not require access to shared data, it is a well-suited to measure the scheduling overhead.

Another algorithm we will use for benchmarking is matrix multiplication. This algorithm requires accessing global data and executing calculations within the function, allowing us to evaluate the performance of our implementation, not just scheduling overhead. We've implemented the divide and conquer version of matrix multiplication, that is cache oblivious[41].

For comparison, we will test our implementation(Coros), with two implementations of openMP standard, and the oneTBB library. We selected these two libraries due to their widespread use in parallel computing and different level of complexity. OneMP is well-known for simple parallel tasking and loop parallelism, while oneTBB is more complex library with its own concurrent structures, scheduler, etc. A detailed examination of these libraries is provided in an earlier section Available tools.

In our implementation we will begin with the most basic scheduler type, global queue guarded by mutex. In each subsequent benchmark, we will progress towards more sophisticated implementation.

Each benchmark was compiled with the GCC compiler, except for openMP libc++ implementation, which was compiled with Clang. The compilation flags used were `-std=c++20 -O3`. All of the benchmarks were ran on AMD EPYC 7H12 64-Core Processor, featuring 64 cores that are divided into 4 NUMA nodes. Typically, our benchmarks will utilize 1 to 16 threads to correspond with the core count of a single NUMA node.

6.1 Global queue and work stealing

In this benchmark, we aim to compare our two implementations: one utilizing a work-stealing scheduler and the other a global queue, to measure scheduler overhead. Each will compute the 30th Fibonacci number and we'll progressively increase the number of workers(threads) executing the calculation. All of the benchmarks were executed using `numacatl -cpunodebind=1 -membind=1`, forcing the OS scheduler to schedule threads on the same NUMA node.

From Figure 6.1, it is clear that our global queue implementation does not scale well, primarily due to the overhead created by blocking individual threads from accessing the critical section. Additionally, the GCC implementation does not scale effectively. We speculate that the OpenMP

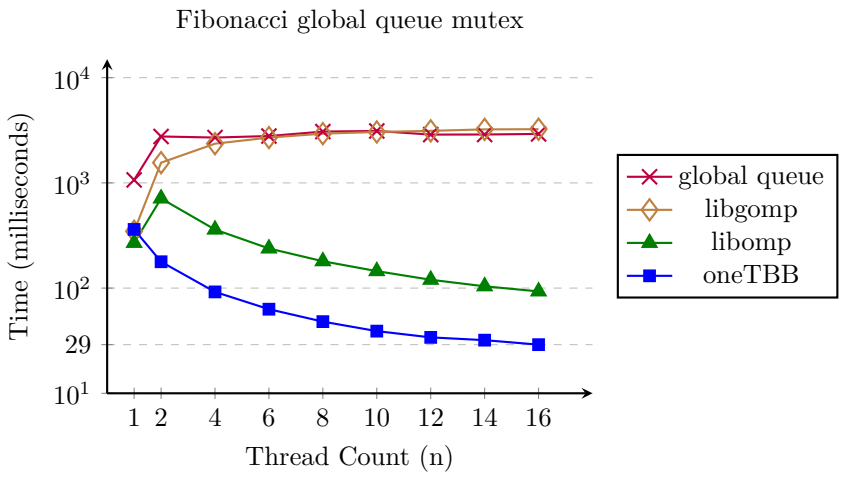

Figure 6.1 Comparison of the global queue implementation with other libraries.

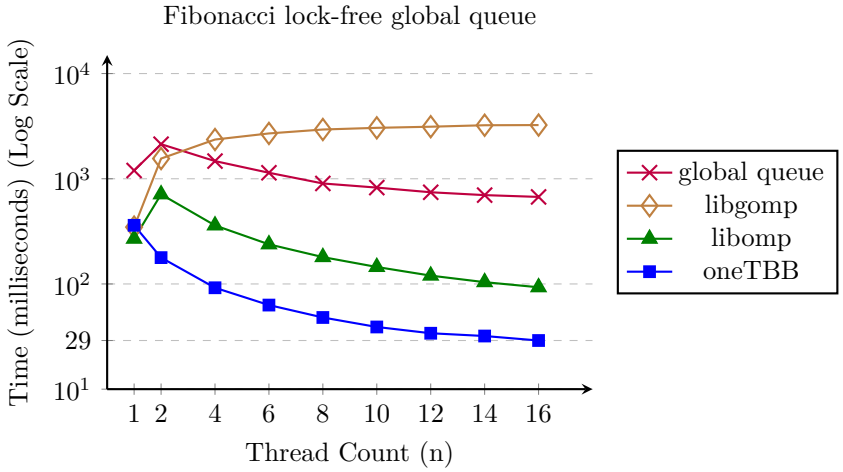

Figure 6.2 Comparison of the lock-free global queue implementation with other libraries.

implementation in libgomp utilizes some version of a global queue. In contrast, both oneTBB and libomp scale as expected; as the number of workers increases, the total execution time decreases.

We can improve our global queue algorithm by making it lock-free, detailed description of our lock-free global queue is in Scheduling.

As demonstrated in Figure 6.2 our implementation performs much better with a lock-free global queue. As the number of threads increases, the total execution time decreases. This is a significant improvement, compared to global queue with mutex. However, these results are still not comparable to the times achieved by oneTBB or libomp. As a next step, we will change the scheduling strategy; instead of using a global queue, we will implement a simple version of work-stealing that uses a mutex.

Results from Figure 6.3 show how much better does work stealing scheduling scale, even the mutex version, compared to a global queue. Naturally, next improvement is to implement a lock-free work scheduling algorithm.

When we initially executed the next benchmark, the results were surprisingly inconsistent, showing significant time variations between individual runs. It appeared as though an element of randomness affected the process. Although the lock-free structures decreased scheduling over-

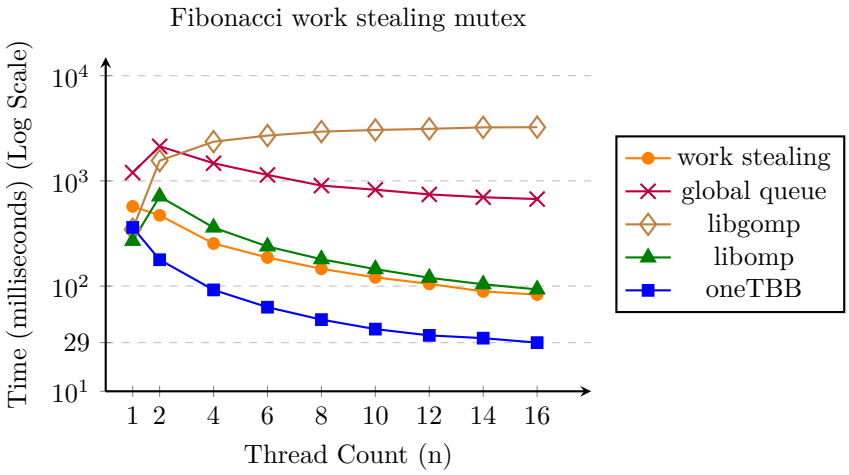

Figure 6.3 Comparison of the work-stealing implementation with other libraries.

head, they introduced an element of "unpredictability". This was particularly evident when compared to oneTBB or OpenMP.

We used a tool in Linux called perf[21] to analyze our program. This tool displays individual function calls and measures time or overhead spent in different functions. We noticed that a system call to the scheduler was consistently at the top of the list. Further analysis revealed that mutex calls throughout the program were causing threads to be put to sleep and rescheduled, when unable to secure a lock, leading to considerable overhead. However, once we changed the entire program to a lock-free version, not just the data structures, the irregularity in data disappeared. For comparison, we measured the variance before and after the change.

Version	Variance
Few mutex calls	129.79
Pure lock free	0.18

Table 6.1 Difference in variance in data, caused by utilizing of mutex.

Measured data clearly demonstrate, that even when using lock-free data structures, protecting critical sections with mutexes can cause a significant difference in individual runs.

One place where the lock was called, potentially leading to rescheduling, was our counter used in the process of checking if awaiting `Task` can be resumed. Second occurrence was more subtler. The C++20 standard[17] introduced change that allows using `shared_ptr` with atomics `atomic<shared_ptr<T>>`. However, this is not yet implemented in the libc++ as of time writing this thesis, so we had to use deprecated functions like `atomic_store_explicit`. While these calls technically make the `shared_ptr` atomic, the underlying implementation relies on mutexes. The standard does not specify the exact implementation details for these operations An alternative approach to solving this problem of putting individual threads to sleep and awaking them, would be to use spinlocks. Spinlock can for example be implemented with atomics and loop.

In Figure 6.4, we have included only our lock-free implementation, libomp, and oneTBB. We omitted the libgomp implementation, as it is evident from previous graphs that it does not scale well. From this data, it is clear that our implementation outperforms libomp and is relatively close in performance to the oneTBB library.

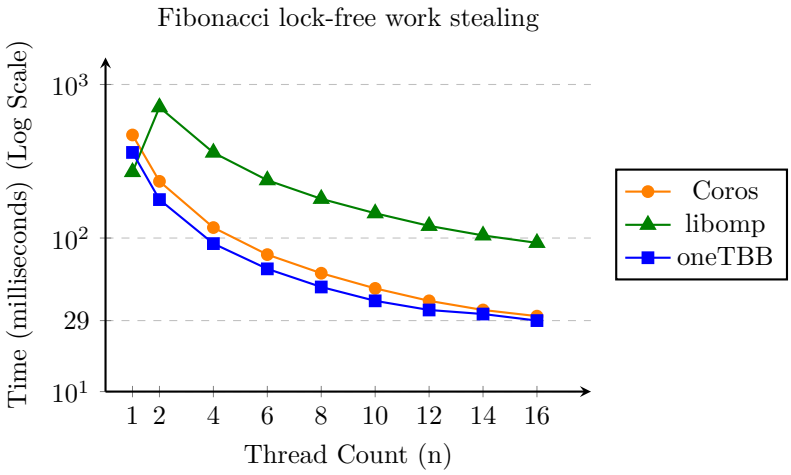

Figure 6.4 Comparison of the work-stealing lock free implementation with other libraries.

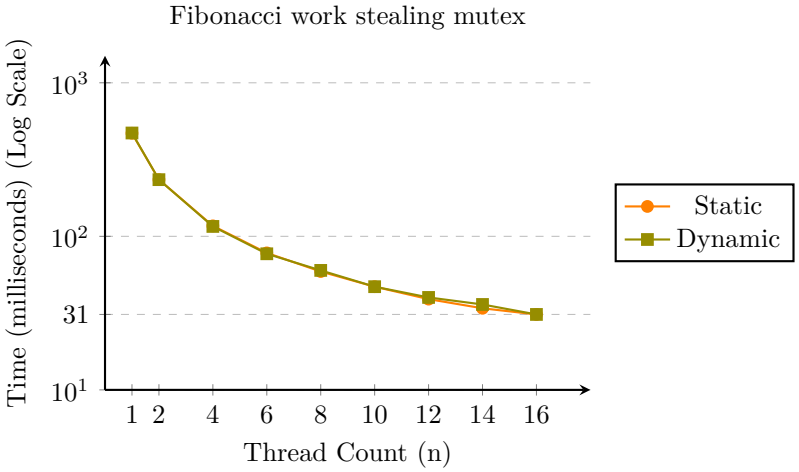

Figure 6.5 Comparison of static vs dynamic polymorphism with regard to scheduling individual tasks.

6.2 Static and dynamic polymorphism

This benchmark examines the differences between static and dynamic polymorphism in C++. Static polymorphism involves the use of template programming in C++, which adds complexity to the program and makes it harder to debug. We hypothesize, that replacing dynamic polymorphism with static might improve performance by eliminating a layer of indirection(vtables).

In our code, when a `Task` awaits another task or multiple tasks, it needs to inform the scheduler to schedule these tasks. In the dynamic version, the `Task` communicates with the scheduler through an interface. In the static version, we use templates, replacing the interface with a template parameter, which allows the `Task` to directly call the scheduling function. This operation is particularly frequent in our Fibonacci benchmark. We are testing whether the use of static or dynamic polymorphism makes a noticeable difference in these frequent operations.

As shown by Figure 6.5, there is no significant difference between dynamic polymorphism and static polymorphism in this instance. The observed difference could be too small, or perhaps compiler optimizations have mitigated it. In our case, replacing interfaces with templates does not yield a significant performance benefit. Given the complexity that templates can introduce to

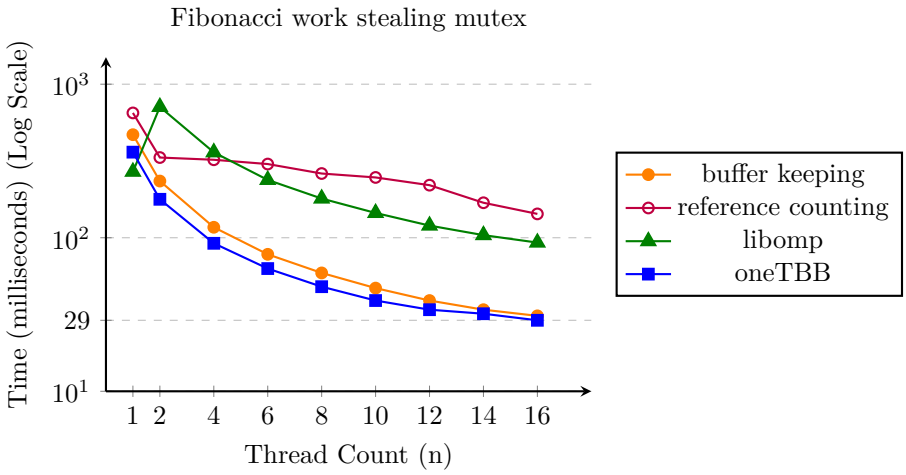

Figure 6.6 Comparison of buffer keeping and reference counting versions.

the code, it is better to use dynamic polymorphism and resort to templates only when necessary, at least in our case.

6.3 Reference counting vs buffer keeping

In this benchmark, we examine two possible implementation of work scheduling algorithm[39]. Both of these are described in detail in Scheduling section. Briefly, one implementation destroys smaller buffers that are not necessary anymore during the execution of the computation. Second implementation, stores pointers for older buffers and destroys all of them after the computation.

Reference counting version effectively doubles the amount of memory needed. However, this benchmark depends on the initial size of the buffers. If we set initial size of the buffers large enough that there is no need for buffer growth, we won't see any difference. For this test, we set initial size of the buffers to 64.

Figure 6.6 clearly demonstrates that reference counting version scales significantly worse than the buffer keeping version. In case we can afford the additional memory cost, it is better to use buffer keeping. Our final implementation uses the buffer keeping version of the algorithm.

6.4 Memory usage

In the previous section, we looked at the two version of our work stealing algorithm, focusing on the memory-performance trade-off. This benchmark evaluates the different memory usages of individual libraries. To measure a memory usage, we utilized the `Time`[42] command on Linux, which allows us to obtain the maximum resident set size for a process. This measurement indicates the peak physical memory consumption of the process.

Examination of Figure 6.7 produced unexpected results. We had anticipated that oneTBB would show higher memory consumption, considering it is a robust and highly customizable framework. However, the high memory consumption of libomp was unexpected. We are pleased with our implementation's relatively low memory consumption compared to other libraries.

We think the difference can be attributed to different exception handling, scheduling and overall functionality. For a more in-depth analysis, tools like `Valgrind` could be used to obtain more detailed information about memory allocations.

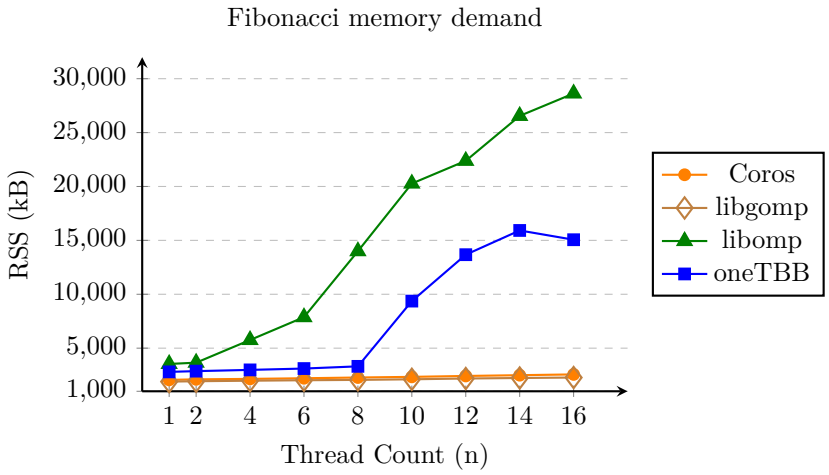

Figure 6.7 Comparison of memory usage of individual libraries.

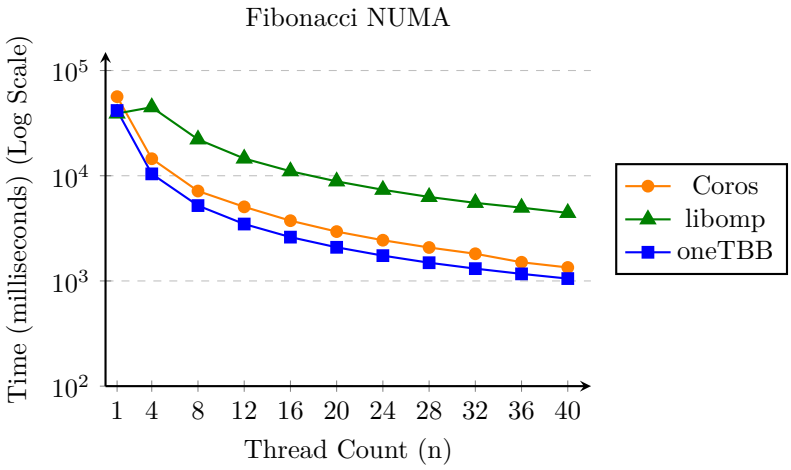

Figure 6.8 Testing NUMA affinity on Fibonacci computation.

6.5 NUMA

In this test, we will use more than 16 threads, up to 40 workers/threads. Additionally, we won't be using the `numactl` command to force CPU affinity. Our aim is to test whether libraries like OpenMP and oneTBB can leverage NUMA architecture by default. Both libraries offer methods to utilize NUMA nodes: OpenMP allows setting an affinity, and oneTBB can create a `task_arena` with set affinity. However, we choose not to utilize these options. Programming NUMA aware programs adds an additional layer of complexity. We want to find out whether these libraries inherently take advantage of NUMA.

Analysis of Figure 6.8 suggests that neither OpenMP nor oneTBB leverage NUMA nodes by default. The oneTBB library outperformed our implementation even in the single NUMA node test, and it is expected to scale better in this test as well. This benchmark focuses solely on measuring scheduling overhead, and, as previously mentioned, it does not access any global data that would invalidate cache lines.

In our next benchmark, we will focus on matrix multiplication. This test is not only heavy on scheduling overhead but also requires data for computing individual matrix elements. If there is

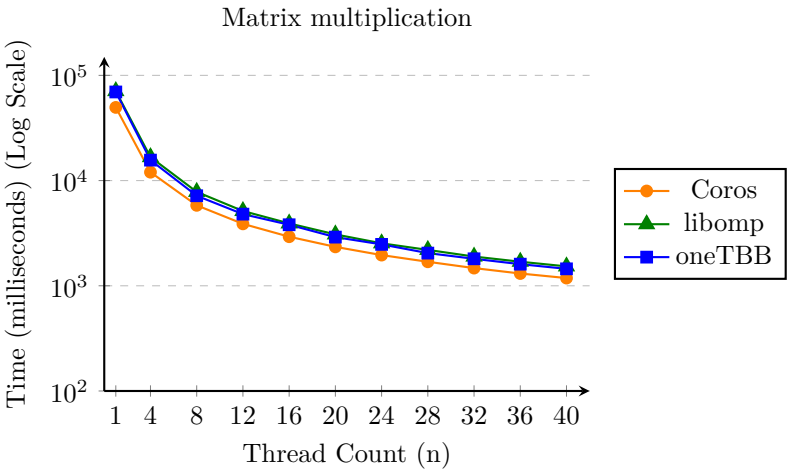

Figure 6.9 Testing NUMA affinity on matrix multiplication.

any NUMA optimization, we might observe an improvement. Caches can be invalidated between tasks and accessing a processor's local memory is faster than accessing a non-local memory in NUMA architectures. Therefore, if tasks are primarily scheduled on one or two NUMA nodes, data access should be faster.

Data in Figure 6.9 were surprising for us. We did not anticipate our implementation outperforming oneTBB in this benchmark. We executed two variants of this test with oneTBB: one utilized `parallel_invoke`, and the other, `task_group`. The first version, using `parallel_invoke`, should theoretically be a bit faster than the one with `task_group`, as `parallel_invoke` is static and the number of tasks cannot change during an execution, unlike `task_group`. However, both versions achieved almost identical times. We believe that with the correct setup, oneTBB could outperform our implementation in this test. Nonetheless, in this benchmark, our implementation did better.

These results also suggest that there is no default optimization for NUMA nodes in oneTBB or OpenMP. However, the scheduling algorithm can also play a significant role. Our lock-free algorithm, which gives limited opportunities for the OS scheduler to reschedule threads to different NUMA nodes, combined with the manner in which work-stealing populates its task queue, may render the factor of CPU affinity negligible.

The reason we did not include libgomp is that it continued the same trend observed in the previous benchmarks. The scheduling overhead increased with the rising number of threads. Furthermore, in the matrix multiplication benchmark, the total execution time significantly increased, and the tests took too long to execute.

6.6 Summary

In the Coroutines section, we have detailed the workings of coroutines and the mechanisms associated with them. We then described our implementation, beginning with basic building blocks such as `awaiter`, `promise`, `Task`, and `Wait_task`. Next, we outlined our two implementations of the scheduling algorithms. Finally, we brought all these elements together and tested our implementation, Coros, in the Performance section.

We conducted multiple benchmarks measuring scheduling overhead, memory demands, and CPU affinity effects. Additionally, we compared two versions of our work-stealing scheduler implementations, as well as static versus dynamic polymorphism in our scheduling algorithm. Benchmarking can be a challenging process, and results may vary across different machines. We

have made an effort to assess the quality of each implementation from multiple perspectives.

One of the biggest surprises that emerged from the data was the significant difference between the libgomp and libomp implementations of the OpenMP standard. It appears that the GCC's version is likely using a global queue for scheduling, which introduces a considerably large overhead. However, this difference may be negligible in algorithms where each task is computationally heavy, especially when compared to the speedup gained by parallelizing the process.

We are satisfied with the results of our implementation. Our library, which is not NUMA-aware, implements a work-stealing algorithm, does not utilize platform-specific tools and is not as robust as other libraries, serves as a proof of concept, that a fast multithreaded library can be built using standard tools available in C++20.

Chapter 7
Server

7.1 Connection handling

A basic workflow of our server is to accept a new connection and serve the request. We won't employ any of the typical optimizations. We want a simple web server and experiment how well it scales with multi threading.

On linux there are three options:

- poll[43]
- select[44]
- epoll[45]

select has a limitation that it can only monitor a small number of file descriptors, set by constant `FD_SETSIZE` which is usually 1024. It is recommended to use poll instead of select.

While poll offers similar functionality to select, it can handle more file descriptors than select.

Epoll is the most robust one out of these 3 options. Offers an edge triggered and a level triggered mode and also can operate in asynchronous mode. It is also the most efficient option. To work with epoll we need to create an epoll instance, it is a kernel object. We can pass descriptors and events we are interested in to the epoll instance to monitor them. To do that we need to wrap the descriptor into a file structure called `epoll_event`. We then call `epoll_ctl`, to add the `epoll_event` into the epoll instance for monitoring. So we will use epoll to monitor file descriptors for possible I/O operations.

We will use epoll in edge triggered and asynchronous mode. This allows us to asynchronously check for coming connections and requests. Edge triggered mode, gives a signal only when there s a change in state of the file descriptor. For example when the file descriptor is ready to be written to without blocking.

```
struct epoll_event event;
event.data.fd = socket_fd;
event.events = EPOLLIN | EPOLLET;
int s = epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &event);
```

Code listing 7.1 Registering a socket with an epoll instance.

In this code snippet, we create an event that is set to edge-triggered mode using the `EPOLLET` flag. Additionally, we can specify the `EPOLLEXCLUSIVE` flag, which allows the event to be 'waited on' by multiple threads. This prevents the thundering herd[46] problem by ensuring that only one thread processes the event at a time.

To create a server we need to open a socket, that users can communicate with. The socket will accept TCP communication. We can register this socket's file descriptor with the epoll instance to monitor for any incoming connections.

```
while (true) {
  struct sockaddr in_addr;  // (1)
  socklen_t in_len = sizeof(in_addr);
  int infd = accept(socket_fd, &in_addr, &in_len);  // (2)
  if (infd == -1) {
    if ((errno == EAGAIN) || (errno == EWOULDBLOCK)) {  // (3)
      break;
    } else {
      ... // (4)
    }
  }
}
```

Code listing 7.2 Accepting new connections.

On line (3), we accept a new connection using the accept[47] function, utilizing the `sockaddr` struct created on line (1). The code on line (3) checks if all available connections have been read. This is done by checking `EWOULDBLOCK` and `EAGAIN`. These two errors signalize that trying to accept a new connection would block. This is crucial because we use an epoll instance with an event set to edge-triggered mode[45], which requires reading all incoming connections on the socket. Once all connections are accepted, the loop terminates. Line (4) handles different errors that do not indicate that all incoming connections have been read and that some error has occurred.

Once the connection is accepted, we register it with an epoll instance to monitor for requests. Processing a simple HTTP request involves checking the headers and processing the payload, if any. In our scenario, we respond by returning simple plain text back to the user.

```
string httpResponse = "HTTP/1.1␣200␣OK\r\n"  // (1)
  "Content-Type:␣text/plain\r\n"
  "Content-Length:␣" + to_string(to_string(result).size()) + "\r\n"
  "\r\n" + to_string(result) + "\r\n";
int response_remaining = httpResponse.length();
do {
  int s = write(event.data.fd,  // (2)
                httpResponse.c_str(),
                strlen(httpResponse.c_str()));
  response_remaining -= s;
} while (response_remaining > 0);  // (3)
close(request_file_descriptor);  // (4)
```

Code listing 7.3 Sending back a response in our server implementation.

Initially, we parse the request, this is not shown in the code snippet, then on line (1), we construct a response that is subsequently sent back to the user on line (2). We utilize a loop on line (3) in cases where the entire response does not fit into the buffer and needs to be split. After the complete response has been sent, the connection is closed. This process follows a simple one-connection, one-response model without support for keep-alive.

Parsing the request in our server follows a similar pattern to sending a response: the entire request is read in a loop. Once the request has been fully read, the function proceeds to form the response. Given we expect only one type of request for our server, no further parsing is needed.

These two operations form the main loop of our server. Either the server accepts a new connection or processes a request. In the next code snippet we show this main loop.

```
int n = check_events(epoll_fd, events);  // (1)
for (int i  = 0; i < n;i++) {
  if (check_for_error(events[i])) continue;  // (2)
  if (events[i].data.fd == socket_fd) {  // (3)
    accept_new_connections(socket_fd, epoll_fd);
  } else {
    read_requests(events[i]);
  }
}
```

Code listing 7.4 Main loop in our server, that either accepts new connections or processes requests.

On line (1), we query the epoll instance for events, by passing the epoll file descriptor and an array intended to be populated with events to the `check_events` function. This function returns the number of events that have been populated into the `events` array. Next, we loop over these events to check for errors. If an error is encountered, it is logged on line(2), and the loop continues. If no error is encountered, the event is then processed.

Line (3) determines whether the event is a new a request for our server or a new connection. In the case of a new connection, it is accepted and then registered with the epoll instance to monitor for incoming data and requests. In case of request, it is processed.

7.2 Multithreading

In the previous section, we described how individual connections are handled by our server. In this section, we will describe the different versions of our server that we plan to benchmark. Each version utilizes the connection handling method described earlier, either in their original form or with modifications to enable scaling with our multithreaded library.

In our tests we will benchmark 3 version of our server :

- single loop
- multiple loops
- worker version

The single-loop server version is the most simple implementation out of the three. Server runs a single loop for accepting new connections and processing requests. All the requests are processed synchronously. Once the requests have been retrieved from an epoll instance, they are processed immediately.

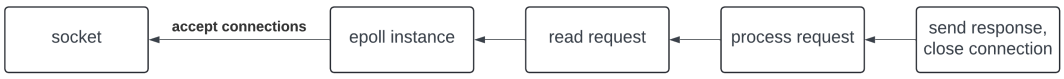

Figure 7.1 Single loop server.

The multiple loops implementation, shown in Figure 7.2, follows the same basic pattern as the single-loop version but extends it across multiple threads. Each thread operates its own epoll instance to monitor for new connections and requests. Similar to the single-loop model, as soon as a request is received, it is immediately processed by the thread that retrieved it from its respective epoll instance.

Another change involves registering the socket with the epoll instance using the `EPOLLEXCLUSIVE` flag. This is necessary because multiple threads can accept new connections and add them to

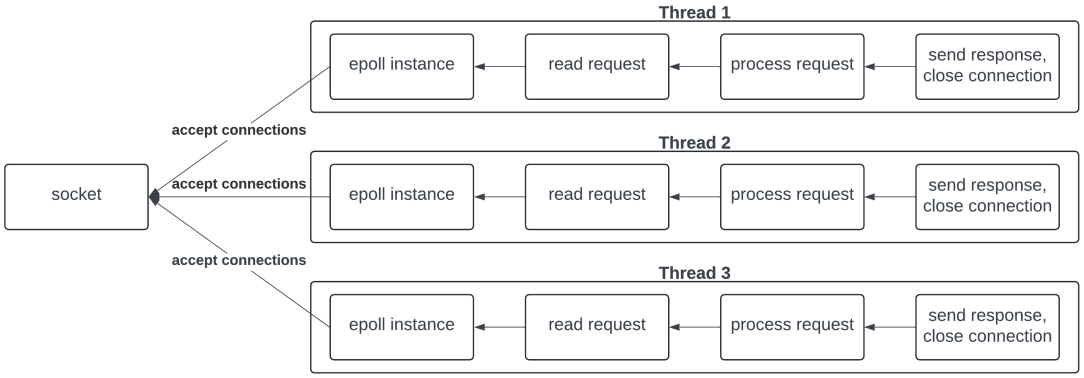

Figure 7.2 Multiple loops server version.

their own epoll instances. To ensure that only one thread accepts each new connection, we use the `EPOLLEXCLUSIVE` flag.

Worker version is implemented with the work-stealing scheduler. On one thread the main loop is running, similar to the loop in single loop server version, except for processing the request it creates a task and passes it to the worker pool for processing. So the main loop only accepts new connections and puts them into the worker pool. Essentially wrapping the connection in a standalone task, that can be processed by a worker. Sending a response and closing the connection is handled by the worker thread.

Once a worker begins processing a request, it is permitted to create sub-problems within the same thread pool. This means that requests and sub-problems generated by different workers are within the same pool.

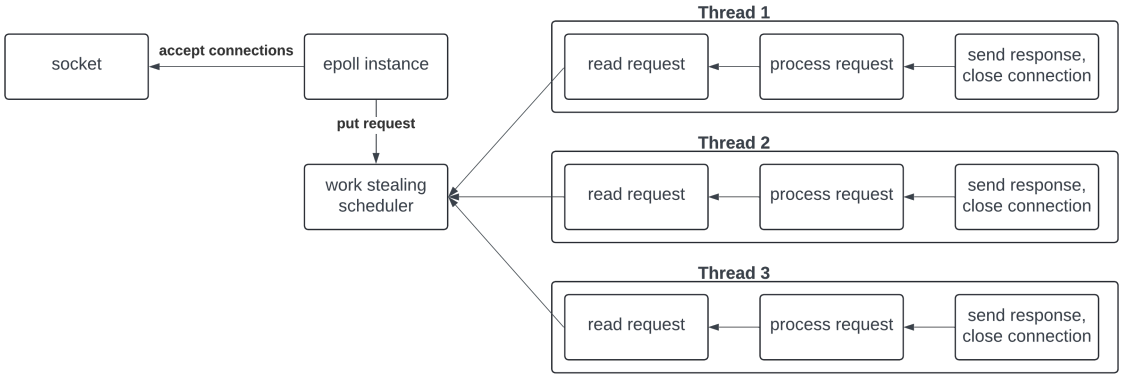

Figure 7.3 Worker model server.

Each server will use 8 threads, except the single loop version, in which only one loop is running to process the requests. Multiple loops version will run 8 loops for accepting new connections and processing requests. The last version of our server will use 1 thread for accepting connections and putting tasks into the work stealing scheduler, and 7 threads for processing individual requests represented as standalone tasks.

Chapter 8

Server experiments

In this section, we will evaluate our server implementations under various types of load. We have designed three types of tests to assess performance:

- static content
- CPU-bound
- IO-bound

In each of these benchmarks, we will measure the response times for requests, focusing on the average response time, dispersion in the data, and the maximum response time for each request. This analysis will provide a comprehensive understanding of how individual server implementations perform under different types of requests and the quality of user experience they are likely to offer.

Another good metric could be timed-out connections; however, we have decided to extend the timeout duration to 30 seconds. This ensures all connections are included in the statistics, allowing readers to determine what response time they would consider unacceptable.

We won't compare our server implementations with other servers such as NGINX[48] and Apache[49]. This decision is based on several factors that would skew the accuracy of such comparisons. Firstly, our server does not implement the complete HTTP parsing process and lacks support for the keep-alive header. Additionally, other servers employ different methods for executing scripts, such as FastCGI [50] [51]

We have decided not to support the keep-alive feature primarily because it introduces an unwanted variable into our testing. Specifically, the duration for which connections are kept alive could influence our test results.

Additionally, full-fledged server implementations include routing, which our simplified server model does not support. Similar difference apply to other servers developed using technologies such as Node.js[52] or Go[9].

Given our decision to conduct our tests without optimizations, we aim to accurately assess how well individual implementations perform under specific types of requests. Consequently, we will not compare our implementation with full-fledged web servers, as previously stated, because such comparisons would not yield accurate results. Although a comparison with systems like Apache's Multi-Processing Module (MPM)[53] might be interesting, it is not the main focus of this thesis.

8.1 Testing tools

The most common type of server testing is load testing. This involves subjecting a web server to a specific load to collect various metrics that assess the server's performance. Typical metrics include:

- Number of requests per second
- Average response time
- Failed requests
- Resource utilization

Testing how many concurrent connections a server can handle is not the only important metric to measure. The number of requests per second can increase, however the delay of a response typically increases. This typically holds to a certain point, after which the number of request per second decreases, implying the server cannot handle so many concurrent requests.

It is up to us to decide what is the optimal response delay we are able to tolerate, this is typically determined from the server's role. In case if web sites the delay can be a little bit higher than for example in some real-time systems.

Failed requests is also an important metric to keep track of. If the servers in a certain load will experience failed requests, we will note this in the text.

Another important thing to keep look at is memory and CPU saturation. This metric can show where the bottle neck is. For example with Apache web server, memory saturation could be a problem, since it spawns a separate process for each request. With threads and coroutines, we do not expect memory saturation to be a problem. With our test we expect the bottle neck to be CPU.

Many freely available tools exist for server testing, different tools support different range of protocols, for example Apache JMeter support HTTP, FTPS, SMTP and many more. However, our focus will be on tools that support HTTP, as our server supports HTTP requests. In this section, we will cover a few tools that may be used in our testing process.

8.1.1 ApacheBench

ApacheBench[54] is a straightforward web server testing tool designed specifically for HTTP servers. It can be easily installed on many Linux distributions as it comes bundled with the Apache web server.

```
ab -n100 -c10 http://127.0.0.1:8080
```

Code listing 8.1 Using ApacheBench to launch 10 concurrent connections, totalling 100 requests.

This command sets `-n 100`, the total number of requests, and `-c 10` to specify the level of concurrency. The final part of the command is the URL targeted for testing. This command will send a total of 100 requests to the server at a rate of 10 requests at a time, which means ApacheBench will attempt to keep 10 concurrent connections open to the server.

ApacheBench also provides additional options for more customized testing, by specifying the header fields using the `-H` option, for example for keep-alive header.

After completing the test, ApacheBench reports various metrics including the total time taken to complete the test, the number of requests per second, the total amount of data transferred, the number of failed requests, and the average time per request.

ApacheBench is a straightforward and user-friendly tool that reports the essential metric needed for performance testing.

8.1.2 WRK

WRK[55] is an open-source HTTP benchmarking tool, that can put a heavy load on the server, thanks to thread scaling. It allows users to specify the total number of threads the tool can use.

```
wrk -t2 -c100 -d30s http://127.0.0.1:8080
```

Code listing 8.2 WRK utilizing 2 threads to keep 100 open connections for 30 seconds.

This command runs WRK with 2 threads, establishes 100 concurrent connections to the server for 30 seconds. WRK also supports usage of additional header fields using the `-H` option.

The statistics reported back contain the total number of requests, latency for a request, and total requests per second, and more. Similar to ApacheBench.

8.1.3 Siege

Siege[56] is another HTTP load testing tool that, similar to WRK, allows users to specify the number of concurrent connections and the total duration of the test.

```
siege -c100 -t30s http://1270.0.0.1:8080
```

Code listing 8.3 Launching siege with 100 concurrent connections for 30 seconds.

Siege offers an additional functionality, such as the ability to hit multiple URLs from a single file, introduce delays between requests, cookies and custom headers. It provides detailed reports that include metrics like successful transactions, failed transactions, the number of transactions(requests) per second, and failed requests.

8.2 Final setup

We have chosen to use WRK for our tests due to its convenient feature that allows specifying the duration of the test rather than the total request count. Therefore, all tests described in the subsequent sections will be conducted using the following WRK command:

```
wrk -t2 -c[N] -d5m http://127.0.0.1:8080
```

Code listing 8.4 WRK setup for testing our server implementations.

This command launches WRK for a duration of five minutes, with a number of concurrent connections denoted `N`. To stress the server we will progressively increase the `N` with each test iteration. Another advantage of WRK is its multithreading feature, which allows for the adjustment of thread count. For instance, if two threads cannot effectively manage a specific number of concurrent connections, the number of threads can be increased.

All tests were conducted on an Amazon EC2 instance with 16 CPUs.

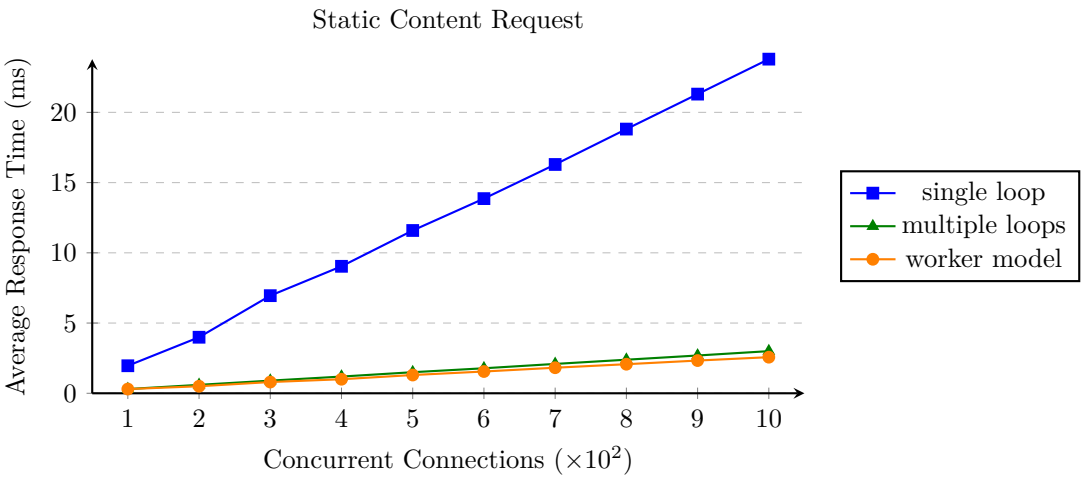

Figure 8.1 Average response times in static content test.

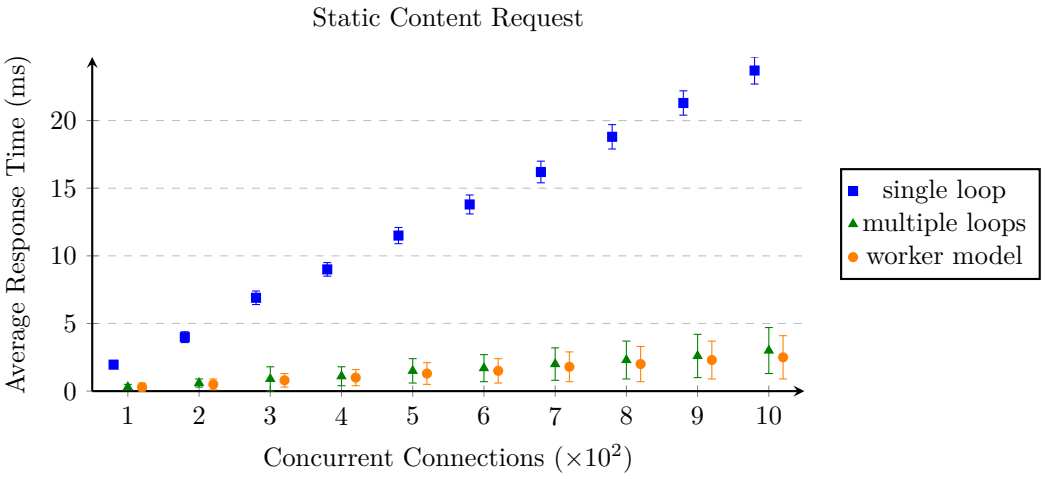

Figure 8.2 Dispersion in response times in static content test. Points are shifted to improve readability.

8.3 Static content

Serving static content, such as web pages, is a common task for web servers. Typically, a user requests a webpage from a server, which then responds by sending back an HTML document. Our test will be very simple, user sends a request and the server sends back a HTML document. We will measure the time it takes for the user to get the response with the web page.

From Figure 8.1, it is evident that both the worker model and the multiple loops model significantly outperform the single loop variant. This outcome is not surprising. Additionally, there is no significant difference in performance between the worker model and the multiple loops model.

When we examine the dispersion of data in Figure 8.2, we find that the dispersion for each server implementation is close to the average response time. This observation suggests that there is no distinct advantage to using the worker model, as simply scaling individual event loops yields comparable results.

Figure 8.3 follows the same pattern seen in previous results. Given similar average response time and minimal dispersion of data, the maximal response times are also close to each other.

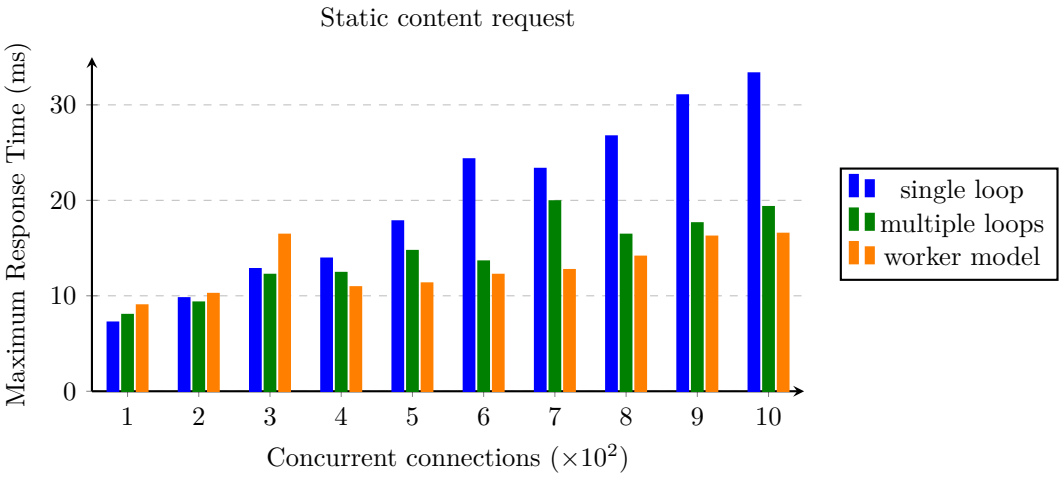

Figure 8.3 Maximum response times in static content test.

Serving static content benchmark demonstrates that the work-stealing scheduling approach, and thus the worker model, does not yield significant benefits compared to the multiple loops model. However, this observation also indicates that the scheduling overhead associated with the work-stealing scheduler does not introduce significant delay in response time. In the worker model, each request must be scheduled, whereas in the multiple loops model, each thread processes its own requests directly.

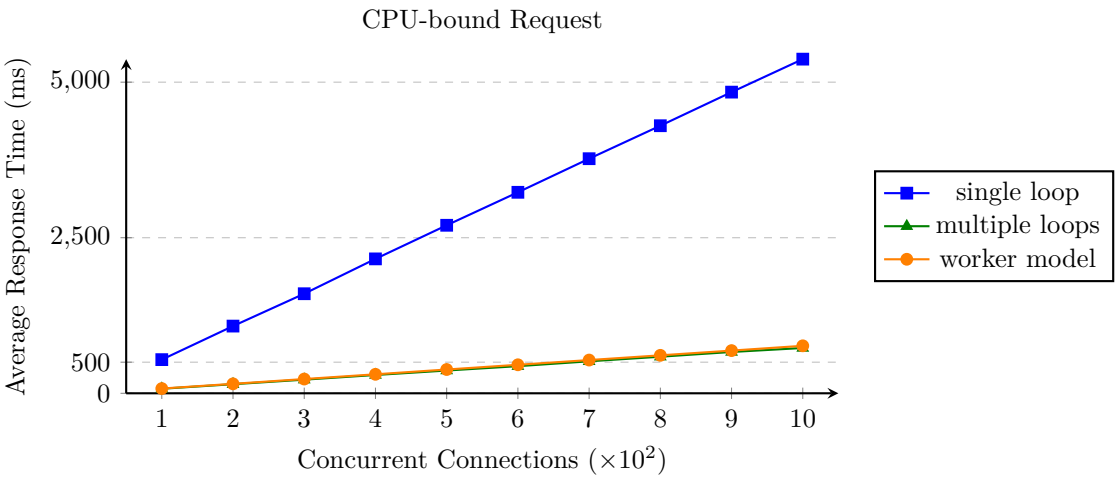

Figure 8.4 Average response times in CPU-bound test.

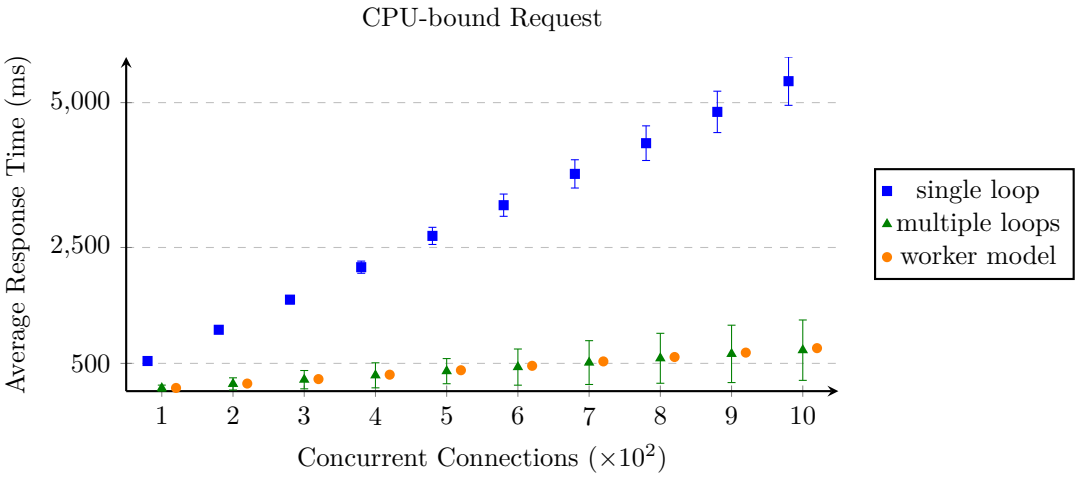

Figure 8.5 Dispersion in response times in CPU-bound test. Points are shifted to improve readability.

8.4 CPU-bound

In this test, each request will require the server to perform a CPU-intensive task, specifically calculating two Fibonacci numbers, adding them together, and then sending back the sum in the response. We chose to calculate Fibonacci numbers for two reasons: they are easy to implement and allow the problem to be divided into sub-problems. This division enables schedulers to effectively distribute the load across different threads.

Similar to the previous test, the single loop version is a single event loop that accepts new connections and processes requests sequentially. This means that it immediately begins calculating the Fibonacci sum upon receiving a request and sends back the response before processing another request.

The multiple loops version operates similarly to the single loop version but is scaled across multiple threads.

In the worker model, the entire request is passed to the worker thread pool. Each worker takes a request and begins the task of calculating the Fibonacci sum. Individual Fibonacci numbers are calculated using a similar method as described in 5.15. Each request generates two

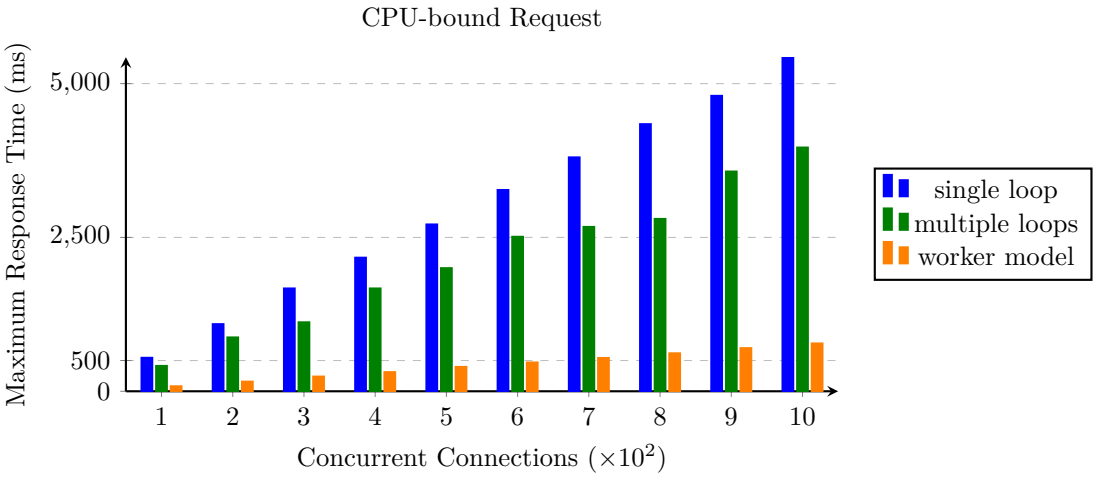

Figure 8.6 Maximum response times in CPU-bound test.

additional tasks for calculating Fibonacci numbers. Both the new requests and the Fibonacci number calculations are managed within the same worker pool, not in separate pools.

From Figure 8.4, it is clear that both the multiple loops and worker model yield similar average response times. These results are very consistent with those from the previous Static content test.

In Figure 8.5, we observe a significant difference in data dispersion, particularly between the multiple loops and worker model. Although these models produce similar average response times, their dispersion varies considerably. The multiple loops model exhibits much greater dispersion, which implies a significant difference in response times for individual requests. In such cases, some users might experience very fast response times, while others may experience significantly slower ones. In contrast, the worker model tends to provide a more uniform experience, offering similar response times to all users.

We believe the significant difference in data dispersion can be attributed to the different order in which requests are processed. In the multiple loops model, each loop polls its own epoll instance for requests in 'batches,' meaning that multiple requests are retrieved from the epoll instance simultaneously. This approach is far more efficient than polling for a single request each time.

Each loop in the multiple loops model retrieves its own batch of requests, sized N. The last request in the batch is processed only after the first N-1 requests have been completed. This creates delay for the elements that are processed later in the batch.

In the worker model, each request is added to a thread pool with a work-stealing scheduler. This means, there is "global order" in which the requests are processed. The first available worker thread picks up the next request in this sequence, ensuring that all workers are effectively processing from one 'big queue.' Similar to the multiple loops model, the polling in the worker model is conducted in batches, and individual requests are subsequently placed into the worker pool.

One way to decrease the dispersion in the multiple loops model would be to reduce the 'batch size.' However, smaller batch sizes would result in more frequent epoll polling.

Figure 8.6 reflects the high dispersion in data for the multiple loops model. The maximum response times for the multiple loop variant are significantly higher than those of the worker model, and they are also comparable to those of the single loop version. This indicates that some users may experience significant delays in receiving their requests.

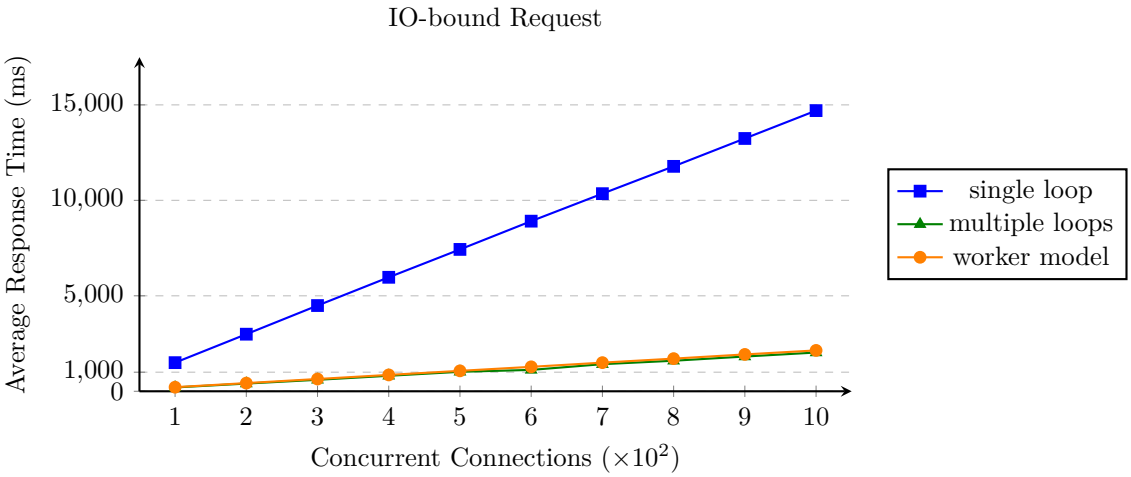

Figure 8.7 Average response times in IO-bound test.

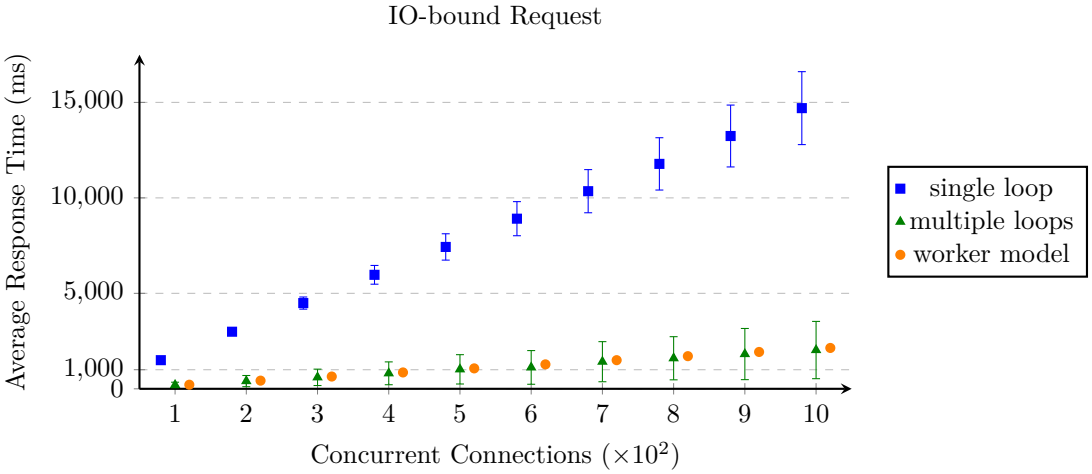

Figure 8.8 Dispersion in response times in IO-bound test. Points are shifted to improve readability.

8.5 I/O-bound

Another common type of operation for web servers is IO-bound tasks. For instance, when a user requests a list of movies, data must be retrieved from a database. Retrieval involves several delays: network latency, as the database is likely hosted on a separate server, and processing time required for the database server to handle the request. This delays whole movie list retrieval operation and the current thread executing this request must wait for the result.

In our experiment, instead of deploying an actual database, we will simulate it by introducing a delay in our code. While the straightforward approach might be to use `sleep`, this method could introduce an unwanted "variable". Specifically, the operating system might choose to put the thread to sleep and reschedule it later, which could influence our measurements.

We will simulate delays using a simple loop that spins for a specified duration. To prevent the compiler from optimizing away this loop, we have used `volatile` variable. This loop is designed to busy-wait, holding the thread active for a randomly generated duration of 10-20 milliseconds.

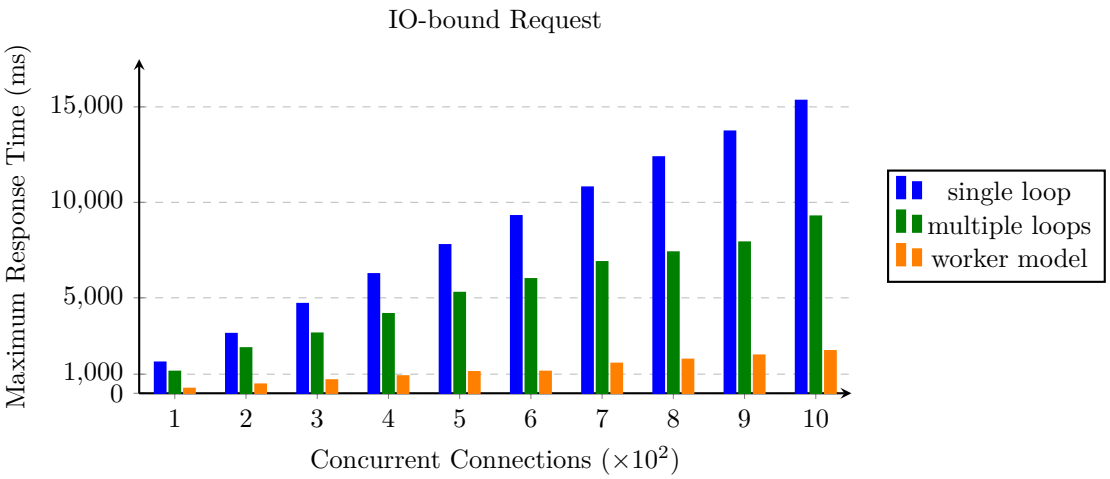

Figure 8.9 Maximum response times in IO-bound test.

```
while (chrono::system_clock::now() < end_time) {
  dummy = 0; //volatile variable
}
```

Code listing 8.5 `Loop used to simulate a delay for an I/O request.`

Before running the tests, we executed the code with optimizations enables to verify that the delay was not optimized out. Making sure, our measurements are correct.

The difference compared to a CPU-bound task is that the IO-bound task consists of one big task that suspends the thread for a specific duration. Unlike in CPU-bound benchmark, it does not create any additional sub-problems. In our setup, all tasks in the pool are requests. Once a worker starts executing a task, it busy-waits by spinning for a predetermined amount of time.

This execution is similar in the single loop and the multiple loop variant, except there is no worker pool and once a thread starts processing a request it spins for predetermined amount of time, for each request.

Figure 8.7 illustrates that the average response time for the worker model and the multiple loops model are nearly identical. The dispersion in the data, as shown in another Figure 8.8, is similar to the dispersion observed in previous CPU-bound benchmarks. This shows that the ability to subdivide tasks into sub-tasks does not play a significant role in our case. This conclusion is also supported by Figure 8.9, which reflects the maximum response times.

This observation indicates that these types of problems behave similarly to CPU-bound problems. However, instead of busy waiting, the processor could be utilized to perform useful work during this time. With a slight modification to our server implementation, we can try to leverage this property.

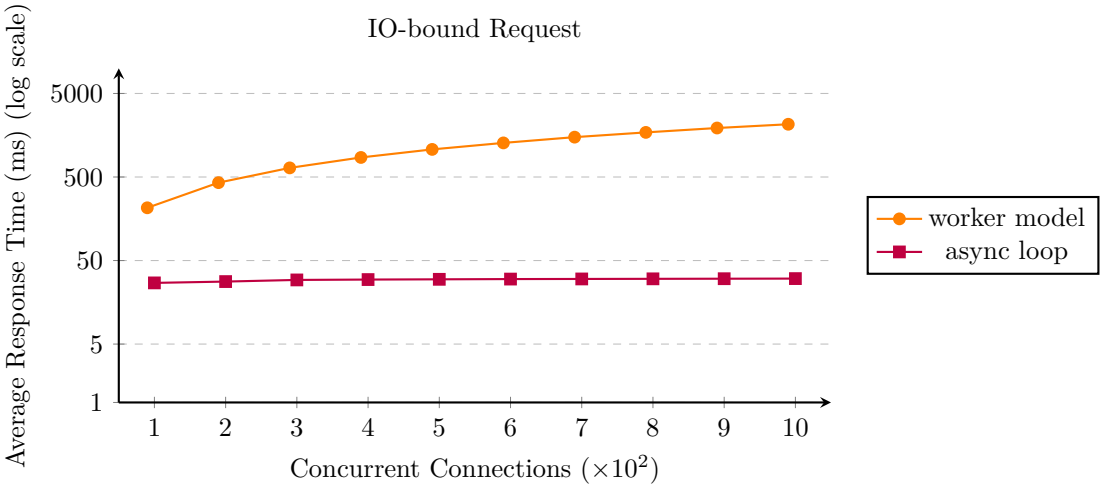

Figure 8.10 Average response times in IO-bound async test.

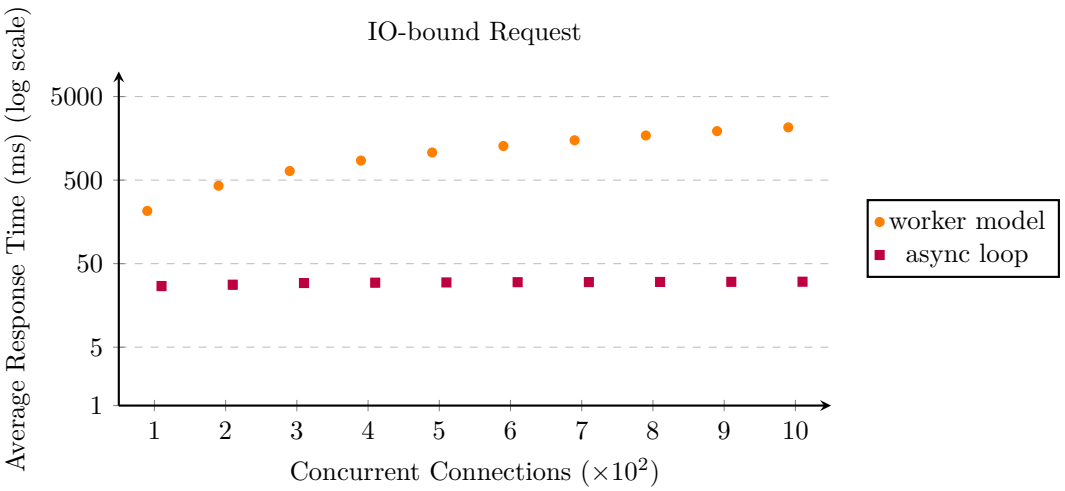

Figure 8.11 Dispersion in response times in IO-bound async test. Points are shifted to improve readability.

8.6 I/O-bound async

In this benchmark, we will use the same type of requests as those in the I/O-bound test. Each request will introduce a delay ranging from 10 to 20 milliseconds to simulate an I/O operation.

As previously mentioned, we will implement minor modifications to our server setup, creating our own asynchronous loop model. We will introduce an awaiter that suspends the current task, and subsequently reschedules it back into the queue. The implementation details for this awaiter are provided in the code attached to this thesis.

This new server implementation is similar to the worker model: we have a single thread that accepts new requests from the epoll instance and places them into a worker pool. The worker pool utilizes global queue scheduling. Basically, a worker takes a request, starts processing it, creates an awaiter, and checks if the predetermined time has elapsed. If not, the task is put back

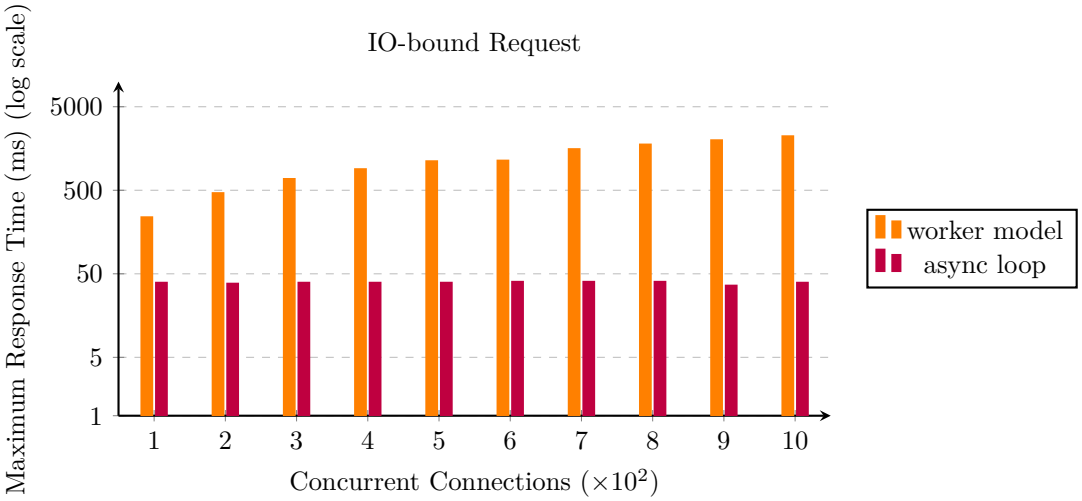

Figure 8.12 Maximum response times in IO-bound async test.

into the queue. This process effectively shuffles tasks to the back of the queue, allowing workers to process new tasks.

Once a worker retrieves a task for which the necessary time has elapsed, the response is sent back to the user and the connection is closed. This mimics the behavior of an I/O operation.

Results presented in Figure 8.10 clearly demonstrate that the asynchronous loop model outperforms the worker model. In the Performance chapter, we've shown that global queue scheduling has larger overhead compared to work-stealing scheduling. Despite this, the asynchronous model, even with its global queue scheduling, still outperforms the worker model.

The dispersion in data for the asynchronous loop model is minimal, as illustrated in Figure 8.11. Similar results were observed in previous tests for the worker model, indicating that both server variants provide a consistent user experience in terms of response delays. Given small average response time and small dispersion, maximal response values are also small for async model demonstrated by Figure 8.12.

Although our async loop model is not a fully fledged asynchronous event loop, in the sense that there are inefficiencies that can be addressed. For instance, it may be more effective to keep a task within the asynchronous loop and iterate over it until completion, rather than rescheduling the task each time it is checked. We believe that this would bring even greater performance benefit.

8.7 Summary

We conducted multiple test to asses the performance of different thread pool implementations. Static content benchmark demonstrated that the work-stealing scheduling did not outperform the multiple loops model. However, in the CPU-bound and I/O-bound tests the work-stealing model showed benefits compared to multiple loops model. Here, the work-stealing model has significantly lower dispersion in response times, which can have a huge impact on user experience. These findings suggest that the work-stealing scheduler offers notable benefits in scenarios involving more CPU-intensive tasks.

In our IO-bound tests, we observed that even a provisional asynchronous loop outperforms the work-stealing scheduler for IO-bound tasks. The difference heavily depends on the blocking duration of individual tasks.

Our testing shows that no single scheduling strategy excels across all types of web server

requests. For an effective server scaling strategy, it is crucial to pick the proper scaling model according to type of requests.

Chapter 9
Conclusion

To design and implement our own version of a thread pool, we began with a research of the currently available tools and libraries. Some libraries, such as TaskfLow, offered novel approaches, focusing on ease of use in expressing problems through graph parallelism. Others offered a more traditional task parallelism approach. The CppCoro library was a pioneer in using coroutines for task-based parallel programming while they were still under partial implementation. However, development of this library has since been abandoned.

We decided to implement our own thread pool library using coroutines, as they are now supported by all major compilers and have been incorporated into the standard. Utilizing coroutines as wrappers for tasks enables coroutine to coroutine transfer, providing an efficient way for task management. However, the specifications for coroutines in the standard can make their implementation complex, adding a level of difficulty.

Another important design decision involved selecting an appropriate scheduling algorithm. With a mechanism for tasks suspension and task encapsulation facilitated by coroutines, we were able to easily experiment with multiple variants of our thread pool implementation. These ranged, from simpler approaches using mutexes to more complex ones using lock-free data structures. Subsequently, we compared our implementation against other popular multithreading libraries to assess its performance. Benchmarks showed that our thread pool has significantly lower scheduling overhead compared to OpenMP implementations and comparable results to oneTBB, even surpassing it in one of the tests.

To effectively test our thread pool implementation against various types of server requests, we implemented our own simple server. This server utilizes Epoll for efficient management of connections.

Finally, we tested various thread pool implementations against various types of server requests. The results confirmed that there is no 'one-size-fits-all' solution; different scheduling strategies are suited to different types of requests. Individual implementations varied in terms of average response times, dispersion of data, and maximum response times. These metrics clearly demonstrated that the choice of a scheduling strategy can have a significant impact on user experience, highlighting the importance of selecting the appropriate thread pool configuration based on the specific demands of the server environment.

Bibliography

1. CONTRIBUTORS, Wikipedia. *Pthreads - Wikipedia* [online]. 2024. [visited on 2024-04-01]. Available from: `https://en.wikipedia.org/wiki/Pthreads`.
2. MICROSOFT. *CreateThread function (processthreadsapi.h) - Win32 apps* [online]. 2023. [visited on 2024-04-01]. Available from: `https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread`.
3. INC., Apple. *Apple Tech Talks Video* [online]. 2024. [visited on 2024-04-01]. Available from: `https://developer.apple.com/videos/play/tech-talks/110147/`.
4. MICROSOFT. *Processes and Threads - Win32 apps* [online]. 2021. [visited on 2024-04-01]. Available from: `https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads`.
5. LABS, Linux Kernel. *Processes - Linux Kernel Labs* [online]. [N.d.]. [visited on 2024-04-01]. Available from: `https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html`.
6. DEVELOPERS, Linux Kernel. *CFS Scheduler Design - Linux Kernel Documentation* [online]. [N.d.]. [visited on 2024-04-01]. Available from: `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`.
7. BLUEPRINT, Code. *Linux Preemption, Latency, Throughput* [online]. 2019. [visited on 2024-05-01]. Available from: `https://www.codeblueprint.co.uk/2019/12/23/linux-preemption-latency-throughput.html`.
8. KHAN, Imran. *Task Priority - Oracle Linux Blog* [online]. 2023. [visited on 2024-05-01]. Available from: `https://blogs.oracle.com/linux/post/task-priority`.
9. TEAM, Go Dev. *Go: The Go Programming Language* [online]. [N.d.]. [visited on 2024-04-01]. Available from: `https://go.dev`.
10. DEVELOPERS, Tokio. *Tokio - Asynchronous runtime for Rust* [online]. [N.d.]. [visited on 2024-05-01]. Available from: `https://tokio.rs`.
11. LAMPORT, Leslie. How to Make Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE TRANSACTIONS ON COMPUTERS*. 1979, vol. C-28, no. 9, pp. 690–691. Available also from: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1675439&tag=1`.
12. PETER SEWELL Susmit Sarkar, Luc Maranget. *Supplemental Test Document 7* [online]. 2012. Available also from: `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`.

13. LAMPORT, L. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*. 1997, vol. 46, no. 7, pp. 779–782. ISSN 0018-9340. Available from DOI: 10.1109/12.599898.

14. OWENS, Scott; SARKAR, Susmit; SEWELL, Peter. A Better x86 Memory Model: x86-TSO. In: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2009, pp. 391–407. ISBN 9783642033599. ISSN 1611-3349. Available from DOI: 10.1007/978-3-642-03359-9_27.

15. PULTE, Christopher; FLUR, Shaked; DEACON, Will; FRENCH, Jon; SARKAR, Susmit; SEWELL, Peter. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages*. 2017, vol. 2, no. POPL, pp. 1–29. ISSN 2475-1421. Available from DOI: 10.1145/3158107.

16. TEAM, The Rust Programming Language. *Atomics - The Rust Programming Language Nomicon* [online]. [N.d.]. [visited on 2024-04-01]. Available from: https://doc.rust-lang.org/nomicon/atomics.html.

17. *ISO/IEC 14882:2020 Programming languages – C++* [International Standard]. Geneva, Switzerland: International Organization for Standardization, 2020. Available also from: https://www.iso.org/standard/79358.html.

18. *Memory order* [online]. 2023. [visited on 2024-04-01]. Available from: https://en.cppreference.com/w/cpp/atomic/memory_order.

19. BATTY, Mark; OWENS, Scott; SARKAR, Susmit; SEWELL, Peter; WEBER, Tjark. Mathematizing C++ concurrency. In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2011. POPL '11. Available from DOI: 10.1145/1926385.1926394.

20. WIKIPEDIA CONTRIBUTORS. *Mutual exclusion* [online]. 2023. [visited on 2023-12-01]. Available from: https://en.wikipedia.org/wiki/Mutual_exclusion.

21. *perf(1) - Linux man page* [online]. 2023. [visited on 2024-01-01]. Available from: https://man7.org/linux/man-pages/man1/perf.1.html.

22. WIKIPEDIA CONTRIBUTORS. *Comapre-and-swap* [online]. 2023. [visited on 2023-12-01]. Available from: https://en.wikipedia.org/wiki/Compare-and-swap.

23. WIKIPEDIA CONTRIBUTORS. *Futures and promises* [online]. 2023. [visited on 2023-12-01]. Available from: https://en.wikipedia.org/wiki/Futures_and_promises.

24. *GNU Compiler Collection (GCC) libgomp* [online]. [N.d.]. [visited on 2023-12-01]. Available from: https://github.com/gcc-mirror/gcc.

25. KOWALKE, Oliver. *Boost.Coroutine* [online]. 2009. [visited on 2023-12-01]. Available from: https://www.boost.org/doc/libs/1_84_0/libs/coroutine/doc/html/index.html.

26. *OpenMP 5.2 Specification* [online]. 2021. [visited on 2023-12-01]. Available from: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

27. CORPORATION, Intel. *oneTBB Developer Guide* [online]. 2023. [visited on 2023-12-01]. Available from: https://oneapi-src.github.io/oneTBB/main/tbb_userguide/title.html.

28. CORPORATION, Intel. *oneTBB — oneAPI Specification 1.3-rev-1* [online]. 2022. [visited on 2023-12-01]. Available from: https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html.

29. HALPERN, Pablo. *Work Stealing* [online]. 2015. [visited on 2023-12-01]. Available from: https://www.youtube.com/watch?v=iLHNF7SgVN4.

30. HUANG, Tsung-Wei. *A General-purpose Parallel and Heterogenous Task Programming System using C++* [online]. 2020. [visited on 2023-12-01]. Available from: https://www.youtube.com/watch?v=iLHNF7SgVN4.

31. CONTRIBUTORS, Taskflow. *Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System* [online]. [N.d.]. [visited on 2023-12-01]. Available from: https://github.com/taskflow/taskflow.
32. BAKER, Lewis; CONTRIBUTORS. *cppcoro: A library of C++ coroutine abstractions for the coroutines TS* [online]. [N.d.]. [visited on 2023-12-01]. Available from: https://github.com/lewissbaker/cppcoro.
33. STELLAR-GROUP. *HPX: The C++ Standard Library for Parallelism and Concurrency* [online]. [N.d.]. [visited on 2023-12-01]. Available from: https://github.com/STEllAR-GROUP/hpx.
34. DOCUMENTATION, HPX. *ParalleX—a new execution model for future architectures* [online]. [N.d.]. [visited on 2023-12-01]. Available from: https://hpx-docs.stellar-group.org/branches/master/singlehtml/index.html#parallex-a-new-execution-model-for-future-architectures.
35. MICHAŁ DOMINIAK Georgy Evtushenko, Lewis Baker el al. *P2300R7: std::execution* [online]. 2023. [visited on 2024-01-01]. Tech. rep. Available from: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r7.html#intro.
36. NISHANOV, Gor. *P0913R0: Add symmetric coroutine control transfer* [online]. 2018. [visited on 2024-01-01]. Tech. rep. ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee. Available from: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0913r0.html.
37. BAKER, Lewis. *C++ Coroutines: Understanding Symmetric Transfer* [online]. 2020. [visited on 2024-01-01]. Available from: https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer.
38. DESROCHERS, Cameron. *concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++11* [online]. [N.d.]. [visited on 2024-01-01]. Available from: https://github.com/cameron314/concurrentqueue.
39. CHASE, David; LEV, Yossi. Dynamic circular work-stealing deque. In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005. SPAA05. Available from DOI: 10.1145/1073970.1073974.
40. LÊ, Nhat Minh; POP, Antoniu; COHEN, Albert; ZAPPA NARDELLI, Francesco. Correct and efficient work-stealing for weak memory models. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013. PPoPP '13. Available from DOI: 10.1145/2442516.2442524.
41. FRIGO, Matteo; LEISERSON, Charles E.; PROKOP, Harald; RAMACHANDRAN, Sridhar. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*. 2012, vol. 8, no. 1, pp. 1–22. ISSN 1549-6333. Available from DOI: 10.1145/2071379.2071383.
42. *time(1) - Linux man page* [online]. 2023. [visited on 2024-01-01]. Available from: https://man7.org/linux/man-pages/man1/time.1.html.
43. *poll(2)* [online]. 2023. [visited on 2024-02-01]. Available from: https://man7.org/linux/man-pages/man2/poll.2.html.
44. *select(2)* [online]. 2023. [visited on 2024-02-01]. Available from: https://man7.org/linux/man-pages/man2/select.2.html.
45. *epoll(7)* [online]. 2023. [visited on 2024-02-01]. Available from: https://man7.org/linux/man-pages/man7/epoll.7.html.
46. CONTRIBUTORS, Wikipedia. *Thundering herd problem* [online]. 2023. [visited on 2023-02-01]. Available from: https://en.wikipedia.org/wiki/Thundering_herd_problem.
47. *accept(2) - Linux manual page* [online]. 2023. [visited on 2023-02-01]. Available from: https://man7.org/linux/man-pages/man2/accept.2.html.

48. NGINX, Inc. *NGINX | High Performance Load Balancer, Web Server, Reverse Proxy* [online]. [N.d.]. [visited on 2024-03-01]. Available from: `https://www.nginx.com/`.

49. FOUNDATION, The Apache Software. *Apache HTTP Server Project* [online]. [N.d.]. [visited on 2024-03-01]. Available from: `https://httpd.apache.org`.

50. FOUNDATION, The Apache Software. *mod_fcgid - A FastCGI implementation for Apache HTTP Server* [online]. 2024. [visited on 2024-03-01]. Available from: `https://httpd.apache.org/mod_fcgid/`.

51. NGINX, Inc. *Module ngx_http_fastcgi_module* [online]. [N.d.]. [visited on 2024-03-01]. Available from: `http://nginx.org/en/docs/http/ngx_http_fastcgi_module.html`.

52. FOUNDATION, Node.js. *Node.js* [online]. [N.d.]. [visited on 2024-03-01]. Available from: `https://nodejs.org/en`.

53. FOUNDATION, The Apache Software. *Multi-Processing Modules (MPM)* [online]. 2024. [visited on 2024-03-01]. Available from: `https://httpd.apache.org/docs/2.4/mpm.html`.

54. *ab - Appache HTTP server benchmarking tool* [online]. 2023. [visited on 2024-02-01]. Available from: `https://httpd.apache.org/docs/2.4/programs/ab.html`.

55. *WRK* [online]. 2023. [visited on 2024-02-01]. Available from: `https://github.com/wg/wrk`.

56. *siege(1)* [online]. 2023. [visited on 2024-02-01]. Available from: `https://linux.die.net/man/1/siege`.

Contents of the Attached Medium

- `code` ..... Folder with source code
  - `benchmarks` ..... Source code for individual tests
  - `coros` ..... Source code for Coros library
- `thesis` ..... Folder with text for the thesis
  - `thesis_latex` ..... LaTeX code for the thesis
  - `thesis.pdf` ..... Text of the thesis