INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Teste e depuração de software concorrente - Desafios e Soluções

HUGO ALEXANDRE SANTOS SILVA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Testing and Debugging Concurrent Software – Challenges and Solutions

Hugo Silva

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Advisor: Luís Nogueira

Porto, June 26, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 26, 2025

Hugo Alexandre Santos Silva

Abstract

Concurrency refers to the execution of multiple entities simultaneously, and due to current technological advancements, it has seen incorporation of its use in many different types of software development, ever-growing in popularity due to its useful and unique capabilities. Due to its higher complexity when compared to standard sequential processes, concurrency introduces non-determinism to the software world, which can cause new problems to arise, necessitating specialized approaches for correct detection.

Despite its usage growth, concurrency concerns are more often than not considered with dedicated urgency, often trusted by system/software architectures or operating systems to correctly manage the non-determinism spectrum of the process. Consequently, developers and testers often lack a comprehensive understanding of various concurrency defects, their associated failure conditions, and events.

While many languages support a set of tools for concurrent development, as well as some available debuggers and IDE's on the market having concurrent defect identification capabilities, most approaches are not really used or known in a real-world scenario or are only briefly explained to developers. Thus, many developers and testers remain unaware of their existence and potential benefits.

We aim to deliver a more in-depth view of concurrent development and concurrent defect detection and avoidance for developers and testers by delving into these language tools, concurrent development techniques and concurrent capable debuggers and uncover further usages besides simple cases and show how these tools can contribute to better designed concurrent systems.

Keywords: Concurrency, Defects, Debuggers, Tools, Techniques, Software Development

Contents

List of Figures

List of Source Code

Chapter 1

Introduction

This chapter aims to introduce the reader to the context and problem at hand, as well as give brief descriptions about the testing process and goals of the overall scope

This chapter contains a description of the context of the problem at hand alongside the project's goals, a list of ethical considerations and the organization methodology adopted for the project.

1.1 Context and Problem

Concurrency, the simultaneous execution of multiple entities, is an omnipresent reality in systems and software engineering, adding a layer of complexity that profoundly affects testing. It introduces non-deterministic behaviour and a myriad of concurrency defects, necessitating specialized approaches for detection.

Concurrency and parallelism are integral to a wide range of domains, from web-distributed systems to video games and animation engines. For example, web systems leverage concurrency to enhance user experience, while gaming and animation rely on concurrent and parallel processing to efficiently utilise GPU and CPU cores. However, developing concurrent systems involves a trade-off: sacrificing logic simplicity and dynamic storage in favour of time efficiency, which introduces complex challenges.

Despite its widespread use, there is a persistent misconception among developers that concurrency issues are inherently associated with operating systems or software architectures and thus inherently addressed by theses systems. In truth, these systems do have some features that help mitigate possible concurrent problems, but are only an additional helping resource to developers. This assumption leads to a lack of awareness and understanding of concurrency related defects, their conditions, and the events that trigger them.

Concurrency presents several significant challenges. Their inherently complex nature makes them difficult to design and debug. The non-deterministic behaviour in execution timing caused by factors like CPU/GPU loads, network traffic or communication pipelines can lead to inconsistent results and new types of defects unique to distributed systems, such as race conditions, heisenbugs and deadlocks caused by misusage of concurrent components. From a perspective of a sequential debugger, the aforementioned defects are impossible to detect.

Although academic research has introduced numerous testing techniques and algorithms for managing concurrent components, most of these innovations are ignored in real-world software development. This gap leaves many developers and testers unaware of these solutions and their potential to mitigate concurrency-related issues. There are also some defect

discovery tools for concurrent environments, although most of them don't see much use in the professional world due to their complexity and misuse caused by misinformation and misconceptions of the common developer.

By addressing these challenges and bridging the gap between academic research and practical application, developers can better manage the complexities of concurrent systems and improve software reliability.

1.2 Goals

This thesis aims to investigate various concurrency models, their associated types of concurrency defects, and key debugging and testing techniques to uncover these defects. Consequently, this thesis dives into concurrent development, design techniques, and debug tools currently available to mitigate defects.

The ultimate goal of this document is to equip developers and testers with valuable insights and raise awareness about current development practices and tools available for testing and debugging concurrent systems. It emphasizes the use of language built-in features and debuggers that can enhance traditional testing methodologies and facilitate the detection of concurrency-related issues. By incorporating test cases and debugging environments tailored to concurrent systems, developers can strengthen their software development and testing workflows.

Due to time constraints, the documentation and testing efforts are focused on a specific subset of relevant languages and tools that are widely used in professional settings. Nonetheless, the concepts and theories explored in this paper are largely universal and can be adapted to other environments beyond those directly tested.

1.3 Test Process

Two main types of tests are employed: one dedicated to techniques and language-built support tools and general concurrency development practices, and the other specifically for debuggers. This division is intended to emphasise that concurrent bugs, such as data races and deadlocks, often arise from development oversights or poorly designed concurrent systems. In reality, most modern programming languages evolve dynamically to support concurrency development, with each iteration introducing safer and simpler methods for managing concurrent systems.

As previously mentioned, the first batch of tests will evaluate built-in tools and their utility in developing robust and bug-free software. These tests will highlight the advantages and disadvantages of the tools selected. The critical factors considered in these tests will be the robustness of the solutions and performance.

The second batch of tests will be conducted on a set of programs known to contain concurrent defects, such as race conditions and deadlocks, focusing solely on the debuggers. These tests aim to evaluate each debugger's effectiveness, ease of use, flexibility in data visualization, performance and overhead introduced from additional debug data.

It is also important to note that every test conducted, whether applied to tools or debuggers, will be described step-by-step. This approach ensures detailed documentation of each tool

and debugger capabilities and particularities, while also enabling readers to reproduce the test cases if desired.

1.4 Ethical Considerations

Given the project's objectives, the ethical framework primarily emphasizes the integrity of the work presented and the potential consequences of misinformation on distributed software systems and their associated business models. As the project heavily depends on investigating recent advancements in concurrent debugging, meticulous care is required in collecting and analyzing information to ensure the accuracy and reliability of the conclusions drawn.

During the planning phase, several engineering and professional codes of conduct were reviewed to align the project's ethical practices with its requirements. Three key codes of conduct were selected, with the project primarily adhering to the IPP Code of Conduct [1]. Within this framework, particular attention was given to Articles 6, 8, and 10, as they align with the project's emphasis on truthfulness and objectivity.

Articles six and eight strictly prohibit plagiarism and mandate proper identification for all information, documents, or software utilized during the project. The use of any external work must be appropriately credited, and the submission of false statements is explicitly forbidden.

Article 10 underscores the importance of ensuring the accuracy of all statements in the project. It requires verification of the literature collected and tests conducted. Furthermore, it mandates that data analysis and results obtained be presented in a manner that is impartial, objective, and straightforward.

Other codes of ethics, such as the NSPE Code of Ethics for Engineers [2] and the ACM/IEEE-CS Software Engineering Code [3], were also considered. These prioritise security and practical development conducts, ensuring the software developed is incapable of causing machine, human or economic harm. Given the inherent unpredictability of distributed systems, the project places special emphasis on identifying and addressing risk factors. These include the potential harm to physical systems caused by undetected bugs, which may lead to scenarios such as device soft-locking and other problems that potentially could cause revenue loss to those affected.

By adhering to these ethical principles, the project aims to uphold the highest standards of integrity and responsibility, ensuring that its contributions are both reliable and ethically sound.

Chapter 2

State of the Art

This chapter is entirely dedicated to the study of Concurrency and the many shapes and forms it takes in the development world. Firstly, a brief introduction about concurrency is given. Then it proceeds in studying bugs and defects prevalent and inherent to concurrency in order to contextualise the necessity of debuggers and specific development techniques for concurrent systems.

Finally, the document highlights the current built-in language tools available for concurrent development in C, C++, Java, C#, and Go. In addition, each language-specific section explores recent advancements and ongoing research related to external tools and emerging debuggers that focus on analyzing and detecting concurrent behavior. While many of these tools are still in their early stages and not yet widely adopted in professional environments, they represent significant progress in the field of concurrent debugging and development. These new tools mark a shift from traditional debuggers, which primarily emphasise data manipulation, toward solutions that prioritise defect detection and concurrency-specific analysis.

2.1 Research Methodology

This thesis makes use of the PRISMA research methodology, which is structured into three main phases: identification, screening, and inclusion of relevant resources.

2.1.1 Identification Phase

The identification phase involved constructing and applying strategic search strings within scientific article databases to locate potential resources related to concurrency and concurrency defects. This process began with defining a focused research string, which was then used across multiple libraries to gather resources. Duplicate entries across sources were also removed during this phase. Emphasis was placed on articles, journals, and books during research.

The research string used was:

- *Concurrency AND Concurrent Debugging AND >= 2022*

It was then applied in the following libraries:

- **ACM Digital Library:** 1504 results
- **Science Direct:** 477 results

The SJR (Scientific Journal of Rankings), while used, was very limited in its search functionality rendering results largely irrelevant, with most articles unrelated to concurrency. As such, no resources from this source were included.

In total, 1,981 resource papers were initially collected, most of which addressed concurrency in general or sequential debugging rather than the specific focus of concurrency defects.

2.1.2 Screening Phase

The screening phase focused on analyzing resource titles and abstracts to identify resources that could contribute meaningfully to the study of concurrency defects. The selection aimed to avoid overly specific tools or technologies to ensure broader applicability.

The resources were categorized into three primary topics

- **Concurrency Theory:** Focused on theoretical frameworks for concurrent development and architectures, this topic served to provide foundational knowledge in the State of the Art section and emphasized the importance of understanding concurrency for effective usage of debugging techniques and tools. Information obtained within this scope can be generally applied to every technology that supports concurrency, with many programming languages already having native support for concurrent development.
- **Concurrent Defects and Best Practices:** In-depth study of defects such as variable races, locks, atomic blocks and other miscellaneous problems that are inherently associated with concurrency. Also studied universal techniques for mitigating these defects, which are critical for effective concurrent software development. Understanding these defects and development techniques heavily helps in developing concurrency software and so, it is of utmost importance to be included.
- **Concurrent Bug Discovery Tools:** Study of reviewed solutions proposed by recent studies on debuggers designed for concurrency defect analysis and tools aligned with these identified defects, while avoiding overly specific or narrowly applicable tools to maintain relevance across a broader scope of stakeholders.

From the initial pool of 1,981 results, a title and abstract analysis reduced the selection to 23 potential results across the three topics.

2.1.3 Inclusion Phase

A detailed literature review of the 23 shortlisted articles determined the final set of resources included in the thesis. The results for each topic were as follows:

- **Concurrency Theory:** Five articles were shortlisted, but only one was included in the thesis. This limited selection reflects the focus of the study, which is not concurrency theory itself but its application in addressing concurrency defects. A single, comprehensive source was deemed sufficient to introduce concurrency and contextualise subsequent findings.
- **Concurrent Defects and Best Practices:** Many articles in this category overlapped with studies on concurrent debugging tools. Three articles were individually evaluated and excluded as their content was already covered in-depth by other selected resources. Ultimately, five resources addressing defects were included in total.

- **Concurrent Bug Discovery Tools:** Articles on debugging tools constituted the majority of the references. Of the 17 initially shortlisted articles, 15 were included in the final reference list. These covered diverse tools targeting a range of concurrent defects, from race conditions to atomic block analysis. The larger number of articles in this category reflects the study's objective to provide comprehensive insights into various tools and techniques.

In total, 16 resources were selected for inclusion in the thesis and incorporated into the State of the Art section. This selection ensured a balanced representation of theoretical, practical, and tool-based approaches to addressing concurrency defects.

2.2 Concurrency

This thesis draws significantly on the definitions presented in Stephen Cleary's book [4], which describes concurrency as a broad topic that encompasses systems capable of performing *"more than one thing at a time"*.

Concurrency is a general concept that encompasses various software designs and architectures. A system is considered concurrent or distributed when its processes do not follow a strictly sequential and predefined execution order. The primary goal of concurrency is to enhance system performance, often at the cost of other resources, such as memory.

This section aims to bring awareness and explain different widely used concurrency models in professional concurrent development and dive into multiprocessor's persistent and relaxed memory models, since they can affect the behaviour of concurrent systems.

2.2.1 Memory Models

An additional consideration is whether the concurrent system is employed on a processor with persistent or relaxed memory models. This distinction significantly influences the execution of processes due to different manners of memory management between models.

The PMEM-Spec (Persistent Memory Speculation) study [5] characterizes persistent memory models as systems that establish a "persist-order that controls the order in which stores update persistent memory" to manage shared memory in concurrent environments. Consequently, systems can be divided in consistent behavioral patterns and thus persistent models are suitable for greater control and reduced flexibility over concurrent actions such as read-/write operations. Although less popular, persistent models still have space in the market, as PMEM-Spec demonstrates that in some cases, persistent models are better suited for development and can actually be faster than relaxed models.

On the other hand, relaxed memory models make use of hardware and compiler optimizations [6] in order to achieve better performance. These approaches result in a complex and non-linear way of managing shared memory, and though it may not affect the behaviour observed of sequential code, the application of relaxed memory models in concurrent environments can lead to significant issues if the system lacks robust synchronization mechanisms. In comparison to persistent memory models, relaxed models are more appealing in the current market with major multiprocessors such as x86, ARM and languages such as C, C++ and Java utilizing said relaxed models [7].

2.2.2 Concurrency Models

Different concurrency models were developed in order to tackle specific problems and bring a universal language to concurrent development. This thesis explores the following models:

- Thread Model
- Parallelism Model
- Future Model
- Coroutine Model
- Actor Model
- CSP Model

Each of these contain several advantages, disadvantages and uses cases in the concurrent scope, with each language deciding to adopt a set of models to create their own concurrency tools. Understanding the differences between them is crucial for efficiently developing a distributed environment that meets the specific needs of a project.

Languages such as Go and Erlang invest heavily in message-based models such as the CSP and Actor models while languages such as C++ and Java focus more on extensive tools for thread and parallel management. The tools available for each language at study are later analyzed in the Debugging concurrency 2.4 chapter.

Thread Model

The concept of concurrency is often closely associated with threads and multi-threaded systems, alongside the principle of parallelism. Fundamentally, threads function as "independent executors" [4], capable of performing specific tasks required by a process, which is a running instance of a program. Processes frequently employ numerous threads, collectively referred to as thread pools, to distribute workloads across different components of the program. In this context, a sequential program can be regarded as a single-threaded program, where the process utilizes only one thread, the "main thread".

A key capability of multi-threaded design over a sequential approach lies in the ability of threads to perform context switching within a process. This allows a thread to pause its execution at any given point and resume it later as dictated by the scheduler. A scheduler, as defined, "allows one process to use the CPU while the execution of another process is on hold (in a waiting state) due to the unavailability of resources such as I/O, thereby ensuring full utilization of the CPU" [8]. Scheduling mechanisms are typically dependent on the operating systems and extend to managing threads as well, since all threads are encapsulated within processes.

Figure 2.1 illustrates the behaviour of a multi-threaded system within a process. In the example, three threads, each assigned specific tasks, alternate execution through context switching. While concurrency conveys the notion of performing multiple tasks simultaneously, the execution of a multi-threaded process remains fundamentally sequential when applied to a single-core processor. As demonstrated, no two threads are active concurrently on a single core; instead, they execute in turns. Threads in a waiting state are labelled as suspended.

The adoption of threads over a purely sequential approach offers significant advantages, particularly in terms of resource utilization and flexibility. Threads are especially beneficial in applications that involve blocking resources, such as input/output operations (e.g., server applications) or graphical user interfaces. This is because a process can switch from a thread that is waiting for input to a thread who is ready to continue execution.

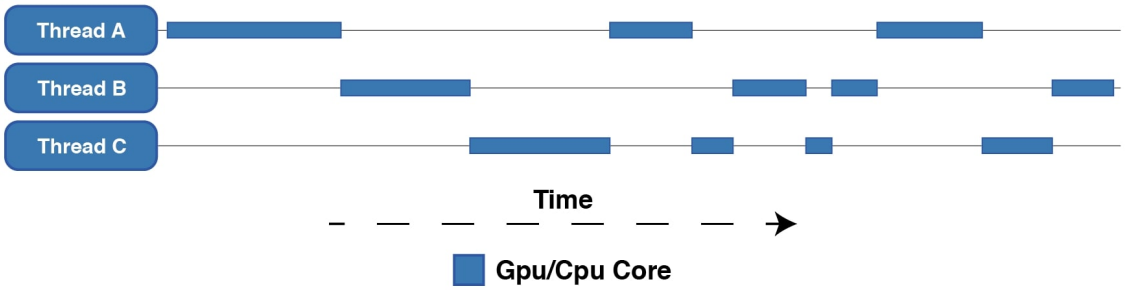

Figure 2.1: Execution of a Multi-Threaded Process

Parallelism Model

Parallelism differs from multi-threading primarily in its execution model. Rather than relying on multiple threads within a single process that are executed in turns, parallelism involves the use of multiple processes or threads that run independently and simultaneously on separate CPU or GPU cores. This approach is particularly well-suited for dividing computational workloads into independent tasks and distributing them across available cores. By doing so, it maximizes CPU/GPU utilization, increasing throughput and enhancing overall performance.

Stephen Cleary's work categorizes parallelism into two primary forms:

- Data parallelism: This form emphasizes managing and processing data. Each piece of data being processed should be largely independent from other pieces.
- Task parallelism: This form involves a pool of tasks, where each task is also mostly independent of the others. Tasks may be dynamic in nature, meaning that "if one task results in several additional tasks, they can be added to the pool of work" [4].

Similar to the thread model described in Section 2.2.2, multiple processes can also run concurrently on the same processor core. These processes are capable of context switching, governed by a scheduler, a mechanism similar to that employed in the thread-based model.

Parallelism is particularly prevalent in rendering graphically intensive applications. By distributing workloads across the numerous cores of a GPU, parallelism ensures smooth rendering of graphics and efficient handling of complex visual computations. Given the architectural characteristics of GPU cores, the processes executed within these cores are typically simple, often mathematical in nature, and share minimal memory, if any.

Additionally, many servers adopt the parallel model to process multiple user requests simultaneously. Each request is handled independently, resulting in greater efficiency and reduced latency when serving concurrent users compared to a sequential approach.

Figure 2.2 provides a visual representation of how a parallel system, utilizing multiple processes and cores, computes tasks over time in comparison to the thread model. It is worth noting that the figure assumes each core runs only one process at a time. However, if Core 1, for instance, was executing two processes simultaneously, Process A could be divided into smaller chunks of work over the same time period (similar to figure 2.1), while Processes B and C would continue their respective execution on different cores uninterrupted.

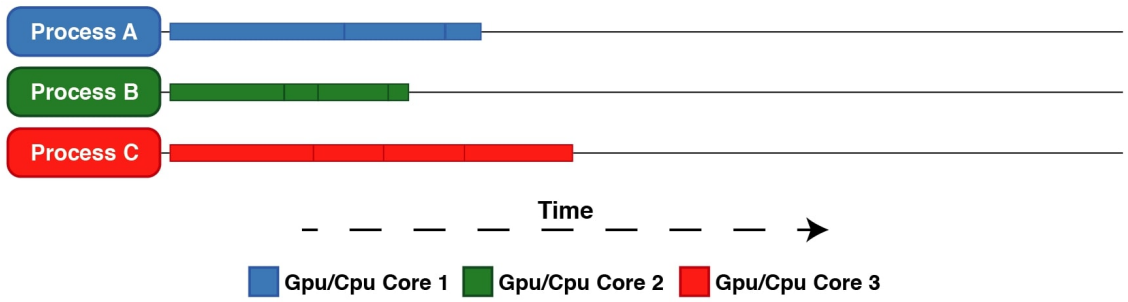

Figure 2.2: Parallel Processes running uniquely on each core

Future Model

Originating from functional programming, the future (or promise) model is built on the concept that "a specific operation will eventually complete in the future" [4]. It assumes that the initial result is unknown, typically because the required computation is still in progress, but a response with the result will arrive eventually. This model introduces architectures and systems to the concept of asynchrony. Asynchronous operations enable tasks to start and run independently of the main execution flow (figure 2.3). This approach enhances responsiveness, allowing applications to remain interactive while performing long-running tasks in the background. It also reduces idle time and simplifies I/O operations. The future model achieves this through callbacks and promises to manage task completion. Notably, it can also operate in single-threaded environments, optimizing CPU usage without the need for multiple threads.

Unlike multithreading or parallelism, the future model emphasizes non-blocking operations, where tasks do not pause the main thread or process while waiting for resources or results. It is event-driven, relying on mechanisms such as events, callbacks, and promises to handle task completion. This model employs specific types, objects, and keywords to create a callback system, enabling methods to exhibit asynchronous behaviour.

Stephen Cleary's book [4] explores the implementation of this model in `C#`, introducing key constructs such as the `async` and `await` keywords, as well as `Promise` and `Future` object concepts that are similarly implemented in other popular programming languages. The `async` keyword is often used in method declarations, allowing asynchronous code execution within the method. It is paired with the `await` keyword, which, as the name suggests, pauses the execution until the promise or future is fulfilled. The `Future` object represents an asynchronous activity that will eventually complete.

This model is particularly well-suited for web servers, enabling them to handle thousands of concurrent requests efficiently. It is also widely used in UI-focused applications to ensure smooth user interactions by keeping interfaces responsive. Additionally, it plays a significant role in data-fetching scenarios, facilitating efficient retrieval of data without blocking the program's execution.

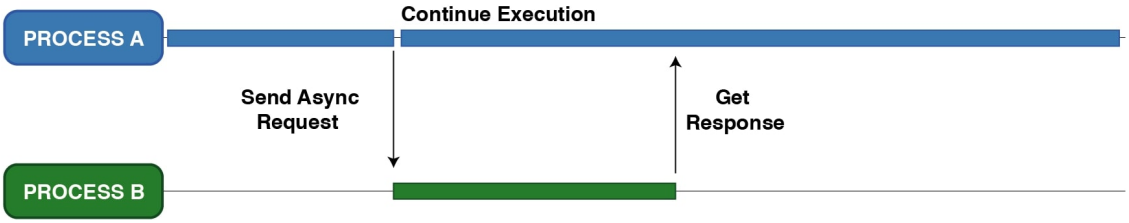

Figure 2.3: Example of asynchronous behaviour between processes

Coroutine Model

The concept of coroutines was first proposed in the 1960s and was initially described as "subroutines that act as the master program" [9]. Coroutines are lightweight instances of suspendable computations, making them somewhat analogous to threads.

Unlike threads, suspended coroutines do not block the overall execution flow. Instead, they can pause their execution and later resume in different contexts. For example, a coroutine might suspend its activity on one thread and resume on another.

Among the models discussed in this section, the coroutine model is perhaps the least seen, as only a few relevant languages have integrated it directly into their toolsets. Languages such as Kotlin and C# provide built-in support for coroutines in their toolset, while others rely on alternative mechanisms to simulate their behaviour. Coroutines are especially significant in software like the Unity game engine and serve as inspiration for Go's goroutines.

The Actor Model

The actor model is a concurrency model that bases itself in the notion that everything is an actor. The big emphasis of the actor model is that *"actors do not communicate by sharing memory, but instead share memory by communicating with each other"* [10]. The model describes the actors as persistent asynchronous entities defined by internal states and addresses. Berb [11] describes these as entities capable of the following:

- **Internal Management**: Actors are capable of taking local decisions upon receiving a message. A simple example of that would be changing an actor's internal state.
- **Scale**: Actors are able to "spawn" new actors in case the application needs more resources at a given point.
- **Send Messages**: Since everything is considered an actor in the model and actors are capable of receiving messages, actors are also able to send new messages to other actors in the system.
- **Prepare for future messages**: Finally, these entities are capable of defining beforehand the behaviour to take upon receiving the next future message.

Addresses and actors are not exclusive, meaning that many actors may have the same address and an actor may have multiple addresses. These addresses contain location and transport information about the actors, which helps in the actor message targeting. Furthermore, messages are sent without intermediary channels, applying a "best-effort" approach and dependent on the pipeline they are using to communicate. Each actor has also an associated "mailbox" (exemplified in figure 2.4) that keeps messages organized within a queue and enable the actor to tackle each message independently.

The model shows itself to be very flexible since the main focus is the sending and receiving of messages between actors. According to the official Akka website [12], an actor model framework for java, these actors can be completely different in behaviour or even live in different machines as long as they are capable of receiving messages and use the memory communicated to them. Consequently, the model is suitable to be applied to asynchronous and reactive concurrent systems, since it bases itself in an asynchronous message-based approach. Taking a look into the practical side of the model, languages such as Erlang and Elixir have native actor-based tools and other widely used languages such as C++, Java and .NET possess frameworks capable of incorporating the actor model into the code base. Below are some examples of aforementioned frameworks:

- Akka (Java)
- Orleans (.NET)
- CAF (C++)
- Celluloid (Ruby)
- Pulsar (Python)

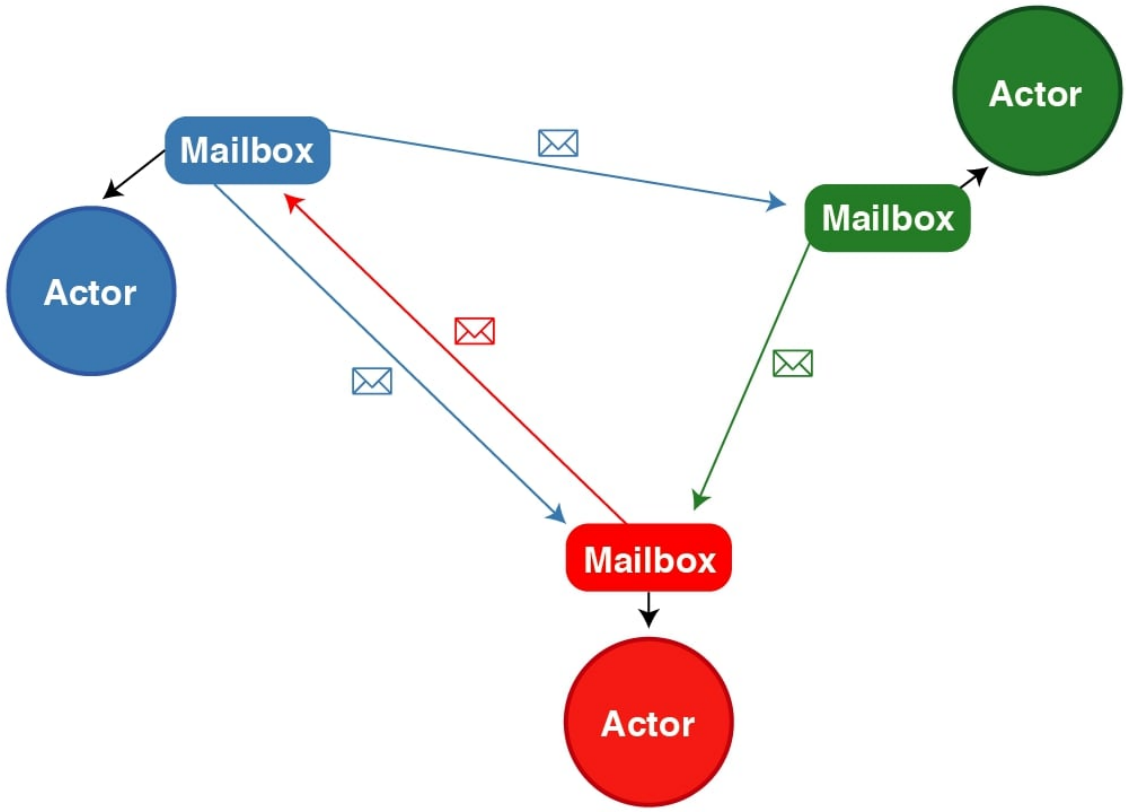

Figure 2.4: Message system between actors

CSP Model

The Communicating Sequential Processes (CSP) concurrency model was first introduced by Tony Hoare in 1978. In his paper [13], Hoare describes processes as independent blocks of computer code that execute independently of one another. In this context, even simple functions can be considered processes. The CSP model shares similarities with the actor

model, as both propose the concept of processes that do not share memory but instead communicate via a message-passing system. However, the key distinction lies in how messages are exchanged: while the actor model uses asynchronous message passing, the CSP model enforces synchronization during inter-process communication.

In CSP, message passing is treated as a fundamental programming primitive [13]. Processes can execute concurrently, synchronizing their actions by waiting for corresponding input or output commands from other processes. These commands are executed sequentially to prevent race conditions and reduce the likelihood of deadlocks. Furthermore, messages are delivered in the exact order in which they were sent, ensuring consistency in communication. Deadlocks can still occur if a process waits for a message that is never sent; however, compared to other types of deadlock scenarios, these are typically easier to identify and resolve due to their simple nature.

Figure 2.5, derived from Dev's article on CSP and the actor model [14], illustrates the typical behaviour of a system implementing the CSP model to achieve concurrency. It highlights the synchronous nature of such systems. As shown, a channel writer blocks execution until a channel reader processes the message. The primary advantage of this blocking mechanism is that the channel only ever needs to hold one message at a time, simplifying communication and resource management.

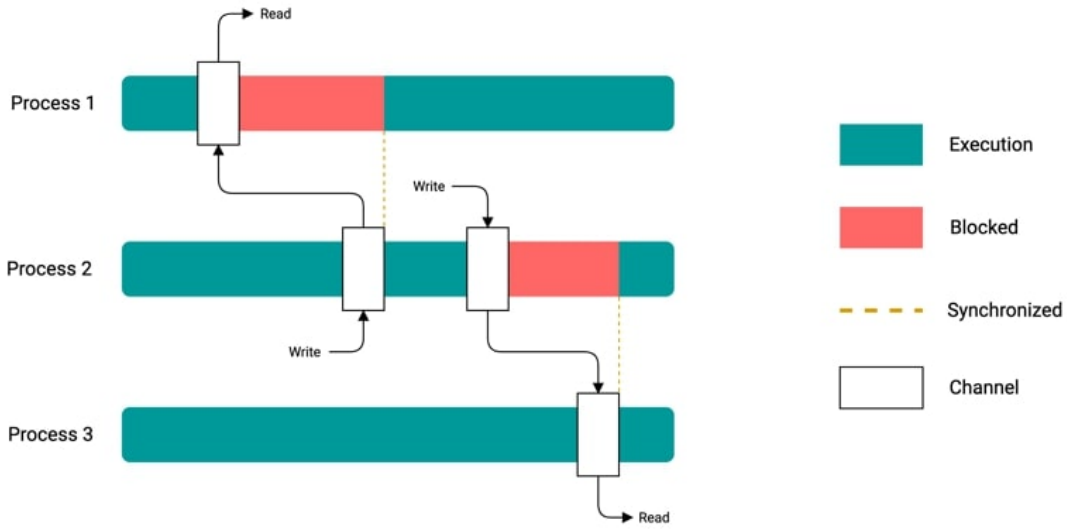

Figure 2.5: Example of concurrent system with CSP model incorporation [14]

2.3 Concurrent Defects

While concurrency introduces opportunities to enhance performance, responsiveness, flexibility, and scalability in distributed systems, it also increases system complexity. This complexity gives rise to unique and challenging defects that occur during concurrent execution. Some defects found in sequential systems are also made worse. Each concurrent design paradigm is susceptible to specific types of defects, making it essential to conduct a thorough analysis of the most prevalent issues in concurrent systems.

This section focuses on identifying and exploring common concurrent defects, progressing from the most recognized and frequent to more abstract challenges. This thesis specifically addresses:

- **Race conditions**
- **Deadlocks**
- **Heisenbugs**

Primary focus on race conditions and deadlocks will be given, along with their debugging techniques. Additionally, it examines a debugging-related challenge of name heisenbugs, which can hinder the effectiveness of general debugging techniques and tools.

2.3.1 Race conditions

Race conditions are one of the most prevalent defects on concurrent programs and are caused when two or more threads or processes access unprotected shared resources (variables, files or memory) at the same time, and the final outcome depends on the unpredictable timing or interleaving of their executions. This in turn causes the program's behaviour to change from run to run, depending on the order in which the threads/processes are schedulled for execution. If left unprotected, race conditions more often than not lead to defects that are hard to reproduce due to their non-determinant behaviour.

The below source code serves to demonstrate a very common misuse of multi-threading causing race conditions between threads. The program implements a simple bank logic where multiple threads can add or subtract quantities of money of a single bank account:

```
#include <iostream>
#include <thread>
#include <chrono>

// shared resource
int balance = 0;

void deposit(int amount)
{
    int temp = balance;
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    temp += amount;
    balance = temp;
}

int main()
{
    std::thread t1(deposit, 30);
    std::thread t2(deposit, 50);

    t1.join();
    t2.join();

    std::cout << "Final Balance: " << balance << "\n";

    return 0;
}
```

Listing 2.1: Race Condition: Bank example (C++)

The program creates two threads, Thread **t1** and Thread **t2**. Thread t1 adds to the account's balance 30 and Thread t2 adds 50. When the program is asked to show what the balance value is at the end, the expected result is 80, which is the addition of 30 from t1 and 50 from t2, respectively.

The same compiled execution file was run 10 times independently, and the results obtained varied from 50 or 30 but never 80. In reality, since the balance variable is a shared and unprotected resource, both t1 and t2 modify the shared resource locally and without synchronization, only storing the last value altered. Since the order of execution of threads is relative and dependent of the scheduler, it is possible for t1 to use the deposit function first or the opposite to happen, and thus explaining the final balance values of 30 and 50. This difference in values is a clear problem to be addressed since faulty read or write operations can lead to wrong balance values for the bank account in this context.

A common solution to potential data races is the incorporation of atomic operations. Atomic operations ensure that shared memory accessed by multiple processes remains secure and synchronized. Typically, atomic operations are reserved for primitive types and provide a limited set of functions and methods. These operations allow resources to be utilized in a manner that guarantees synchronization, even if proper locking and unlocking of resources are not rigorously implemented. Atomic operations are generally executed at the lowest level of the compilation process, making them unusable for complex user-defined classes. Instead, programming languages often associate the atomic reference with a primitive variable types for subsequent use. Once a variable is inferred as atomic, it gains access to functions designed to preserve its atomic state. Referring to the code snippet in 2.1, if the balance value were updated using an atomic operation, the final deposit would always result in a value of 80, regardless of the execution order of threads determined by the scheduler. A simple modification to achieve this would be as follows:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <atomic>

// shared resource
std::atomic<int> balance(0);

void deposit(int amount)
{
    balance.fetch_add(amount);
}

int main()
{
    std::thread t1(deposit, 30);
    std::thread t2(deposit, 50);

    t1.join();
    t2.join();

    std::cout << "Final Balance: " << balance << "\n";

    return 0;
}
```

Listing 2.2: Atomic operations: Fix of the Race condition on Bank (C++)

When implementing the fix, the program creates an atomic integer variable to manage the balance value of the simulated bank. Whenever threads t1 and t2 invoke the deposit function, the method `fetch_add(int value)` ensures that the addition of the specified amount is executed atomically, thereby preventing data races. This guarantees that the final value of balance printed at line 22 will consistently be 80. Atomic primitives provide developers with the ability to designate specific regions of the program where data synchronization is critical. Although user-defined classes cannot directly utilise the atomic primitive, their variables can leverage atomic synchronization when applied to their primitive types.

Locks and atomic primitives serve similar purposes in ensuring shared memory synchronization. However, the distinction lies in the intended operation on the shared memory. Critical sections of a program protected by locks can also incorporate atomic functions. Locked critical areas are typically used to execute logic while ensuring that the data within remains consistent. If shared memory is modified within these critical sections, it remains synchronized after the lock is released, allowing subsequent processes to access it safely. Atomic primitives only ensure that modifications to shared memory occur in a singular, synchronized manner without requiring explicit locking instead.

Race conditions are hard to debug because they might only surface intermittently, under specific timing conditions. In critical systems, such bugs can lead to severe issues like data corruption, crashes, or security vulnerabilities. There is also the possibility on false-positives and false-negatives when debugging data races, where data races do happen but the tools and debuggers are unable to catch it (often due to specific structures) or vice-versa. Debuggers and tools focused on detecting and evaluating race conditions make for the majority of reviewed material in the Debugging Concurrency section 2.4.

2.3.2 Deadlocks

Most programming languages support the notion of locks, where a thread or process locks a resource or region to avoid data races between threads. But this lock solution also brings with it another big problem in the concurrent scope, deadlocks. They occur whenever two or more threads or processes are waiting for each other to release resources, resulting in a permanent state of waiting where none of the entities can proceed, causing the program to halt indefinitely.

For a deadlock to occur, four conditions, otherwise known as the Coffman conditions must occur in simultaneous:

- **Mutual Exclusion:** A resource is held in non-shareable mode, where only one thread can use it at a time.
- **Hold and Wait:** A thread holding one resource is waiting to acquire additional resources held by other threads.
- **No Preemption:** Resources cannot be forcibly removed from a thread, and thus they must be released voluntarily.
- **Circular Wait:** A circular chain of threads exist where each thread waits for a resource held by the next thread in the chain

Below is a program demonstrating a deadlock example in practice:

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex resource1;
std::mutex resource2;

void method1()
{
    // lock resource 1
    std::lock_guard<std::mutex> lock1(resource1);
    std::cout << "Thread 1: Locked Resource 1\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    // wait to lock resource 2
    std::lock_guard<std::mutex> lock2(resource2);
    std::cout << "Thread 1: Locked Resource 2\n";
}

void method2()
{
    // lock resource 2
    std::lock_guard<std::mutex> lock1(resource2);
    std::cout << "Thread 2: Locked Resource 2\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    // wait to lock resource 1
    std::lock_guard<std::mutex> lock2(resource1);
    std::cout << "Thread 2: Locked Resource 1\n";
}

int main()
{
    std::thread t1(method1);
    std::thread t2(method2);

    t1.join();
    t2.join();

    return 0;
}
```

Listing 2.3: Deadlock example (C++)

Similar to the race condition program example 2.1, Threads t1 and t2 are initialized at the same time, with t1 and t2 calling method1 and method2, respectively. Within method1, thread t1 acquires resource1 and locks it, waiting a second to try and acquire resource2 next. At the same time, thread t2 acquires and locks resource2, waits a second and tries to acquire resource1. Since t1 has possession of resource1 and never unlocks it, t2 waits infinitely for resource1 to be available. The same happens with t1 and resource2. Since both t1 and t2 can't continue execution due to waiting forever for resources that will not get unlocked, this causes the program to halt.

A common approach to preventing deadlock is to maintain a consistent order of locking resources, regardless of which thread is accessing them. Although this method provokes a slight performance decrease, it is an effective technique for eliminating deadlock conditions.

Considering this approach, a possible fix for the previously shown code snippet would be as follows:

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <chrono>
5
6  std::mutex resource1;
7  std::mutex resource2;
8
9  void method1()
10 {
11     std::lock(resource1, resource2);
12     std::lock_guard<std::mutex> lock1(resource1, std::adopt_lock);
13     std::lock_guard<std::mutex> lock2(resource2, std::adopt_lock);
14     std::cout << "Thread 1: Locked Both Resources\n";
15 }
16
17 void method2()
18 {
19     std::lock(resource1, resource2);
20     std::lock_guard<std::mutex> lock1(resource1, std::adopt_lock);
21     std::lock_guard<std::mutex> lock2(resource2, std::adopt_lock);
22     std::cout << "Thread 2: Locked Both Resources\n";
23 }
24
25 int main()
26 {
27     std::thread t1(method1);
28     std::thread t2(method2);
29
30     t1.join();
31     t2.join();
32
33     std::cout << "\n";
34
35     return 0;
36 }
```

Listing 2.4: Deadlock fix example (C++)

As previously noted, both Thread t1 and Thread t2 attempt to acquire resource1 and resource2 (lines 11 and 19). Both resources are instances of the mutex class, described as "a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads" [15]. Either t1 or t2 successfully acquires ownership of the resources using `std::lock`, which ensures that one thread must wait until the required resources become available. After successfully locking the resources, t1 or t2 releases the locks at the end of its critical section, enabling the other thread to proceed with locking the resources and proceeding its execution and thereby preventing a deadlock state. This implementation prioritizes the prevention of shared data conflicts over performance, ensuring that data is not simultaneously accessed or modified by multiple threads.

Deadlocks are particularly problematic since they completely halt the execution of the program and possibly escalating to bigger consequences in the system altogether. Database systems when in deadlock, cause the program to wait and thus no data can be retrieved from that point forward, deeming such system as useless. Distributed systems also heavily

suffer from deadlocks, since it quickly propagates through multiple nodes. Finally, embedded systems suffer the most from deadlocks. Since they are applied in real life situations such in healthcare or automotive devices, a halt on the program can cause system failures and endanger lives.

2.3.3 Heisenbugs

Heisenbugs are software bugs that seem to disappear or alter their behaviour when a debugger tries and study the origin of the problem. The term comes from an observation made by Werner Heisenberg, a physicist who found that the act of observing a quantum mechanical behaviour actually alters it. Heisenbugs can also be addressed as *"the probe effect"*, referencing the behaviour change when a test probe is attached to a device.

The following are common heisenbug triggers:

- **Output Statements:** Statements within a program steal execution time that would go to specific thread or process operations. This can cause changes in the execution order of threads or processes.
- **Running Debugger:** Running a debugger can change the program behaviour in subtle ways. In this case, debuggers can trigger changes in memory addresses of variables.
- **Compiler Optimizations:** Values that an optimized program would normally keep in registers are often pushed to main memory. These changes can affect floating-point comparisons for example, since the value in memory may have smaller range and accuracy than the value in the register.
- **Unprotected Variable Usage:** Utilizing non-initialized variables may change its address or initial value during debugging. Following an invalid pointer may also lead to different places while debugging.
- **Breakpoints:** Debuggers commonly allow the use of breakpoints or provide other user interfaces that cause additional source code to be executed which, similar to output statements, can change the timings, states and execution order of the programs.

2.4 Debugging Concurrency

The incorporation of concurrency into systems introduces new categories of bugs, such as race conditions 2.3.1 and deadlocks 2.3.2, which substantially increase the complexity of debugging. Programming languages have continued to evolve, with each iteration integrating safer and more advanced tools while still providing granular options for more experienced developers. As concurrency becomes increasingly vital in various domains of software development, the most widely used programming languages have adapted by providing extending built-in support for concurrency, with some even prioritizing concurrency as a central focus.

Concurrency dates back to the 1960s. Early efforts to address concurrency-related defects focused on applying sequential debugging methods to individual processes or threads which were very ineffective due to not being able to catch non-determinant behaviour. During the 1980s and 1990s, concurrency evolved with the introduction of specialized languages such as Ada, Modula II and Erlang which emphasized concurrency. Additionally, mainstream programming languages began incorporating concurrency tools to facilitate concurrent development.

A pioneer study conducted in 1989 [16] identified concurrency issues such as Heisenbugs and data races that remain prevalent today. The study highlighted both the advantages and limitations of sequential debuggers when applied to concurrent processes. While these debuggers excel in non-time-dependent scenarios, they are less effective when process ordering or heisenbugs are a concern. The concept of "Event Histories", proposed an initial system capable of storing certain program traces ending up evolving into race condition and process reordering debuggers today.

This section provides an in-depth analysis of concurrency built-in support and external concurrent-driven tools available in C/C++, Java, and C# (.NET). Additionally, this thesis also explores concurrency mechanisms and debugging tools for Go specifically, offering insights into its unique and important approach to concurrent programming.

2.4.1 General Concurrency Support Tools

Before delving into the support tools provided by specific programming languages, it is essential to comprehend the underlying concepts of the standard classes and types that these languages commonly implement. This foundational understanding is necessary to determine the environments for which these tools are best suited. Concurrency in most general-purpose programming languages is typically structured around several key components:

- Thread-like Structures
- Parallel systems
- Mutual Exclusion (Locks)
- Atomic Variables & Operations
- Thread-safe Collections & Data Structures
- Condition Variables
- Asynchronous Systems

As discussed in the Multi-Threading section 2.2.2, a thread can be considered an "independent executor," defined as a sequence of instructions and functions that can execute concurrently with other threads. Programming languages designed to support concurrent development must provide mechanisms to manage threads effectively. These mechanisms include support for thread creation, management, coordination, and identification.

As already mentioned in the Parallelism section 2.2.2 *"parallelism involves the use of multiple processes that run independently and simultaneously on separate CPU or GPU cores"*. For that reason, languages should have tools available with working up close with hardware like the C language, or provide automatic adjustments within their tools for parallel development.

Mutual exclusion (mutex) algorithms are fundamental to preventing multiple threads from accessing shared resources simultaneously, an thus maintaining synchronization among threads. Mutexes ensure that resources are allocated to a specific thread and are released when no longer needed, allowing other threads to access them. Mutexes can express themselves in many forms such as read-write lock systems, which differentiate between write locks and read locks. This distinction is significant because only write operations modify shared memory, making read locks more flexible in thread management. Mutexes also include variants such as semaphore locks, which restrict access to a finite number of resources, monitor locks that combine mutexes with condition variables, and spinlocks, which are optimized for

low-level scenarios where traditional locks are costly. Despite their utility, locking systems introduce disadvantages, including additional overhead and the risk of poorly designed locks causing deadlocks. These systems also often struggle to scale effectively with large numbers of threads or processes. Fortunately, modern programming languages address these challenges with robust tools such as `std::lock_guard` in C++, `synchronized` in Java, and `Goroutines` in Go. These abstractions simplify resource management by automatically handling locking and unlocking operations, reducing the likelihood of errors.

In the domain of synchronization, atomic variables and operations provide thread-safe mechanisms for handling primitive variables or memory addresses without requiring locks. These ensure that updates to variables occur atomically, meaning no other thread can observe intermediate states during the operation. Atomic variables are a powerful solution for preventing race conditions. They are typically faster than mutexes, as they avoid context switching and naturally prevent deadlocks due to their atomic nature, which stems from relying on low-level hardware instructions to maintain atomicity. However, their use is generally limited to primitive types. Common operations supported by atomic variables include:

- **Read/Write:** Load and store of variables
- **Increment/Decrement:** Increase or decrease value
- **Fetch-and-Add/Subtract:** Addition of subtraction of a value with return of the previous value of the variable
- **Compare-and-Swap:** Updates a variable only if it matches a specified value
- **Bitwise Operations:** Operations such as AND, OR, XOR, and others.

Compared to traditional locking systems, atomic variables offer several advantages. They introduce minimal overhead, they are faster and simpler to use than managing mutexes in most scenarios and they generally scale better, thanks to their lock-free algorithms, particularly in systems with many processors. Some languages, such as Java, even provide built-in thread-safe collections and data structures that utilise atomic operations under the hood, simplifying their usage.

Condition variables represent another synchronization primitive, allowing threads to halt their execution (wait) until signalled by another thread that a specific condition has been satisfied. Condition variables are generally used alongside mutexes and typically support three primary operations:

- **Wait:** Blocks the calling thread until some other thread signals the specified condition
- **Signal:** Wakes up one thread waiting on a specified condition
- **Broadcast:** Wakes up all threads waiting on a specified condition

Finally, asynchronous support in programming languages enables the development of asynchronous and reactive systems. Constructs such as promises and the async-await workflow allow processes to send requests to other processes without halting their execution. These constructs operate on the promise that the requested information will be provided at a later time (Figure 2.3). As emphasized in the Future model section 2.2.2, asynchronous programming enhances system responsiveness and is widely employed in I/O operations and to improve user experiences in applications.

When attention is turned to generalized tools and debuggers, only few propositions have some relevancy in the field. For example, Dominik [17] proposes a generalized record-and-replay solution for threads and locks, enabling the reproduction of actor-based program behaviors. The tool emphasizes uniformity, transparency, performance, and trace compactness by recording non-deterministic interactions and reliably replaying synchronization events. Further from that, debuggers and/or tools mainly focus on specific technologies.

2.4.2 C/C++ Language

The C programming language remains one of the oldest yet most relevant languages still widely employed in professional software development. Its enduring significance stems from its speed, efficiency, and relative ease of adoption. C grants developers substantial control, allowing them to work closely with hardware components. The language continues to play a pivotal role in various critical fields of computer science and engineering, including operating system kernels, automotive systems, and space exploration. C++, a language derived from C, builds upon its foundation by introducing additional features such as object-oriented programming and safer memory management tools. This thesis examines tools for both C and C++ at the same time due to C++'s full compatibility with C source code and its ability to leverage the same compilers as C.

In the context of concurrency support, modern C provides straightforward built-in functionality, including threads, atomic operations, mutual exclusion, condition variables, and thread-specific storage The language offers essential tools for concurrent programming through two dedicated libraries: `thread.h` and `stdatomic.h`.

The `threads.h` library defines functions and types for thread management, including creating (`thrd_create`), joining (`thrd_join`), and detaching (`thrd_detach`) threads. It also offers utilities like `thrd_sleep` for pausing thread execution and `thrd_yield` for yielding the current thread's execution slice.

The library also contains the `mtx_t` identifier for mutual exclusion algorithms in order of preventing race conditions, providing functions like `mtx_init`, `mtx_lock`, `mtx_trylock`, and `mtx_unlock`. These functions help ensure that only one thread accesses a critical section at a time. Additionally, the `call_once` function guarantees that a specific initialization occurs only once, even when called from multiple threads.

Furthermore, the `threads.h` library also provides support for condition variables (`cnd_t` identifier). These facilitate thread synchronization by allowing threads to wait for certain conditions to be met. Functions such as `cnd_wait`, `cnd_signal`, and `cnd_broadcast` are used to manage these synchronization points effectively.

Finally the `thread.h` library support the `tss_t` data type which identifies a thread-specific storage object. Even if, shared every thread will have its own instance of the variable, with different values.

The `stdatomic.h` library solely introduces atomic types and functions to perform lock-free, thread-safe operations on shared data. This includes functions for storing (`atomic_store`), loading (`atomic_load`), and modifying values atomically, ensuring proper memory synchronization between threads. C support atomic functions to every primitive type found within the language, serving as a very complete support tool for atomic development. Furthermore,

the library also makes available flag types and operations and memory synchronization ordering, where the former enables memory ordering constraints (`memory_order`) or the use of thread and signal fences (`atomic_thread_fence` and `atomic_signal_fence`).

It's important to note that these features are optional. Macros specifically tailored for the compiler can be used in order to activate or disable the aforementioned library tools, `__STDC_NO_THREADS__` and `__STDC_NO_ATOMICS__` for `thread.h` and `stdatomic.h`, respectively. All of this information is available on the C concurrency support library page [18] with further usage examples.

C++ enhances its concurrency support [15] by reusing the foundational tools of C while introducing additional features and functionalities to make concurrent development safer, flexible and more automated. Among these enhancements is the `<thread>` library, which provides the standard `std::thread` class for creating and managing threads, building upon C's implementation. Furthermore, with the advent of C++20, the `std::jthread` class was introduced, offering automatic joining upon destruction, simplifying thread management.

Few additions were made from the previous `<stdatomic.h>` library, with the only relevant addition of `std::atomic_ref`, which allows atomic operations on non-atomic objects, enhancing flexibility in concurrent programs.

Contrary to threads and atomic tools, C++ introduces a varied selection of new mutexes in its `<mutex>` library, including C's implementation (`std::mutex`) but adding more specific types, such as `std::recursive_mutex`, `std::timed_mutex`, and `std::shared_mutex`. Furthermore, C++ offers generic mutex management tools such as `std::lock_guard`, `std::unique_lock`, and `std::shared_lock` for more granular control over locking mechanisms.

Condition variables suffer a slight change for added flexibility and integration with the C++ type system, adapting C's implementation into `condition_variable` which only provides condition variables associated with `std::unique_locks`. In order to have any lock type associated with a condition variable, C++ also provides `condition_variable_any`.

C++ sees the addition of asynchronous concurrent tools such as promises and future. The `<future>` library provides `std::future`, `std::promise`, and `std::async`, facilitating asynchronous task execution and result retrieval, a feature not present in C's original support libraries.

Additional synchronization primitives such as semaphores, latches and barrier lock types were added in C++20 introducing `std::counting_semaphore` and `std::binary_semaphore` for semaphore locks, `std::latch` for latch locks and `std::barrier` for barrier locks, providing developers with more tools for complex synchronization scenarios.

Cooperative cancellation mechanisms such as `std::stop_token`, `std::stop_source`, and further `std::stop_callback` were also introduced in C++20, allowing for more controlled and responsive thread cancellation, enhancing the robustness of multi-threaded applications.

C++26, the most recent version of C++ also saw the introduction of some additional tools specific to resolve access-deletion races for read-copy-update mechanisms and hazard pointers.

Articles [19], [20], [21] and [22] propose debugger solutions in order to further help developers during concurrent development. When it comes to C/C++, most propositions fall under debuggers capable identifying data races and data de-synchronization and identifying and

saving execution orders of threads and processes enabling developers to reproduce buggy program states without the non-determinant behaviour of concurrent systems involved.

GAMBIT [19] proposes an interactive debugging environment integrated into the GNU Debugger (gdb) for use with relaxed memory models. It identifies problematic variables, visualizes the range of possible values during execution, and suggests fixes. While these fixes prioritise consistency over performance, they also detect "Lost Pushes" blocks of code where values or variables are lost due to race conditions suggests source code changes to make those sections atomic. Additionally, it isolates threads to provide deeper insights into their behaviour. During testing, the environment demonstrated efficiency and precision, making it a viable tool for concurrent debugging alongside the widely used gdb tool.

RaceDebugger [20] proposes a framework that integrates dynamic slicing with constraint solving to identify and analyse race conditions (similar to GAMBIT). It isolates buggy behaviors through "race manifestation" and filters relevant events to pinpoint root causes. Tested on real-world programs, the framework achieved a high success rate, diagnosing 376 out of 382 race conditions while significantly reducing the number of context switches. This framework simplifies debugging by effectively isolating complex race conditions. Unfortunately, this framework is only capable of race condition indentification an requires extra time for setting up process.

Period [21] enhances Controlled Concurrency Testing (CCT) for C/C++ programs. Using LLVM and Linux's deadline-based scheduling, it systematically explores thread interleavings to uncover concurrency bugs, including null-pointer dereferences and deadlocks. Tested on real-world benchmarks, the tool outperformed six baseline CCT techniques, identifying previously unknown bugs. However, its reliance on LLVM and the tool's limited compiler compatibility remains as its biggest drawback, since LLVM does not support GCC, the most widely known and used debugger for C/C++.

PMRace [22] focuses on detecting concurrency bugs in persistent memory models. Persistent Memory (PM) introduces unique challenges, such as inter-thread inconsistencies and synchronization errors, which it addresses through PM-aware coverage guided fuzzing and post-failure validation. During testing, it identified 14 unique bugs, highlighting its effectiveness in diagnosing PM-specific concurrency issues.

2.4.3 Java

Java provides one of the most comprehensive sets of features for concurrent programming among modern languages. The `java.util.concurrent` package equips developers with tools such as the executors framework, concurrent collections, synchronizers, locking mechanisms, atomic variables, and the fork/join framework. Due to the extensive range of features available, this thesis focuses on central components and specialized tools that help mitigate defects in concurrent programming.

Although Java includes a standard thread implementation via `java.lang.Thread`, executors and thread pools simplify thread management, making them more practical and recommended. The `ExecutorService` interface automates the life-cycle management of `Executor` classes. Executors provide a straightforward interface for launching tasks. Executor services then allow for orderly shutdowns without requiring manual configuration. Additionally, task scheduling is supported through `ScheduledExecutorService`, while pooling configurations can be managed using `ThreadPoolExecutor` with further scheduling capabilities available with `ScheduledThreadPoolExecutor`. These tools abstract thread creation

and management, facilitating the development of concurrent systems that are easier to understand and implement.

Recently, Java introduced virtual threads [23] that, although instances of `java.lang.Thread`, they are not bound to specific operating system threads, making them generally more lightweight. While virtual threads execute on operating system threads, they can be suspended during I/O operations. When a suspension occurs, the corresponding OS thread is available to switch virtual thread ownership and handle tasks for other virtual threads. Virtual threads can be created by using `Thread.ofVirtual()`. In a manner similar to regular threads, they can also be instantiated using executors and thread builders. Virtual threads are particularly well-suited for high-throughput concurrent scenarios with minimal call stacks, as they employ an implementation similar to virtual memory, mapping a large number of virtual threads onto a small number of OS threads. Consequently, the use of thread-local variables is discouraged.

Java also offers thread-safe variants of common collections and data structures to ensure safe concurrent access and prevent race conditions. For example, `ConcurrentHashMap` allows concurrent read and write operations without locking the entire map, while collections such as`CopyOnWriteArrayList` and `CopyOnWriteArraySet` are optimized for scenarios where read operations vastly outnumber writes, creating new copies of the underlying data structures for write operations. Additionally, Java provides `BlockingQueue` implementations such as `LinkedBlockingQueue`, `ArrayBlockingQueue`, and `PriorityBlockingQueue`, which are designed for producer-consumer scenarios, blocking when adding to a full queue or retrieving from an empty one.

To control thread execution flow and synchronization, Java provides a variety of synchronizers. The `Semaphore` class limits access to a resource by managing a fixed number of permits. `CountDownLatch` allows threads to wait until a set of operations is completed by other threads. The `CyclicBarrier` class ensures that multiple threads wait for each other to reach a common barrier point, while the `Exchanger` class facilitates data exchange between two threads by enabling them to swap objects at a synchronization point.

For more granular development over synchronization, `java.util.concurrent.locks` is the provided package by Java for resource locking and the `java.util.concurrent.atomic` package for atomic operations. Atomic variables in Java operate similarly to those in C++ 2.4.2, providing atomic operations for primitive types. Furthermore, Java offers the `AtomicReference` class, which enables atomic operations on object references. While powerful, atomic references are best suited for scenarios requiring simple atomic operations where monitor-based synchronization is not ideal. For more complex requirements, Java provides locking mechanisms such as the `ReadWriteLock` interface, which maintains separate locks for read-only and write operations, ensuring mutual exclusion for each. The `ReentrantReadWriteLock` implementation extends this interface by supporting lock reentry and lock acquisition interruption without imposing reader or writer preferences.

Java also provides support and built-in classes and interfaces specific for parallel development through the Fork/Join framework. The `ForkJoinTask` class is a thread-like entity that is much lighter than standard thread and thus, it enables the `ForkJoinPoll` class to host huge numbers os tasks and sub-tasks.

Finally, Java also provides asynchronous specific support with the `Future` interface, ideal for development of Asynchronous and Reactive systems. The `CompletableFuture` class implements the future interface and adds further refinements to the asynchronous formula,

enabling asynchronous execution, chaining of actions after task completion, multiple future combinations and further exception handling. This built-in support classes and interfaces can then be incorporated into more other components such as the executor services specific tools for reactive systems such as `Publisher`, `Subscriber`, `Subscription` and `Processor` components found within the Flow package inside the concurrent package of Java.

The language's concurrency support is vast and versatile, offering tools that range from basic thread management to advanced parallel processing frameworks. Despite these extensive capabilities, concurrent defects can still occur. The IntelliJ IDEA IDE [24] includes a built-in debugger that allows developers to control individual thread execution, enabling the identification of issues such as race conditions. This IDE serves as a strong foundation for debugging due to its ability to integrate with various Java applications and frameworks, support for external plugins and extensions, and widespread adoption in professional software development.

During research, both [25] and [26] demonstrated potential when testing and debugging concurrent environments. DCCond [25] proposes a high-level model for expressing synchronization and process ordering in Java. It alerts developers to inconsistencies in shared objects and provides tools to isolate sequences of instructions, identifying critical regions and potential ownership transactions between processes. By leveraging semantics, it identifies and monitors these regions using annotations like @Condition. Although DCCond under performs in low-concurrency scenarios compared to other tools, it excels in high-concurrency environments, making it particularly useful for complex applications.

ThreadRadar [26] proposes a tool for concurrency and performance issues visualisation, with possible IntelliJ IDEA IDE integration. By clustering threads based on runtime metrics and presenting data in a compact radial format, it offers developers more complex insights into thread behaviour than the standard given by the IDE. Although its user base remains limited, initial feedback indicates its utility as a visualization tool.

Unfortunately, most of the Java tools examined were either minimal in functionality or primarily aimed at enhancing existing systems already integrated into mainstream debuggers and IDEs.

2.4.4 C# & .NET

C# provides concurrency support similar to that of Java and C++ at a high level, offering standard thread classes and synchronization primitives such as mutexes, semaphores, and monitors under the `System.Threading` namespace. These tools allow developers to exercise fine-grained control over thread management.

Like Java, C# also includes thread-safe alternatives to common collections and data structures, available under the `System.Collections.Concurrent` namespace. Examples include `ConcurrentDictionary`, `ConcurrentQueue`, `ConcurrentStack`, and lastly `ConcurrentBag`. These collections help reduce the need for manual synchronization, and thus minimizing the risk of race conditions.

A significant focus of C# is its built-in support for parallel programming through the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). The TPL, which is part of the `System.Threading` and `System.Threading.Tasks` namespaces, provides constructs such as `Task`, `Parallel.For`, and `Parallel.ForEach` to enable efficient parallel execution. It also dynamically adjusts the level of concurrency to maximise processor utilization. PLINQ is

described as *"a parallel implementation of the Language-Integrated Query (LINQ) pattern"* [27]. LINQ allows developers to integrate queries directly into C#, and PLINQ extends this functionality to enable parallel operations, improving performance on multi-core systems by facilitating data parallelism.

For asynchronous programming, C# provides the `async` and `await` keywords, enabling non-blocking operations that enhance the responsiveness and efficiency of applications. These features are particularly valuable for developing asynchronous and reactive systems.

These tools collectively provide a comprehensive framework for managing concurrency in C# applications, enabling developers to write efficient, scalable, and responsive programs, especially when the language is often used in conjunction with the Visual Studio IDE [28]. Visual Studio offers a powerful many powerful tools from copilots to an integrated debugger tool capable of saving information about specific thread traces and locations. It offers a visual tool capable of showing what each thread was executing when the program reaches a selected breakpoint put by the developer. It also is capable of setting watches on certain variables in other to visualise its different values during process interleavings, flag threads that are deemed important to follow, freeze and thaw thread execution in order to control the order in which threads perform work and follow singular threads with conditional breakpoints. Due to its popularity and significant support for C# and C++, the Visual Studio Debugger is one of the best debuggers on the market, if not the best.

Beyond Microsoft's ecosystem, FlakePro [29] offers additional support for addressing flaky test failures caused by concurrency issues in .NET systems. By combining static and dynamic analysis, FlakePro is capable of identifying and reproducing specific thread interleavings that lead to failures. When applied to 22 Microsoft projects, FlakePro successfully reproduced 26 of 31 flaky tests, providing a practical and effective solution for debugging concurrency-related issues in .NET applications.

2.4.5 Go Language

Go is a general purpose programming language known for its robust support for concurrency, providing developers with built-in primitives that simplify the development of concurrent applications. Go's concurrency model differs from standard general purpose languages, investing in Communicating Sequential Processes (CSP), promoting a paradigm where concurrency is achieved through communication rather than shared memory.

Go's makes use of lightweight thread-like structures, which the language calls `Goroutines`. The Go runtime is responsible for managing these goroutines, allowing the creation of thousands of concurrent tasks without significant overhead.

To facilitate communication and synchronization between goroutines,`Channels` are introduced allowing goroutines to create channels between them, where they safely exchange data between them without explicit locking. This channels can be buffered or unbuffered, providing flexibility in designing concurrent software. If channels are not appropriate,

It also provides mutex tools in case explicit locking between goroutines is still needed. These tools ensure that only one goroutine accesses a critial section of code at a time, at the cost of possible deadlock states.

Still tackling synchronization, Go provides the `WaitGroup` type that is used to wait for a collection of goroutines to finish executing, blocking the program until all tasks are completed.

Finally, Go provides a new `select` statement, very similar to switch statements, but instead used to enable goroutines to wait on multiple communication operations, proceeding with the first that becomes ready, useful for handling multiple channels and implementing timeouts.

Tools [30],[31] and [32] are all external concurrent development tools, focused on making concurrent development in Go easier and safer by adding new implementations of existing built-in tools and implementing further debug capabilities on the Go workflow.

GFuzz [30] focuses on data races and execution ordering bugs, addressing challenges unique to Go's channel-based communication model. The tool is able to mutate the order of concurrent messages. It them generates new message orders based on feedback from the program execution, prioritizing orders that are more likely to uncover bugs, being able to capture both blocking and non-blocking concurrency bugs. During testing, it identified 184 previously unknown bugs, demonstrating its superiority over static tools such as GCatch in detecting complex concurrency issues. Due to its dynamic analysis, the tools introduces overhead and runtime checks but remains feasible for in-house testing.

Conc's [31] mission is to make concurrency for go even easier and safer, adapting Go's concurrency support tools and implementation a safer and easier version. The goals of the tools is to *"make it harder to leak goroutines"*, *"handle panics gracefully"* and *"make concurrent code easier to read"*. Conc is used my thousands of people everyday and is very easy to install and quickly apply the tool in Go software. In its source code page [31], they also have some quick examples to demonstrate conc's power.

Where conc focuses on giving the developer easier and safer implementations of the Go's concurrent support, cff focuses on managing and minimizing damage coming for concurrent defects. Cff, a concurrency toolkit for go, gives the developer *"a pool of goroutines to run all operations, preventing issues arising from unbounded goroutine growth"*, also preventing panics by instead crashing defected code in a predictable manner, eliminating non-determinant behaviors. The toolkit is known to be applied in professional development, since it was developed and currently used by Uber Technologies Inc.

Chapter 3

Testing Concurrency

This section is dedicated to the practical application of several tools and concepts discussed in Chapter 2. It explores concurrent design patterns, language-specific features, and debuggers that assist in writing robust concurrent code and identifying potential concurrency-related defects.

The examples presented range from simple, easy-to-follow scenarios to more complex applications. The projects presented were built from scratch to demonstrate the practical use of concurrency tools across different programming languages, offering an interactive perspective on how these can be leveraged effectively.

Although the study aims to thoroughly explore relevant tools, design approaches, and debugging techniques, it does not try to use all available tools due to time constraints. Instead, it focuses on providing clear justifications for specific tool usage over alternatives and outlining optional development paths that could be taken depending on the context.

The section begins with a brief overview of each developed application to provide context on the concurrency design choices made during implementation. These overviews will also discuss the language-specific tools employed, highlighting why particular decisions were favored over others. The goal is to illustrate the versatility of concurrency in software development and to emphasise the importance of aligning concurrency strategies with the specific requirements of an application to minimise potential defects.

Finally, this section explores the use of debuggers, illustrating how these tools can help identify issues, reduce the inherent unpredictability of concurrent systems, and test specific execution paths while monitoring their runtime behaviour. Since the objective is to assess the unique capabilities of each debugger, the study adopts a `ptrace`-based approach. This method deliberately avoids incorporating techniques such as library pre-loading or integrating external tools via compiler or interpreter instrumentation. The chosen approach enables binary analysis without requiring recompilation, reinterpretation, or direct access to the source code. Although alternative methods may offer lower overhead or even superior performance in some cases, the flexibility of the `ptrace` approach aligns more closely with the study's goal of evaluating debuggers in a non-intrusive setting.

3.1 Test Methodology

This chapter will focus primarily on evaluating debuggers while also introducing some language-specific tools used in the development of concurrent systems. Since most modern languages offer similar concurrency mechanisms, differing mainly in syntax and internal implementation, this provides a consistent foundation for testing concurrency-related issues such as

race conditions and deadlocks, making comparisons across tools equally meaningful. All three debuggers will be assessed based on the following criteria:

- **Capability / Efficiency**: Can the debugger detect various types of concurrency defects? Which issues does it fail to identify?
- **Usability**: Is the information provided by the debugger clearly organized and easily digestible? Does increased efficiency compromise usability, or vice versa?
- **Versatility**: Can the debugger be effectively applied across multiple technologies or languages with consistent results?
- **Overhead**: What is the performance cost of enabling debugging features? How significantly does it impact execution and the project's size?

As outlined in previous chapters, the identification of race conditions and deadlocks will be the central focus of this evaluation. Each debugger will undergo a basic demonstration test, followed by the concurrent programs developed during the thesis. This approach aims to offer both a broad comparison of debugger capabilities and a deeper insight into their real-world usage, highlighting how each tool performs under practical development scenarios.

3.2 Ray Tracer

This project revolves around the development of a basic ray tracer, an application that simulates the behaviour of light to generate photorealistic images. Ray tracing is capable of reproducing a wide range of optical effects, including reflection, refraction, soft shadows, scattering, depth of field, motion blur, caustics, ambient occlusion, and chromatic dispersion.

Due to its complexity, ray tracing was, for a very long time, limited to static scenes. Even with early concurrency optimizations, rendering a single high-fidelity image could take hours or even days. As such, its primary applications were found in visual effects, computer-generated imagery (CGI), and rendering tasks where long render times were acceptable. However, advancements in computing hardware, especially in graphics processing units (GPUs), have made ray tracing viable for real-time applications such as video games.

This project closely follows the "Ray Tracing in One Weekend" book series [33]. The series provides the core foundations for constructing a ray tracing engine. While the algorithms introduced are not necessarily representative of cutting-edge implementations, they still are a solid gateway to the key principles of ray tracing and provide a platform that incetivizes experimentation, optimization, and feature expansion.

Given that the focus of this thesis is studying concurrency, from its development into the project to the ways to detect defects, the scope in the demonstration given below are narrowed to study the addition of parallel systems to help with rendering times. Ray tracers serve as an excellent case study in parallel programming, as even minor concurrency issues, such as data races, manifest in visibly incorrect or incomplete renderings.

The ray tracing process can be generalized into three major steps:

1. **Scene Setup**: Load geometric primitives, materials, and lighting into the virtual scene.
2. **Ray Launching and Intersection Testing**: Cast a ray from the camera through each pixel in the image plane and determine intersections between the ray and scene objects. If multiple intersections occur, retain only the one closest to the camera.

Finally, calculate the final colour based on surface material, lighting, reflections, and refractions.

3. **Image Construction**: Store the computed colour for each pixel and export the result as an image.

Currently, objects are loaded sequentially into a list of hittable objects. This step could be parallelized, such as using multiple threads to add objects, though proper synchronization would be necessary to avoid data races.

The `render()` function is the core of the application. It iterates through each pixel in the image, computes its colour, and writes it to an output file:

```
void render(const hittable& world)
{
  initialize();
  image_file << "P3\n" << image_width << ' ' << image_height << "\n255\n
    ";

  for (int j = 0; j < image_height; j++)
  {
    for (int i = 0; i < image_width; i++)
    {
      color pixel_color(0, 0, 0);
      for (int sample = 0; sample < samples_per_pixel; sample++)
      {
        ray r = get_ray(i, j);
        pixel_color += check_intersection(r, max_depth, world);
      }
      write_color(image_file, pixel_sample_scale * pixel_color);
    }
  }

  image_file.flush();
  image_file.close();
}
```

Listing 3.1: Initial Render Function (C++)

This function can be divided into four major stages of computation: **Camera initialization**, **Ray Generation**, **Intersection Testing**, **Building an image**. The camera class behaves like a renderer, being the hub for all render components and properties.

The render function then computes a unique ray for each pixel. Rays can be expressed by the equation ‘P(t) = O + td‘, where ‘O‘ represents its origin, ‘d‘ the direction, and ‘t‘ a scalar representative of the distance between a ray and an intersected object. Since it matters only to know what lies in front of the ray, only positive ‘t‘ values are taken into account.

Following ray generations, the application starts computing the interaction of those rays with scene objects. This step is the most computationally intensive, as it involves the use of techniques such as recursive reflection, refraction, and light scattering calculations.

Finally, the application uses PPM files to generate images. In the books, this file format was chosen due to its simplicity and straightforwardness. PPM files can be built with the following rules:

```
PPM File Format

P3                  # file type
400 225             # width height
255                 # maximum color value (]0, 65536])
166 183 193         # red green blue
...                 # height x width rows of RGB values for each pixel
```

Listing 3.2: PPM File Structure

Since ppm files require the application to write RGB colour values for each pixel in the view port to their bodies, it is necessary to implement a function tasked with this. Fortunately, with C++ built-in input/output libraries, this tasked is relatively easy. The following snippet demonstrates how colour values are written into the ppm file:

```
void write_color(std::ofstream& file, const color& pixel_color)
{
  auto r = pixel_color.x();
  auto g = pixel_color.y();
  auto b = pixel_color.z();

  r = linear_to_gamma(r);
  g = linear_to_gamma(g);
  b = linear_to_gamma(b);

  // Translate the [0,1] component values to color range [0,255].
  static const interval intensity(0.000, 0.999);
  int rbyte = int(256 * intensity.clamp(r));
  int gbyte = int(256 * intensity.clamp(g));
  int bbyte = int(256 * intensity.clamp(b));

  // Write out the pixel color components.
  file << rbyte << ' ' << gbyte << ' ' << bbyte << '\n';
}
```

Listing 3.3: Write Colour Function (C++)

Anti-aliasing is also implemented via super-sampling (multiple rays per pixel) represented by the innermost loop. Consequently, the ‘render()‘ becomes even more computation heavy, dominating execution times. How can concurrency be employed into the current application state in order to speed up the rendering process?

While GPU-based computation is the optimal solution for real-time rendering, this thesis focuses on CPU-based concurrency. CPU-based ray tracing is slower but serves as an excellent framework for evaluating theoretical concurrency models and can be translated relatively easy into shaders (GPU focused programs).

The main opportunity for introducing concurrency is in parallelizing the ‘render‘ function. A straightforward and effective approach is to use thread objects to split up the computational workload. By leveraging a thread pool, the overhead of managing threads can be reduced by reusing a fixed set of worker threads. In the following sections, two solutions will be explored: a custom thread pool implementation and one based on C++ utilities. These examples aim to highlight both the complexities of building a safe concurrent environment and how built-in tools often simplify the process, resulting in cleaner and more robust implementations. While explaining these solutions, the Visual Studio 2022 debugger and the GDB will both be used in order to better visualise and catch possible defects during development.

The final objective is to significantly diminishing rendering times with concurrency. For reference, the current sequential version of the ray tracer took 2 hours and 20 minutes to render the following scene:

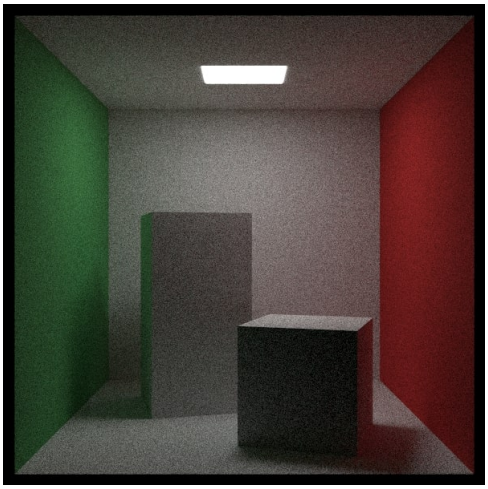

Figure 3.1: Cornell Box Scene (600x600 resolution)

3.3 Chat Application

This project involves the development of a real-time chat application that enables multiple users to communicate with each other simultaneously. To achieve this functionality, the application uses socket programming and a multi-threaded architecture, ensuring a smooth and responsive experience for all connected clients.

Socket programming refers to the development of software that operates on multiple computers connected via a network. Sockets are responsible for establishing and maintaining connections between devices [34]. In this application, sockets are used to connect individual clients to the central server, with dedicated input and output channels that facilitate data exchange.

The server plays a central role in managing client connections and handling communication through their respective sockets. Additionally, it is solely responsible for accessing and retrieving data from the application's database, which it shares with the clients when needed. Clients themselves do not have direct access to the database.

To support message handling, both the server and the clients utilise message flags at the beginning of each message. These flags indicate the type of message that is being transmitted. For example, a message beginning with the 'AUTH' flag informs the server that the client is trying to authenticate and gain access to the chat environment.

The application makes use of an internal class named `ConnectionHandler`. Each of these objects is responsible for storing information about a unique and exclusive client socket. This means that it is impossible by design that two connections are handled by two or more `ConnectionHandler` objects. Furthermore, every time a connection to a new client is established with the server, a unique thread in the thread-pool is responsible for dealing with the newly created connection handler. This design diminishes possible concurrency problems without the implicit use of synchronization techniques.

```java
public class ChatServer implements Runnable {
  ...
  @Override
  public void run() {
    try {
      server = new ServerSocket(PORT_NUMBER);
      threadPool = Executors.newCachedThreadPool();

      while (isServerActive) {
        Socket client = server.accept();
        ConnectionHandler handler = new ConnectionHandler(client);
        connections.add(handler);
        threadPool.execute(handler);
      }
    } catch (IOException e) {
      Shutdown();
    }
  }
  ...
  class ConnectionHandler implements Runnable {
    ...
  }
}
```

Listing 3.4: ChatServer Concurrent Design (Java)

Each `ConnectionHandler` instance encapsulates a unique 'Socket' object corresponding to a single client. It also holds the input and output streams used for communication and stores the client's nickname, allowing other users to identify the sender of each message.

```java
public class ChatServer implements Runnable {
  ...
  class ConnectionHandler implements Runnable {
    private final Socket client;
    private BufferedReader input;
    private PrintWriter output;
    private String nickname;
    ...
  }
}
```

Listing 3.5: Connection Handler unique Socket system (Java)

The `ConnectionHandler` is responsible for managing incoming client messages and instructing the server to broadcast them to all other connected clients.

For broadcasting purposes, the server still maintains a list of the current active connection handlers. Since the `Broadcast()` is at the server level, race conditions and general undefined behaviour of concurrency can still occur at said method if the appropriate actions are not taken.

Other potential concurrency problems can arise with non thread-safe database execution, and dealing with shared resources between handlers. These possible problems will be analyzed further in this document, when studying IntelliJ IDEA's debugger capabilities.

That said, the current implementation does not verify whether a user attempting to log in is already active in the chat. This oversight can lead to multiple clients being logged in

under the same account simultaneously. While this represents a functional flaw, it is more accurately categorized as an implementation issue rather than a multi-threaded problem.

3.4 Debuggers

Debuggers often offer a variety of visualization tools to aid in development. Although they are known primarily for finding bugs, they have a ton of uses beyond the obvious scope:

- **Bug detection and fixing:** Standard debugging tasks to identify and resolve issues in code
- **Code analysis:** Debuggers enable step-by-step execution of a program, helping developers understand its flow and logic. They are also useful for becoming familiar with new or unfamiliar code bases.
- **Behaviour modification for testing:** Debuggers allow for the reproduction of complex scenarios and test setups, allowing developers to switch the application's state to simulate different situations.
- **Enhancing logging:** Additional logging points can be added dynamically during debugging sessions.
- **UI Rendering Inspection (Painting Debugging):** Some debuggers are able to analyse UI rendering behaviour, allowing inspection of object rendering at a granular level.
- **Memory usage analysis:** Insight into how memory is allocated and consumed during runtime.

While this chapter has covered several core uses of a debugger, it's important to note that debuggers can vary widely in their capabilities. Some provide rich graphical interfaces with advanced visualization tools tailored for specialized tasks, while others offer simpler, more streamlined environments that focus on clarity and straightforward commands.

That said, the foundational concepts of debugging tend to be consistent across tools. Once a developer understands how to use one debugger, that knowledge can often be transferred to others with relative ease. This learning process is comparable to picking up different imperative programming language, where the underlying logic remains the same, but the syntax used to express it differs.

Due to time constraints, this thesis will focus on analyzing the capabilities of a select group of debuggers commonly used in professional development environments. These include:

- IntelliJ IDEA Debugger
- Visual Studio 2022 Debugger
- GNU Debugger (GDB)

The thesis will explore each debugger's strengths in detecting, understanding, and visualizing concurrent behaviour, with particular emphasis on identifying and resolving concurrency-related defects.

3.4.1 IntelliJ IDEA Debugger

The debugger offers both basic tools and advanced features, allowing developers to tailor their debugging environment to complex application architectures. Since this thesis focuses on detecting and observing concurrent behaviour, it will not explore every available debugging feature. Instead, it emphasys is given to the core tools relevant to concurrency debugging.

Intellij's debugger has two main core features for concurrency [35]:

- **Thread-Specific Breakpoints:** These allow developers to monitor and analyse the behaviour of individual threads independently. This is particularly useful when diagnosing obscure race conditions or deadlocks, as it enables the recreation of specific execution paths without relying on randomized or repeated testing.
- **Async stack traces:** Allow for developers to analyse stack traces that may come from multiple threads, since asynchronous code is often executed between multiple threads. It estabilishes a connections between frames of different thread stack traces. These frames are automatically shown in the debugger UI, having async related traces separated from other traces. Intellij IDEA also allows the developer to insert async annotations inside the code in order to watch method or variable async stack traces.

Given the importance of breakpoints in concurrent debugging, it is worthwhile to examine their capabilities in greater detail before presenting a practical concurrency debugging example.

Breakpoints

Breakpoints are essential tools in any debugger. They allow developers to pause program execution at specified points to examine its state and behaviour. Once a breakpoint is added, it remains active unless explicitly deleted or muted. Muted breakpoints can later be reactivated.

IntelliJ's breakpoints are highly customizable. They range from simple line breaks to conditional logic triggers and logging instructions. The four primary types of breakpoints are [36]:

- **Line Breakpoints:** Pause execution at a specific line of code.
- **Method Breakpoints:** Trigger when entering or exiting a method or any of its implementations, useful for analyzing method entry/exit behaviour.
- **Field Watchpoints:** Suspend execution when a particular field is read or modified, aiding in the identification of unexpected data changes.
- **Exception Breakpoints:** Trigger when a 'Throwable' is thrown, enabling analysis of the context surrounding exceptions.

Breakpoints can also be configured to deactivate automatically after being hit or activate only after a dependent breakpoint is triggered. Furthermore, there is logging options:

- **Breakpoint hit messages**
- **Stack trace logging:** Logs the current call stack to reveal the program's path to the breakpoint.
- **Evaluate and log:** Logs messages conditionally, based on a user-defined expression.

These configurations help isolate specific execution paths and ensure that the debugger only pauses when necessary. Additional filtering options allow further control:

- **Catch Class filters:** Restrict breakpoint activation based on the class handling the exception.
- **Instance filters:** Limit breakpoint behaviour to a specific object instance.
- **Caller filters:** Enable breakpoints only when a method is invoked from certain parts of the code base.

Debugging Concurrent Environments

Concurrency-related bugs are inherently more difficult to detect and reproduce than single-threaded bugs, due to their non-deterministic nature. A system may run smoothly for a long period before failing unexpectedly due to a rare threading issue.

IntelliJ's debugger allows developers to observe each thread individually. The same breakpoint can be used to inspect the behaviour of distinct threads, enabling precise replication of complex concurrency issues, such as race conditions or deadlocks.

Consider the following test case in Java: two threads attempt to add the same value to a list using the `addIfAbsent()` function. The test asserts that the final size of the list should be 1. The `addIfAbsent()` method checks if a value is absent from the list and adds it if so. The `list` is a synchronized version of a standard Java `List<>`, meaning all method calls are thread-safe. However, even using synchronized collections, race conditions can still occur frequently.

```
1  public class ConcurrencyTest {
2    @Test
3    public void main() throws InterruptedException {
4      assert(1 == work().size());
5    }
6
7    static List<Integer> work() throws InterruptedException {
8      final List<Integer> list = Collections.synchronizedList(new
       ArrayList<>());
9      Thread t = new Thread(() -> addIfAbsent(list, 17));
10     t.start();
11     addIfAbsent(list, 17);
12     t.join();
13     System.out.println(list);
14     return list;
15   }
16
17   private static void addIfAbsent(List<Integer> list, int x) {
18     if (!list.contains(x)) {
19       list.add(x);
20     }
21   }
22 }
```

Listing 3.6: Intellij IDEA's Concurrency Test Example

Although this test typically passes, repeated execution eventually results in failure, where the list contains duplicate values, such as [17, 17]. Even though |`list.contains()` and

`list.add()` are synchronized individually, the `addIfAbsent()` method itself is not synchronized. Consequently, both threads can pass the `contains()` check before either calls `add()`, rendering the if condition ineffective.

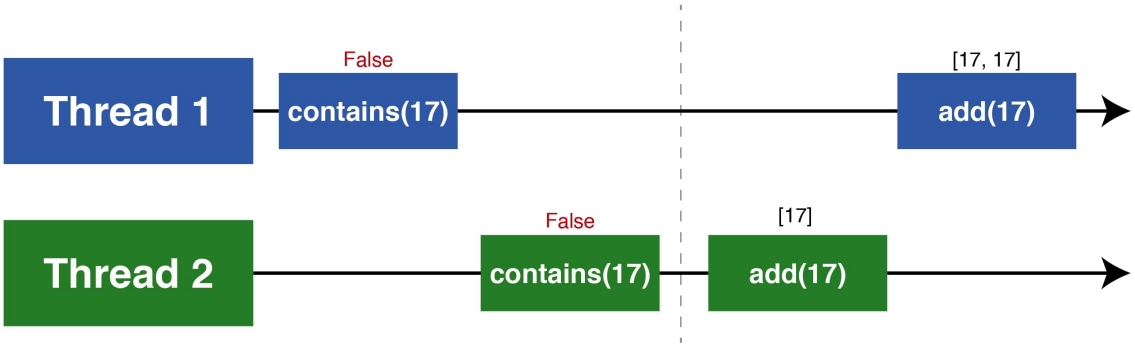

Figure 3.2: IntelliJ IDEA concurrency test possible race condition

To reproduce this issue using the debugger:

1. Set a breakpoint at the conditional check: `if (!list.contains(x))`.
2. Change the breakpoint scope from **All** to **Thread**.
3. Run the program in debug mode.

When the execution hits the breakpoint, IntelliJ suspends the relevant threads. From the debugger's thread panel, threads can be selected individually to step through.

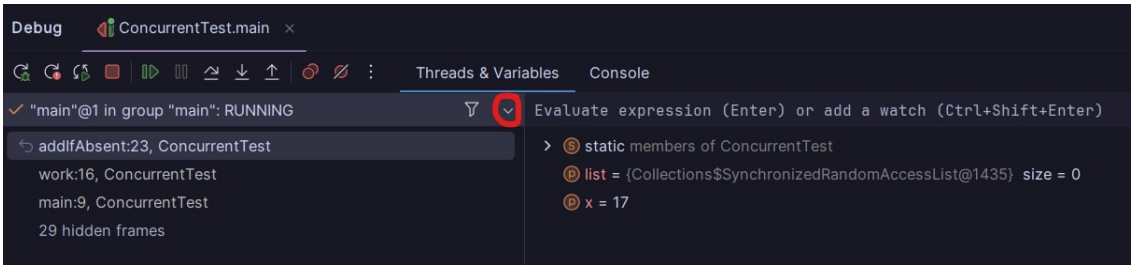

Figure 3.3: IntelliJ IDEA's Debugger interface

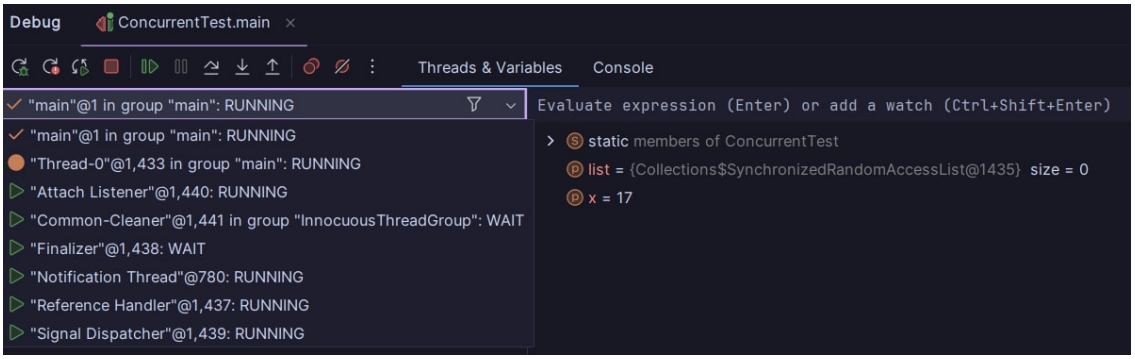

Figure 3.4: IntelliJ IDEA's Debugger thread selection

In the test, two threads are present: the main thread and "Thread-0".

- Step over the if block in the main thread to reach `list.add(x)`.

- Switch to "Thread-0" and do the same.
- As a result, both threads will add the same value to the list.
- Upon continuing execution, the list ends up as [17, 17], and the test fails.

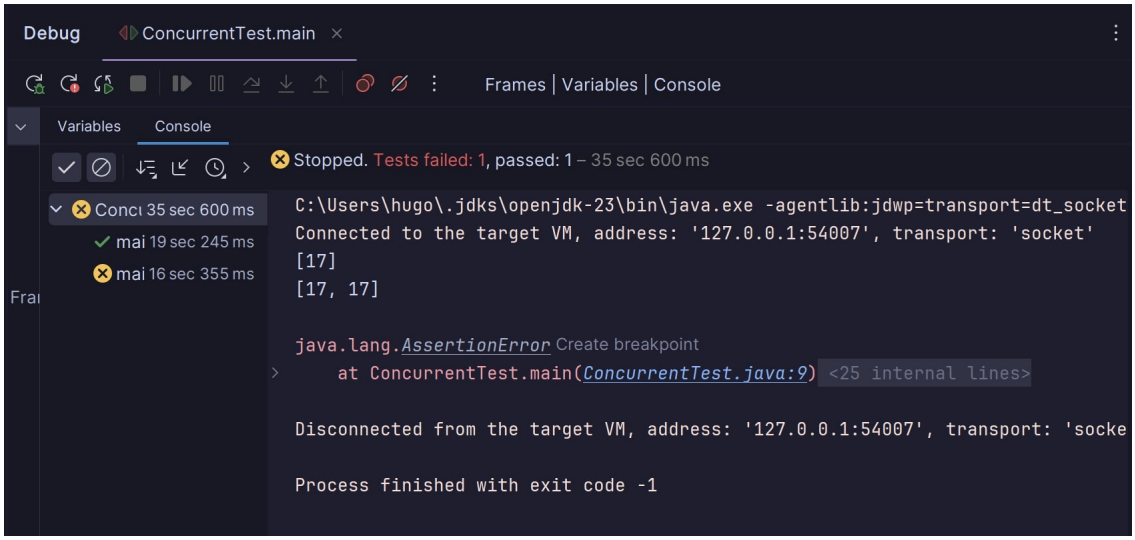

Figure 3.5: IntelliJ IDEA's concurrency example defected result

The bug occurs because `addIfAbsent()` does not enforce synchronization on the method level. A simple fix is to synchronise the entire method, ensuring mutual exclusion during the check-and-add operation:

```
private static void addIfAbsent(List<Integer> list, int x) {
  synchronized(list) {
    if (!list.contains(x)) {
      list.add(x);
    }
  }
}
```

Listing 3.7: Intellij IDEA's Concurrency Test fix

With this fix, only one thread can execute `addIfAbsent()` at a time. When rerunning the program with debugging:

- The second thread will be blocked until the first finishes.
- In the debugger interface, selecting **Get Thread Dump** reveals that "Thread-0" is now in a waiting state.

This feature is also helpful in identifying deadlocks, as it lists all active threads and shows when one is blocked due to contention.

When testing and analyzing concurrency-related issues, it is essential to first evaluate the implemented concurrent design and identify areas where complex or problematic interactions may occur. Once these areas are recognized, language-specific tools and debuggers can be used to examine the robustness of those critical regions by executing various trace paths and observing the program's behaviour under different conditions.

Figure 3.6: IntelliJ IDEA's thread state visualizer

Due to the inherently non-deterministic nature of multi-threaded execution, it is impossible to guarantee that all issues have been resolved after fixing known bugs. However, by leveraging debugging tools and concurrency testing strategies, developers can significantly reduce the likelihood of encountering defective program runs.

When applying these techniques to the chat application 3.3, a new set of potential concurrency issues emerges. Given that the client's primary role is limited to sending and receiving messages from the server, it presents limited interest from a concurrency analysis standpoint, except in UI-related or logic-specific cases. Therefore, the server component becomes the focal point for detailed examination.

On the server side, the most vulnerable part of the system are the message broadcasting mechanism and database management. This is because of two main aspects in the code:

1. Server uses a standard `ArrayList<>` to store connection handler objects, which of course, are **not thread-safe**. This can lead to problems where the list is updated at the same time a thread is accessing its elements.
2. Functions `VerifyUser()` and `CreateUser()` both make calls to the database, and since the database itself is not thread-safe, there could be race conditions where a connection handler alters the information on the database, and other connection handlers receive unsynced data.

To demonstrate this issue, an experiment was conducted to provoke an error triggered by a concurrent modification of the connections list during iteration. A breakpoint was placed on the line `connections.add(handler)` to capture this behaviour:

- Client 1 initiates a broadcast, entering the 'for' loop that iterates over the server's connection list.
- Client 2 simultaneously establishes a new connection, prompting the server to add its corresponding handler to the same list
- Server adds a new connection to the list while Client 1 is still iterating through the elements of the list.

This simultaneous access can lead to concurrent modification exceptions or unexpected behaviour, underlining the importance of synchronizing access to shared resources in concurrent environments.

There are several strategies that can be used to ensure that the `connections` list is thread-safe. One straightforward approach is to convert `connections` into a synchronized list

using Java's `Collections.synchronizedList(...)` method. This guarantees that each method call on the list is thread-safe, reducing the risk of concurrency-related issues.

```
public class ChatServer implements Runnable {
  private final List<ConnectionHandler> connections;
  ...
  public ChatServer() {
    connections = Collections.synchronizedList(new ArrayList<>());
    ...
  }
}
```

Listing 3.8: ChatServer connections thread-safe list with synchronized list

An alternative to this approach is the use of the `CopyOnWriteArrayList` class. According to the official Java documentation, it's "a thread-safe variant of `ArrayList` in which all mutative operations (add, set, etc.) are implemented by making a fresh copy of the underlying array." [37]

This class is especially useful in scenarios where read operations are significantly more frequent than write operations. However, it is less efficient when the application involves a high number of writes. In the context of a chat server managing many simultaneous socket connections, frequent client connections and disconnections could degrade performance due to the overhead introduced by copying the list on each write.

```
public class ChatServer implements Runnable {
  private final CopyOnWriteArrayList<ConnectionHandler> connections;
  ...
  public ChatServer() {
    connections = new CopyOnWriteArrayList<>();
    ...
  }
}
```

Listing 3.9: ChatServer connections thread-safe list with CopyOnWriteArray

A third approach involves creating a snapshot of the connection list when performing a broadcast. By taking a synchronized snapshot, the application separates the iteration from the original list, allowing simultaneous addition of new connections without causing concurrent modification exceptions. The only requirement is that the creation of the snapshot itself must be synchronized to ensure consistency:

```
public void Broadcast(String message) {
  List<ConnectionHandler> snapshot;
  synchronized (connections) {
    snapshot = new ArrayList<>(connections);
  }

  for (ConnectionHandler handler : snapshot) {
    handler.SendMessage(message);
  }
}
```

Listing 3.10: ChatServer list snapshot approach

While this implementation introduces a bit more complexity, it enhances readability by clearly indicating where synchronization occurs. It also provides a practical balance between safety and performance by isolating the potentially unsafe iteration from concurrent mutations.

By using synchronized collections or snapshot-based techniques, developers can ensure that all operations on `connections` are thread-safe. This eliminates the risk of modifying the list while broadcasting messages, allowing developers to focus on other potential concurrency issues within the system.

Having identified the potential for data races during database interactions, primarily due to the non-thread-safe nature of direct database access, mitigating these issues requires a similar approach to that used above, ensuring that access to shared resources is appropriately synchronized across concurrent handlers. In the case of database operations, this can be achieved through two main strategies: explicit synchronization or the use of connection pooling with concurrent capabilities.

Even when using separate connections, the underlying mechanisms of the database, especially during write operations, can still lead to race conditions. This is particularly problematic in scenarios with simultaneous inserts, such as multiple users registering concurrently. A straightforward solution is to wrap each database access in a synchronized block, thereby enforcing exclusive access during critical operations.

```
1  public class MessageRepo {
2    private final Object lock = new Object();
3
4    public void AddNewMessage(int userId, String content) {
5        String addNewMessageSql = "INSERT INTO messages (userid, content)
     VALUES (?, ?)";
6
7      synchronized(lock) {
8        try (Connection connection = DriverManager.getConnection(dbPath);
9             PreparedStatement statement =
10              connection.prepareStatement(addNewMessageSql)) {
11
12           statement.setInt(1, userId);
13           statement.setString(2, content);
14           statement.executeUpdate();
15         } catch (SQLException ex) {
16             System.out.println("[DB Failure] " + ex.getMessage());
17         }
18     }
19   }
20
21   // apply the same logic to all other methods
22 }
```

Listing 3.11: Database synchronization by manual locking

While this solution is relatively easy to implement and effectively prevents race conditions, it does so by serializing access to the repository. This can introduce performance bottlenecks, particularly under heavy client load, where it can consequently introduce deadlocks into the implementation.

A more scalable alternative involves using a connection pool designed for concurrent environments. This project incorporates HikariCP [38], a high-performance JDBC connection pool library. HikariCP manages a pool of pre-initialized database connections, allowing the

application to reuse them rather than instantiating new connections for every operation. This improves both performance and resource efficiency.

Importantly, HikariCP is designed with concurrency in mind. According to its documentation: "HikariCP contains a custom lock-free collection called a ConcurrentBag. The idea was borrowed from the C# .NET ConcurrentBag class, but the internal implementation quite different. The ConcurrentBag provides a lock-free design, thread local caching, queue-stealing and direct hand-off optimizations, resulting in a high degree of concurrency, extremely low latency, and minimized occurrences of false-sharing." [38]

This makes HikariCP a compelling choice for concurrent database management in multi-threaded server applications.

The following example demonstrates how HikariCP can be integrated:

```java
// DatabaseManager.java
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import javax.sql.DataSource;

public class DatabaseManager {
    private static final HikariDataSource dataSource;

    static {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:sqlite:database.db");
        config.setMaximumPoolSize(10);
        dataSource = new HikariDataSource(config);
    }

    public static DataSource getDataSource() {
        return dataSource;
    }
}
```

Listing 3.12: HikariCP basic configuration

```
import javax.sql.DataSource;

// UserRepo.java
public class UserRepo {
  private final DataSource dataSource;

  public UserRepo() {
    dataSource = DatabaseManager.getDataSource();
    try (Connection connection = dataSource.getConnection()) {
        ...
    } catch (...) {}
  }

  public boolean AddUser(User user) {
      String addUserSql = "INSERT INTO users (name, password) VALUES (?, ?)";

      try (Connection connection = dataSource.getConnection(); ... )) {
          ...
      } catch (...) {}
  }
}
```

Listing 3.13: HikariCP usage to enable safe database accesses

While this thesis does not seek to explore third-party libraries or external tools that enhance concurrent development, it is important to acknowledge their existence and relevance. Tools like HikariCP offer powerful, well-tested concurrency mechanisms that can greatly simplify the development of thread-safe applications, improving both performance and maintainability.

IntelliJ IDEA's debugger places significant emphasis on breakpoint management, especially when working with concurrent environments. Its design prioritizes simplicity and ease of use, offering straightforward tools for monitoring thread behaviour. Although incapable of detecting data races on its own, it focuses on presenting useful and direct data that can guide developers into uncovering these problems. The same is applied to deadlocks, where the only diagnostic tool available is the thread state viewer, which, although often sufficient, feels underwhelming compared to other debuggers at study.

Another limitation of IntelliJ IDEA's debugger lies in its language support. It is tightly coupled with the IDE and is only available for a select group of languages: Java, Kotlin, Scala, and Groovy. JetBrains addresses support for other languages by offering separate IDEs tailored to different technologies. Although these IDEs share a similar user experience, the debugger's features and capabilities may differ between them, adjusting to each language-specific needs. This fragmented approach ultimately restricts the debugger's versatility and availability.

Performance is another consideration to take upon. Among the tools examined, IntelliJ IDEA's debugger generally introduced the most file size overhead, due to how it stores debug information. When building a debug version of the program, the compiler generally includes a number of debug information tables to each '.class' file used in the project. This means that the amount of overhead between debug and release mode is directly linked to the project's scope which can vary greatly. Unfortunately, due to the nature of the Server application, no execution time tests were able to be conducted.

In conclusion, IntelliJ IDEA's debugger is a sufficient tool, especially for those new to debugging multithreaded programs. Its user-friendly interface and minimal learning curve make it accessible, but this comes at the expense of efficiency and advanced features. Developers working on complex or high-performance concurrent applications may find its simplicity limiting in the long run.

3.4.2 Visual Studio 2022 Debugger

Many of the features present on the debuggers already tackled are present in Visual Studio 2022. Furthermore, it possess a wider range of visualization tools with intent of keeping a tight grip of the different states of a program during execution. The official Visual Studio 2022 documentation website [39] heavily details each of these windows and menus, but since the goal is to provide a general overview and how this tool can be widely used to fix issues or watch code behaviour, a more practical approach to explanations is expected instead of a more in-depth of all the capabilities of each tool. By consequence, all information shared on specifications of these tools are based on their official pages.

Like the other debuggers, breakpoints are an essential tool in Visual Studio 2022 and can be suited with conditional expressions and filters to study specific execution behaviors. Tracepoints are simply breakpoints that print messages to the output window. Since they are essentially breakpoints, all customization that can be applied to breakpoints is valid for tracepoints. Breakpoints can be divided into several categories:

- **Line Breakpoints**: Evaluate the state of a thread or process in a specific line of code
- **Function Breakpoints**: Evaluate the state of a thread or process when accessing a specific function. It is useful for when the function name is known but not its location or if there are functions with the same name and a developer wishes to break on them all
- **Data Breakpoints (.NET Core 3.x and 5+)**: Breakpoints that break when a specific object's property changes. They are dependent on CPU architecture with different data limits for different processors. This can make debugging code in common paths such as game loop or a utility API much easier because a breakpoint in those functions can be configured to enable only if the function is invoked from a specific part of the application.
- **Dependent Breakpoint**: Break execution only if another breakpoint is first hit.
- **Temporary breakpoint**: Debugger only pauses on the breakpoint once and then it removes it immediately after

Specifically for C# projects, the debugger also enables the observation of specific objects by letting developer create object IDs for specific instances of reference types, using them as breakpoint conditions.

The debugger has several other windows to visualize concurrent behaviour upon registered hit. These expand to thread states, thread execution paths, task and asynchronous management in case of a task-based architecture being used and even variable or function watches within concurrent behaviour. Most of these windows also possess different grouping filters available, helping developers focus on specific groups of entities if wished so. There are three main windows specialized in concurrent debugging:

- Threads / Tasks Window

- GPU Threads Window
- Parallel Stacks Window
- Parallel Watch Window

The threads window tracks all used threads during execution. When a breakpoint is hit, the window provides direct and simple information about a collection of threads. Information on a thread's origin (where and who was it spawned from), and state are then shown. Still on this window, developers can stop and resume the execution of different threads to test specific execution paths, similar to the Thread mode seen in IntelliJ IDEA's debugger. Of course, this window is also capable of organizing and filtering specific threads depending on the conditions imposed by the user. Additionally in C++ projects, the debugger offers a GPU thread management window. As the name entails, it is very similar to what is found in the Thread Window but nurtured to threads running on the GPU.

The tasks window has the same exact function as the threads window, "except it shows information about `System.Threading.Tasks.Task` or `task_handle`" objects [39]. Like threads, tasks can be run concurrently but have the benefit of multiple tasks being run by the same thread. The most relevant information of this window is the status of each task, identifying if a task is schedulled, active, awaiting or even deadlocked.

Introduced in version 17.5 in 2022, the Parallel Stacks Window offers a tool designed to visualize and navigate the call stacks of threads or tasks. It offers three different views, threads, tasks and method views, allowing developers to focus debugging on thread, `Task` objects or specific method call stacks. Similarly to all other windows, the Parallel Stacks Window is also equipped with filtering options. It is mostly used for analyzing and understanding asynchronous behaviors between threads or tasks, but it also provides an excellent base for deadlock identification.

Finally, the parallel watch window has the function of keeping track of values that one expression holds on multiple threads. Each row in the window represents a thread, but the same thread can appear in two or more rows. Each row represents a function call whose function signature matches the function on the current stack frame.

Debugging Concurrent Environments

An important consideration when using debuggers is that the information provided can vary significantly depending on the programming language in use. This also applies to the way language-specific built-in tools interact with the debugger. To illustrate this, let's revisit the deadlock scenario briefly introduced in Section 2 and analyse how Visual Studio's debugger handles thread management and displays relevant information. Specifically, we will examine how the debugger behaves differently with C# (managed code) versus C++ (native code). Let's begin with the C# version of the code. Adapting the original snippet is straightforward:

```
static void ExecuteThread1() {
  lock(lock1) {
    Console.WriteLine("Thread 1: Holding lock 1...");
    Thread.Sleep(100);
    Console.WriteLine("Thread 1: Waiting for lock2...");
    lock (lock2) {
      Console.WriteLine("Thread 1: Acquired lock2!");
    }
  }
}

static void ExecuteThread2() {
  lock(lock2) {
    Console.WriteLine("Thread 2: Holding lock2...");
    Thread.Sleep(100);
    Console.WriteLine("Thread 2: Waiting for lock1...");
    lock (lock1) {
      Console.WriteLine("Thread 2: Acquired lock1!");
    }
  }
}
```

Listing 3.14: Achieving a Deadlock State in C sharp

When this program is executed within Visual Studio's debugger, a deadlock occurs as expected. By utilizing the Parallel Stacks Window and the Threads Window, developers can quickly detect and analyse the deadlock. Upon opening the Parallel Stacks window, Visual Studio automatically flags the deadlock situation:

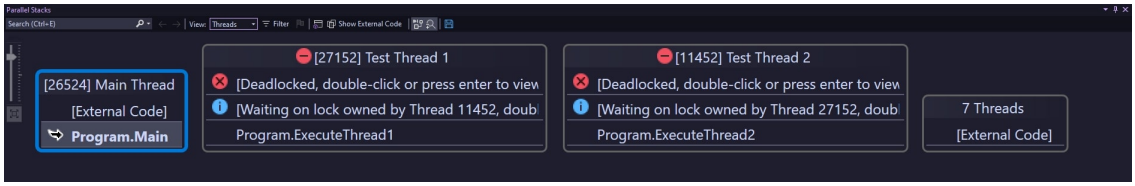

Figure 3.7: Parallel Stacks Deadlock Warning

The debugger not only identifies the deadlock but also highlights the exact line where the issue could be located. As shown in the image, double-clicking the warning takes the developer directly to the source of the deadlock, even switching the viewed thread automatically. Although the Call Stack window also indicates a deadlock, the Parallel Stacks window serves as a significant enhancement, offering clearer navigation and context within concurrent environments. This makes it especially useful in more complex applications with multiple threads.

Unfortunately, when trying to recreate the same warnings on C++'s version, the information given by the debugger is way less conclusive, there isn't a clear flag identifying the deadlock, and further investigation into the external code needs to be taken. This happens because, even though it's the same IDE and thus the same debugger, visual studio divides code into two distinct categories: native code and managed code.

However, attempting the same test in C++ yields much less informative results. The debugger fails to clearly flag the deadlock, offering no direct warning and requiring deeper investigation into the program's execution. This difference in behaviour arises from Visual Studio's internal distinction between managed code (like C#) and native code (like C++).

By Visual Studio's official documentation, managed code can be described as *"code whose execution is managed by a runtime. In this case, the runtime in question is called the Common Language Runtime or CLR"* [40]. By contrast, in native code such as in C++ and C applications, the programmer is in charge of everything, since the program is directly turned into binary for the operating system to run. Consequently, the debugger has access to less metadata when compared with manage code applications and thus, is less capable of providing debug information.

Lets now use these tools to aid in the addition of concurrency into the ray tracer application. The idea revolves in developing a two solutions to divide rendering work through multiple threads.

Unfortunately, C++ does not have a thread pool implementation. Instead of manually managing `std::thread` objects, a thread pool enables more efficient resource management by reusing worker threads. A basic thread pool requires a list of worker threads, a queue for task management and synchronization primitives such as mutexes and condition variables.

```
1 std::vector<std::thread> workers; // jthread also work here
2 std::queue<std::function<void>> tasks;
3 std::mutex queue_mtx;
4 std::condition_variable condition;
5 bool stop;
```

Listing 3.15: Thread Pool Class Members (C++)

Synchronization is imperative in order to avoid multiple worker threads acessing and modifying the task queue at the same time. A thread pool can be described with the following diagram:

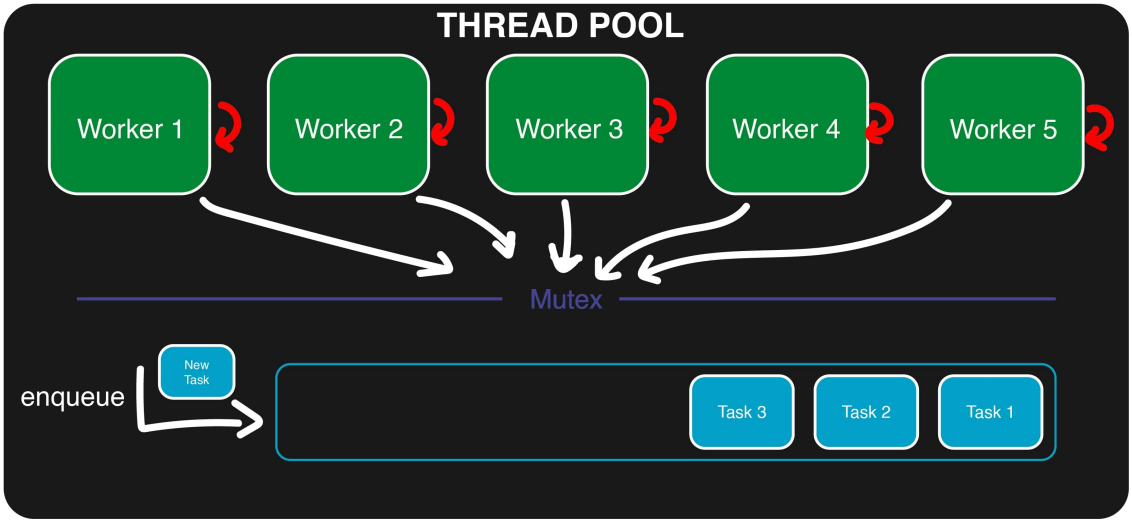

Figure 3.8: Thread Pool Diagram

Normally, thread pools are initialized with a specific number of worker threads. These worker threads need to be programmed in order to wait on avaible tasks. Condition variables let threads wait on new available tasks. Upon receiving a notification of a new available task, a worker thread takes responsibility of said task, pops it from the task queue, unlocks the queue and proceeds to executed its given task. With the help of lambda functions, the logic

can also be programmed directly inside the function, instead of calling an external function, providing better code readability.

```
thread_pool(size_t n_threads) : stop(false)
{
  workers.emplace_back([this]
  {
    while (true)
    {
      std::unique_lock<std::mutex> lock(queue_mtx);
      condition.wait(lock, [this] { return stop || !tasks.empty(); });

      if (stop && tasks.empty())
        return;

      auto task = std::move(tasks.front());
      tasks.pop();
      lock.unlock();
      task();
    }
  });
}
```

Listing 3.16: Task Handling Logic (C++)

Tasks need to be added to the tasks queue safely. This is done by locking access to the queue while the addition occurs. Tasks are enqueued as follows:

```
template <class T>
void enqueue(T&& task)
{
  std::unique_lock<std::mutex> lock(queue_mtx);
  tasks.emplace(std::forward<T>(task));
  lock.unlock();
  condition.notify_one();
}
```

Listing 3.17: Task Enqueue Logic (C++)

Applying the thread pool to the render function:

```
thread_pool pool(4);
for (int j = 0; j < image_height; j++)
{
  for (int i = 0; i < image_width; i++)
  {
    pool.enqueue([this, &world, i, j]
    {
      // color detection code
    });
  }
}
```

Listing 3.18: Applying The Thread Pool To The Render Function (C++)

The result is an incomplete image with corrupted data. This is clear by inspecting the data in written to the image file:

```
P3
400 225
255
192 192206 227
 191207 229
 166206  183 227
193
85 113 71
155 173 178
```

Listing 3.19: Scene Render - Corrupted Data

It is evident that a writing-related bug occurs when storing RGB values into the file. A number of corrupted rows appear, and the application fails to write all the expected rows.

Let's start by placing a function breakpoint on the `write_color` function. Upon opening the Threads Window, it is possible filter thread by name in order to organize all the thread pool's worker threads into a single group. There could be hints of potential race conditions or even deadlocks. Something that comes to mind, is the possibility of multiple worker threads writing into the file at the same time, causing a data race. By suspending (freeze) and resuming (thaw) threads, it's possible to observe an execution state where all worker threads are active inside `write_color`, specifically during the actual file write operation. The fact that this state can be reached strongly suggests a race condition is present during file writing.

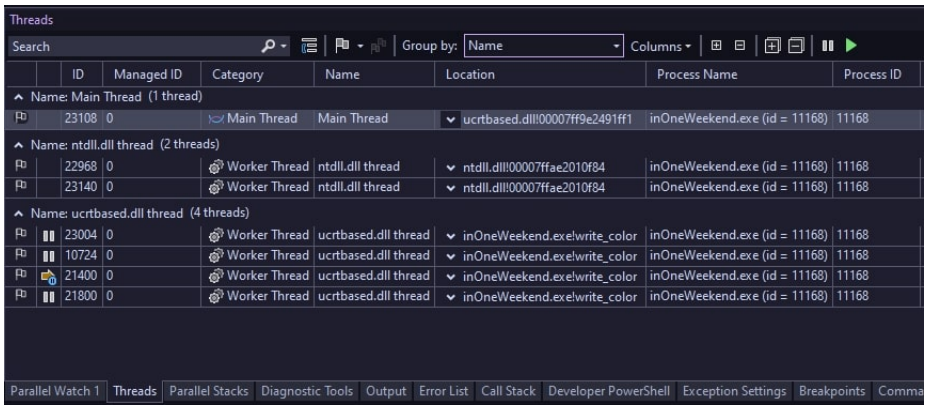

Figure 3.9: Threads Window

While the Threads window tracks the state of each worker thread, additional tools such as the Parallel Stacks and Parallel Watch windows can aid in identifying deadlocks or monitoring key variables.

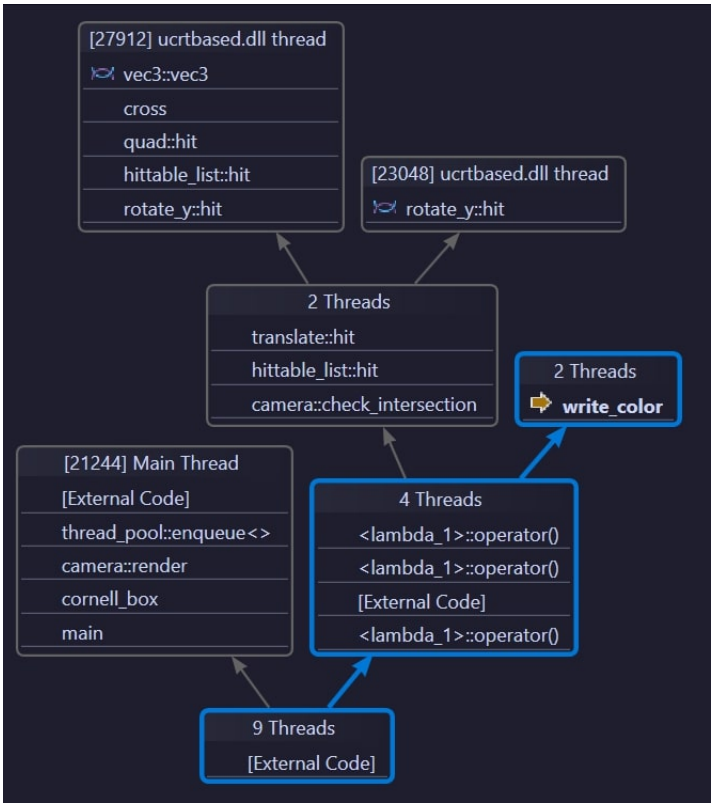

Figure 3.10: Parallel Stacks Window

The Parallel Stacks window does not report any unusual execution paths while debugging, which rules out deadlocks between worker threads. With that eliminated, the root cause can be confidently identified as a race condition. To further confirm, we can inspect what's being written into the file using the Parallel Watch window.

Unfortunately, the application uses `std::ofstream` type to deal with file management and directly watching a `std::ofstream` object doesn't reveal the file's contents. Instead, it is possible to buffer the output in a `std::string` and examine its contents before writing it to the file:

```
1 std::ostringstream oss;
2 oss << rbyte << ' ' << gbyte << ' ' << bbyte << '\n';
3 std::string data = oss.str();
4 file << data;
```

Listing 3.20: Buffering Colour Data For Debugging

A typical valid entry for data might look like this:

When modifying the execution trace so that all worker threads write to ‘data‘ simultaneously, the debugger detects unexpected behaviour. Depending on the internal implementation of C++, two types of issues were observed: either the string is flagged red and its contents appear corrupted, or the debugger fails to read the string altogether:

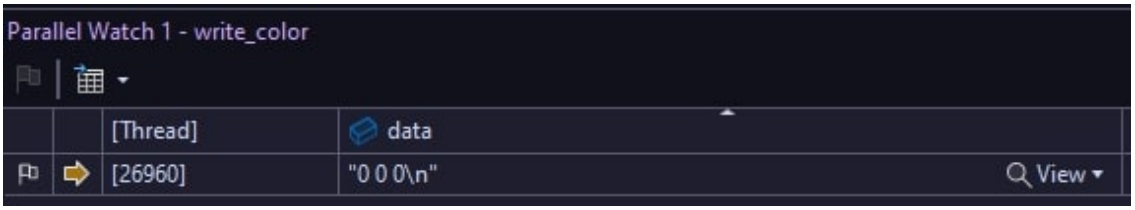

Figure 3.11: Parallel Watch Expected Behaviour

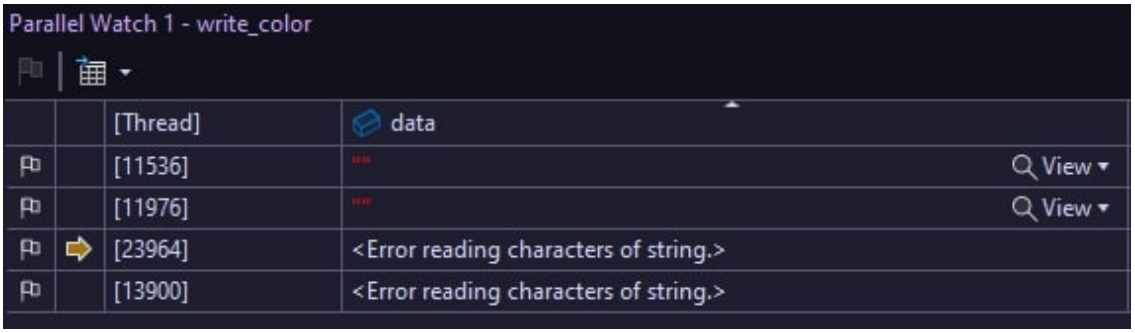

Figure 3.12: Parallel Watch Faulty Behaviour

This confirms that the problem is a race condition, caused by multiple threads writing to the file concurrently. To resolve this, some form of data synchronization is necessary. The most straightforward and obvious solution is to protect each write operation with a mutex:

```
1 std::lock_guard<std::mutex> lock(render_mtx);
2 write_color(image_file, pixel_sample_scale * pixel_color);
```

Listing 3.21: Write Colour Function Synchronization

This eliminates the corrupted rows, but the output image is still incomplete—containing fewer than the expected 360,000 rows (based on a 600x600 resolution). During testing, the number of rows written varied between 450 and 550. This is because the render function may terminate while worker threads are still processing tasks. Consequently, the thread pool is shut down prematurely, halting further pixel rendering.

To guarantee all pixels are rendered before termination, the thread pool must be modified to track and manage task completion. New synchronization primitives are introduced. When queuing a task, additional logic is added to notify the system once all tasks are completed. Thanks to lambda captures, this can be embedded directly in the task:

```
template <class T>
void enqueue(T&& task)
{
  std::unique_lock<std::mutex> u_lock(mtx);
  tasks.emplace([this, task = std::forward<T>(task)]
  {
    task();

    // add aditional commands to the the task
    if (--tasks_in_progress == 0)
    {
      std::unique_lock<std::mutex> lock(mtx);
      tasks_done_cv.notify_all();
    }
  });
  ++tasks_in_progress;
  u_lock.unlock();
  cv.notify_one();
}

std::condition_variable tasks_done_cv;
std::atomic<size_t> tasks_in_progress{0};
```

Listing 3.22: Thread Pool Enqueue Function With Task Progress

Finally, let's add a `wait()` function to the thread pool that blocks until all tasks have completed. This function uses `tasks_done_cv` to ensure no tasks are left in the queue and none are still in progress. It does not join the worker threads, allowing them to remain available for future tasks. With this in place, adding a simple `pool.wait()` call at the end of the rendering loop ensures that all pixel tasks are completed before the rest of render instructions are proceeded:

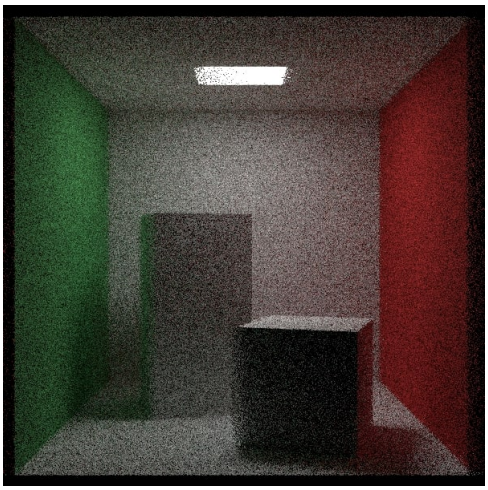

Figure 3.13: Cornell Box Faulty Render (600X600 resolution)

While an image is finally produced, closer inspection reveals misplaced pixels, such as stray red pixels along the left edge and blurry object boundaries. This is not random noise due to insufficient samples but rather the result of pixels being written out of order. This occurs because some threads finish tasks sooner than others, and the writing is done directly within the tasks themselves.

A reliable fix for this situation is to load all pixel colors into a buffer, essentially a vector of colors, reserving space on the buffer depending on image size and mapping the coordinates of the pixel to a index on the buffer. Once every task is finished, all information added to the frame buffer can be loaded safely into the image file in sequential order:

```
void render(const hittable& world)
{
    ...
  for (int j = 0; j < image_height; j++)
  {
    for (int i = 0; i < image_width; i++)
    {
      pool.enqueue([this, &world, i, j]
      {
        ...
        frame_buffer[j * image_width + i]
          = pixel_sample_scale * pixel_color;
      });
    }
  }
    ..
  for (int i = 0; i < image_height * image_width; i++)
      write_color(image_file, frame_buffer[i]);
    ...
}

std::vector<color> frame_buffer;
```

Listing 3.23: Rendering Image By Buffering

Since each worker processes unique '(i, j)' coordinates, no data races occur while writing to the buffer. Therefore, additional synchronization mechanisms like mutexes become unnecessary. Furthermore, because the final file write occurs after all rendering is complete, pixel order is preserved regardless of task execution timing. The final, correctly rendered image 3.1 was obtained in approximately 26 minutes, a **83% rendering time reduction**.

An alternative to the implementation seen above is to use C++17's `<execution>` header, which allows algorithms such as `std::for_each` to be executed in parallel using policies like `std::execution::par`. Under the `std::execution::par` specification, the loop is tasked with spawning an *"unspecified number of data partitioning chunks"* [41] and scheduling them on the Windows thread pool with no resource restrictions. This is turn means that every loop represents an independent and concurrent task running with no sense of synchronization.

The idea is exactly the same as the thread pool implementation above but these algorithms remove a lot of bloatware and aim to keep parallel programming simpler and safer. For starters, the thread pool class is now not necessary. Instead, due to `std::for_each` parameter specifications, there is a need to load all indexes between 0 and the image's width and height onto a respective horizontal and vertical vectors to serve as iterators.

The following example demonstrates parallelizing both the vertical and horizontal iterations of the render loop:

```
1
2  void render(const hittable& world)
3  {
4      std::for_each(std::execution::par, vertical_iterator.begin(),
       vertical_iterator.end(),
5      [this, &world](int j)
6      {
7          std::for_each(std::execution::par, horizontal_iterator.begin(),
       horizontal_iterator.end(),
8          [this, &world, j](int i)
9          {
10             // intersection code
11         });
12     });
13 }
14
15 std::vector<int> vertical_iterator;
16 std::vector<int> horizontal_iterator;
```

Listing 3.24: Render Function - C++ Parallel Utility Version

Upon running the program again, shockingly the render is obtained in 42 min, a **70% time reduction** from the initial render and significantly slower when compared to the custom thread pool's 26 minutes.

Visual Studio's debugger is arguably the most complex in terms of usage. While it offers a comprehensive feature set, it can be overwhelming compared to GDB and IntelliJ IDEA's debugger. The sheer volume of information presented to the developer at once can sometimes obscure what's most relevant. For instance, simulating a specific execution path often requires manually freezing and thawing multiple threads, an approach that contrasts with the simpler, more intuitive thread control mechanisms available in GDB and IntelliJ IDEA. Unlike those tools, Visual Studio does not provide a straightforward option for thread-specific debugging or an easy toggle for debugging all threads simultaneously.

The debugger's greatest strength lies in its ability to combine multiple specialized windows for concurrent programming analysis. Developers can use tools like the Threads window alongside Parallel Stacks to simulate execution and obtain real-time insight into potential deadlocks. In this way, Visual Studio's debugger proves highly efficient, allowing developers to tailor their approach by using an array of concurrent debugging aids.

However, this power comes at the cost of usability. The interface can feel bloated, prioritizing depth of insight over user-friendliness. Another downside is the debugger's inconsistent performance across different programming languages and execution contexts. Its efficiency, as well as the volume of information provided, can vary significantly between managed and native code applications, diminishing its practical utility.

Moreover, Visual Studio's debugger introduces noticeable overhead between Debug and Release mode. This can hinder productivity depending on the nature of the application being developed. Upon rendering a simpler scene than the one already demonstrated, the Debug build finished in 6883.2ms with a final executable file size of roughly 0.66MB, a significant 194% time increase and 46% size increase over Release's rendering time of 2613.83ms and file size of 0.45 MB. For more complex scenes, both rendering times and file sizes can grow even wider. The extent of the overhead also depends on language-specific factors and the volume of debugging information extracted. C# applications, for example, tend to

have more overhead than C or C++ programs, due to Visual Studio's ability to retain more detailed debug metadata.

Like IntelliJ IDEA, Visual Studio's debugger is limited to the languages supported by the IDE itself. It supports C++, C, C#, Visual Basic, F#, JavaScript/TypeScript, and Python, making it more versatile than IntelliJ IDEA's debugger and comparable to GDB in terms of language support.

In conclusion, Visual Studio's debugger stands out as a powerful yet complex tool. Its robust concurrency debugging features and language versatility make it a strong asset, though these come at the cost of usability. For advanced developers working on multithreaded or cross-language applications, its capabilities can be invaluable, provided they are willing to navigate its steep learning curve.

3.4.3 GNU Debugger (GDB)

The GNU Debugger (GDB) [42] is a foundational tool for software debugging. First introduced in 1986, it remains a powerful terminal-based debugging utility that can also integrate with IDEs such as Visual Studio to provide a graphical interface. Due to its longevity, GDB has become a standard debugger for UNIX-like systems and supports a wide variety of languages including C, C++, Go, Ada, Rust, Assembly, and more.

Compared to more modern, integrated tools, GDB typically requires more manual setup to function properly. While IDE-based debuggers often enable full debugging capabilities with a simple drop-down selection for a "Debug" or "Release" build, GDB relies on the developer to configure the build environment appropriately. Certain features are only available if specific libraries are linked accordingly. For instance, debugging multi-threaded C++ applications effectively requires linking threading libraries. Although the process is not generally overly complex, it is more involved than using GUI-based tools.

At its core, GDB relies on a variety of breakpoint mechanisms. These structures are highly flexible and adaptable to various debugging contexts, including multi-threaded environments. Breakpoints in GDB support conditional expressions, limited executions (temporary breakpoints), and can be thread-specific. GDB also provides convenience variables to assist developers during debugging sessions, such as `$bpnum`, `$_hit_bpnum` and `$_hit_locno`, used to identify the last breakpoint set, the last breakpoint hit and the location index of the last breakpoint hit, respectively. Types of breakpoints include:

- **Line breakpoints**: Created with `break location [if condition]`.
- **Function breakpoints**: Implicitly set when specifying a function name.
- **Watchpoints**: Monitor variable access (read: `rwatch`, write: `watch`, both: `awatch`) and can be scoped to threads or memory addresses.
- **Catchpoints**: Capture specific exception behaviour, such as `catch throw` or `catch rethrow` in C++. Syntax may vary from language to language.

GDB includes robust tools for analyzing multi-threaded behaviour. When a program spawns threads during a debug session, GDB uses two types of identifiers each thread, a GDB specific thread ID and an LWP (Lightweight Process) ID used internally by variables and functions. These IDs can be checked by using `info threads`, listing all threads with their status, IDs, and top stack frames. Developers can filter this view using flags like `-running` or `-stopped` and switch between threads using `thread thread-id`. Breakpoints can be bound to specific

threads using `break location thread thread-id`. These breakpoints are automatically removed when their corresponding thread terminates. By default, GDB operates in the All-Stop Mode, meaning, all threads halt when any breakpoint is hit. However, this behaviour can be modified by using Non-Stop Mode, making threads run independently unless individually paused instead or changing the `scheduler-locking` and `scheduler-multiple` settings:

- Run only current thread: `scheduler-locking on`.
- Step only current thread: `scheduler-locking step`.
- Resume multiple selected threads: `set schedule-multiple on`

Additionally, GDB supports non-intrusive debugging with `set observer on`, which prevents memory writes or breakpoint insertions and automatically enables Non-Stop Mode

Debugging Concurrent Environments

Similar to what was done in the Visual Studio section, lets start by testing the debugger in simple deadlock scenario first, and then apply it to the ray tracer, to test for race conditions. First, the program needs to be compilled with `-g` (genaral debug information) or `-ggdb` (GDB specific debug information) flags. Since the program also makes use of the `<thread>` library, the `-lpthread` flag also needs to be included. Then, when the program is built, gdb can finally be used on the final executable file:

```
g++ deadlock_eg.cpp -ggdb -lpthread -o deadlock_eg
gdb deadlock_eg
```

Listing 3.25: Standard GDB Debug Mode Compilation

With GDB open, it is possible to run the program as natural with ‘run‘ or ‘r‘, wait for it to reach the deadlock state and stop execution at that moment. From there it is possible to obtain the current backtrace of the program by using ‘where‘ or ‘bt‘. Other information like the ‘mutex‘ resources and thread information used by the program is essential here. Those can be obtained by typing ‘print resourceX‘ and ‘info threads‘, respectively. All these commands can be combined into a single one with the use of ‘define‘, which lets developers create custom commands. This is a very useful tool to reduce debugging session times if used efficiently. All the information essential to identify the location of the deadlock can thus, be grouped by this feature:

```
define deadlock_debug
> bt
> info threads
> print resource1
> print resource2
> end
```

Listing 3.26: GDB Complex Command Example

The following data was obtained:

Although verbose, the output contains several key insights that can be extracted through careful analysis:

- At the point of interruption, the main thread is waiting for all other threads to join within the `main()` function, as seen in frames #5 and #6.

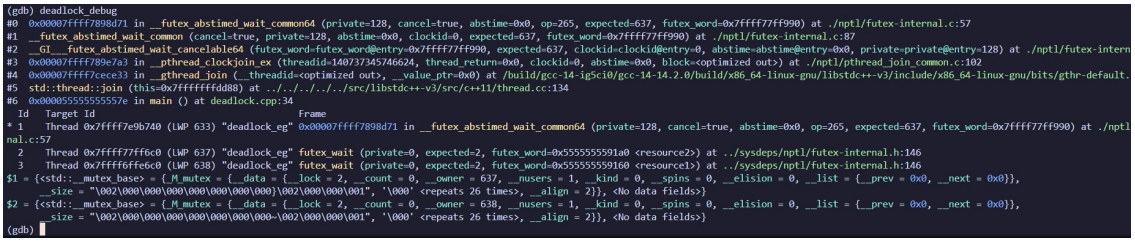

Figure 3.14: GDB - Data obtained from custom command

- The `info threads` command lists all threads along with their identifiers and execution points at the time the program was paused. Both Thread 2 and Thread 3 are shown to be blocked, each waiting on a resource that is currently locked (`futex_waiting`). Specifically, Thread 2 is waiting for *resource2*, while Thread 3 is waiting for *resource1*.
- By printing each mutex resource, it is possible to identify the thread that currently owns it. In this case, *resource1* is owned by Thread 2 and *resource2* by Thread 3. This ownership is determined by inspecting the `__owner` field of each mutex and matching it with the corresponding thread's LWP ID.

It is clear that both Thread 2 and Thread 3 are holding resources that the other is waiting for. In case of a more complex scenario or if needed, each thread presented by `info threads` is also acessable, each with their own specific backtraces available. For example, by prompting `thread 2` to change thread and using `where` or `bt`, it is possible to analyse in depth the stack frames of Thread 2:

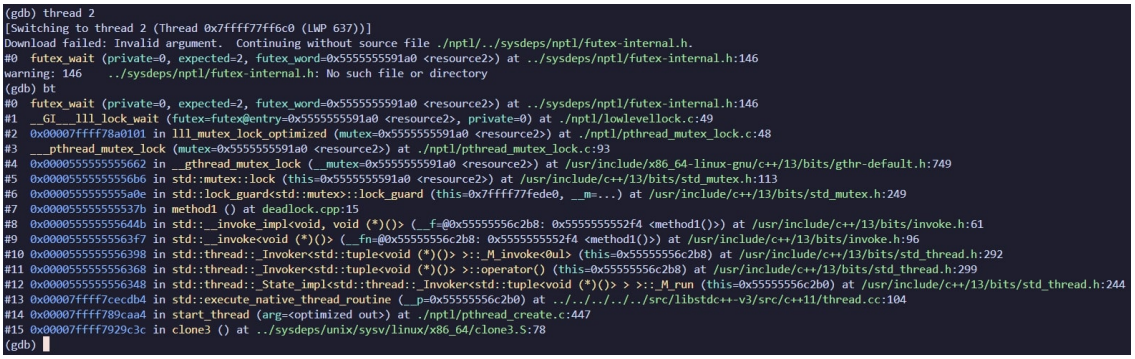

Figure 3.15: Thread-specific backtraces

Again, although much of the information obtained is not that useful, there are still stack frames that can guide the developer to the root of the problem. In this case GDB points to the function *method1()*, specifically at line 15 as the root problem. Thread 2 tries a lock on a resource and puts itself in a permanent state of wait. The debugger also flags *resource2* as the resource Thread 2 is trying to lock, as observable between frames #0 to #5. A similar backtrace can be obtained for Thread 3 if the same process is to be repeated. GDB does not necessarily identify this situation as a deadlock by itself, but still provides crucial information to reach such conclusions.

Now let's use GDB in order to debug the ray tracer application, by applying the same core debugging paths taken in the Visual Studio 2022 section. First and foremost, it is important to note that two different compilers were used to compile the same ray tracer, MSVC for Visual Studio and GCC for GDB. In practice not much differs from the two versions since

CMake is being used to build the project but, for both GCC and Clang CMake builds to work, `target_link_libraries(exec_file_name tbb)` must be added to the build configuration file since GCC does not automatically linking libraries as opposed to MSVC. Since the building process is now made inside the terminal, the debug executable is built with the following process:

```
# Build procedure for Unix-based systems
cmake -B build/Debug -DCMAKE_BUILD_TYPE=Debug
cmake --build build/Debug

# Build procedure for Windows systems
cmake -B build
cmake --build build --config Debug
```

Listing 3.27: Cmake - Building in Debug Mode

With the executable now located in *build/Debug*, GDB can be launched using the command `gdb build/Debug/exec_file_name`. From here, the same steps used in the Visual Studio debugger section is followed in order to verify whether GDB can detect or help with race conditions. Specifically, breakpoints are set at the suspected failure points, where one or more threads may attempt to write to the `std::ofstream` simultaneously. Another breakpoint was also inserted in the line below just to have more control over the execution traces. The objective here is to create a scenario where two or more threads are inside the `std::ofstream` at the same time. By default `scheduler-locking` is set to off, meaning that when `continue`, `step` or `next` is used, all threads run at the same time. As aforementioned, this can changed by turning on the `scheduler-locking` or enabling the step mode, so that each thread can run individually and thus, offering greater control over the execution path. By combining these settings with thread switching and individual thread stepping the following program state can be reached:

```
[Switching to Thread 0x7ffff55eb6c0 (LWP 1600)]

Thread 6 "inOneWeekend" hit Breakpoint 1, write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
33          file << rbyte << ' ' << gbyte << ' ' << bbyte << '\n';
(gdb) info threads
  Id   Target Id                                  Frame
  1    Thread 0x7ffff7f30780 (LWP 1592) "inOneWeekend" 0x000055555588c00 in thread_pool::enqueue<camera::render(hittable const&)::{lambda()#1}>(camera::render(hittable const&)::{lambda()#1}&&) (
    this=0x7fffffffd7b0, task=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/thread_pool.h:43
  2    Thread 0x7ffff75ef6c0 (LWP 1596) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  3    Thread 0x7ffff6dee6c0 (LWP 1597) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  4    Thread 0x7ffff65ed6c0 (LWP 1598) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:34
  5    Thread 0x7ffff5dec6c0 (LWP 1599) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:34
* 6    Thread 0x7ffff55eb6c0 (LWP 1600) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  7    Thread 0x7ffff4dea6c0 (LWP 1601) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:34
  8    Thread 0x7ffff45e96c0 (LWP 1602) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  9    Thread 0x7ffff3de86c0 (LWP 1603) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  10   Thread 0x7ffff35e76c0 (LWP 1604) "inOneWeekend" 0x000055555555ae2a in write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  11   Thread 0x7ffff2de66c0 (LWP 1605) "inOneWeekend" 0x000055555555ae2a in write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  12   Thread 0x7ffff25e56c0 (LWP 1606) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
  13   Thread 0x7ffff1de46c0 (LWP 1607) "inOneWeekend" write_color (file=..., pixel_color=...) at /mnt/d/Code Exercises/Raytracing-in-One-Weekend/src/color.h:33
(gdb)
```

Figure 3.16: GDB - Two threads attempting to write to the same object at the same time

As shown in the figure above, both Thread 10 and Thread 11 attempt to write to the same memory location simultaneously. While this situation may not always lead to a race condition, it strongly indicates that one is likely to occur, highlighting that the code is not thread-safe. Similar to Visual Studio's debugger, inspecting the image file during execution yields little insight, with it only providing mostly metadata about the object and not its content during write operations. To overcome this, the approach of loading the image data into a `std::string` can equally be applied here, evaluating its value, and then writing it to the file. However, stepping through multiple threads in GDB to confirm data races can more tedious and time-consuming compared to Visual Studio's approach. Finally, similar to what was observed in the deadlock test, GDB cannot automatically detect race conditions.

Instead, it relies on the developer to interpret the available data and identify such issues manually.

GDB functions more as an informative debugger rather than a tool that offers direct solutions to developers. While its terminal-based interface can be difficult to navigate, GDB cannot automatically detect deadlocks or race conditions, but it provides highly detailed insights on the objects contained within a program's execution. Debugging multi-threaded environments with GDB largely involves manually examining each component and deducing potential issues, rather than relying on the tool to pinpoint or suggest the root cause. GDB often outputs large volumes of information, much of which may be irrelevant to the developer. Although its setup process is relatively simple, it is still considerably more complex compared to the user-friendly interfaces offered by modern IDEs. Ultimately, while GDB excels at capturing low-level details of concurrent execution, it heavily relies on the interpretation and diagnosis of the developer.

When it comes to overhead, the Debug build registered significant increases of both executable file size and rendering times. In the tests conducted, the Debug build resulting file size was approximately 1.95MB, representing a staggering 527% size increase compared to the 0.31MB of the Release build. While the impact on rendering time was much less dramatic, it was still notable with the same scene in the Debug build completed in roughly 42 seconds and 35 seconds in the Release build, reflecting a 20% time increase. It is also important to note that both performance and size overheads are also influenced by the compiler in use. As such, these results are not directly comparable to those obtained in Visual Studio, which relies on MSVC rather than GCC.

Due to its raw output and terminal-based interface, GDB lacks the user-friendliness found in modern IDEs. However, this is offset by its versatility, as it supports a wide range of programming languages, many of which still remain relevant. When it comes to developing and debugging concurrent applications, GDB is best used in conjunction with external tools that specialize in identifying concurrency issues. One such example is Clang's ThreadSanitizer[43], a tool specifically designed to detect data races during program execution. In conclusion, while GDB is a powerful and flexible debugger, it still lacks built-in capabilities for automatically identifying or suggesting concurrency issues such as deadlocks and data races, a limitation that appears to be shared by the other debuggers analyzed in this study.

Chapter 4

Conclusion

This section aims to summarise the insights gathered from the tests and research conducted in the previous sections. It highlights the advantages and disadvantages of the debuggers studied, suggests potential use cases, outlines possible future work beyond the time constraints of this thesis, and concludes with final thoughts and considerations.

4.1 Results

The evaluation of false positives and false negatives was not addressed in Section 3. While the methods applied generally produce reliable and consistent results, the inherent complexity of concurrent systems introduces challenges. Interactions between various concurrency mechanisms, such as atomic operations, mutual exclusions, and other synchronization techniques, can sometimes lead to incorrect assessments. This includes both false positives, where race conditions are reported despite not existing, and false negatives, where actual race conditions go undetected and the execution path is incorrectly deemed valid.

Based on the data collected and tests conducted during the development phase, it is evident that most modern programming languages and technologies used in professional environments offer a variety of built-in concurrency mechanisms, ranging from standard constructs like mutual exclusions, atomic operations, thread abstractions, and asynchronous functions, to more advanced paradigms that treat concurrency as a core feature, as seen in languages like Go and Erlang. Through the years, these tools have become safer and easier to use, directly impacting and benefiting the multi-threaded scope of software development.

As reiterated throughout this study, debuggers and programming languages differ significantly in how they handle concurrency, both in terms of features and complexity. Languages like C++ give developers low-level control over concurrent constructs, which increases flexibility but also complexity. By contrast, languages like Go abstract concurrency into simpler, high-level constructs. While standardizing concurrency mechanisms across languages might seem beneficial, diversity exists for a reason and different languages are optimized for different domains. For example, Erlang excels at building highly concurrent, asynchronous server-like systems, while C++ is better suited for performance-critical applications such as graphics engines.

While language-level concurrency tools have improved, debuggers, have lagged behind. Most provide only basic support to help developers identify concurrency issues. As demonstrated during testing, identifying deadlocks and race conditions, a fundamental concurrency problem, can still be a challenge, with only Visual Studio standing out for its ability to detect deadlock situations on its own, although its effectiveness still remains language-dependent.

In some cases, setting up debugging environments can also be unnecessarily complex, often hindering productivity. What starts as simple breakpoint management can quickly evolve into a series of micro-adjustments, making the debugging process, in some extreme cases, more bothersome than manual inspection. For instance, GDB often overwhelms users with low-level data. On the other hand, IntelliJ IDEA does a much better job of presenting digestible data but lacks the more advanced capabilities. Visual Studio, despite its dependency on the language being used, emerged as the most feature-complete debugger for concurrent applications, especially with its Parallel Stacks window, which, when effective, offers a clear and intuitive interface for navigating concurrent states. Another critical observation is the performance overhead introduced by debuggers. While necessary for capturing runtime metadata, this overhead can severely impact application performance and size. For example, in the case of the ray tracer, rendering a scene in debug mode instead of release mode can result in hours of additional rendering time, depending on the scene. Though this is a natural trade-off when debugging, it becomes an important consideration for applications that do not require deep inspection or complex debugging environments. Unfortunately, when it comes to detecting concurrent problems by themselves, debuggers still fall very short, often requiring help of external tools that specialise in detecting these problems.

4.2 Future Work

This thesis aimed to highlight how debuggers and language-level tools contribute to building safer and more reliable concurrent applications, especially in multithreaded contexts. Several paths for future work emerge from this study.

While this thesis focused on IntelliJ IDEA, Visual Studio, and GDB, there are many other IDEs and debuggers that could be evaluated. Another promising direction would be to incorporate these debuggers into automated testing workflows. CLI-based debuggers like GDB are well-suited for scripting and automation, but if GUI-based debuggers such as IntelliJ's or Visual Studio's could also be integrated into test frameworks, this could make debugging more interactive and accessible while improving issue detection during testing.

Debuggers still struggle to precisely identify concurrent problems and contextualise the environment in which they occur. Most current debuggers display thread states, shared resources and thread local data and expect the developer to deduce the problem. The next step for debugger evolution would be to develop features that automatically detect these conditions, clearly indicating which threads are involved and highlighting the specific code sections contributing to the issue.

Future research could be expanded to include a broader range of projects, those involving intensive inter-process communication, greater complexity, or distributed architectures across multiple machines or processes. Applying the same methodology used in the previous sections to these more demanding scenarios could yield valuable insights. Additionally, more exhaustive testing, such as the analysis of false positives and false negatives, could significantly impact how a tool or debugger is evaluated in terms of efficiency, robustness, and scalability.

4.3 Final Conclusions

Programming languages have steadily expanded their concurrency toolkits to make development easier, safer, and more robust, especially in performance-critical or communication-heavy systems. Similarly, debuggers have advanced, offering more visualizations and intuitive data representations, but still falling short on analysis. In this thesis, three debuggers, GDB, IntelliJ IDEA, and Visual Studio 2022, were analyzed in terms of scope, performance, usability, and efficiency. Selecting the right debugger is often a matter of the type of application being developed and the concurrency model used.

Debuggers come in many forms, from powerful data visualisation command-line tools like GDB to feature-rich but complex environments like Visual Studio 2022. One major area for improvement lies in how these tools interact with developers, especially for concurrency-related issues. Reducing the learning curve while offering meaningful insights, like race condition and deadlock visualization and smarter breakpoints, should be a priority for future debugger development.

Bibliography

[1] ISEP. "Higher Institution of Engineering of Porto's Code of Conduct". In: *Code of Conduct* (Nov. 2020), pp. 193–204. url: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf.

[2] National Society of Professional Engineers. "NSPE Code of Ethics for Engineers". In: *NSPE Code of Ethics for Engineers* (2019). url: https://www.nspe.org/resources/ethics/code-ethics.

[3] Keith Miller Don Gotterbarn and Simon Rogerson. "ACM Ethics: Software Engineering Code". In: *Software Engineering Code* (Nov. 1997). url: https://ethics.acm.org/code-of-ethics/software-engineering-code/.

[4] Stephen Cleary. "Concurrency in C# Cookbook". In: *Concurrency in C# Sharp Cookbook, 2nd Edition* (2019). url: https://github.com/hungdangit95/IT-Book/blob/master/Concurrency%20in%20C%23%20Cookbook%20-%20Stephen%20Cleary.pdf.

[5] Jungi Jeong and Changhee Jung. "PMEM-Spec: Persistent Memory Speculation". In: *PMEM-Spec: Persistent Memory Speculation* (2021). url: https://dl.acm.org/doi/pdf/10.1145/3445814.3446698.

[6] Luc Maranget, Susmit Sarkar, and Peter Sewell. "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models". In: *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models* (2012). url: https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[7] *Relaxed-Memory Concurrency*. url: https://www.cl.cam.ac.uk/~pes20/weakmemory/.

[8] Study Tonight. *CPU Scheduling in Operating Systems*. url: https://www.studytonight.com/operating-system/cpu-scheduling.

[9] Ana Lúcia de Moura and Roberto Ierusalimschy. "Revisiting Coroutines". In: *Revisiting Coroutines* (2009). url: https://dl.acm.org/doi/pdf/10.1145/1462166.1462167.

[10] The Go blog. *Share memory by communicating*. url: https://go.dev/blog/codelab-share.

[11] Berb. *Actor-based Concurrency*. url: https://berb.github.io/diploma-thesis/original/054_actors.html.

[12] Akka. *How the Actor Model Meets the Needs of Modern, Distributed Systems*. url: https://doc.akka.io/libraries/akka-core/current/typed/guide/actors-intro.html.

[13] C.A.R. Hoare. "Communicating Sequential Processes". In: *Communicating Sequential Processes* (1978). url: https://dl.acm.org/doi/pdf/10.1145/359576.359585.

[14] DEV. *CSP vs Actor model for concurrency*. url: https://dev.to/karanpratapsingh/csp-vs-actor-model-for-concurrency-1cpg.

[15] C++ reference. *C++ Concurrency support library*. url: https://en.cppreference.com/w/cpp/thread.

[16] Charles E. Mcdowell and David P.Helmbold. "Debugging Concurrent Programs". In: *Debugging Concurrent Programs* (1989). url: https://dl.acm.org/doi/10.1145/76894.76897.

[17] Dominik Aumayr. "Debugging Support for Multi-paradigm Concurrent Programs". In: *Debugging Support for Multi-paradigm Concurrent Programs* (2019). url: https://dl.acm.org/doi/10.1145/3359061.3361074.

[18] C reference. *C Concurrency support library*. url: https://en.cppreference.com/w/c/thread.

[19] Aakanksha Verma et al. "Interactive Debugging of Concurrent Programs under Relaxed Memory Models". In: *Interactive Debugging of Concurrent Programs under Relaxed Memory Models* (2020). url: https://dl.acm.org/doi/10.1145/3368826.3377910.

[20] Long Zheng et al. "Towards Concurrency Race Debugging: An Integrated Approach for Constraint Solving and Dynamic Slicing". In: *Towards Concurrency Race Debugging: An Integrated Approach for Constraint Solving and Dynamic Slicing* (2018). url: https://dl.acm.org/doi/10.1145/3243176.3243206.

[21] Cheng Wen et al. "Controlled Concurrency Testing via Periodical Scheduling". In: *Controlled Concurrency Testing via Periodical Scheduling* (2022). url: https://dl.acm.org/doi/10.1145/3510003.3510178.

[22] Zhangyu Chen et al. "Efficiently Detecting Concurrency Bugs in Persistent Memory Programs". In: *Efficiently Detecting Concurrency Bugs in Persistent Memory Programs* (2022). url: Concurrency_Bugs_in_Persitent_Memory.

[23] Oracle. *Virtual Threads*. url: https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html.

[24] JetBrains. *Intellij IDEA IDE*. url: https://www.jetbrains.com/idea/.

[25] David Neves and Hervé Paulino. "Condition-Based Synchronization in Data-Centric Concurrency Control". In: *Condition-Based Synchronization in Data-Centric Concurrency Control* (2022). url: https://dl.acm.org/doi/10.1145/3477314.3507120.

[26] Oliver Moseler, Lucas Kreber, and Stephan Diehl. "ThreadRadar: A Thread-Aware Visualization for Debugging Concurrent Java Programs". In: *ThreadRadar: A Thread-Aware Visualization for Debugging Concurrent Java Programs* (2021). url: https://dl.acm.org/doi/10.1145/3481549.3481566.

[27] Microsoft. *Introduction to PLINQ*. url: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq.

[28] Microsoft. *Visual Studio*. url: https://visualstudio.microsoft.com/.

[29] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. "FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests". In: *FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests* (2022). url: https://doi.org/10.1145/3540250.3558956.

[30] Ziheng Liu et al. "Who Goes First? Detecting Go Concurrency Bugs via Message Reordering". In: *Who Goes First? Detecting Go Concurrency Bugs via Message Reordering* (2022). url: https://dl.acm.org/doi/10.1145/3503222.3507753.

[31] Camden Cheek. *conc: better structured concurrency for go*. url: https://github.com/sourcegraph/conc.

[32] Uber Technologies Inc. *cff: a concurrency toolkit for Go*. url: https://uber-go.github.io/cff.

[33] Peter Shirley, Trevor D Black, and Steve Hollasch. *Ray Tracing in One Weekend Book Series*. url: https://raytracing.github.io.

[34] Oracle Java Documentation. *What is a Socket?* url: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html.

[35] Jetbrains. IntelliJ IDEA. *Detect concurrency issues*. url: https://www.jetbrains.com/help/idea/detect-concurrency-issues.html.

[36] Jetbrains. IntelliJ IDEA. *Debug code*. url: https://www.jetbrains.com/help/idea/debugging-code.html.

[37] Oracle Java Documentation. *CopyOnWriteArrayList*. url: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.

[38] Brett Wooldridge. *HikariCP official repository*. url: https://github.com/brettwooldridge/HikariCP.

[39] Microsoft. *Debug Multithreaded Applications*. url: https://learn.microsoft.com/en-us/visualstudio/debugger/debug-multithreaded-applications-in-visual-studio?view=vs-2022.

[40] Microsoft. *What is "managed code"?* url: https://learn.microsoft.com/en-us/dotnet/standard/managed-code.

[41] Microsoft. *C++: <execution>*. url: https://learn.microsoft.com/en-us/cpp/standard-library/execution?view=msvc-170.

[42] Clang. *ThreadSanatizer*. url: https://clang.llvm.org/docs/ThreadSanitizer.html.

[43] Microsoft. *What is "managed code"?* url: https://learn.microsoft.com/en-us/dotnet/standard/managed-code.