Comparing Threads and Co-routines using a GCC-based Function call tracer

Siddhartha Malladi
115933956
Stony Brook University

Vishnutej Reddy
115934115
Stony Brook University

1 Introduction

Our project explores the performance of C++20 coroutines compared to the traditional multithreaded approach in programming. To accurately assess their efficiency, we developed a GCC-based function call tracer library. This tool logs the entry and exit points of each function and the time stamp, along with the function call backtrace. This tracing library is convenient for developers who wish to debug their programs without making any changes to their program. Preloading this library and executing any user program would print the execution trace into a file. In case of production code, which usually goes in hundreds and thousands of lines, this tracer library comes handy to assert the program execution is inline with the design of the application. As we print the backtrace and timestamp of each function call entry and exit, we can potentially catch bottlenecks in program execution at the function call level and also find potential bugs in the program with respect to the function call execution sequence by inspecting the back-trace.
We tested the performance using the tasks - Matrix multiplication, In-memory Key-value storage, and File I/O operations, all implemented using both coroutines and multithreading. Our findings indicate that matrix multiplication performs better with multithreading due to the absence of synchronisation overhead. This is because Matrix product operation is parallelizable by design into multi threads without the need of any synchronisation. In case of File I/O operations, there is actually contention involved between multiple threads and there is a need for synchronisation between threads using locks. File I/O operations actually benefit from coroutines based implementation because there is no synchronisation overhead of locks in coroutine based implementation. The performance of the in-memory key-value store was similar for both the approaches. This highlights the unique advantages and suitability of each method depending on the task.

2 Motivation

Co-routines are a recent development in C++20, designed to enhance the way programmers handle asynchronous and concurrent programming. Unlike traditional threads, which are managed by the OS and involve significant overhead related to context switching and memory usage, coroutines operate within the application's control, allowing more granular management of execution flow. Also, coroutines are easier to debug because of the single unit of execution with tools like `gdb`. Threads are harder to debug because of the uncertainty involved in thread scheduling because of OS thread scheduler intervention and also multiple threads sharing the same data structures. Given their potential for simplifying code and reducing resource consumption, we embarked on a project to empirically measure the performance of co-routines relative to threads.

To facilitate our performance measurements, we developed our own GCC-based function call tracer library. This custom tool allowed us to instrument executable dynamically, logging detailed information about function entry and exit points along with backtraces, without altering the original program code. Such dynamic instrumentation can help us with a clear view of how coroutines and threads behave at run-time. Additionally, for precise timing measurements, we utilized various timer functions provided by the `libc` API. This combination of custom tracing capabilities and standard timing methods enabled a robust framework for accurately assessing and contrasting the efficiency and resource utilisation of both concurrent programming models.

Let us compare in what scenarios and how co-routines improve upon threads:

Simplification of Asynchronous Code: In traditional asynchronous programming, managing the flow of execution can be cumbersome due to the reliance on callbacks, leading to a situation commonly known as "callback hell," where nested callbacks become difficult to manage and error-prone. Co-routines offer a cleaner solution by using operations such as `co_await` to suspend execution until the awaited task completes, thereby maintaining a logical flow that resembles synchronous code. This is particularly beneficial in I/O operations, which are inherently non-blocking and are traditionally managed with multiple callbacks, potentially complicating error handling and response sequencing.

Efficient resource utilisation: Designed to be stack-less or use split stacks; they suspend their state with a small memory footprint and resume later with minimal overhead. This efficiency is achieved by storing the state of the function in a separate heap-allocated structure, rather than using the call stack. This approach allows for hundreds of thousands of coroutines to be active concurrently without the significant memory overhead associated with traditional thread stacks. As a result, applications that require handling numerous lightweight tasks simultaneously, such as handling user connections on a web server or managing fine-grained tasks within a large-scale computation, can do so more efficiently.

Support for Lazy Evaluation: Co-routines facilitate the creation of lazy-evaluated sequences, which are evaluated only as needed and do not require the pre-computation and storage of entire data sets. This capability is particularly useful for generating data dynamically, which is common in scenarios such as generating infinite sequences or streams of data that depend on user input or other runtime conditions. For example, co-routines can be used to efficiently manage data streams in applications that process large or infinite datasets, allowing them to operate on one element at a time as data becomes available or is required, thus minimising memory usage and enhancing performance.

Improved Readability and Maintainability: By eliminating callbacks and reducing the reliance on complex state machines or external state management libraries, coroutines make codebases easier to read and maintain. This simplification comes from being able to write linear execution flows even in the context of non-linear programming tasks such as event handling or asynchronous processing.

3 Implementation Details

3.1 Tracer Library

GCC provides built-in support for profiling with special hook functions, `__cyg_profile_func_enter` and `__cyg_profile_func_exit` [1]. These functions are invoked automatically by GCC at the beginning and end of each function call when the program is compiled with the `-finstrument-functions` flag. `__cyg_profile_func_enter` is called when a function is entered, and it receives pointers to the function being called and its caller. Similarly, `__cyg_profile_func_exit` is called just before a function returns, also receiving pointers to the current function and its caller. These hooks [2] are powerful tools for building custom profiling utilities, allowing developers to track function execution without modifying the source code, making them ideal for performance analysis and debugging complex software applications.

In our tracer library, we use the `gettimeofday()` function within `__cyg_profile_func_enter` and `__cyg_profile_func_exit` to record precise timestamps for function entries and exits, capturing time down to microseconds. Additionally, we employ `dladdr()` to fetch readable function names from raw pointers for both the called functions and their callers. This method helps log both time and function details simultaneously, providing clear insights into the execution flow and performance of the application.

Applied `__attribute__((no_instrument_function))` attribute to the hook functions and any utility functions they called. This prevents the hooks themselves from being instrumented, avoiding infinite recursion and reducing performance overhead. Compiled the hooks into a shared library (`libhooks.so`), which can be dynamically loaded into the application using `LD_PRELOAD`. This approach allows the tracer to be used with any application without static linkage to the application.

The below is a sample program and output of the tracer:

```
#include <iostream>
#include <unistd.h> // For sleep()
// Function declarations
void C() {
    sleep(7); // Sleep for 7 seconds
}
void B() {
    sleep(5); // Sleep for 5 seconds
    C(); // Call function C
}
void A() {
    sleep(2); // Sleep for 2 seconds
    B(); // Call function B
}
int main() {
    A(); // Start the function stack by
        calling A
    return 0;
}
```

Output of the program when compiled with `-finstrument`, `PRELOAD` and run with `libhooks.so` :

```
1715025463.427757 - Exit: <unknown> called:
    __libc_csu_init
1715025463.427767 - Enter: main called:
    __libc_start_main
1715025463.427789 - Enter: _Z1Av called: main
1715025465.428340 - Enter: _Z1Bv called: _Z1Av
1715025470.428566 - Enter: _Z1Cv called: _Z1Bv
1715025477.428752 - Exit: _Z1Cv called: _Z1Bv
1715025477.428800 - Exit: _Z1Bv called: _Z1Av
```

```
1715025477.428819 - Exit: _Z1Av called: main
1715025477.428832 - Exit: main called:
    __libc_start_main
```

3.2 Concurrency Models:

3.2.1 Matrix Multiplications:

We tested with different matrix sizes as the time to calculate increases exponentially.

Threads: This approach divides the matrix multiplication task into smaller blocks by rows and assigns each block to a separate thread. The threads work in parallel to compute their assigned portions of the result matrix.

Coroutines: Utilizes C++20 coroutines to represent the matrix multiplication work at the finest level (calculating a single element of the result matrix). Each coroutine pauses (yields) after calculating each element.

Pseudo Code:

```
Given matrices $A$ and $B$:
  Initialization: Let $C$ be a new matrix.
  Algorithm:
  • For $i$ from 1 to $n$:
      – For $j$ from 1 to $p$:
          ∗ Let sum $= 0$
          ∗ For $k$ from 1 to $m$:
              · Set sum $\leftarrow$ sum $+ A_{ik} \cdot B_{kj}$
          ∗ Set $C_{ij} \leftarrow$ sum
  Output: Return $C$
```

3.2.2 In-memory Database:

We used an `unordered_map` and a custom class to implement a keyvalue store with get, set, delete, and range query functionalities. A workload generator will generate the operations based on our specified ratio of writes to reads. In the threads, each operation uses mutex locks for safety. We used 10 threads to perform the operations.

3.2.3 File Operations:

Generates random read, write, and search operations. This is an I/O-bound workload. Simple text is written and read from the file. Only 1 `fstream` is used as it wouldn't cause confusion in code and will be simple to manage.

Threads: Number of operations are equally divided among threads. Locking is done for each operations such that only 1 operation could access the file at one point of time.

Co-routines: Each operation is treated as a separate coroutine. As the execution is linear no need of locking mechanism.

3.3 Experimental Setup

Our setup consists of a system running macOS 14 with an 8-core ARM64 processor, 8GB RAM, and a 256GB SSD. We utilized the GCC compiler(13.1) and CMake (version 3.28)

4 Challenges and future enhancements

We included specific compiler flags while compiling tracer library as well as concurrency models. **–export-dynamic**, this flag instructs the linker to add all symbols, not just used ones, to the dynamic symbol table. This is critical because it ensures that function symbols are available for runtime resolution, enabling `dladdr()` to retrieve function names from addresses. We compiled both the tracer lib and concurrency models at **`-O0`** optimization level. For a function call tracer, where accuracy in mapping the compiled code back to the source code is crucial, disabling optimizations ensures that the behavior of the program remains predictable and more closely aligned with the source code. We used **-g** flag to compile tracer library and concurrency models. This flag tells the compiler to include debug information in the executable, which is critical for debugging. The debug information includes data about variable types, structures of data, and the mapping of the program back to the source code.

At first, while implementing concurrency models we relied on C++ STL, for eg: vectors. This resulted in many nested `libc` calls related to C++ STL and caused a heavy instrumentation overhead which is greater than 1000% of execution time without instrumentation. This is because the tracer library instruments every function in the executable that also includes `libc` functions. We later moved on to implement concurrency models without using C++ STL. For eg: use `int**` instead of `vector<vector<int»`. Below is sample trace from tracer when implemented with C++ STL. This includes trace from C++ STL functions which we are not interested in.

```
1714951262.73823 - Enter function: main (0
    x7f4ac640dd3c) called by: __libc_start_main
     (0x7f4ac4dc1c87)
1714951262.73861 - Enter function:
    _Z27performMatrixMultiplicationv (0
    x7f4ac640d825) called by: main (0
    x7f4ac640dd5d)
1714951262.73881 - Enter function: _ZNSaIiEC1Ev
     (0x7f4ac640e1d0) called by:
    _Z27performMatrixMultiplicationv (0
    x7f4ac640d885)
1714951262.73898 - Enter function:
    _ZN9__gnu_cxx13new_allocatorIiEC1Ev (0
    x7f4ac640e94c) called by: _ZNSaIiEC1Ev (0
    x7f4ac640e1fb)
1714951262.73915 - Exit function:
    _ZN9__gnu_cxx13new_allocatorIiEC1Ev (0
    x7f4ac640e94c) called by: _ZNSaIiEC1Ev (0
    x7f4ac640e1fb)
1714951262.73926 - Exit function: _ZNSaIiEC1Ev
    (0x7f4ac640e1d0) called by:
    _Z27performMatrixMultiplicationv (0
    x7f4ac640d885)
1714951262.73937 - Enter function:
    _ZNSt6vectorIiSaIiEEC1ESt16initializer_ (0
    x7f4ac640e248) called by:
    _Z27performMatrixMultiplicationv (0
    x7f4ac640d8ad)
1714951262.73950 - Enter function:
    _ZNSt12_Vector_baseIiSaIiEEC2ERKS0_ (0
    x7f4ac640e9c4) called by:
    _ZNSt6vectorIiSaIiEEC1ESt16initializer_ (0
    x7f4ac640e29f)
1714951262.73965 - Enter function:
    _ZNSt12_Vector_baseIiSaIiEE12_Vector_ (0
    x7f4ac640f480) called by:
    _ZNSt12_Vector_baseIiSaIiEEC2ERKS0_ (0
    x7f4ac640e9fa)
1714951262.73980 - Enter function:
    _ZNSaIiEC1ERKS_ (0x7f4ac640f9fc) called by:
     _ZNSt12_Vector_baseIiSaIiEE12_Vector_ (0
    x7f4ac640f4b6)
1714951262.73996 - Enter function:
    _ZN9__gnu_cxx13new_allocatorIiEC2ERKS1_ (0
    x7f4ac64102d0) called by: _ZNSaIiEC1ERKS_
    (0x7f4ac640fa32)
1714951262.74010 - Exit function:
    _ZN9__gnu_cxx13new_allocatorIiEC2ERKS1_ (0
    x7f4ac64102d0) called by: _ZNSaIiEC1ERKS_
```

A simple way to tackle above problem is, before printing function trace in the tracer library, check if the function names match a string pattern associated with the functions in our concurrency models. Only print the functions which match the pattern. This significantly reduces instrumentation overhead. We will use this logic to enhance our tool in future.

Currently, we are only fetching function entry, exit timestamps and names in the tracer library. We cannot differentiate between function calls made by different threads. We can enhance the tracer library in future to read the function arguments also to differentiate between the threads.

5 Results

From our performance measurements for matrix multiplication in Figure. 1 using GCC-based function traces and C++ timer functions, the threads perform much better compared to co-routines as the matrix size increases well beyond 1000. Co-routines demonstrate negligible time consumption similar to threads for matrices of size below 100 but as the matrix size scales beyond 1000 there is a divergence in the way

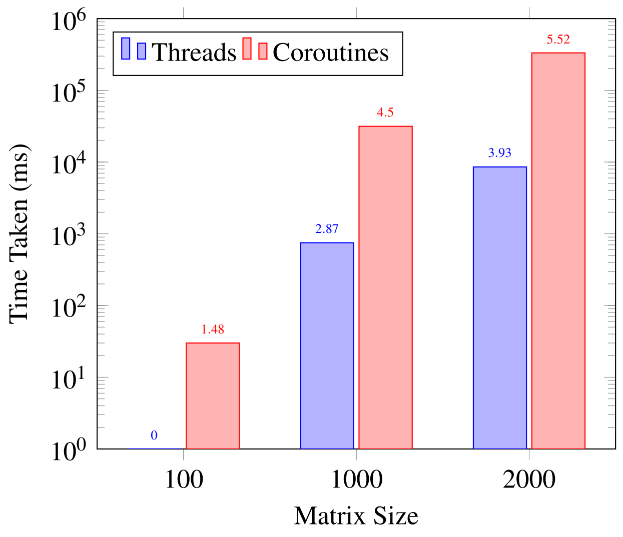

Figure 1: Performance Comparison of Matrix Multiplication

both graphs scale. Because the matrix multiplication operation can be divided parallelly into different threads with each thread handling product of a unique pair of row and column at a time, there is no synchronization or communication overhead between the threads. As matrix size increases beyond 1000 threads, they are able to leverage hardware parallelism better. This suggests that threads are better suited for computationally intensive tasks without synchronization overhead because they better leverage hardware parallelism. In-memory

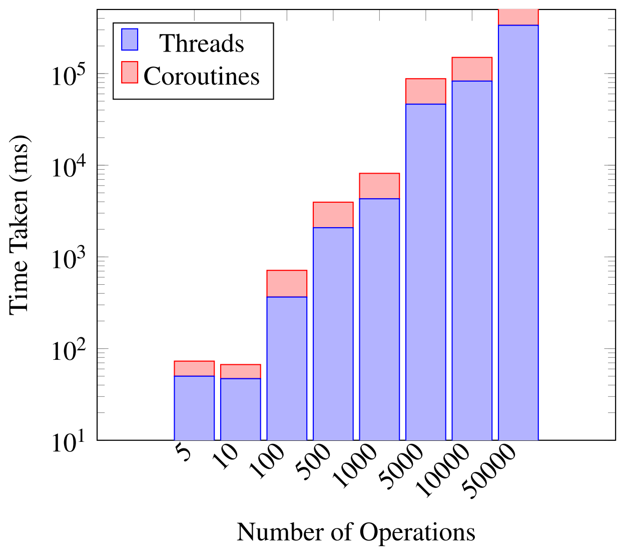

Figure 2: Performance Comparison of file operations

database operations does not show significant difference in performance between threads and coroutines. In Figure 3, threads outperformed coroutines by only 17%, a difference that may not justify the typical power and memory costs of threads.

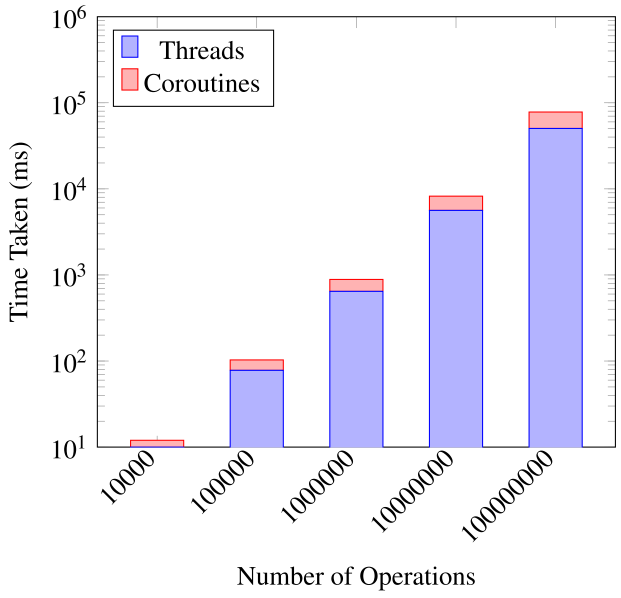

Figure 3: Performance comparison of in-memory database operations

We did a performance analysis on File I/O operations between threads and co-routines in Figure. 2. As you scale the operations beyond 10000, the coroutines perform at least 50% better compared to threads. This improvement is mainly attributed to synchronization overhead between threads when performing I/O operations. We included conditional variables - shared mutexes, while implementing threads to avoid race conditions. The coroutine implementation doesn't involve any condition variables, instead it relies on coroutine primitives co_await and co_yield in its implementation and hence it scales better compared to threads when you increase the number of operations.

6 Conclusion

Our project clearly demonstrates the importance of selecting an appropriate concurrency model based on the specific requirements of an application's workload.

For operations like Matrix multiplication, which can be decomposed into multiple parallel tasks, threads showed better performance. This advantage stems from threads' ability to fully utilize hardware parallelism, operating independently on different segments of data without interference.

Conversely, for file I/O operations, which often involve significant synchronization overhead between threads during read-write processes, Coroutines proved to be more effective. Coroutines minimize the performance degradation typically associated with thread synchronization, offering a scalable alternative for managing high-concurrency I/O tasks.

In the case of in-memory key-value stores, the performance was similar between threads and Coroutines, with a slight advantage seen with Coroutines due to their lower context-switching overhead. Since locking in key-value stores is generally restricted to specific elements rather than entire structures, the reduction in locking overhead with Coroutines does not translate into a substantial performance gain as it does with file I/O.

Overall, our project confirms that the decision on whether to use coroutines or threads should be strategically made, reflecting the operational demands of the application. While coroutines provide significant benefits for tasks with frequent synchronization, threads remain superior for computationally intensive tasks that are inherently parallelizable.

References

[1] Using the gnu compiler collection (gcc): instrumentation options. Accessed on May 6, 2024.

[2] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A. Selcuk Uluagac. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, Jun 2017.