FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

EVALUATION OF THREAD POOL IMPLEMENTATIONS BY RESOLUTION OF WEBSERVER REQUESTS

ING. MARTIN MUCHA   SUPERVISOR: ING. DANIEL LANGR, PH.D.

Motivation

With the increasing number of cores in modern CPUs and the growing popularity of architectures like ARM, which offers a relaxed memory model, effective scheduling and synchronization are critical to unlocking performance benefits.

One well-known design pattern for parallel execution is the thread pool, a collection of pre-spawned threads available to execute tasks. This pattern significantly reduces the overhead of creating new threads for each task, and it has become widespread in areas such as high-performance computing (HPC) and web servers.

However, the C++23 standard still lacks built-in support for thread pools or other higher-level parallel constructs, forcing developers to rely on third-party libraries. These libraries often present challenges in terms of complexity, performance limitations and integration with modern C++ features. To address these shortcoming, we developed a custom library (**Coros**) with focus on :

- Seamless integration with modern C++.
- High performance across multiple workloads.
- Ease of use, reducing the complexity for developers.

Conclusion

Coroutines, as a fundamental building block for both parallel and asynchronous libraries, have shown promising results. Our implementation outperforms most of the compared libraries in various benchmarks. Stress testing under different server request scenarios demonstrated that, with minor adjustments, our solution can efficiently handle a wide range of workloads.

Furthermore, our library proves that with modern C++ standard tools and careful management of the memory model, a fast and efficient cross-platform multithreading library can be built.

Publication

The original solution was enhanced by adding new features and conducting additional testing on ARM architecture. The library is currently available as open source.

Problem

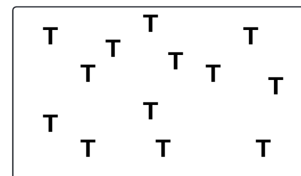
Our research identified two main design challenges that must be solved to build a parallel library:

- **Representation individual tasks/workload**: Simple functions are insufficient when tasks need to be paused and resumed.
- **Scheduling/mapping of tasks onto threads**: Efficiently assigning tasks to available threads is critical for performance.

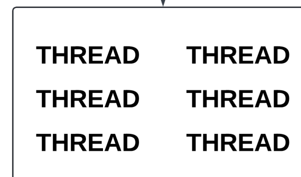
Design

To address the first problem, we employed a **novel approach** using C++20 coroutines. This method enables straightforward, cross-platform encapsulation of tasks.

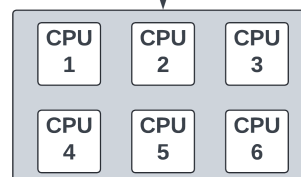
For the scheduling problem, we developed our own parallel constructs and synchronization primitives, leveraging the coroutine interface. This in combination with a **lock-free work-stealing** approach, allows for efficient task scheduling/mapping.



OUR LIBRARY



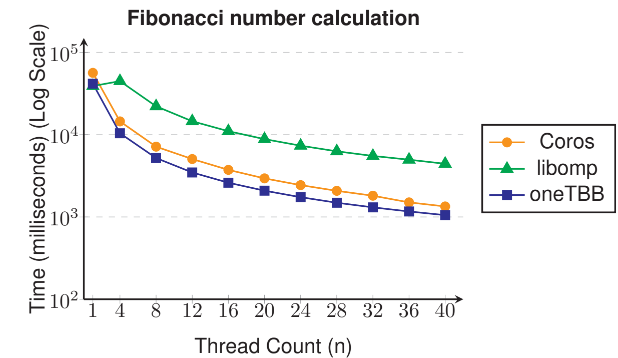
OPERATING SYSTEM

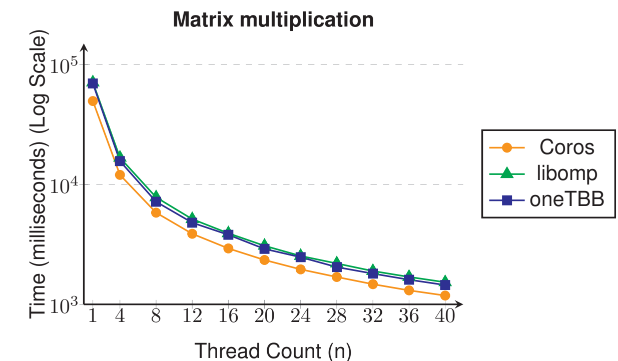


- **Zero additional stack space**: Thanks to utilization of coroutine-to-coroutine control transfer, no extra stack space is consumed, compared to regular function calls.
- **Relaxed atomics**: These are a core part of the library, offering performance gains through weaker synchronization.
- **Memory arena**: Since suspension points are known at compile time, memory can be reused effectively between individual suspension points.
- **Exception propagation**: Coroutines allow for easy exception capture, and when combined with modern C++ features, this enables elegant exception propagation to the user for efficient handling.

Performance

- Evaluated various implementations and their impact on performance.
- Tests covered both scheduling efficiency and memory consumption.
- All tests were performed on an AMD EPYC 7H12 64-Core Processor.





Server testing

We conducted additional testing with different types of web requests. The results showed:

- **Work-stealing** within thread pool proved to be a more efficient scaling method compared to horizontal scaling in certain scenarios.
- **Asynchronous workloads** can be efficiently handled by our existing solution with minor modifications, leveraging the coroutine interface.