https://doi.org/10.31891/csit-2025-1-9
UDC 004.89: 004.3

Dmytro PUYDA
Lviv Polytechnic National University

SCHEDULING: A FAST C++ THREAD POOL IMPLEMENTATION CAPABLE OF EXECUTING TASK GRAPHS

In this paper, the author presents a simple and efficient C++ thread pool implementation capable of executing task graphs. The conducted experiments demonstrate that the proposed solution achieves CPU performance comparable to Taskflow, a highly optimized library for parallel and heterogeneous programming. The implementation is small and straightforward, consisting of less than one thousand lines of C++ code at the time of writing this paper.

Besides its potential applications in commercial and scientific projects, the code presented in this paper is also part of the System Programming course taught at the Department of Electronic Computing Machines at Lviv Polytechnic National University. Due to its simplicity, the code has proven to be a good introduction to asynchronous programming and task graphs, helping students to understand the subject and pass technical interviews at IT companies.

At the time of writing this paper, the project has gained 30 stars on GitHub and has 5 unique cloners and 29 unique visitors over the past two weeks.

Some technical aspects, related to the use of work-stealing deques in the proposed solution, were discussed with Prof. Tsung-Wei Huang, a developer of Taskflow.

The proposed solution offers a simple yet powerful alternative to existing task systems for C++ projects of varying complexity. It is fast, simple, and easy to use. The minimalistic design ensures that users do not sacrifice CPU performance for features which they do not need. New features can be easily added if required, as the solution is open-source, concise, and easy to understand. The previous results are published in the archive https://arxiv.org/html/2407.15805v2.

The author would like to thank Jason Turner and Prof. Tsung-Wei Huang for useful discussions. The author also would like to thank Chunel Feng for bringing up an interesting example that helped to improve the implementation of the suggested solution.

Keywords: Chase-Lev deque, multithreading, task graph, thread pool, work-stealing.

Дмитро ПУЙДА
Національний університет «Львівська політехніка»

SCHEDULING: ШВИДКА РЕАЛІЗАЦІЯ ПУЛУ ПОТОКІВ З МОЖЛИВІСТЮ ВИКОНАННЯ ГРАФІВ ЗАДАЧ НА С++

У статті автор представляє просту та ефективну реалізацію пулу потоків на C++, здатну виконувати графи задач. Проведені експерименти показують, що запропоноване рішення забезпечує продуктивність ЦП, співмірну з Taskflow, високооптимізованою бібліотекою для паралельного та гетерогенного програмування. Реалізація невелика і проста, на момент написання цієї статті складається з менше тисячі рядків коду C++.

Окрім можливого застосування в комерційних і наукових проектах, код, представлений у цій статті, також є частиною курсу «Системне програмування», який викладається на кафедрі електронних обчислювальних машин Національного університету «Львівська політехніка». Завдяки своїй простоті код виявився хорошим вступом до асинхронного програмування та графів завдань, допомагаючи студентам зрозуміти тему та пройти технічну співбесіду в IT-компаніях.

На момент написання цієї статті проект отримав 30 зірок на GitHub і має 5 унікальних клонувальників і 29 унікальних відвідувачів за останні два тижні.

Деякі технічні аспекти, пов'язані з використанням у пропонованому рішенні кодів крадіжки робіт, обговорювалися з професором Цун-Вей Хуангом, розробником Taskflow.

Пропоноване рішення пропонує просту, але потужну альтернативу існуючим системам завдань для C++-проектів різної складності. Він швидкий, простий і простий у використанні. Мінімалістичний дизайн гарантує, що користувачі не жертвуватимуть продуктивністю ЦП заради функцій, які їм не потрібні. За потреби можна легко додати нові функції, оскільки рішення є відкритим, лаконічним і легким для розуміння. Попередні результати опубліковані в архіві https://arxiv.org/html/2407.15805v2.

Автор хотів би подякувати Джейсону Тернеру та професору Цунг-Вей Хуанг за корисні обговорення. Автор також хотів би подякувати Чунелю Фену за наведений цікавий приклад, який допоміг покращити реалізацію запропонованого рішення.

Ключові слова: дек Чейза-Лева, багатопотоковість, граф завдань, пул потоків, викрадення роботи.

Introduction

Multithreading plays an important role in modern software development. When used wisely, adding more threads can significantly improve the CPU performance of your application. However, it is well known that using more threads does not always mean getting better performance. There are at least two common issues that can happen in multi-threaded software. Firstly, when the number of threads exceeds the capabilities of your hardware, context switching can have a dramatic impact on the performance of your application. Secondly, creating and destroying threads frequently can have significant performance overhead.

To overcome these issues, thread pools are widely used. Instead of creating and destroying threads directly, developers submit tasks to a thread pool instance. A thread pool typically creates a specified number of worker threads running in the background. When there are no tasks, the worker threads are idle. When a task is submitted, one of the available worker threads eventually picks up the task and executes it. If all worker threads are busy executing other tasks, the new task remains in a task queue until one of the worker threads becomes available.

As of C++23, there is no thread pool in the ISO standard. However, there are a variety of libraries providing thread pool implementations that can be used in production. To name just a few: Intel TBB [1], Boost.Asio [2], Taskflow [3, 4], CGraph[5], BS::thread_pool [6,7], etc. Still, many commercial projects use their own thread pool and task system implementations. For example, a large project the author worked on had a complex task system with extensive usage of C++ macros.

Software description

In this paper, the author suggests a minimalistic, simple and fast work-stealing thread pool implementation capable of executing task graphs. Benchmarks comparing the suggested implementation with Taskflow are provided. The suggested implementation uses C++20 but can be updated to comply with older versions of the standard if needed.

The proposed solution has the following advantages:

- It is fast and developed with performance in mind. See the benchmark results below for details.
- It is simple, short, and minimalistic. At the time of writing this paper, the solution consists of less than one thousand lines of C++ code. New features can be added easily if required.
- It does not use third-party dependencies and relies only on the C++20 ISO standard.

Software architecture

The solution has two classes exposed to the user:

- ThreadPool: A work-stealing thread pool implementation that can be used to execute asynchronous tasks and task graphs.
- Task: Represents a node in a task graph and allows users to build task graphs.

Chase-Lev deque

The idea of work-stealing queues is to provide each worker thread with its own task queue to reduce thread contention. When a task is submitted, it is pushed to one of the queues. The thread owning the queue can eventually pick up the task and execute it. If there are no tasks in the queue owned by a worker thread, the thread attempts to steal a task from another queue.

Work-stealing queues are typically implemented as lock-free deques. The owning thread pops elements at one end of the deque, while other threads steal elements at the other end.

Implementing a work-stealing deque is not an easy task. The Chase-Lev deque [8,9] is one of the most commonly used implementations of such a deque. The reference C11 and ARMv7 implementations, as well as the proof of correctness of the ARMv7 code, are given in [9]. However, [9] does not include the proof of correctness of its C11 implementation. Corrections of the C11 implementation provided in [9] were suggested [10, 11]. Later, a few proofs of correctness were given (see, e.g., [12, 13]). Frameworks for automatic inference and validation of memory fences have also been used to validate Chase-Lev deque implementations (e.g., [11, 14]).

The original C11 implementation of the Chase-Lev deque [9], as well as many updated implementations (e.g., [11]), uses atomic thread fences without associated atomic operations. When compiling with Clang thread sanitizer, GCC 13 issues a warning saying that ‘atomic_thread_fence’ is not supported with ‘-fsanitize=thread’. Thread sanitizers may produce false positives when atomic thread fences are used. For example, when using the Taskflow implementation of the work-stealing deque, the sanitizer detects data races in the solution suggested in this paper. The Taskflow implementation of the deque contains the following lines of code:

```
std::atomic_thread_fence(std::memory_order_release);
_bottom[p].data.store(b + 1, std::memory_order_relaxed);
```

If memory_order_relaxed is replaced here by memory_order_release, the sanitizer stops detecting the data races. This might indicate a false positive related to the usage of std::atomic_thread_fence. It is worth noting that Taskflow unit tests and examples pass with the sanitizer even though std::atomic_thread_fence is used.

An example of a work-stealing deque implementation that does not use std::atomic_thread_fence can be found in Google Filament [15], licensed under the Apache License 2.0. When using the implementation from Google Filament, the sanitizer does not detect data races in the suggested solution. The Filament implementation had to be updated to make the deque variable-sized.

Concurrent push and pop operations are not allowed in most implementations of the work-stealing deque. To ensure that there are no concurrent push and pop operations, mappings from thread ID to queue indices are typically used. When a task is pushed to or popped from a queue, the correct queue is usually found using the current thread ID. Unlike this typical approach, the solution suggested in this paper uses a thread-local variable to find the correct task queue. Unfortunately, at the time of writing this paper, there seems to be not enough compiler support to present the suggested solution in the form of a cross-platform module.

Task graphs

To execute task graphs, simple wrappers over an std::function<void()> are used. Each wrapper stores references to successor tasks and the number of uncompleted predecessor tasks. When the thread pool executes a task, it first executes the wrapped function. Then, for each successor task, it decrements the number of uncompleted predecessor tasks. One of the successor tasks, for which the number of uncompleted predecessor tasks becomes equal to zero, is then executed on the same worker thread. Other successor tasks, for which the number of uncompleted predecessor tasks becomes equal to zero, are submitted to the same thread pool instance for execution.

Benchmarks

Experiments demonstrate that, thanks to its simplicity and minimalism, the solution achieves good CPU performance compared to alternatives used in both scientific and commercial projects. Figures 1 and 2 present some benchmark examples at the time of writing this paper, highlighting the efficiency of the proposed solution. Taskflow examples and benchmarks are used to compare the proposed solution and Taskflow. In Figure 1, to evaluate execution of a large number of asynchronous tasks, we use the Taskflow example that calculates Fibonacci numbers recursively without memoization. In Figure 2, to evaluate execution of a simple task graph, we use the Taskflow example of matrix multiplication. More benchmark results can be found at https://github.com/dpuyda/scheduling.

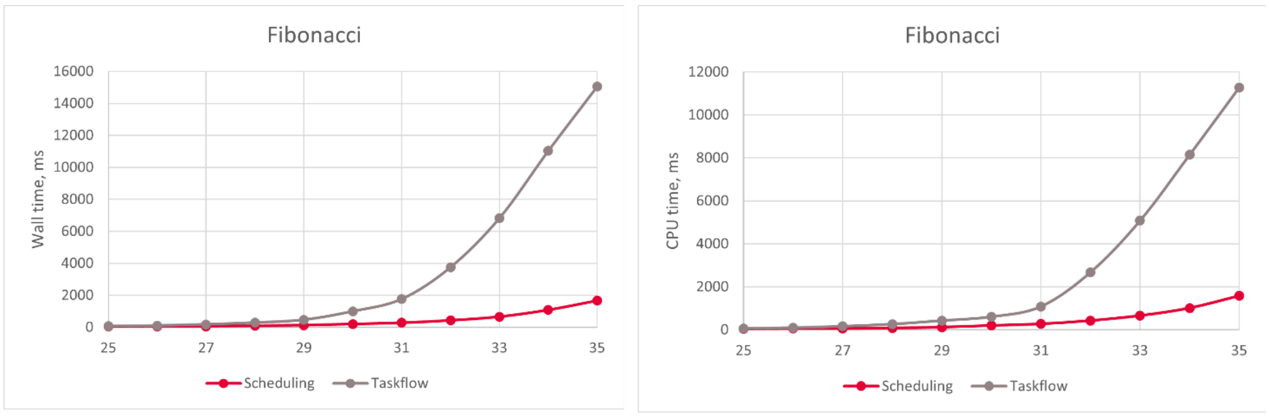

Fig. 1. Fibonacci benchmarks

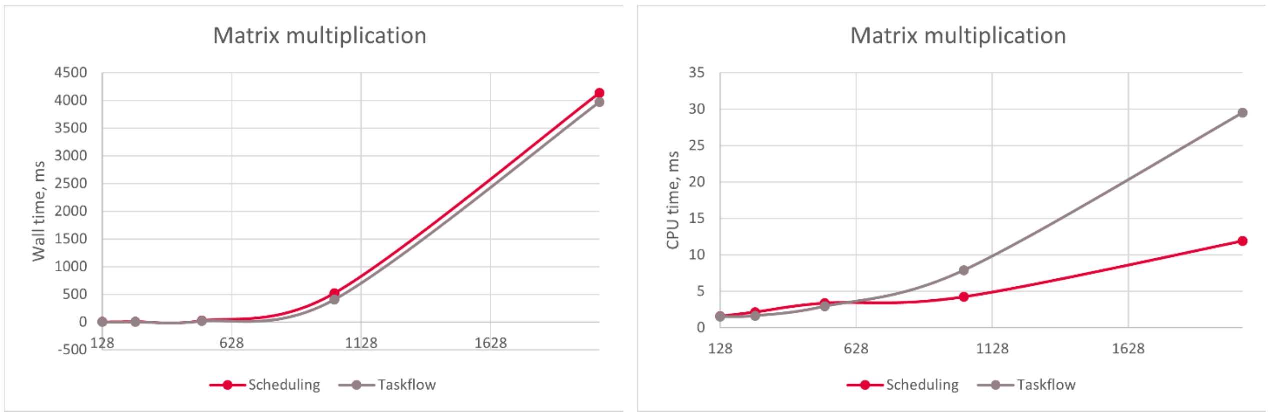

Fig. 2. Matrix multiplication benchmarks

Software functionalities

The proposed software allows users to:

- Submit asynchronous tasks for execution to a thread pool instance.
- Build task graphs and submit them for execution to a thread pool instance.

Illustrative examples

In this section, we give an idea of how to use the solution suggested in this paper. More details about how to use the suggested solution can be found at https://github.com/dpuyda/scheduling.

Asynchronous tasks

Here, we briefly describe how to execute asynchronous tasks using the suggested solution.
To execute an asynchronous task, first, create a ThreadPool instance. For example:

```
#include "scheduling/scheduling.hpp"
```

```
...
scheduling::ThreadPool thread_pool;
```

In the constructor, the ThreadPool class creates several worker threads that will be running in the background until the instance is destroyed. As an optional argument, the constructor of the ThreadPool class accepts the number of threads to create. By default, the number of threads is equal to std::thread::hardware_concurrency().

When the ThreadPool instance is created, submit a task. For example:

```
thread_pool.Submit([] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "Completed\n";
});
```

A task is a function that does not accept arguments and returns void. Use lambda captures to pass input and output arguments to the task if needed. Eventually, the task will be executed on one of the worker threads owned by the ThreadPool instance.

Task graphs

Here, we briefly describe how to execute task graphs using the suggested solution.

A task graph is a collection of tasks and dependencies between them. Dependencies between tasks define the order in which the tasks should be executed. Consider a simple illustrative example of a task graph. Assume that we want to calculate the arithmetic expression (a + b) * (c + d), and that each operation in this expression (including getting the values of a, b, c, and d) takes time. To optimize execution time, we can start by getting the values of a, b, c, and d in parallel. Then, once we know the values of a and b, we can start calculating the value of a + b, and once we know the values of c and d, we can start calculating the value of c + d. The values of a + b and c + d can be calculated in parallel. Once a + b and c + d are calculated, we can start calculating the product (a + b) * (c + d).

The code snippets below illustrate how to execute the above task graph using the suggested solution. To define a task graph, create an iterable collection of Task instances. For example:

```
#include "scheduling/scheduling.hpp"
...
std::vector<scheduling::Task> tasks;
```

Add elements to tasks. For example, add tasks to calculate the value of (a + b) * (c + d) asynchronously. First, add tasks to get the values of a, b, c and d:

```
int a, b, c, d;

auto& get_a = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  a = 1;
});

auto& get_b = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  b = 2;
});

auto& get_c = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  c = 3;
});

auto& get_d = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  d = 4;
});
```

Next, add tasks to calculate a + b and c + d:

```
int sum_ab, sum_cd;
```

```
auto& get_sum_ab = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  sum_ab = a + b;
});

auto& get_sum_cd = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  sum_cd = c + d;
});
```

Finally, add the task to calculate the product (a + b) * (c + d):

```
int product;

auto& get_product = tasks.emplace_back([&] {
  std::this_thread::sleep_for(std::chrono::seconds(1));
  product = sum_ab * sum_cd;
});
```

When all tasks are added, define task dependencies. The task get_sum_ab should be executed after get_a and get_b. Similarly, the task get_sum_cd should be executed after get_c and get_d. Finally, the task get_product should be executed after get_sum_ab and get_sum_cd:

```
get_sum_ab.Succeed(&get_a, &get_b);
get_sum_cd.Succeed(&get_c, &get_d);
get_product.Succeed(&get_sum_ab, &get_sum_cd);
```

When dependencies between tasks are defined, create a ThreadPool instance and submit the task graph for execution:

```
scheduling::ThreadPool thread_pool;
thread_pool.Submit(tasks);
```

Impact & Conclusions

Besides its potential applications in commercial and scientific projects, the code presented in this paper is also part of the System Programming course taught at the Department of Electronic Computing Machines at Lviv Polytechnic National University. Due to its simplicity, the code has proven to be a good introduction to asynchronous programming and task graphs, helping students to understand the subject and pass technical interviews at IT companies.

At the time of writing this paper, the project has gained 30 stars on GitHub and has 5 unique cloners and 29 unique visitors over the past two weeks.

Some technical aspects, related to the use of work-stealing deques in the proposed solution, were discussed with Prof. Tsung-Wei Huang, a developer of Taskflow.

The proposed solution offers a simple yet powerful alternative to existing task systems for C++ projects of varying complexity. It is fast, simple, and easy to use. The minimalistic design ensures that users do not sacrifice CPU performance for features which they do not need. New features can be easily added if required, as the solution is open-source, concise, and easy to understand. The previous results are published in the archive https://arxiv.org/html/2407.15805v2.

Acknowledgements

The author would like to thank Jason Turner and Prof. Tsung-Wei Huang for useful discussions. The author also would like to thank Chunel Feng for bringing up an interesting example that helped to improve the implementation of the suggested solution.

References

1. https://github.com/oneapi-src/oneTBB
2. https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio.html
3. https://github.com/taskflow/taskflow
4. Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin and Yibo Lin, Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System, IEEE Transactions on Parallel and Distributed Systems, Volume 33,
5. https://github.com/ChunelFeng/CGraph
6. https://github.com/bshoshany/thread-pool

7. Barak Shoshany, A C++17 thread pool for high-performance scientific computing, SoftwareX, vol. 26, 101687 (2024)
8. David Chase and Yossi Lev, Dynamic circular work-stealing deque, SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, pp. 21--28 (2005)
9. Nhat Minh Lê, Antoniu Pop, Albert Cohen and Francesco Zappa Nardelli, Correct and efficient work-stealing for weak memory models, PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 69--80 (2013)
10. Brian Norris and Brian Demsky, CDSchecker: checking concurrent data structures written with C/C++ atomics, ACM SIGPLAN Notices, Volume 48, Issue 10, pp. 131--150 (2013)
11. Peizhao Ou and Brian Demsky, AutoMO: automatic inference of memory order parameters for C/C++11, ACM SIGPLAN Notices, Volume 50, Issue 10, pp. 221--240 (2015)
12. Suha Orhun Mutluergil and Serdar Tasiran, A mechanized refinement proof of the Chase–Lev deque using a proof system, Computing, Volume 101, pp. 59–-74 (2018)
13. Jaemin Choi, Formal Verification of Chase-Lev Deque in Concurrent Separation Logic, arXiv:2309.03642 (2023)
14. Michael Kuperstein, Martin Vechev and Eran Yahav, Synthesis of memory barriers, US patent, US8839248B2 (2014)
15. https://github.com/google/filament

Дмитро Пуйда **Dmytro Puyda**	PhD, Assistant of the Department of Electronic Computing Machines, Lviv Polytechnic National University, Lviv, Ukraine, e-mail: dmytro.v.puida@lpnu.ua dpuyda@gmail.com	Кандидат фізико-математичних наук, асистент кафедри електронних обчислювальних машин, Національний університет «Львівська політехніка».