The Journal of Open Source Software

A C++17 Thread Pool for High-Performance Scientific Computing

Barak Shoshany[1]

1 Brock University

DOI: 10.21105/joss.03291

Software

- Review
- Repository
- Archive

Editor: George K. Thiruvathukal

Reviewers:

- @zeroset
- @cedricchevalier19

Submitted: 08 May 2021
Published: 29 June 2021

Summary

Modern scientific research often requires analyzing, processing, and/or simulating a large amount of data. Such computational tasks typically require more computing power than the average laptop can provide, and are executed on high-end desktop computers, computer clusters, or supercomputers instead. However, the main benefit of such high-performance computing systems is not in being able to compute a single task faster; rather, it is in being able to compute many different tasks in parallel, a technique called multithreading (Williams, 2019). The goal of the C++ thread pool class we present here is to allow users, even those unfamiliar with multithreading, to create high-performance multithreaded software via an intuitive and easy-to-use class interface.

Statement of need

Since C++11, the C++ (Stroustrup, 2013) standard library has included built-in low-level multithreading support using constructs such as `std::thread`. However, `std::thread` creates a new thread each time it is called, which can have a significant performance overhead. Furthermore, it is possible to create more threads than the hardware can handle simultaneously, potentially resulting in a substantial slowdown.

In this paper we present a modern C++17-compatible (ISO, 2017) thread pool implementation, the class `thread_pool`, which avoids these issues by creating a fixed pool of threads once and for all, and then reusing the same threads to perform different tasks throughout the lifetime of the pool. By default, the number of threads in the pool is equal to the maximum number of threads that the hardware can run in parallel.

The user submits tasks to be executed into a queue. Whenever a thread becomes available, it pops a task from the queue and executes it. Each task is automatically assigned an `std::future`, which can be used to wait for the task to finish executing and/or obtain its eventual return value.

In addition to `std::thread`, the C++ standard library also offers the higher-level construct `std::async`, which may internally utilize a thread pool - but this is not guaranteed, and in fact, currently only the MSVC implementation (Microsoft Corporation, 2016) of `std::async` uses a thread pool, while GCC and Clang do not. Using our custom-made thread pool class instead of `std::async` allows the user more control, transparency, and portability.

High-level multithreading APIs, such as OpenMP (OpenMP Architecture Review Board et al., 2020), allow simple one-line automatic parallelization of C++ code, but they do not give the user precise low-level control over the details of the parallelization. The thread pool class presented here allows the programmer to perform and manage the parallelization at the lowest

level, and thus permits more robust optimizations, which can be used to achieve considerably higher performance.

We performed performance tests using our thread pool class on a 12-core / 24-thread high-end desktop computer and a 40-core / 80-thread Compute Canada node, using GCC on Linux. We chose to benchmark matrix operations as they are very commonly used in many different types of scientific software, while also being among the easiest operations to parallelize. On both test systems, for matrix multiplication and generation of random matrices, we found a speedup factor roughly equal to the number of CPU cores plus 30%, which saturates the upper bound of the expected speedup from a hyperthreading CPU (Casey, 2011).

This library was created with simplicity and ease-of-use in mind, and is aimed at computational researchers in all fields of science, from students to experts. Our hope is that anyone with intermediate knowledge of C++ and algorithms will be able to use our thread pool class to incorporate high-performance multithreading into their scientific software smoothly and efficiently.

Overview of features

- **Fast:**
  - Built from scratch with maximum performance in mind.
  - Suitable for use in high-performance computing nodes with a very large number of CPU cores.
  - Compact code, to reduce both compilation time and binary size.
  - Reusing threads avoids the overhead of creating and destroying them for individual tasks.
  - A task queue ensures that there are never more threads running in parallel than allowed by the hardware.
- **Lightweight:**
  - Only ~180 lines of code, excluding comments, blank lines, and the two optional helper classes.
  - Single header file: simply `#include "thread_pool.hpp"`.
  - Header-only: no need to install or build the library.
  - Self-contained: no external requirements or dependencies. Does not require OpenMP or any other multithreading APIs. Only uses the C++ standard library, and works with any C++17-compliant compiler.
- **Easy to use:**
  - Very simple operation, using a handful of member functions.
  - Every task submitted to the queue automatically generates an `std::future`, which can be used to wait for the task to finish executing and/or obtain its eventual return value.
  - Optionally, tasks may also be submitted without generating a future, sacrificing convenience for greater performance.
  - The code is thoroughly documented using Doxygen comments - not only the interface, but also the implementation, in case the user would like to make modifications.
- **Additional features:**
  - Automatically parallelize a loop into any number of parallel tasks.
  - Easily wait for all tasks in the queue to complete.
  - Change the number of threads in the pool safely and on-the-fly as needed.

- Fine-tune the sleep duration of each thread's worker function for optimal performance.
- Monitor the number of queued and/or running tasks.
- Pause and resume popping new tasks out of the queue.
- Synchronize output to a stream from multiple threads in parallel using the `synced_stream` helper class.
- Easily measure execution time for benchmarking purposes using the `timer` helper class.

Usage example

In the following simple example, we create a thread pool with 4 threads, and submit 12 tasks to be executed by the threads. Each task simply sleeps for half a second and then prints out `Task i done`, where `i` is the number of the task. We monitor the execution progress while the tasks are being executed. The `synced_stream` helper class is used to synchronize the output from different tasks.

```
#include "thread_pool.hpp"

void sleep_half_second(const size_t &i, synced_stream *sync_out)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    sync_out->println("Task ", i, " done.");
}

void monitor_tasks(const thread_pool *pool, synced_stream *sync_out)
{
    sync_out->println(pool->get_tasks_total(),
                      " tasks total, ",
                      pool->get_tasks_running(),
                      " tasks running, ",
                      pool->get_tasks_queued(),
                      " tasks queued.");
}

int main()
{
    synced_stream sync_out;
    thread_pool pool(4);
    for (size_t i = 0; i < 12; i++)
        pool.push_task(sleep_half_second, i, &sync_out);
    monitor_tasks(&pool, &sync_out);
    std::this_thread::sleep_for(std::chrono::milliseconds(750));
    monitor_tasks(&pool, &sync_out);
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    monitor_tasks(&pool, &sync_out);
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    monitor_tasks(&pool, &sync_out);
}
```

The output of this program will be (up to the order of execution of the tasks):

```
12 tasks total, 0 tasks running, 12 tasks queued.
```

```
Task 0 done.
Task 1 done.
Task 2 done.
Task 3 done.
8 tasks total, 4 tasks running, 4 tasks queued.
Task 4 done.
Task 5 done.
Task 6 done.
Task 7 done.
4 tasks total, 4 tasks running, 0 tasks queued.
Task 8 done.
Task 9 done.
Task 10 done.
Task 11 done.
0 tasks total, 0 tasks running, 0 tasks queued.
```

Many other examples, as well as detailed documentation for the available member functions and variables, may be found in the `README.md` file in the GitHub repository.

References

Casey, S. D. (2011). How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, *6*(1 p.11). https://software.intel.com/content/www/us/en/develop/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-wi html

ISO. (2017). *ISO/IEC 14882:2017: Programming languages — c++*. ISO. https://www.iso.org/standard/68564.html

Microsoft Corporation. (2016). *<future> functions*. https://docs.microsoft.com/en-us/cpp/standard-library/future-functions.

OpenMP Architecture Review Board, Klemm, M., & Supinski, B. R. de. (2020). *OpenMP application programming interface specification version 5.1*. ISBN: 9781795759885

Stroustrup, B. (2013). *The c++ programming language: The c++ programm lang_p4*. Pearson Education. ISBN: 9780133522853

Williams, A. (2019). *C++ concurrency in action*. Manning Publications. ISBN: 9781617294693