Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

A C++17 thread pool for high-performance scientific computing

Barak Shoshany

Department of Physics, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, Ontario, L2S 3A1, Canada

ARTICLE INFO

Keywords:
C++
Parallel computing
High-performance computing

ABSTRACT

We present a modern C++17-compatible thread pool implementation, built from scratch with high-performance scientific computing in mind. The thread pool is implemented as a single lightweight and self-contained class, and does not have any dependencies other than the C++17 standard library, thus allowing a great degree of portability. In particular, our implementation does not utilize any high-level multithreading APIs, and thus gives the programmer precise low-level control over the details of the parallelization, which permits more robust optimizations. The thread pool was extensively tested on both AMD and Intel CPUs with up to 40 cores and 80 threads.

Code metadata

Current code version	v4.0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00214
Permanent link to Reproducible Capsule	
Legal Code License	MIT license
Code versioning system used	Git
Software code languages, tools, and services used	C++
Compilation requirements, operating environments	None
If available Link to developer documentation/manual	https://github.com/bshoshany/thread-pool/blob/master/README.md
Support email for questions	bshoshany@brocku.ca

1. Introduction

Multithreading [1] is essential for modern high-performance computing. Since C++11, the C++ [2–4] 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.

The library presented here contains a thread pool class, `BS::thread_pool`, which avoids these issues by creating a fixed pool of threads once and for all, and then continuously reusing the same threads to perform different tasks throughout the lifetime of the program. 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 retrieves the next task from the queue and executes it. The pool automatically produces an `std::future` for each task, which allows the user to wait for the task to finish executing and/or obtain its eventual return value, if applicable. Threads and tasks are autonomously managed by the pool in the background, without requiring any input from the user aside from submitting the desired tasks.

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 depends on the implementation. Using our custom-made thread pool class instead of `std::async` ensures that a thread pool is indeed utilized on any C++17-compliant compiler, and gives the user much more fine-tuned control, transparency, and portability.

The design of this package was guided by four important principles. First, *compactness*: the entire library consists of just one small self-contained header file, with no other components or dependencies. Second, *portability*: the package only utilizes the C++17 standard library [5], without relying on any compiler extensions or 3rd-party

E-mail address: bshoshany@brocku.ca.

https://doi.org/10.1016/j.softx.2024.101687
Received 29 July 2022; Received in revised form 14 May 2023; Accepted 10 March 2024
Available online 22 March 2024

libraries, and is therefore compatible with any modern standards-conforming C++17 compiler on any platform. Third, *ease of use*: the package is extensively documented, and programmers of any level should be able to use it right out of the box.

The fourth and final guiding principle is *performance*: each and every line of code in this library was carefully designed with maximum performance in mind, and performance was tested and verified on a variety of compilers and platforms. Indeed, the library was originally designed for use in the author’s own computationally-intensive scientific computing projects, running both on high-end desktop/laptop computers and high-performance computing nodes.

Other, more advanced multithreading libraries may offer more features and/or higher performance. However, they typically consist of a vast codebase with multiple components and dependencies, and involve complex APIs that require a substantial time investment to learn. This library is not intended to replace these more advanced libraries; instead, it was designed for users who do not require very advanced features, and prefer a simple and lightweight package that is easy to learn and use and can be readily incorporated into existing or new projects.

2. Overview of features

Our thread pool package was built from scratch with maximum performance in mind, and is suitable for use in high-performance computing nodes with a very large number of CPU cores. The code is compact, to reduce both compilation time and binary size. Reusing threads by using a thread pool avoids the overhead of creating and destroying them for individual tasks, and a task queue ensures that there are never more threads running in parallel than allowed by the hardware.

The package is lightweight, provided as a single header file, BS_thread_pool.hpp, with only 304 lines of code, excluding comments, blank lines, and lines containing only a single brace. Since it is a header-only package, there is no need to install or build the library. The package is also self-contained, with no external requirements or dependencies. As it uses only the C++ standard library, it is fully portable and works with any C++17-compliant compiler on any system.

The thread pool class itself has only a handful of member functions, so the package is simple and easy to use. Every task submitted to the queue using the submit_task() member function automatically generates an std::future, which can be used to wait for the task to finish executing and/or obtain its eventual return value. Loops can be automatically parallelized into any number of parallel tasks using the submit_loop() member function, which similarly generates a future. If futures are not needed, tasks may be submitted using detach_task(), and loops can be parallelized using detach_loop() – sacrificing convenience for even 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. The included test program BS_thread_pool_test.cpp can be used to perform exhaustive automated tests and benchmarks, and also serves as a comprehensive example of how to properly use the package.

The helper class template BS::multi_future can be used to track the execution of multiple futures at once – for example, when using submit_loop(). In addition, the optional header file BS_thread_pool_utils.hpp contains several useful utility classes: BS::synced_stream allows synchronizing output to a stream from multiple threads in parallel, BS::timer easily measures execution time for benchmarking purposes, and BS::signaller allows sending simple signals between threads.

To wait for all tasks in the queue to complete, the user may use the wait(), wait_for(), and wait_until() member functions. Changing the number of threads in the pool safely and on-the-fly can be done as needed using the reset() member function. The number of queued and/or running tasks can be monitored using the get_tasks_queued(), get_tasks_running(), and get_tasks_total() member functions. The pool may be freely paused and resumed using the pause(), unpause(), and is_paused() member functions; when paused, threads do not retrieve new tasks out of the queue.

3. Code example

We refer the reader to the README.md file in the library’s GitHub repository for the full documentation, including many code examples demonstrating every single feature of the library. Here we will just provide a simple program which demonstrates some of the main features, along with line-by-line analysis.

The sample program encrypts or decrypts any string using the ROT13 algorithm (shift each letter up by 13, cycling from Z back to A). In practice, there is no reason to use multithreading for such a trivial operation, but it is intended to simulate parallelizing a more computationally expensive encryption algorithm, which would be too long to include here.

```
 1  #include "BS_thread_pool.hpp"       // BS::multi_future, BS::thread_pool
 2  #include "BS_thread_pool_utils.hpp" // BS::synced_stream
 3  #include <chrono>                   // std::chrono
 4  #include <string>                   // std::string
 5  #include <thread>                   // std::this_thread
 6
 7  int main()
 8  {
 9      BS::synced_stream sync_out;
10      BS::thread_pool pool(4);
11      std::string input = "Gurer vf n gurbel juvpu fgngrf gung vs rire nalbar
            qvfpbiref rknpgyl jung gur Havirefr vf sbe naq jul vg vf urer, vg jvyy
            vafgnagyl qvfnccrne naq or ercynprq ol fbzrguvat rira zber ovmneer naq
            varkcyvpnoyr. Gurer vf nabgure gurbel juvpu fgngrf gung guvf unf
            nyernql unccrarq.";
12      std::string output(input.size(), ’ ’);
13      auto loop = [&input, &output, &sync_out](const size_t a, const size_t b)
14      {
15          sync_out.println("Submitted range [", a, ",", b, ")");
16          for (size_t i = a; i < b; ++i)
17          {
18              if (input[i] >= ’A’ && input[i] <= ’Z’)
19                  output[i] = (((input[i] - ’A’) + 13) % 26) + ’A’;
20              else if (input[i] >= ’a’ && input[i] <= ’z’)
21                  output[i] = (((input[i] - ’a’) + 13) % 26) + ’a’;
22              else
23                  output[i] = input[i];
24              std::this_thread::sleep_for(std::chrono::milliseconds(5));
25          }
26          sync_out.println("Finished range [", a, ",", b, ")");
27      };
28      const BS::multi_future<void> mf = pool.submit_blocks<size_t>(0,
            input.size(), loop);
29      sync_out.println("Waiting for the computation to complete... ");
30      mf.wait();
31      sync_out.println("Output: ", output);
32  }
```

In line 1, we include the header file BS_thread_pool.hpp, which contains the main BS::thread_pool class and the BS::multi_future helper class template. In line 2, we include the optional header file BS_thread_pool_utils.hpp, which contains the BS::synced_stream utility class, which is used to print synchronized output to a stream from multiple threads. In line 9, we create the object sync_out from the BS::synced_stream class.

In line 10, we create the thread pool itself, from the main thread pool class BS::thread_pool. We name the object pool and specify the desired number of threads, 4, as an argument; if this is not specified, the total number of available hardware threads will be used.

In line 11, we provide the input string; since ROT13 is its own inverse, the same code can be used for both encryption and decryption. In line 12 we create an empty string in which to store the output. Next, we would like to loop over the input string’s characters and encode them one by one. For improved performance, we would like to parallelize this loop using the thread pool.

In line 28, we call the thread pool’s submit_blocks() member function. This function works similarly to submit_loop(), which we mentioned above, except that it allows the user to handle each parallelized block of indices as a whole, instead of each index individually.

The template argument of submit_blocks() specifies the type of the indices, in this case size_t. The first two function arguments specify the range of the loop, in this case from 0 up to (but not including) `input.size()`. The third argument is the function specifying the actual code of the loop that is to be parallelized.

`submit_blocks()` returns an object from the helper class template `BS::multi_future`. This object, which we name `mf`, acts as a container of futures; they do not return any values (hence, the return type is specified as `void`), and are only used to wait until the loop finished, which we do using the `wait()` member function in line 30. Note that this is a crucial step; if we do not wait for the future, then in line 31, when we print the output, the loop may still be in progress, and the output may be incomplete.

When `submit_blocks()` runs, it will split the range `[0,input.size())` into 4 blocks - one for each thread in the pool. The number of blocks can be changed by specifying it as an additional argument to `submit_blocks()`, after the loop function.

In lines 13–27 we define a lambda expression, `loop`, which captures the input and output strings, as well as the synchronized stream. The lambda expression has two arguments, `a` and `b`, which will be populated by `submit_blocks()` with the first and last index of each block respectively. We print these indices in line 12 to demonstrate how the loop is split into blocks.

In line 16 we specify a `for` loop, which is the actual loop to be parallelized. Any loop parallelized by `submit_blocks()` should have a `for` statement with these exact parameters. This means that each run of the lambda expression `loop` will perform part of the overall loop from 0 to `input.size()`, with the start and end indices of each part (or block) determined automatically by `submit_blocks()`.

If we used `submit_loop()` instead, the `loop` function would have been simpler, accepting only a single index. In this example we chose to use `submit_blocks()` for pedagogical purposes, so that we can print out the range of indices of each block explicitly. Please see the `README.md` file in the library's GitHub repository for examples of using `submit_loop()`.

In lines 18–23 we perform the actual ROT13 encoding. If the character is a letter, we add 13 to its ASCII value, cycling back from Z to A. Otherwise, we keep the character as is. In line 24 we sleep for 5 ms, to simulate a more complicated algorithm. In line 26, we notify that the calculation of this specific block has ended.

In line 29, after submitting the parallelized loop, we print a message informing the user that we are waiting for the loop to complete. Finally, in line 31, after waiting for the `BS::multi_future`, we print the output string. In these lines, as well as lines 15 and 26, we use the `println()` member function of `BS::synced_stream`. This function takes an arbitrary number of arguments, and prints them one after the other.

The `BS::synced_stream` class uses an `std::mutex` internally to ensure that when two different threads try to print to the stream, the second thread will wait until the first thread is finished before printing its own output. This ensures that the outputs from the different threads do not overlap.

Here is a sample output of this program:

```
Waiting for the computation to complete...
Submitted range [0,69)
Submitted range [69,138)
Submitted range [138,207)
Submitted range [207,275)
Finished range [0,69)
Finished range [69,138)
Finished range [138,207)
Finished range [207,275)
Output: There is a theory which states that if ever anyone discovers exactly
        what the Universe is for and why it is here, it will instantly disappear
        and be replaced by something even more bizarre and inexplicable. There is
        another theory which states that this has already happened.
```

4. Use in research

Since its first release in April 2021, the package has received more than 1,850 stars on GitHub, and many fixes, improvements, and new features were added based on contributions from the community. Even before this paper was published, the package was cited in several research works, such as [6–8], and [9]. The package is also being used by the author and his students for several ongoing scientific computing projects, and is being used by many other software developers for other scientific and non-scientific purposes.

5. Performance tests

The bundled test program, `BS_thread_pool_test.cpp`, performs simple benchmarks by filling a specific number of vectors of fixed size with values. The program decides how many vectors to use, and of what size, by testing how many are needed to reach a certain target duration in a single-threaded computation. This ensures that the test takes approximately the same amount of time on all systems, and is thus more consistent and portable.

Once the appropriate number and size of vectors has been determined, the program allocates the vectors and fills them with values, calculated using a simple mathematical formula to simulate real-world computations. This operation is performed both single-threaded and multithreaded, with the multithreaded computation spread across multiple tasks submitted to the pool.

Several multithreaded tests are performed, and the program keeps increasing the number of blocks submitted to the pool until it finds the optimal value. Often, the optimal number of blocks is much higher than the number of hardware threads, but if the number is too high it will result in diminishing returns. Each test is repeated multiple times, with the run times averaged over all runs of the same test. The run times of the tests are compared, and the maximum speedup obtained is calculated.

As an example, here are the results of the benchmarks from a Digital Research Alliance of Canada node equipped with two 20-core/40-thread Intel Xeon Gold 6148 CPUs (for a total of 40 cores and 80 threads), running CentOS Linux 7.9.2009. The tests were compiled using GCC v13.2.0 with the `-O3` and `-march=native` flags. In these tests, the library was able to speed up numerical calculations by 56.9x when utilizing 80 concurrent threads compared to a single thread. The output of the test program was as follows:

```
======================
Performing benchmarks:
======================
Using 80 threads.
Determining the number of elements to generate in order to achieve an
      approximate mean execution time of 50 ms with 80 tasks...
Each test will be repeated up to 30 times to collect reliable statistics.
Generating 27962000 elements:
[......]
Single-threaded, mean execution time was 2815.2 ms with standard deviation 3.5
      ms.
[......]
With   2 tasks, mean execution time was 1431.3 ms with standard deviation 10.1
      ms.
[.......]
With   4 tasks, mean execution time was 722.1 ms with standard deviation 11.4 ms.
[..............]
With   8 tasks, mean execution time was 364.9 ms with standard deviation 10.9 ms.
[.............................]
With  16 tasks, mean execution time was 181.9 ms with standard deviation 8.0 ms.
[...............................]
With  32 tasks, mean execution time was 110.6 ms with standard deviation 1.8 ms.
[...............................]
With  64 tasks, mean execution time was 64.0 ms with standard deviation 6.3 ms.
[...............................]
With 128 tasks, mean execution time was 59.8 ms with standard deviation 0.8 ms.
[...............................]
With 256 tasks, mean execution time was 59.0 ms with standard deviation 0.0 ms.
[...............................]
With 512 tasks, mean execution time was 52.8 ms with standard deviation 0.4 ms.
[...............................]
With 1024 tasks, mean execution time was 50.7 ms with standard deviation 0.9 ms.
[...............................]
With 2048 tasks, mean execution time was 50.0 ms with standard deviation 0.5 ms.
[...............................]
With 4096 tasks, mean execution time was 49.4 ms with standard deviation 0.5 ms.
[...............................]
With 8192 tasks, mean execution time was 50.2 ms with standard deviation 0.4 ms.
Maximum speedup obtained by multithreading vs. single-threading: 56.9x, using
      4096 tasks.

++++++++++++++++++++++++++++++++++++++++
Thread pool performance test completed!
++++++++++++++++++++++++++++++++++++++++
```

These two CPUs have 40 physical cores in total, with each core providing two separate logical cores via hyperthreading, for a total of 80 threads. Without hyperthreading, we would expect a maximum theoretical speedup of 40x. With hyperthreading, one might naively expect to achieve up to an 80x speedup, but this is in fact impossible, as each pair of hyperthreaded logical cores share the same physical core's resources. However, generally we would expect an estimated 30% additional speedup [10,11] from hyperthreading, which amounts to around 52x in this case. The speedup of 56.9x in our performance test exceeds this estimate.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This publication is dedicated to my wife, Pauline. Her endless love, support, and encouragement have been a great source of motivation for working on this and other projects.

References

[1] Williams A. C++ Concurrency in Action. Manning Publications; 2019.

[2] Stroustrup B. The C++ Programming Language. Pearson Education; 2013.

[3] Stroustrup B. Programming: Principles and Practice Using C++. Pearson Education; 2014.

[4] Stroustrup B. A Tour of C++. Pearson Education; 2018.

[5] ISO/IEC 14882:2017. Programming Languages - C++. ISO; 2017, URL https://www.iso.org/standard/68564.html.

[6] Welsh C, Xu J, Smith L, König M, Choi K, Sauro HM. libRoadRunner 2.0: A High-Performance SBML Simulation and Analysis Library. 2022, http://dx.doi.org/10.48550/arXiv.2203.01175, URL https://arxiv.org/abs/2203.01175.

[7] Lehmusvaara J. Robust Multi-Class Decision Trees. 2021, URL https://trepo.tuni.fi/bitstream/handle/10024/136142/LehmusvaaraJohannes.pdf.

[8] Liu W, Wan Y, Zhang Y, Yao Y, Liu X, Shi L. Efficient Matching of LiDAR Depth Map and Aerial Image Based on Phase Mean Convolution. Geomat Inf Sci Wuhan Univ 2022. http://dx.doi.org/10.13203/j.whugis20210524, URL http://ch.whu.edu.cn/en/article/doi/10.13203/j.whugis20210524.

[9] Marks B, Yang H. Two-Phase Commit Using Blockchain. 2022, URL https://www.scs.stanford.edu/22sp-cs244b/projects/Two-Phase%20Commit%20Using%20Blockchain.pdf.

[10] Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, et al. Hyper-Threading Technology Architecture and Microarchitecture. 6, (1):2002, p. 4–15, URL https://www.moreno.marzolla.name/teaching/HPC/vol6iss1_art01.pdf.

[11] Casey SD. How to Determine the Effectiveness of Hyper-Threading Technology with an Application, URL https://web.archive.org/web/20210422160435/https://software.intel.com/content/www/us/en/develop/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application.html.