T-holder: A Custom Thread-Holding Library

Mingun Cho
mgcho@ucdavis.edu
University of California, Davis
Davis, California, USA

Henry Chou
hechou@ucdavis.edu
University of California, Davis
Davis, California, USA

Qin Liu
qinli@ucdavis.edu
University of California, Davis
Davis, California, USA

Piyush Kumbhare
pakumbhare@ucdavis.edu
University of California, Davis
Davis, California, USA

Aaron Nguyen
aanguy@ucdavis.edu
University of California, Davis
Davis, California, USA

Kevin Nhu
knhu@ucdavis.edu
University of California, Davis
Davis, California, USA

1 Introduction

Parallel computing is essential for modern high-performance computing. By leveraging multiple processors, parallel computing allows tasks to be divided and executed simultaneously, significantly reducing execution time and enhancing scalability. Currently, programmers rely on multi-threaded libraries such as `pthreads`, `OpenMP`, and `std::thread` from the standard C++ library to implement parallelism in C or C++ programs. However, they all experience an overhead for creating and launching new threads due to the allocation of resources to each thread, leading to a significant performance degradation. While it is possible to amortize the thread spawning overhead by making the threads spend most of their time working, a program that involves multiple sections of parallelism and synchronization will face a significant performance overhead due to multiple costs of thread creation.

Existing strategies aiming to mitigate thread creation overhead typically employ static thread pools, where a predetermined number of threads remain idle and ready to perform assigned tasks. While static thread pools effectively reduce the number of system calls by frontloading thread creation, they introduce significant limitations. For instance, if an application utilizes a static thread pool of size eight, eight `clone3` syscalls are made at initialization. If a ninth task is queued for parallel execution, the application encounters two common scenarios:

(1) The thread creation function blocks until an existing thread becomes available, introducing a bottleneck and severely impacting performance in scenarios demanding high concurrency;
(2) The thread pool dynamically allocates new threads upon reaching capacity, improving performance but increasing memory usage since these additional threads and their associated resources persist for the duration of the application's execution.

Both scenarios present clear trade-offs, forcing developers to choose between performance degradation or increased memory consumption.

To address these challenges, we introduce **T-holder**[1], a custom thread-holding library designed to efficiently manage thread lifecycle by dynamically reusing threads instead of repeatedly invoking costly system calls. Unlike traditional static thread pools, Tholder employs an idle thread management system featuring configurable thread timeout values. When a thread completes its assigned task, rather than immediately terminating, it remains idle within a global thread pool, ready for quick reassignment. If the thread remains idle beyond a configurable duration, it exits gracefully, freeing associated resources.

T-holder significantly reduces overhead from frequent `clone3` syscalls, particularly benefiting programs that repeatedly spawn and join threads across multiple parallel sections. When tasks are requested via `tholder_create()`, the library prioritizes assigning work to existing idle threads, creating new threads only when necessary. Additionally, T-holder employs a dynamic resizing strategy for its internal thread pool structure. When the existing thread pool is saturated, the pool size dynamically doubles. However, memory allocation for new threads occurs lazily—only when a queued task requires execution—minimizing memory footprint while maintaining high performance.

Furthermore, T-holder is developed with ease of integration in mind, offering an API fully compatible with existing multi-threaded programs that utilize pthreads model. Developers can transition from pthreads to T-holder by simply replacing the pthreads function calls with the following core functions provided by T-holder:

(1) `int tholder_create(...)` - Searches for an inactive thread in the pool and assigns tasks to either an existing idle thread or creates a new thread if needed;
(2) `int tholder_join(tholder_t th, void **thread_return)` - Blocks execution until the specified thread finishes its task, utilizing condition variables for efficient synchronization;
(3) `void tholder_init(size_t num_threads)` - Preallocates thread pool slots for optimized performance, with an optional default initialization of eight threads;
(4) `void tholder_destroy()` - Cleans up the thread pool and associated resources, recommended to be called at the end of the main thread, as it is not thread-safe.

By reducing the overhead of frequent thread creation and destruction, T-holder aims to improve the execution time for programs with multiple sections of parallelism and synchronization. Since threads are being reused more efficiently instead of constantly invoking system calls, we expect a noticeable decrease in thread management overhead, leading to more efficient CPU utilization and reduced latency in task execution. This would be particularly beneficial for applications that frequently spawn and join threads as discussed more in Section 2.1. With the use of T-holder, we anticipate that our custom thread-holding library will lead to faster execution times and more efficient resource utilization with high

[1]https://github.com/MangoShip/ECS251Project

adaptability and usability given that the existing pthreads system calls can be seamlessly replaced by that of T-holder's.

2 Evaluation methods and experimental setup

We evaluate the performance of the T-holder on the UCD CSIF computer (Intel i7-9700 @ 3.00GHz, 8 cores) by averaging the execution time of 10 runs per configuration of each parallel algorithm (Section 2.1). We run our experiments on Ubuntu 22.04 and compile our program using GCC 11.4.0 and `-O3` optimization flag. For the performance comparison, we also collect the execution time of sequential, pthreads, and OpenMP variants of each algorithm. For OpenMP implementation, we also additionally collect results when enabling a global flag `OMP_WAIT_POLICY` which allows reusing threads between each parallel section by making idle threads spin for a short period of time before going to sleep.

2.1 Algorithms

We plan to leverage existing algorithms that contain multiple, alternating parallel and serial sections of the code to assess the performance of T-holder. These algorithms have multiple parallel code sections that require creating and launching threads multiple times which incurs significant overhead. By creating threads once and reusing them across the parallel sections, we would like to assess the extent to which T-holder reduces this overhead. We choose the following 5 algorithms that meet these requirements.

2.1.1 Parallel Merge Sort. Parallel merge sort is structured with an initial phase where subarrays are sorted concurrently in a parallel manner while a subsequent serial phase handles the merging of these sorted segments [5]. This recurring pattern of multiple parallel and serial phases aligns with our requirements as we suspect that since the number of threads required might be consistent across the sorting stages, persistent threads can yield significant performance improvements without leading to resource under-utilization.

2.1.2 Parallel Radix Sort. Parallel radix sort [3] involves reading a binary bit of each number in a list in parallel starting from the least significant bit. The total number of 0 and 1 bits is then collected to construct a histogram which is used to compute a new index of each number in a list. This process continues until reading the most significant bit of each number, then the list will be sorted after applying the new indexes. With a series of parallel and sequential sections, this algorithm is well-suited for evaluating the performance of the T-holder.

2.1.3 Cholesky Factorization Algorithm. The next algorithm we propose to use is the Cholesky factorization algorithm [4], which decomposes a symmetric positive-definite matrix into a product of a lower triangular matrix and its transpose. In its parallel implementation, independent computations (such as updating sub-matrices) are performed concurrently, followed by a serial synchronization step to enforce data dependencies. This algorithm is well-suited here since the computation alternates between parallel matrix updates and serial synchronization steps, ensuring that the parallel work is executed more efficiently during factorization of matrices, especially for ones that are extremely large.

2.1.4 Parallel Breadth-First Search. In terms of Parallel Breadth-First Search [2], a level-by-level traversal of a graph is implemented where each level's nodes are processed concurrently before a serial phase aggregates the discovered neighbors for the next level to be processed again in a parallel manner, and so on. By maintaining a stable pool of threads across the levels of the graph, thread recreation at each level is prevented. This reuse of threads across level-synchronous parallel phases cuts down on the overhead inherent in repeatedly creating threads for each new graph level.

2.1.5 Parallel PageRank. Finally, we will test our threading library with Parallel PageRank [1]. In this algorithm, the ranking of nodes in the graphs is computed in a repeated manner over several iterations, where each iteration consists of a parallel phase (updating node scores based on neighbor contributions) and a serial phase (aggregating and normalizing results). Our threading library could maintain a stable pool of threads that persist across the iterative phases, which might yield performance benefits since PageRank is often used on large-scale graphs with many pages linked with each other.

3 Results

We will now discuss the results of the T-holder library tested on all five aforementioned algorithms, compared to other well-known parallel implementations, such as pthreads and OpenMP. To maintain consistency, all testing was completed on a Linux Machine (courtesy of UC Davis Computer Science Instructional Facility), which features an Intel Core i7-9700 processor with 8 cores and 8 threads. The in-depth results can be accessed through this link. [2]

3.1 Parallel Merge Sort

The first algorithm we tested our T-holder library with is Merge Sort. However, our T-holder-based iterative variant performed with mixed results depending on the size of the array used. For instance, the array with 10 million elements resulted in T-holder only being faster than that of the OpenMP merge sort version but for the biggest array size of 1 billion elements, T-holder was faster than both the recursive OpenMP version and the Iterative pthreads version. For both 16 and 32 threads, the T-holder version again performed with mixed results. In fact, the T-holder gets overtaken by more of the other parallelized versions as we get to bigger and bigger arrays, and the T-holder version of merge sort becomes one of the slowest parallelized ones by the time that the 1 billion sized array is used. We suspect that these diminishing returns are likely due to synchronization overhead of our T-holder library as well as increased contention for shared physical resources. However, not all algorithms underperformed with the T-holder library as some performed better compared to the other parallelized methods, with the most notable improvements being shown under Parallel Radix Sort.

3.2 Parallel Radix Sort

Next, we tested the performance of the Parallel Radix Sort algorithm. In summary, the T-holder implementation resulted in the

[2]https://docs.google.com/spreadsheets/d/1FfpMFS5kUe0nz_Cc1lMXzODLZfXkx2zO4nF5duY2loU/edit?usp=sharing

highest performance, followed by pthreads, OpenMP, OpenMP with OMP_WAIT_POLICY enabled, and the sequential variant of Radix Sort. Since OMP_WAIT_POLICY consumes CPU cycles due to thread spin before going to sleep, this negatively affects the performance of any sequential work between parallel sections. As the size of a list (N) increases, we also observe a slight increase of reduce in the execution time of the T-holder implementation over the pthreads implementation, demonstrating that the T-holder scales well with the Parallel Radix Sort.

3.3 Cholesky Factorization

We then proceeded to test the parallel performance of the Cholesky Factorization algorithm. Like Merge Sort, our T-holder implementation performed with mixed results depending on the size of the positive-definite matrix being decomposed. On one hand, the T-holder implementation exhibited the best performance when tested with smaller matrix sizes such as 1000x1000 and 2000x2000. However, as the matrix size progressively increased to 4000x4000 and 8000x8000, the T-holder implementation began lagging behind the pthread-based parallel implementation and the OpenMP-based implementation, with and without OMP_WAIT_POLICY enabled. Much like Merge Sort, it is suspected that these diminishing returns that we are experiencing are likely attributed to the difference in overhead incurred by synchronization between our T-holder implementation and other parallelized versions of this algorithm.

For smaller matrix sizes, there is noticeably less work to be done for each parallel step where every off-diagonal element for each matrix column is computed. As a result, a smaller workload may benefit from a simpler scheduling or locking mechanism that our T-holder implementation offers, as a result of a significant synchronization overhead difference. For larger matrix sizes, however, there are noticeably more elements per matrix column, so considerably more work is being done for each parallel step. As a result, the synchronization overhead difference is less noticeable compared to total execution time, thus possibly leading to slight performance degradation compared to other implementations with more sophisticated load balancing or barrier implementations (like OpenMP or pthreads).

3.4 Parallel Breadth-First Search

We also tested the performance of the Breadth-First Search (BFS) algorithm. Results showed strong performance by the T-holder-based BFS across various graph sizes from 10 to 10^9 nodes. Notably, T-holder demonstrated significant advantages with considerably lower execution times compared to both pthreads- and OpenMP-based parallel implementations. Interestingly, other parallelism implementations, specifically pthreads and OpenMP, showed increased execution times even compared to the serial implementation of BFS. Only the T-holder implementation consistently achieved faster execution times than the serial version. This behavior is likely due to significant overhead from thread management and synchronization in pthreads and OpenMP, whereas the T-holder's thread reuse strategy tries to minimize such overhead.

In contrast, the overhead associated with thread creation and deletion in other baseline parallel implementations outweighed the performance benefits gained from parallelization, leading to longer execution times than the serial implementation. These findings indicate that while traditional parallel implementations like pthreads and OpenMP serve as relevant baselines, their overhead limits their effectiveness for certain parallel algorithms. This highlights the advantage of T-holder's more efficient thread handling approach since the lower overhead of T-holder seems to be the main reason that this algorithm is finally able to beat its serial counterpart.

3.5 Parallel PageRank

Our final algorithm that we tested was the PageRank algorithm. Interestingly, it was noticed that the T-holder version of PageRank outperforms the pthread version of PageRank in the smaller and larger matrix sizes, but underperformed in the middle sizes. We suspect that this is due to the timing of the parallel and serial sections, as well as the number of parallel sections that were running. As we increase the size of the matrix, the amount of time spent in the parallel sections increases as well, which leads to the creation and destruction time of threads to be much less significant during runtime. This is why small matrix sizes can see a significant increase in performance under T-holder, while the performance improvements becomes less prominent as we increase the size of the matrix.

However, when we get to very large matrix sizes, we notice a significant improvement in performance with the T-holder library, as well as a significant increase in the amount of time needed to run (compared to running with smaller matrix sizes). This is likely due to an increase in the amount of parallel sections that are needed in order to run the threshold. It can be observed that the amount of parallel sections that are required to run the matrix size of 50,000 is significantly increased. This means that there are a lot more times that this algorithm under the pthread implementation would create and destroy threads, and thus providing more opportunities for T-holder version to save time by keeping the threads alive in the thread pool during the parallelized execution of the PageRank algorithm.

References

[1] 2016. Parallel Pagerank using Pthreads. https://github.com/nikos912000/parallel-pagerank/tree/master.

[2] Egon Elbre. 2018. A Tale of BFS: Going Parallel. https://medium.com/@egonelbre/a-tale-of-bfs-going-parallel-cdca89b9b295.

[3] ROLLAND HE. [n. d.]. PARADIS: A PARALLEL IN-PLACE RADIX SORT ALGORITHM. https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/he.pdf.

[4] Luis Miguel Pinho. 2023. Real-Time Parallel Programming: State of Play and Open Issues. *arXiv preprint arXiv:2303.11018* (2023).

[5] Reza Zadeh, Matriod and Stanford 2016. CME 323: Distributed Algorithms and Optimization. https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture04/cme323_lec4.pdf.