T-holder:
A Custom Thread-Holding Library

Mingun Cho, Henry Chou, Piyush Kumbhare, Qin Liu, Aaron Nguyen, Kevin Nhu

UCDAVIS

Problem Statement & Context

- Parallel computing is essential for modern high-performance computing.
    - Divide and execute tasks simultaneously
    - pthreads, OpenMP, std::thread (C++)
- In parallel computing, there exists a performance overhead of spawning threads.
    - Memory allocation of the child thread's stack, stack/frame pointers, and scheduling

Problem (cont.)

- Amortize the thread spawning cost by making threads to spend most of their time working.
    - Pay the thread spawning cost only once.
- **Problem: What if an algorithm requires a series of parallel and sequential sections?**
    - Pay the thread spawning costs multiple times → Performance degradation

Design Goals

- Eliminate paying the cost of creating a thread twice
  - If a thread is idle and available to use, just tell *it* to run the new task instead of spawning a new thread
- Avoid fixed thread pool sizes
  - Unlike in fixed thread pool implementations, user should not be limited by the size of the pool, and should instead be able to call as many threads in a non-blocking manner
- Easy to implement for existing programs that use pthreads
  - The library API should remain virtually unchanged to make transitioning easy

Our Implementation

- tholder library (stands for p**T**hread **Holder**)
  - **Dynamic thread pool**
    - Doubles if size exceeds capacity
    - Memory is only allocated if a thread occupies a slot (only pay for what you use!)
    - Inactive threads will wait a small amount of time before exiting
  - ***pthread_create*** → ***tholder_create***
    - Searches the thread pool for an inactive index
    - If a thread is alive and idle, signal it to wake up and run the new task
    - Else spawn a thread using ***pthread_create***
  - ***pthread_join*** → ***tholder_join***
    - Instead of waiting for the thread to exit, waits on a region of memory containing the result of the task
    - Blocks until signaled by the auxiliary thread that completed the task

Results

Simple Test

- C program that simply spawns & joins 1000 threads 100 times
- Run with ***strace -c*** to view syscall summary
- Demonstrates ***tholder***'s ability to scale well when compared to ***pthread***

pthread:

% time	seconds	usecs/call	calls	errors	syscall
47.64	0.829015	8	100000		clone3
15.80	0.275047	1	200001		rt_sigprocmask
15.58	0.271121	2	99601		munmap
10.69	0.186037	1	99608		mprotect
10.26	0.178539	1	99612		mmap

Time spent making syscalls: **1.740282 sec**

tholder:

% time	seconds	usecs/call	calls	errors	syscall
98.89	0.261866	1	199558	1	futex
0.59	0.001550	12	126		clone3
0.34	0.000897	3	253		rt_sigprocmask
0.09	0.000234	2	100		write
0.06	0.000159	9	17		mprotect
0.03	0.000084	4	21		mmap

Time spent making syscalls: **0.264795 sec**

(Note: Each thread in this test does negligible work)

Target Algorithms

Recall the Problem:

What if an algorithm requires a series of parallel and sequential sections?

- Pay the thread spawning costs multiple times → Performance degradation
- With ***tholder***, we should expect to see less threads being unnecessarily created/destroyed and instead being reused

Target Algorithms

- Parallel Merge Sort
  - Iterative version merges adjacent subarrays via a preallocated temporary buffer, subarrays are sorted per thread in increasing sizes
  - Recursive version divides array into two halves recursively, two threads are spawned at each recursive level to concurrently sort them
- Parallel Radix Sort
  - Iteratively compute a new index of each number based on binary representation of each digit.
- Parallel Cholesky Factorization
  - Decomposes symmetric, positive-definite matrix into product of lower triangular matrix and its transpose.
  - Iteratively compute elements of lower triangular matrix (instead of full Gaussian elimination).
- Parallel Breadth First Search
  - Parallel BFS distributes workload of exploring nodes at each level among multiple threads for simultaneous frontier expansion
  - Involves partitioning current level's nodes, synchronizing shared data, and load-balancing to ensure efficient exploration of the graph
- Parallel PageRank
  - Iteratively finds ranking of “pages” (nodes) using matrix of node connections
  - Parallelism occurs during matrix-vector multiplication

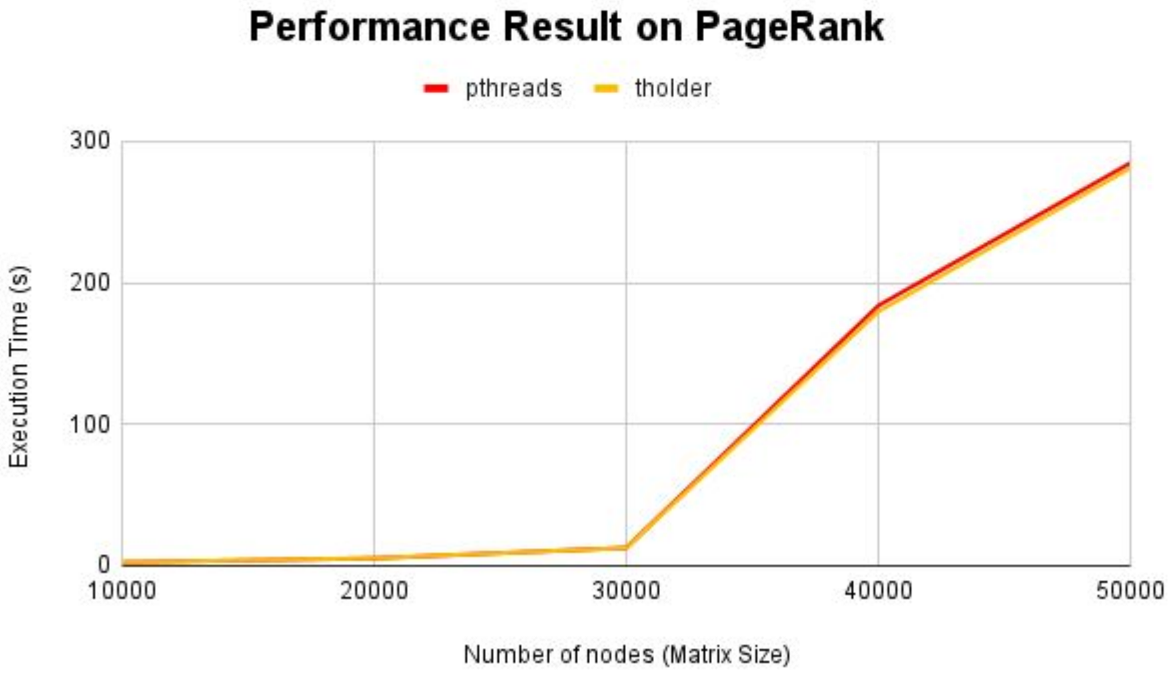
Results

- Evaluated on UCD CSIF computer
  - CPU: i7-9700 @ 3.00 GHz 8 Cores (8 Threads)
    - Average over 10 runs

N (E+06)	sequential	pthreads	tholder
1	0.036904	0.040301	0.031536
5	0.239912	0.209245	0.192538
10	0.509961	0.431088	0.39524
50	2.791394	1.999022	1.899493
100	5.781306	4.085281	3.926801
500	30.923298	22.148238	20.67424
1000	63.970296	45.856447	42.831539

Results

- Evaluated on UCD CSIF computer
  - CPU: i7-9700 @ 3.00 GHz 8 Cores (8 Threads)
    - Average over 10 runs



N	serial	parallel	tholder
10000	10.736194	2.450111	2.469506
20000	39.384124	5.272352	5.391035
30000	89.906582	12.638849	12.5506
40000	1218.560219	183.799351	179.550875
50000	1888.929354	284.561385	281.002061

Future Work

- Known Issues
  - Unable to call ***tholder_create*** in a recursive fashion
  - ***pthread*** attributes cannot be propagated to threads created by ***tholder***
- Potential Improvements
  - Reduce the number of ***futex*** syscalls
    - Would require a redesign of much of the library, but can greatly reduce the time spent
  - Thread timeout duration (currently 1 ms)
    - What value works best overall?
    - Is it possible to dynamically change this value at runtime based on analyzing time taken to create/destroy ***pthreads***?

Thanks for listening

https://github.com/MangoShip/ECS251Project