Due: See website for due date. (Late days may be used.)

What to submit: Upload a tar ball using the p2 identifier that includes the following files:
- `partner.json` with the SLO IDs in the format described for Project 1.
- `threadpool.c` with your code.
- `threadpool.pdf` with your project description. Use a suitable word processing program to produce the PDF file.

We will be using the provided `fjdriver.py` file to test your code. Please see Section 3.5 for more information.

1 Background

The last 2 decades have seen the widespread use of processors that support multiple cores that can act as independent CPUs. Today, even processors used in smartphones contain 4 or more cores. Software has been slow to catch up, despite calls for programming models that make it easy to write scalable programs for multicore systems [1].

As a case study, consider the `std::async` function that is part of the C++11 standard.[1]. The reference documentation on cppreference.com provides the example shown in Figure 1.

This toy example sums up the elements of a vector, which here are initialized to 1, using a recursive divide-and-conquer approach. At each level of recursion, the array is subdivided into two equal parts, one of which is passed to `std::async` to be executed in a separate thread, whereas the other part is recursively performed by the calling thread. `std::async` returns a handle of type `std::future`, which represents a reference to a result of a computation that is executed asynchronously. When the computation's result is needed, a thread may invoke the future's `get()` method. `get()` will return the result, arranging for—or waiting for—its computation as necessary.

[1] You will not need to learn C++ for this project, I am just using it as a motivating example

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
    auto len = std::distance(beg, end);
    if(len < 1000)
        return std::accumulate(beg, end, 0);

    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(100000000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end())
              << '\n';
}
```

Figure 1: A parallel sum implementation in C++11. This is a slightly modified version of the example published at http://en.cppreference.com/w/cpp/thread/async. Instead of 10,000, this program is summing up a vector with 100,000,000 elements.

Compiling and running this program under g++ 8.5.0 one obtains the following output:

```
$ g++ -pthread -O2 cppasyncpsum.cc -o cppasyncpsum
$ ./cppasyncpsum
terminate called after throwing an instance of 'std::system_error'
  what():  Resource temporarily unavailable
Aborted (core dumped)
```

The reason for this failure is that C++11's `std::async` is implemented by blindly spawning kernel-level threads (roughly 10^5 of them), without any regard to the amount of resources used by those threads.[2]

This example motivates the need for frameworks that do better than spawning one thread for each parallel task.

In this project, you will create a small fork/join framework that allows the parallel execution of divide-and-conquer algorithms such as the one shown in the example in Figure 1 in a resource-efficient manner. To that end, you will create a thread pool implementation for dynamic task parallelism, focusing on the execution of so-called fork/join tasks. Your

[2] C++23 may include support for executors as P044R14 was adopted in 2019.

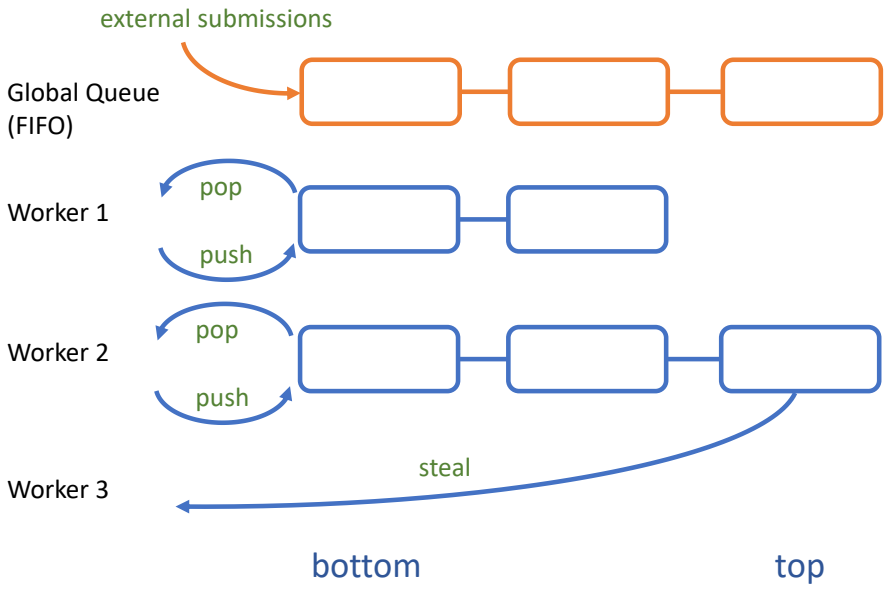


Figure 2: A work stealing thread pool. Worker threads execute tasks by popping them from the bottom of their deques. If they run out of work, they first attempt to dequeue tasks from a global submission queue. Failing that, they attempt to steal tasks from the top of other workers' deques. New tasks may be submitted externally to the global queue, but tasks spawned during the execution of a task are pushed onto the bottom of executing workers' deques.

implementation should avoid excessive resource use in order to avoid crashes like the one seen in this example.

2 Thread Pools and Futures

Your fork-join thread pool should implement the following API:

```
/**
 * threadpool.h
 *
 * A work-stealing, fork-join thread pool.
 */

/*
 * Opaque forward declarations. The actual definitions of these
 * types will be local to your threadpool.c implementation.
 */
struct thread_pool;
struct future;

/* Create a new thread pool with no more than n threads. */
struct thread_pool * thread_pool_new(int nthreads);
```

```
/*
 * Shutdown this thread pool in an orderly fashion.
 * Tasks that have been submitted but not executed may or
 * may not be executed.
 *
 * Deallocate the thread pool object before returning.
 */
void thread_pool_shutdown_and_destroy(struct thread_pool *);

/* A function pointer representing a 'fork/join' task.
 * Tasks are represented as a function pointer to a
 * function.
 * 'pool' - the thread pool instance in which this task
 *          executes
 * 'data' - a pointer to the data provided in thread_pool_submit
 *
 * Returns the result of its computation.
 */
typedef void * (* fork_join_task_t) (struct thread_pool *pool, void * data);

/*
 * Submit a fork join task to the thread pool and return a
 * future.  The returned future can be used in future_get()
 * to obtain the result.
 * 'pool' - the pool to which to submit
 * 'task' - the task to be submitted.
 * 'data' - data to be passed to the task's function
 *
 * Returns a future representing this computation.
 */
struct future * thread_pool_submit(
        struct thread_pool *pool,
        fork_join_task_t task,
        void * data);

/* Make sure that the thread pool has completed the execution
 * of the fork join task this future represents.
 *
 * Returns the value returned by this task.
 */
void * future_get(struct future *);

/* Deallocate this future.  Must be called after future_get() */
void future_free(struct future *);
```

2.1 Work Stealing

There are at least two common ways in which multiple threads can share the execution of dynamically created tasks: work sharing and work stealing. In a work sharing approach, tasks are submitted to a single, central queue from which all threads remove tasks. The

drawback of this approach is that this central queue can become a point of contention, particularly for applications that create many small tasks.

Instead, a work stealing approach is recommended [2] which has been shown to (at least potentially) lead to better load balancing and lower synchronization requirements. In a work stealing pool, each worker thread maintains its own local queue of tasks, as shown in Figure 2. Each queue is a double-ended queue (deque) which allows insertion and removal from both the top and the bottom. When a task run by a worker spawns a new task, it is added (pushed) to the bottom of that worker's queue. Workers execute tasks by popping them from the bottom, thus following a LIFO order. If a worker runs out of tasks, it checks a global submission queue for tasks. If a task can be found in it, it is executed. Otherwise, the worker attempts to steal tasks to work on from the top of other threads' queues.

In this assignment, you are asked to implement a work stealing thread pool. Since work stealing is purely a performance optimization, you may for reduced credit (corresponding to a B letter grade) implement a work sharing approach.

2.2 Helping

A naive attempt at implementing `future_get` would have the calling thread block if the task associated with that future has not yet been computed. "Blocking" here means to wait on a synchronization device such as a semaphore until it is signaled by the thread computing the future. However, this approach risks thread starvation: if a worker thread blocks while attempting to call `future_get` it is easily possible for all worker threads to be blocked on futures, leading to a deadlock because no worker threads are available to compute the tasks on which the workers are blocked!

Instead, worker threads that attempt to resolve a future that has not yet been computed must help in its execution. If the future's task has not yet started executing, the worker should steal it and execute it itself. If it has started executing, the worker has two choices: it could wait for it to finish, or it could help by executing tasks spawned by the task being joined, hoping to speed up its completion.

For the purposes of this assignment, we assume a fully-strict model as defined in [2]. A fully-strict model requires that tasks join tasks they spawn — in other words, every call to submit a task has a matching call to `future_get()` within the same function invocation. In this sense, all tasks spawned by a task can be considered subtasks that need to complete before the task completes. All our tests will be fully strict computations, which encompass a wide range of parallel computations.

Restricting ourselves to fully-strict computation for this project simplifies helping because it is always safe for workers intending to help to steal any task as long as they steal from the top of any other worker's queue. Safety here refers to the absence of execution deadlock.

Note that in a fully-strict model, in combination with helping, worker threads will never be in a situation where they are looking for tasks on their own queue: this is because any subtask spawned from a task they were working on will be joined before that task returns. In this situation, the worker will either directly execute that task via helping, or the task will have been stolen by some other worker. In no case will a worker return from executing a task and find other, unfinished tasks on its queue. (Recall that all tasks added to a worker thread's queue are subtasks of a task the worker was executing at the time.)

2.3 Implementation

Except for constraints imposed by the API and resource availability, you have complete freedom in how to implement your thread pool. Numerous strategies for stealing, helping, blocking, and signaling are possible, each with different trade-offs.

You will need to design a synchronization strategy to protect the data structures you use, such as flags representing the execution state of each tasks, the local queues, and the global submission queue, and possibly others. You will need a signaling strategy so that worker threads learn about the availability of tasks in the global queue or in other threads' queues.

2.4 Basic Strategy

A basic strategy would be to use locks, condition variables, and the provided list implementation (known to you from prior projects), which allows constant-time insertion and removal of list elements and which can be used to implement a deque.

You will have to define private structures `struct future` and `struct thread_pool` in `threadpool.c`. A future should store a pointer to the function to be called, any data to be passed to that function, as well as the result (when available). You will have to define appropriate variables to record the state of a future, such as whether its execution has started, is in progress, or has completed, as well as which queue the future is in to keep track of stealing.

A thread pool should keep track of a global submission queue, as well as of the worker threads it has started. In the work stealing approach, each worker thread requires its own queue. You will also need a flag to denote when the thread pool is shutting down.

You will need to create a static function that performs the core work of each worker thread. You will pass this function to `pthread_create()`, along with an argument (such as a pointer to a preallocated struct) allowing the thread to identify its position in the worker pool and to obtain a reference to the pool itself.

thread_pool_submit(). You should allocate a new future in this function and submit it to the pool. Since the same API is used for external submissions (from threads that are not part of the pool) and internal submissions (from threads that are part of the pool),

you will need to use a thread-local variable to distinguish those cases. The thread local variable could be used to quickly look up the information pertaining to the submitting worker thread for internal submissions.

future_get(). The calling thread may have to help in completing the future being joined, as described in Section 2.2. Helping is required for both work sharing and work stealing approaches.

thread_pool_shutdown_and_destroy(). This function will shut down the thread pool. Already executing futures must complete; queued futures may or may not complete.

The calling thread must join all worker threads before returning. Do not use `pthread_cancel()` because this function does not ensure that currently executing futures run to completion; instead, use a flag and an appropriate signaling strategy.

Upon destruction, a threadpool should deallocate all memory that was allocated on behalf of the worker threads.

future_free(). Frees the memory for a future instance allocated in `thread_pool_submit()`. This function is called by the client. Do not call it in your thread pool implementation.

Hints: A key challenge in this project is to ensure that updates to the state of a future are done atomically with respect to the presence (or absence) of this future in its respective queue (global or per-worker, depending on approach). You must avoid a situation in which a worker thread scanning the global queue or a peer worker's local queue "sees" a future in said queue, is about to steal it, while the worker executing that task's parent task attempts to join it and execute it via the helping path. Only one thread must succeed in executing the task – if the thread stealing the task executes it, the helping thread must either wait or engage in helping the executor. If the helping thread executes it, the thread attempting to steal must act as if the task had not been in the queue. In particular, if the thread helping wins this race, the future may be completed (and immediately after, deallocated via the `future_free()`), so any pointers obtained by the worker thread may no longer be valid if you allowed this situation to occur.

A recommended approach is to maintain the invariant that only tasks that are available for execution be maintained in any queue/list. Moving a task from the "new" to the "being executed" state should be atomic with respect to the removal of this task from the queue in which it is contained. Recall that `list_remove()` modifies a linked list and need to be protected by the same mechanism as other operations on that list.

2.5 Advanced Strategies/Extra Credit

Real-world fork/join implementations employ a number of optimizations designed to minimize per-task synchronization overhead. For instance, a crucial optimization is to speed up the common case of adding an element to or removing it from the bottom of

a worker's queue. This optimization is possible because only the current worker may add tasks to the queue, and only the current worker removes task from the bottom of its queue. Stealers remove tasks from the top of the queue.

The first optimized implementations used the THE protocol, inspired by Djikstra's mutual exclusion algorithm [4], which is further described in [5] and [6]. Chase and Lev presented a version of a work-stealing deque in [3], but their paper contains a number of errors. A corrected version using C11 atomics is presented by Lê et al. [7], whose code you may reuse for this project. [3]

A second possible extension would be to support computations that are not fully-strict, but still recursive. We define "recursive" such that it is possible to execute them on a single worker thread using helping, even though they do not meet the definition of being fully strict, perhaps because futures are passed among tasks before being joined.

If the computation is not fully-strict but still recursive, a deadlock situation could arise where one worker steals a task that, in order to complete, requires the results of a task whose execution has been started by the stealing thread, but not yet finished. Systems such as CILK [5] avoid this by using a technique known as continuation stealing [8] in which it is possible for other worker threads to continue (and complete) a spawning task. However, continuation stealing requires compiler support since another thread would need access to the task's local variables. [4] Systems that exploit child stealing, such as the thread pool you are building in this assignment, have to impose constraints on stealing for non-strict computations. A technique such as leap frogging [9] could be used, which keeps track of the depth of each task in the computation graph and provides a rule that allows or disallow stealing.

A third possible extension would be to support fully general acyclic computational graphs. (The assumption of being acyclic is necessary because graphs with cyclic dependencies are impossible to schedule.) Note that the given API is not suited for arbitrary dependency DAGs in the presence of child stealing. To support arbitrary DAGs in the absence of continuation stealing, an inverted control flow model must be used, perhaps similar to that used in Java's CountedCompleter classes. In other words, such tasks are not joined, but rather a callback will be executed once they complete, allowing dependent tasks to be scheduled.

If you implement any of these strategies, be sure to discuss it in your project description so that TAs may award extra credit if warranted.

[3] Warning: these implementations are designed for more general frameworks, they do not represent something you can simply drop in. Your implementation still must support global submissions, and it must reclaim all memory it uses. This is not shown in the paper. Another key difference is that this implementation does not support removal of a future from the middle of a queue; you will need to think about how this would affect the case where `future_get` would attempt to execute a task still on a queue.

Lastly, keep in mind that if you start adding C11 atomics, you'll lose the ability to check for race conditions with Helgrind/DRD because those tools do not understand the semantics implied by C11 atomics.

[4] Read Robison's Primer [URL] to learn more about child vs continuation stealing.

3 Additional Notes

3.1 Semaphores vs Condition Variables

We do not recommend that you use semaphores to signal your worker threads regarding the availability of new tasks, because semaphores do not perform well in the presence of large numbers of signals. Recall that signaling a semaphore entails incrementing its internal count, which requires an atomic write operation on its internal state. In the context of cache-coherent multiprocessors, this causes a transition into the "modified" state in the accessing core's cache, which causes a frequently updated semaphore to ping pong between caches. By contrast, condition variables are implemented in a way that handles the common case of signaling with no waiter present using the "shared" state - the condition variable records if any waiters are present and does not require updating state when a call to `pthread_cond_signal` does not actually signal (unblock) any threads. In addition, we recommend that you intentionally break the rule of signaling with the lock held (which Helgrind otherwise warns you about) for this case, while still making sure that your threadpool makes progress eventually.

You may still find semaphores useful, potentially, to implement waiting on individual tasks.

3.2 Avoiding False Sharing

As you tune the performance of your implementation, be on the outlook for false sharing. False sharing occurs when per-thread data structures are allocated within the same cache line, which may happen, for instance, for neighboring array elements. A potential mistake is to have a contiguous array of per-thread (per-worker) structures which are not meant to be shared but allocated closely together. To avoid this, add padding to ensure they are allocated in different cache lines. A cache line is 64 bytes long on our machines.

3.3 Grading

Grading will be based on a combination of factors, including

- Complexity. As noted above, you may either pursue a work stealing approach or, for a simpler implementation, a work sharing approach.
- Correctness. We expect your code to produce the correct result. Since you are writing a concurrent program, the results may vary between runs if your implementation is incorrect; we will run your code multiple times and expect it to complete correctly each time we run it. You should perform similar stress testing.

  We also expect your code to be correct when we restrict the number of threads in the pool to be 1, which requires a correct implementation of helping.

- Thread Safety. Your code must not contain race conditions. You should run the code using the Helgrind race condition checker. If Helgrind flags any warnings, you should address them. If you believe Helgrind's warnings are spurious because you are making use of advanced synchronization facilities or atomic variables that trigger false positives, provide a rigorous proof.
- Speedup. For some of the benchmarks we provide, we will measure the speedup obtained using your thread pool. The `fjdriver.py` script will compile your thread pool, link it with our tests, and benchmark it. It will then prepare a file you may upload to the scoreboard (via `fjpostresults.py`) to compare your results to those of others. For development, we recommend that you keep your `threadpool.c` file inside the `tests/` subfolder of your git repository – this way, you can build and test with `make` until you are ready to use the test driver.
- Memory Reclamation. Multi-threaded programs are particularly prone to use-after-free errors when one thread still holds a reference to an allocated block another thread concurrently frees. For this reason, we will test that your threadpool deallocates all memory when it is destroyed.

The scoreboards are unofficial in that your final grade will be determined when the TAs check and benchmark your code. However, we will use the scoreboard as a yardstick to determine high-performing and low-performing implementations. In particular, if you see that for a particular test some implementations provide speedup that is a multiple of what your implementation provides, you may conclude that your implementation may impose unnecessary serialization or have other bottleneck factors you should try to address.

For grading, we will award credit for

- **Meeting Minimum Requirements,** which for this project include a working thread pool implementation that can execute a specific set of parallel programs correctly. `fjdriver.py` will flag whether you have met minimum requirements, but keep in mind that during grading, we will run the required tests multiple times and expect them to pass every time.
- **Robustness,** as measured by the ability to successfully and reliably complete a number of more complex applications within a test-specific timeout.
- **Performance,** as measured by the speedup obtained for more complex applications/tests. **Note:** to obtain a performance score, you must have met minimum functionality requirements since there is little point in investigating the performance of non-functional or buggy code. Before the project deadline, we will publish the specific performance requirements, which depend on the current semester's hardware and software environment; see scoreboard above.

3.4 Honor Code

As usual, all work submitted must be yours and be created from scratch by all group members this semester. You may not reuse code from any implementations you may find online without the instructor's permission (and the permission of the author, if applicable). If in doubt, you must ask. Otherwise, the collaboration policy described in the syllabus applies.

3.5 Running Experiments

We will use the machines of the rlogin cluster for testing, so make sure your code runs there when invoking `fjdriver.py`. Starting with the Summer 2020 semester, these are dual-socket Intel Xeon CPUs with 16 cores each, providing 32 cores total, which are presented to the OS as 64 CPUs (2 hyperthreads per core). Since hyperthreads typically do not add significant speedup, if any, for CPU-bound tasks, we will not run with more than 32 threads.

Perform these experiments on an unloaded machine on the rlogin cluster. Unloaded means that 'uptime' should report a load average close to 0, so that all processors are available for your experiment. Coordinate with other students by avoiding running your benchmarks if you notice that other students are running theirs; use the forum or email if necessary. `fjdriver.py` will output a message and wait if run on a machine with a non-zero load average.

3.6 Additional Requirements.

- The use of git.cs.vt.edu is required as in project 1.
- The upstream repository is `https://git.cs.vt.edu/cs3214-staff/threadlab`
- After forking the repository, be sure to set access to private. Not doing so is a potential honor code violation.
- All code for this project must be contained in `threadpool.c`, mainly to simplify testing. We also believe that the complexity of this assignment, at least in its basic form, should not necessitate the use of multiple source files.
- Do not change any of the other files! (If you do, such changes will not be taken into account when grading and you may you fail the grading process.)
- Your code must compile without warnings. The Makefile enforces this via `-Werror`.

- You should not define any global variables that become global symbols, and you should not need to define any static variables with the exception of C11 `_Thread_local` thread-local variables.
- You should not define any extern global functions other than the ones asked for - use static functions as necessary.
- The submission check script may impose additional requirements to simplify automatic grading. Please work with teaching staff on any questions you encounter.
- Updates to these requirements may be posted on the website or the forum (in a 'pinned' post at the top of the Forum board).

Good Luck!

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[3] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, 2005.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, 1998.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[7] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, 2013.

[8] Arch Robison. A primer on scheduling fork-join parallelism with work stealing, 2014.

[9] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fourth ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM.