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

What to submit: The project will be submitted in multiple stages. You should submit a tarball for each stage.

Upload a tar ball using that includes the following files:
- `partner.json` with the SLO IDs in the format described for Project 1.
- `git.json` with the URL of the git repository as in Project 1.
- `threadpool.c` with your code.
- (Last checkpoint only) `threadpool.pdf` or `threadpool.md` 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 4.8 for more information.

1 Background

The last two 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].

In this project, you will create a small fork/join framework that allows the parallel execution of divide-and-conquer algorithms in a resource-efficient manner. To that end, you will create a thread pool implementation for dynamic task parallelism, first using work sharing, then using work stealing.

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.
 * If any of the threads cannot be created, print
 * an error message and return NULL. */
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 Sharing vs 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,

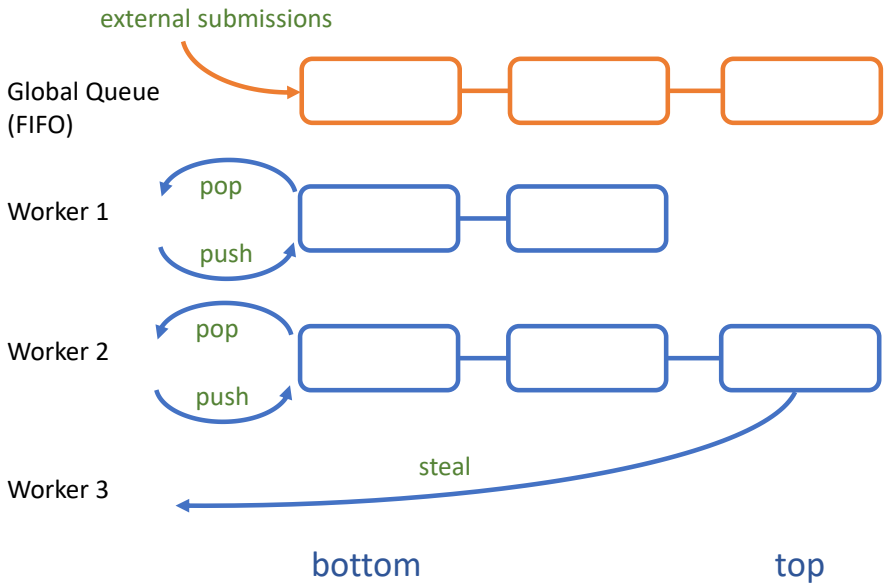

Figure 1: A work stealing thread pool. Worker threads execute tasks by popping them from the bottom of their deques. If they run out of work, they first attempt to dequeue tasks from a global submission queue. Failing that, they attempt to steal tasks from the top of other workers' deques. New tasks 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.

particularly for applications that create many small tasks. The advantage is its relative simplicity.

Work stealing approaches [2] have been shown to 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 1. 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 first a work sharing and then a work stealing thread pool.

2.2 Handling Dependent Tasks

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 condition variable or 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 must steal it and execute it itself.[1] 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.

Note that we have used the words "help" or "helping" in two contexts now: first, helping when an not-yet-started task is being joined. This mode of helping, also referred to as inline execution, is mandatory in order to avoid deadlock. Here, a thread "helps" themselves because no one else is around to take on the task they've spawned.

Second, we have said that a worker that is trying to join a task that's already in progress because it was stolen by another thread may help by taking on tasks that the "thief" needs to complete in order to complete the stolen task. This mode of "helping the thief" represents a policy decision that can improve performance, but it is not mandatory for a functioning thread pool. [2]

Additional explanation on helping is provided in Appendix B.

2.3 Additional Requirements

These apply to both work sharing and work stealing.

External threads must not help. Only worker threads should help when they encounter not-yet-finished tasks. Do not let external threads help with tasks - these threads should wait for a worker to complete a submitted task. This way, it becomes easier to tune the pool's concurrency level to match the number of physical processors or core dedicated to the threadpool. Use a thread-local variable to distinguish between the two. This can be a boolean in a work-sharing approach, and it could be a pointer to a per-worker data structure later when implementing work stealing.

Thread pools should be good citizens. When no tasks have been submitted, your thread pool should not consume significant CPU resources — rather, all worker threads should be in the `BLOCKED` state.

[1] Note that this applies to both work sharing and work stealing.

[2] Therefore, you should consider implementing it only after you have a working implementation; and this applies only to your work-stealing implementation in which the concept of thieves exists.

3 Implementation

When implementing the final stage, you have complete freedom in how to implement your thread pool, except for constraints imposed by the API and resource availability. Numerous strategies for stealing, helping, blocking, and signaling are possible, each with different trade-offs. That said, we will provide a scaffolding you should follow.

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.

3.1 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. For work-sharing, a single queue suffices. 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 when implementing work-stealing. The thread local variable could be used to quickly look up the information pertaining to the submitting worker thread for internal submissions. No such distinction is necessary when implementing work-sharing.

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. You will need to distinguish worker and non-workers from stage 3 on up since non-workers must not help.

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 [3]

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.

4 Implementation Stages

To provide scaffolding, we require that you implement this project in stages. At the end of each stage, you will submit your work.

4.1 Stage 1: A basic pool skeleton.

For stage 1, you should implement only the functionality of starting the prerequisite number of worker threads and then shutting them down. You will need to design a way to keep track of the threads. The worker threads themselves should not do any work; simply have them wait on a condition variable until the shutdown flag is true.

Don't implement `thread_pool_submit()` and `future_get()` yet.

At this point, you should pass only test 9; all other tests must fail.

4.2 Stage 2: A work-sharing, single-lock thread pool without helping.

Implement a work-sharing, single-lock thread pool. Add a single global queue to your threadpool as well as a single lock. This lock will synchronize all operations involving multiple threads. You will need to add a pointer to each future to find the lock protecting the pool.

Implement `thread_pool_submit` such that it allocates new tasks, initializes them, and then puts tasks into a single list associated with the threadpool. In `future_get`, always block on the task if it's not done - do not implement helping. Use a condition variable for workers to wait for tasks or pool shutdown, and give each future a condition variable so you can wait for the task in `future_get`.

At this point, you will pass a limited number of tests, namely all those that do not require helping.

[3] None of our tests will shut down the pool while there are outstanding tasks, that is, all externally submitted tasks will have been joined.

```
./threadpool_test
./threadpool_test2
./threadpool_test3 -n 2
./threadpool_test4
./threadpool_test5
./threadpool_test6 -n 100
./threadpool_test7
./threadpool_test8 -n 50
./threadpool_test9 -n 50
./kenken3 -d 2 -n 100
```

At this stage, your thread pool resembles a basic thread pool that can be used for independent tasks.

4.3 Stage 3: Adding Helping.

Complete your single-lock, work-sharing implementation by adding helping, and specifically the first form of helping in which a worker thread calling `future_get` will execute tasks that have not been started.

You should respect the rule about external threads not helping. Do not implement helping in-progress tasks yet.

At this point, you should pass the tests listed under minimum requirements, but some of the tests will time out.

4.4 Stage 4: Adding Work Stealing (Single-Lock).

Now add per-worker queues and the work-stealing mechanism. Tasks that worker submit should be queued on their local queues, whereas tasks that external threads submit should be queue on the global queue. Workers should first service the global queue and if empty, attempt to steal from other worker's queues. (The case where a worker will find tasks on its own queue will not occur for our fully-strict workloads.)

Your implementation at this point should still use a single lock. This will not increase the number of tests you pass yet.

4.5 Stage 5: Fully-fledged Work Stealing + Performance Optimization

In the final stage, break the global lock and use a finer-grained synchronization strategy. See Section A.1 for a recommend locking strategy that uses per-worker rather than per-future locks.

You can now also implement helping thiefs to investigate if this strategy leads to further speedup, as well as test out and investigate other optimizations as per the optimization guide C.

4.6 Grading

Grading will be based on a combination of factors, including

- 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 before submitting.
- Thread Safety. Your code must not contain race conditions. You should run the code under 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.
- Resource Use. Outside of dedicated high-performance computing (HPC) fork-join implementations are required to coexist with other code and share computational resources. One of our tests ensures that your pool does not consume resources when idle.

The scoreboards are unofficial in that your final score for stage 5 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

- **Correctness** `fjdriver.py` will flag whether you have passed the basic tests, 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 num-

ber 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 for stage 5, which depend on the current semester's hardware and software environment; see scoreboard above.

4.7 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. The use of generative AI for code is permitted.

4.8 Running Experiments

We will use the machines of the rlogin cluster for testing, so make sure your code runs there when invoking `fjdriver.py`. These are currently 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.

4.9 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.
- To facilitate the automated grading of your git usage, please follow the following rules:
  - Do not rename the repo when you fork it.

  - Do not create a git group; fork the repo under the namespace of one of the two group members.
  - Make sure that, once you have finished, your final product will be on the default master branch.
  - Make sure that the git commit log on this branch shows the contributions of both team partners under their CS pid.
  - You may use branches during development, but if you do, make sure to merge those branches. Don't squash your commits when you do so.
  - You must use git.cs.vt.edu and not any external git server.

- 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 thread-local variables using the appropriate C11, C14, or GNU-C conventions.
- 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.

Good Luck!

A Debugging and Implementation Hints.

There are many pitfalls in multithreaded programming that can result in bugs in your program, so it is important to know some debugging strategies.

- Do not run the test driver `fjdriver.py` every time to test your program. You can build your code in the 'tests' directory and directly run the tests until they work.
- Use the available data race detection tools: Helgrind and DRD. To use Helgrind, run your code with `valgrind --tool=helgrind ...` followed by the name of the executable you would normally run. If you are not properly waiting for work on your condition variable, it is possible that your program does not make progress under Helgrind. You can identify this case by changing Helgrind's internal scheduling policy–add the `--fair-sched=yes` switch to valgrind's options. If your program makes progress under these settings, look for a bug that causes it to busy-wait.
- If your program appears "stuck" (does not make progress), then this can be the case for multiple reasons. A common reason is deadlock where all threads are in the `BLOCKED` state because they are waiting to either acquire a lock or waiting to be signaled on a condition variable or semaphore. You can identify this case by observing zero CPU utilization and using the 'ps' command to check the (Linux) process state of your threads. If your program has deadlocked, then attach gdb (or start the program under gdb) and examine the backtraces of all threads. Linux also allows you to identify which thread holds a lock by examining its internals with gdb's print command. If your program has not deadlocked (or only some threads have), you can still use the same strategy to obtain information about each thread's progress.

Signaling Strategy. You need to solve multiple signaling problems in this project: idle worker threads must be signaled when either a global task is submitted or when a task becomes available for stealing. To that end, you should use a single condition variable. You can't use more than one because a worker can wait on only one condition variable at a time, but an idle worker needs to be woken up in both cases.

An additional signaling problem to solve involves when a thread can't help (the thief) and is blocked on an in-progress task. This thread will need to be signaled when said task has completed, which can be accomplished using a condition variable or semaphore associated with this task/future.

A.1 Per-future locks vs per-workqueue locks

Once you break the single lock in stage 5 into multiple locks, a key challenge will be 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 `future_free()`, so any pointers obtained by and stored in local variables of 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 are contained 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 thus needs to be protected by the same mechanism as other operations on that list.

A design that seems natural at first is to have locks for each future which protect a future's fields such as its state, and a lock for each worker that protects its fields such as the queue of tasks this worker currently maintains. The problem with this design, however, is that it makes it more difficult to avoid the atomicity violation described above. A worker thread attempting to steal a task would lock the queue first and then lock the future it sees there. Concurrently, a thread calling `future_get` would need to lock the future first, but it cannot commit to running the future's task unless it also has the lock that protects the queue. It would need to acquire both, and moreover, after acquiring the queue lock it would need to make sure that no worker has started the task. We do not recommend this design because

- it is complex and error prone
- requires dealing with the difference in locking order to avoid deadlock (first instance, by making the second lock acquisition a trylock with a suitable strategy on failure).
- does not yield the highest performance because it involves 2 lock acquisitions on the hot path that involves `future_get`.

Instead, we recommend a design where at least a future's state field is protected not by a lock associated with the future, but by the lock that protects the queue into which the future is inserted upon creation. Note that the lock protecting a future must be constant and cannot change throughout the life of a future, so this lock must be continued to be used even if the future is at some point stolen.[4]

B Fully-strict Workload Model

For the purposes of this assignment, we assume a fully-strict model as defined in [2]. A fully-strict model requires that tasks join all tasks they spawn — in other words, every

[4] As a side note, stealing a future will also not add it to the thief's queue. Idle workers will steal only one task at a time, and then continue to execute this task directly rather than adding it to their queue.

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 (even though your thread pool implementation will not need to keep track of which task is a subtask of which other task). 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 thiefs 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 that have just finished a task 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.)

C Optimization Guide

C.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 read-modify-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.

C.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.

C.3 Additional Optimizations and Extensions

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. [5]

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. [6] 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

[5] 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. Do not assume that tasks are joined in LIFO order.

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.

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

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.

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.