ECE465 Checkpoint 2b
Distributed Graph Coloring

Jonathan Lam & Henry Son

February 21, 2021

Since the diagrams were written with TikZ, it was easier (and prettier) to typeset everything in LaTeX than convert and embed all the visuals into the GitHub Markdown files. I'll probably end up all of the reports (i.e., everything but basic build instructions) over to LaTeX because it looks nicer and has a more academic feel.

1 Updates from Checkpoint 2a

The handshake was left mostly unchanged, as it was already working fine for the first part. The algorithm was barely functional for the first part of the assignment, so the main goal was to fix up these errors. This was two-fold: fixing a issue that involved short writes, and synchronizing the start of an iteration in the algorithm in a way that avoids a disastrous race condition.

Performance-wise, we did not have enough time to achieve a speedup over single-node application. We spent much time on debugging (enough to start jeopardizing other schoolwork) and were only barely able to finish making the application run reliably.

There was also some general cleanup of code and adding additional features, such as a quiet flag to suppress logging, and disabling printing of nodes (otherwise the 10000 node graph generates a 2.5GB logfile). More details on the fixes and performance issues follow the figures below.

2 Figures

See the following pages for details on the entire distributed algorithm. Note that the terms "client" and "worker" are used interchangeably and refer to all of the non-server nodes.

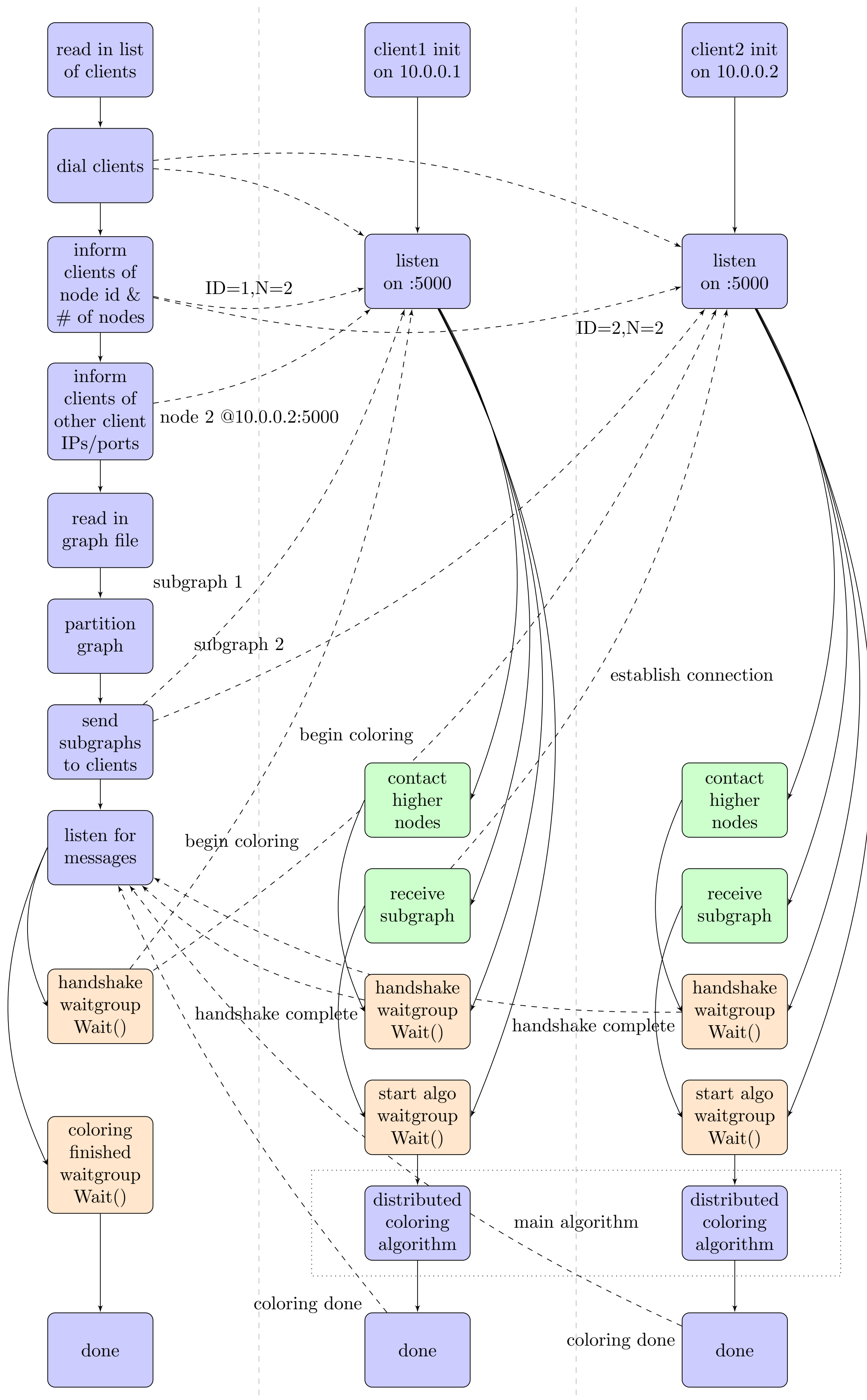


Figure 1: Communication protocol for the coloring algorithm for a two-worker network. Solid and dashed lines represent intra- and inter-node (socket) communication, respectiely. Blue blocks represents sequential code, green represents handlers to socket messages, and orange represents waitgroup (semaphore) action.

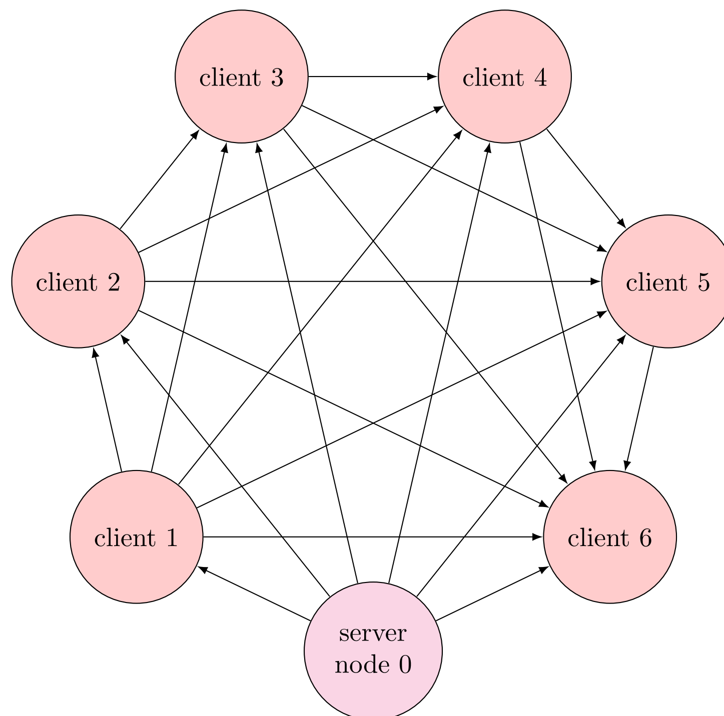


Figure 2: Illustration of the node dialing order for a six-worker network. The server ("node 0") dials each worker node, informing them of their node ID (1-6), total number of nodes (7), and the addresses (IP:port) of higher-indexed nodes. Each node then dials each of its higher-ordered nodes until a complete graph of peer-to-peer (TCP socket) connections is formed. (This process is relatively brief, so the imbalance is inconsequential.) Only then is the handshake complete.

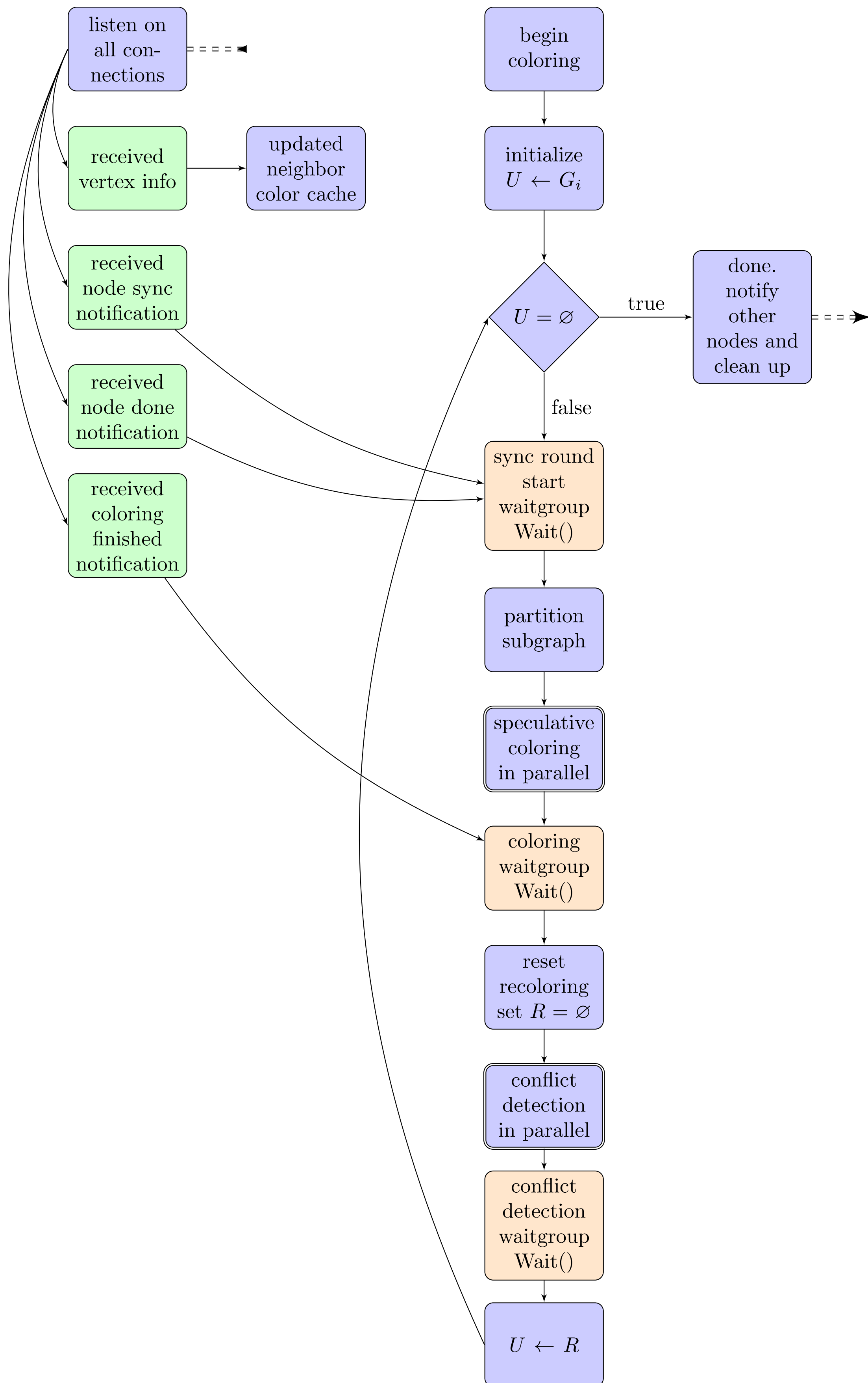
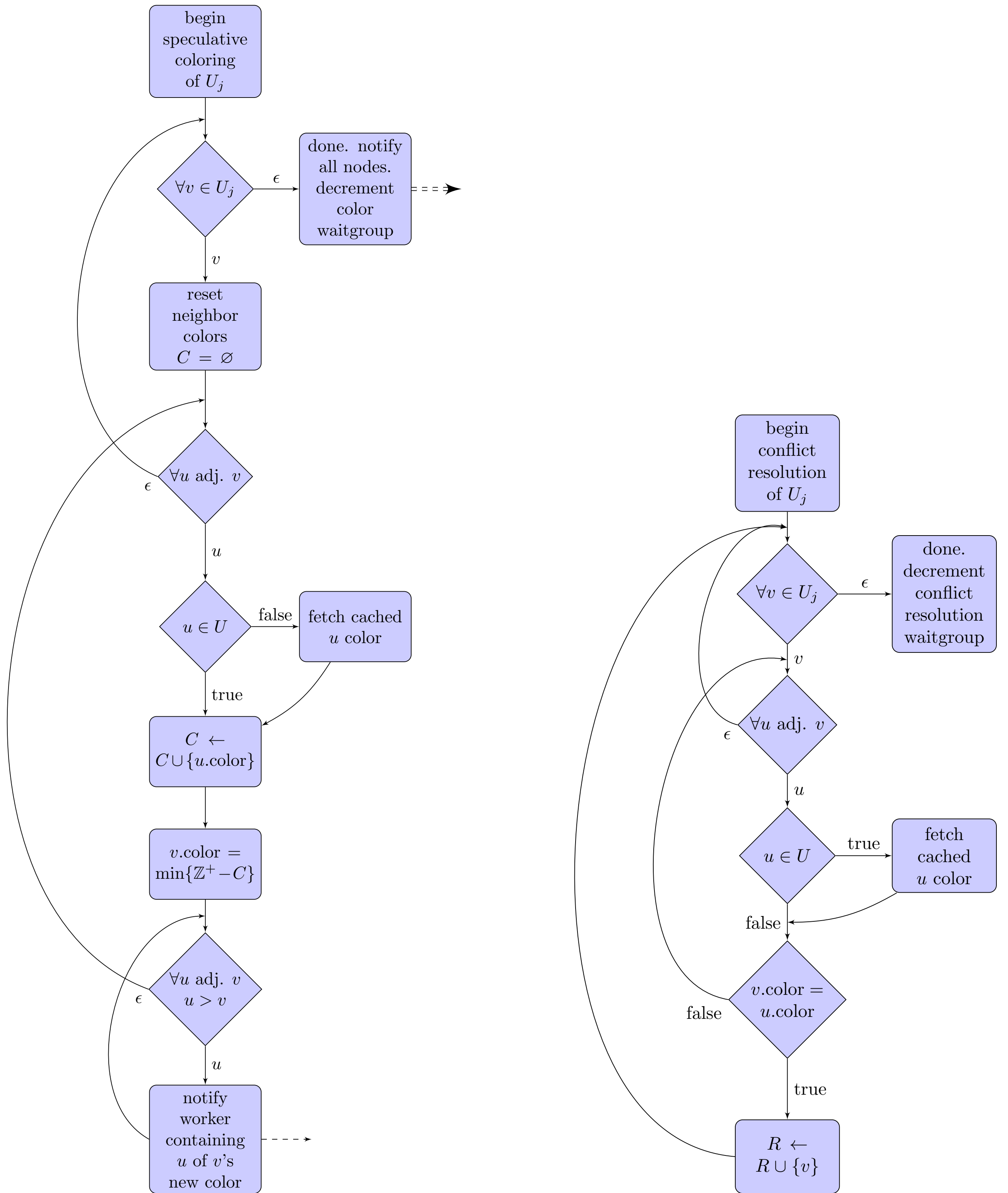


Figure 3: Detail of the "distributed coloring algorithm" block from Figure 1 for a single worker node, based on the general framework proposed by Gebremedhin et al. (2005). The double-dashed arrow represents the set of connections to all of the other nodes. The blocks that involve parallel processing are indicated by a double border. The listener (the same listener set up during the procedure) is in a separate goroutine and therefore will not block when the mainline is blocking on a semaphore.



(a) The speculative coloring algorithm for a single thread

(b) The conflict detection algorithm for a single thread

Figure 4: Detail of the speculative coloring and conflict detection algorithms in Figure 3. Note that the graph is first partitioned amongst the worker nodes $G = \bigcup G_i$. For a particular worker, U represents the nodes to be colored and is set to G_i initially. It is then partitioned again into $U = \bigcup U_j$, and thus we achieve a fairly high level of parallelization. This part of the algorithm is not much different from the single-node variant, except that we require fetching cached color values of neighbors that lie outside the current subgraph, and we have to notify worker nodes containing neighbors of nodes that have been recolored. To reduce network I/O, we arbitrarily only notify the higher-indexed neighbors when a node is colored (hence the $u > v$ condition).

3 Fixing concurrency issues

3.1 Short writes

This was a tricky issue for a number of reasons:

- It's not a common issue (it doesn't appear anywhere on an Internet search)
- The write buffer was often not full.
- The listener (reader) thread doesn't hang.
- The typical network socket buffer is large (16kB, according to `man 7 tcp`), and substantially larger than the default buffer size for `bufio.Writer` (4kB).

The only thing that did make sense was that a short write is a reasonable (?) response for a FIFO like a network socket or a pipe, but it must not be common on ordinary network sockets because an Internet issue doesn't turn it up.

Painstaking debugging revealed that the short writes were really very random and the buffer was not full at all, and it was due to concurrent calls to the `Write()` method of the underlying socket which returned 0. This is solved by making all writes to any particular network socket a critical section.

(Debugging first revealed the fact that a short write causes a `bufio.Writer` to reject *all future writes* and maintain its error until a call to the writer's `Reset()` method is made, which also discards the buffer. Only when messing around with this did the real culprit come up.)

3.2 Synchronizing the start of a coloring round

The code in Figure 5 shows the general distributed algorithm sequence, along with the synchronization steps. The problem with this code is that `colorWg` may be decremented before it is set. A possible series of events, assuming a two-worker system is: the first node (node1) begins a round, and broadcasts `ROUND_SYNC`, and then it sleeps on `startWg.Wait()`. The second node (node2) begins a round, broadcasts `ROUND_SYNC`, receives node1's `ROUND_SYNC` message (releasing `startWg`), and doesn't sleep on `startWg.Wait()` since the semaphore is released. It (node2) continues on its mainline thread: it sets `startWg` and `colorWg` to their appropriate values, performs its speculative coloring, broadcasts `SPEC_COLOR_DONE`, and sleeps on `colorWg.Wait()`. Then, node1's listener thread receives the `ROUND_SYNC` message, freeing the `startWg` semaphore (but this doesn't cause the mainline thread to immediately wake/get scheduled in), and it receives the `SPEC_COLOR_DONE` message, decrementing the `colorWg` semaphore to -1. This is due to the fact that node1's mainline thread still hasn't woken yet and set the `colorWg` semaphore appropriately, and the negative semaphore raises a panic.

```
colorWg.Add(0)
startWg.Add(nNodes - 2)
U := G_i
while U is not empty:

    // round sync
    broadcast ROUND_SYNC message
    startWg.Wait()

    // these have to happen after
    // startWg because we need to know
    // how many nodes are left
    startWg.Add(nNodes - 2)
    colorWg.Add(nNodes - 2)

    // perform speculative coloring ...
    speculativeColor()
    broadcast SPEC_COLOR_DONE message
    colorWg.Wait()

    // conflict detection
    R := []
    conflictDetection()
    U = R

// node completely done
broadcast NODE_DONE message
```

(a) Mainline thread

```
function handler_ROUND_SYNC() {
    startWg.Done()
}

function handler_NODE_DONE() {
    nNodes = nNodes - 1
    startWg.Done()
}

function handler_SPEC_COLOR_DONE() {
    colorWg.Done()
}
```

(b) Listener thread

Figure 5: A seemingly innocuous bit of synchronization that caused a major headache. Note that the number of nodes may decrease throughout the lifetime of the algorithm (due to nodes completing before other nodes), so it is important to count the number of `ROUND_SYNC` and `NODE_DONE` messages before setting the semaphores for the round; thus lines 13 and 14 explicitly follow the `startWg`. However, this causes problems: after `startWg` is released, the mainline thread may not get scheduled in immediately; it is possible that the listener thread may receive the `SPEC_COLOR_DONE` message and try to decrement `colorWg` before it is set, which causes a negative WaitGroup error.

One proposed way around this is to set `colorWg` before `startWg` is released, i.e., that `ROUND_SYNC` should increment `colorWg` in its handler, and this replaces line 14 of the mainline. However, this causes the possible race condition: assuming a two-worker system, node1 can be sleeping on `colorWg.Wait()`. Node2 then finishes speculative coloring, releasing `colorWg`, but node1's mainline thread does not get scheduled in immediately. Node2 continues to run, finishes the round, begins the next round, and broadcasts the `ROUND_SYNC` message. Node1's listener thread receives the message and decrements `colorWg` (per the new change) to a value of 1. However, since the mainline thread hasn't

woken yet, we are greeted with a error of "sync: WaitGroup is reused before previous Wait has returned."

A second proposed method to get around this is to perform locking of critical sections (e.g., using mutexes) to try eradicate the mistiming issues. However, no suitable way to do this was found that would block all mistimed WaitGroup operations and not also deadlock the main thread.

A third proposed method was to use a series of ACK messages (similar to TCP ACKs) to signal when a node is ready to accept WaitGroup operations. However, this adds an additional layer of synchronization that slows down the algorithm further and introduces an additional message type and handler, which is a lot of complexity (and thus a decrease in maintainability).

The final (working) solution is based on the idea that the fault in the original algorithm is simply due to a scheduling race condition. This solution involves trying to decrement `colorWg` in the handler for `SPEC_COLOR_DONE`; if this operation fails, then we catch the panic (exception) and force Go to yield this thread to the scheduler. This process is repeated as many times as necessary until the operation succeeds, at which point we know that the main thread has updated `colorWg`'s counter.

This method is a little tricky, since the only way to catch a panic is by a `defer`red handler on the stack; this defer method has to be recursive but have the proper break condition. The other tricky part is that when `colorWg.Done()` causes the negative WaitGroup panic, it still decrements the pointer; so, to ensure consistency, `colorWg` must be (atomically) re-incremented to zero, or else there might be problems on the mainline thread when it tries to correctly increment `colorWg`. Thus this introduces a critical section around the decrementing of `colorWg` and around the setting of `colorWg` in the mainline thread.

The last step is to schedule out the offending thread if it panics (i.e., if it runs too early). `runtime.Gosched()` does exactly this.

This last method is perhaps not great because it doesn't avoid the error, but instead it provides a sort of atomic "test and set" (TAS)-like functionality by using a lock and a semaphore. The benefit is that it only requires an extra mutex (one per socket connection) and, while it forces rescheduling (usually an expensive operation), this is only the Goroutine scheduler (more lightweight than the OS scheduler) and forces the correct (desired) scheduling order.

```
1   // recursive error recovery "trick" -- see documentation
2   var retry func()
3   retry = func() {
4       // consume error; keep retrying until success
5       if err := recover(); err != nil {
6
7           // if this happens, then lock already acquired
8           ws.ColorWg.Add(1)
9           ws.ColorWgLock.Unlock()
10
11          // algorithm is done, don't need to continue listening
12          if ws.State == distributed.STATE_FINISHED {
13              return
14          }
15
16          defer retry()
17
18          // give control to someone else (hopefully the main worker
19          // thread)
20          runtime.Gosched()
21
22          // retry
23          ws.ColorWgLock.Lock()
24          ws.ColorWg.Done()
25          ws.ColorWgLock.Unlock()
26      }
27  }
28  defer retry()
29
30  logger.Printf("Node %d has finished a round.\n", nodeIndex[0])
31
32  // decrease the number of nodes we are waiting for
33  ws.ColorWgLock.Lock()
34  ws.ColorWg.Done()
35  ws.ColorWgLock.Unlock()
```

Figure 6: The fix to the synchronization problem. Note that this also requires acquiring the `ws.ColorWgLock` for line 14 of 5 (and releasing it thereafter).

4 Performance issues

Mostly due to a lack of time, we haven't been able to improve performance by much. A 10000 vertex graph with an average degree of 1000, on a server and four virtual clients running on the same machine, takes almost a minute to color using the distributed algorithm, while it takes roughly 17ms using the single-node, multi-threaded algorithm. This is due to the following reasons:

4.1 Unavoidable overhead from distributed algorithm

The biggest drawback of this algorithm is that the graph is now distributed across the RAM of multiple nodes, meaning that efficient access to a node's neighbors (which is critical for this coloring algorithm) is much, much slower. As a result, a good partitioning is highly important for performance.

There is also the overhead of requiring each node to be synchronized at the start of each round, which waste CPU time. Logging also adds overhead. Running all the clients on the same host probably causes a slowdown as well due to sharing the same network interface.

4.2 Graph generation and partitioning

Real-world graphs tend to be very diverse in terms of vertex degree, and generally are heavily skewed with a large proportion of nodes having very small degree. This also means that they can be efficiently partitioned in a way that minimizes cross-edges between subgraphs. Gebremedhin et al. (2005) used real-world graphs from a disease dataset and partitioned using the METIS graph partitioning package, while our graphs were generated with a uniform degree and partitioned into linear blocks. In other words, we performed no optimization yet on minimizing internode I/O, and this means that much of our algorithm time is probably stuck in networking time.

4.3 Increased vertex info buffering

Currently, due to the lack of an extra data structure to store multiple vertices, vertices are sent and received one at a time. I/O speed can probably be improved by sending and receiving in larger chunks.

5 Next steps

The next step is to move this onto a truly distributed system (multiple physical/virtual hosts rather than the same physical and virtual host) using AWS infrastructure-as-code. This will provide the challenge of working with more numerous but less-individually-powerful nodes (standard T2 instances are single-threaded and are RAM-limited).