Lec 6: MapReduce Q&A

03/04/2021

Agenda

1. Lab1 solution walkthrough
2. Some alternative solution designs
3. Design mistakes and bugs
4. General tips
5. Q&A (submitted + any other questions you have!)

Code Walkthrough

1. Implement RPC structs
2. Write coordinator handers for RPCs
3. Create worker loop to send out GetTask RPCs and handle replies
4. Write worker helper functions to handle temporary/intermediate files
5. Implement worker map
6. Implement worker reduce
7. Create coordinator loop to handle requests and assign tasks

Alternative Synchronization Designs

- Wait in worker instead of coordinator
  - Sleep every loop if no task is available
  - Pros/Cons?

- Use time.Sleep() in coordinator instead of CondVar

- Channels (which actually are built using locks!)

Note: Waiting for map tasks to be done/synchronization is on a *single* server (cross-server communication only done by RPC!)

```go
/*
 * In this example, worker IDs are sent over a channel to the coordinator (e.g., when workers
 * connect over the network to the coordinator); workers may fail and restart, so the
 * coordinator must continuously read from the channel to issue new workers tasks.
 */
func Coordinator(workers chan int, numTask int, call_worker func(worker int, task int) bool) {
    // Hint: use bounded queues when possible
    tasks := make(chan int, numTask)
    done := make(chan bool)

    /*
     * IssueWorkersTasks goroutine creates per-worker threads
     * for every worker sent on the channel
     */
    go func() {
        for {
            // select blocks until one of the cases can run
            // if exit is unblocked, thread will return
            select {
            case worker := <-workers:
                go issueWorkerTaskThread(worker, done, tasks, call_worker)
            case <-exit:
                return
            }
        }
    }()

    /*
     * Add tasks to the work queue
     * this will not block because of the tasks channel size specification
     */
    for task := 0; task < numTask; task++ {
        tasks <- task
    }
    for i := 0; i < numTask; i++ {
        // blocks until a task is done
        <-done
    }

    // close tasks channel to indicate that tasks are all done
    // worker threads exit
    close(tasks)

    // indicate to IssueWorkersTasks threads
    // that no more tasks need to be issued
    exit <- true
}
```

```go
/*
 * issueWorkerTaskThread sends tasks to the worker
 * (call_worker sends an RPC to the worker with task information)
 *
 * If task fails, it is re-added to the task queue,
 * and chooses another pending task.
 * If no tasks are pending, thread exits.
 */
func issueWorkerTaskThread(
    worker int,
    done chan bool,
    tasks chan int,
    call_worker func(worker int, task int) bool) {

    for task := range tasks {
        if call_worker(worker, task) {
            // tell coordinator that a task is done
            done <- true
        } else {
            tasks <- task
        }
    }

}
```

```go
/*
 * In this example, worker IDs are sent over a channel to the coordinator (e.g., when workers
 * connect over the network to the coordinator); workers may fail and restart, so the
 * coordinator must continuously read from the channel to issue new workers tasks.
 */
func Coordinator(workers chan int, numTask int, call_worker func(worker int, task int) bool) {
    // Hint: use bounded queues when possible
    tasks := make(chan int, numTask)
    done := make(chan bool)

    /*
     * IssueWorkersTasks goroutine creates per-worker threads
     * for every worker sent on the channel
     */
    go func() {
        for {
            // select blocks until one of the cases can run
            // if exit is unblocked, thread will return
            select {
            case worker := <-workers:
                go issueWorkerTaskThread(worker, done, tasks, call_worker)
            case <-exit:
                return
            }
        }
    }()

    /*
     * Add tasks to the work queue
     * this will not block because of the tasks channel size specification
     */
    for task := 0; task < numTask; task++ {
        tasks <- task
    }
    for i := 0; i < numTask; i++ {
        // blocks until a task is done
        <-done
    }

    // close tasks channel to indicate that tasks are all done
    // worker threads exit
    close(tasks)

    // indicate to IssueWorkersTasks threads
    // that no more tasks need to be issued
    exit <- true
}
```

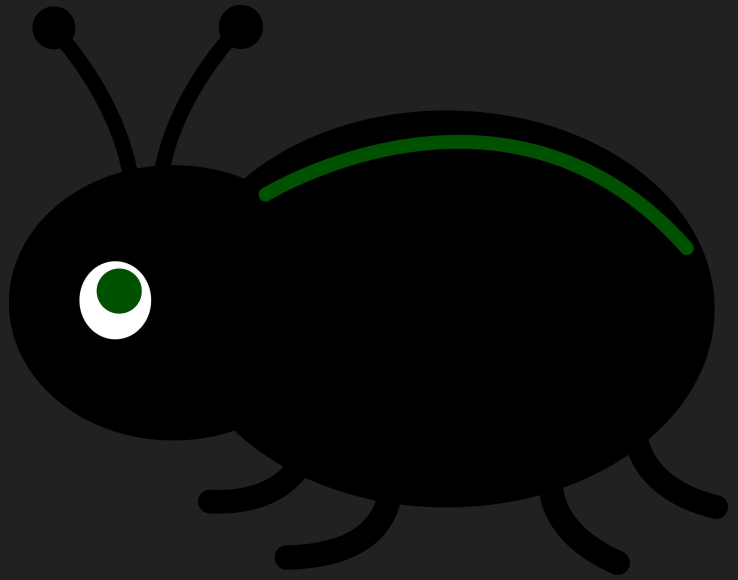
```go
/*
 * issueWorkerTaskThread sends tasks to the worker
 * (call_worker sends an RPC to the worker with task information)
 *
 * If task fails, it is re-added to the task queue,
 * and chooses another pending task.
 * If no tasks are pending, thread exits.
 */
func issueWorkerTaskThread(
    worker int,
    done chan bool,
    tasks chan int,
    call_worker func(worker int, task int) bool) {

    for task := range tasks {
        if call_worker(worker, task) {
            // tell coordinator that a task is done
            done <- true
        } else {
            tasks <- task
        }
    }

}
```

Common (but Passing) Design Mistakes

- Pushing too much work to the coordinator
  - Coordinator sorts results
  - Coordinator reads file contents

- Sending redundant RPCs

Interesting Bugs?



General Tips

- Printfs for debugging:
  - Conditional Printf (see DPrintf in raft/utils.go)
  - Formatting tricks: color scheme for RPCs, columns, server IDs
  - Redirect output to file (command &> debug_output.txt)

- Type Ctrl-\ to kill a program and dump all its goroutine stacks

- Defers pushed onto stack, executed in FIFO order when fxn returns

```
mu.Lock()
defer mu.Unlock()
defer FxnToRunBeforeUnlock()
```

Your questions!

MapReduce Questions

- MapReduce seems fairly easy to use for counting and sorting; what are some more complex tasks?
  - Many data mining, ML tasks (see Hadoop apps)
  - Anything that takes local computed statistics and computes a global statistic over them

- Can we use Raft or something similar to make the coordinator fault-tolerant?
  - Yes… but simpler to checkpoint progress

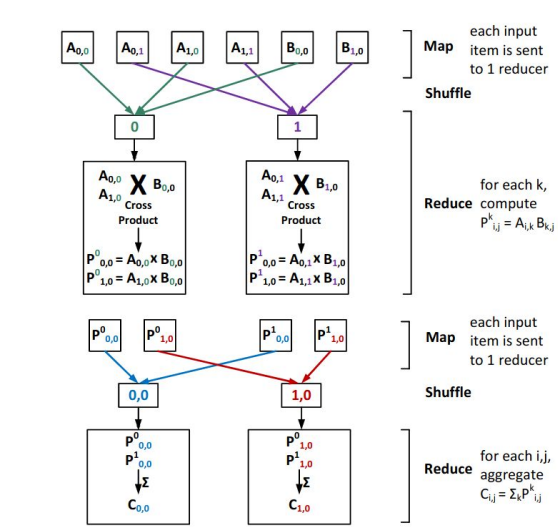


Fig. 2. CPMM: Cross Product based Matrix Multiplication

MapReduce Questions

- When does the shuffle (combiner) step happen, and what does it do?
  - Combining occurs at map (e.g., summing up word counts), result written to intermediate file
  - Sorting occurs at reduce after *all* (potentially combined) outputs from map are read by reducer
- Is there a successor to MapReduce that doesn't splitting a task to map and reduce phases, or has looser requirements?
  - Google Cloud Dataflow (user-specified directed computation graph and application code for individual nodes, execution plan dataflow graph)
- How are inputs partitioned in practice?
  - Natural divisions, or split into reasonable sizes of work (map task latency)
  - Up to programmer to specify

MapReduce Questions

- Why do mappers store files locally and not on GFS?
  - As the paper states: Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS) is stored on the local disks of the machines
- Are leaders necessary to have for distributed systems?
  - No, think of Bitcoin → decentralized
- Running MR on distributed servers---how?
  - Set up your RPCs to communicate over TCP/IP instead of than Unix sockets (see commented out line in Coordinator.server())
  - Read/write files using a shared file system (ssh into multiple Athena cluster machines at MIT, which use AFS to share files; or you could rent a couple AWS instances and use S3 for storage)

Code Design

- How to organize massive pieces of code? (e.g., for lab 2)
  - Separate by RPC sender+handler (use different files)
  - Put all definitions of state (structs) together
  - Factor out common pieces of code into functions (e.g., checking for stale term)
  - Good environment (vim + ctags, autocomplete, keyword search etc.) can help a lot!
- Has using Go for labs decreased the amount of time students spend debugging compared to C++?
  - Go uses garbage collection! (How many of you ran into segfaults?)

Code Design

- What is the difference between using a pointer and a value for the arguments object to an RPC call?
  - Passing a reference can be cheaper (doesn't require copying the struct to invoke RPC)
- Using both locks and channels possible?
  - Yes! You'll do so in Raft too
- How to choose timeouts? How sensitive are tests to timeouts?
  - What is reasonable: want to allow for chance to receive RPCs from other servers, so has to be greater than heartbeat; will likely need to experiment

Implementation Questions

- Why do we have a timeout to retry failed tasks in the labs, instead backup tasks as described in the MapReduce paper?
  - Paper describes timeouts are used to restart tasks when workers fail
  - Backup tasks as described are used to speed lagging tasks at the end of execution
  - Our design uses timeouts **both** to detect worker failure, and also to detect slow tasks!
- If the servers were not in the same machine, do you have to use different synchronization approaches?
  - Nope! All synchronization is local; RPCs are used for cross-server communication

Implementation Questions

- What were common sources of race conditions?
  - Forgetting to lock/unlock; using a counter of number tasks completed, which might accidentally count one task twice
  - Data races are undefined behavior! (Might *seem* benign, but you should handle them)
- What is a clean way to exit the workers and coordinator?
  - Send an “Exit” RPC from coordinator to worker as final response to GetTask RPC
  - Messy exit ok?
- Where do unexpected EOF errors come from?
  - RPC Clients (workers) calling socket when coordinator has closed
  - https://github.com/golang/go/blob/master/src/net/rpc/client.go#L157