Relatório da Atividade 3: Map Reduce

Isabelle Ferreira de Oliveira
CES-27 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

***Resumo*—Esse relatório documenta um trabalho com o modelo de programação MapReduce de forma sequencial e distribuída.**

***Index Terms*—Map Reduce, Golang, sistema sequencial, sistema distribuído**

I. Implementação

A. Parte 1: Trabalhando em modo sequencial

1) Tarefa 1.1: Bastou percorrer a lista de palavras presentes em *words* e, para cada palavra, adicionar a um array de mapreduce.KeyValue um novo item, contendo a palavra em questão como atributo *Key* e "1"como atributo *Value*.

```
func mapFunc(input []byte) (result
    []mapreduce.KeyValue) {

  // [...]

  var item mapreduce.KeyValue

  for _, word := range words {
    word = strings.ToLower(word)
    item.Key = word
    item.Value = "1"
    result = append(result, item)
  }

  return result
}
```

2) Tarefa 1.2: Bastou percorrer o array *input* de mapreduce.KeyValue, atualizando um mapa a fim de agrupar as palavras repetidas nesse *input*. O atributo Key de cada item desse *input* era a palavra que serviria de chave nesse mapa. Caso a palavra já estivesse no mapa, acrescentava-se o valor contido no atributo Value do item ao valor já presente no mapa. Caso contrário, esse novo item era adicionado no mapa, com o atributo Value sendo setado como valor inicial dessa chave. Após isso, percorreu-se esse mapa, adicionando os elementos no array *result* de mapreduce.KeyValue. Era necessário, também, vazer as conversões de inteiro para string, e vice-versa, para adicionar corretamente nos mapas e no array *result* de mapreduce.KeyValue.

```
func reduceFunc(input []mapreduce.KeyValue)
    (result []mapreduce.KeyValue) {

  var mapAux map[string]int =
      make(map[string]int)
  var value int

  for _,item := range input {
    value, _ = strconv.Atoi(item.Value)
    _, ok := mapAux[item.Key]
    if ok {
      mapAux[item.Key] += value
    } else {
      mapAux[item.Key] = value
    }
  }

  var itemAux mapreduce.KeyValue

  for key,value := range mapAux {
    itemAux.Key = key
    itemAux.Value = strconv.Itoa(value)
    result = append(result, itemAux)
  }

  return result
}
```

3) Tarefa 1.3: Executou-se o comando *"./wordcount -mode sequential -file files/teste.txt -chunksize 100 -reducejobs 2"*, conforme sugerido no roteiro.

4) Tarefa 1.4: Executou-se o comando da Tarefa 1.3, dessa vez para o arquivo *files/musica.txt*. Os valores de *chunksize* e *reducejob* foram alterados para: 100 e 2; 100 e 1; e 102400 e 2. O arquivo *musica.txt* pode ser observado no código enviado em anexo ao relatório.

B. Parte 2: Trabalhando em modo distribuído

1) Tarefa 2.1: Bastou, para cada elemento presente no canal *master.failedWorkerChan*, deletar esse worker que falhou da lista de workers *master.workers* e decrescer o total de workers disponível *master.totalWorkers*. Antes e depois dessa alteração, foi utilizado o *Lock()* e o *Unlock()*, respectivamente, do *master.workersMutex*, a fim de evitar acesso simultâneo à região crítica.

```
func (master *Master) handleFailingWorkers() {

  for elem := range master.failedWorkerChan {
    master.workersMutex.Lock()
    delete(master.workers, elem.id)
```

```
      master.totalWorkers--
      master.workersMutex.Unlock()

      fmt.Println("Removing worker", elem.id,
          "from master list")
    }
}
```

2) Tarefa 2.2: Primeiramente foi necessário criar um novo canal (*failedOperationChan*) na *Master*, para transmitir a operação que falhou. É importante salientar também a inicialização do canal *master.failedOperationChan*.

```
type Master struct {
  // [...]

  // Fault Tolerance
  failedOperationChan chan *Operation
}
```

Era necessário também adicionar a operação falhada ao canal quando essa operação falhasse. Também foi removido o *wg.Done()* desse momento, uma vez que essa operação não foi realmente concluída.

```
func (master *Master)
    runOperation(remoteWorker *RemoteWorker,
    operation *Operation, wg *sync.WaitGroup)
    {

  // [...]

  if err != nil {
    log.Printf("Operation %v '%v' Failed.
        Error: %v\n", operation.proc,
        operation.id, err)

    master.failedWorkerChan <- remoteWorker
    master.failedOperationChan <- operation
  } else {
    wg.Done()
    master.idleWorkerChan <- remoteWorker
  }
}
```

Foi criada uma função para ser rodada em thread que lidasse com as operações falhadas (função *handleFailingOperations*). Essa função ficava a espera de alguma operação ser colocada no canal *master.failedOperationChan* e, cada vez que isso acontecia, esperava-se o próximo worker ser adicionado ao canal *master.idleWorkerChan*, a fim de essa operação falhada ser atribuída a esse worker. Essa nova atribuição era feita através da chamada de uma nova thread executando *master.runOperation*.

```
func (master *Master)
    handleFailingOperations(wg
    *sync.WaitGroup) {

  var operation *Operation
  var worker *RemoteWorker
  var ok bool

  for {
    operation, ok =
        <-master.failedOperationChan
    if !ok { break }

    worker, _ = <-master.idleWorkerChan

    go master.runOperation(worker, operation,
        wg)
  }
}
```

Essa função *master.handleFailingOperations* foi chamada em thread na função *schedule*.

```
func (master *Master) schedule(task *Task,
    proc string, filePathChan chan string)
    int {

  var (
    wg      sync.WaitGroup
    filePath string
    worker  *RemoteWorker
    operation *Operation
    counter int
  )

  master.failedOperationChan = make(chan
      *Operation, RETRY_OPERATION_BUFFER)

  go master.handleFailingOperations(&wg)

  // [...]
}
```

3) Tarefa 2.3: Executou-se o comando *"./wordcount -mode sequential -file files/teste.txt -chunksize 100 -reducejobs 2"*, além de dois workers. Na primeira situação, um dos workers falhou na operação 2 (map operation). Já na segunda situação, um dos workers falhou na operação 3 (reduce operation).

4) Tarefa 2.4: Executou-se o comando *"./wordcount -mode sequential -file files/pg1342.txt -chunksize 102400 -reducejobs 5"* em um primeiro momento; e o mesmo comando com a diferença de ter apenas 1 reducejob em um segundo momento. Em ambas as situações, dois workers falharam e um terceiro não falhou.

II. Resultados e Conclusões

A. Teste 1

Este foi o caso com um processo solicitando a CS e, depois que ele liberasse, outro processo solicitando a CS, sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura 1.

Na Figura 1, para P1, as setas azuis representam requests e as verdes, replies; para P3, essa setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada é da forma "*relógio lógico, < timestamp, id >*"; já no reply, a forma é "*relógio lógico, 'reply'*". Além disso, a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

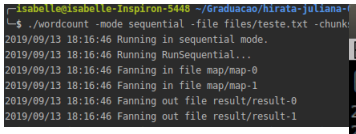

Figura 1. Teste

Foi simulada essa situação acima co
mentado no laboratório, tendo os resulta
Figuras de 2 a 5. Na simulação, ao in
requisição, é o P2 a faz, mas isso não pr

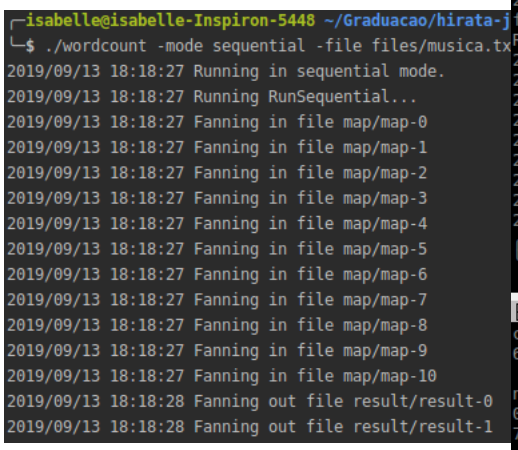

Figura 2. Exemplo do funcionamento da tarefa c
processo 1.

Figura 3. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

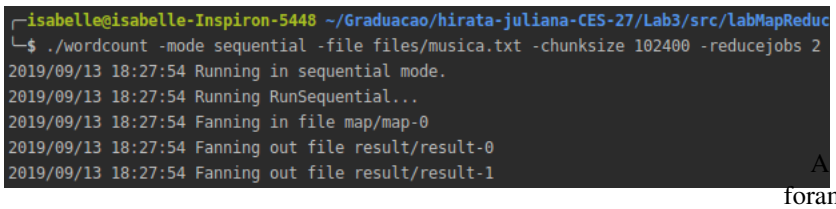

Figura 4. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

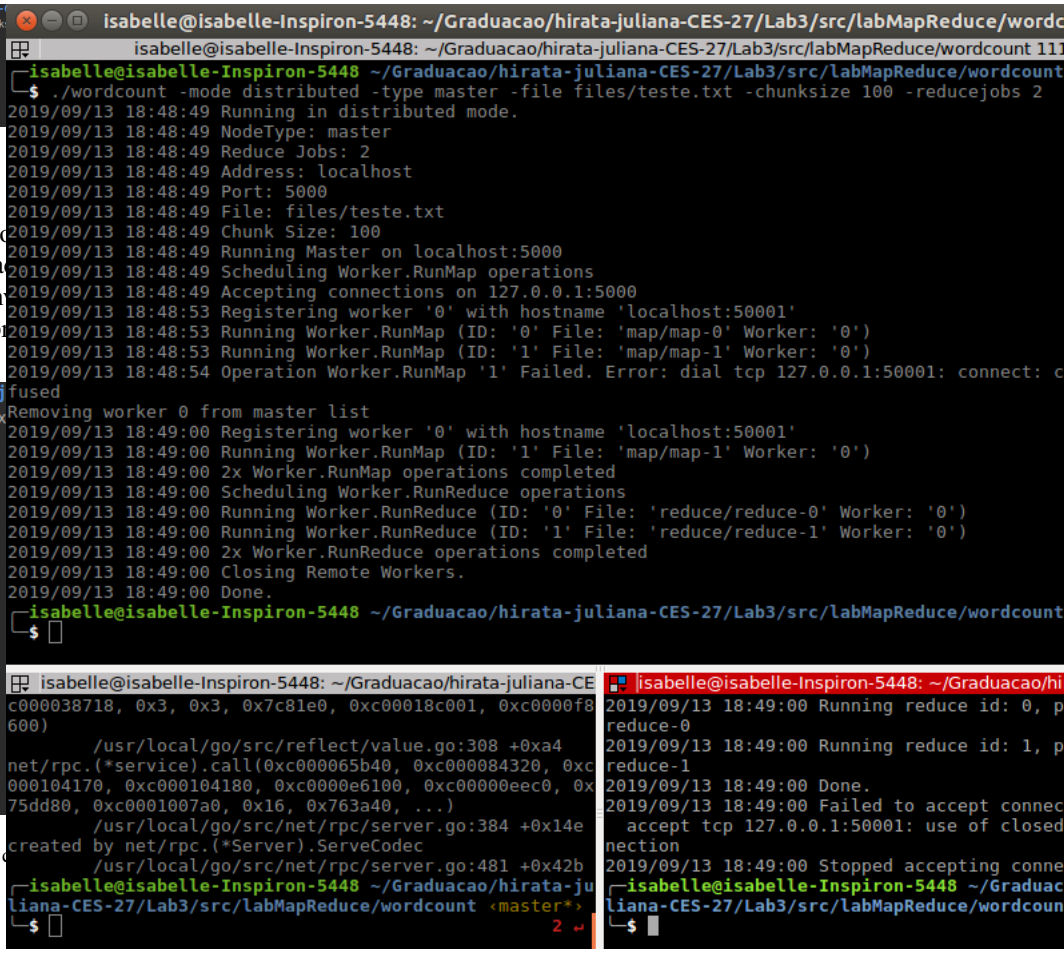

Figura 5. Exemplo do funcionamento da tarefa com 3 processos. Tela do SharedResource.

A fim de entender melhor cada estágio do funcionamento, foram realizados os mesmos testes novamente, agora com prints de debug. Dessa forma, esses resultados estão apresentados nas Figuras de 6 a **??**.

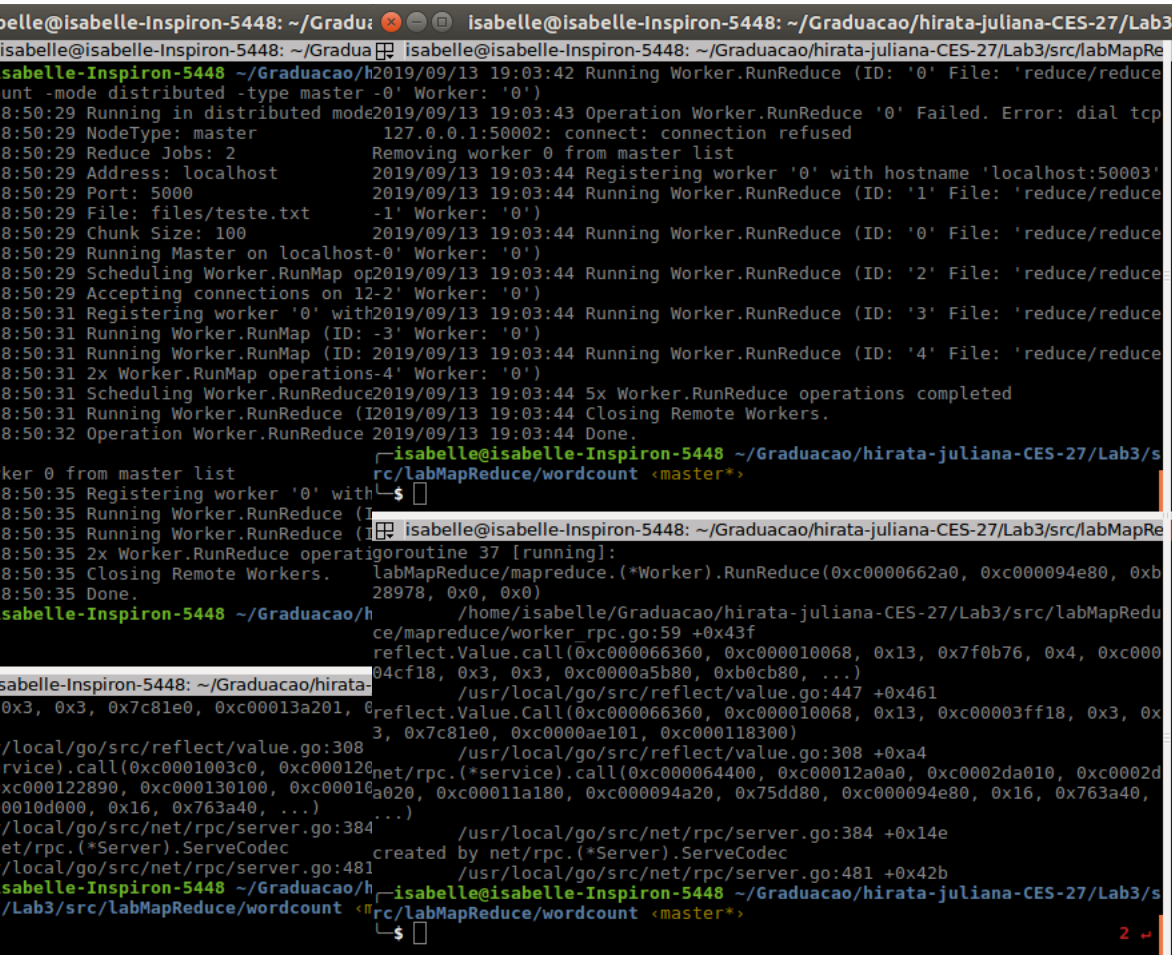

Figura 6. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 1.

.

Figura 7. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

.

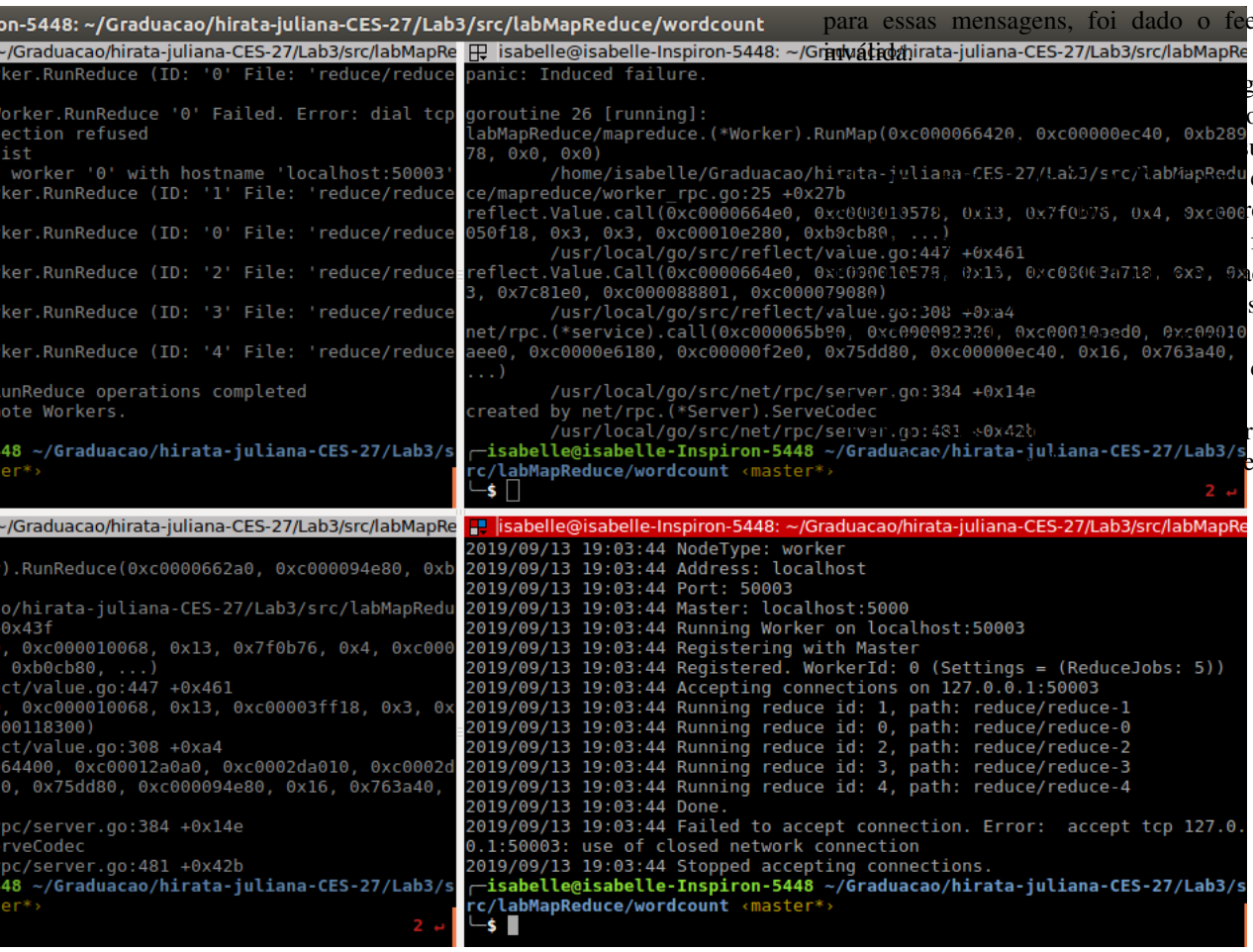

Figura 8. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

Como esses resultados foram condizentes com os resultados esperados, leva-se a perceber que a implementação da Tarefa foi feita corretamente. Mas, antes de concluir-se algo, foi-se realizado um segundo teste.

B. Teste 2

Este foi o caso com processos solicitando a CS "simultaneamente", sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura **??**.

Análogo ao teste 1, na Figura **??**, para P1, as setas azuis representam requests e as verdes, replies; e para P4, essa setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada também é da forma "*relógio lógico, < timestamp, id >*"; assim como no reply, que a forma também é "*relógio lógico, 'reply'*". Além disso, nesse caso também a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

Foi simulada essa situação acima com o código implementado no laboratório, tendo os resultados apresentados nas Figuras de **??** a **??**. Durante o período na CS, também foi digitado mensagens no terminal dos processos, para que se verifisse o funcionamento da validação da mensagem. Assim, para essas mensagens, foi dado o feedback de mensagem inválida.