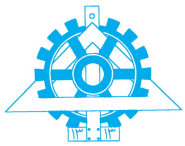
مبانی رایانش توزیع شده
بهار ۱۴۰۴



پروژه اول

۱۴۰۴/۱/۲۹

پیاده‌سازی $MapReduce$ به صورت توزیع شده

مقدمه

در این پروژه، شما یک سیستم $MapReduce$ را به صورت توزیع شده پیاده‌سازی خواهید کرد. در این پروژه، شما چندین فرآیند $Worker$ را پیاده‌سازی می‌کنید که وظیفه اجرای تسک‌های Map و $Reduce$ را بر عهده دارند. تسک‌های Map و $Reduce$، نتایج پردازش خود را در فایل‌ها ذخیره کرده و ورودی‌هایشان را نیز از فایل‌های مربوطه دریافت می‌کنند. همچنین یک فرآیند هماهنگ‌کننده ($Coordinator$) که وظایف را به $Worker$ها واگذار کرده و $Worker$ های $Fail$ شده را مدیریت می‌کند را پیاده سازی خواهید کرد.

تعریف مسئله

وظیفه شما پیاده‌سازی یک سیستم $MapReduce$ توزیع‌شده شامل دو برنامه است:

- هماهنگ‌کننده ($Coordinator$)
- کارگر ($Worker$)

در این مسئله در ابتدا یک فرآیند هماهنگ‌کننده اجرا خواهد شد، و سپس یک یا چند فرآیند کارگر به‌صورت موازی اجرا خواهند شد. در یک سیستم واقعی، کارگران روی چندین ماشین به صورت مجزا اجرا می‌شوند، اما در این پروژه برای سادگی مسئله، فرض می‌کنیم همه فرآیندها روی یک ماشین اجرا خواهند شد.
کارگران از طریق مکانیزم RPC با هماهنگ‌کننده ارتباط برقرار می‌کنند. هر فرآیند کارگر به‌صورت تکراری در یک حلقه اجرا خواهد شد طوری‌که از هماهنگ‌کننده درخواست یک تسک (Map یا $Reduce$) می‌کند و سپس ورودی تسک را از یک یا چند فایل می‌خواند و تسک را اجرا می‌کند و سپس خروجی تسک را در یک یا چند فایل می‌نویسد و در نهایت دوباره از هماهنگ‌کننده درخواست یک تسک جدید می‌کند. همچنین هماهنگ‌کننده باید متوجه شود که اگر یک کارگر ظرف مدت معقولی (در این پروژه ۱۰ ثانیه) اجرای تسک را تکمیل نکرد، همان تسک را به یک کارگر دیگر واگذار کند.

ساختار پروژه

برای انجام این پروژه، شما نیاز به نصب Go دارید. در صورت نیاز می‌توانید آخرین نسخه موجود را از وبسایت رسمی آن دریافت کنید. به‌طور کلی، در این پروژه دایرکتوری‌های $mrapps$، $main$ و mr از جمله پرکاربردترین بخش‌ها هستند و نقش کلیدی را در این پروژه دارند. کدهای مربوط به توابع $main$ برای هماهنگ‌کننده و کارگر در $main/mrcoordinator.go$ و $main/mrworker.go$ قرار دارند. این فایل‌ها را تغییر ندهید. توصیه می شود که سورس کد این دو فایل را به صورت دقیق مطالعه کنید. در واقع این دو فایل نقطه شروعی برای اجرای فایل‌های دایرکتوری mr هستند.

شما باید پیاده‌سازی خود را در فایل‌های $mr/coordinator.go$، $mr/worker.go$ و $mr/rpc.go$ انجام دهید. یک پیاده‌سازی ساده از $MapReduce$ به صورت ترتیبی در فایل $src/main/mrsequential.go$ در اختیار شما قرار داده شده است. این پیاده‌سازی به صورت ترتیبی و در یک فرایند واحد، توابع map و $reduce$ را اجرا می‌کند. همچنین چند مسئله مختلف مانند شمارشگر کلمات در $mrapps/wc.go$ و یک ایندکس‌گر متنی در $mrapps/indexer.go$ که با استفاده از فریمورک $MapReduce$ حل شده است به صورت از پیش آماده شده برای شما آورده شده است. برنامه $mrsequential.go$ خروجی خود را در فایل $mr\text{-}out\text{-}0$ قرار می‌دهد. ورودی آن از فایل‌های متنی به نام $pg\text{-}xxx.txt$ که در دایرکتوری $main$ پروژه هستند، گرفته می‌شود. شما می‌توانید از کد $mrsequential.go$ برای راهنمایی استفاده کنید. همچنین توصیه می‌شود که به $mrapps/wc.go$ نگاهی بیندازید تا با نحوه پیاده‌سازی یک برنامه به روش $MapReduce$ آشنا شوید.

```
└──src
    ├──main
    │   ├── Contains all input files required for this project (`pg-*.txt`).
    │   ├── Includes test scripts and main files necessary to execute the project.
    │   ├── Carefully examine `mrcoordinator.go`, `mrsequential.go`, and
`mrworker.go`.
    │
    ├──mr
    │   ├── This is the directory where you need to add your implementation for the
coordinator, worker, and RPC components.
    │
    ├──mrapps
    │   ├── Contains sample problems that demonstrate solutions using the MapReduce
framework.
```

نحوه اجرای کد

شما می‌توانید برنامه شمارش کلمات را به صورت ترتیبی به شرح زیر اجرا کنید. لازم به ذکر است که *build* کردن یک پلاگین در زبان *go* در سیستم‌عامل ویندوز پشتیبانی نمی‌شود.

```
$ cd ~/6.5840
$ cd src/main
$ go build -buildmode=plugin ../mrapps/wc.go
$ rm mr-out*
$ go run mrsequential.go wc.so pg*.txt
$ more mr-out-0
A 509
ABOUT 2
ACT 8
...
```

برای اجرای برنامه *MapReduce* توزیع شده بر روی برنامه شمارشگر کلمات در ابتدا اطمینان حاصل کنید که پلاگین شمارشگر کلمات را مجددا بیلد کنید.

```
$ go build -buildmode=plugin ../mrapps/wc.go
```

سپس در دایرکتوری *main*، هماهنگ‌کننده را اجرا کنید:

```
$ rm mr-out*
$ go run mrcoordinator.go pg-*.txt
```

آرگومان *pg-*.txt* در *mrcoordinator.go* فایل‌های ورودی را مشخص می‌کند؛ که هر فایل ورودی یک تسک *Map* خواهد بود.

در یک یا چند پنجره دیگر، می‌توانید کارگران را به صورت زیر اجرا کنید:

```
$ go run mrworker.go wc.so
```

پس از اینکه هماهنگ‌کننده و کارگران کار خود را به پایان رساندند، خروجی را در فایل‌های *mr-out-** بررسی کنید. در نهایت پیاده سازی شما در صورتی درست تلقی می‌شود که، ترکیب مرتب‌شده‌ی فایل‌های خروجی اجرای توزیع شده باید با خروجی اجرای ترتیبی (*mrsequential.go*) مطابقت داشته باشد به عنوان مثال:

```
$ cat mr-out-* | sort | more
A 509
ABOUT 2
ACT 8
...
```

تست کردن پیاده‌سازی انجام شده

یک تست‌اسکریپت در فایل *main/test-mr.sh* آورده شده است. این تست‌ها بررسی می‌کنند که آیا برنامه‌های (پلاگین) شمارشگر کلمات (*wc*) و ایندکس‌کننده (*indexer*) خروجی درستی تولید می‌کنند یا خیر. همچنین، این تست‌ها بررسی می‌کنند که وظایف *Map* و *Reduce* به صورت موازی اجرا شوند و سیستم بتواند در صورت خرابی کارگران، خود را بازیابی کند.
جهت اجرای تست‌ها، دستور زیر را اجرا کنید:

```
$ cd ~/6.5840/src/main
$ bash test-mr.sh
```

با اجرای این تست، احتمالا برنامه شما متوقف خواهد شد، زیرا در ابتدای کار هماهنگ‌کننده هیچ‌وقت متوقف نمی‌شود. برای رفع این مشکل، مقدار $ret := false$ را در تابع *Done* در *mr/coordinator.go* به *true* تغییر دهید تا هماهنگ‌کننده بلافاصله متوقف شود. سپس دوباره تست را اجرا کنید:

```
$ bash test-mr.sh
```

خروجی احتمالی در صورت ناموفق بودن تست به صورت زیر خواهد بود:

```
*** Starting wc test.
sort: No such file or directory
cmp: EOF on mr-wc-all
--- wc output is not the same as mr-correct-wc.txt
--- wc test: FAIL
$
```

دلیل پاس نشدن این تست این است که پیاده‌سازی اولیه شما هیچ فایل خروجی‌ای تولید نمی‌کند ($mr\text{-}out\text{-}X$)، بنابراین تست پاس نمی‌شود.
پس از تکمیل پیاده‌سازی، خروجی تست موفق‌آمیز باید به شکل زیر باشد:

```
$ bash test-mr.sh
*** Starting wc test.
--- wc test: PASS
*** Starting indexer test.
--- indexer test: PASS
*** Starting map parallelism test.
--- map parallelism test: PASS
*** Starting reduce parallelism test.
--- reduce parallelism test: PASS
*** Starting job count test.
--- job count test: PASS
*** Starting early exit test.
--- early exit test: PASS
*** Starting crash test.
--- crash test: PASS
*** PASSED ALL TESTS
$
```

پیام‌های خطای قابل چشم‌پوشی

ممکن است پیام‌های خطایی از $Go\ RPC$ ببینید، مانند:

```
2025/02/11 13:27:09 rpc.Register: method "Done" has 1 input parameters; needs exactly
    three
```

می‌توانید این پیام‌ها را نادیده بگیرید، زیرا تابع $Done$ از طریق RPC فراخوانی نمی‌شود.
همچنین ممکن است خطاهایی مانند زیر دریافت کنید:

```
2025/02/11 16:21:32 dialing:dial unix /var/tmp/5840-mr-501: connect: connection refused
```

رخداد این پیام‌ها طبیعی است و زمانی رخ می‌دهد که یک کارگر سعی می‌کند به هماهنگ‌کننده متصل شود، اما هماهنگ‌کننده قبلاً متوقف شده است. مشاهده چند نمونه از این خطا در هر تست مشکلی ندارد.

قوانین پیاده‌سازی

- هر یک از تسک‌های Map خروجی را در $nReduce$ فایل واسط قرار می‌دهد. به عبارت دیگر هر تسک Map باید $nReduce$ فایل میانی تولید کند تا هر تسک $Reduce$ بتواند از آن‌ها استفاده کند.
- توجه شود که مقدار $nReduce$ به عنوان آرگومان به $MakeCoordinator()$ در $mrcoordinator.go$ ارسال می‌شود.
- خروجی هر یک از تسک‌های Map باید در دایرکتوری یکسان ذخیره شود تا هر تسک $Reduce$ بتواند به آن دسترسی داشته باشد.
- فایل‌های واسط برای ذخیره خروجی هر تسک Map باید با فرمت $mr\text{-}X\text{-}Y$ نام‌گذاری شوند، به طوری که X شماره تسک Map و Y شماره تسک $Reduce$ باشد.
- استفاده از $JSON$ برای ذخیره خروجی تسک Map

برای ذخیره خروجی هر تسک Map در فایل‌ها، می‌توان از پکیج $encoding/json$ استفاده کرد، به عنوان مثال:

نوشتن خروجی در فایل $JSON$:

```
enc := json.NewEncoder(file)
for _, kv := range intermediate {
    err := enc.Encode(&kv)
}
```

خواندن از فایل $JSON$:

```
dec := json.NewDecoder(file)
for {
```

```
    var kv KeyValue
    if err := dec.Decode(&kv); err != nil {
        break
    }
    kva = append(kva, kv)
    }
```

- هر کلید خروجی از *Map* باید بر اساس مقدار دریافتی از تابع زیر به یک *Reduce* خاص تخصیص داده شود. این تابع در فایل *mr/worker.go* قرار دارد.

```
// use ihash(key) % NReduce to choose the reduce
// task number for each KeyValue emitted by Map.
//
func ihash(key string) int {
   h := fnv.New32a()
   h.Write([]byte(key))
   return int(h.Sum32() & 0x7fffffff)
}
```

- خروجی هر تسک *Reduce* باید در فایل با نام *mr-out-X* قرار گیرد، طوری‌که *X* شماره *Reduce* مورد نظر است.

- فرمت هر خط در *mr-out-X* باید به صورت *"key value"* و با قالب *"%v %v"* در زبان *Go* باشد. می‌توانید نمونه‌ی صحیح را در *mrsequential.go* در خطی که با *"this is the correct format"* مشخص شده است، ببینید. توجه شود که در صورت رعایت نکردن این فرمت، تست‌ها پاس نمی‌شوند.

- توجه شود که *worker* ها تسک‌های *reduce* را نباید اجرا کنند تا زمانی که اجرای تسک‌های *map* تمام نشده باشد.

- تابع *Done()* موجود در فایل *mr/coordinator.go* باید به گونه‌ای پیاده‌سازی شود که در صورت اتمام اجرای *MapReduce* به صورت کامل، مقدار *true* را به *main/mrcoordinator.go* بازگرداند و سپس اجرای *mrcoordinator.go* متوقف می‌شود.

- توجه شود که اجرای *Worker* ها زمانی‌که همه تسک‌ها پایان پذیرفت، باید متوقف شود. بدین منظور یک روش ساده این است که مقدار بازگشتی تابع *Call()* را بررسی کنید؛ اگر *Worker* نتواند به هماهنگ‌کننده متصل شود، می‌تواند فرض کند که کار تمام شده و از برنامه خارج شود.

- برای آشنایی بیشتر با نحوه خواندن ورودی‌های *Map* و مرتب‌سازی کلید/مقدارهای میانی بین مراحل *Map* و *Reduce* همچنین نحوه ذخیره خروجی *Reduce* در فایل‌ها *(mr-out-X)* می‌توانید از کد *mrsequential.go* ایده بگیرید.

- لازم به ذکر است که از آنجایی که *Coordinator* به‌عنوان یک سرور *RPC* به صورت همروند اجرا می‌شود، باید هنگام دسترسی به داده‌های مشترک استفاده از مکانیزم‌هایی همانند قفل‌ها را در نظر بگیرید.

- توجه کنید که اجرای *Worker* ها ممکن است تصادفاً متوقف شود در این صورت هماهنگ‌کننده (*Coordinator*) نمی‌تواند تشخیص دهد که آیا یک *Worker* کرش کرده است، کند شده است یا گیر کرده است بدین منظور برای مدیریت کردن این سناریو، در صورتی که یک *Worker* بیش از ۱۰ ثانیه اجرای تسکی را تکمیل نکرد، *Coordinator* فرض می‌کند که آن *Worker* از کار افتاده است و آن تسک را به یک کارگر دیگر تخصیص می‌دهد.

- برای جلوگیری از مشاهده فایل‌های نیمه‌نوشته‌شده در صورت وقوع *Crash*، مقاله *MapReduce* [۱] از یک ترفند استفاده می‌کند: ابتدا داده‌ها در یک فایل موقت نوشته می‌شوند و پس از اتمام نوشتن، فایل به‌صورت اتمیک *(atomic)* تغییر نام داده می‌شود. در *Go*، می‌توان از *ioutil.TempFile* (یا *os.CreateTemp* در نسخه‌های ۱۷.۱ به بعد) برای ایجاد یک فایل موقت استفاده کرد و سپس با *os.Rename* نام آن را نهایی کرد.

- برای آزمایش *Crash Recovery*، می‌توانید از افزونه *mrapps/crash.go* استفاده کنید. این افزونه به‌صورت تصادفی در توابع *Map* و *Reduce* از برنامه خارج می‌شود.

- در *RPC* زبان *Go*، فقط فیلدهایی از یک *struct* ارسال می‌شوند که نام آن‌ها با حروف بزرگ شروع شود. همچنین، اگر یک *struct* شامل *sub-struct* های دیگر باشد، فیلدهای آن‌ها نیز باید با حروف بزرگ شروع شوند تا قابل ارسال باشند.

- هنگام فراخوانی تابع *Call()* در *RPC* ساختار *reply* باید به‌صورت پیش‌فرض مقداردهی نشده باشد. به عبارت دیگر فراخوانی‌های *RPC* باید به این شکل انجام شوند:

```
reply := SomeType{}
call(..., &reply)
```

بدون اینکه قبل از فراخوانی، مقدار خاصی به فیلدهای *reply* اختصاص داده شود. اگر *reply* حاوی مقادیر غیرپیش‌فرض باشد، ممکن است سیستم RPC به‌طور نامحسوس مقادیر نادرستی را برگرداند.

به نکات زیر توجه کنید:

- مهلت ارسال در سربرگ تمرین همچنین در ایلرن درج شده‌است.
- گزارش باید در $LaTeX$ نوشته شود. فایل تمرین ارسالی باید شامل فایل‌های مورد نیاز به جهت اجرای فایل $LaTeX$ به همراه PDF باشد. نام فایل را به‌صورت زیر انتخاب کنید:

```
PR1_Student#_Name
```

- ارسال با تأخیر پروژه، تنها طبق قوانین درس امکان‌پذیر بوده است.

موفق باشید.

مراجع

[1] M. Dayalan, "Mapreduce: simplified data processing on large clusters," Commun. ACM, vol.51, pp.107–113, 2008.