Project 2: MapReduce

CSE 130 Principles of Computer Systems Design

Spring 2023

Project 2 is out!

- MapReduce-style Multi-threaded Data-Processing Library
- Uses POSIX Threads (pthread) to process data in parallel
- Due 4/30
- GitHub Classroom to create a repo, submit on Gradescope

MapReduce

> MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model...

"MapReduce: Simplified Data Processing on Large Clusters" Dean & Ghemawat

Programming Model

Users provide two functions: `map` and `reduce`

> Map takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

> The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

Example: Word Counting Problem (1/3)

Suppose that there are a large collection of text documents and we want to count the number of occurrences of each word.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v); Emit(AsString(result));
```

Example: Word Counting Problem (2/3)

Input: (document name, document contents)

```
[("file1", "hello world"), ("file2", "good afternoon world")]
```

Map(Input)

```
[
  ("hello", 1), ("world", 1), ("good", 1),
  ("afternoon", 1), ("world", 1)
]
```

Example: Word Counting Problem (3/3)

Group by Key

```
{
  [("hello", 1)],
  [("world", 1), ("world", 1)],
  [("good", 1)],
  [("afternoon", 1)],
}
```

Reduce

```
[("hello", 1), ("world", 2), ("good", 1), ("afternoon", 1)]
```

Project

Complete the `map_reduce` function in `mr.c`

```
void map_reduce(mapper_t mapper, size_t num_mapper, reducer_t reducer,
                size_t num_reducer, kvlist_t* input, kvlist_t* output);
```

- `mapper` is a function that performs the `map` operation.
- `num_mapper` is the number of threads used to execute `mapper`.
  - For example, if `num_mapper == 8`, `map_reduce` will spawn eight threads, each processing a subset of the data
- `reducer` is a function that performs the `reduce` operation.
- `num_reducer` is the number of threads used to execute `reducer`.
- `input` is a list of key-value pairs that represent the data to process
- `output` is a list of key-value pairs that `map_reduce` writes the results to.

`kvlist.h`

`map_reduce` needs to work with lists of key-value pairs. Because C arrays are hard to work with (especially when the size is not known), `kvlist.h` provides two data structures, `kvpair_t` and `kvlist_t`.

`kvpair_t`

`kvpair_t` stores a pair strings: key and value.

```c
typedef struct kvpair_t {
  char *key;
  char *value;
} kvpair_t;

// creates a new `kvpair_t` by copying the provided `key` and `value`
kvpair_t *kvpair_new(char *key, char *value);

// creates a copy of `kv`.
kvpair_t *kvpair_clone(kvpair_t *kv);

// `kvpair_free` frees `kvpair_t`
void kvpair_free(kvpair_t **kv);

// `kvpair_update_value` updates the value of the pair.
void kvpair_update_value(kvpair_t *pair, char *new_value);
```

`kvlist_t`

`kvlist_t` is a linked list of `kvpair_t`

```c
// `kvlist_new` creates a new `kvlist_t`.
kvlist_t *kvlist_new(void);

// `kvlist_free` frees `kvlist_t`. It also frees all pairs inside the list.
void kvlist_free(kvlist_t **lst);

// `kvlist_append` appends the pair `kv` to the list `lst`.
void kvlist_append(kvlist_t *lst, kvpair_t *kv);

// `kvlist_extend` concatenates two lists `lst` and `lst2`.
void kvlist_extend(kvlist_t *lst, kvlist_t *lst2);

// `kvlist_sort` sorts the list by keys.
void kvlist_sort(kvlist_t *lst);

// `kvlist_print` prints the contents of `lst` to the file descriptor `fd`.
void kvlist_print(int fd, kvlist_t *lst);
```

`kvlist_iterator_t`

Use `kvlist_iterator_t` to iterate through lists.

```
/**
 * `kvlist_iterator_new` creates a new iterator.
 */
kvlist_iterator_t *kvlist_iterator_new(kvlist_t *lst);

/**
 * `kvlist_iterator_next` returns the next `kvpair_t`.
 * It returns `NULL` if there is no more pair.
 */
kvpair_t *kvlist_iterator_next(kvlist_iterator_t *it);

/**
 * `kvlist_iterator_free` frees `kvlist_iterator_t`.
 */
void kvlist_iterator_free(kvlist_iterator_t **it);
```

`kvlist.h` in action

```c
// construct a list
kvlist_t* list = kvlist_new();
// append 3 pairs
kvlist_append(list, kvpair_new("key1", "value1"));
kvlist_append(list, kvpair_new("key2", "value2"));
kvlist_append(list, kvpair_new("key3", "value3"));
// construct an iterator
kvlist_iterator_t* itor = kvlist_iterator_new(list);
while(true) {
  kvpair_t* pair = kvlist_iterator_next(itor);
  if(pair == NULL) {
    // `kvlist_iterator_next` returns `NULL` at the end of list
    break;
  }
  printf("key = %s, value = %s\n", pair->key, pair->value);
}
// cleanup
kvlist_iterator_free(&itor);
kvlist_free(&list); // will free the list and pairs
```

`hash.h`

Use `hash.h` to hash strings.

```
unsigned long hash(char *str);
```

`map_reduce` Structure

There are five phases in map_reduce as follows:

- Split Phase: Split the input list into num_mapper smaller lists.
- Map Phase: Spawn num_mapper threads and execute the provided map function.
- Shuffle Phase: Shuffle mapper results to num_reducer lists.
- Reduce Phase: Spawn num_reducer threads and execute the provided reduce function.
- Concatenate the resulting lists to get a single list.

Split Phase

In the split phase (also called the partition phase), you split the input list into `num_mapper` smaller lists so that each smaller list can be processed by different threads independently.

Map Phase

In the map phase, you create `num_mapper` threads. Each thread is responsible for a smaller list from the previous phase and calls the `mapper` function to obtain a new list.

Shuffle Phase

In the shuffle phase, you create `num_reducer` independent lists that can be processed in the next phase. Since you need to provide all pairs with the same key to the `reducer` function, the same key must be assigned to the same list.

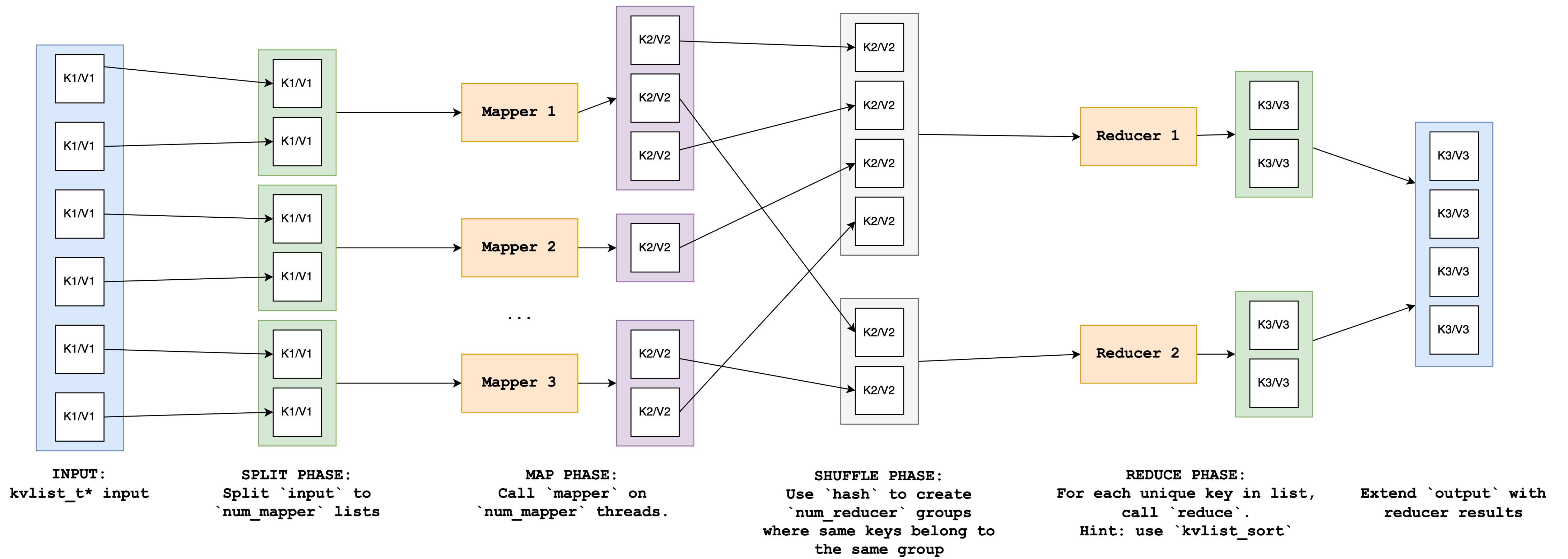
Reduce Phase

In the reduce phase, you create `num_reducer` threads. Each thread is responsible for a smaller list from the previous phase and calls the `reducer` function to aggregate results.

When calling `reducer`, you need to construct a list of all pairs with the same key. There are many ways to do this, but one way is to use the `kvlist_sort` function.

Output

You need to store the results in the `output` list passed as an argument. Use `kvlist_extend` to move pairs to `output`.


K1/V1
K1/V1
K1/V1
K1/V1
K1/V1
K1/V1
Mapper 1
Mapper 2
...
Mapper 3
K2/V2
Reducer 1
Reducer 2
K3/V3
INPUT:
kvlist_t* input
SPLIT PHASE:
Split `input` to
`num_mapper` lists
MAP PHASE:
Call `mapper` on
`num_mapper` threads.
SHUFFLE PHASE:
Use `hash` to create
`num_reducer` groups
where same keys belong to
the same group
REDUCE PHASE:
For each unique key in list,
call `reduce`.
Hint: use `kvlist_sort`
Extend `output` with
reducer results

Additional Functionality

In addition, your implementation must do the following:

- You should not have a `main` function in `mr.c`.
- `make` must create `mr.o`. We will use this object file to link your code with our test programs.
- Your code must use POSIX threads (`pthread.h`).
- Your code must not cause segfaults.
- All source files must be formatted using clang-format. Run `make format` to format `.c` and `.h` files.
- Your `map_reduce` must not leak memory. Use valgrind to check memory leaks.

Testing with `word-count`

`word-count` is a variant of the word-counting example from the previous section. It is invoked with three or more arguments: `word-count $NUM_MAPPER $NUM_REDUCER file ...`.

- `$NUM_MAPPER` is a positive integer that specifies the number of threads used for the `map` function.
- `$NUM_REDUCER` is a positive integer that specifies the number of threads used for the `reduce` function.
- `file ...` is one or more (text) files.

Suppose that you have a file `hello.txt` whose content is "hello, world!".

`pthread` API

> POSIX threads are a set of functions that support applications with requirements for multiple flows of control, called threads, within a process.

A couple of functions you might find useful:

- `pthread_create`: create threads
- `pthread_join`: to wait for the termination of the thread
- `pthread_mutex_init` / `pthread_mutex_destroy` to initialize/destroy mutex
- `pthread_lock` / `pthread_unlock` to lock/unlock mutex

Creating Threads

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
                   void* (*start_routine)(void*), void* arg);
```

Joining Threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Example: Create Thread

https://replit.com/@shumbo/cse-130-pthread-demo

```c
#include <stdio.h>
#include <pthread.h>

void* thread_fn(void* arg) {
  printf("hello from sub thread\n");
  return NULL;
}

int main(int argc, char** argv) {
  pthread_t t;
  pthread_create(&t, NULL, thread_fn, NULL);
  printf("hello from main thread\n");
  pthread_join(t, NULL);
}
```

Example: Pass value to threads

https://replit.com/@shumbo/cse-130-pthread-args

```
#include <stdio.h>
#include <pthread.h>

void* thread_fn(void* arg) {
  int num = *(int*)arg;
  printf("subthread received %d\n", num);
  return NULL;
}

int main(int argc, char** argv) {
  int num = 5;
  pthread_t t;
  pthread_create(&t, NULL, thread_fn, &num);
  printf("main thread passed %d\n", num);
  pthread_join(t, NULL);
}
```

Example: Data Race

https://replit.com/@shumbo/cse-130-race

```c
#include <stdio.h>
#include <pthread.h>

volatile int x;

void* increment_x(void* arg) {
  int num = *(int*)arg;
  for(int i = 0; i < num; i++) {
    x += 1;
  }
  return NULL;
}

int main(int argc, char** argv) {
  int num = 1000 * 1000 * 100;
  pthread_t t1, t2;
  pthread_create(&t1, NULL, increment_x, &num);
  pthread_create(&t2, NULL, increment_x, &num);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("%d * 2 = %d\n", num, x);
}
```

Example: `mutex`

https://replit.com/@shumbo/cse-130-mutex

```c
#include <stdio.h>
#include <pthread.h>

volatile int x;
pthread_mutex_t mutex;

void* increment_x(void* arg) {
  int num = *(int*)arg;
  for(int i = 0; i < num; i++) {
    pthread_mutex_lock(&mutex);
    x += 1;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main(int argc, char** argv) {
  int num = 1000 * 1000 * 100;
  pthread_mutex_init(&mutex, NULL);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, increment_x, &num);
  pthread_create(&t2, NULL, increment_x, &num);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_mutex_destroy(&mutex);
  printf("%d * 2 = %d\n", num, x);
}
```