

CMU 15-418/618: Parallel Computer Architecture and Programming Practice Exercise 6

Problem 1: Seeing How the Studying is Going

The night before the exam, Prof. Kayvon inspects the 15-418 web site logs to see if his students are studying. The site generates a log of all lecture slide comments, with the following format per line.

```
studentid  lecture_title  slide_number  comment
```

The log file is large (lots of studying! yay!), and is partitioned into 100 GB chunks, named `chunk_xxx.txt` stored in a shared file system accessible to all nodes on the `latedays` cluster. Below is Prof. Kayvon's correct, but inefficient, code to count the number of comments posted on a particular slide. Notice the program is written a very Spark-like style (notice pieces of code that mimic Spark's `load`, `map`, `filter`, and `count` operators generate arrays). **The code below is run by all cluster nodes.**

In the code below, assume `vector<T>` is an array of elements of type `T` that supports `size()` and `append(T)`

```
void barrier();           is a barrier across all nodes in the cluster
int  receiveInt(int src);  receives an integer from node src
int  sendInt(int dst, int value); sends an integer to node dst

// OPERATION 1: LOAD 100 GB FROM DISK //////////////////////////////////////

FILE* file = // open chunk_xxx.txt, where xxx is the id of the current node
vector<string> lines;
while (!end_of_file(file))
    lines.append( file.readLine() );

barrier(); // OPERATION 2: FILTER //////////////////////////////////////
           // Assume views is 30 GB after the filter (lots of long comments)

vector<string> views;
for (int i=0; i<lines.size(); i++) {
    string lecture_title;
    int    slide_number;
    parseLine(lines[i], &lecture_title, &slide_number); // extract title and slide number
    if (lecture_title == "In-Memory Distributed Computing using Spark" && slide_number == 28)
        views.append(lines[i]);
}

barrier();

// code continued on next page ...
```

```
// OPERATION 3: MAP: EXTRACT COMMENT ////////////////////////////////// (Assume comments is 20 GB)

vector<string> comments;
for (int i=0; i<views.size(); i++) {
    string comment;
    parseComment(views[i], &comment);
    comments.append(comment);
}

barrier(); // OPERATION 4: COUNT COMMENTS //////////////////////////////////

if (get_node_id() == 0) {
    int count = comments.size();
    for (int i=1; i<num_nodes; i++)
        count += receiveInt(i);
    printf("%d\n", count);
} else
    sendInt(0, comments.size());
```

- A. (4 pts) Prof. Kayvon runs the program on `latedays` and it immediately fails due to running out of memory. (`latedays` nodes only have 16 GB of RAM). One answer would be to store the multi-GB intermediate arrays to disk, but there's a much better solution. Please rewrite the program so that it runs efficiently on the cluster. (**Your implementation should not exceed a node's memory capacity, and it should avoid unnecessary synchronization.** Very rough pseudocode that indicates the order of computations is fine. No need to be detailed.)

```
int count;
while (!end_of_file) {
    vector<string> line = file.readline();
    String lecture_title;
    int slide_number;
    parse_line(line, &lecture_title, &slide_number);
    if (lecture_title == "In-Memory Distributed Computing using Spark" && slide_number == 28 )
        count++;
}
if (get_node_id() == 0) {
    ...
} else sendInt(0, count);
```

- B. (1 pt) What is the per-node memory consumption of your implementation assuming no line of the log exceeds 1 MB? (a simple estimate is fine)

line size: 1MB

int count size: 4B

String lecture_title size: 44B

int slide_number size: 4B

consumption = 1MB + 52B

C. (4 pts) Now Prof. Kayvon wants to process the log to produce a set of files on disk where each file corresponds to a **single slide** from the Spark lecture, and contains a list of comments for that slide. In other words, there will be files for: `slide_0.txt`, `slide_1.txt`, `slide_2.txt`, etc. Please assume:

- Each file is written entirely by **exactly one node in the cluster**, which is determined by the function `hashToNode(int x)` which returns the node responsible for writing `slide_x.txt`.
- The file is written in a single call to the function `writeFile(int slideNum, vector<string> comments)`.

For simplicity, you can assume that nodes now have an infinite amount of memory. Sketch an algorithm for executing this analysis operation on the cluster. Very brief pseudocode is desired (we don't want to read very detailed code), but we do want to know the details of what (if any) data nodes exchange, and about any barriers across nodes that are required in your implementation.

```
int num_of_works = 0;
int my_id = get_node_id();
int num_of_pages = get_num_pages("spark");
for (int i = 1; i < num_of_pages; i++) {
    if (hashToNode(i) == my_id)
        num_of_works++;
}

String[num_of_pages] comment_array;
while (!end_of_file) {
    String line = file.readline();
    String lecture_title;
    int slide_no;
    String comment;
    parse_line(line, &lecture_title, &slide_no, &comment);
    if (lecture_title == "spark")
        comment_array[slide_no].append(comment)
}

for (int i = 0; i < num_of_pages; i++) {
    int node_id = hashToNode(i);
    if (node_id != my_id) send_comment(node_id, i, comment_array[i]);
    else writeFile(i, comment_array[i]);
}

for (int i = 0; i < num_nodes; i++) {
    if (i != my_id) {
        while (num_of_works != 0) {
            int slide_no;
            String comment = recv_comment(i, &slide_no);
            writeFile(slide_no, comment);
            num_of_works--;
        }
    }
}
```

D. (1 pts) Contrast the amount of memory your implementation in part C uses compared to your implementation in part A. What is the fundamental reason for this difference? (in other words, why can't you achieve the footprint efficiencies observed before?)

Because there are multiple dependences exist in the view we produce, each node will produce many comments that belongs to other node, we need first save it and then send all the comments to the owner (decided by `hashToNode`), so we cannot just handle one line each time and then go to the next line, we need reuse the data.

Problem 2: Controlling DRAM

Consider a computer with a single DIMM containing eight 1 MB DRAM chips (8 MB total capacity). Each DRAM chip has one bank and row size 2 kilobits (256 bytes). As discussed in class, the DIMM is connected to the memory controller via a 64-bit bus, with 8-bits per cycle transferred from each chip.

Assume that:

- Contiguous 1 MB regions of the physical address space are mapped to a single DRAM chip. 256 consecutive physical address space bytes are in a row. 1048576 consecutive bytes fill a DRAM chip.
- Physical address 0 maps to chip 0, row 0, column 0. Physical address 1048576 maps to chip 1, row 0, column 0, etc.

Given these assumptions, reading a 64-byte cache line beginning at address X requires the following memory-controller logic, presented in C code below: (the data ends up in `cache_line`)

```
char cache_line[64];

// compute DRAM chip, row, col for address X
int chip = X / 1048576;
int row = (X % 1048576) / 256;
int col = (X % 1048576) % 256

for (int i=0; i<64; i++) {

    // Read one byte from each DRAM chip at given row and column (eight in total)
    // so that the byte from chip j ends up in 'from_dram[j]'. Assume necessary
    // DRAM row and column activations are performed inside DIMM_READ_FROM_CHIPS.

    char from_dram[8];
    DIMM_READ_FROM_CHIPS(row, column, from_dram);

    cache_line[i] = from_dram[chip];

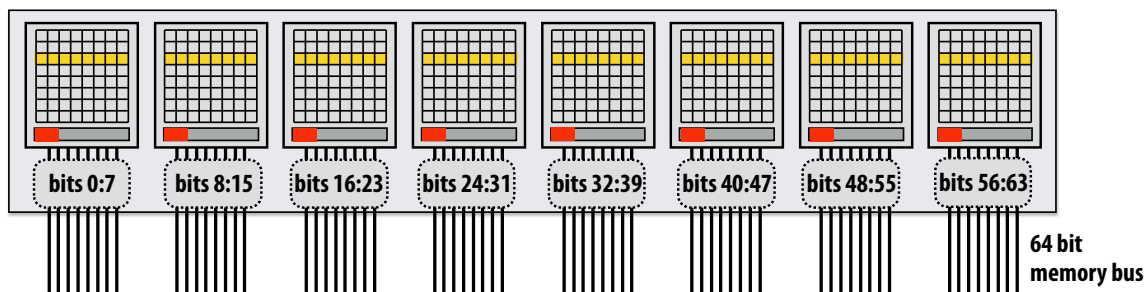
    column++; // move to next byte in column
}
```

Questions are on next page...

- A. (2 pts) Explain why 64 iterations (64 reads from the DRAM chips) are required to populate the buffer `cache_line`.

Because each iteration we only read 1 bytes from address X which owned by chip $X / 1048576$, so if we want read 64-byte cache line start from address X, we need read 64 times.

- B. (3 pts) Now assume the address space is **byte-interleaved across the DRAM chips** as discussed in class and shown in the Figure below. (Byte X in the address space is stored on chip $X \% 8$.) Please provide C-like pseudocode for reading the 64-byte cache line at address X from DRAM into `cache_line`. Your code should make a series of calls to `DIMM_READ_FROM_CHIPS`.



Hint: Recall that each DRAM chip row is 256 bytes.

```
char cache_line[64];
int cache_line_len = 0;
int row_start = x / 2048;
int row_end = (x + 64) / 2048;
int col_start = (x % 2048) / 8;
int col_end = ((x + 64) % 2048) / 8;
int chip_start = x % 8;
int chip_end = (x + 64) % 8;

for (int i_row = row_start; i_row < row_end; i_row++) {
    for (int i_col = col_start; i_col < col_end; i_col++) {
        char from_dram[8];
        DIMM_READ_FROM_CHIPS(i_row, i_col, from_dram);
        if (i_col == col_start) {
            memcpy(cache_line + cache_line_len, from_dram[chip_start], 8 - chip_start);
            cache_line_len += 8 - chip_start;
        } else if (i_col == col_end - 1) {
            memcpy(cache_line + cache_line_len, from_dram, chip_end);
            cache_line_len += chip_end;
        } else {
            memcpy(cache_line + cache_line_len, from_dram, 8);
            cache_line_len += 8;
        }
    }
}
```

- C. (2 pts) How much higher “effective bandwidth” is achieved using the interleaved mapping from part B than the original blocked mapping from part A?

When reading memory in part A only one memory bus is used by reading operation, however, part B utilizes all the buses of chip to fetch memory value.

- D. (3 pts) Imagine the byte interleaved memory system from part B is connected to a dual-core CPU. The memory controller uses a naive **round-robin policy** to schedule incoming memory requests from the cores (it services a request from core 0, then core 1, then core 0, etc.) All requests from the same core are processed in FIFO order.

Both cores execute the following C code **on different 4 MB arrays**. Simply put, each thread is linearly scanning through different regions of memory.

```
int A[N];           // let N = 1M, so this array is 4MB
int sum = 0;        // assume 'sum' is register allocated
for (int i=0; i<N; i++)
    sum += A[i];
```

Assume that the cores request data from memory at granularity of 8 bytes. On this system, you observe that when running two threads, the overall aggregate bandwidth from the memory system *is lower* than when one thread is executing the same code. Why might this be the case? (Hint: we are looking for an answer that pertains to DRAM chip behavior: consider locality, but which kind?)

For the single thread situation, since each time when thread reads from memory, it will at least get 8 bytes data, so when it read the next value, there will be a cache hit. However, if we have two threads interleave the memory, each time we read the first 4 bytes and then give bus to other thread, we will lose the next 4 bytes memory value. So, switching between two threads will affect the locality.