CS 61C MapReduce, Coherency, Atomics

Spring 2019 Discussion 12: April 17, 2019

1 MapReduce

For each problem below, write pseudocode to complete the implementations. Tips:

- The input to each MapReduce job is given by the signature of `map()`.
- `emit(key k, value v)` outputs the key-value pair `(k, v)`.
- `for var in list` can be used to iterate through `Iterables` or you can call the `hasNext()` and `next()` functions.
- Usable data types: `int`, `float`, `String`. You may also use lists and custom data types composed of the aforementioned types.
- `intersection(list1, list2)` returns a list of the intersection of `list1`, `list2`.

1.1 Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has.

Declare any custom data types here:

```
CoinPair:
    String person
    String coinType
```

```
map(________________, ________________):

map(String person, String coinType):
    key = (person, coinType)
    emit(key, 1)
```

```
reduce(________________, ________________):

reduce(CoinPair key, Iterable<int> values):
    total = 0
    for count in values:
        total += count
    emit(key, total)
```

1.2 Using the output of the first MapReduce, compute each person's amount of money. `valueOfCoin(String coinType)` returns a float corresponding to the dollar value of the coin.

```
map(________________, ________________):

map(CoinPair key, int amount):
    emit(key.person,
         valueOfCoin(key.coinType) * amount)
```

```
reduce(________________, ________________):

reduce(String key, Iterable<float> values):
    total = 0
    for amount in values:
        total += amount
    emit(key, total)
```

2 MapReduce/Spark Practice: Optimize Your GPA

2.1 Given the student's name and course taken, output their name and total GPA.

Declare any custom data types here:

```
CourseData:
    int courseID
    float studentGrade // a number from 0-4
```

```
map(________________, ________________):

map(String student, CourseData value):
    emit(student, value.studentGrade)
```

```
reduce(________________, ________________):

reduce(String key, Iterable<float> values):
    totalPts = 0
    totalClasses = 0
    for grade in values:
        totalPts += grade
        totalClasses += 1
    emit(key, totalPts / totalClasses)
```

3 Coherency and Atomic Operations

The benefits of multi-threading programming come only after you understand concurrency. Here are two of the most common concurrency issues:

1. **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. This can often be solved by bypassing the cache and writing directly to memory, i.e. using volatile keywords in many languages.

2. **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-threading programming, the **interleaving** of R, M, W stages often produces a lot of issues.

In order to solve the problems created by Read-modify-write, we have to rely on the idea of **uninterrupted execution**, also known as atomic execution.

In RISC-V, we have two categories of atomic instructions:

1. **Load-reserve, store-conditional**: allows us to have uninterrupted execution across multiple instructions

2. **Amo.swap**: allows for uninterrupted memory operations within a single instruction

Both of these can be used to achieve atomic primitives. Here are examples for each:

```
Test-and-set

Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

```
Compare-and-swap

# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0
    otherwise
cas:
    lr.w t0, (a0) # Load original value.
    bne t0, a1, fail # Doesnt match, so fail.
    sc.w a0, a2, (a0) # Try to update.
    jr ra # Return.
fail:
    li a0, 1 # Set return to failure.
    jr ra # Return.
```

Instruction definitions:

1. **L**oad-**r**eserve: Loads the four bytes at `M[R[rs1]]`, writes them to `R[rd]`, sign-extending the result and registers a reservation on that word in memory.
2. **S**tore-**c**onditional **rd, rs2, (rs1)**: Stores the four bytes in register `R[rs2]` to `M[R[rs1]]`, provided there exists a load reservation on that memory address. Writes 0 to `R[rd]` if the store succeeded, or a nonzero error code otherwise.
3. **Amoswap rd, rs2, (rs1)**: Atomically, puts the sign-extended word located at `M[R[rs1]]` into `R[rd]` and puts `R[rs2]` into `M[R[rs1]]`.

Explanations for both methodologies:

1. **Test-and-set**: We have a lock stored at the address specified by `a0`. We utilize `amoswap` to put in 1 and get the old value. If the old value was a 1, we would not have changed the value of the lock and we will realize that someone currently has the lock. If the old value was a 0, we will have just ”locked” the lock and can continue with the critical section. When we are done, we put a 0 back into the lock to ”unlock” it.
2. **Compare-and-swap**: CAS tries to first reserve the memory and gets the value stored and compares it to the expected value. If the expected value and the value that was stored do not match, the entire process fails and we must restart to update based on the new information. Otherwise, we register a reservation on the memory and try to store the new value. If the exit code is nonzero, something went wrong with the store and we must retry the entire LR/SC process. Otherwise with a zero exit code, we continue into the critical section, then release the lock.

3.1 Why do we need special instructions for these operations? Why cant we use normal load and store for `lr` and `sc`? Why cant we expand `amoswap` to a normal load and store?

For lr and sc, after lr, other threads cannot write to the location marked reserve, hence the value loaded from memory (a3 in the above example) will be unchanged between lr and sc. For amoswap, it does load and store in one single CPU cycle, hence the operation is atomic and uninterruptable.

3.2 Now that we have atomic operations, let's try to experiment with them. Let us try to implement an algorithm that enforces ordered thread execution. This means that if we have four threads, thread 0 goes first, thread 1 goes next, etc. For this problem assume that a1 holds the location of a piece of memory we have access to for the entire duration of our algorithm. Also, we can assume there exists a label get_thread_num that returns the thread's number in a0. Try to fill in the blanks below. Please use LR/SC for this problem:

```
        addi t0, x0, 0

        ________________________________ # Setup for the first (0-th) thread
        ...
        # Assume we now spawn 4 threads in this code
        ...

Check: jal ____________________________ # Get the current thread number
                                        # Get the ID of the next thread that should operate
        ________________________________ # (make sure this can't get interfered with)


        ________________________________
Done:   addi t0, t0, 1

        ________________________________ # Set which thread is next to run

        bne ____________________________
        ...
        # Assume we now join the 4 threads in this code
        ...
        jr ra
```

```
        addi t0, x0, 0
        sw t0, 0(a1)
        ...
        # Assume we now spawn 4 threads in this code
        ...
Check: jal ra, get_thread_num
        lr.w t0, (a1)
        bne a0, t0, Check
Done:   addi t0, t0, 1
        sc.w a0, t0, (a1)
        bne a0, x0, Check
        ...
        # Assume we now join the 4 threads in this code
        ...
        jr ra
```