Programming Principles

Midterm Solution

2016

Exercise 1: Associativity (25 points)

For the this exercise, you will look at multiplication of 2 by 2 matrices containing 32-bits integers. Remember that multiplication of such matrices is defined as follows:

$$AB = C \iff \forall\ 1 \leq i \leq 2, 1 \leq k \leq 2.\ C_{ik} = \sum_{j=1}^{2} A_{ij} \cdot B_{jk}$$

Also note that 32-bits integers form a commutative ring, meaning that both addition and multiplication are associative and commutative, and that multiplication distributes over addition.

Question 1: Definition (5 points)

State the associative property for matrix multiplication.

Answer

$$(A_{m\times n}B_{n\times p})C_{p\times q} = A_{m\times n}(B_{n\times p}C_{p\times q})$$

For all $A_{m\times n}$, $B_{n\times p}$ and $C_{p\times q}$ matrices.

Question 2: Proof of associativity (15 points)

Prove that multiplication of 32-bits integer matrices of dimension 2 by 2 is associative.

Hint: To show that two matrices X and Y are equal, you may show that for all indexes i and j that X_{ij} and Y_{ij} are equal.

Answer

Let A, B and C be arbritrary 32-bits integer matrices of dimension 2 by 2. Let $i \in [1, 2]$, $l \in [1, 2]$ be arbitrary indexes.

We show that $((AB)C)_{il} = (A(BC))_{il}$.

$$
\begin{aligned}
((AB)C)_{il} &= \sum_{k=1}^{2} (AB)_{ik} \cdot C_{kl} && \text{Definition of matrix multiplication} \\
&= \sum_{k=1}^{2} (\sum_{j=1}^{2} (A_{ij} \cdot B_{jk})) \cdot C_{kl} && \text{Definition of matrix multiplication} \\
&= \sum_{k=1}^{2} \sum_{j=1}^{2} ((A_{ij} \cdot B_{jk}) \cdot C_{kl}) && \text{Distributivity of multiplication} \\
&= \sum_{j=1}^{2} \sum_{k=1}^{2} ((A_{ij} \cdot B_{jk}) \cdot C_{kl}) && \text{Associativity and commutativity of addition} \\
&= \sum_{j=1}^{2} \sum_{k=1}^{2} (A_{ij} \cdot (B_{jk} \cdot C_{kl})) && \text{Associativity of multiplication} \\
&= \sum_{j=1}^{2} A_{ij} \cdot \sum_{k=1}^{2} (B_{jk} \cdot C_{kl}) && \text{Distributivity of multiplication} \\
&= \sum_{j=1}^{2} A_{ij} \cdot (BC)_{jl} && \text{Definition of matrix multiplication} \\
&= (A(BC))_{il} && \text{Definition of matrix multiplication}
\end{aligned}
$$

QED

Question 3: Associativity in parallel programming (5 points)

Why is associativity interesting in the context of parallel programming?

1. Associative operations always have efficient lock-free implementations.
2. The order of parameters of an associative operation can be freely swapped, giving better latency.
3. **Associativity opens up possibilities to balance work.**
4. Associative operations are more efficiently executed on Intel processors.

Exercise 2: Parallelizing Fold with Minus (25 points)

Let `a` be an `Array[Int]`. We are interested in the value of `acc` computed by the following piece of code:

```
var acc = 0
for(i <- (a.length - 1) to 0 by (-1))
  acc = a(i) - acc
```

For example, with `a = Array(4, 3, 6, 2)` we obtain `acc = 4-(3-(6-(2-0))) = 5`.

Question 1 (10 points)

Complete the template shown below so that the value of `acc` returned by the template is same as that computed by the above code snippet. For you reference, `a.zipWithIndex` returns a new array consisting of all elements of `a` paired with their index. Eg. `Array(1, 2, 5).zipWithIndex = Array((1, 0), (2, 1), (5, 2))`

```
val acc = a.zipWithIndex
  .map{ case (value, index) =>
     if(index % 2 == 0) value
     else -value
  }.sum
```

Question 2 (10 points)

We are now interested by a more efficient way of computing `acc`, which combines the two traversals of the array and make the traversal in parallel. For that, we will use the `parallel` construct we saw in the lectures. For your reference, you will below an implementation of the `parallel` construct. Note that `t = Task { op }` creates a new thread `t` to execute `op`, and `t.join()` waits and returns the result of `op` when `t` finishes.

```
def parallel[A, B](op1: =>A, op2: =>B): (A, B) = {
  val res1 = Task { op1 }
  val res2 = op2
  (res1.join(), res2)
}
```

Complete the code snippet shown below so that `computeAcc(a, 0, a.length)` returns the value of `acc`. The function may recursively call itself on an array segment between indices `start` (inclusive) and `end` (exclusive). Some input/output examples: `computeAcc(Array(1,2,7,13,5), 0, 5) = -2` `computeAcc(Array(1,2,7,13,5), 1, 4) = -8`

```
def computeAcc(a: Array[Int], start: Int, end: Int): Int = {
  if(end - start == 1) {
    if(start % 2 == 0) a[start]
    else -a[start]
  } else {
    val mid = (start + end) / 2
    val (lacc, racc) = parallel(computeAcc(a, start, mid), computeAcc(a, mid, end))
    lacc + racc
  }
}
```

Question 3 Suppose we call `computeAcc` with the following arguments:

```
computeAcc(Array(112, 97, 114, 97, 108, 108, 101, 108), 0, 8)
```

How many `Task`s will be created when the above described implementation of `parallel` is used?

Number of tasks = 7. In general, n -1 tasks will be created for a array of size `n`. (Remember the there is no threshold in this setting). The number of tasks created is same as the number of internal nodes in a complete binary tree with `n` leaves, where leaves correspond to the case `end - start == 1`.

Exercise 3: Parallel Quick Sort (25 points)

In this exercise, you are required to design and analyze a parallel version of the quick sort algorithm. Consider the function `qsort` shown below that implements a *sequential* quick sort algorithm for sorting (in ascending order) a *list* of `Int`.

```
def qsort(list: List[Int]): List[Int] = {
  if (list.size <= 1) list
  else {
    val (smaller, eqs, bigger) = partition(list, choosePivot(list))
    qsort(smaller) ++ eqs ++ qsort(bigger)
  }
}
```

A brief explanation of the quick sort algorithm

(a) If the input list has zero or one element, the algorithm returns the list as it is.

(b) Otherwise, the algorithm chooses an element of the input list referred to as the *pivot*.

(c) It then partitions the input list into three parts such that the first part (`smaller`) contains all elements smaller than the pivot, the second part (`eqs`) contains all elements equal to the pivot, and the last part (`bigger`) contains all elements greater than the pivot. Note that the size of the parts may depend on the choice of the pivot, and may not necessarily be equal.

(d) Finally, the algorithm recursively sorts the first and the last parts, and concatenates the results.

If the *pivot* is chosen arbitrarily, the *depth* of the `qsort` algorithm is $O(n^2)$ in the worst case, for a list of size n. Now say, instead of arbitrarily choosing the pivot, we always choose the *median* of the list as the pivot. The median of a list of integers is the middle element in the sorted order of the integers. We can compute the *median* of an arbitrary list `l` (of size n) in worst case *linear time*: *O(n)* by using an algorithm called *median of medians*. Note that when the median is chosen as the pivot, at most $\lfloor \frac{(n-1)}{2} \rfloor$ elements are smaller than the pivot, and at most $\lceil \frac{(n-1)}{2} \rceil$ elements are greater than the pivot.

Question 1 (15 points) Show a parallel implementation of the `qsort` function whose *depth* is *O(l.size)*, given that `choosePivot` uses the median. You can use the `parallel` construct explained in the previous question. You don't have to show the implementations of the `partition` or the `choosePivot` functions.

```
def qsort(list: List[Int]): List[Int] = {
  if (list.size <= 1) list
  else {
    val (smaller, eqs, bigger) = partition(list, choosePivot(list))
    val (lres, rres) = parallel(qsort(smaller), qsort(bigger))
    lres ++ eqs ++ rres
  }
}
```

Question 2 (10 points) Prove that the depth of the parallel `qsort` function is *O(l.size)*. Note that for computing the depth of `qsort` you need to come up with a sutiable *depth* for the `partition` function. You need not prove the *depth* of the `partition`, or `choosePivot` functions.

For pedagogical reasons, we provide a detailed formal proof below. Let a, b, and c denote some positive constants.

if list.size <= 1, $D(qsort(list)) = a$

Otherwise, $D(qsort(list)) = D(choosePivot(list))+D(parition(list, \cdots))+D(parallel(qsort(smaller), qsort(bigger))+ Depth(lres + +eqs + +res)$.

The functions `choosePivot`, `partition` and list concatenation can be implemented in time linear in the size of their arguments. Each of these operations are invoked with lists of size less than `list.size`, which implies that their depth is upper bounded by $O(list.size)$. Note that depth is always less than or equal to the sequential execution time. Therefore,

$$\begin{aligned} D(qsort(l)) &\leq b * l.size + c + D(parallel(qsort(smaller), qsort(bigger)) \\ &= b * l.size + c + max(D(qsort(smaller), qsort(bigger)))) \qquad \text{by the definition of depth of the parallel construct} \end{aligned}$$

Let $D(qsort(l))$ be denoted using $D(n)$ where $n = l.size$.

$$\begin{aligned} D(n) &\leq b * n + c + max(D(\lfloor \frac{(n-1)}{2} \rfloor), D(\lceil \frac{(n-1)}{2} \rceil)) && \text{since the pivot is the median} \\ &\leq b * n + c + D(\frac{n}{2}) && \text{because depth, like time, is monotonously increasing function.} \end{aligned}$$

The above recurrence can be solved in many ways e.g. using Master's Theorem. However, here, let us use induction. Induction Hypothesis: $D(n) <= 2(b + c) * n + a$. This satisfies the base case $n <= 1$ where $D(1) = a$. Consider the case $n > 1$.

$$\begin{aligned} D(n) &\leq b * n + c + D(\frac{n}{2}) \\ &\leq b * n + c + 2(b + c) * \frac{n}{2} + a && \text{By inductive hypothesis.} \\ &\leq 2(b + c) * n && \text{since } n \geq 2 \end{aligned}$$

QED

Exercise 4: Read/Write Lock (25 points)

In this question, you have to implement a class `ReadWrite` that implements a *read/write* lock. Unlike a traditional monitor that has a single operation to build mutual exclusion blocks (`synchronized`), read/write locks have two methods:

```
read(op)  // performs ‘op‘ as a reader
write(op)  // performs ‘op‘ as a writer
```

The `read` and `write` methods should satisfy the following constraints:

- Only one writer may own the lock at any point in time. That is, no two `write` operations can execute concurrently.
- Multiple readers can own the lock concurrently. That is, mutiple `read` operations can happen concurrently.
- A writer may own the lock only if no reader owns it. That is, a `write` operation can happen only if there are no concurrent `read` operations.

Question 1 (15 points)

Implement both `read` and `write` in the class stub on the *next page* and write a small explanation about it just below: (declare any vars you'd need at the begining of the class). You may use `synchronized`, `wait`, and `notifyAll` methods to implement the `read` and `write` methods.

Explain your solution in at most six sentences here

Hints:

- Since multiple concurrent reads should be allowed to happen, it is important that the `op` operation in `read` is itself executed outside of a synchronized block.
- You may want to keep track of the number of reader executing at the same time.

Answer

```
class ReadWrite extends Monitor {
  var readers : Int = 0

  def read[T](op: => T): T = {
    synchronized {
      readers += 1
    }
    try {
      op
    }
    finally {
      synchronized {
        readers -= 1
        if (readers == 0)
          notifyAll()
      }
    }
  }

  def write[T](op: => Unit) = synchronized {
    while (readers > 0) {
      wait()
    }
    op
  }
}
```

Question 2 (10 points) In most real world application, reads are far more frequent than writes; the lock implemented in *question 1* would be highly inefficient as reads may block writes from happening.

Implement a read/write lock, based on the previous question's implementation, ensuring that any write operation that entered the synchronized block (i.e. that own the lock) takes precedence over any new reader. That is, if a writer has entered the synchronized block of a `write` no new reader is permitted to enter a `read`, but existing readers are allowed to complete.

Answer

```
class ReadWrite extends Monitor {

  var pendingWriters: Int = 0
  var readers: Int = 0

  def read[T](op: => T): T = {
    synchronized {
      while(pendingWriters > 0) {
        wait()
      }
      readers += 1
    }
    try {
      op
    }
    finally {
      synchronized {
        readers -= 1
        if (readers == 0)
          notifyAll()
      }
    }
  }

  def write[T](op: => Unit): Unit = synchronized {
    pendingWriters += 1
    while (readers > 0) {
      wait()
    }
    try {
      op
    }
    finally {
      pendingWriters -= 1
      if (pendingWriters == 0)
        notifyAll()
    }
  }
}
```