Midterm #2

CMSC 433: Programming Language Technologies and Paradigms

November 20, 2013

Name _____

Instructions

Do not start until told to do so!

- This exam has 10 double-sided pages (including this one); make sure you have them all
- You have 75 minutes to complete the exam
- The exam is worth 100 points. Allocate your time wisely: some hard questions are worth only a few points, and some easy questions are worth a lot of points.
- If you have a question, please raise your hand and wait for the instructor.
- In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Question	Points	Score
1	Definitions	20	
2	Short Answer	14	
3	Concurrency Concepts	16	
4	Erlang	30	
5	Parallelization	20	
	Total	100	

1. Definitions (20 points total, 2 points each).

   Next to each term on the left, write the letter on the right that corresponds to the definition of the term, or a true statement about it.

	(a) The number of operations processed per unit time
____ Data parallel	(b) An example would be computing total counts of words in a document in parallel
____ Task parallel	(c) Code implementing it often uses lock-based synchronization
____ Pipeline parallel	(d) An action involving an object in which some methods are only valid in certain states
	(e) A synonym for deadlock
____ Latency	(f) Predicts a task's possible speedup based on the number of processors and the amount of strictly-serial work
____ Throughput	(g) Predicts how many threads you should create based on the number of processors and the ratio of wait time to compute time
____ State-dependent action	(h) The time to complete a single operation
	(i) A lack of forward progress despite being able to run
____ Optimistic concurrency	(j) A kind of Executor that always has a fixed number of threads
	(k) An example would be adding 1 to every element of an array in parallel
____ Starvation	(l) Code implementing it often uses compare-and-swap operations
____ Thread pool	(m) An action involving an object implementing a finite state machine
____ Amdahl's Law	(n) A pattern that separates the creation of tasks from the assignment to threads for execution
	(o) An example would be computing the average and the median of an array in parallel

Answer:

(k), (o), (b), (h), (a), (d), (l), (i), (n), (f)

2. Short answer (14 points)

(a) (2 points) Does Erlang use static typing or dynamic typing?

Answer:

Dynamic typing

(b) (2 points) True or false: It is possible that a program using a thread pool that deadlock if the pool has too few threads allocated to it.

Answer:

True.

(c) (5 points) What is the difference between a task and a thread?

Answer:

A task is a conceptual unit of work, while a thread is a processing resource that can be used to run tasks.

(d) (5 points) Is it possible to have a data race in Erlang in the same way as in a Java program? Why or why not?

Answer:

No, because in Erlang data is immutable. Instead, using the server behavior (among other patterns) we can have atomicity violations that resemble data races.

3. Concurrency/parallelism concepts in Java (16 points)

   (a) (6 points) Consider the following interface definition:

```
public interface MyPair<T> {
    // Assign value to first/second element of pair
    void putFirst(T value);
    void putSecond(T value);

    // Retrieve first/second values; else throw exception if no value exists
    T getFirst() throws java.util.NoSuchElementException;
    T getSecond() throws java.util.NoSuchElementException;

    // Get first/second values, or else wait til there if not present
    T waitForFirst();
    T waitForSecond();
}
```

   Based on the type and description of each method, write **B**, **G**, or **N** to its left, based on whether it implements *balking*, *guarding* (whether by suspension or busywaiting), or *neither*, respectively.

   Answer:

   N, N, B, B, G, G

(b) (5 points) The following code uses fork/join to compute the sum of an array in parallel. Unfortunately, it is unlikely to perform well as the number of processors increases. Briefly explain why and suggest how you would fix the problem.

```
public class SumSolver extends RecursiveAction {
  private final static int THRESHOLD = 2;
  private static int[] array;
  private final int start, end;
  private int result;

  private SumSolver(int start, int end) {
      this.start = start;
      this.end = end;
  }

  private void compute() {
    if ((end-start) < THRESHOLD) {
      for (int i = start; i<end; i++)
        result = result+array[i];
    }
    else {
      int m = (end-start) / 2;
      SumSolver left, right;
      left = new SumSolver(start, end-m);
      right = new SumSolver(end-m,end);
      invokeAll(left,right);
      result = left.result + right.result;
    }
  }

  public static int sum(int[] array) {
    int nThreads = 9;
    int arrSz = array.length;
    ForkJoinPool pool = new ForkJoinPool(nThreads);
    SumSolver solver = new SumSolver(0,arrSz);
    solver.array = array;
    pool.invoke(solver);
    return solver.result;
  }
}
```

Answer:

There are two problems. First, the number of threads is fixed. This should be changed to related to the number of processors. Second, the threshold is too small. We should change this code to increase `THRESHOLD` *based on experimental evidence, so that the cost of further subdivision of the tasks does not overwhelm the benefits of parallelism.*

(c) (5 points) Consider the following static method, which computes a map from the number of occurrences of an element in the given array `arr`, returning the result as a `HashMap`. As an example, given the array `{1,1,4,7,1}` as input, this method would return the `HashMap` mapping `1` to `3`, `4` to `1`, and `7` to `1`.

Unfortunately, this code is unlikely to perform well, even as we increase the number of processors. Briefly explain why, and suggest how you might fix it.

```
public static HashMap<Integer,Integer> countEm(final int arr[]) {
  final HashMap<Integer,Integer> counts = new HashMap<Integer,Integer>(11);
  int nProc = Runtime.getRuntime().availableProcessors();
  int sz = arr.length / nProc;
  ExecutorService ex = Executors.newFixedThreadPool(nProc);
  for (int i = 0; i<nProc; i++) {
    final int start = i*sz;
    final int end = i == (nProc-1) ? arr.length : (i+1)*sz;
    Runnable r = new Runnable() {
            public void run() {
              for (int j = start; j<end; j++) {
                synchronized (counts) {
                  Integer n = counts.get(arr[j]);
                  if (n != null) counts.put(arr[j],(int)n+1);
                  else           counts.put(arr[j],1);
                }
              }
            }
        };
    ex.execute(r);
  }
  ex.shutdown();
  ex.awaitTermination(...);
  return counts;
}
```

Answer:

The issue is that all of the threads are synchronizing on the hashmap, which limits the amount of parallelism: almost all of the work is within the synchronized block. To fix this you should use a ConcurrentHashMap or some other form of optimistic concurrency, essentially using `compareAndSet` *to perform the update.*

4. (Erlang, 30 points)

Look at each of the Erlang programs below. Along with each program, we will provide a call; say what the call will do. If it returns, indicate the value returned; if it fails with some exception, indicate the exception; or if it does not return, say so. Explain your reasoning if you hope for partial credit. You may assume all of the programs compile.

(a) (6 points) What will `go(4)` do?

```
go(0) -> false;
go(1) -> true;
go(X) -> go(X-2).
```

Answer:

Returns false.

(b) (5 points) What will `go([1,2],3)` do?

```
go([],N) -> N;
go([_H|T],M) -> M + go(T,M).
```

Answer:

Returns 9.

(c) (5 points) What will `go(self())` do?

```
go(Pid) ->
  Pid ! { one, 1 },
  receive
    { one, M } -> M
  end.
```

Answer:

Returns 1.

(d) (5 points) What will `go(3)` do?

```
go(N) ->
  go(N,self()),
  receive X -> X end.

go(N,Pid) when N > 0 ->
  Qid = spawn(fun () ->
                receive X -> X end,
                go(N-1,Pid)
              end),
  Qid ! 2;
go(0,Pid) ->
  Pid ! 0.
```

Answer:

Returns 0.

Next page ...

The next three questions concern the following code. Look at the calls in each question, and trace through the code to see what happens.

```
box_loop(Box) ->
  receive
    {From, boxget} ->
        case Box of
          [H|T] -> From ! {self(), H}, box_loop(T);
          _Else -> From ! {self(), failed}, box_loop(Box)
        end;
    {boxput, P} ->
        box_loop([P|Box]);
    reset -> box_loop([])
  end.

% API code
make_box() ->
  spawn(fun() -> box_loop([]) end).

boxget(Box) ->
  Box ! {self(), boxget},
  receive
    {Box, N} -> N
  end.

boxput(Box,N) -> Box ! {boxput, N}.

reset(Box) -> Box ! reset.
```

(e) (3 points) What will the following code do?

```
P = make_box(), boxput(P,1), boxget(P).
```

Answer:

Returns 1.

(f) (3 points) What will the following code do?

```
P = make_box(), boxput(P,1), reset(P), boxput(P,2), boxput(P,3),
boxget(P).
```

Answer:

Returns 3.

(g) (3 points) What will the following code do?

```
P = make_box(), boxput(P,1), reset(P),
boxget(P).
```

Answer:

Returns failed.

5. Coding parallelization (20 points)

The method `divisibles` given below computes how many numbers in the first array are divisible by numbers given in the second array. For example, suppose we have the following:

```
int v[] = {2,4,7,8,100,345,18,27,45,67,37,23};
int divs[] = {2,3,4};
```

Then calling `divisibles(v,divs)` will return the array `{5,4,3}` since five numbers in `v[]` are divisible by 2, four are divisible by 3, and three are divisible by 4.

```
public class DoWork {
  public static boolean isDivisibleBy(int v, int divisor) {
    int x = v / divisor;
    return (x * divisor == v);
  }
  public static int nDivisibleBy(int[] v, int start, int end, int divisor) {
    int cnt = 0;
    for (int i = start; i<end; i++) {
      if (isDivisibleBy(v[i],divisor))
        cnt++;
    }
    return cnt;
  }
  public static int[] divisibles(int[] v, int[] divisors) {
    int[] counts = new int[divisors.length];
    for (int i = 0; i<divisors.length; i++) {
        counts[i] = nDivisibleBy(v,0,v.length,divisors[i]);
    }
    return counts;
  }
}
```

On the next page, implement a parallelized version of the `divisibles` static method (without changing any of the code in the other two methods). Your algorithm does not need to be optimal but should be expected to beat a single-threaded version for sufficiently large input arrays. (If you like, point out the circumstances you expect your algorithm to do well in.)

Please use real code to the extent possible, but pseudocode if you must. You can define helper methods/classes if you like. Refer to other questions on the exam for API calls you can use when implementing this method. You may also find the following API calls useful:

```
ExecutorService:  <T> Future<T> submit(Callable<T> task)
Future<T>:  T get()
Callable<V>:  V call()
```

The first `ExecutorService` method submits a task for execution and returns a `Future` representing the pending results of that task. The second `Future` method waits for the computation to complete, then returns the result. The last is the method that actually computes a `Callable` result.

(Show your work here.)

Answer:

Here is a sample solution. It is naive in parallelizing based on the number of divisors. You could imagine using fork/join to subdivide further, considering parts of each source array in parallel, for a single divisor.

```
public static int[] divisibles(final int[] v, final int[] divisors) throws Exception {
  int[] counts = new int[divisors.length];
  int nProc = Runtime.getRuntime().availableProcessors();
  ExecutorService ex = Executors.newFixedThreadPool(nProc);
  Future[] results = new Future[divisors.length];
  for (int i = 0; i<divisors.length; i++) {
    final int idx = i;
    results[i] = ex.submit(new Callable<Integer>() {
            public Integer call() {
                return (Integer)nDivisibleBy(v,0,v.length,divisors[idx]);
            }
        });
  }
  for (int i = 0; i<divisors.length; i++) {
    counts[i] = (Integer)results[i].get();
  }
  ex.shutdown();
  return counts;
}
```