1 Exam sample

1. Consider the following code for enqueueing a continuation on a future (the `set()` function is guaranteed to be called exactly once by the user code):

```
1  template<typename T>
2  class Future {
3      list<function<void(T)> > continuations;
4      T val;
5      bool hasValue;
6      mutex mtx;
7  public:
8      Future() :hasValue(false) {}
9      void set(T v) {
10         val = v;
11         hasValue = true;
12         unique_lock<mutex> lck(mtx);
13         for(function<void(T)>& f : continuations) {
14             f(v);
15         }
16         continuations.clear();
17     }
18     void addContinuation(function<void(T)> f) {
19         if(hasValue) {
20             f(val);
21         } else {
22             unique_lock<mutex> lck(mtx);
23             continuations.push_back(f);
24         }
25     }
26 };
```

What kind of concurrency issue does it present?

A: a call to `set()` can deadlock if simultaneous with the call to `addContinuation()`;

B: two simultaneous calls to `addContinuation()` may deadlock;

C: two simultaneous calls to `addContinuation()` may lead to a corrupted `continuations` vector;

D: simultaneous calls to `addContinuation()` and `set()` may lead to continuations that are never executed;

E: simultaneous calls to `addContinuation()` and `set()` may lead to continuations that are executed twice;

F: to solve all concurrency issues, one has to move the content of line 12 to between lines 9 and 10;

G: to solve all concurrency issues, one has to move the content of line 12 to between lines 13 and 14;

H: to solve all concurrency issues, one has to move the content of line 22 to between lines 18 and 19;

I: to solve all concurrency issues, one has to move both the content of line 12 to between lines 9 and 10 and the content of line 22 to between lines 18 and 19;

J: to solve all concurrency issues, one has to move both the content of line 12 to between lines 13 and 14 and the content of line 22 to between lines 18 and 19;

2. Consider the following code for computing the scalar product of two vectors:

```
1  int scalarProduct(vector<int> const& a,
2      vector<int> const& b,
3      int nrThreads)
4  {
5      int result = 0;
6      vector<thread> threads;
7      threads.reserve(nrThreads);
8      int begin = 0;
9      for(int i=0 ; i<nrThreads ; ++i) {
10         int end = begin + a.size()/nrThreads;
11         threads.emplace_back([&a, &b, begin, end, &result]() {
12             int part = 0;
13             for(int i=begin ; i<end ; ++i) {
14                 part += a[i]*b[i];
15             result += part;
16         });
17         begin = end;
18     }
19     for(int i=0 ; i<nrThreads ; ++i) {
20         result += threads[i].get();
21     }
22     return result;
23 }
```

What kind of issues does it have? How to solve them?

A: one thread has much more multiplications to perform (in line 14) than the others;

B: one thread has much fewer multiplications to perform (in line 14) than the others;

C: some of the terms are not added into the final sum;

D: some of the terms are added twice or more times into the final sum;

E: there are some concurrency issues requiring the use of mutexes or atomic variables;

F: the parallelism doesn't work because each thread is started only after the previous one ends;

G: to solve issues A–D, one could replace line 10 with `int end = begin + (a.size()+nrThreads-1)/nrThreads`;

H: to solve issues A–D, one could replace line 10 with `int end = begin + (a.size()-begin)/(nrThreads-i)`;

I: to solve issues A–D, one could replace line 10 with `int end = i*a.size()/nrThreads`;

J: to solve issues A–D, one could replace line 10 with `int end = (i+1)*a.size()/nrThreads`;

3: We want a distributed program that computes the prime numbers up to a number `maxN`, using MPI.

You shall implement two functions (in C++, Java or C#):

```
vector<int> primes(int maxN, int nrProcs,
        vector<int> const& primesToSqrtN);
```

that will run on the process 0 of the `MPI_COMM_WORLD` communicator, where `maxN` is the value up to which the primes are to be generated, `nrProcs` is the number of processes in the world communicator, and `primesToSqrtN` contains the list of (already generated) primes up to the square root of `maxN`; it can be used to check if a candidate number is prime or not by testing if it is divisible to a number on this list.

```
void worker(int myId, int nrProcs);
```

will run on all other processesin the world communicator, with `myId` representing the rank and `nrProcs` the number of processes.

2 Answers

1. Answer: D, I

   Explaination (not to be written at the exam):

   There is no deadlock since there is a single mutex and no other blocking call — therefore, A and B are false.

   The `continuations` vector is always handled with the mutex locked — so, C is also false.

   Since `hasValue` is set and checked without taking the mutex, race conditions are possible involving it. If `addContinuation()` sees it as `false`, but `set()` executes the already enqueued continuations before `addContinuation()` adds the one given to it as an argument, then that continuation is never executed. So, D is true.

   On the other hand, `addContinuation()` either executes the continuation or adds it to the continuations list, and the continuation list is executed only once. So, E is false.

   We need to make setting or checking of `hasValue` atomic with the corresponding operation on the `continuations` vector. So, we need to lock the mutex before setting or checking `hasValue` — that is, the answer I is correct.

2. Answer: C, E, H, J.

   Explaination (not to be written at the exam):

   As implemented, every thread executes `a.size()/nrThreads` multiplications, so A and B are both false.

   Since the beginning of the range processed by one thread is the end of the range for the next one, with no overlap and nothing is skipped over. So, D is false. However, the last `a.size()%nrThreads` elements are never processed, so, C is true.

   There are 3 variables shared between threads: `a`, `b`, and `result`. The first two are read-only, so, no concurrency issue exists with them. However, `result` is used in read-compute-write operations in line 15 by all threads. Without it being atomic or protected by a mutex, this is not correct, so, E is true.

   The threads are all created by the cycle in lines 9–18 and they are then waited for in the cycle in lines 19–21. So, all threads can execute in parallel, therefore, F is false.

Out of the solutions candidates G–J, we have: G can process elements beyond the end of the vector (if `a.size()` is not a multiple pf `nrThreads`); H gives the next thread the remaining elements divided to the remaining threads and is correct; I will give nothing to process to the first thread (the first value of `end` will be 0), and will not process the last elements; finally, J is correct.

3. The idea is to split the set of integers between $\sqrt{N}$ and N into equal interval, and have each process check the numbers in that interval by testing each number if divisible with all the primes up to $\sqrt{N}$. Process 0 will start by broadcasting the list of primes up to $\sqrt{N}$, and, in the end, it will collect the primes found by each of the workers.

The C++ implementation is (note: you can write it in Java or C#, as well):

```
// finds the interval where process with the given id is to search for primes,
// and also does the actual search
void primesInInterval(int myId, int nrProcs, int maxN,
    vector<int> const& primesToSqrtN, vector<int>& result)
{
    // we split in equal parts the interval from primesToSqrtN.back() (inclusive)
    // up to maxN+1 (exclusive) and take our share
    int begin = myId*(maxN+1-primesToSqrtN.back())/nrProcs;
    int end = (myId+1)*(maxN+1-primesToSqrtN.back())/nrProcs;
    for(int candidate=begin ; candidate<end ; ++candidate) {
        size_t i=0;
        while(i<primesToSqrtN.size() && candidate % primesToSqrtN[i] != 0) ++i;
        if(i == primesToSqrtN.size()) {
            result.push_back(candidate);
        }
    }
}

vector<int> primes(int maxN, int nrProcs,
        vector<int> const& primesToSqrtN)
{
    // broadcast input data
    int metadata[2];
    metadata[0] = maxN;
    metadata[1] = primesToSqrtN.size();
    MPI_Bcast(metadata, 2, MPI_INT, 0);
    MPI_Bcast(primesToSqrtN.data(), primesToSqrtN.size(),
        MPI_INT, 0);

    // we do our own part
    vector<int> result;
    primesInInterval(0, nrProcs, maxN, primesToSqrtN, result);

    // collect data from the other processes
    for(int i=1 ; i<nrProcs ; ++i) {
        MPI_Status status;
        int size;
```

```
        MPI_Recv(&size, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
        vector<int> tmpVec(size);
        MPI_Recv(&tmpVec.data(), size, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
        copy(tmpVec.begin(), tmpVec.end(), back_inserter(result));
    }
    return result;
}

void worker(int myId, int nrProcs)
{
    // get input data
    int metadata[2];
    MPI_Bcast(metadata, 2, MPI_INT, 0);
    vector<int> primesToSqrtN(metadata[1]);
    MPI_Bcast(primesToSqrtN.data(), primesToSqrtN.size(),
        MPI_INT, 0);

    // we do our own part
    vector<int> result;
    primesInInterval(myId, nrProcs, metadata[0], primesToSqrtN, result);

    // send back the result
    int size = result.size();
    MPI_Ssend(&size, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Ssend(result.data(), size, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
```