UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFR11023 PARALLEL PROGRAMMING LANGUAGES AND SYSTEMS (LEVEL 11)

**Friday 22nd May 2015**

**09:30 to 11:30**

## INSTRUCTIONS TO CANDIDATES

**Answer any TWO questions.**

**All questions carry equal weight.**

**CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

Year 4 Courses

Convener: I. Stark
External Examiners: A. Cohn, T. Field

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Discuss the conceptual differences between the ways monitors are supported in the Pthreads library and in the Java language. You are not required to remember precise details of syntax. [*5 marks*]

   (b) With the help of a simple state diagram, explain what is meant by the term *signal-and-continue* monitor semantics. [*5 marks*]

   (c) Write and explain code which defines a monitor to implement a semaphore. Semaphores should be initialized to zero (i.e. there is no need to provide a constructor operation, just show how you would implement P() and V()). You can write in any language, or pseudocode, as long as it is very clear where you are using standard monitor operations. You should assume signal-and-continue semantics. [*6 marks*]

   (d) Write and explain code which uses a monitor to control the behaviour of a very simple bank account object. The account only needs to store the current balance, and should offer two methods

   - `payIn (int n)` which adds `n` to the current balance,
   - `withdraw (int n)` which subtracts `n` from the current balance provided that the balance is at least `n`.

   You should assume that many concurrent threads may be attempting these operations arbitrarily. A call to `payIn` should always succeed. A call to `withdraw` when the balance is less than the amount to be withdrawn should be blocked until the balance has become large enough to allow the withdrawal to proceed.

   You can write in any language, or pseudocode, as long as it is very clear where you are using standard monitor operations. You should assume signal-and-continue semantics. [*9 marks*]

2. (a) Briefly discuss the main characteristics of the "interacting peers" parallel pattern, independently of whether the underlying model is shared variable or message passing. [*4 marks*]

(b) Consider the pseudocode below. It was used in lectures to describe a threaded shared variable implementation of the Jacobi method as an example of the interacting peers pattern. All of the necessary synchronization has been removed from the code.

```
1  shared real grid[n+2, n+2], newgrid[n+2, n+2];
2  shared bool converged; local real diff;
3  co [i = 1 to n, j = 1 to n] {
4    initialise grid;
5    do {
6      converged = true;                              ## provisionally
7      newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
8                      grid[i,j-1] + grid[i,j+1])/4; ## new value
9      diff = abs (newgrid[i,j] - grid[i,j]);         ## local change
10     if (diff > EPSILON) converged = false;         ## any one will do
11     grid[i,j] = newgrid[i,j];                      ## copy back
12   } while (not converged);
13 }
```

Describe what you would add to this in order to correctly synchronize the program, and explain the problems that each of your additions addresses. You can use the line numbers to assist in your explanation. [*8 marks*]

(c) Suppose you were asked to re-implement the Jacobi example, using Linda (rather than shared variables) as the interaction mechanism. Describe the decisions you would have to make, and how you would address them using Linda primitives. As with the pseudocode above, you should have one parallel process per grid point. You should explain process creation, data representation and synchronization issues. [*8 marks*]

(d) Linda's tuple-handling primitives are sometimes compared to message-passing send and receive operations. Explain why designing an implementation of Linda `out()` is a more challenging task than designing an implementation of `MPI_Send()`. [*5 marks*]

3. (a) Consider the following code, extracted from the "prime sieve" MPI program discussed in lectures. It is the source for the "sieve" process.

```
1   int main(int argc, char *argv[]) {
2    MPI_Comm predComm, succComm; MPI_Status status;
3    int myprime, candidate;
4
5    int firstoutput = 1;
6    MPI_Init (&argc, &argv);
7    MPI_Comm_get_parent (&predComm);
8    MPI_Recv(&myprime, 1, MPI_INT, 0, 0, predComm, &status);
9    printf ("%d is a prime\n", myprime);
10   MPI_Recv(&candidate, 1, MPI_INT, 0, 0, predComm, &status);
11   while (candidate!=-1) {
12     if (candidate%myprime) {
13        if (firstoutput) {
14          MPI_Comm_spawn("sieve", argv, 1, MPI_INFO_NULL, 0,
15                         MPI_COMM_WORLD, &succComm, MPI_ERRCODES_IGNORE);
16          firstoutput = 0;
17        }
18        MPI_Send(&candidate, 1, MPI_INT, 0, 0, succComm);
19     }
20     MPI_Recv(&candidate, 1, MPI_INT, 0, 0, predComm, &status);
21   }
22   if (!firstoutput) MPI_Send(&candidate, 1, MPI_INT, 0, 0, succComm);
23   MPI_Finalize();
24  }
```

i. Identify the MPI processes that this program creates. [*2 marks*]

ii. All of the `MPI_Send` and `MPI_Recv` calls in the code use the value `0` for the fourth parameter, which provides the destination id for sends and the source id for receives. However, this does not mean that all sends and receives across the execution target a single process. Explain how the programmer has used other MPI concepts to ensure the correct message flow. [*5 marks*]

iii. Suppose all `MPI_Send` calls in the source were replaced with `MPI_Ssend`. Explain what this means and discuss whether it might have any impact on the correctness of the program. [*3 marks*]

*QUESTION CONTINUES ON NEXT PAGE*

*QUESTION CONTINUED FROM PREVIOUS PAGE*

(b) Here is the API for MPI's operation `MPI_Comm_split`.

```
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

i. Explain the purpose of this operation, and the way in which its parameters should be used to achieve this effect. [*5 marks*]

ii. Suppose that `MPI_COMM_WORLD` contains `p` processors, where `p` is a power of 2, and that each process contains an integer variable called `myval`. We want to sum the `myval` values, leaving the result in variable `total` of process 0. Without using `MPI_Reduce`, `MPI_Scan` or their variants, sketch and explain pseudocode for a recursive function

```
void sumToP0 (MPI_Comm c)
```

written so that the call

```
sumToP0 (MPI_COMM_WORLD);
```

achieves the desired result by splitting `c` into two equal parts, summing recursively in each part, then adding the two recursively found sub-sums into a single overall total. Minor syntactic errors will not be penalized. [*10 marks*]