Theory & Practice of High Performance Computing: CS3101. **UWO, March, 4, 2015.**

Student name:	
Student ID number:	

Guidelines. The exam is closed book and all notes are forbidden. The duration is 1 hour 40 minutes. There are 17 pages in the exam. The last three pages are blank: they can be used as scratch paper and will not be marked. The exam consists of 3 exercises located from Page 2 to Page 14. The mark allotment and a suggested time allotment are provided in the table below. All answers should be written in the *answer boxes*. No justifications for the answers are needed. You are expected to do this exam on your own without assistance from anyone else in the class. If possible, please avoid pencils and use pens with dark ink. Thank you.

Marks. Please, **do not write** anything in the table below.

Exercise	Maximum Mark	Expected Time				
1	50	50 min .				
2	25	25 min.				
3	25	25 min.				
TOTAL	100	1h40				

Exercise 1: multiple choice questions

In each case, **zero, one or more answers** may be correct; indicate **all correct answers**.

(1) Consider the following `Julia` function `f`:

```
function f(u,v)
   n=length(u)
   [u[i] + v[i] for i=1:n]
end
```

which assumes that `u` and `v` are vectors of equal length.

- (a) the function call `f(u,v)` prints "Hello World"
- (b) the function call `f(u,v)` does not return anything
- (c) the function call `f(u,v)` returns the sum of the vectors `u` and `v`
- (d) the function call `f(u,v)` returns the sum of the vectors `u` and `v` provided that their coefficients are integer numbers.

(2) A *multi-core processor* is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.

- (a) True
- (b) False

(3) Consider the following `Julia` function and commands

```
function producer()
   produce("start")
   for n=1:2
       produce(2n)
   end
   produce("stop")
end

for x in Task(producer)
         println(x)
end
```

After executing them, one sees the following (and only the following) output value

(a) "Task"

(b) "start"
2
4
"stop"

(c) "start"

(d) "producer"

(e) "consumer"

(4) Consider the following `CilkPlus` function

```
int fib(int n)
{
   if (n < 2) return n;
   int x, y;
   x = cilk_spawn fib(n-1);
   y = fib(n-2);
   cilk_sync;
   return x+y;
}
```

which computes the `n`-th Fibonacci number. For the function call `fib(n)` to execute correctly, the corresponding code must run on:

(a) a multi-core processors with at least n cores,
(a) a multi-core processors with at least 2^n cores,
(c) any multi-core processor systems where the `CilkPlus` run-time is installed.

(5) Consider a `CilkPlus` program whose DAG has work $T_1 = 96$ and span $T_\infty = 10$. On a multi-core processor with $P = 24$ cores, the DAG model (as discussed in class) predicts that the minimum running time T_P on P cores is at least:

(a) 4
(b) 10

(6) Consider a `CilkPlus` program whose DAG has work $T_1 = 96$ and span $T_\infty = 2$. On a multi-core processor with $P = 24$ cores, the DAG model (as discussed in class) predicts that the minimum running time T_P on P cores is at least:

(a) 4
(b) 2

(7) Consider the following `Julia` session where two methods are proposed for computing the square of a random matrix.

```
#method 1
A = rand(1000,1000)
Bref = @spawn A^2
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
fetch(Bref)
```

(a) In the first method, a random matrix is constructed locally, then sent to another processor where it is squared.
(b) In the first method, a random matrix is both constructed and squared on another processor.
(c) In the second method, a random matrix is constructed locally, then sent to another processor where it is squared.
(d) In the second method, a random matrix is both constructed and squared on another processor.

(8) Consider the following `Julia` session:

```
n = @parallel (+) for i=1:10
            i
end
```

After executing the above, the value of `n` is:

(a) 10
(b) 55
(c) the number of processors involved in this `Julia` session
(d) a remote reference

(9) Consider the distributed array `da` defined as follows

```
da = @parallel [2 * i for i = 1:10]
```

and assume that 2 workers, with numbers `2` and `3`, own `da`: Worker `2` owns the first 5 elements and Worker `3` owns the last ones. What does the following command return?

```
fetch(@spawnat 2 da[3])
```

(a) `6`
(b) `2`
(c) `3`
(d) `Bound error`

(10) Consider again the distributed array `da` defined as follows

```
da = @parallel [2 * i for i = 1:10]
```

and assume again that 2 workers, with numbers `2` and `3`, own `da`: Worker `2` owns the first 5 elements and Worker `3` owns the last ones. What does the following command return?

```
[(@spawnat p sum(localpart(da))) for p=procs(da)]
```

(a) An array of two remote references
(b) `[2 3]`

(11) Consider again the distributed array `da` defined as follows

```
da = @parallel [2 * i for i = 1:10]
```

and assume again that 2 workers, with numbers `2` and `3`, own `da`: Worker `2` owns the first 5 elements and Worker `3` owns the last ones. What does the following command return?

```
map(fetch, { (@spawnat p sum(localpart(da))) for p=procs(da) })
```

(a) A 2-element array with entries `30` and `80`
(b) A 2-element array with remote references as entries

(12) Consider again the distributed array `da` defined as follows

```
da = @parallel [2 * i for i = 1:10]
```

and assume again that 2 workers, with numbers `2` and `3`, own `da`: Worker `2` owns the first 5 elements and Worker `3` owns the last ones. What does the following command return?

```
reduce(+, map(fetch, { (@spawnat p sum(localpart(da))) for p=procs(da) ]))
```

(a) `2`
(b) `110`

(13) Consider the following `Julia` functions defined in the same `Julia` session, which is using 4 workers in addition to the master.

```
@everywhere function fib(n)
                if (n < 2) then
                    return n
                else return fib(n-1) + fib(n-2)
                end
            end
```

```
@everywhere function fib_parallel(n)
          if (n < 40) then
              return fib(n)
          else
              x = @spawn fib_parallel(n-1)
              y = fib_parallel(n-2)
              return fetch(x) + y
          end
       end
```

What is the value `fib_parallel(4)`?

(a) `3`
(b) `5`
(c) `8`

For each of the following pseudo-code (using the same syntax and semantics as `Cilk++` or `CilkPlus`) choose the proper estimate(s) for the work or span using the big-O notation. Note that these estimates depend on the following:

- `n` which is an `int` variable,
- W_A and S_A which are the work and span of the function call `A()`, respectively

Note also that `void A (void)` is a `C++` function.

(14)
```
/* Program 1 */


for (int i = 0; i < n ; i++) {
    A();
}
```

What is the work of the above code?

(a) $n \times W_A$
(b) W_A

(15)
```
/* Program 2 */

for (int i = 0; i < n ; i++) {
    A();
}
```

What is the span of the above code?

(a) $n \times S_A$
(b) S_A

(16)
```
/* Program 3 */

cilk_for (int i = 0; i < n ; i++) {
    A();
}
```

What is the work of the above code?

(a) $n \times W_A$
(b) $W_A \times \log(n)$

(17)
```
/* Program 4 */

cilk_for (int i = 0; i < n ; i++) {
    A();
}
```

What is the span of the above code?

(a) $S_A \times \log(n)$
(b) $S_A + \log(n)$

(18)

1. `/* Program 5 */`

```
cilk_for (int i = 0; i < n ; i++) {
    cilk_for (int k = 0; k < n ; k++) {
        for (int j = 0; j < 4; j++) {
            A();
        }
    }
}
```

What is the work of the above code?

(a) $O(n^3 \times W_A)$
(b) $O(W_A \times n^2 \log(n))$

(19)

2. `/* Program 6 */`

```
cilk_for (int i = 0; i < n ; i++) {
    cilk_for (int k = 0; k < n ; k++) {
        for (int j = 0; j < 4; j++) {
            A();
        }
    }
}
```

What is the span of the above code?

(a) $O(n + S_A)$
(b) $O(\log(n) + S_A)$

Exercise 2: `Julia` **questions with short answers**

(1) Write a `Julia` function `dotproduct_serial(U,V)` computing the dot product of the vectors of `U` and `V` in a serial fashion, that is, without using any parallel constructs.

```
function dotproduct_serial(u,v)
n = length(u) # length of the vector
sum = 0
for i=1:n
sum = sum + u[i] * v[i]
end
sum
end

# test
u=Array(Int,4)
v=Array(Int,4)
for i =1:4
u[i] = i
v[i] = i+1
end

dotproduct_serial(u,v)
```

(2) Write a `Julia` function `dotproduct_parallel_reduction(U,V)` computing the dot product of the vectors of `U` and `V` using parallel reduction and thus `Julia`'s construct `@parallel`.

```
function dotproduct_parallel_reduction(u, v)
n = length(u) # length of the vector
sum = @parallel (+) for i = 1:n
u[i] * v[i]
end

sum
end

# test
u=Array(Int,4)
v=Array(Int,4)
for i =1:4
u[i] = i
v[i] = i+1
end

dotproduct_parallel_reduction(u, v)
```

(3) Next, write a `Julia` function `dotproduct_parallel_dnc(U,V)` computing the dot product of the vectors of `U` and `V` using a divide-and-conquer approach (with a base case of your choice). For this function, we shall assume that `U` and `V` are shared arrays.

```
@everywhere function dotproduct_parallel_dnc(u, v)
n = length(u) # midlle point
if n ==1
return u[1]*v[1]
end

m = floor(Integer, n/2) # midlle point
k = m+1

f = @spawn dotproduct_parallel_dnc(u[1:m], v[1:m])
s = dotproduct_parallel_dnc(u[k:n], v[k:n])

return fetch(f) + s
end

# test
u = SharedArray(Int, (1,4), init=true)
v = SharedArray(Int, (1,4), init=true)
for i =1:4
u[i] = i
v[i] = i+1
end
dotproduct_parallel_dnc(u, v)
```

(4) Consider the `Julia`'s function below for multiplying two square matrices `A` and `B` of order `n`. Using `Julia`'s construct `@spawnat` and `fetch` make a parallel version of that function that uses 4 processors.

```
function four_quadrant_mat_mul_serial(A, B, n)

        C = zeros(n, n)
        d = div(n,2)
        e = d+1
        C[1:d, 1:d] = A[1:d, 1:d] * B[1:d, 1:d] +
                      A[1:d, e:n] * B[e:n, 1:d]
        C[1:d, e:n] = A[1:d, 1:d] * B[1:d, e:n] +
```

```
                A[1:d, e:n] * B[e:n, e:n]
        C[e:n, 1:d] = A[e:n, 1:d] * B[1:d, 1:d] +
                A[e:n, e:n] * B[e:n, 1:d]
        C[e:n, e:n] = A[e:n, 1:d] * B[1:d, e:n] +
                A[e:n, e:n] * B[e:n, e:n]
        C
end
```

```
@everywhere function four_quadrant_mat_mul_serial(A, B, n)

C = zeros(n, n)
d = div(n,2)
e = d+1
f = @spawn A[1:d, 1:d] * B[1:d, 1:d] + A[1:d, e:n] * B[e:n, 1:d]
g = @spawn A[1:d, 1:d] * B[1:d, e:n] + A[1:d, e:n] * B[e:n, e:n]
h = @spawn A[e:n, 1:d] * B[1:d, 1:d] + A[e:n, e:n] * B[e:n, 1:d]
j = A[e:n, 1:d] * B[1:d, e:n] + A[e:n, e:n] * B[e:n, e:n]

C[1:d, 1:d] = fetch(f)
C[1:d, e:n] = fetch(g)
C[e:n, 1:d] = fetch(h)
C
end
```

Exercise 3: writing a parallel `CilkPlus` function

Consider the problem of multiplying an $n \times m$ matrix A by an m-vector b. The resulting n-vector d is given by the equation

$$d[i] = \sum_{j=1}^{m} A[i][j] \cdot b[j],$$

for $i = 1, 2, \ldots, n$.

(1) For the following pseudo-code (using the same syntax and semantics as `CilkPlus` determine the **work** and the **span**

```
/* Program 1 */
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < m; ++j)
      d[i] += A[i][j] * b[j];
}
```

work: O(m*n)
span: O(m*n)

(2) Same question as above for the following `CilkPlus` pseudo-code

```
/* Program 2 */
cilk_for (int i = 0; i < n; ++i) {
   for (int j = 0; j < m; ++j)
      d[i] += A[i][j] * b[j];
}
```

work: O(m*n)
span: O(m + log(n))

(3) Complete the following `CilkPlus` pseudo-code such that it realizes a divide-and-conquer implementation of the multiplication of an $n \times m$ matrix A by an m-vector b. Note that **two lines** (and only two lines) of pseudo-code need to be completed.

```
/* Program 3 */
MAT-VEC (d, A, b, m, i, j) {
   if (i == j) {
      for (int k = 1; k < m; ++k)
        d[i] += A[i][k] * b[k];
   }
   else {
      k = ⌊ (i+j)/2 ⌋;
      /* The line below needs to be completed */
      cilk_spawn MAT-VEC (d, A, b, m, i, k);
      MAT-VEC (d, A, b, m, k+1, j);
      /* The line below needs to be completed */
      cilk_sync;
   }
}
```

(4) When the function MAT-VEC `(d, A, b, m, 1, n)` is performed, how many recursive calls take place along the critical path? Deduce an estimate of the **span** of the algorithm completed at the previous question.

There are log(n) recursive calls along the critical path.

$S(n) = O(log(n)) + O(m)$

$O(log(n))$: the heigh of the recursive spawning tree,

$O(m)$ is the executing of the for-loop.

Note that the for-loop only executed at the bottom of recursion tree (after $log(n)$ recursive calls), each by an individual processor.