CS4402/CS9535b: Corrected exercises (including Quiz 1). UWO, February 27, 2014.

Student ID number:
Student Last Name:

Guidelines. The quiz consists of two exercises. All answers should be written in the *answer boxes*. No justifications for the answers are needed, unless explicitly required. You are expected to do this quiz on your own without assistance from anyone else in the class. If possible, please avoid pencils and use pens with dark ink. Thank you.

Exercise 1. For each of the following pseudo-code (using the same syntax and semantics as `Cilk++`)

1. sketch the shape of the instruction stream DAG,
2. determine the **work**,
3. determine the **span**.

Your estimate will depend on the following quantities:

- `n`, which is an `int` variable,
- `void A (void)` and `void B(int m)`, which are two `C++` functions,
- W_A and S_A, which are the work and span of the function call `A()`,
- $W_B(n)$ and $S_B(n)$, which are the work and span of the function call `B(n)`.

Of course, you can use the big-Oh, max and sigma notations, when appropriate. For instance $O(n)$ or $\max(S_B(i), 0 \leq i \leq n-1)$ or $\sum_0^{n-1} W_B(i)$.

1. `/* Program 1 */`

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

Answer 1

The work and the span are in the order of $\Theta(nW_A)$ and $\Theta(nS_A)$ respectively, thus, in the order of $\Theta(n)$ and $\Theta(n)$, since W_A and S_A are constant (i.e indepedent of n).

2. `/* Program 2 */`

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

Answer 2

The work and the span are in the order of $\Theta(nW_A)$ and $\Theta(\log(n) + S_A)$ respectively, thus, in the order of $\Theta(n)$ and $\Theta(\log(n))$, since W_A and S_A are constant (i.e indepedent of n).

3. `/* Program 3 */`

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

Answer 3

The work and the span are in the order of $\Theta(\sum_{i=0}^{i=n-1} W_B(i))$ and $\Theta(\max_{i=0}^{i=n-1} S_B(i) + \log(n))$ respectively.

4. `/* Program 4 */`

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

Answer 4

The work and the span are in the order of $\Theta(4nW_A)$ and $\Theta(\log(n) + 4S_A)$ respectively, thus, in the order of $\Theta(n)$ and $\Theta(\log(n))$, since W_A and S_A are constant (i.e indepedent of n).

5. `/* Program 5 */`

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

Answer 5

The work and the span are in the order of $\Theta(4nW_A)$ and $\Theta(\log(n) + 2 + S_A)$ respectively, thus, in the order of $\Theta(n)$ and $\Theta(\log(n))$, since W_A and S_A are constant (i.e indepedent of n).

6. `/* Program 6 */`

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

Answer 6

The work and the span are in the order of $\Theta(4nW_A)$ and $\Theta(n(2 + S_A))$ respectively, thus, in the order of $\Theta(n)$ and $\Theta(n)$, since W_A and S_A are constant (i.e indepedent of n).

7. `/* Program 7 */`

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

Answer 7

The work and the span are in the order of $\Theta(4nW_A + \sum_{i=0}^{i=n-1} W_B(i)))$ and $\Theta(\log(n) + 2 + S_A + \max_{i=0}^{i=n-1} S_B(i))$ respectively, thus, in the order of $\Theta(n + \sum_{i=0}^{i=n-1} W_B(i))$ and $\Theta(\log(n) + \max_{i=0}^{i=n-1} S_B(i))$, since W_A and S_A are constant (i.e indepedent of n).

Exercise 2. Let A be a square matrix of order $n \geq 2$, where n is assumed to be a power of 2. The goal of this exercise is to write a parallel algorithm or program which will compute the inverse of A. To do so, we will rely on a classical formula from Linear Algebra. We shall analyze the work and the critical path of this algorithm (or program). We assume that we have two parallel sub-algorithms at our disposal:

- $\mathrm{ADD(C, T, n)}$ computing the sum of two square matrices C and T (C is replaced by $C + T$) of order n in work $A_1(n) = \Theta(n^2)$ and with critical path $A_\infty(n) = \Theta(\log(n))$;
- $\mathrm{MULT(C, A, B, n)}$ computing the product of two square matrices A and B (C is replaced by AB) of order n in work $M_1(n) = \Theta(n^3)$ and with critical path $M_\infty(n) = \Theta(\log^2(n))$;

We start by reviewing this classical formula. We partition A into four square blocks of order $n/2$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{1}$$

Let O and I be the zero and identity matrices of order $n/2$, respectively. We assume that the upper right block A_{11} is an invertible matrix. Then we define $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ (called the *Schur complement*). We assume that S is also invertible. Then we have:

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ O & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & O \\ O & S^{-1} \end{bmatrix} \begin{bmatrix} I & O \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \tag{2}$$

Let us denote by U, D, L the above three matrices respectively.

1. Write a `Cilk`-like program computing the inverse of A based on the above formula. We will **assume** that at each recursive call the upper right and the Schur complement are invertible. You shall try to make the critical path as small as possible.
2. Estimate $I_1(n)$ the work of your algorithm
3. Estimate $I_\infty(n)$ the critical path of your algorithm.
4. Is your algorithm in the NC clas? Explain.

Before writing a `Cilk`-like program, let us first write an *informal algorithm*:

1. Compute A_{11}^{-1} by a recursive call,
2. Compute $A_{11}^{-1}A_{12}$ and $A_{21}A_{11}^{-1}$ in parallel,
3. Compute $A_{21}A_{11}^{-1}A_{12}$,
4. Compute S,
5. Compute S^{-1} by a recursive call,
6. Compute $U\ D$
7. Compute and return $U\ D\ L$.

In the following program, we use four subroutines `Opposite`, `ZeroMatrix`, `UnitMatrix` and `CopyMatrix`. They respectively compute the opposite of a matrix, set a matrix to the zero one, set a matrix to the unit one and copy a matrix into another one. For each of these subroutines, the output is the first argument and the other ones are the input data.

Note that the proposed procedure `Inverse` uses its third and fourth arguments as work space. This procedure does not use any local variables. We use the following notation in this program. For a square matrix `M` of even order, we denote by `M[i,j]` the element at the intersection of row `i` and `j`; we denote by `M{1,1}`, `M{1,2}`, `M{2,1}`, `M{2,2}` the top-left, top right, bottom-left, bottom-right blocks of `M` (each of order half that of `M`).

```
/* I, A, W are 3 square matrices of order n    */
/* C is a square matrix of order n/2           */
/* A is the input, W and C are work space,     */
/* I is the output, that is, the inverse of A  */
/* A is overwritten                            */
void Inverse(I, A, W, C, n)
  if n=1
  then I[1,1] := 1 / A[1,1]
  else
      spawn Inverse(I{1,1}, A{1,1}, W{2,2}, C{1,1}, n/2)
      /* Now I{1,1} contains the inverse of A{1,1} */
      spawn Opposite(W{2,2}, A{1,2}, n/2)
      sync
      spawn Mult(W{2,1}, A{2,1}, I{1,1}, n/2)
      /* Now W{2,1} contains A{2,1} * I{1,1} */
      spawn Mult(W{1,2}, I{1,1}, W{2,2}, n/2)
      /* Now W{1,2} contains the opposite of I{1,1} * A{1,2}   */
      sync
      spawn Mult(W{2,2}, A{2,1}, W{1,2}, n/2)
      sync
      spawn Add(A{1,2}, A{2,2}, W{2,2}, n/2)
      /* Now A{1,2} contains the Schur complement */
      spawn CopyMatrix(A{2,1}, W{2,1}, n/2)
      /* Now A{2,1} contains A{2,1} * I{1,1} */
      sync
      spawn Inverse(I{2,2},  A{1,2}, W{1,1}, C{1,1}, n/2)
      /* Now I{2,2} contains the inverse of the Schur complement */
      sync
      spawn UnitMatrix(W{1,1}, n/2)
      spawn ZeroMatrix(W{2,1}, n/2)
      spawn UnitMatrix(W{2,2}, n/2)
      /* Now W = U */
      spawn  ZeroMatrix(I{2,1}, n/2)
      spawn  ZeroMatrix(I{1,2}, n/2)
      /* Now I = D */
      spawn CopyMatrix(C, A{2,1}, n/2)
      sync
      spawn Mult(A, W, I, n)
      sync
      spawn ZeroMatrix(W{1,2}, n/2)
      spawn CopyMatrix(W{2,1}, C, n/2)
      sync
      spawn Mult(I, A, W, n)
      return
```

For the analysis of the work and the critical path, we neglect the subroutines `Opposite`, `ZeroMatrix`, `UnitMatrix` and `CopyMatrix` for simplicity.
Based on the informal algorithm, the critical path $I_\infty(n)$ satisfies:

$$\begin{aligned} I_\infty(n) &= 2\,I_\infty(n/2) + 2\,M_\infty(n/2) + 2\,M_\infty(n) + A_\infty(n/2) \\ &= 2\,I_\infty(n/2) + 2\,\Theta(\log^2(n/2)) + 2\,\Theta(\log^2(n)) + \Theta(\log(n)) \\ &= 2\,I_\infty(n/2) + \Theta(\log^2(n)) \\ &= \Theta(n) \end{aligned}$$

Therefore our procedure is **not** in the NC class.
Then, the work satisfies:

$$\begin{aligned} I_1(n) &= 2\,I_1(n/2) + 3\,M_1(n/2) + 2\,M_1(n) + A_1(n/2) \\ &= 2\,I_1(n/2) + \Theta(n^3) \\ &= \Theta(n^3) \end{aligned}$$

Exercise 3. Consider the following `Cilk++` code fragment.

```
static const int COUNT = 4;
static const int ITERATION = 1000000;
long arr[COUNT];
long do_work(long k){
  long x = 15;
  static const int nn = 87;
  for (long i = 1; i < nn; ++i)
    x = x / i + k % i;
  return x;
}
int cilk_main(){
  for (int j = 0; j < ITERATION; j++)
    cilk_for (int i = 0; i < COUNT; i++)
      arr[i] += do_work( i + j);
}
```

1. Estimate the (theoretical) parallelism of this program.
2. Explain why the burdened span of this program is much higher than the (theoretical) span. You can sketch the shape of the instruction stream DAG to support your explanation.
3. How can we fix the performance bottleneck of this program?

Note: Recall that `Cilk++` estimates T_p as $T_p = T_1/p + 1.7\,\mathrm{burden\_span}$, where $\mathrm{burden\_span}$ is 15000 instructions times the number of continuation edges along the critical path.

Answer 8

1. Let us write n instead of `ITERATION`. Observe that the work and span of the `do_work` are constant W and S (i.e. indepedent of n). Moreover we have $W = S$ since `do_work` is pure serial code. Thus the work and span of the above program are respectively propoertional to $4nW$ and $n(W + 2)$. Thus the theoretical parallelism is about 4.

2. Each of the n iterations of the outer for-loop will execute, in sequence, the ineer `cilk_for` loop. Therefore, the burdened span of this program is n times 15000 instructions times 2. The factor 2 is the (theoretical) span of the inner loop. This burdened span is likely to be greater than $4nW$. Hence, this parallel program is likely to be slower than its serial counterpart.

3. It is easy to verify that one can exchange the two for-loops. This reduces to burdened span to $2 \times (15000 + W)$. On a 4-core machine, this modified program is likely to reach a speedup factor of 4.