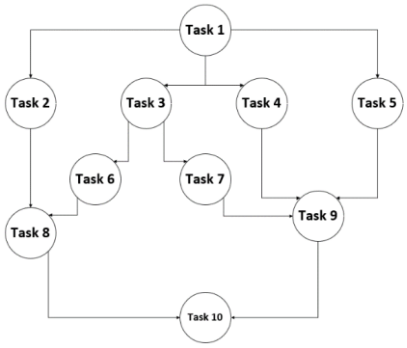
Amirkabir University of Technology
(Tehran Polytechnic)

Department of Computer Engineering,
Multicore Programming (Spring 2024),
Assignment 1

Question 1: Answer to the following questions according to the figure.



a) Determine the maximum degree of concurrency and the length of the critical path.

b) Assume that you have 4 processors that can do each task in 2 seconds. If it takes 1 second to transfer the data between two processors, make an assignment of each task to each processor and calculate how long it takes to run all the tasks. For example if Task 1 happens in processor 1 and Task 2 happens in processor 2, you have to transfer the data from processor 1 to processor 2 before executing Task 2. However, if Task 1 and Task 3 are assigned to the same processor there's no need for any data transfer. You can assume that data transfers do not block each other.

c) Let t be the number of tasks in a dependency graph with maximum degree of concurrency d and a critical path of length l. Show that the upper and lower bounds for the maximum degree of concurrency are $\left\lceil \frac{t}{l} \right\rceil \leq d \leq t - l + 1$?

Question 2: We're trying to visualize how Leibniz formula for calculating π converges to $\frac{\pi}{4}$. Therefor we need to calculate $\sum_{k=0}^{i} \frac{(-1)^k}{2k+1}$ for each i in range [0, M]. The following is the serial implementation of this program.

```
int M;
int numerator = 1;
float sum = 0;
float values[M];
for (int i = 0; i < M; i++) {
   sum += (float)numerator / (float)(2 * i + 1);
   values[i] = sum;
   numerator *= -1;
}
```

a) Draw the dependency graph for above code for M = 7, determine the maximum degree of concurrency and the length of the critical path.

b) Using Task Parallelism model, map the operations to two processes in order to achieve a higher speed in a dual-core system. Draw the dependency graph and determine the maximum degree of concurrency and the length of the critical path. Note that you should change the order of operations.

Hint:

```
a = k1
b = a + k2
c = b + k3
d = c + k4
```

Could be rewritten as

```
a = k1
b = a + k2
c = k3
d = c + k4
c += b
d += b
```

Question 3: Let's assume we have N natural numbers in range [1, M]. We want to sort these numbers using bucket-sort with B buckets. The following is a sample pseudo-code of the serial version of this program (for simplicity assume that M is divisible by B and also; N and B are divisible by P).

```
int N, M, B;
unsigned int arr[N];
vector<int> buckets[B];
vector<int> sorted;
int k = M / B;
for (int i = 0; i < N; i++)
   buckets[arr[i] / k].push_back(arr[i]);
for (int i = 0; i < B; i++) {
   bucket[i].merge_sort();
   sorted.append(bucket[i]);
}
```

a) Write a pseudo-code for a parallel version of this program mapping the operations to **P processes**, based on input data partitioning. The following is a sample pseudo-code for executing a code section using a specific process. Inside the block for each process, every time a global variable is called, a (t_{comm} * variable size) time penalty should be added to operation time. To mitigate this issue, try using local variables (defined inside process block) as much a possible. After calling "wait_for_process" function, the variables called using "keep" in process block will be sent back to the main block. For every variable kept a (t_{comm} * variable size) time penalty should be added to operation time. Keep in mind that you can use the functions from pseudo-code above but vector data type isn't thread safe.

```
using process k {
   vector<int> a;
   int b;
   // your code
   keep a, b; // a and b will be returned to the main process
} // This block of code runs in a non-blocking manner
// you can do other stuff here
wait_for_process(k);
```

b) Write a pseudo-code for a parallel version of this program mapping the operations to **P processes**, based on output partitioning. The rules described in section (a) still hold.

c) Which method do you think results in a higher speedup? (explain)

Question 4: We have two vectors of length N, and we want to calculate the rotational convolution of these two vectors. The formula for rotational convolution is as follows:

$$\forall n \in [0, N]: c[n] = \sum_{i=0}^{N} a[i] \times b[(n-i)\ rem\ N] \qquad (Note: -1\ rem\ N = N-1)$$

The following is a sample pseudo-code of the serial version of this program.

```
int N, k;
int a[N], b[N], c[N];
for (int i = 0; i < N; i++) {
   k = i;
   for (int j = 0; j <= i; j++) {
      c[i] += a[j] * b[k]
      k--;
   }
   k = N - 1;
   for (int j = i+1; j < N; j++) {
      c[i] += a[j] * b[k]
      k--;
   }
}
```

a) Write a pseudo-code for a parallel version of this program mapping the operations to **P processes**, based on output partitioning. Which Parallel Program Model did you use to achieve the desired outcome? (For simplicity assume N is divisible by P)

b) Write a pseudo-code for a parallel version of this program mapping the operations to **P processes**, based on input data partitioning. (Hint: Only partition one of the input vectors)

c) Considering total amount of input and output data transfer of each process (as described in section (a) of question 3), which method results in less data transfer? Assuming all the data used by each process is duplicated for local usage, which method results in less memory usage?