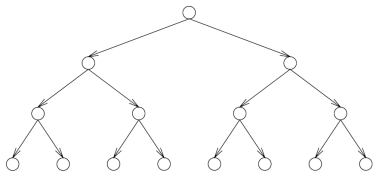
SABANCI UNIVERSITY

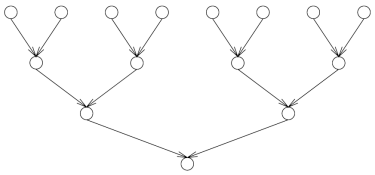
CS406/531: Parallel Computing
Spring 2023-2024
HW III

Q1 (20 pts): For the task graphs given below, determine the following:

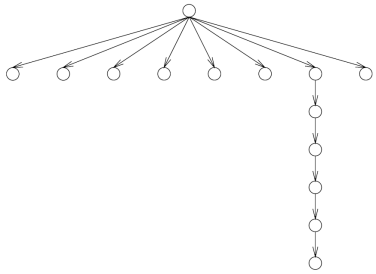
1) Maximum degree of concurrency.
2) Critical path length.
3) Maximum achievable speedup over one process assuming that an arbitrarily large number of processes are available.
4) The minimum number of processes needed to obtain the maximum possible speedup.
5) The maximum achievable speedup with 4 threads.



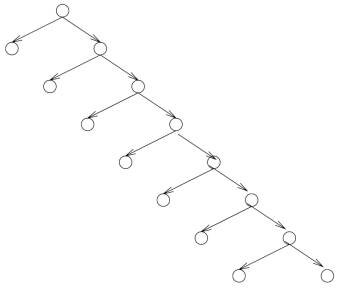
(a)



(b)



(c)



(d)

Q2 (10 pts): Sketch out how to rewrite the following code to improve its cache locality and locality in registers

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      a[j][i] = a[j-1][i-1] + c[j];
```

Briefly explain how your modifications have improved memory access behaviour of the computation.

Q3 (10pts): Suppose a fraction **r** of the runtime of a serial program is "*perfectly parallelized*," and the remaining fraction **1 − r** is "*inherently serial*." Give an upper bound on the speedup one can have.

Q4 (10 pts): Describe the following terms and their differences in at most four sentences each.

a) (**5 pts**) Latency **and** bandwidth
b) (**5 pts**) Spatial locality **and** temporal locality

Q5 (15 pts): NVIDIA P100 GPU has 3584 single-precision floating point cores with a peak 10.6 teraflops of single-precision floating point computation (1 teraflop = 1000 gigaflops = 1012 floating point operations per second). The interface to global memory has a bandwidth of 720 GBytes/sec.

How many floating point operations must the GP100 perform on each float loaded from global memory if it is to achieve its peak floating point operation rate (and not by limited by memory bandwidth)?

Q6 (15 pts): For the following vector addition kernel and the corresponding kernel launch code, answer each of the questions below, assuming that the code is running on a GPU on nebula.

```
1  __global__ void vecAddKernel (float* A, float* B, float* C, int n)
2  {
3    int i = threadIdx.x + blockDim.x * blockIdx.x * 2;
4
5    if (i < n) { C_d[i] = A_d[i] + B_d[i]; }
6    i += blockDim.x;
7    if (i < n) { C_d[i] = A_d[i] + B_d[i]; }
8  }
9
10 int vectAdd (float* A, float* B, float* C, int n)
11 {
12   // Parameter "n" is the length of arrays A, B, and C.
13   int size = n * sizeof (float);
14   cudaMalloc ((void **)&A_d, size);
15   cudaMalloc ((void **)&B_d, size);
16   cudaMalloc ((void **)&C_d, size);
17   cudaMemcpy (A_d, A, size, cudaMemcpyHostToDevice);
18   cudaMemcpy (B_d, B, size, cudaMemcpyHostToDevice);
19
20   vecAddKernel<<<ceil (n / 2048.0), 1024>>> (A_d, B_d, C_d, n);
21   cudaMemcpy (C, C_d, size, cudaMemcpyDeviceToHost);
22 }
```

a) (**3 pts**) If the size n of the A, B, and C arrays is 50,000 elements each, how many thread blocks are generated?
b) (**3 pts**) If the size n of the A, B, and C arrays is 50,000 elements each, how many warps are there in each thread block?
c) **(3 pts**) If the size n of the A, B, and C arrays is 50,000 elements each, how many threads in total will be created for the grid launched on line 20?
d) (**3 pts**) If the size n of the A, B, and C arrays is 50,000 elements each, is there any control divergence (only some threads are working within a warp) during the execution of the kernel? Explain why or why not. If so, identify the block number(s) and warp number(s) that causes the control divergence.
e) **(3 pts**) Identify the line number(s) at which control diverges for each warp that you have identified.

Q7 (**20 pts**) Please answer the following:

a. **(5 pts)** What is the maximum speedup of the computation given that only %60 of the computation can be executed in parallel? Explain clearly. There will be no credit for a single, numeric value.
b. **(5 pts)** If you have **p** processors, what is the maximum speedup you can get? Is it possible to achieve greater speedup and, if so, under what circumstance might this happen?
c. **(5 pts)** Given a sequential algorithm always requiring **n** operations on **n** numbers, you implemented a parallel version requiring **(n/p + log p)** time. What is the efficiency (E) of your algorithm?
d. **(5 pts)** (Continue from part (c)): When (up to how many processors) is your algorithm cost optimal? Show your math.