PROBLEM 3. Let G be a directed graph with n vertices. For simplicity we identify the vertex set to the set of positive integers $\{1, 2, \dots, n\}$. To each couple (i, j), with $1 \leq i, j \leq n$, we associate a weight $\omega_{i,j}$ such that:

(i) $\omega_{i,j}$ is a non-negative integer if and only if (i, j) is an arc in G,
(ii) $\omega_{i,j}$ is $+\infty$ if and only if (i, j) is not an arc in G.

We assume $\omega_{i,j} = 0$ for all $1 \leq i \leq n$. If $x_1, x_2, \dots, x_m$ are $m \geq 2$ vertices of G such that $(x_1, x_2), (x_2, x_3), \dots, (x_{m-1}, x_m)$ are all arcs of G, we say that $p = (x_1, x_2, \dots, x_m)$ is a path in G from x_1 to x_m; moreover the weight of p is denoted by $\omega(p)$ and defined by

$$\omega(p) = \omega_{x_1,x_2} + \omega_{x_2,x_3} + \cdots + \omega_{x_{m-1},x_m}$$

For each couple (i, j) which is not an arc in G it Is natural to ask whether

(1) there is a path in G from i to j, and
(2) if such path exists, then compute the minimal weight of such a path.

This question is often referred as SAP for All-Pair Shortest Paths. The celebrated Floyd-Warshall algorithm solves ASAP by computing a matrix path as follows:

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
             path[i][j] = min (path[i][j], path[i][k] + path[k][j]);
```

after initializing `path[i][j]` to $\omega_{i,j}$. For more details, please refer to the Wikipedia page of the Floyd-Warshall algorithm.

Question 1. Is it possible to turn the *Floyd-Warshall algorithm* into a parallel algorithm for the fork-join parallelism? If yes, analyze the work, the span and the parallelism of this algorithm.

Yes. The most inner loop can be parallelized using the divide and conquer method, using a cilk_for. In the case of this improvement. The work will stay at $\theta(n^3)$, while there will be an improvement in the span because of the introduction of the cilk_for, hence, $T_\infty = \theta(n^2 \times \log n)$

Question 2. Discuss the data locality of the above sequential Floyd-Warshall algorithm (not your parallel version of it). Doing a formal cache complexity analysis is not required.

The cache locality in the above algorithm is great when $2 \times n \leq$ cache size. That is because the processor will be able to cache both the i-th and k-th rows of the matrix on cold misses. However, the higher the n value the bigger the chance of introducing capacity misses.

One way to obtain a better algorithm for ASAP (in terms of parallelism and data locality) is to apply a divide and conquer approach. To this end we view $(\omega_{i,j})$ as an $n \times n$-matrix, denoted by W. We also view the targeted results, namely the values (`path[i][j]`) as an $n \times n$-matrix, denoted $\overline{W}$.

Before stating the divide and conquer formulation, we introduce a few notations. Let X, Y be square matrics (of the same order) whose entries are non-negative or $+\infty$. W

- XY the min-plus product of X by Y (obtain from the usual matrix multiplication by replace $+$ (resp. $\times$) by min (resp. $+$))
- $X \vee Y = \min(X, Y)$ the element-wise minimum of the tow matrices X and Y.

We are ready to state the divide and conquer formulation. If we decompose W into four $n/2 \times n/2$-blocks, namely

$$W = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

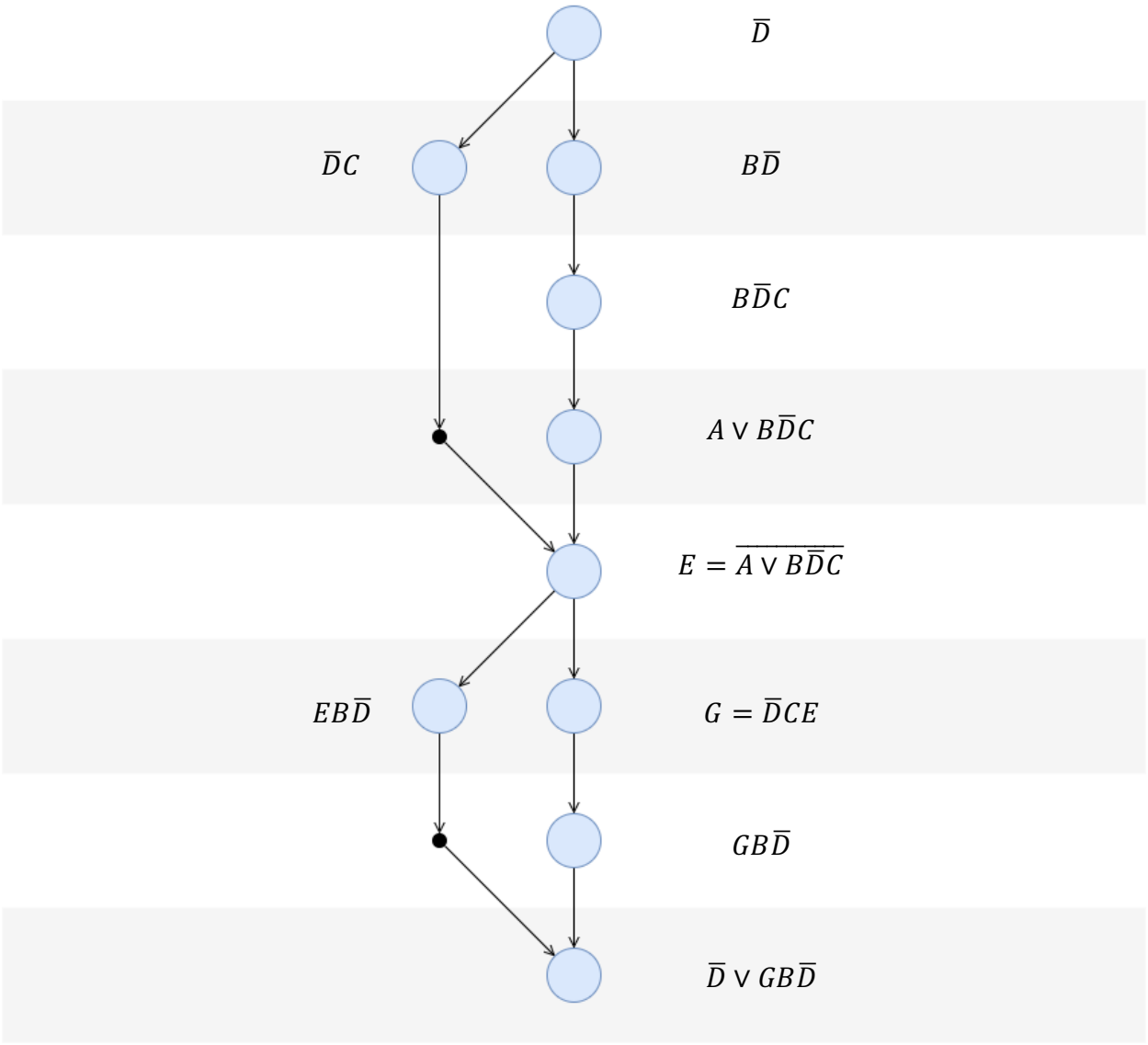
then

$$\overline{W} = \begin{pmatrix} E & EB\overline{D} \\ G & \overline{D} \vee GB\overline{D} \end{pmatrix}$$

where we have $E = \overline{\overline{A} \vee B\overline{D}C}$ and $G = \overline{D}CE$. We shall admit that these formulas are correct (even though proving them is not that hard).

Question 3. Propose an algorithm for computing $\overline{W}$ in the fork-join parallelism model.

The following DAG proposes a fork-join parallelism model for computing $\overline{W}$. The order at which the variables are computed is chosen to maximize the parallelism as it computes the variables that are essential to the computation of other variables down the pipeline. Every call to the bar function is a blocking call that is computed on the main thread.



Question 4. Analyze the work, the span and parallelism of your algorithm.

Work:

$$T_1 = 2 \times T\left(\frac{n}{2}\right) + \text{Multi}(n) + \text{Min}(n)$$

Trivially,

$$T_1 = 2 \times T\left(\frac{n}{2}\right) + n^3 + n^2$$

Using the master theorem,

$$a = 2, b = 2, c = 3$$

$$\log_b{}^a < c$$

$$\log_2{}^2 < 3$$

$$1 < 3$$

Therefore,

$$T_1 = \theta(n^3)$$

Span:

$$T_\infty = 2 \times T\left(\frac{n}{2}\right) + \text{Multi}(n) + \text{Min}(n)$$

Trivially,

$$T_\infty = 2 \times T\left(\frac{n}{2}\right) + n^2 \times \log n + 1$$

Using the master theorem,

$$a = 2, b = 2, c = 2$$

$$\log_b{}^a < c$$

$$\log_2{}^2 < 3$$

$$1 < 3$$

Therefore,

$$T_\infty = \theta(n^2 \times \log n)$$

There exist alternative algorithms for the ASAP problem which rely on the min-plus multiplication. A simple one is based on the observation that $\overline{W} = W^n$ (and in fact W^{n-1}) where is the *n*-th power W is computed for min-plus multiplication using repeated squaring.

Question 5. Propose such an algorithm. You are welcome to use the literature of simply to use the one suggested above.

The algorithm consists of recursively creating a tree where every leaf value is A. The algorithm then collapses all the leafs and internal nodes using the min-plus function until the root is reached.

Question 6. Analyze the work, the span and parallelism of this third algorithm.

Work:

$$T_1 = 2 \times T\left(\frac{n}{2}\right) + \text{Multi}(n)$$

Trivially,

$$T_1 = 2 \times T\left(\frac{n}{2}\right) + n^3$$

Using the master theorem,

$$a = 2, b = 2, c = 3$$

$$\log_b{}^a < c$$

$$\log_2{}^2 < 3$$

$$1 < 3$$

Therefore,

$$T_1 = \theta(n^3)$$

Span:

$$T_\infty = 2 \times T\left(\frac{n}{2}\right) + \text{Multi}(n)$$

Trivially,

$$T_\infty = 2 \times T\left(\frac{n}{2}\right) + n^2 \times \log n$$

Using the master theorem,

$$a = 2, b = 2, c = 2$$

$$\log_b{}^a < c$$

$$\log_2{}^2 < 3$$

$$1 < 3$$

Therefore,

$$T_\infty = \theta(n^2 \times \log n)$$

Question 7. Realize a Julia or CilkPlus a multithreaded implementation of that algorithm.

Program could be found under src/ASAP
Make command: make
Run command: ./ASAP < test.txt make