PARALLEL (AND BIG DATA) ALGORITHMS

Parallel Random Access Machines (PRAM)

- In the PRAM model, we consider p number of RAM processors, each with its **own local registers**, which all have access to a **global memory**.
- Time is divided into **synchronous** steps and in each step, each processor can do a RAM operation or it can **read/write to one global memory location**.
- The model has four variations with regard to how concurrent reads and writes to one global memory are resolved:

1. Exclusive Read Exclusive Write (EREW)
2. Concurrent Read Exclusive Write (CREW)
3. Exclusive Read Concurrent Write (ERCW)
4. Concurrent Read Concurrent Write (CRCW)

When concurrent writes on the same memory location are allowed, there are variations on how the output is determined. A simple rule is to assume that an arbitrarily chosen one of the write operations takes effect.

Similar to NC, we use variants with index k- e.g. CRCW(k) - to denote decision problems that can be computed by the corresponding version of the PRAM model with **poly(n) processors** and in **O(log^k n)** time steps.

Q) Maximum, using Fewer Processors
The maximum of n entries can be computed in **O(log log n)** time-steps, using the **CRCW** version of PRAM with n processors.

- Computing the maximum of n numbers can be done in **O(log n)** time by any of the variants (and in particular, in the weakest of them, **EREW**), using a **simple binary tree**.
- And it is also known that computing this maximum requires **Ω(log n) time-steps** in the **EREW** version.
- However, this can be done in **O(1) time-steps** in the **CRCW** model, using **O(n^2)** processors, as follows:
- Initialize n entries in the register to be all 0.
- Use n^2 processors, one responsible to compare one pair of numbers.
- If the ith item loses the comparison (i.e., is less than some jth one), write 1 in the ith entry of the register.
- Then, make n processors check the n register entries and if an entry is still 0, which means it did not lose a comparison, write its value in the output register.

-

Parallel minimum spanning tree algorithm

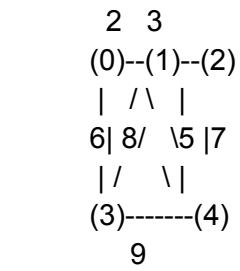
The minimum spanning tree (MST) of a graph is a tree that spans all the vertices of the graph while minimizing the sum of the weights of its edges. The problem of finding the MST of a graph is a fundamental problem in graph theory and has many applications in computer science and other fields.

The parallel minimum spanning tree algorithm is a parallel algorithm for finding the MST of a graph using a parallel computing environment. The algorithm is based on the Boruvka's algorithm, which is a sequential algorithm that repeatedly adds edges to the MST until all the vertices are connected. The parallel version of the algorithm divides the vertices into groups and computes the minimum spanning tree for each group in parallel.

Here's an overview of how the algorithm works:

1. Divide the vertices of the graph into groups of roughly equal size. Each group will be processed by a separate processor in parallel.
2. Each processor computes the minimum edge connecting its assigned vertices to any other vertex in the graph. This can be done using a sequential MST algorithm, such as Prim's or Kruskal's algorithm, or by computing the minimum edge for each vertex and taking the minimum of all these edges.
3. Each group of vertices is represented by a set of edges, where each edge connects a vertex in the group to a vertex outside the group. Each processor sends its set of edges to a central processor.
4. The central processor computes the minimum edge connecting each group to any other group and adds these edges to the MST. This can be done using a sequential MST algorithm or by computing the minimum edge for each group and taking the minimum of all these edges.
5. The vertices are divided into new groups based on the connected components of the MST so far. Each group is processed in parallel as before, until the entire graph is connected.
6. The parallel minimum spanning tree algorithm can achieve significant speedup over the sequential algorithm, especially for large graphs and large numbers of processors. However, the algorithm requires careful load balancing to ensure that each processor has roughly the same amount of work to do, and the communication overhead can be significant for certain parallel computing environments.

Let's consider the following undirected, weighted graph:



We want to find the minimum spanning tree (MST) of this graph using the parallel minimum spanning tree algorithm with 2 processors.

Step 1: Divide the vertices into groups
We can divide the vertices into two groups as follows:

Processor 1: vertices 0, 1, 2
Processor 2: vertices 3, 4

Step 2: Compute the minimum edges for each group
Each processor computes the minimum edge connecting its assigned vertices to any other vertex in the graph:

Processor 1: Minimum edge is (0, 1) with weight 2
Processor 2: Minimum edge is (3, 4) with weight 9

Step 3: Send edges to the central processor
Each processor sends its set of edges to the central processor:

Processor 1: {(0, 1), (1, 2)}
Processor 2: {(3, 4)}

Step 4: Compute the minimum edges connecting each group
The central processor computes the minimum edges connecting each group to any other group:

Minimum edge connecting group 1 to group 2 is (1, 4) with weight 7

Step 5: Divide vertices into new groups
We can divide the vertices into new groups based on the connected components of the MST so far:

Processor 1: vertices 0, 1, 2, 4
Processor 2: vertex 3

Now we repeat the process for each group:

Processor 1 computes the minimum edge connecting its assigned vertices to any other vertex in the graph, which is (1, 2) with weight 3.

Processor 2 computes the minimum edge connecting its assigned vertex to any other vertex in the graph, which is (3, 4) with weight 9.

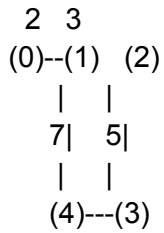
Each processor sends its set of edges to the central processor:

Processor 1: {(0, 1), (1, 2), (1, 4)}
Processor 2: {(3, 4)}

The central processor computes the minimum edges connecting each group to any other group:

No new edges are added, since the groups are already connected.

The final MST of the graph is:



This example shows how the parallel minimum spanning tree algorithm can be used to find the minimum spanning tree of a graph using a parallel computing environment. By dividing the vertices into groups and computing the minimum spanning tree for each group in parallel, we can take advantage of the parallelism offered by modern processors to speed up the computation.

On what basis did you divide the vertices into groups?

In the example I provided, I divided the vertices into two groups based on the number of processors available, since we had 2 processors:

Processor 1: vertices 0, 1, 2

Processor 2: vertices 3, 4
However, there are many different ways to divide the vertices into groups, and the optimal way to do so may depend on the characteristics of the graph and the computing environment.

For example, we might want to divide the vertices into groups based on their geographic location, if we know that vertices that are geographically close to each other are more likely to be connected by edges. Alternatively, we might want to divide the vertices into groups based on some other property of the graph, such as the degree of each vertex or the weight of its edges.

In general, the goal is to divide the vertices into groups in a way that allows us to minimize the communication overhead between processors while still ensuring that each group contains enough information to compute the minimum spanning tree.

Pointer jumping, also known as pointer chasing, is a technique used in parallel processing where a processor reads a memory address from one location, uses that address as a pointer to read another location, and repeats the process until a desired result is obtained. This technique is often used in applications such as graph traversal, matrix multiplication, and scientific simulations.