ECS-189A Special Topics in TCS April 6, 2023

Lecture 2

Lecturer: Slobodan Mitrović

1 Introduction

Over the last couple of decades, we have witnessed a massive increase in volumes of generated data. To cope with this influx of information, a number of frameworks for parallel and distributed computation have been deployed. Among the most famous frameworks are MapReduce, Flume, Spark, Hadoop, and Dryad. To rigorously study the capabilities of systems for large-scale computation, the scientific community developed a model called *Massively Parallel Computation* (MPC) [KSV10, GSZ11, BKS17].

MPC is now a de facto standard for analyzing large-scale computation from a theoretical perspective. We will study the MPC algorithm for graph algorithmic problems in the following lectures. The goal is to equip students with some fundamentals of MPC algorithms design.

2 Preliminaries

2.1 Notation

We will use $G = (V, E)$ to denote a graph on vertex set V and edge set E. If not stated otherwise, we will use n to denote $|V|$ and m to denote $|E|$.

We will use notation $\tilde{O}(f)$ to hide poly-logarithmic factors in f, i.e., $\tilde{O}(f) = O(f \cdot \operatorname{poly}\log f)$.

2.2 The MPC model

In the MPC model, when discussing a problem on an input graph $G = (V, E)$, we assume that the graph G, which has $n = |V|$ vertices, is partitioned among M machines and each machine knows only some of the edges and vertices. A key parameter of the model is the memory S per machine. Since the machines should be able to hold the graph together, we have that $M \cdot S \geq (|V| + |E|)$, and it is common to assume that this is tight up to logarithmic factors, i.e., $M \cdot S = \tilde{O}(|V| + |E|)$. We usually refer to $M \cdot S$ as *global memory* or *total memory*, while the memory per machine S is often called *local memory*.

Initially, the input consisting of the edges and vertices of the graph is divided arbitrarily among all machines, subject to the constraint that each machine holds at most S words. Computation proceeds in synchronous rounds, where per round, a machine executes (usually polynomial-time) computation on the data it holds. Afterward, there is a round of communication where each machine can send some data to every other machine – thus, the communication network among the machines is the complete graph. The only restriction on the communication is that the total amount of data that one machine sends and receives cannot exceed its local memory S. The primary complexity measure is the number of rounds used to solve a given graph problem, where the fewer, the better.

MPC regimes: For many problems, the local memory parameter S impacts the difficulty of the problem significantly — problems get harder as we reduce S. Considering this, throughout the literature, the focus has been primarily on three regimes of this MPC:

(A) *super-linear* memory regime, when $S = n^{1+c}$ for some positive constant $c > 0$,

(B) *near-linear* memory regime, when $S = \tilde{O}(n)$, and

(C) *sub-linear* memory regime, when $S = n^{1-c}$ for some positive constant $c > 0$.

Often the algorithms in the strongly super-linear regime or strongly sub-linear regime do not depend on the exact value of the constant c, and their complexity degrades by only a constant factor if we change c.

Machine IDs: Without loss of generality, we will assume that each machine knows M and that the IDs of the machines are 1 through M.

3 Connected Components in the MPC Super-linear Memory Regime

We will now design an algorithm for computing connected components (CC) in undirected graphs in the super-linear memory regime of MPC. The classical textbook algorithm for CC designed to run on a *single* machine employs depth-first-search graph traversal. Surely, that algorithm can also be simulated in MPC, but it is unclear how to simulate DFS efficiently in MPC. We will see how to solve the CC problem in $O(1/c)$ many MPC rounds when $S = n^{1+c}$.

When faced with a distributed computation, it is typically required to partition the input data somehow. Our partitioning scheme will partition the current graph arbitrarily onto the smallest possible number of machines while ensuring that no machine is sent more than S edges. After that, each machine performs some data reduction, i.e., our graph will gradually shrink. This process is repeated as long as our graph does not fit on a single machine. The entire process is described as Algorithm 1.

Input : Graph $G = (V, E)$
Space per machine S

1 $\widehat{M} \leftarrow \lceil \frac{m}{S} \rceil$ `/* We assume that originally edges are distributed across the machines with IDs` $1, \ldots, \widehat{M}$. `Note that` m/S `machines suffice to hold all the edges. */`

2 Label each edge $e \in E$ as "active".

3 **while** $\widehat{M} > 1$ **do**

4 **for** *ID* $i \in \left\{1, \ldots, \widehat{M}\right\}$ *independently* **do**

5 Compute a spanning forest F_i on the "active" edges that machine i contains.

6 All edges on machine i not in F_i are labeled as "inactive".

7 Send all the active edges on machine i to machine with ID equal to $\lceil \frac{i}{n^c} \rceil$.

8 $\widehat{M} \leftarrow \left\lceil \frac{\widehat{M}}{n^c} \right\rceil$

9 Output the connected components of the "active" edge-set on the machine with ID $= 1$.

Algorithm 1: An algorithm for computing connected components in the super-linear memory regime of MPC.

In Algorithm 1, the variable $\widehat{M}$ should be thought of as the number of machines used to distribute the currently "active" edges. Observe that $\widehat{M}$ decreases from step to step. Intuitively, this makes sense as Line 6 reduces the number of "active" edges, and hence fewer machines are needed to store them. When only a single machine suffices to store all the active edges, we can simply output the connected components of those edges, as done by Line 9.

We will now analyze the correctness and efficiency of Algorithm 1. To analyze correctness, we need to address the following: no machine will send more than S words of data in a single round; no machine will receive more than S words of data in a single round; and the output indeed corresponds to the connected components of the input graph G. To analyze efficiency, it suffices to comment on the number of rounds.

3.1 Round complexity

Lemma 1. *Algorithm 1 can be executed in $O(1/c)$ MPC rounds.*

Proof. First, each iteration of the while-loop can be executed in $O(1)$ rounds. To see that, observe that Lines 5 and 6 are done locally on each machine, so it can be executed in a single round of computation. Line 7 is done in the same round of computation as the previous two lines – each machine creates a message that it wants to send to other machines and then sends those messages at once.

Second, it remains to analyze the number of iterations of the while-loop. Note that the value of $\widehat{M}$ drives it. Since $\widehat{M}$ is initialized to $O(m/S) \in O(n^2/S)$ and in each iteration it gets reduced by a factor of $\Theta(n^c)$, we have that within $O(2/c) = O(1/c)$ iterations the value of $\widehat{M}$ will become 1 or less. □

3.2 Outgoing and incoming message sizes

It is trivial to argue that no machine sends more than the number of edges it has in its memory – it directly follows from Line 7 and the fact that a spanning forest has at most $n - 1 \ll n^{1+c}$ many edges.

It remains to show that no machine receives more than n^{1+c} edges. The machines that send edges to machine j have IDs i such that $j = \lceil i/n^c \rceil$. It means that $i/n^c \leq j < i/n^c + 1$. This further implies that $(j-1)n^c < i \leq jn^c$. That is, there are at most n^c machines that send a message to machine j. Each such machine sends at most $n - 1$ edges, which amounts to at most n^{1+c} edges sent to machine j.

3.3 Correctness

It remains to show that the final output represents CCs of G.

Lemma 2. *Let CC_{final} be the CC output by Line 9 of Algorithm 1 for an input graph G. CC_{final} are the CCs of G as well.*

Proof. The statement of this claim is equivalent to: vertices u and v are in the same CC of G iff they are in the same CC of CC_{final}. We prove the two directions of "iff" separately.

Direction: If u and v are in the same CC of CC_{final}, then they are in the same CC of G. If u and v are in the same CC of CC_{final}, then machine 1 after the while-loop contains a path P between u and v. Since each of the edges of P belong to G, a path in G connects u and v. Hence, they belong to the same CC of G.

Direction: If u and v are in the same CC of G, then they are in the same CC of CC_{final}. Toward a contradiction, assume that u and v are not in the same CC of CC_{final}.

Let iteration t be *the last* one of the while-loop in whose beginning u and v were in the same connected component if one considers only "active" edges. Let P be a $u - v$ path among "active" edges at the beginning of iteration t. Since we are assuming u and v are not connected via "active"

edges in iteration $t+1$, there exists at least one edge of P which became "inactive" during iteration t. Consider each such edge $e = \{x, y\}$ of P separately.

Let machine i have e at the beginning of iteration t. Recall that each machine is finding a spanning forest on its set of edges, and each of the edges in the spanning forest remains "active" throughout that iteration. So, if machine i marks as "inactive" the edge e, its spanning forest will contain a path between x and y, i.e., between the endpoints of e, consisting of "active" edges only. But this implies that even after marking some edges as "inactive" in iteration t, the vertices u and v will remain connected via "active" edges and hence connected at the beginning of iteration $t+1$. This contradicts our assumption that there exists t which is the last such iteration, and hence proves that u and v are in the same CC of CC_{final} as well. □

References

[BKS17] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):1–58, 2017.

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*, volume 7074, pages 374–383. Springer, 2011.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.