Chapter 12
Conclusions

We have reached the end of this course. In this chapter we will review what we have learned, and we will also have a brief look at what else is there in the field of distributed algorithms. The exercises of this chapter form a small research project related to the distributed complexity of locally verifiable problems.

12.1 What Have We Learned?

By now, you have learned a new mindset—an entirely new way to think about computation. You can reason about distributed systems, which requires you to take into account many challenges that we do not encounter in basic courses on algorithms and data structures:

- Dealing with *unknown systems*: you can design algorithms that work correctly in any computer network, no matter how the computers are connected together, no matter how we choose the port numbers, and no matter how we choose the unique identifiers.
- Dealing with *partial information*: you can solve graph problems in sublinear time, so that each node only sees a small part of the network, and nevertheless the nodes produce outputs that are globally consistent.

- Dealing with *parallelism*: you can design highly parallelized algorithms, in which several nodes take steps simultaneously.

These skills are in no way specific to distributed algorithms—they play a key role also in many other areas of modern computer science. For example, dealing with unknown systems is necessary if we want to design *fault-tolerant* algorithms, dealing with partial information is the key element in e.g. *online algorithms* and *streaming algorithms*, and parallelism is the cornerstone of any algorithm that makes the most out of modern *multicore CPUs*, *GPUs*, and *computing clusters*.

12.2 What Else Exists?

Distributed computing is a vast topic and so far we have merely scratched the surface. This course has focused on what is often known as *distributed graph algorithms* or *network algorithms*, and we have only focused on the most basic models of distributed graph algorithms. There are many questions related to distributed computing that we have not addressed at all; here are a few examples.

12.2.1 Distance vs. Bandwidth vs. Local Memory

Often we would like to understand computation in two different kinds of distributed systems:

(a) *Geographically distributed networks*, e.g. the Internet. A key challenge is large distances and communication latency: some parts of the input are physically far away from you, so in a fast algorithm you have to act based on the information that is available in your local neighborhood.

(b) *Big data systems*, e.g. data processing in large data centers. Typically all input data is nearby (e.g. in the same local-area network). However, this does not make problems trivial to solve fast: individual computers have a limited *bandwidth* and limited amount of *local memory*.

The LOCAL model is well-suited for understanding computation in networks, but it does not make much sense in the study of big data systems: if all information is available within one hop, then in LOCAL model it would imply that everything can be solved in one communication round!

Congested Clique Model. Many other models have been developed to study big data systems. From our perspective, perhaps the easiest to understand is the *congested clique* model [3, 7, 9]. In brief, the model is defined as follows:

- We work in the CONGEST model, as defined in Chapter 5.
- We assume that the underlying graph G is the complete graph on n nodes, i.e., an n-clique. That is, every node is within one hop from everyone else.

Here it would not make much sense to study graph problems related to G itself, as the graph is fixed. However, here each node v gets some local input $f(v)$, and it has to produce some local output $g(v)$. The local inputs may encode e.g. some input graph $H \neq G$ (for example, $f(v)$ indicates which nodes are adjacent to v in H), but here it makes also perfect sense to study other computational problem that are not related to graphs. Consider, for example, the task of computing the matrix product $X = AB$ of two $n \times n$ matrices A and B. Here we may assume that initially each node knows one column of A and one column of B, and when the algorithms stops, each node has to hold one column of X.

Other Big Data Models. There are many other models of big data algorithms that have similar definitions—all input data is nearby, but communication bandwidth and/or local memory is bounded; examples include:

- *BSP model* (bulk-synchronous parallel) [10],
- *MPC model* (massively parallel computation) [5], and

- *k-machine model* [6].

Note that when we limit the amount of local memory, we also implicitly limit communication bandwidth (you can only send what you have in your memory). Conversely, if you have limited communication bandwidth and a fast algorithm, you do not have time to accumulate a large amount of data in your local memory, even if it was unbounded. Hence all of these model and their variants often lead to similar algorithm design challenges.

12.2.2 Asynchronous and Fault-Tolerant Algorithms

So far in this course we have assumed that all nodes start at the same time, computation proceeds in a synchronous manner, and all nodes always work correctly.

Synchronization. If we do not have any failures, it turns out we can easily adapt all of the algorithms that we have covered in this course also to asynchronous settings. Here is an example of a very simple solution, known as the *α-synchronizer* [1]: all messages contain a piece of information indicating "this is my message for round i", and each node first waits until it has received all messages for round i from its neighbors before it processes the messages and switches to round $i + 1$.

Crash Faults and Byzantine Faults. Synchronizers no longer work if nodes can fail. If we do not have any bounds on the relative speeds of the communication channels, it becomes impossible to distinguish between e.g. a node behind a very slow link and a node that has *crashed*.

Failures are challenging even if we work in a synchronous setting. In a synchronous setting it is easy to tell if a nodes has crashed, but if some nodes can misbehave in an arbitrary manner, many seemingly simple tasks become very difficult to solve. The term *Byzantine failure* is commonly used to refer to a node that may misbehave in an arbitrary manner, and in the quiz (Section 12.3) we will explore some challenges of solving the *consensus problem* in networks with Byzantine nodes.

Self-Stabilization. Another challenge is related to consistent initialization. In our model of computing, we have assumed that all nodes are initialized by using the $\text{init}_{A,d}$ function. However, it would be great to have algorithms that converge to a correct output even if the initial states of the nodes may have been corrupted in an arbitrary manner. Such algorithms are called *self-stabilizing* algorithms [4].

If we have a deterministic T-time algorithms A designed in the LOCAL model, it can be turned into a self-stabilizing algorithm in a mechanical manner [8]: all nodes keep track of T possible states, indicating "what would be my state if now was round i", they send vectors of T messages, indicating "what would be my message for round i", and they repeatedly update their states according to the messages that they receive from their neighbors. However, if we tried to do something similar for a randomized algorithm, it would no longer converge to a fixed output—there are problems that are easy to solve with randomized Monte Carlo LOCAL algorithms, but difficult to solve with self-stabilizing algorithms.

12.2.3 Other Directions

Shared Memory. Our models of computing can be seen as a *message-passing system*: nodes send messages (data packets) to each other. A commonly studied alternative is a system with *shared memory*: each node has a shared register, and the nodes can communicate with each other by reading and writing the shared registers.

Physical Models. Our models of computing are a good match with systems in which computers are connected to each other by physical wires. If we connect the nodes by wireless links, the *physical properties of radio waves* (e.g., reflection, refraction, multipath propagation, attenuation, interference, and noise) give rise to new models and new algorithmic challenges. The *physical locations* of the nodes as well as the properties of the environment become relevant.

Robot Navigation. In our model, the nodes are active computational entities, and they cannot move around in the network—they can only send information around in the network. Another possibility is to study computation with autonomous agents ("robots") that can move around in the network. Typically, the nodes are passive entities (corresponding to possible physical locations), and the robots can communicate with each other by e.g. leaving some tokens in the nodes.

Nondeterministic Algorithms. Just like we can study nondeterministic Turing machines, we can study nondeterministic distributed algorithms. In this setting, it is sufficient that there exists a *certificate* that can be verified efficiently in a distributed setting; we do not need to construct the certificate efficiently. Locally verifiable problems that we have studied in this course are examples of problems that are easy to solve with nondeterministic algorithms.

Complexity Measures. For us the main complexity measure has been the number of synchronous communication rounds. Many other possibilities exist: e.g., how many messages do we need to send in total?

Practical Aspects of Networking. This course has focused on the theory of distributed algorithms. There is of course also the practical side: We need physical computers to run our algorithms, and we need networking hardware to transmit information between computers. We need modulation techniques, communication protocols, and standardization to make things work together, and good software engineering practices, programming languages, and reusable libraries to keep the task of implementing algorithms manageable. In the real world, we will also need to worry about privacy and security. There is plenty of room for research in computer science, telecommunications engineering, and electrical engineering in all of these areas.

12.2.4 Research in Distributed Algorithms

There are two main conferences related to the theory of distributed computing:

- PODC, the ACM Symposium on Principles of Distributed Computing: https://www.podc.org/
- DISC, the International Symposium on Distributed Computing: http://www.disc-conference.org/

The proceedings of the recent editions of these conferences provide a good overview of the state-of-the-art of this research area.

12.3 Quiz

In the *binary consensus problem* the task is this: Each node gets 0 or 1 as input, and each node has to produce 0 or 1 as output. All outputs must be the same: you either have to produce all-0 or all-1 as output. Moreover, if the input is all-0, your output has to be all-0, and if the input is all-1, your output has to be all-1. For mixed inputs either output is fine.

Your network is a complete graph on 5 nodes; we work in the usual LOCAL model. You are using the following 1-round algorithm:

- Each node sends its input to all other nodes. This way each node knows all inputs.
- Each node applies the majority rule: if at least 3 of the 5 inputs are 1s, output 1, otherwise output 0.

This algorithm clearly solves consensus if all nodes correctly follow this algorithm. Now assume that node 5 is controlled by a *Byzantine adversary*, while nodes 1–4 correctly follow this algorithm. Show that now this algorithm *fails* to solve consensus among the correct nodes 1–4, i.e., there is some input so that the adversary can force nodes 1–4 to

produce *inconsistent* outputs (at least one of them will output 0 and at least one of them will output 1).

Your answer should give the inputs of nodes 1–4 (4 bits), the messages sent by node 5 to nodes 1–4 (4 bits), and the outputs of nodes 1–4 (4 bits). An answer with these three bit strings is sufficient.

12.4 Exercises

In Exercises 12.1–12.4 we use the tools that we have learned in this course to study *locally verifiable problems* in cycles in the LOCAL model (both deterministic and randomized). For the purposes of these exercises, we use the following definitions:

A locally verifiable problem $\Pi = (\Sigma, \mathbf{C})$ consists of a finite alphabet Σ and a set $\mathbf{C}$ of allowed *configurations* (x, y, z): $x, y, z \in \Sigma$. An assignment $\varphi : V \to \Sigma$ is a *solution* to Π if and only if for each node u and its two neighbors v, w it holds that

$$\big(\varphi(v), \varphi(u), \varphi(w)\big) \in \mathbf{C} \quad \text{or} \quad \big(\varphi(w), \varphi(u), \varphi(v)\big) \in \mathbf{C}.$$

Put otherwise, if we look at any three consecutive labels x, y, z in the cycle, either (x, y, z) or (z, y, x) has to be an allowed configuration.

You can assume in Exercises 12.1–12.4 that the value of n is known (but please then make the same assumption consistently throughout the exercises, both for positive and negative results).

Exercise 12.1 (trivial and non-trivial problems)**.** We say that a locally verifiable problem $\Pi_0 = (\Sigma_0, \mathbf{C}_0)$ is *trivial*, if $(x, x, x) \in \mathbf{C}_0$ for some $x \in \Sigma_0$. We define that *weak c-coloring* is the problem $\Pi_1 = (\Sigma_1, \mathbf{C}_1)$ with

$$\Sigma_1 = \{1, 2, \ldots, c\},$$
$$\mathbf{C}_1 = \big\{(x_1, x_2, x_3) \;\big|\; x_1 \neq x_2 \text{ or } x_3 \neq x_2\big\}.$$

That is, each node must have at least one neighbor with a different color.

(a) Show that if a problem is trivial, then it can be solved in constant time.

(b) Show that if a problem is not trivial, then it is at least as hard as weak c-coloring for some c.

▷ *hint A*

Exercise 12.2 (hardness of weak coloring)**.** Consider the weak c-coloring problem, as defined in Exercise 12.1.

(a) Show that weak 2-coloring can be solved in cycles in $O(\log^* n)$ rounds.

(b) Show that weak c-coloring, for any $c = O(1)$, requires $\Omega(\log^* n)$ rounds in cycles.

▷ *hint B*

What does this imply about the possible complexities of locally verifiable problems in cycles?

Exercise 12.3 (randomized constant time)**.** Show that if a locally verifiable problem Π can be solved in constant time in cycles with a randomized LOCAL-algorithm, then it can be solved in constant time with a deterministic LOCAL-algorithm.

▷ *hint C*

Exercise 12.4 (deterministic speed up)**.** Assume that there is a deterministic algorithm A for solving problem Π in cycles with a running time $T(n) = o(n)$. Show that there exists a deterministic algorithm A' for solving Π in $O(\log^* n)$ rounds.

What does this imply about the possible complexities of locally verifiable problems in cycles?

▷ *hint D*

⋆⋆ **Exercise 12.5.** Prove or disprove: vertex coloring with $\Delta+1$ colors in graphs of maximum degree Δ can be solved in $O(\log \Delta + \log^* n)$ rounds in the LOCAL model.

▷ *hint E*

12.5 Bibliographic Notes

The exercises in this chapter are inspired by Chang and Pettie [2].

12.6 Bibliography

[1] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

[2] Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. *SIAM Journal on Computing*, 48(1):33–69, 2019. arXiv:1704.06297, doi:10.1137/17M1157957.

[3] Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, tri again": Finding triangles and small subgraphs in a distributed setting. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, 2012. arXiv:1201.6652, doi:10.1007/978-3-642-33651-5_14.

[4] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[5] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*. 2010. doi:10.1137/1.9781611973075.76.

[6] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, 2015. doi:10.1137/1.9781611973730.28.

[7] Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proc. 32nd Annual ACM symposium on Principles of Distributed Computing (PODC 2013)*, 2013. arXiv:1207.1852, doi:10.1145/2484239.2501983.

[8] Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: self-stabilization on speed. In *Proc. 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, 2009. doi:10.1007/978-3-642-05118-0_2.

[9] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.

[10] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. doi:10.1145/79173.79181.

12.7 Hints

A. Show that if a problem is not trivial, then each configuration must use a distinct adjacent label.

B. Give an algorithm for turning a weak c-coloring into a 3-coloring in $O(c)$ rounds.

C. Use an argument to boost the failure probability of a constant-time randomized algorithm. A constant-time algorithm cannot depend on the size of the input. Consider a network N such that a randomized algorithm succeeds with probability $p < 1$. Boost this by considering the same algorithm on network N' that consists of many copies of N.

D. Use a similar argument as in Exercise 11.4. In this case we want a speedup simulation in the LOCAL model, so we also need to simulate the identifiers. Color the network so that the colors look locally like unique identifiers. Then simulate algorithm A using the colors instead of real identifiers.

Since A runs in time $o(n)$, we can find a *constant* n_0 such that $T(n_0) \ll n_0$. On any network with $n > n_0$ nodes, find a coloring with n_0-colors such that two nodes with the same color have distance at least $2T(n_0) + 3$. Show that this can be done in $O(\log^* n)$ rounds. Then run A on N, using the coloring instead of unique identifiers. Show that this simulation can be done in constant time. Show that this simulation is correct in every 1-neighborhood.

E. This is an open research question.