Distributed Computing

BY

Panagiota Fatourou

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

AND BY

Stefan Schmid

Technical University of Berlin
D-10587 Berlin, Germany
and
Telekom Innovation Laboratories (T-Labs)
Ernst Reuter Platz 7, D - 10587 Berlin, Germany
schmiste@gmail.com

After serving for more than 6 years as the editor of the Distributed Computing Column (DC) of the Bulletin of the European Association for Theoretical Computer Science (BEATCS), I am glad to leave the column in the hands of Stefan Schmid, who will be the next editor. Stefan and I will co-edit the DC column for this and the next issue of BEATCS and he will replace me right after. I wish Stefan that his tenure as editor will be as pleasurable as mine was. I am really grateful to the authors that have contributed to the column during these years and I deeply thank them for their great contributions!

Panagiota Fatourou

I am very happy to take over the responsibility for the Distributed Computing column of the BEATCS. My goal is to continue providing an interesting column format where researchers have the opportunity to give survey-like but technical overviews of recent advances in distributed computing, written for a broad TCS audience. I would like to thank Panagiota for her great work and commitment over the last years, and I am very proud to be able to co-edit two issues together with her. I wish Panagiota all the best for the future. I would also like to thank Jukka Suomela for his excellent survey on recent advances in lower bounds for distributed graph algorithms. page).

Stefan Schmid

A Closer Look at Concurrent Data Structures and Algorithms

Gadi Taubenfeld
The Interdisciplinary Center, P.O.Box 167, Herzliya 46150 Israel
tgadi@idc.ac.il

Abstract

In this survey article, I will present three ideas, regarding the construction of concurrent data structures and algorithms, recently published in [27, 28, 29].

The first idea is that of *contention-sensitivity*. A contention-sensitive data structure is a concurrent data structure in which the overhead introduced by locking is eliminated in common cases, when there is no contention, or when processes with non-interfering operations access it concurrently. This notion is formally defined, several contention-sensitive data structures are presented, and transformations that facilitate devising such data structures are discussed.

The second idea is the introduction of a new synchronization problem, called *fair synchronization*. Solving the new problem enables to automatically add strong fairness guarantees to existing implementations of concurrent data structures, without using locks, and to transform any solution to the mutual exclusion problem into a fair solution.

The third idea is a generalization of the traditional notion of *fault tolerance*. Important classical problems have no asynchronous solutions which can tolerate even a single fault. It is shown that while some of these problems have solutions which guarantee that in the presence of *any* number of faults *most* of the correct processes will terminate, other problems do not even have solutions which guarantee that in the presence of just *one* fault at least one correct process terminates.

1 Introduction

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section

code, within which the process is guaranteed exclusive access. Any sequential data structure can be easily made concurrent using such a locking approach. The popularity of this approach is largely due to the apparently simple programming model of such locks, and the availability of lock implementations which are reasonably efficient.

Using locks may, in various scenarios, degrade the performance of concurrent applications, as it enforces processes to wait for a lock to be released. Moreover, slow or stopped processes may prevent other processes from ever accessing the data structure. Locks can introduce false conflicts, as different processes with non-interfering operations contend for the same lock, only to end up accessing disjoint data. This motivates attempts to design data structures that avoid locking.

The advantages of concurrent data structures and algorithms which completely avoid locking are that they are not subject to priority inversion, they are resilient to failures, and they do not suffer significant performance degradation from scheduling preemption, page faults or cache misses. On the other hand, such algorithms may impose too much overhead upon the implementation and are often complex and memory consuming.

We consider an intermediate approach for the design of concurrent data structures. A *contention-sensitive* data structure is a concurrent data structure in which the overhead introduced by locking is eliminated in common cases, when there is no contention, or when processes with non-interfering operations access it concurrently. When a process invokes an operation on a contention-sensitive data structure, in the absence of contention or interference, the process must be able to complete its operation in a small number of steps and without using locks. Using locks is permitted only when there is interference [27]. In Section 2, the notion of contention-sensitivity is formally defined, several contention-sensitive data structures are presented, and transformations that facilitate devising such data structures are discussed.

Most published concurrent data structures which completely avoid locking do not provide any fairness guarantees. That is, they allow processes to access a data structure and complete their operations arbitrarily many times before some other trying process can complete a single operation. In Section 3, we show how to automatically transfer a given concurrent data structure which avoids locking and waiting into a similar data structure which satisfies a strong fairness requirement, without using locks and with limited waiting. To achieve this goal, we introduce and solve a *new* synchronization problem, called *fair synchronization* [29]. Solving the new problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any lock (i.e., a solution to the mutual exclusion problem) into a fair lock.

In Section 4, we generalize the traditional notion of fault tolerance in a way which enables to capture more sensitive information about the resiliency of a con-

current algorithm. Intuitively, an algorithm that, in the presence of any number of faults, always guarantees that all the correct processes, except maybe one, successfully terminate (their operations), is more resilient to faults than an algorithm that in the presence of a single fault does not even guarantee that a single correct process ever terminates. However, according to the standard notion of fault tolerance both algorithms are classified as algorithms that can not tolerate a single fault. Our general definition distinguishes between these two cases.

It is well known that, in an asynchronous system where processes communicate by reading and writing atomic read/write registers important classical problems such as, consensus, set-consensus, election, perfect renaming, implementations of a test-and-set bit, a shared stack, a swap object and a fetch-and-add object, have no deterministic solutions which can tolerate even a single fault. In Section 4, we show that while, some of these classical problems have solutions which guarantee that in the presence of *any* number of faults most of the correct processes will successfully terminate; other problems do not even have solutions which guarantee that in the presence of just *one* fault at least one correct process terminates.

Our model of computation consists of an asynchronous collection of n processes which communicate asynchronously via shared objects. Asynchrony means that there is no assumption on the relative speeds of the processes. Processes may fail by crashing, which means that a failed process stops taking steps forever. In most cases, we assume that processes communicate by reading and writing atomic registers. By atomic registers we always mean atomic read/write registers. In few cases, we will also consider stronger synchronization primitives. With an atomic register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action.

In a model where participation is required, every correct process must eventually execute its code. A more interesting and practical situation is one in which participation is *not* required, as assumed when solving resource allocation problems or when designing concurrent data structures. In this paper we always assume that participation is not required.

2 Contention-sensitive Data Structures

2.1 Motivation

As already mentioned in Section 1, using locks may, in various scenarios, degrade the performance of concurrent data structures. On the other hand, concurrent data structures which completely avoid locking may impose too much overhead upon the implementation and are often complex and memory consuming.

We propose an intermediate approach for the design of concurrent data structures. While the approach guarantees the correctness and fairness of a concurrent data structure under all possible scenarios, it is especially efficient in common cases when there is no (or low) contention, or when processes with non-interfering operations access a data structure concurrently.

2.2 Contention-sensitive data structures: The basic idea

Contention for accessing a shared object is usually rare in well designed systems. Contention occurs when multiple processes try to acquire a lock at the same time. Hence, a desired property in a lock implementation is that, in the absence of contention, a process can acquire the lock extremely fast, without unnecessary delays. Furthermore, such fast implementations decrease the possibility that processes which invoke operations on the same data structure in about the same time but not simultaneously, will interfere with each other. However, locks were introduced in the first place to resolve conflicts when there is contention, and acquiring a lock *always* introduces some overhead, even in the cases where there is no contention or interference.

We propose an approach which, in common cases, eliminates the overhead involved in acquiring a lock. The idea is simple: assume that, for a given data structure, it is known that in the absence of contention or interference it takes some fixed number of steps, say at most 10 steps, to complete an operation, not counting the steps involved in acquiring and releasing the lock. According to our approach, when a process invokes an operation on a given data structure, it first tries to complete its operation, by executing a short code, called the *shortcut code*, which does not involve locking. Only if it does not manage to complete the operation fast enough, i.e., within 10 steps, it tries to access the data structure via locking. The shortcut code is required to be *wait-free*. That is, its execution by a process takes only a finite number of steps and always terminates, regardless of the behavior of the other processes.

Using an efficient shortcut code, although eliminates the overhead introduced by locking in common cases, introduces a major problem: we can no longer use a sequential data structure as the basic building block, as done when using the traditional locking approach. The reason is simple, many processes may access the same data structure simultaneously by executing the shortcut code. Furthermore, even when a process acquires the lock, it is no longer guaranteed to have exclusive access, as another process may access the same data structure simultaneously by executing the shortcut code.

Thus, a central question which we are facing is: if a sequential data structure can not be used as the basic building block for a general technique for constructing a contention-sensitive data structure, then what is the best data structure to use?

Before we proceed to discuss formal definitions and general techniques, which will also help us answering the above question, we demonstrate the idea of using a shortcut code (which does not involve locking), by presenting a contention-sensitive solution to the binary consensus problem using atomic read/write registers and a single lock.

2.3 A simple example: Contention-sensitive consensus

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. A consensus algorithm is an algorithm that produces such an agreement. While various decision rules can be considered such as "majority consensus", the problem is interesting even when the decision value is constrained only when all processes are unanimous in their opinions, in which case the decision value must be the common opinion. A consensus algorithm is called *binary* consensus when the number of possible initial opinions is two.

Processes are not required to participate in the algorithm. However, once a process starts participating it is guaranteed that it may fail only while executing the shortcut code. We assume that processes communicate via atomic registers. The algorithm uses an array $x[0..1]$ of two atomic bits, and two atomic registers y and *out*. After a process executes a **decide**() statement, it immediately terminates.

CONTENTION-SENSITIVE BINARY CONSENSUS: program for process p_i with input $in_i \in \{0, 1\}$.

shared $x[0..1]$: array of two atomic bits, initially both 0
y, *out* : atomic registers which range over $\{\perp, 0, 1\}$, initially both $\perp$

1 $x[in_i] := 1$ // start shortcut code
2 **if** $y = \perp$ **then** $y := in_i$ **fi**
3 **if** $x[1 - in_i] = 0$ **then** $out := in_i$; **decide**(in_i) **fi**
4 **if** $out \neq \perp$ **then decide**(out) **fi** // end shortcut code
5 lock **if** $out = \perp$ **then** $out := y$ **fi** unlock ; **decide**(out) // locking

When a process runs alone (either before or after a decision is made), it reaches a decision after accessing the shared memory at most five times. Furthermore, when all the concurrently participating processes have the same preference – i.e., when there is no interference – a decision is also reached within five steps and without locking. Two processes with conflicting preferences, which run at the same time, will not resolve the conflict in the shortcut code if both of them find $y = \perp$. In such a case, some process acquires the lock and sets the value of *out* to be the final decision value. The assignment $out := y$ requires two memory references and hence it involves two atomic steps.

2.4 Progress conditions

Numerous implementations of locks have been proposed over the years to help coordinating the activities of the various processes. We are not interested in implementing new locks, but rather assume that we can use existing locks. We are not at all interested whether the locks are implemented using atomic registers, semaphores, etc. We do assume that a lock implementation guarantees that: (1) no two processes can acquire the same lock at the same time, (2) if a process is trying to acquire the lock, then in the absence of failures some process, not necessarily the same one, eventually acquires that lock, and (3) the operation of releasing a lock is wait-free.

Several progress conditions have been proposed for data structures which avoid locking, and in which processes may fail by crashing. *Wait-freedom* guarantees that *every* active process will always be able to complete its pending operations in a finite number of steps [8]. *Non-blocking* (which is sometimes called lock-freedom) guarantees that *some* active process will always be able to complete its pending operations in a finite number of steps [13]. *Obstruction-freedom* guarantees that an active process will be able to complete its pending operations in a finite number of steps, if all the other processes "hold still" long enough [9]. Obstruction-freedom does not guarantee progress under contention.

Several progress conditions have been proposed for data structures which may involve waiting. *Livelock-freedom* guarantees that processes not execute forever without making forward progress. More formally, livelock-freedom guarantees that, in the absence of process failures, if a process is active, then *some* process, must eventually complete its operation. A stronger property is *starvation-freedom* which guarantees that each process will eventually make progress. More formally, starvation-freedom guarantees that, in the absence of process failures, every active process must eventually complete its operation.

2.5 Defining contention-sensitive data structures

An implementation of a contention-sensitive data structure is divided into *two* continuous sections of code: the *shortcut code* and the *body code*. When a process invokes an operation it first executes the shortcut code, and if it succeeds to complete the operation, it returns. Otherwise, the process tries to complete its operation by executing the body code, where it usually first tries to acquire a lock. If it succeeds to complete the operation, it releases the acquired lock(s) and returns. The problem of implementing a contention-sensitive data structure is to write the *shortcut code* and the *body code* in such a way that the following *four* requirements are satisfied,

- **Fast path:** In the absence of contention or interference, each operation must

be completed while executing the shortcut code only.

- **Wait-free shortcut:** The shortcut code must be wait-free – its execution should require only a finite number of steps and must always terminate. (Completing the shortcut code does not imply completing the operation.)

- **Livelock-freedom:** In the absence of process failures, if a process is executing the shortcut code or the body code, then some process, not necessarily the same one, must eventually complete its operation.

- **Linearizability:** Although operations of concurrent processes may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in their "real-time" order.

In [27], several additional desirable properties are defined.

2.6 An example: Contention-sensitive election

The *election problem* is to design an algorithm in which all participating processes choose one process as their leader. More formally, each process that starts participating eventually decides on a value from the set $\{0, 1\}$ and terminates. It is required that exactly one of the participating processes decides 1. The process that decides 1 is the elected leader. Processes are not required to participate. However, once a process starts participating it is guaranteed that it will not fail. It is known that in the presence of one crash failure, it is not possible to solve election using only atomic read/write registers [18, 31].

The following algorithm solves the election problem for any number of processes, and is related to the splitter constructs from [14, 16, 17]. A single lock is used. It is assumed that after a process executes a **decide**() statement, it immediately terminates.

CONTENTION-SENSITIVE ELECTION: Process i's program

shared x, z: atomic registers, initially $z = 0$ and the initial value of x is immaterial
b, y, *done*: atomic bits, initially all 0
local *leader*: local register, the initial value is immaterial

```
1   x := i                                              // begin shortcut
2   if y = 1 then b := 1; decide(0) fi                  // I am not the leader
3   y := 1
4   if x = i then z := i; if b = 0 then decide(1) fi fi // I am the leader!
                                                        // end shortcut
```

```
5   lock                                                          // locking
6   if z = i ∧ done = 0 then leader = 1                           // I am the leader!
7       else await b ≠ 0 ∨ z ≠ 0
8            if z = 0 ∧ done = 0 then leader = 1; done := 1       // I am the leader!
9                 else leader = 0                                  // I am not the leader
10           fi
11  fi
12  unlock ; decide(leader)                                       // unlocking
```

When a process runs alone before a leader is elected, it is elected and terminates after accessing the shared memory *six* times. Furthermore, all the processes that start running *after* a leader is elected terminate after three steps. The algorithm does not satisfy the disable-free shortcut property: a process that fails just before the assignment to b in line 2 or fails just before the assignment to z in line 4, may prevent other processes spinning in the *await* statement (line 7) from terminating.

2.7 Additional results

The following additional results are presented in [27].

- A contention-sensitive double-ended queue. To increase the level of concurrency, *two* locks are used: one for the left-side operations and the other for the right-side operations

- Transformations that facilitate devising contention-sensitive data structures. The first transformation converts any contention-sensitive data structure which satisfies livelock-freedom into a corresponding contention-sensitive data structure which satisfies starvation-freedom. The second transformation, converts any obstruction-free data structure into the corresponding contention-sensitive data structure which satisfies livelock-freedom.

- Finally, the notion of a *k-contention-sensitive* data structure in which locks are used only when contention goes above k is presented. This notion is illustrated by implementing a 2-contention-sensitive consensus algorithm. Then, for each $k \geq 1$, a progress condition, called *k-obstruction-freedom*, is defined and a transformation is presented that converts any k-obstruction-free data structure into the corresponding k-contention-sensitive data structure which satisfies livelock-freedom.

3 Fair Synchronization

3.1 Motivation

As already discussed in Section 1, using locks may degrade the performance of synchronized concurrent applications, and hence much work has been done on the design of wait-free and non-blocking data structures. However, the wait-freedom and non-blocking progress conditions do not provide fairness guarantees. That is, such data structures may allow processes to complete their operations arbitrarily many times before some other trying process can complete a single operation. Such a behavior may be prevented when fairness is required. However, fairness requires waiting or helping.

Using helping techniques (without waiting) may impose too much overhead upon the implementation, and are often complex and memory consuming. Does it mean that for enforcing fairness it is best to use locks? The answer is negative. We show how any wait-free and any non-blocking implementation can be automatically transformed into an implementation which satisfies a very strong fairness requirement without using locks and with limited waiting.

We require that no beginning process can complete two operations on a given resource while some other process is kept waiting on the same resource. Our approach, allows as many processes as possible to access a shared resource at the same time as long as fairness is preserved. To achieve this goal, we introduce and solve a new synchronization problem, called *fair synchronization*. Solving the fair synchronization problem enables us to add fairness to existing implementations of concurrent data structures, and to transform any solution to the mutual exclusion problem into a fair solution.

3.2 The fair synchronization problem

The fair synchronization problem is to design an algorithm that guarantees fair access to a shared resource among a number of participating processes. Fair access means that no process can access a resource twice while some other trying process cannot complete a single operation on that resource. There is no a priori limit (smaller than the number of processes n) on the number of processes that can access a resource simultaneously. In fact, a desired property is that as many processes as possible will be able to access a resource at the same time as long as fairness is preserved.

It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, fair and exit. Furthermore, it is assumed that the entry section consists of two parts. The first part, which is called the *doorway*, is *fast wait-free*:

its execution requires only a (very small) *constant* number of steps and hence always terminates; the second part is the *waiting* part which includes (at least one) loop with one or more statements. Like in the case of the doorway, the exit section is also required to be fast wait-free. A *waiting* process is a process that has finished its doorway code and reached the waiting part of its entry section. A *beginning* process is a process that is about to start executing its entry section.

A process is *enabled* to enter its fair section at some point in time, if sufficiently many steps of that process will carry it into the fair section, independently of the actions of the other processes. That is, an enabled process does not need to wait for an action by any other process in order to complete its entry section and to enter its fair section, nor can an action by any other process prevent it from doing so.

The **fair synchronization problem** is to write the code for the entry and the exit sections in such a way that the following three basic requirements are satisfied.

- **Progress:** *In the absence of process failures and assuming that a process always leaves its fair section, if a process is trying to enter its fair section, then some process, not necessarily the same one, eventually enters its fair section.*

  The terms deadlock-freedom and livelock-freedom are used in the literature for the above progress condition, in the context of the mutual exclusion problem.

- **Fairness:** *A beginning process cannot execute its fair section twice before a waiting process completes executing its fair and exit sections once. Furthermore, no beginning process can become enabled before an already waiting process becomes enabled.*

  It is possible that a beginning process and a waiting process will become enabled at the same time. However, no beginning process can execute its fair section twice while some other process is kept waiting. The second part of the fairness requirement is called *first-in-first-enabled*. The term *first-in-first-out* (FIFO) fairness is used in the literature for a slightly stronger condition which guarantees that: no beginning process can pass an already waiting process. That is, no beginning process can enter its fair section before an already waiting process does so.

- **Concurrency:** *All the waiting processes which are not enabled become enabled at the same time.*

It follows from the *progress* and *fairness* requirements that *all* the waiting processes which are not enabled will eventually become enabled. The concurrency requirement guarantees that becoming enabled happens simultaneously, for all the waiting processes, and thus it guarantees that many processes will be able to access their fair sections at the same time as long as fairness is preserved. We notice that no lock implementation may satisfy the concurrency requirement.

The processes that have already passed through their doorway can be divided into two groups. The enabled processes and those that are not enabled. It is not possible to always have all the processes enabled due to the fairness requirement. All the enabled processes can immediately proceed to execute their fair sections. The waiting processes which are not enabled will eventually simultaneously become enabled, before or once the currently enabled processes exit their fair and exit sections. We observe that the stronger FIFO fairness requirement, the progress requirement and concurrency requirement cannot be mutually satisfied.

3.3 The fair synchronization algorithm

We use one atomic bit, called *group*. The first thing that process i does in its entry section is to read the value of the *group* bit, and to determine to which of the two groups (0 or 1) it should belong. This is done by setting i's single-writer register $state_i$ to the value read.

Once i chooses a group, it waits until its group has priority over the other group and then it enters its fair section. The order in which processes can enter their fair sections is defined as follows: If two processes belong to different groups, the process whose group, as recorded in its *state* register, is *different* from the value of the bit *group* is enabled and can enter its fair section, and the other process has to wait. If all the active processes belong to the same group then they can all enter their fair sections.

Next, we explain when the shared *group* bit is updated. The first thing that process i does when it leaves its fair section (i.e., its first step in its exit section) is to set the *group* bit to a value which is *different* from the value of its $state_i$ register. This way, i gives priority to waiting processes which belong to the same group that it belongs to.

Until the value of the *group* bit is first changed, all the active processes belong to the same group, say group 0. The first process to finish its fair section flips the value of the *group* bit and sets it to 1. Thereafter, the value read by all the new beginning processes is 1, until the group bit is modified again. Next, *all* the processes which belong to group 0 enter and then exit their fair sections possibly at the same time until there are no active processes which belong to group 0. Then

all the processes from group 1 become enabled and are allowed to enter their fair sections, and when each one of them exits it sets to 0 the value of the *group* bit, which gives priority to the processes in group 1, and so on.

The following registers are used: (1) a single multi-writer atomic bit named *group*, (2) an array of single-writer atomic registers *state*[1..*n*] which range over $\{0, 1, 2, 3\}$. To improve readability, we use below subscripts to index entries in an array. At any given time, process i can be in one of four possible states, as recorded in it single-writer register $state_i$. When $state_i = 3$, process i is not active, that is, it is in its remainder section. When $state_i = 2$, process i is active and (by reading *group*) tries to decide to which of the two groups, 0 or 1, it should belong. When $state_i = 1$, process i is active and belongs to group 1. When $state_i = 0$, process i is active and belongs to group 0.

The statement **await** *condition* is used as an abbreviation for **while** ¬*condition* **do** *skip*. The *break* statement, like in C, breaks out of the smallest enclosing *for* or *while* loop. Finally, whenever two atomic registers appear in the same statement, two separate steps are required to execute this statement. The algorithm is given below.[1]

A Fair Synchronization Algorithm: `process i's code` ($1 \le i \le n$)

Shared variables:

group: atomic bit; the initial value of the group bit is immaterial.
state[1..*n*]: array of atomic registers, which range over $\{0, 1, 2, 3\}$
Initially $\forall i : 1 \le i \le n : state_i = 3$ `/* processes are inactive */`

```
1   state_i := 2                                          /* begin doorway */
2   state_i := group                          /* choose group and end doorway */
3   for j = 1 to n do                                     /* begin waiting */
4       if (state_i ≠ group) then break fi          /* process is enabled */
5       await state_j ≠ 2
6       if state_j = 1 − state_i                      /* different groups */
7       then await   (state_j ≠ 1 − state_i) ∨ (state_i ≠ group) fi
8   od                                                      /* end waiting */
9   fair section
10  group := 1 − state_i                                    /* begin exit */
11  state_i := 3                                              /* end exit */
```

In line 1, process i indicates that it has started executing its doorway code. Then,

[1] To simplify the presentation, when the code for a fair synchronization algorithm is presented, only the entry and exit codes are described, and the remainder code and the infinite loop within which these codes reside are omitted.

in *two* atomic steps, it reads the value of *group* and assigns the value read to $state_i$ (line 2).

After passing its doorway, process i waits in the *for loop* (lines 3–8), until all the processes in the group to which it belongs are simultaneously enabled and then it enters its fair section. This happens when either, ($state_i \neq group$), i.e. the value the *group* bit points to the group which i does *not* belong to (line 4), or when all the waiting processes (including i) belong to the same group (line 7). Each one of the terms of the await statement (line 7) is evaluated separately. In case processes i and j belong to different groups (line 6), i waits until either (1) j is not competing any more or j has reentered its entry section, or (2) i has priority over j because $state_i$ is *different* than the value of the *group* bit.

In the exit code, i sets the *group* bit to a value which is different than the group to which it belongs (line 10), and changes its state to not active (line 11). We notice that the algorithm is also correct when we replace the order of lines 9 and 10, allowing process i to write the group bit immediately before it enters its fair section. The order of lines 10 and 11 is crucial for correctness.

We observe that a *non* beginning process, say p, may enter its fair section ahead of another waiting process, say q, twice: the first time if p is enabled on the other group, and the second time if p just happened to pass q which is waiting on the same group and enters its fair section first. We point out that omitting lines 1 and 5 will result in an incorrect solution. It is possible to replace each one of the 4-valued single-writer atomic registers, by three *separate* atomic bits.

An *adaptive* algorithm is an algorithm which its time complexity is a function of the actual number of participating processes rather than a function of the total number of processes. An adaptive fair synchronization algorithm using atomic register is presented in [29]. It is also shown in [29] that $n - 1$ read/write registers and conditional objects are necessary for solving the fair synchronization problem for n processes. A conditional operation is an operation that changes the value of an object only if the object has a particular value. A *conditional object* is an object that supports only conditional operations. Compare-and-swap and test-and-set are examples of conditional objects.

3.4 Fair data structures and fair locks

In order to impose fairness on a concurrent data structure, concurrent accesses to a data structure can be synchronized using a fair synchronization algorithm: a process accesses the data structure only inside a fair section. Any data structure can be easily made fair using such an approach, without using locks and with limited waiting. The formal definition of a fair data structure can be found in [29].

We name a solution to the fair synchronization problem a (finger) *ring*.[2] Using a single *ring* to enforce fairness on a concurrent data structure, is an example of coarse-grained *fair* synchronization. In contrast, fine-grained *fair* synchronization enables to protect “small pieces" of a data structure, allowing several processes with *different* operations to access it completely independently. For example, in the case of adding fairness to an existing wait-free queue, it makes sense to use two rings: one for the enqueue operations and the other for the dequeue operations.

We assume the reader is familiar with the definition of a deadlock-free mutual exclusion algorithm (DF-ME). By composing a fair synchronization algorithm (FS) and a DF-ME, it is possible to construct a *fair* mutual exclusion algorithm (FME), i.e., a fair lock. The entry section of the composed FME algorithm consists of the entry section of the FS algorithm followed by the entry section of the ME algorithm. The exit section of the FME algorithm consists of the exit section of the ME algorithm followed by the exit section of the FS algorithm. The doorway of the FME algorithm is the doorway of the FS algorithm.

4 Fault Tolerance

4.1 Motivation

According to the standard notion of fault tolerance, an algorithm is t-resilient if in the presence of up to t faults, *all* the correct processes can still complete their operations and terminate. Thus, an algorithm is *not* t-resilient, if as a result of t faults there is *some* correct process that can not terminate. This traditional notion of fault tolerance is not sensitive to the *number* of correct processes that may or may not complete their operations as a result of the failure of other processes.

Consider for example the renaming problem, which allows processes, with distinct initial names from a large name space, to get distinct new names from a small name space. A renaming algorithm that, in the presence of any number of faults, always guarantees that *most* of the correct processes, but not necessarily all, get distinct new names is clearly more resilient than a renaming algorithm that in the presence of a single fault does not guarantee that even one correct process ever gets a new name. However, using the standard notion of fault tolerance, it is not possible to compare the resiliency of such algorithms – as both are simply not even 1-resilient. This motivates us to suggest and investigate a more general notion of fault tolerance.

We generalize the traditional notion of fault tolerance by allowing a limited number participating correct processes not to terminate in the presence of faults.

[2] Many processes can simultaneously pass through the ring’s hole, but the size of the ring may limit their number.

Every process that do terminate is required to return a correct result. Thus, our definition guarantees safety but may sacrifice liveness (termination), for a limited number of processes, in the presence of faults. The consequences of violating liveness are often less severe than those of violating safety. In fact, there are systems that can detect and abort processes that run for too long. Sacrificing liveness of few processes allows us to increase the resiliency of the whole system.

4.2 A general definition of fault tolerance

For the rest of the section, n denotes the number of processes, t denotes the number of faulty processes, and $N = \{0, 1, ..., n\}$.

> **Definition:** For a given function $f : N \to N$, an algorithm is (t, f)-*resilient* if in the presence of t' faults at most $f(t')$ participating correct processes may *not* terminate their operations, for every $0 \leq t' \leq t$.

It seems that (t, f)-*resiliency* is interesting only when requiring that $f(0) = 0$. That is, in the absence of faults all the participating processes must terminate their operations. The standard definition of t-resiliency is equivalent to (t, f)-resiliency where $f(t') = 0$ for every $0 \leq t' \leq t$. Thus, the familiar notion of *wait-freedom* is equivalent to $(n-1, f)$-resiliency where $f(t') = 0$ for every $0 \leq t' \leq n-1$. The new notion of (t, f)-resiliency is quite general, and in this section we focus mainly on the following three levels of resiliency.

- An algorithm is *almost-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = 1$, for every $1 \leq t' \leq t$. Thus, in the presence of any number of up to t faults, all the correct participating processes, except maybe one process, must terminate their operations.

- An algorithm is *partially-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$ and $f(t') = t'$, for every $1 \leq t' \leq t$. Thus, in the presence of any number $t' \leq t$ faults, all the correct participating processes, except maybe t' of them must terminate their operations.

- An algorithm is *weakly-t-resilient* if it is (t, f)-*resilient*, for a function f where $f(0) = 0$, and in the presence of any number of up to $t \geq 1$ faults, if there are *two* or more correct participating processes then one correct participating process must terminate its operation. (Notice that for $n = 2$, if one process fails the other one is not required to terminate.)

For $n \geq 3$ and $t < n/2$, the notion of weakly-t-resiliency is strictly weaker than the notion of partially-t-resiliency. For $n \geq 3$, the notion of weakly-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For $n \geq 3$ and $t \geq 2$,

the notion of partially-t-resiliency is strictly weaker than the notion of almost-t-resiliency. For all n, partially-1-resiliency and almost-1-resiliency are equivalent. For $n = 2$, these three notions are equivalent. We say that an algorithm is *almost-wait-free* if it is *almost*-$(n-1)$-*resilient*, i.e., in the presence of any number of faults, all the participating correct processes, except maybe one process, must terminate. We say that an algorithm is *partially-wait-free* if it is *partially*-$(n-1)$-*resilient*, i.e., in the presence of any number of $t \leq n-1$ faults, all the correct participating processes, except maybe t of them must terminate.

4.3 Example: An almost-wait-free symmetric test-and-set bit

A test-and-set bit supports two atomic operations, called *test-and-set* and *reset*. A test-and-set operation takes as argument a shared bit b, assigns the value 1 to b, and returns the previous value of b (which can be either 0 or 1). A reset operation takes as argument a shared bit b and writes the value 0 into b.

The *sequential specification* of an object specifies how the object behaves in sequential runs, that is, in runs when its operations are applied sequentially. The sequential specification of a test-and-set bit is quite simple. In sequential runs, the first test-and-set operation returns 0, a test-and-set operation that happens immediately after a reset operation also returns 0, and all other test-and-set operations return 1. The consistency requirement is linearizability.

The algorithm below is for n processes each with a unique identifier taken from some (possibly infinite) set which does not include 0. It makes use of exactly n registers which are long enough to store a process identifier and one atomic bit. The algorithm is based on the symmetric mutual exclusion algorithm from [24].[3]

The algorithm uses a register, called *turn*, to indicate who has priority to return 1, $n-1$ *lock* registers to ensure that at most one process will return 1 between resets, and a bit, called *winner*, to indicate whether some process already returned 1. Initially the values of all these shared registers are 0. In addition, each process has a private boolean variable called *locked*. We denote by *b.turn*, *b.winner* and *b.lock*[*] the shared registers for the implementation of a specific test-and-set bit, named b.

[3] A symmetric algorithm is an algorithm in which the only way for distinguishing processes is by comparing identifiers, which are unique. Identifiers can be written, read and compared, but there is no way of looking inside any identifier. Thus, identifiers cannot be used to index shared registers.

AN ALMOST-WAIT-FREE SYMMETRIC TEST-AND-SET BIT: `process p's program.`

```
function test-and-set (b:bit) return:value in {0, 1};                    /* access bit b */
1    if b.turn ≠ 0 then return(0) fi;                                          /* lost */
2    b.turn := p;
3    repeat
4            for j := 1 to n − 1 do                                        /* get locks */
5                if b.lock[j] = 0 then b.lock[j] := p fi od
6            locked := 1;
7            for j := 1 to n − 1 do                                   /* have all locks? */
8                if b.lock[j] ≠ p then locked := 0 fi od;
9    until b.turn ≠ p or locked = 1 or b.winner = 1;
10   if b.turn ≠ p or b.winner = 1 then
11           for j := 1 to n − 1 do                              /* lost, release locks */
12               if b.lock[j] = p then b.lock[j] := 0 fi od
13           return(0) fi;
14   b.winner := 1; return(1).                                             /* wins */
end_function

function reset (b:bit);                                                  /* access bit b */
1    b.winner := 0; b.turn := 0;                                       /* release locks */
2    for j := 1 to n − 1 do
3            if b.lock[j] = p then b.lock[j] := 0 fi od.
end_function
```

In the test-and-set operation, a process, say p, initially checks whether $b.turn \neq 0$, and if so returns 0. Otherwise, p takes priority by setting $b.turn$ to p, and attempts to obtain all the $n-1$ locks by setting them to p. This prevents other processes that also saw $b.turn = 0$ and set $b.turn$ to their ids from entering. That is, if p obtains all the locks before the other processes set $b.turn$, they will not be able to get any of the locks since the values of the locks are not 0. Otherwise, if p sees $b.turn \neq p$ or $b.winner = 1$, it will release the locks it holds, allowing some other process to proceed, and will return 0. In the reset operation, p sets $b.turn$ to 0, so the other processes can proceed, and releases all the locks it currently holds. In [28], it is shown that even in the absence of faults, any implementation of a test-and-set bit for n processes from atomic read/write registers must use at least n such registers.

4.4 Additional results

The following additional results are presented in [28].

Election. It is known that there is no 1-resilient election algorithm using atomic registers [18, 31]. It is shown that:

> *There is an almost-wait-free symmetric election algorithm using* $\lceil \log n \rceil +$ 2 *atomic registers.*

The known space lower bound for election in the absence of faults is $\lceil \log n \rceil + 1$ atomic registers [24].

Perfect Renaming. A *perfect* renaming algorithm allows n processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ...n\}$. A *one-shot* renaming algorithm allows each process to acquire a distinct new name just once. A *long-lived* renaming algorithm allows processes to repeatedly acquire distinct names and release them. It is shown that:

> (1) *There is a partially-wait-free symmetric one-shot perfect renaming algorithm using* (*a*) $n - 1$ *almost-wait-free election objects, or* (*b*) $O(n \log n)$ *registers.* (2) *There is a partially-wait-free symmetric long-lived perfect renaming algorithm using either* $n - 1$ *almost-wait-free test-and-set bits.*

It is known that in asynchronous systems where processes communicate by atomic registers there is no 1-resilient perfect renaming algorithm [18, 31].

Fetch-and-add, swap, stack. A *fetch-and-add* object supports an operation which takes as arguments a shared register r, and a value *val*. The value of r is incremented by *val*, and the old value of r is returned. A *swap* object supports an operation which takes as arguments a shared registers and a local register and atomically exchange their values. A shared stack is a linearizable object that supports push and pop operations, by several processes, with the usual stack semantics. It is shown that:

> *There are partially-wait-free implementations of a fetch-and-add object, a swap object, and a stack object using atomic registers.*

The result complements the results that in asynchronous systems where processes communicate using registers there are no 2-resilient implementations of fetch-and-add, swap, and stack objects [8].

Consensus and Set-consensus. The *k-set consensus* problem is to find a solution for n processes, where each process starts with an input value from some domain, and must choose some participating process' input as its output. All n processes together may choose no more than k distinct output values. The 1-set consensus problem, is the familiar consensus problem. It is shown that:

(1) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using either atomic registers or sending and receiving messages. In particular, for $n \geq 3$, there is no weakly-1-resilient consensus algorithm using either atomic registers or messages.* (2) *For $n \geq 3$ and $1 \leq k \leq n-2$, there is no weakly-k-resilient k-set-consensus algorithm using almost-wait-free test-and-set bits and atomic registers.*

These results strengthen the know results that, in asynchronous systems where processes communicate either by atomic registers or by sending and receiving messages, there is no 1-resilient consensus algorithm [7, 15], and there is no k-resilient k-set-consensus algorithm [3, 11, 22].

5 Related Work

All the ideas and results presented in this survey are from [27, 28, 29]. Mutual exclusion locks were first introduced by Edsger W. Dijkstra in [5]. Since than, numerous implementations of locks have been proposed [20, 25]. The fair synchronization algorithm, presented in Section 3.3, uses some ideas from the mutual exclusion algorithm presented in [26].

Algorithms for several concurrent data structures based on locking have been proposed since at least the 1970's [2]. Speculative lock elision [21], is a hardware technique which allows multiple processes to concurrently execute critical sections protected by the same lock; when misspeculation, due to data conflicts, is detected rollback is used for recovery, and the execution fall back to acquiring the lock and executing non-speculatively.

The benefits of avoiding locking has already been considered in [6]. There are many implementations of data structures which avoid locking [12, 19, 25]. Several progress conditions have been proposed for data structures which avoid locking. The most extensively studied conditions, in order of decreasing strength, are wait-freedom [8], non-blocking [13], and obstruction-freedom [9]. Progress conditions, called k-waiting, for $k \geq 0$, which capture the "amount of waiting" of processes in asynchronous concurrent algorithm, are introduced in [30].

Extensions of the notion of fault tolerance, which are different from those considered in Section 4, were proposed in[4], where a precise way is presented to characterize adversaries by introducing the notion of disagreement power: the biggest integer k for which the adversary can prevent processes from agreeing on k values when using registers only; and it is shown how to compute the disagreement power of an adversary.

Linearizability is defined in [13]. A tutorial on memory consistency models can be found in [1]. Transactional memory is a methodology which has gained

momentum in recent years as a simple way for writing concurrent programs [10, 12, 23]. It has implementations that use locks and others that avoid locking, but in both cases the complexity is hidden from the programmer.

6 Discussion

None of the known synchronization techniques is optimal in all cases. Despite the known weaknesses of locking and the many attempts to replace it, locking still predominates. There might still be hope for a "silver bullet", but until then, it would be constructive to also consider integration of different techniques in order to gain the benefit of their combined strengths. Such integration may involve using a *mixture* of objects which avoid locking together with lock-based objects; and, as suggested in Section 2, *fusing* lockless objects and locks together in order to create new interesting types of shared objects.

In Section 3, we have proposed to enforce fairness as a wrapper around a concurrent data structure, and studied the consequences. We have formalized the fair synchronization problem, presented a solution, and then showed that existing concurrent data structures and mutual exclusion algorithms can be encapsulated into a fair synchronization construct to yield algorithms that are inherently fair. Since many processes may enter their fair sections simultaneously, it is expected that using fair synchronization algorithms will not degrade the performance of concurrent applications as much as locks. However, as in the case of using locks, slow or stopped processes may prevent other processes from ever accessing their fair sections.

Finally, in Section 4, we have refined the traditional notion of *t-resiliency* by defining the finer grained notion of (t, f)-*resiliency*. Rather surprisingly, while some problems, that have no solutions which can tolerate even a single fault, do have solutions which satisfy almost-wait-freedom, other problems do not even have weakly-1-resilient solutions.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[3] E. Borowsky and E. Gafni. Generalizecl FLP impossibility result for *t*-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 91–100, 1993.

[4] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmanns. The disagreement power of an adversary. In *Proc. 28th ACM Symp. on Principles of Distributed Computing*, pages 288–289, 2009.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[6] W. B. Easton. Process synchronization without long-term interlock. In *Proc. of the 3rd ACM symp. on Operating systems principles*, pages 95–100, 1971.

[7] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[8] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[9] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, page 522, 2003.

[10] M. P. Herlihy and J.E.B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[11] M. P. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, July 1999.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008. 508 pages.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *toplas*, 12(3):463–492, 1990.

[14] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[15] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[16] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[17] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Information and Computation* 233 (2013) 12–31. (Also in: LNCS 1914 Springer Verlag 2000, 164–178, DISC 2000.)

[18] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

[19] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer. ISBN 978-3-642-32027-9, 515 pages, 2013.

[20] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[21] R. Rajwar and J. R. Goodman, Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. 34th Inter. Symp. on Microarchitecture*, pp. 294–305, 2001.

[22] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29, 2000.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, 1995.

[24] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

[25] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall. ISBN 0-131-97259-6, 423 pages, 2006.

[26] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.

[27] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd international symposium on distributed computing*, September 2009. *LNCS 5805* Springer Verlag 2009, 157–171.

[28] G. Taubenfeld. A closer look at fault tolerance. In *Proc. 31st ACM Symp. on Principles of Distributed Computing*, pages 261–270, 2012.

[29] G. Taubenfeld. Fair synchronization. In *27th international symposium on distributed computing*, October 2013. *LNCS 8205* Springer Verlag 2013, 179–193.

[30] G. Taubenfeld. Waiting without locking. Unpublished manuscript, 2014.

[31] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.

Local Coordination and Symmetry Breaking

Jukka Suomela
Helsinki Institute for Information Technology HIIT,
Department of Information and Computer Science,
Aalto University, Finland · jukka.suomela@aalto.fi

Abstract

This article gives a short survey of recent *lower bounds* for *distributed graph algorithms*. There are many classical graph problems (e.g., maximal matching) that can be solved in $O(\Delta + \log^* n)$ or $O(\Delta)$ communication rounds, where n is the number of nodes and Δ is the maximum degree of the graph. In these algorithms, the key bottleneck seems to be a form of *local coordination*, which gives rise to the linear-in-Δ term in the running time. Previously it has not been known if this linear dependence is necessary, but now we can prove that there are graph problems that can be solved in time $O(\Delta)$ independently of n, and cannot be solved in time $o(\Delta)$ independently of n. We will give an informal overview of the techniques that can be used to prove such lower bounds, and we will also propose a roadmap for future research, with the aim of resolving some of the major open questions of the field.

1 Introduction

The research area of distributed computing studies computation in large computer networks. The fundamental research question is *what can be computed efficiently in a distributed system*. In distributed systems, communication is several orders of magnitude slower than computation: while a modern computer can perform arithmetic operations in a matter of *nanoseconds*, it can easily take dozens of *milliseconds* to exchange messages between two computers over the public Internet. To understand which computational tasks can be solved efficiently in a computer network, it is necessary to understand what can be solved with *very few communication steps*.

1.1 Model of Computing

Perhaps the most successful theoretical model for studying such questions is the LOCAL model [25, 32]: We have an unknown graph G that represents a computer network. Every node of G is a computer and every edge of G is a communication link. Initially, each computer is only aware of its immediate surroundings. Computation proceeds in synchronous rounds; in each round, all nodes exchange messages with their neighbours. Eventually, all nodes have to stop and announce their local outputs—that is, their own part of the solution. For example, if we are studying graph colouring, each node has to output its own colour.

The *distributed time complexity* of a graph problem is the smallest t such that the problem can be solved with a distributed algorithm in t communication rounds. In the LOCAL model, parameter t plays a dual role—it represents both *time* and *distance*: in t rounds, all nodes can learn everything about graph G in their radius-t neighbourhood, and nothing else. In essence, a distributed algorithm with a running time of t is a mapping from radius-t neighbourhoods to local outputs—see Figure 1 for an illustration.

Therefore the defining property of fast distributed algorithms is *locality*: in a fast distributed algorithm each node only needs information from its local neighbourhood. In many cases, if we are given a graph problem, it is fairly easy to qualitatively classify it as a "local" or "global" problem; the key challenge is to understand precisely *how* local a given problem is.

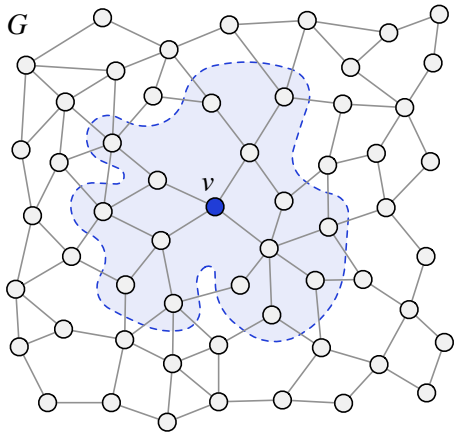


Figure 1: If we have a distributed algorithm with a running time of $t = 2$ rounds, then whatever node v outputs is a function of the information available in its radius-t neighbourhood (highlighted).

1.2 Symmetry Breaking vs. Local Coordination

Distributed time complexity and locality are commonly studied as a function of two parameters:

- n, the number of nodes in the network,
- Δ, the maximum degree of a node (the maximum number of neighbours).

As we will see, these parameters are often related to two different challenges in the area of distributed algorithms:

1. complexity as a function of n is related to *symmetry breaking*,
2. complexity as a function of Δ is related to *local coordination*.

The first aspect has been relatively well understood since Linial's seminal work in the 1990s [25]. However, our understanding of the second aspect has been poor—until very recently.

In this article, we will study the challenge of local coordination in more depth. We will survey recent work that has enabled us to better understand the distributed time complexity of certain graph problems not only as a function of n, but also as a function of Δ. Hopefully these preliminary results will provide a starting point for a research program that aims at resolving the long-standing open questions related to the distributed time complexity of classical graph problems such as maximal matchings, maximal independent sets, and graph colouring.

2 The LOCAL Model of Distributed Computing

To get started, let us first define the LOCAL model of distributed computing a bit more carefully. A reader familiar with the model can safely skip this section; further information can be found in many textbooks [27, 32, 34].

We will study distributed algorithms in the context of graph problems. We are given an unknown graph G, and the algorithm has to produce a feasible solution of the graph problem. Each node of graph G is a computational entity, and all nodes run the same distributed algorithm A. Eventually, each node v has to stop and produce its own part of the solution—the *local output* $A(G, v)$.

For example, if our goal is to find a proper vertex colouring of G, then the local output $A(G, v)$ will be the colour of node v in the solution. If our goal is to find a maximal matching, then the local output $A(G, v)$ will indicate whether v is matched and with which neighbour.

2.1 Synchronous Message Passing

In the LOCAL model, each node has a unique identifier, and initially each node knows only its own unique identifier and its degree in graph G. Computation proceeds in *synchronous communication rounds*. In every round, each node v that is still running performs the following steps, synchronously with all other nodes:

1. v sends a message to each of its neighbours,
2. v receives a message from each of its neighbours,
3. v updates its local state, and
4. possibly v stops and announces its local output $A(G, v)$.

The outgoing messages are a function of the old local state, and the new local state is a function of the old state and the messages that the node received.

The *running time* of an algorithm is defined to be the number of communication rounds until all nodes have stopped and announced their local outputs. This is the key difference between centralised and distributed computing: in the context of centralised algorithms we are interested in the number of *elementary computational steps*, while in the context of distributed algorithms the key resource is the number of *communication steps*. For our purposes, the cost of local computation is negligible.

2.2 Time and Distances Are Equivalent

In the LOCAL model, information can propagate at a maximum speed of 1 edge per round. If a node v stops after t rounds, then whatever v outputs can only depend on the information that was available within distance t from v in the input graph. In essence, a time-t algorithm in the LOCAL model is just a *mapping from radius-t neighbourhoods to local outputs*; recall Figure 1.

A bit more formally, let us write $x_t(v)$ for the local state of a node v after round t. Initially, $x_0(v)$ consists of just the unique identifier of node v and its degree. The key observation is that $x_{t+1}(v)$ is a function of $x_t(v)$ and the messages that v received from its neighbours during round $t + 1$. For each neighbour u of v, the message sent by u to v during round $t + 1$ is a function of $x_t(u)$. Hence the new state $x_{t+1}(v)$ is simply a function of the old states $x_t(v)$ and $x_t(u)$, where u ranges over the neighbours of v.

In essence, in each communication round, each node just updates its local state based on the local states in its radius-1 neighbourhood. By a simple induction, the local state $x_t(v)$ is a function of the initial states $x_0(u)$, where u ranges over all nodes that are within distance t from v. Hence we can see that if a node stops after round t, its local output can only depend on the information that was available within distance t from v.

Conversely, it is easy to design an algorithm in which all nodes can gather their radius-t neighbourhoods in t communication rounds. As a corollary, if graph G is connected and has a diameter of D, then in time $t = D + O(1)$ all nodes can learn everything about the structure of the input graph G. Therefore in time $t = D + O(1)$ we can also solve any computable graph problem: each node can simply gather full information on the input, and then locally solve the problem and output the relevant part of the solution. The diameter is at most $O(n)$, and therefore linear-time algorithms are entirely trivial in the LOCAL model; the research on the LOCAL model focuses on algorithms that run in sublinear time.

3 Example: Maximal Matchings

As a concrete example, let us consider the problem of finding a *maximal matching* with a distributed algorithm in the LOCAL model. Recall that a *matching* of a simple undirected graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that each node is incident to at most one edge of M, and a matching is *maximal* if it is not a proper subset of another matching.

3.1 Simple Distributed Algorithms

It is easy to come up with a distributed algorithm that finds a maximal matching. Here is a simple example:

- Initialise $M \leftarrow \emptyset$.
- Use the unique node identifiers to assign a unique label for each edge.
- Repeat until G is empty:
  - Find the set $X \subseteq E$ that consists of the edges that are *local minima* with respect to the edge labels, i.e., their labels are smaller than the labels of any of their neighbours.
  - Add X to M, and remove all edges adjacent to X from G.

Unfortunately, the running time of such an algorithm can be linear in n, which makes it uninteresting from the perspective of the LOCAL model.

In the above algorithm, a key drawback is that set X can be very small in the worst case. We can do better with a simple *randomised* distributed algorithm [1, 20, 26]. Instead of picking local minima, we can pick a random matching X. More precisely, each edge e attempts to join X with a certain (carefully chosen) probability, and if none of its neighbours attempts to join simultaneously, we will

have $e \in X$. The basic idea is that we can eliminate a constant fraction of the edges in each step, and it can be shown that the running time will be $O(\log n)$ with high probability.

3.2 State of the Art

The fastest distributed algorithms can find maximal matchings much faster than in logarithmic time—at least in sparse graphs. To better characterise the running time, we will use two parameters: n, the number of nodes, and Δ, the maximum degree of the graph. As a function of n and Δ, currently the fastest algorithms are these:

- Barenboim et al. [7]: a randomised algorithm, time $O(\log \Delta + \log^4 \log n)$.
- Hańćkowiak et al. [18]: a deterministic algorithm, time $O(\log^4 n)$.
- Panconesi and Rizzi [31]: a deterministic algorithm, time $O(\Delta + \log^* n)$.

Here $\log^* n$ is the iterated logarithm of n, a very slowly growing function.

3.3 Focus on Low-Degree Graphs

In general, charting the landscape of distributed time complexity throughout the (n, Δ)-plane is an arduous task. In this article, we will zoom into one corner of the plane: we will focus on the case of $n \gg \Delta$.

In this region, the Panconesi–Rizzi algorithm, which runs in time $O(\Delta + \log^* n)$, is the fastest known algorithm for finding maximal matchings. While the expression of the running time may not look particularly natural, it turns out there are many other problems for which the fastest algorithms in sparse graphs have a running time of precisely $O(\Delta + \log^* n)$—the key examples are maximal independent sets, vertex colouring, and edge colouring [5, 6, 21, 31].

While many different algorithms with precisely the same running time exist, we do not know if any of these are optimal. In particular, it is not known if any of these problems could be solved in time $o(\Delta) + O(\log^* n)$.

In what follows, we will dissect the running time of the Panconesi–Rizzi algorithm in two terms, $O(\log^* n)$ and $O(\Delta)$, and give an intuitive explanation for each of them. To do this, we will consider two corner cases: one in which we can find maximal matchings in time $O(\log^* n)$, and one in which the problem can be solved in time $O(\Delta)$. As we will see, there is a very good justification for the term $O(\log^* n)$, but there is no lower bound that would justify the existence of the term $O(\Delta)$.

3.4 Maximal Matchings as a Symmetry-Breaking Problem

First, let us have a look at the term $O(\log^* n)$. Maximal matchings are fundamentally a *symmetry-breaking problem*. To see this more clearly, we will define a restricted version in which we are left with a *pure* symmetry-breaking problem; in essence, we will eliminate the term Δ from the running time.

This turns out to be easy: we can simply focus on cycle graphs, in which case we have a constant maximum degree of $\Delta = 2$. We are now left with the seemingly trivial problem of finding a maximal matching in a cycle.

This special case highlights the need for symmetry breaking. While the topology of the input graph is symmetric, the output cannot be symmetric: any maximal matching has to contain some of the edges, but not all of them. We will have to resort to some means of symmetry breaking.

Problems of this type can be solved efficiently with the help of the Cole–Vishkin algorithm [8]. This is a deterministic distributed algorithm that can be used to e.g. colour a cycle with $O(1)$ colours in time $O(\log^* n)$. The algorithm relies heavily on the existence of unique node identifiers; it extracts symmetry-breaking information from the unique identifiers.

In essence, the Cole–Vishkin algorithm is a *colour reduction algorithm*. The unique identifiers provide a colouring with a large number of colours. Typically it is assumed that the range of unique identifiers is polynomial in n, and hence our starting point is a poly(n)-colouring of the cycle. The Cole–Vishkin algorithm then reduces the number of colours from any number ℓ to $O(\log \ell)$ in one step; it compares the *binary representations* of the old colours of adjacent nodes, and uses the index of the bit that differs, together with its value, to construct a new colour. Overall, we need only $O(\log^* n)$ iterations to reduce the number of colours to $O(1)$.

Once we have a colouring of the vertices with $O(1)$ colours, it is easy to find a maximal matching. For example, we can use the vertex colouring to derive an edge colouring. Then each colour class is a matching, and we can construct a maximal matching by considering the colour classes one by one. Hence the pure symmetry-breaking version of maximal matchings can be solved in time $O(\log^* n)$.

Remarkably, this is also known to be optimal. Linial's seminal lower bound [25] shows that symmetry breaking in a cycle requires $\Omega(\log^* n)$ rounds with deterministic algorithms, and Naor [28] shows that this holds even if we consider randomised algorithms.

In summary, pure symmetry breaking problems are very well understood. As a simple corollary, it is not possible to find a maximal matching in time $O(\Delta) + o(\log^* n)$, or in time $f(\Delta) + o(\log^* n)$ for any function f.

3.5 Maximal Matchings as a Coordination Problem

Let us now turn our attention to the term $O(\Delta)$. We will argue that maximal matchings are also a *coordination problem*. To see this more clearly, we will again define a restricted version in which we are left with a *pure* coordination problem, which can be solved in time $O(\Delta)$ independently of n.

Of course we cannot simply set $n = O(1)$ to get rid of the term $O(\log^* n)$. However, we can consider a situation in which we have already solved the symmetry-breaking problem. We will *assume* that we are given a *bipartite graph*, in which one part is coloured black, another part is white. Our goal is to find a maximal matching in such a graph.

In essence, each black node should try to pick one of its white neighbours as its partner, and conversely each white node should try to pick one of its black neighbours as its partner. Here we have the coordination challenge: without any coordination, several black nodes may try to pick the same white node simultaneously.

In bipartite graphs, this coordination problem can be solved in time $O(\Delta)$ with a simple proposal algorithm [17]. The algorithm repeatedly performs the following steps:

1. Black nodes send "proposals" to their white neighbours, one by one (e.g. in the order of their unique identifiers).

2. White nodes "accept" the first proposal that they get (breaking ties with e.g. the unique identifiers of the senders), and "reject" all other proposals.

Each proposal–acceptance pair forms an edge in the matching. A black node will stop as soon as it becomes matched, or it runs out of white neighbours to whom to send proposals (in which case all of the neighbours are already matched). A white node will stop as soon as it becomes matched, or all of its black neighbours have stopped (in which case all of the neighbours are already matched). Clearly the output is a maximal matching.

While the simple proposal algorithm runs in time $O(\Delta)$ independently of n, it is not known if it is optimal. More specifically, it is not known if we can find a maximal matching in bipartite 2-coloured graphs in time $o(\Delta)$ independently of n. However, this is the best algorithm that we currently have for this problem; we have not been able to break the linear-in-Δ boundary (without introducing some dependence on n in the running time). Many other distributed algorithms have a similar issue at their core: in one way or another, there is a need for local coordination, and this is resolved in a fairly naive manner by considering neighbours one by one.

While symmetry breaking is well understood, local coordination is poorly understood. There are hardly any lower bounds. For bipartite maximal matchings

we can apply the lower bound by Kuhn et al. [22–24], which shows that bipartite maximal matching requires $\Omega(\log \Delta)$ rounds in the worst case. However, this still leaves an *exponential* gap between the upper bound and the lower bound.

A particularly intriguing special case is that of *regular graphs*, i.e., the case that all nodes have the same degree of Δ. In this case, the simple proposal algorithm is still the fastest algorithm that we have, and hence the best known upper bound is still $O(\Delta)$. Somewhat surprisingly, there are no lower bounds at all for this case; the above-mentioned lower bound by Kuhn et al. cannot be applied here any more. Hence we have very simple coordination problems whose distributed time complexity is not understood at all.

4 Towards Coordination Lower Bounds

We have seen that in low-degree graphs, the fastest algorithm for maximal matchings has a running time of $O(\Delta + \log^* n)$. Here the term $O(\log^* n)$ is related to the well-known challenge of symmetry breaking, while the term $O(\Delta)$ appears to be related to a poorly-understood challenge of local coordination. The problem cannot be solved in time $O(\Delta) + o(\log^* n)$, but it is a major open question if it can be solved in time $o(\Delta) + O(\log^* n)$. Let us now have a look at possible ways towards answering this question.

4.1 Completable but Tight Problems

As we have discussed, maximal matchings are not an isolated example. There are numerous other graph problems for which the time complexity as a function of n is well-understood (at least for a small Δ), but the time complexity as a function of Δ is not really understood at all.

Perhaps the most prominent example is proper vertex colouring with $\Delta + 1$ colours [5, 6, 21]. This is yet another problem that can be solved in $O(\Delta + \log^* n)$ time, and it is known to require $\Omega(\log^* n)$ time, but it is not known if the problem can be solved in $o(\Delta) + O(\log^* n)$ time.

Other examples of such problems include edge colouring with $2\Delta - 1$ colours, and the problem of finding a maximal independent set. Incidentally, all of these problems can be characterised informally as follows:

1. Any partial solution can be *completed*. In particular, a greedy algorithm that considers nodes or edges in an arbitrary order can solve these problems.

2. However, there are situations in which a partial solution is *tight* in the sense that there is only one way to complete it. For example, if we have already

coloured all Δ neighbours of a node, and we have a colour palette of size $\Delta + 1$, we may have only 1 possible colour left.

While some counterexamples exist, it seems that many problems of this form have distributed algorithms with a running time of, e.g., $O(\Delta + \log^* n)$ or simply $O(\Delta)$, and we do not know if these algorithms are optimal. However, as soon as we step outside the realm of "completable but tight" problems, we see plenty of examples of running times that are either sublinear or superlinear in Δ:

1. If we drop the first restriction, running times typically increase to *superlinear* in Δ. Examples of such problems include matchings without short augmenting paths [2], which can be solved in time $\Delta^{O(1)}$, and locally optimal semi-matchings [9], which can be solved in time $O(\Delta^5)$.

2. If we drop the second restriction, there are many problems that can be solved in time *sublinear* in Δ. A good example is vertex colouring with $O(\Delta^2)$ colours. With such a large colour palette, local coordination becomes much easier: Linial's algorithm [25] solves this problem in time $O(\log^* n)$, independently of Δ.

Hence to resolve the long-standing open questions related to the distributed time complexity, it seems that we need to better understand the issue of local coordination in the context of "completable but tight" problems.

4.2 Major Hurdles and Recent Advances

Looking back at the previous attempts to prove e.g. tight lower bounds for the distributed time complexity of maximal matchings, it now seems that one of the major hurdles can be summarised as follows:

1. We do not understand the issue of local coordination even in isolation.

2. To prove tight lower bounds, we would need to understand not just local coordination in isolation, but also the complicated interplay of local coordination and symmetry breaking.

While the second issue seems to be still far beyond the reach of current research, we are now finally making progress with the first issue. The key idea is to identify *pure coordination problems*. These are "completable but tight" problems that can be solved in time $O(\Delta)$ independently of n, but cannot be solved in time $o(\Delta)$ independently of n. Such problems do not need any symmetry breaking, and hence we can focus solely on local coordination.

Now we finally have the first natural example of a pure coordination problem for which we can prove tight lower bounds: *maximal fractional matching*. This

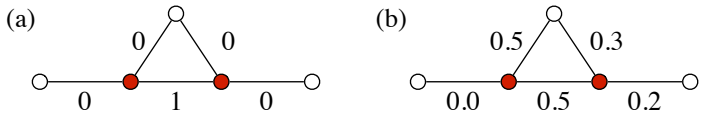


Figure 2: (a) A maximal matching. (b) A maximal fractional matching. The saturated nodes are highlighted.

was known to be solvable in time $O(\Delta)$ independently of n [3], and there is now a lower bound that shows that the problem cannot be solved in time $o(\Delta)$ independently of n [14]. In the next section, we will discuss this problem in more depth.

5 Recent Progress: Maximal Fractional Matchings

A *fractional matching* is a linear programming relaxation of a matching. While maximal matchings are a symmetry-breaking problem, it turns out that the problem of finding a *maximal fractional matching* is a pure coordination problem—it does not require any symmetry breaking.

5.1 Problem Formulation

If a matching can be interpreted as an integer-valued function $y\colon E \to \{0,1\}$ that indicates which edges are in the matching, a fractional matching is a real-valued function $y\colon E \to [0,1]$.

Formally, let $G = (V,E)$ be a simple undirected graph and let $y\colon E \to [0,1]$ associate weights to the edges of G. Define for each $v \in V$ the sum of incident edges

$$y[v] := \sum_{e \in E : v \in e} y(e).$$

The function y is called a *fractional matching* if $y[v] \leq 1$ for each node v. A node v is *saturated* if $y[v] = 1$. A fractional matching is *maximal* if each edge has at least one saturated endpoint.

Informally, in a maximal fractional matching we cannot increase the weight of any edge without violating a constraint. See Figure 2 for examples.

In combinatorial optimisation, maximal fractional matchings and maximal matchings have similar applications. It is well known that we can use a maximal matching to construct a 2-approximation of a minimum vertex cover. It turns out that we can use maximal fractional matchings equally well: in any maximal fractional matching, the set of saturated nodes forms a 2-approximation of a minimum vertex cover [4].

5.2 No Need for Symmetry Breaking

From the perspective of centralised algorithms, the distinction between maximal matchings and maximal fractional matchings is not particularly interesting; either of the problems can be solved easily in linear time. However, the two problems are very different from the perspective of efficient distributed or parallel algorithms.

While the problem of finding a maximal matching requires symmetry breaking (recall Section 3.4), this is not the case with maximal fractional matchings. Consider, for example, a k-regular graph. In any such graph, there is a trivial maximal fractional matching that sets $y(e) = 1/k$ for all $e \in E$. In essence, highly symmetric graphs are trivial from the perspective of maximal fractional matchings; only non-symmetric inputs require some effort.

The general case is not that easy to solve efficiently, but it turns out that there is an algorithm [3] that finds a maximal fractional matching in $O(\Delta)$ time in the LOCAL model. The running time does not have any dependence on n, but it is still linear in Δ, as the algorithm resorts to a fairly naive one-by-one approach to overcome challenges related to local coordination.

In summary, we can characterise the state-of-the-art algorithms for these two problems as follows:

1. Maximal matchings:
   - symmetry breaking necessary
   - algorithms resort to local coordination
   - time $O(\Delta + \log^* n)$.
2. Maximal fractional matchings:
   - symmetry breaking not needed
   - algorithms resort to local coordination
   - time $O(\Delta)$.

5.3 A New Lower Bound

Maximal fractional matchings are a genuine example of a pure coordination problem. They are a "completable but tight" problem, in the sense discussed in Section 4.1. They do not need any symmetry breaking. Most importantly, now we can show that the $O(\Delta)$-time algorithm is optimal (at least for sparse graphs): the problem cannot be solved in time $o(\Delta)$, independently of n, with any distributed algorithm (deterministic or randomised).

Before we continue, it is important to emphasise that the result does not yet tell anything more than what is stated above. In particular, it has not yet been

ruled out that there could exist a sublinear-in-Δ algorithm whose running time depends moderately on n. For example, algorithms of a running time $o(\Delta) + O(\log^* n)$ cannot be yet excluded. This is also the reason why this result does not tell anything interesting about maximal matchings: a simple corollary would be that maximal matchings cannot be found in time $o(\Delta)$ independently of n, but this we already know, as any algorithm for maximal matchings has a running time $\Omega(\log^* n)$ that certainly depends on n. Nevertheless, this result demonstrates that there are techniques with which we can prove tight lower bounds for pure coordination problems, and this hopefully paves the road for future research in which we can tackle also algorithms with running times that have a moderate dependence on n.

A key insight in the proof is that we will not try to prove the result directly for the LOCAL model, but we will first consider *weaker* models of distributed computing (Sections 5.4 and 5.7). In weaker models, lower bounds are of course easier to prove but less interesting. Only then we will amplify the result so that we have similar lower bounds also for the LOCAL model.

On a high level, the proof builds on the following techniques:

1. The *unfold-and-mix* technique for proving lower bounds in very weak models of distributed computing (Section 5.6).

2. A general technique for amplifying lower bounds from weak models to the usual LOCAL model (Section 5.8). The main ingredient here is the construction of so-called *homogeneous graphs* (Section 5.9).

Both of these techniques were originally presented in PODC 2012 [13, 19], but it was not until PODC 2014 [14] that we managed to put these techniques together in order to prove lower bounds for the maximal fractional matching problem in the usual LOCAL model.

5.4 Port-Numbering Model and Edge Colouring Model

We start by introducing two models of distributed computing: the *port-numbering model*, in short PN, and the *edge-colouring model*, in short EC. While at least the PN model is also interesting in its own right, for our purposes these models serve as a stepping stone towards more interesting results—lower bounds in the LOCAL model.

In many ways, the PN and EC models are similar to the usual LOCAL model: Each node is an autonomous entity that maintains its own local state. Computation proceeds in synchronous rounds. In each round, all nodes in parallel (1) send messages to their neighbours, (2) receive messages from their neighbours, (3) update their local state, and (4) possibly announce their local output and stop.

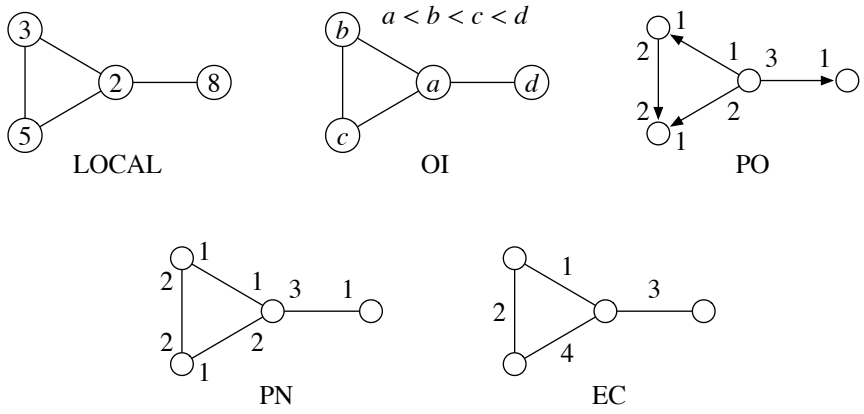
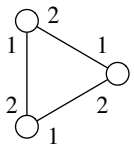


Figure 3: Models LOCAL, OI, PO, PN, and EC.

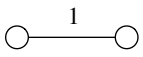
The key difference is related to the initial information that is available in the network. In PN and EC, the nodes are *anonymous*—they do not have any unique identifiers. However, each node can still distinguish between their neighbours; see Figure 3:

- In the PN model, the endpoints of the edges ("ports") are numbered so that a node of degree d is incident to endpoints with numbers $1, 2, \ldots, d$. In essence, a node can refer to its neighbours by numbers $1, 2, \ldots, d$.
- In the EC model, the edges are properly coloured with $O(\Delta)$ colours. Each node knows the colours of the incident edges, and it can use the colours to refer to its neighbours.

Note that EC is strictly stronger than PN. Given an edge colouring, it is easy to derive a port numbering. The converse is not true: for example, in the EC model it is trivial to find an edge colouring, but in the PN model it is not possible in general with any deterministic algorithm. To see this, consider a cycle with a symmetric port numbering; no PN-algorithm can break the symmetry here:



However, note that EC is also a fairly weak model. For example, no deterministic algorithm can break the symmetry in a graph with just two nodes:



In particular, it is not possible to find a proper vertex colouring with either PN or EC algorithms here. While everything was solvable in linear time in the LOCAL model, there are numerous seemingly trivial problems that cannot be solved at all in PN or EC.

5.5 Maximal Matching in the EC Model

Maximal matching cannot be solved with deterministic algorithms in the PN model. However, in the EC model, there is a very simple greedy algorithm for finding a maximal matching M in time $O(\Delta)$. We consider each colour class $1, 2, \ldots, O(\Delta)$ one by one. In step i, we find all edges of colour i that are not yet adjacent to any edge of M, and add them to M. Clearly, the end result is a maximal matching.

A key observation at this point is that maximal matchings in the EC model do not need any symmetry breaking. Symmetry between adjacent edges is already broken by the edge colouring. We only need to deal with local coordination, in order to avoid adding two adjacent edges simultaneously to M.

The greedy algorithm solves the local coordination challenge by a very naive technique: it considers colour classes one by one, which results in a linear-in-Δ running time. The key question is now if this algorithm is optimal in the EC model. Perhaps e.g. a clever divide-and-conquer approach could solve the problem in only $O(\log \Delta)$ time?

It turns out the greedy algorithm is indeed optimal. We can show that there is no algorithm that finds a maximal matching in $o(\Delta)$ time in the EC model [19]. This was the first linear-in-Δ lower bound for a natural coordination problem.

5.6 Unfold and Mix

We will now give an overview of the techniques that can be used to prove lower bounds for the maximal matching problem in the EC model. Assume that we have a deterministic distributed algorithm A that finds a maximal matching in any given graph, for any given edge colouring. With this knowledge, we can construct "fragile" instances in which algorithm A is forced to produce *perfect matchings*. Informally, perfect matchings are more tightly constrained than maximal matchings; minor changes in the input may cause major changes in the output.

To force algorithm A to produce a perfect matching, we will to study graphs with self-loops. For our purposes, a graph G with self-loops is just a compact representation of a large (possibly infinite) graph G' that does not have any loops. To construct G', we just "unfold" all loops of G; see Figure 4.

Each self-loop represents *symmetry*. If e is a self-loop in graph G, and we unfold e to construct graph G', then G' will be symmetric with respect to e. More precisely, in the EC model deterministic algorithms cannot distinguish between

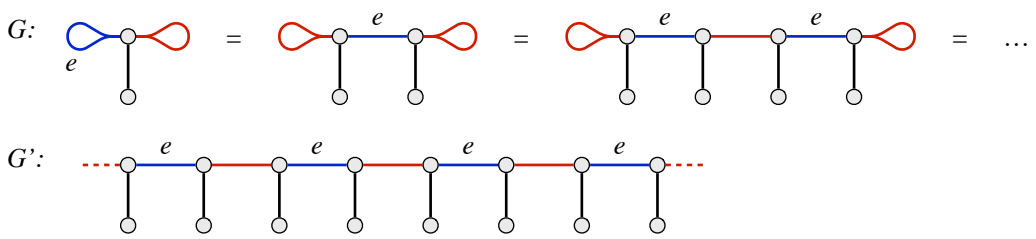


Figure 4: Graph G with self-loops is a compact representation of graph G'.

the two endpoints of e. If one of the endpoints produces the local output indicating that it is unmatched, the other endpoint will produce the same output—but this is a contradiction, as in a maximal matching we cannot have a pair of adjacent nodes such that both of them are unmatched. Therefore all nodes that have self-loops must be always matched. As long as we have at least one self-loop attached to each node, the output will be a perfect matching.

To prove the lower bound of $\Omega(\Delta)$, we will study *critical pairs*. We say that G and H form an r-critical pair of graphs if:

- G has a self-loop e and H has a self-loop f,
- loops e and f have the same colour,
- the radius-r neighbourhoods of e and f are isomorphic,
- algorithm A makes a different decision for e and f.

By a different decision we mean that in graph G algorithm A outputs a matching $A(G)$ with $e \in A(G)$, and in graph H algorithm A outputs a matching $A(H)$ with $f \notin A(H)$. Naturally, if such a pair exists, then the running time of A has to be at least r; otherwise A would not be able to distinguish between e and f.

For any algorithm A, it is fairly straightforward to find a 0-critical pair that consists of just a pair of nodes so that both of them have $\Theta(\Delta)$ self-loops. Once we have a 0-critical pair, we will apply a technique called *unfold-and-mix* repeatedly. In each iteration we will lose some self-loops but gain criticality. We can repeat the process for $\Theta(\Delta)$ times until we run out of self-loops. The end result will be a $\Theta(\Delta)$-critical pair of graphs of maximum degree Δ. The existence of such a pair is enough to demonstrate that the running time of algorithm A is $\Omega(\Delta)$.

The unfold-and-mix technique is illustrated in Figure 5. In each inductive step, our starting point is a k-critical pair of graphs, G and H, with the special loops e and f. Then we

1. "unfold" the loops e and f to obtain graphs GG and HH,
2. "mix" the graphs GG and HH together to obtain another graph GH that combines elements from G and H.

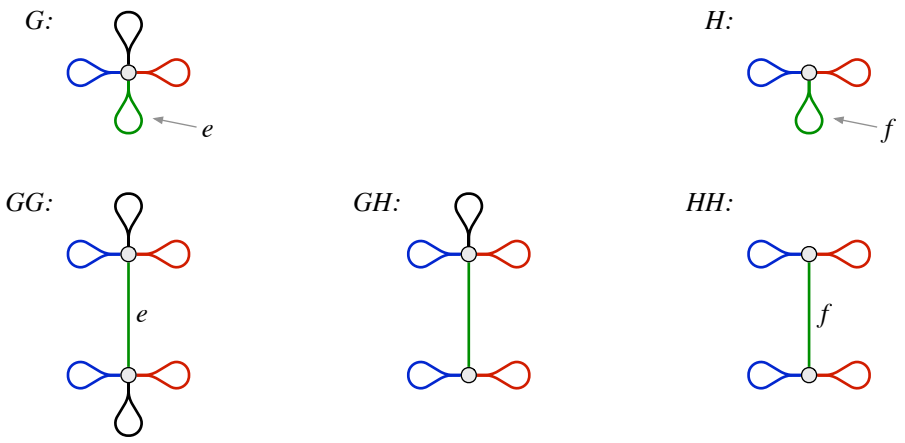


Figure 5: The unfold-and-mix technique.

Intuitively, the output of A in GG and the output of A in HH are not compatible with each other—by assumption, the outputs of e and f are different. Hence the algorithm is now in trouble:

- It has to do something different in GH in comparison with what it did in either GG or HH.

- It cannot sweep the problem under the rug easily, as the instances are "fragile": graph GH still has self-loops, and therefore the algorithm has to output a perfect matching.

With some effort, we can now show that among the three graphs that we have constructed (GG, GH, and HH), we can always find at least two that satisfy the conditions of a $(k+1)$-critical pair.

In essence, each unfold-and-mix step makes the problem instance more difficult from the perspective of the algorithm that we study: the algorithm has to look further (i.e., spend more time) in order to distinguish the two graphs that form a critical pair. This way we can show that algorithm A cannot find a maximal matching in time $o(\Delta)$. Similar ideas can be used to prove an analogous lower bound also for maximal fractional matchings.

5.7 More Models

So far we have seen how to prove tight lower bounds for coordination problems in the EC model. However, we are interested in the usual LOCAL model, and the EC model and the LOCAL model are very different from each other.

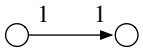
We will introduce two new models that serve as intermediate steps that bridge the gap between the EC model and the LOCAL model; see Figure 3:

- *Port numbering and orientation* (PO): The model is a stronger version of the PN model. In addition to the port numbering, we are also given an *orientation*, i.e., for each edge one of the endpoints is labelled as the head. The orientation does not restrict how we can send information in the network; it is just additional symmetry-breaking information that the algorithm can use.

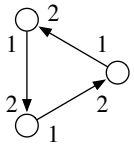
- *Order-invariant algorithms* (OI): This model is a weaker version of the LOCAL model. The nodes have unique identifiers, but algorithms can only use the relative order of the nodes and not the numerical values of the identifiers. Put otherwise, if we relabel the nodes but preserve their relative order, the output of the algorithm must not change.

For randomised algorithms these models are not that interesting, as a randomised algorithm can use random bits to e.g. generate labels that are unique with high probability (at least if we have some estimate of the size of the network). However, for deterministic algorithms the differences between the models PN, PO, OI, and LOCAL become interesting.

First, we can see that the PO model is strictly stronger than the PN model. For example, in a graph with just one edge, a PO algorithm can use the orientation to break the symmetry between the endpoints, while in PN this is not possible:



We can also see that OI is strictly stronger than PO. The OI model is clearly at least as strong as the PO model: the ordering of the nodes can be used to derive both a port numbering and an orientation. Moreover, in the OI model we can always break symmetry e.g. in a cycle (there is always a unique node that is smaller than any other node), while this is not necessarily the case in the PO model:



Finally, there are many problems that can be solved faster in the LOCAL model than in the OI model. Maximal matchings in a cycle are a good example: it takes $\Theta(n)$ time to find a maximal matching in a cycle in the OI model in the worst case, but as we have discussed in Section 3.4, this is possible in $\Theta(\log^* n)$ time in the LOCAL model. In summary, for deterministic algorithms the relative strengths of the models are PN $\subsetneq$ PO $\subsetneq$ OI $\subsetneq$ LOCAL and PN $\subsetneq$ EC.

5.8 Amplifying Lower Bounds

We have seen above that in general, the PO model can be much weaker than the LOCAL model. Indeed, there are many algorithms that exploit the properties of the LOCAL model in order to solve graph problems efficiently. For example, numerous algorithms that solve symmetry breaking in time $O(\log^* n)$ specifically rely on the unique identifiers and cannot be used in the PO model.

However, if we have a look at problems that can be solved in time $f(\Delta)$ independently of n—for example, pure coordination problems—the situation looks very different. As we can see from the survey [33], for numerous classical graph problems, the best $f(\Delta)$-time deterministic approximation algorithms in the LOCAL model do not make any use of unique identifiers. We could easily run the same algorithms in the PO model as well (and sometimes also in the PN model).

It turns out that this is not just a coincidence. We can now prove that the PO, OI, and LOCAL models are equally strong, at least in the following case [13]:

1. We use distributed algorithms with a running time of $f(\Delta)$ for some f, independently of n.

2. We are interested in so-called *simple* PO-*checkable graph optimisation problems*. This includes many classical packing and covering problems such as vertex covers, edge covers, matchings, independent sets, dominating sets, and edge dominating sets.

To prove that PO and LOCAL are equally strong for this family of problems, we proceed in two steps, using the OI model as an intermediate step:

- PO $\approx$ OI: We introduce so-called *homogeneous graphs*, in which nodes are ordered so that the ordering provides as little additional information as possible. There is only a small fraction of nodes for which OI algorithms may have an advantage over PO algorithms. See Section 5.9 for more details.

- OI $\approx$ LOCAL: We apply *Ramsey's theorem* [16] to assign unique identifiers in an unhelpful manner, so that algorithms in the LOCAL model do not have any advantage over OI algorithms.

The use of Ramsey's theorem in such a context is nowadays a standard technique [10, 29]. The key novelty is the introduction of homogeneous graphs, and in particular, a proof that shows that finite high-girth high-degree homogeneous graphs indeed exist.

5.9 Key Ingredient: Homogeneous Graphs

We introduce the following technical definition that is helpful in the context of the OI model. We say that graph G is $(1-\epsilon, r)$-homogeneous if the nodes can be ordered so that a fraction $1-\epsilon$ of all radius-r neighbourhoods are isomorphic, with respect to both topology and ordering. If we have such an ordering of the nodes, then any OI-algorithm with a running time at most r will be in trouble: almost all neighbourhoods look identical.

To show that models PO and OI are equally strong for any $f(\Delta)$-time algorithm, for a broad range of graph problems, it is desirable to have graphs with the following properties, for any g, r, k, and $\epsilon > 0$:

(1) the graph is $(1-\epsilon, r)$-homogeneous,
(2) the graph is $2k$-regular,
(3) the graph has girth at least g, and
(4) the graph is finite.

It turns out that satisfying any three out of the four properties is easy—see Figure 6 for examples:

(a) Regular high-girth graphs satisfy all properties except (1).
(b) Cycles satisfy all properties except (2).
(c) Regular grids satisfy all properties except (3).
(d) Infinite trees satisfy all properties except (4).

However, satisfying all four properties simultaneously is more challenging. The construction that we use in our recent work [15] is based on the Cayley graphs of certain groups (variants of so-called iterated wreath products) that have several desirable properties: moderate growth, relatively large girth [12], and a convenient geometric embedding.

5.10 Putting Everything Together

With the help of homogeneous graphs we have been able to show that the models PO, OI, and LOCAL are equally strong from the perspective of $f(\Delta)$-time algorithms. We also know that maximal matchings take $\Omega(\Delta)$ time in the EC model. Ideally, we would now like to put the two results together and show that maximal matchings cannot be solved in $o(\Delta) + O(\log^* n)$ time in the LOCAL model, either.

Unfortunately, we do not know how to do this yet. Even if we put aside the issue of somehow bridging the gap between the EC model and the PO model, we have a much bigger obstacle in front of us: the term $O(\log^* n)$ in the running time is enough to separate the models PO and LOCAL, and kill the argument of Section 5.8. In essence, we still cannot deal with symmetry-breaking problems.

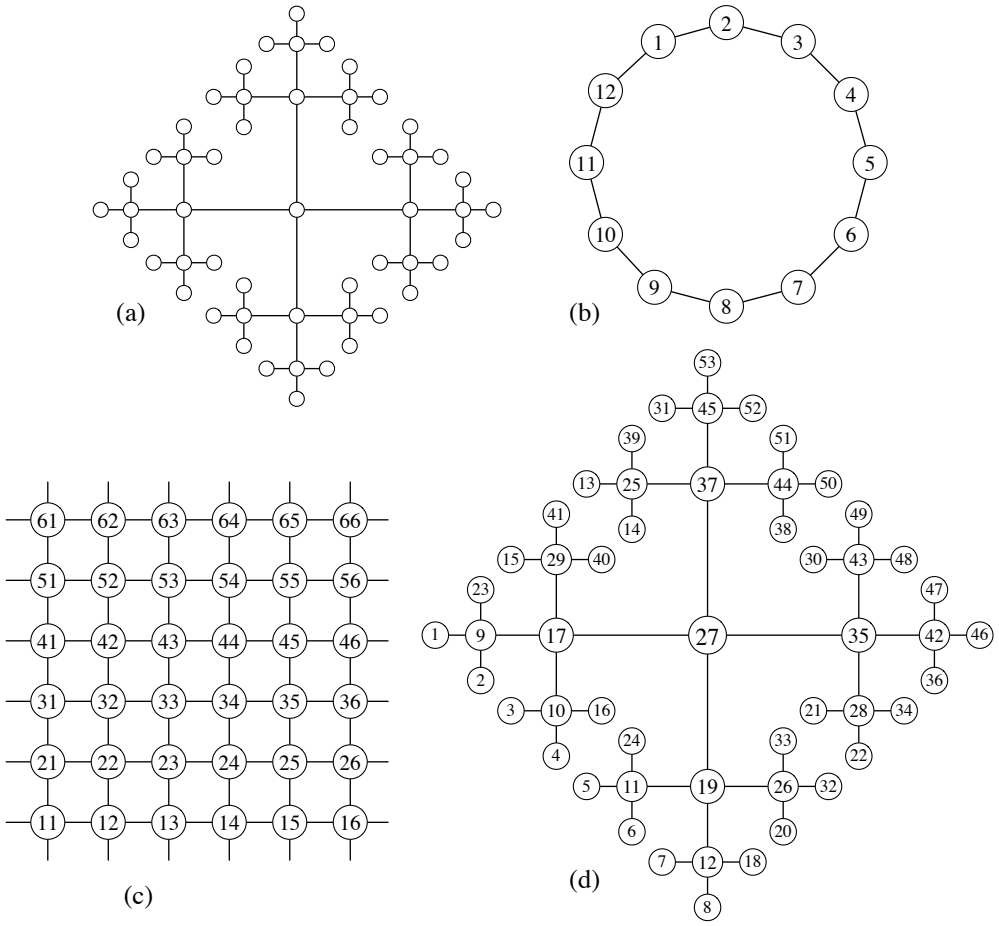


Figure 6: Examples of graphs that satisfy some of the desirable properties from Section 5.9.

However, what we can do is to put these ingredients together to prove a lower bound for maximal *fractional* matchings. We can show that maximal fractional matchings cannot be solved in time $o(\Delta)$ independently of n in the LOCAL model—recall that there is a matching upper bound of $O(\Delta)$. There are many technical details to be taken care of, but the key steps are as follows [14]:

- Prove a lower bound for the EC model, with the help of the unfold-and-mix technique (Section 5.6).
- EC $\approx$ PO: Use the properties of maximal fractional matchings to show that a lower bound for EC implies a lower bound for PO.
- PO $\approx$ OI: Use homogeneous graphs (Section 5.9) to show that a lower bound for PO implies a lower bound for OI.
- OI $\approx$ LOCAL: Use Ramsey's theorem to show that a lower bound for OI implies a lower bound for LOCAL.

Finally, we can prove that in the LOCAL model, randomised $f(\Delta)$-time algorithms cannot do any better than deterministic algorithms with this kind of graph problems, and we have the theorem that we have been looking for: even if we use randomised algorithms, and even if we can exploit the full power of the LOCAL model, there is no distributed algorithm that finds a maximal fractional matching in time $o(\Delta)$ independently of n.

6 Conclusions

So far we have identified the first pure coordination problem—maximal fractional matchings—with the following properties: the problem can be solved in time $O(\Delta)$ independently of n, and there is a matching lower bound showing that it cannot be solved in time $o(\Delta)$ independently of n.

Intuitively, the unfold-and-mix technique shows that there is need for local coordination with nodes that are at distance up to $\Theta(\Delta)$. All other parts of the proof—e.g. homogeneous graphs and Ramsey's theorem—are just technicalities that are needed to show that algorithms cannot "cheat" somehow by exploiting unique node identifiers and randomness.

6.1 Current Obstacles

Our hope is that we could prove that many other problems—for example, maximal matchings—also have a similar source of hardness that is related to local coordination with nodes at distance up to $\Theta(\Delta)$. We conjecture that, for example,

maximal matching cannot be solved in time $o(\Delta) + O(\log^* n)$ with any algorithm. In essence, we believe that the Panconesi–Rizzi algorithm [31] with a running time of $O(\Delta + \log^* n)$ is optimal for $n \gg \Delta$.

Currently, there seem to be two obstacles that prevent us from proving such a theorem, both of which are related to the term $\Theta(\log^* n)$ in the lower bound that we are looking for.

1. The final step OI $\approx$ LOCAL (Section 5.8): The Ramsey-based argument that we use to show that OI and LOCAL are equally strong fails for $\Theta(\log^* n)$-time algorithms.

2. The starting point in the EC model (Section 5.6): In time $\Theta(\log^* n)$ an algorithm can find a vertex colouring that breaks the symmetry between adjacent nodes. Therefore it is no longer possible to use self-loops to construct instances that are "fragile".

However, once again we can try to deal with these two obstacles one by one. It turns out that there is a natural graph problem that would let us focus on the second obstacle first—the problem is bipartite maximal matching that we already discussed in Section 3.5.

6.2 Roadmap for the Future

Recall that maximal matching in bipartite 2-coloured graphs can be solved in time $O(\Delta)$ independently of n. We conjecture that bipartite maximal matchings are a pure coordination problem that cannot be solved in time $o(\Delta)$ independently of n. The current techniques are not yet sufficient to prove it, but it seems that we are now facing just one obstacle—how to use the unfold-and-mix technique to construct "fragile" instances even in the presence of an edge colouring that breaks symmetry.

This suggests the following roadmap for future research:

1. Extend the unfold-and-mix technique so that we can prove a linear-in-Δ bound for bipartite maximal matchings.

2. Then extend the Ramsey-based argument so that we can prove a similar bound for maximal matchings in general.

3. Then extend the techniques so that we can prove similar bounds for other coordination problems, for example, independent sets, vertex colourings, and edge colourings.

This seems to be a long road ahead, but it could lead to a resolution of major open questions related to distributed time complexity. Such results could find applications also in other areas of theoretical computer science. In prior work, tight lower bounds for distributed symmetry breaking have implied tight lower bounds for e.g. decision tree complexity and models of parallel computing [11, 30], and perhaps tight lower bounds for local coordination would find similar applications.

Acknowledgements

This article is based on the material that I presented in the ADGA 2014 workshop, http://adga2014.hiit.fi/. Many thanks to Christoph Lenzen for inviting me to give the talk, to the workshop participants for discussions, to Stefan Schmid for asking me to write this article and for his feedback on it, to Przemysław Uznański and Tuomo Lempiäinen for their helpful comments, and to my coauthors Mika Göös and Juho Hirvonen without whom the results described in this article would not even exist.

References

[1] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

[2] Matti Åstrand, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. Local algorithms in (weakly) coloured graphs, 2010. arXiv:1002.0125.

[3] Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proc. 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2010)*, pages 294–302. ACM Press, 2010. doi:10.1145/1810479.1810533.

[4] Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981. doi:10.1016/0196-6774(81)90020-1.

[5] Leonid Barenboim and Michael Elkin. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. In *Proc. 41st Annual ACM Symposium on Theory of Computing (STOC 2009)*, pages 111–120. ACM Press, 2009. doi:10.1145/1536414.1536432.

[6] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

[7] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012)*, pages 321–330. IEEE Computer Society Press, 2012. doi:10.1109/FOCS.2012.60.

[8] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

[9] Andrzej Czygrinow, Michał Hańćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. Distributed 2-approximation algorithm for the semi-matching problem. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, volume 7611 of *Lecture Notes in Computer Science*, pages 210–222. Springer, 2012. doi:10.1007/978-3-642-33651-5_15.

[10] Andrzej Czygrinow, Michał Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *Proc. 22nd International Symposium on Distributed Computing (DISC 2008)*, volume 5218 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2008. doi:10.1007/978-3-540-87779-0_6.

[11] Faith E. Fich and Vijaya Ramachandran. Lower bounds for parallel computation on linked structures. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1990)*, pages 109–116. ACM Press, 1990. doi:10.1145/97444.97676.

[12] Alex Gamburd, Shlomo Hoory, Mehrdad Shahshahani, Aner Shalev, and Balint Virág. On the girth of random Cayley graphs. *Random Structures & Algorithms*, 35(1):100–117, 2009. doi:10.1002/rsa.20266.

[13] Mika Göös, Juho Hirvonen, and Jukka Suomela. Lower bounds for local approximation. *Journal of the ACM*, 60(5):39:1–23, 2013. doi:10.1145/2528405. arXiv:1201.6675.

[14] Mika Göös, Juho Hirvonen, and Jukka Suomela. Linear-in-Δ lower bounds in the LOCAL model. In *Proc. 33rd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2014)*, pages 86–95. ACM Press, 2014. doi:10.1145/2611462.2611467. arXiv:1304.1007.

[15] Mika Göös and Jukka Suomela. No sublogarithmic-time approximation scheme for bipartite vertex cover. *Distributed Computing*, 27(6):435–443, 2014. doi:10.1007/s00446-013-0194-z. arXiv:1205.4605.

[16] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey Theory*. John Wiley & Sons, New York, 1980.

[17] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 219–225. Society for Industrial and Applied Mathematics, 1998.

[18] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM Journal on Discrete Mathematics*, 15(1):41–57, 2001. doi:10.1137/S0895480100373121.

[19] Juho Hirvonen and Jukka Suomela. Distributed maximal matching: greedy is optimal. In *Proc. 31st Annual ACM Symposium on Principles of Distributed Computing (PODC 2012)*, pages 165–174. ACM Press, 2012. doi:10.1145/2332432.2332464. arXiv:1110.0367.

[20] Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986. doi:10.1016/0020-0190(86)90144-4.

[21] Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proc. 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, pages 138–144. ACM Press, 2009. doi:10.1145/1583991.1584032.

[22] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 300–309. ACM Press, 2004. doi:10.1145/1011767.1011811.

[23] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 980–989. ACM Press, 2006. doi:10.1145/1109557.1109666.

[24] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: lower and upper bounds, 2010. arXiv:1011.5470.

[25] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

[26] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

[27] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.

[28] Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM Journal on Discrete Mathematics*, 4(3):409–412, 1991. doi:10.1137/0404036.

[29] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

[30] Noam Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, 1991. doi:10.1137/0220062.

[31] Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.

[32] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[33] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys*, 45(2):24:1–40, 2013. doi:10.1145/2431211.2431223. http://www.cs.helsinki.fi/local-survey/.

[34] Jukka Suomela. *Distributed Algorithms*. 2014. Online textbook. http://users.ics.aalto.fi/suomela/da/.