HAL
open science

# Locality and Checkability in Wait-Free Computing

Pierre Fraigniaud, Sergio Rajsbaum, Corentin Travers

## To cite this version:

Pierre Fraigniaud, Sergio Rajsbaum, Corentin Travers. Locality and Checkability in Wait-Free Computing. DISC, 2011, Unknown, pp.333-347. hal-00992781

**HAL Id: hal-00992781**

**https://inria.hal.science/hal-00992781v1**

Submitted on 19 May 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Locality and checkability in wait-free computing


Pierre Fraigniaud*
LIAFA
CNRS and U. Paris Diderot
France
pierref@liafa.jussieu.fr

Sergio Rajsbaum†
Instituto de Matemáticas
U. Nacional Autónoma de México
Mexico
rajsbaum@math.unam.mx

Corentin Travers‡
LaBRI
U. of Bordeaux and CNRS
France
travers@labri.fr



**Abstract**

A *task* $T$ is described by a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ is the set of input configurations, $\mathcal{O}$ is the set of output configurations, and $\Delta$ is the specification of the task mapping every input to a set of possible outputs. We consider notions of *locality* that are inherent to the task specification, independently from the computational model. We observe that tasks as different as $(d+1)$-coloring for degree-$d$ graphs, in the network setting, and consensus, in the wait-free setting, both satisfy the local condition $\Delta(\pi(s)) \subseteq \pi(\Delta(s))$ stating that any output $\Delta(\pi(s))$ for a partial input $\pi(s)$ is a partial output $\pi(\Delta(s))$ for the full input $s$. This property, called *monotony*, captures locality in a very general sense, independent of the computing environment, by expressing the relationship between the various scales of computation, from the individual processes to the whole system. Unfortunately, monotony is not sufficient to insure efficient computation in the network setting, neither it is sufficient to insure solvability in the wait-free setting. This paper thus investigates other notions of locality which, in the framework of wait-free computing, open up to new, rich classes of tasks.

First, we define a task to be *projection-closed* if $\pi(\Delta(s)) \subseteq \Delta(\pi(s))$, and prove the perhaps surprising fact that projection-closed tasks are precisely those tasks that are wait-free *checkable*, that is tasks $T = (\mathcal{I}, \mathcal{O}, \Delta)$ for which there exists a wait-free distributed algorithm enabling, given a pair $(s, t)$, $s \in \mathcal{I}$, $t \in \mathcal{O}$, to check whether $t \in \Delta(s)$, i.e., to decide whether $t$ is a valid output for $s$.

Our second main contribution is dealing with a stronger notion of locality, where $\Delta(\pi(s)) = \pi(\Delta(s))$. A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be *locality-preserving* if and only if $\mathcal{O}$ is a covering complex of $\mathcal{I}$, that is there exists a map $p : \mathcal{O} \to \mathcal{I}$ which agrees with $\Delta$ (i.e., $t \in \Delta(p(t))$ for every $t \in \mathcal{O}$). This topological property yields obstacles for wait-free solvability different in nature from the classical agreement impossibility results. Apart from the identity task, locality-preserving tasks are not wait-free solvable. On the other hand, locality-preserving tasks are projection-closed, and therefore they are wait-free checkable. We provide a classification of locality-preserving tasks in term of their computational power, by establishing a correspondence between locality-preserving tasks and subgroups of the *edgepath* group of a complex. Using this correspondence, we prove the existence of hierarchies of locality-preserving tasks, each one containing a universal task (induced by the universal covering complex), and at the bottom the trivial identity task.



**Keywords:** distributed verification, local computing, wait-free, decision task.


**regular paper submission** not eligible for best student paper award
**contact author** Sergio Rajsbaum


*Additional supports from the ANR projects ALADDIN and PROSE, and from the INRIA project GANG.
†Additional supports from UNAM-PAPIIT and PAPIME.
‡Additional supports from ANR project SPREAD and by the INRIA project REGAL.

# 1 Introduction

A *task* is a distributed coordination problem in which each process starts with a private input value, communicates with the other processes by applying operations to shared objects, and eventually decides a private output value. It can be described by a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ is the set of input configurations, $\mathcal{O}$ is the set of output configurations, and $\Delta$ is the specification of the task mapping every input configuration to a set of possible output configurations. A *protocol* is a distributed program that solves a task.

Often it is useful to consider (input or output) configurations where the states of only a subset of the processes are specified. If $s$ is an input configuration, $\pi(s)$ denotes the configuration obtained by projecting out the processes specified by $\pi$. Then, the map $\Delta$ of a task specifies with $\Delta(\pi(s))$ the valid output configurations for $\pi(s)$. For instance, let $G$ be a network of processes, with one process per node, exchanging information along their incident edges. Assume one wants to color the nodes of $G$ in such a way that two adjacent nodes are assigned different colors. The literature tackling this task (see, e.g., [7, 31]) generally assumes that $d+1$ colors are available, where $d$ denotes the maximum degree of $G$. The main motivation for this assumption is that every graph is $(d+1)$-colorable, whereas there are graphs that are not $d$-colorable, like, e.g., the complete graphs, or the cycles of odd length. A more careful look at the specification $\Delta$ of the $(d+1)$-coloring task enables to identify a stronger property: any partial $(d+1)$-coloring $\Delta(\pi(s))$ for a subgraph $\pi(s)$ induced by any set of processes specified by $\pi$, can be extended to a $(d+1)$-coloring $\Delta(s)$ for $s$. In other words, the $(d+1)$-coloring specification $\Delta$ satisfies the *monotony* condition

$$\Delta(\pi(s)) \subseteq \pi(\Delta(s)) \tag{1}$$

for every set of processes $s$, and every projection $\pi$. Instead, the specification of the $d$-coloring task does not satisfy this inclusion, even in networks that are $d$-colorable. For instance, if $s$ denotes the four nodes of a 4-cycle, and $\pi(s)$ denotes two antipodal nodes in this cycle, then the output consisting in coloring the nodes of $\pi(s)$ with distinct colors cannot be extended to a valid output for the whole set $s$. It may thus be not coincidental that 3-coloring the $n$-node ring can be achieved in $O(\log^* n)$ rounds (see [13]) whereas 2-coloring rings (of even size) requires $\Omega(n)$ rounds (see [32]).

The monotony condition expresses a notion of *locality* satisfied by the task, that can be phrased as: *any output for a partial input is a partial output for the full input*. It is important to observe that this notion expresses a form of locality that is inherent to the specification of a task, and independent of the distributed computing model. For instance, at the other extremity of the wide spectrum of distributed computing models, previous work in wait-free computing, where asynchronous processes subject to crash failures communicate via a read/write shared memory, often assumes tasks satisfying the monotony condition. Typically, consensus satisfies it. (Note that the inclusion is strict for consensus, whereas equality holds for $(d+1)$-coloring). Monotony captures locality in a general sense, independent of the computing environment, by expressing the relationship between the various scales of computation. Indeed, monotony relates the specification for subsets of processes to the specification for larger sets. Thus, it relates the individual behavior of each process with the behavior of small group of processes, which in turn is related to the behavior of larger and larger groups, until one reaches the scale of the whole system.

A weaker form of locality results from putting the burden on the protocol to find a right output in $\Delta(\pi(s))$ for $\pi(s)$, that can be extended to an output $\Delta(s)$ for $s$. In other words, instead of imposing a task to satisfy Eq. 1, it might be sufficient to assume the *intersection-closeness* condition

$$\Delta(\pi(s)) \cap \pi(\Delta(s)) \neq \emptyset. \tag{2}$$

This weaker condition is an obvious requirement for a task to be wait-free solvable[1] whereas monotony is not a

[1] Assume it is not satisfied for some $s$ and $\pi$. Consider the scenario in which all processes in $s$ participate, but those in $s \setminus \pi(s)$ are very slow, and do not take any step for some time. The wait-free requirement forces the processes in $\pi(s)$ to decide values alone. Assume they decide values $t \in \Delta(\pi(s))$. Now, the processes in $s \setminus \pi(s)$ wake up, and start taking their steps. They have to decide some values, so that the processes in $s$ collectively produce an output $t' \in \Delta(s)$. This output must satisfy $\pi(t') = t$ because all processes in $\pi(s)$ have already terminated with output $t$. This $t'$ does not exist whenever $\Delta(\pi(s)) \cap \pi(\Delta(s)) = \emptyset$.

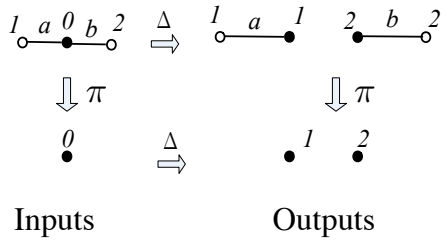



Figure 1: An intersection-closed task, which is not monotone, and not wait-free solvable. The projection $\pi$ depicted is for the black process. The black process always starts with 0. When both run, they must agree on the input of the white process, either 1 or 2. If the white process runs solo, it must decide its own input. If the black process runs solo, it can decide 1 or 2.

necessary condition for wait-free solvability. However, putting this burden on the shoulders of the protocol may be too much. Indeed, for $s \neq s'$ with $\pi(s) = \pi(s')$, it may be the case that $\pi(\Delta(s)) \cap \pi(\Delta(s')) = \emptyset$ even if Eq. 2 is satisfied for all $s, s'$, and $\pi$. See Fig. 1 for an example. In this case, the processes in $\pi(s)$ running alone have no clue whether they have to output a solution extendable to an output for $s$ or for $s'$. This is why tasks are usually assumed to satisfy the monotony condition instead of the weaker intersection-closeness condition.

Unfortunately, neither the intersection-closeness condition nor even the monotony condition provide sufficient constraints for solvability, or for efficient computation. For instance, in the network setting, monotony is not a guaranty for a task to be solved by having every node merely inspecting nodes at a constant distance (cf. the $\Omega(\log^* n)$ lower bound for $(d+1)$-coloring [32]). Neither it is, in the wait-free setting, a guaranty for a task to be solvable (cf. the FLP impossibility result for consensus [17]). This paper investigates two other notions of locality, namely the *projection-closeness* condition, obtained by simply reversing the monotony condition,

$$\pi(\Delta(s)) \subseteq \Delta(\pi(s)) \tag{3}$$

and the one resulting from combining monotony with projection-closeness. These latter two notions are shown to be rich concepts, enabling to capture important features of the computational nature of tasks, at least as far as wait-free computing is concerned.

## 1.1 Our results

The objective of this paper is to investigate the ability of a shared memory system to solve tasks satisfyling various forms of locality. Our investigation is performed in the wait-free setting where computation tolerates the halting failures or delays by $n-1$ out of $n$ processes. In this context, it is convenient to view $\mathcal{I}$ and $\mathcal{O}$ as complexes (including configurations for *any* number of processes, between 1 and $n$), with the specification $\Delta$ mapping every simplex $s \in \mathcal{I}$ to a sub-complex $\Delta(s)$ of $\mathcal{O}$. See Figure 2 for a graphical representation of our results.

First, we show that the projection-closeness condition of Eq. 3 is closely related to the ability of *checking* a task. Informally, for checking a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, every process $i \in [n]$ is given a pair $(s_i, t_i)$, and the participating processes must check that the simplex $t$ with decision values $t_i$, $i = 1, \ldots, n$, is a valid output simplex according to $\Delta$, for an input simplex $s$ with values $s_i$. Deciding the latter is performed according to the (informal) specifications:

- if $t \in \Delta(s)$ then all participating processes must output "yes",
- otherwise at least one participating process must output "no".

Hence checking the task $T$ corresponds to solving a *checking task*, where the input entry of the $i$th process is a pair $(i, (s_i, t_i))$, and where each process must return either "yes" or "no". We prove that a task is wait-free checkable if and only if it is projection-closed (cf. Theorem 3.3). It is remarkable that the locality notion expressed by Eq. 3 captures precisely the ability to wait-free verify the results of a computation, even if these results have been

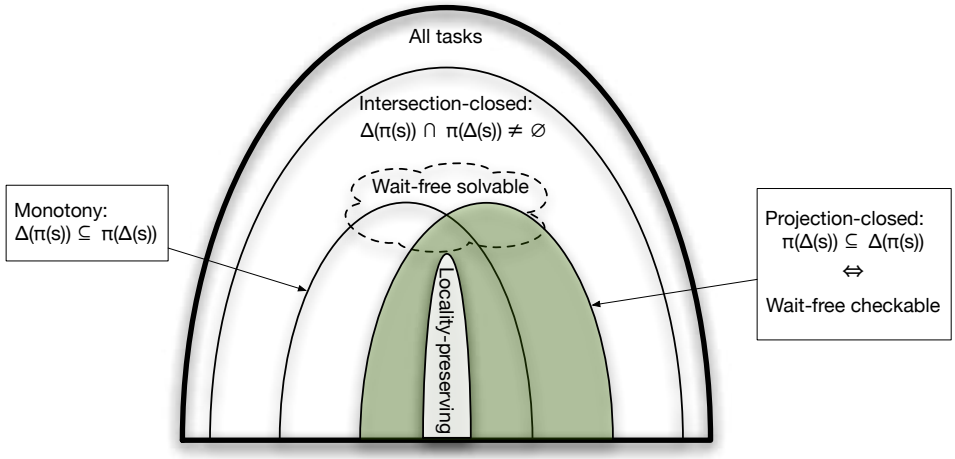



Figure 2: The universe of tasks

obtained using more resources (e.g., oracles) or stronger models (e.g., $t$-resilience). Moreover, we show that the set of projection-closed tasks is large by proving that determining whether a projection-close task is wait-free solvable remains undecidable (cf. Theorem 3.4). This latter result is obtained by proving that every task is equivalent to a wait-free checkable task (via implementations that preserve step-complexity).

Next, we turn our attention to tasks that are both projection-closed and monotone. As for monotony alone, the two conditions combined do not seem to provide sufficient structural constraints for relating them to wait-free computability. Nevertheless, we were able to identify a subclass of these tasks, that offers a stronger notion of locality expressible in the framework of algebraic topology. A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be *locality-preserving* if and only if $\mathcal{O}$ is a *covering complex* of $\mathcal{I}$, that is there exists a map $p : \mathcal{O} \to \mathcal{I}$ which agrees with $\Delta$, i.e.,

$$\exists p : \mathcal{O} \to \mathcal{I} \mid \forall t \in \mathcal{O}, t \in \Delta(p(t)). \tag{4}$$

We show that, indeed, locality-preserving form a subclass of the monotone and projection-closed tasks (cf. Theorem 4.3). The notion of locality captured by locality-preserving tasks is made explicit topologically. Informally, locality-preserving tasks are tasks where, from the perspective of a subset $\pi(s)$ of processes with given input values, the possible inputs to the other processes look identically in structure to their possible outputs (see Figure 3 for an example).

We show that locality-preserving tasks form a wide and rich class of tasks. We *classify* locality-preserving tasks in terms of their computational power. By identifying a correspondence between locality-preserving tasks and covering complexes (the classic algebraic topology notion used to study locality), we prove that locality-preserving task $T$ implements locality-preserving task $T'$ if and only if $H \subseteq H'$, where $H, H'$ are groups associated to each of the tasks. This result demonstrates the existence of an infinite set of partial orders of locality preserving tasks. Each of these partial orders contains a hierarchy of locality-preserving tasks between the trivial identity task, and a universal task for this partial order. Some of these partial orders are finite, while others are infinite. As in [27], we use topology techniques both to prove impossibility results, and to show when one task can implement another.

## 1.2 Related work

The locality notions considered in this paper, are local in the sense that they can be checked individually for pairs $(s, t)$, $s \in \mathcal{I}$, $t \in \mathcal{O}$. The main obstacles to wait-free solvability studied in the past, most notably for set agreement and *renaming* [3], are of a different nature. Indeed, any wait-free protocol is actually a mapping from a subdivision

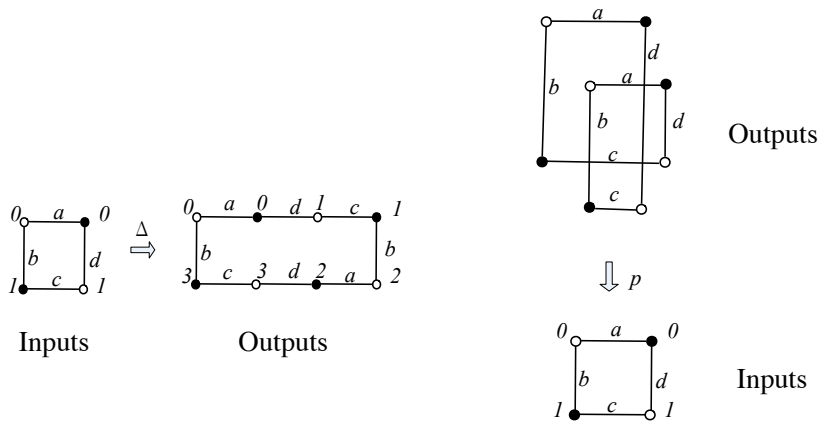



Figure 3: Two-cover task. The task is for two processes, which start with binary inputs. On the left side of the figure, the specification of the task says that if both start with the same value, they must decide the same value: if they start with 0, decide either 0 or 2; if they start with 1, decide 1 or 3. If they start with different values, the valid outputs are defined in the figure, via edge labels $a, b, c$, or $d$. The relation $\Delta$ defines also the possible outputs when only one process runs solo: e.g., when the white process starts with 0, it can decide 0 or 2, while if it starts with 1, it can decide 1 or 3. Notice that $\mathcal{O}$, a cycle of length 8, locally looks like $\mathcal{I}$, a cycle of length 4, in the sense that the 1-neighborhood of each vertex $v$ in $\mathcal{I}$ is identical to the 1-neighborhood of a corresponding vertex in $\Delta(v)$. In the right side of the figure it is shown how $\mathcal{O}$ covers $\mathcal{I}$, by wrapping around it twice, where $p$ identifies edges with the same label.

of the input complex to the output complex [28]. Hence, topological properties must be preserved by a wait-free protocol. Checking these properties is hard, and, in fact, determining whether a task is wait-free solvable in not decidable [22, 26].

A different notion of locality has received lots of attention in the framework of network computing. Specifically, the so-called $\mathcal{LOCAL}$ and $\mathcal{CONGEST}$ models [37] have been designed to study communication locality issues. One prominent result in this framework is the $\Omega(\log^* n)$ lower bound [32] for the number of rounds required to 3-color the nodes of the $n$-node ring network. In several papers in this framework, the main focus is on whether randomization helps [36], on the impact of non-determinism [19], and on the power of oracles providing nodes with information about their environment [18]. The impact on the design of efficient protocols of the absence of a priori knowledge on the global environment has been recently addressed in [30].

Starting with the pioneering work by Angluin [1], covering spaces have been used to derive impossibility results in *anonymous networks*, but only in the 1-dimensional case of graph coverings. Sufficient and sometimes necessary conditions on the communication graph and on the initial common knowledge for solving fundamental distributed problems such as leader election or termination detection are given in, e.g., [1, 12, 35, 11], under several models of local computation. See [12] for an introduction to local computation in anonymous networks.

One can roughly classify the methods for ensuring the correctness of a program as either *verifying*, *testing*, or *checking*. Unlike verifying and testing, checking is performed at run time. A sequential checker [8] consists of a battery of tests (performed at run time) which compare the output of the program with either a predetermined value, or with a function of the outputs of the same program corresponding to different inputs. A similar idea is *spot-checking* [16] where the goal it to know if the output is reasonably correct, i.e., close in some problem-specific distance to the correct output. Related areas may be *learning*, where samples of outputs are used to infer the task it is being solved, as in [21] and *property testing* [24]. Blum et al. [9] introduced the notion of program *testers* and *correctors*, see also [23, 33].

In the parallel and/or distributed computing context, the results are not so advanced as in the sequential setting. Parallel program checking has been studied in the PRAM model [39]. In the synchronous model, distributed self-testing and correcting protocols that tolerate crash failures are presented for the byzantine generals task in

[20]. Self-testing/correcting is reminiscent of the notion of checking as a means of making a distributed algorithm *self-stabilizing*, as explored in [2, 5] in the synchronous setting. In the framework of network computing, distributed verification has been addressed only recently (see, e.g., [14]), though previous research on proof labeling schemes [29] already gave some insights on the ability of checking global predicates locally (see also the recent paper [25]).

### 1.3 Organization

The paper is made up of 4 sections. Section 2 presents the wait-free distributed computing model computing, and recalls the notion of tasks. Our first locality condition, namely projection-closeness, and its relation with wait-free checking is explored in Section 3. Our structural results on the computational power of locality-preserving tasks are presented in Section 4. Due to space limitation, proofs are presented in Appendix A and B.

## 2 Model

We consider a standard read/write shared memory wait-free model. The system consists of $n$ asynchronous processes, denoted by the integers in $[n] = \{1, \ldots, n\}$. The processes can fail by crashing (a crashed process never recovers). Any number of processes can crash, at any time. We recall here the main features of this model, and refer to [6, 28, 34] for a more detailed and accurate description.

The processes that take an infinite number of steps in a run are correct, the others crashed. If a process crashes initially, it does not take any steps, and we say it does not participate in the run. A process participates in a run if it takes at least one step. At the core of the model is the following assumption. We enforce protocols to be *wait-free*, that is to avoid all instructions that would cause a process to wait for the result of the action of another process. In particular, in a wait-free protocol, a process $i$ cannot check whether another process has crashed, or is just very slow.

When solving a task, in each run of a protocol, processes start with private input values. A process $i \in [n]$ is not aware of the inputs of other processes. The initial states of the processes differ only in their input values. Each process $i$ has to eventually decide irrevocably on a value. Consider a run $r$ where only a subset of $k$ processes participate, $1 \leq k \leq n$. These processes have distinct identities $\{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$, where for every $i \in [k]$, $\mathrm{id}_i \in [n]$. A set $s = \{(\mathrm{id}_1, x_1), \ldots, (\mathrm{id}_k, x_k)\}$, is used to denote the input values or decision values in the run, where $x_i$ denotes the value of the process with identity $\mathrm{id}_i$ (either an input value, or a decision value). We denote by $\mathrm{ID}(s)$ the set of identities of the processes in $s$, i.e., $\mathrm{ID}(s) = \{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$. The input values of processes not in $\mathrm{ID}(s)$ are irrelevant, as they do not participate in the run, so they are not included in $s$.

Let $s'$ be a subset of $s = \{(1, x_1), \ldots, (n, x_n)\}$, $\mathrm{ID}(s) = [n]$. We say that $\pi(s) = s'$, for a *projection* $\pi$, that eliminates processes in $\mathrm{ID}(s) \setminus \mathrm{ID}(s')$. Because any number of processes can crash, all such subsets $s'$ of $s$ are of interest, to consider runs where only processes in $\mathrm{ID}(s')$ may participate. That is, the set of possible input sets forms a *complex* because its sets are closed under containment. Similarly, the set of possible output sets also form a complex. Following the topology notation, the sets of a complex are called *simplexes*.

More formally, a *complex* $K$ is a set of vertices $V(K)$, and a family of finite, nonempty subsets of $V(K)$, called *simplexes*, satisfying: (i) if $v \in V(K)$ then $\{v\}$ is a simplex, and (ii) if $s$ is a simplex, so is every nonempty subset of $s$. The *dimension* of a simplex $s$ is $|s| - 1$, the dimension of $K$ is the largest dimension of its simplexes, and $K$ is *pure* of dimension $k$ if every simplex belongs to a $k$-dimensional simplex. A 1-dimensional complex $K$ is thus simply a graph[2]. In distributed computing, one refers to *colored* simplexes (and complexes), since each vertex $v$ of a simplex is labeled with a distinct process identity $i \in [n]$.

[2]It is the graph whose nodes are the vertices of $K$, and whose edges and nodes are the simplexes of $K$. More generally, the concept of complexes is, in some sense, a natural extension of the concept of graphs, to higher dimensions. They can also be viewed as forming a subclass of *hypergraphs*, in which every non-empty subset of an hyperedge must be an hyperedge.

We denote by $\mathcal{I}$ the input complex, and by $\mathcal{O}$ the output complex. An *input-output pair* is a pair $(s, t)$ made of a simplex $s \in \mathcal{I}$ and a simplex $t \in \mathcal{O}$, with $\mathrm{ID}(t) \subseteq \mathrm{ID}(s)$. The strict inclusion $\mathrm{ID}(t_i) \subset \mathrm{ID}(s)$ takes into account the case when the processes in $\mathrm{ID}(s) \setminus \mathrm{ID}(t)$ fail, and do not decide, which, in the wait-free model, should not prevent the participating non-failing processes to decide. A *task* $T$ is described by a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ and $\mathcal{O}$ are pure $(n-1)$-dimensional complexes, and $\Delta$ is a map from $\mathcal{I}$ to the set of non-empty sub-complexes of $\mathcal{O}$, satisfying $(s, t)$ is an input-output pair for every $t \in \Delta(s)$. Intuitively, $\Delta$ specifies, for every simplex $s \in \mathcal{I}$, the valid outputs for the processes in $\mathrm{ID}(s)$ that may participate in the computation. We assume that $\Delta$ is (sequentially) computable.

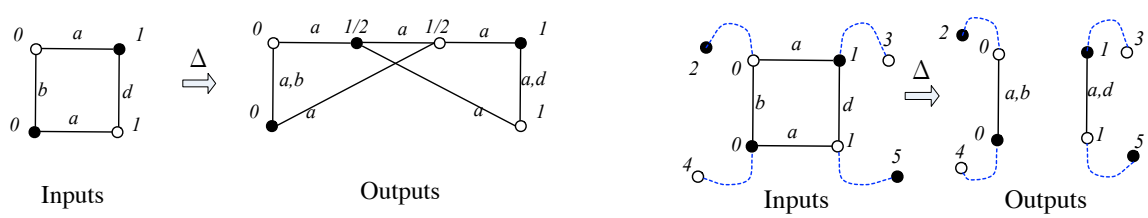
A protocol $\mathcal{A}$ *solves* task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if, for every simplex $s \in \mathcal{I}$, and every run $r$ of $\mathcal{A}$ on $s$, every correct process decides, and the simplex $t$ corresponding to these decisions belongs to $\Delta(s)$.

## 3 Projection-closeness and wait-free checkability

This section addresses the first of the two notions of locality tackled in this paper. The locality considered here is obtained by simply reversing the direction of the inclusion in the monotony condition of Eq. 1.

**Definition 3.1** *A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is* projection-closed *if and only if for every $s \in \mathcal{I}$ and every projection $\pi$, we have $\pi(\Delta(s)) \subseteq \Delta(\pi(s))$.*

Projection-closeness is as poorly related to wait-free solvability as is the intersection-closeness notion of Eq. 2. Consider for example, approximate agreement [15] for two processes, illustrated in Fig. 4a. While it is wait-free solvable, it is not projection-closed. Intuitively, because the *validity* requirement in dimension 0 "if a process runs solo, has to decide its own value" conflicts with the *agreement* requirement in dimension 1 "when processes start with different values, they can decide any values, as long as they are at most $1/2$ apart from each other." More precisely, take the input simplex $s = \{(1, 0), (2, 1)\}$ (top edge labeled $a$ in the figure), and apply the projection $\pi$ that eliminates process 2, to obtain $\pi(\Delta(s))$ which consists of all vertices for process 1 in the output complex, while $\Delta(\pi(s))$ consists of only vertex $t = \{(1, 0)\}$. Notice that for the same reason, consensus is also not projection-closed; in contrast, consensus is not wait-free solvable.




(a) Approximate agreement task

(b) Projection-closed consensus

Figure 4: Two tasks for 2 processes.

Instead, projection-closeness is well related to *checking* the result of a computation supposedly solving a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$. Intuitively, processes get as inputs the vertices of $s, t$, $s \in \mathcal{I}$, $t \in \mathcal{O}$, and want to decide if indeed the computation of $t$ for $s$ was correct, namely, if $t \in \Delta(s)$. That is, each process $i$ gets as input a pair $(x_i, y_i)$, such that the $x_i$ values define $s$, while the $y_i$ values define $t$. Then, checking $T$ corresponds to solving a *checking task* $T_c$:

- If $t \in \Delta(s)$ then all processes that decide must output 1, interpreted as "yes".
- If $t \notin \Delta(s)$ then whenever *all* participating processes decide, one of them must output 0, interpreted as "no". When not all participating processes decide, then some may decide 1 and others 0.

Formally, $T_c = (\mathcal{I} \times \mathcal{O}, S^n, \Delta_c)$. The input complex $\mathcal{I} \times \mathcal{O}$ consists of all simplexes $s \times t$, $s \in \mathcal{I}$, $t \in \mathcal{O}$, $\mathrm{ID}(s) = \mathrm{ID}(t)$, where, for every $i \in [n]$,

$$(i, (x_i, y_i)) \in s \times t \iff \begin{cases} (i, x_i) \in s \\ (i, y_i) \in t \end{cases}$$

The output complex $S^n$ consists of all simplexes where processes decide values in $\{0, 1\}$. Now, to define $\Delta_c$, let $S^J$ be the sub-complex of $S^n$ induced by the processes in $J$, for $J \subseteq [n]$. In other words, $S^J = \pi(S^n)$ where $\pi$ is the projection from $[n]$ to $J$. Also, for $J \subseteq [n]$, let $Y[J] = \{(i, 1), i \in J\}$ be the simplex corresponding to all processes in $J$ outputting "yes", and let $\overline{Y}[J]$ be the complex induced by $Y[J]$. We have now all the ingredients to define $\Delta_c$. For any $s \times t \in \mathcal{I} \times \mathcal{O}$ with $\mathrm{ID}(s) = \mathrm{ID}(t) = J$,

$$\Delta_c\Big(s \times t\Big) = \begin{cases} \overline{Y}[J] & \text{if } t \in \Delta(s) \\ S^J \setminus Y[J] & \text{otherwise.} \end{cases}$$

We now define wait-free checkability as follows:

**Definition 3.2** *A task $T$ is wait-free checkable if and only if its corresponding task $T_c$ is wait-free solvable.*

The following states the exact correspondence between projection-closeness and wait-free checkability.

**Theorem 3.3** *A task $T$ is wait-free checkable if and only if it is projection-closed.*

We now prove that the set of wait-free checkable tasks, or, equivalently, the set of projection-closed tasks, forms a large class of tasks.

**Theorem 3.4** *Determining whether a wait-free checkable task is wait-free solvable is undecidable.*

To establish the theorem, we show that every task is essentially equivalent to a wait-free checkable task, under a very strong notion of equivalence. Following [27], a task $T'$ *implements* task $T$ (both defined on $n$ processes) if one can construct a wait-free protocol for $T$ by calling one instance of a black box for $T'$ followed by any number of operations on read-write registers[3]. We write $T \leq T'$ to emphasize that $T'$ is at least as powerful as $T$. In Section 4.3 we use this implementation, while for the theorem here, it suffices to use its particular case where no operations on read-write registers are used; namely, if there exists a wait-free protocol $\mathcal{A}$ that solves $T$, in which processes can call one instance of a black box that solves $T'$, and do not execute any read-write operations:

**Definition 3.5** *We say $T \leq T'$ if task $T'$ implements task $T$, and $T \preceq T'$ if there is an implementation without executing any read-write operations.*

Based on Definition 3.5, one can define the following equivalence relation $\sim$ between tasks

$$T \sim T' \iff (T \preceq T' \text{ and } T' \preceq T)\ .$$

This equivalence notion is the key ingredient in the proof of Theorem 3.4 below. First consider as an example consensus, which as mentioned above, in not wait-free checkable. However, consensus is equivalent to the *checkable consensus* task, depicted in Figure 4b for two processes. This task is obtained from consensus, by adding new input and output values $2, 3, 4, 5$, and then allowing for processes running solo on inputs $0, 1$, to decide any value (but when a process starts with $2, 3, 4$ or $5$ it decides its own input always). Proof in Appendix A.

[3] A more general notion of implementation allows read-write operations before calling the black box, and calling the black box more than once.

# 4 Locality-preserving tasks

In this section, we turn our attention to locality-preserving tasks, which are both monotone and projection-closed (see Fig. 2) and preserve locality in a strong, topological sense. Before defining the class, we need to recall some simple topology facts. Missing proof can be found in Appendix B.

## 4.1 Preliminaries

For two complexes $K$ and $K'$, a *map* $f : K \to K'$ is a function $f : V(K) \to V(K')$ such that whenever $s = \{v_0, \ldots, v_q\}$ is simplex in $K$, then $f(s) = \{f(v_0), \ldots, f(v_q)\}$ is a simplex in $K'$. The map is *color-preserving* if it preserves ids, that is, for each $v \in V(K)$, $\mathrm{ID}(f(v)) = \mathrm{id}(v)$. Hence, if the map is color preserving, and $f(s) = s'$, then $dim(s) = dim(s')$. An *edge* $e$ in a complex $K$ is an ordered pair of (not necessarily) distinct vertices $e = (u, v)$, where $\{u, v\}$ is a simplex in $K$. The *origin* and *end* of $e = (u, v)$ are respectively denoted $orig(e) = u$ and $end(e) = v$. A *path* $\alpha$ in $K$ is a finite sequence of edges $\alpha = e_1 \bullet e_2 \bullet \cdots \bullet e_k$, where $end(e_i) = orig(e_{i+1})$. The path $\alpha$ is *closed at* $u$ if $orig(\alpha) = orig(e_1) = u = end(e_k) = end(\alpha)$. A complex $K$ is *connected* if for every pair of vertices $u, v$ in $K$, there is a path from $u$ to $v$. A covering complex [38] is the discrete analogue of a covering space. The notion of covering complex formalizes the idea of one complex looking identically to another locally. Its definition is recalled below (see [38]):

**Definition 4.1** *A pair $(\tilde{K}, p)$ is a* covering complex *of a complex $K$ if and only if the following three properties are satisfied:*

1. *$p : \tilde{K} \to K$ is a map;*
2. *$\tilde{K}$ is connected;*
3. *for every simplex $s$ in $K$, $p^{-1}(s)$ is a union of pairwise disjoint simplexes, $p^{-1}(s) = \cup \tilde{s}_i$, with $p|\tilde{s}_i : \tilde{s}_i \to s$ a one-one correspondence for each $i$.*

The simplexes $\tilde{s}_i$ are called the *sheets* over $s$. We often refer to $p$ as a *covering map*. Next observations easily follow from the definition of covering complexes: $K = im(p)$, and hence, $K$ is connected. If $s$ is a $q$-simplex in $K$, each sheet $\tilde{s}_i$ over $s$ is also a $q$-simplex. For each vertex $v$ in $K$, the complex $star(v)$ consists of all simplexes that contain $v$. One can check that $K$ and $\tilde{K}$ are *locally isomorphic* in the sense that if $\tilde{v}$ is such that $p(\tilde{v}) = v$, then $star(v)$ is isomorphic to $star(\tilde{v})$.

Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ be a task. In the following, we assume that the output complex is connected (otherwise, our analysis can be done on each connected component), and that $\mathcal{O}$ does not contain irrelevant simplexes. That is, for each $t \in \mathcal{O}$, there exists $s \in \mathcal{I}$ such that $t \in \Delta(s)$. Let $p : \mathcal{O} \to \mathcal{I}$ a covering map. We say that $p$ *agrees with* $\Delta$ if $p(\tilde{s}) = s \iff \mathrm{ID}(s) = \mathrm{ID}(\tilde{s}) \wedge \tilde{s} \in \Delta(s)$, for every $s \in \mathcal{I}, \tilde{s} \in \mathcal{O}$. In particular, $p$ is color-preserving.

**Definition 4.2** *A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is* locality-preserving *if and only if there exists a covering complex $(\mathcal{O}, p)$ of $\mathcal{I}$, with a map $p$ that agrees with $\Delta$.*

Examples of locality-preserving tasks are depicted in Figures 3,5 and 6, where the color of a vertex (black or white) represents its identity. Instead of adding input and output values to vertexes, labels are added to edges, which is sufficient to specify $\Delta$, as the tasks are both monotone and projection-closed. For example, in Figure 3, the edge labeled $a$ on the left side represents an input simplex. This input simplex is mapped by $\Delta$ to a set of two output simplexes, namely the two edges labeled $a$ on the right side of the picture. In these tasks, the outputs look like the inputs, locally. For example, in Figure 3, the top left corner formed by the white vertex (with the two edges labeled $a$ and $b$) is mapped by $\Delta$ to the two opposite corners that look the same locally.

## 4.2 Characterization

Recall that task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be monotone when $\Delta(\pi(s)) \subseteq \pi(\Delta(s))$ for every $s \in \mathcal{I}$, and any projection $\pi$. In other words, for any $t' \in \Delta(\pi(s))$, there exists $t \in \Delta(s)$ such that $\pi(t) = t'$. We say that $T$ is *strongly monotone* if, for every $s \in \mathcal{I}$, any projection $\pi$, and any $t' \in \Delta(\pi(s))$, there exists a *unique* $t \in \Delta(s)$ such that $\pi(t) = t'$. So, intuitively, a task that is strongly monotone permits to extend any partial output in a unique manner. In order to characterize locality-preserving, we say that a task $T$ is *one-to-one* if and only if, for every pair $(s, s'), s \neq s'$ of 0-dimensional simplexes in $\mathcal{I}$, we have $\Delta(s) \cap \Delta(s') = \emptyset$.

**Theorem 4.3** *A task $T$ is locality-preserving if and only if $T$ is projection-closed, one-to-one, and strongly monotone.*

As every projection-closed task is wait-free checkable (Theorem 3.3), we get the following.

**Corollary 4.4** *Every locality-preserving task is wait-free checkable.*

As demonstrated by the next corollary, few locality-preserving tasks are wait-free solvable. For an input complex $\mathcal{I}$, the *identity task* $T_{Id,\mathcal{I}} = (\mathcal{I}, \mathcal{I}, \Delta)$ over $\mathcal{I}$ simply requires that each process decides its input in every execution: $\forall s \in \mathcal{I}, \Delta(s) = \{s\}$. More generally, we say that a task is an identity task by having each process output a function of its input, without any shared memory operations. Identity tasks with input $\mathcal{I}$ are the tasks equivalent to $T_{Id,\mathcal{I}}$ for the $\sim$ relation.

**Corollary 4.5** *The identity tasks are the only locality-preserving task that are wait-free solvable.*

## 4.3 Hierarchies of locality-preserving tasks

In this section we classify the locality-preserving tasks in term of their relative computing power, that is in their capacity of mutual implementation (Definition 3.5). For the remaining of this section, we fix an arbitrary input complex $\mathcal{I}$ and study the relative power of locality-preserving tasks with input $\mathcal{I}$. We establish that each such locality-preserving task induces subgroups of a group defined from the closed paths in $\mathcal{I}$. Moreover, the relative power of locality-preserving tasks with input $\mathcal{I}$ directly depends on the subgroups they induce.

### 4.3.1 Edgepath groups and locality-preserving tasks

Our exposition follows Rotman [38]. Let $K$ a complex and $v_* \in V(K)$. The *edgepath group* of $K$ with basepoint $v_*$ is $G(K, v_*) = \{[\alpha] \ : \ \alpha \text{ is a closed path at } v_*\}$. $[\alpha]$ consists of the equivalence class of closed paths $\alpha'$ that can be deformed to $\alpha$ along 2-simplexes. More precisely, the paths $\alpha'$ and $\alpha$ are equivalent if one can be obtained from the other by applying the following rule a finite number of times: replace $(u, v) \bullet (v, w)$ by $(v, w)$ or $(v, w)$ by $(u, v) \bullet (v, w)$ whenever $\{u, v, w\}$ is a simplex of $K$. The group operation is path concatenation, which is compatible with path equivalence. Given $\alpha = e_1 \bullet \ldots \bullet e_n$, the inverse of $[\alpha]$ is $[\alpha^{-1}]$ where $\alpha^{-1} = e_n^{-1} \bullet \ldots \bullet e_1^{-1}$ and each $e_i^{-1}$ is the edge obtained by reversing the end and origin of $e_i$. The identity element is $[(v_*, v_*)]$. If $K$ is connected, changing the basepoint results in an isomorphic group ([38], Theorem 1.4).

Covering complexes of $K$ induce subgroups of the edgepath group of $K$. Conversely, every subgroup of the edgepath group is induced by a covering complex. Formally, the notation of a covering complex is extended to *pointed complexes*. $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ is a covering complex when $(\tilde{K}, p)$ is a covering complex of $K$, and $p(\tilde{v}_*) = v_*$. Each covering complex $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ determines a subgroup of $G(K, v_*)$, namely, $p_\#(G(\tilde{K}, \tilde{v}_*))$, where $p_\#$ is the homomorphism induced by $p$, which is one-to-one ([38], Theorem 2.3). Let $H = p_\#(G(\tilde{K}, \tilde{v}_*))$. Keeping $p$ unchanged but choosing a different basepoint $\tilde{v}'_* \in \tilde{K}$ such that $p(\tilde{v}'_*) = p(\tilde{v}_*) = v_*$ may induce a different subgroup $H' = p_\#(G(\tilde{K}, \tilde{v}'_*))$. $H$ and $H'$ are however *conjugate* subgroups of $G(K, v_*)$. Conversely, if $H'$ is conjugate to $H$ then $H' = p_\#(G(\tilde{K}, \tilde{u}))$ for some $\tilde{u} \in V(\tilde{K})$ ([38], Theorem 2.4). Finally, let $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ and $q : (\tilde{J}, \tilde{w}_*) \to (K, v_*)$ two covering complexes of $(K, v_*)$

and $H(= p_{\#}(G(\tilde{K}, \tilde{v}_*))), H'(= q_{\#}(G(\tilde{J}, \tilde{w}_*)))$ the induced subgroups. If $H' \subseteq H$, there exists a unique map $r : (\tilde{J}, \tilde{w}_*) \to (\tilde{K}, \tilde{v}_*)$ such that $pr = q$. Moreover, $r : (\tilde{J}, \tilde{w}_*) \to (\tilde{K}, \tilde{v}_*)$ is a covering complex ([38], Theorem 3.3). In particular, when $H = H'$, $r$ is an isomorphism.

On the other hand, for every subgroup $H$ of $G(K, v_*)$ there exists a connected complex $K_H$ and a map $p$ such that $p : (K_H, \tilde{v}_*) \to (K, v_*)$ is a covering complex for some $\tilde{v}_* \in V(K_H)$ and $p_{\#}(K_H, \tilde{v}_*) = H$ ([38], Theorem 2.8). In particular, the trivial group $\{[(v_*, v_*)]\}$ that consists in the identity element is a subgroup of any subgroup, and the corresponding covering complex $p : (K_u, \tilde{v}_*) \to (K, v_*)$ is called the *universal covering* of $K$, because it covers any other covering of $K$. For example, Figure 6b depicts the universal covering for the complex on the left. A simplex of the complex on the right is sent to the simplex with the same label in the left complex by the universal covering map. The universal covering complex is *simply connected* since $p_{\#}$ is one-to-one and $p_{\#}(G(K_u, \tilde{v}_*)) = \{1\}$.

By definition, each locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ determines a covering complex $p : \mathcal{O} \to \mathcal{I}$ and conversely, a covering complex $p : \mathcal{O} \to \mathcal{I}$ defines a locality-preserving task with input $\mathcal{I}$. It thus follows from the discussion above that every locality-preserving task with input $\mathcal{I}$ induces subgroups of the edgepath group of $\mathcal{I}$, and reciprocally each subgroup induces a locality-preserving task. This is captured by the next lemma.

**Lemma 4.6** *Let $\mathcal{I}$ be a connected input complex and $v_* \in V(\mathcal{I})$. (1) Every subgroup $H \subseteq G(\mathcal{I}, v_*)$ induces a locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ whose associated covering map satisfies $p_{\#}(G(\mathcal{O}, \tilde{v}_*)) = H$ for some $\tilde{v}_* \in V(\mathcal{O})$. (2) Every locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ induces a conjugacy class of subgroups of $G(\mathcal{I}, v_*)$; each subgroup $H$ in the class satisfies $H = p_{\#}(G(\mathcal{O}, \tilde{v}_*))$ for some $\tilde{v}_* \in V(\mathcal{O})$, where $p$ is the covering map associated with $T$.*

Then, Theorem 3.3 in [38] discussed earlier implies the following result.

**Lemma 4.7** *Let $\mathcal{I}$ be an input complex and $v_* \in V(\mathcal{I})$. Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ and $T' = (\mathcal{I}, \mathcal{O}', \Delta')$ two locality-preserving tasks with input $\mathcal{I}$ and $p : \mathcal{O} \to \mathcal{I}$, $p' : \mathcal{O}' \to \mathcal{I}$ their respective covering map. If $p'_{\#}G(\mathcal{O}', \tilde{v}'_*) \subseteq p_{\#}(G(\mathcal{O}, \tilde{v}_*))$ for some vertexes $\tilde{v}'_* \in V(\mathcal{O}')$, $\tilde{v}_* \in V(\mathcal{O})$, there exists a covering complex $p'' : (\mathcal{O}', \tilde{v}'_*) \to (\mathcal{O}, v_*)$ satisfying $p' = pp''$.*

### 4.3.2 Group-based hierarchies of locality-preserving tasks

We consider locality-preserving tasks that can be defined over a given input complex $\mathcal{I}$. The following result explicits the existence of a hierarchy of locality-preserving tasks, using the implementation relation "$\leq$" of Definition 3.5. By Lemma 4.6(2), every locality-preserving task induces a conjugacy class.

**Theorem 4.8** *Let $\mathcal{I}$ be a connected complex. Let $T_{\mathcal{K}} = (\mathcal{I}, \mathcal{K}, \Delta_{\mathcal{K}})$ and $T_{\mathcal{L}} = (\mathcal{I}, \mathcal{L}, \Delta_{\mathcal{L}})$ be two locality-preserving tasks with input complex $\mathcal{I}$. Let $v_* \in V(\mathcal{I})$ and $C_{\mathcal{K}}$ and $C_{\mathcal{L}}$ the conjugacy classes of subgroups of $G(\mathcal{I}, v_*)$ induced by $T_{\mathcal{K}}$ and $T_{\mathcal{L}}$ respectively. $T_{\mathcal{L}} \leq T_{\mathcal{K}}$ if and only if $\exists H_{\mathcal{K}} \in C_{\mathcal{K}}, H_{\mathcal{L}} \in C_{\mathcal{L}}$ such that $H_{\mathcal{L}} \supseteq H_{\mathcal{K}}$.*

We say that a task is *universal* for some set of tasks if it implements any task in that set (in the sense of "$\leq$" in Definition 3.5). Theorem 4.8 implies that the set of locality-preserving tasks with input complex $\mathcal{I}$ has a universal task, which is the locality-preserving task defined by the universal covering of $\mathcal{I}$. More generally, it shows that every locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ lies in between the trivial task $(\mathcal{I}, \mathcal{O}, \Delta)$ and the task defined by the universal covering complex of $\mathcal{I}$. This is illustrated in Figure 7, where the double cycle at the bottom represents the input complex $\mathcal{I}$ and the universal covering at the top defines a universal locality-preserving task $T$ from the covering map $p$. The covering map $p''$ on the left defines another locality-preserving task $T''$ with input $\mathcal{I}$. The trivial subgroup is induced by $T$, which is a subgroup of every subgroup induced by $T''$. Therefore, we have $T'' \leq T$. The complex at the top is also a covering of the complex on the left, with covering map $p'$, which in turn defines another locality-preserving task $T'$.

**Acknowledgments.** We thank Marcelo Aguilar and Javier Bracho for helpful discussions on covering spaces.

# References


[1] Angluin D., Local and Global Properties in Networks of Processors (Extended Abstract). *12th Annual ACM Symposium on Theory of Computing* (STOC), pp. 28–30, 1980.

[2] Awerbuch B., Patt-Shamir B. and Varghese G., Self-stabilization by Local Checking and Correction. *32nd Annual Symposium on Foundations of Computer Science* (FOCS), pp. 268–277, 1991.

[3] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuck R., Renaming in Asynchronous Environment. *Journal of the ACM*, 37(3): 524–548, 1990.

[4] Attiya H. and Rajsbaum S., The Combinatorial Structure of Wait-free Solvable Tasks. *SIAM Journal of Computing,* 31(4): 1286–1313 (2002).

[5] Awerbuch B. and Varghese G., Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. *32nd Annual Symposium on Foundations of Computer Science* (FOCS), pp. 258–267, 1991.

[6] Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations, and Advanced Topics, *Wiley*, 2004.

[7] L. Barenboim and M. Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *29th ACM Symp. on Principles of Distributed Computing* (PODC), pp 410-419, 2010.

Computations.

[8] Blum M. and Kannan S., Designing Programs that Check Their Work. *J. ACM* 42(1): 269–291 (1995).

[9] Blum M., Luby M. and Rubinfeld R., Self-Testing/Correcting with Applications to Numerical Problems. *J. Comput. Syst. Sci.* 47(3): 549–595 (1993)

[10] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.* 105(1): 132-158, 1993.

[11] Chalopin J., Godard E. and Métivier Y., Local Terminations and Distributed Computability in Anonymous Networks. *22nd International Symposium on Distributed Computing* (DISC), pp. 47–62, 2008.

[12] Chalopin J. and Métivier Y., On the Power of Synchronization Between two Adjacent Processes. *Distributed Computing* 23(3): 177-196 (2010).

[13] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking *Information and Control* 70(1): 3253 (1986).

[14] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed Verification and Hardness of Distributed Approximation *43rd ACM Symp. on Theory of Computing* (STOC), 2011.

[15] Dolev D., Lynch N., Pinter S., Stark E. and Weihl W., Reaching Approximate Agreement in the Presence of Faults. *J. ACM* 33(3):499–516 (1986).

[16] Ergün F., Kannan S., Kumar R., Rubinfeld R. and Viswanathan M., Spot-Checkers. *J. Comput. Syst. Sci.* 60(3): 717-751 (2000).

[17] Fischer M., Lynch N. and Paterson M., Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[18] P. Fraigniaud, D. Ilcinkas, and A. Pelc. Communication algorithms with advice. *J. Comput. Syst. Sci.* 76(3-4): 222-232 (2010).

[19] P. Fraigniaud, A. Korman and D. Peleg. Local Distributed Decision. *arXiv:1011.2152*.

[20] Franklin M., Garay J.A. and Yung M., Self-Testing/Correcting Protocols. *13th International Symposium on Distributed Computing* (DISC), pp. 269–284, 1999.

[21] Freund Y., Kearns M., Ron D., Rubinfeld R., Schapire R. and Sellie L., Efficient Learning of Typical Finite Automata from Random Walks. *Inf. Comput.* 138(1): 23–48 (1997).

[22] Gafni E. and Koutsoupias E., Three-Processor Tasks Are Undecidable. *SIAM J. Comput.* 28(3): 970–983 (1999).

[23] Goldwasser S., Gutfreund D., Healy A., Kaufman T. and Rothblum G., A (De)constructive Approach to Program Checking. *40th Annual ACM Symposium on Theory of Computing* (STOC), pp. 143–152, 2008.

[24] Goldreich O., Goldwasser S. and Shari and Ron D., Property Testing and its Connection to Learning and Approximation. *J. ACM* 45(4): 653–750 (1998).

[25] M. Goos and J. Suomela. Locally checkable proofs. *30th ACM Symp. on Principles of Distributed Computing*, 2011.

[26] Herlihy M. and Rajsbaum S., The Decidability of Distributed Decision Tasks (Extended Abstract). *29th Annual ACM Symposium on the Theory of Computing* (STOC), pp. 589–598, 1997.

[27] Herlihy M. and Rajsbaum S., A Classification of Wait-Free Loop Agreement Tasks. *Theor. Comput. Sci.*, 291(1): 55–77, 2003.

[28] Herlihy M. and Shavit N., The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, 1999.

[29] A. Korman, S. Kutten and D. Peleg. Proof Labeling Schemes. *Distributed Computing* **22**, (2010), 215–233.

[30] A. Korman, J.-S. Sereni, and L. Viennot. Toward more Localized Local Algorithms: Removing Assumptions concerning Global Knowledge. *30th ACM Symp. on Principles of Distributed Computing* (PODC), 2011.

[31] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. *25th ACM Symp. on Principles of Distributed Computing* (PODC), pp 7-15, 2006.

[32] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[33] Lipton R., New Directions in Testing. *DIMACS workshop on distributed computing and cryptography*, 2:191–202, 1991.

[34] Lynch N., *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[35] Mazurkiewicz A., Distributed Enumeration. *Inf. Process. Lett.* 61, 233–239 (1997).

[36] M. Naor and L. Stockmeyer, What can be computed locally? *SIAM J. Comput.* 24(6): 1259–1277 (1995).

[37] D. Peleg. Distributed Computing: A Locality-Sensitive Approach. *SIAM*, 2000.

[38] Rotman J., Covering Complexes with Applications to Algebra. *Rocky Mountain J. of Mathematics,* 3(4): 641–674 (1973).

[39] Rubinfeld R., Designing Checkers for Programs that Run in Parallel. *Algorithmica* 15(4): 287–301 (1996).

# A Missing proofs of section 3

**Proof of Theorem 3.3** Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ be a task, and assume that $T$ is projection-closed. We describe a wait-free algorithm that checks $T$. The algorithm starts with input $s \times t \in \mathcal{I} \times \mathcal{O}$ and decides whether $t \in \Delta(s)$. More precisely, each participating process $i \in J = \text{ID}(s)$ is given a pair $(s_i, t_i)$, and the algorithm proceeds at each process $i$ in three atomic steps:

1. write $(i, (s_i, t_i))$ in memory;
2. read memory, to get $s' \times t'$;
3. if $t' \in \Delta(s')$ then output "yes", otherwise output "no".

Before proving the correctness of this algorithm, notice that the simplex $s' \times t'$ obtained by process $i$ after the read instruction is a subset of $s \times t$; namely the union of all the inputs of processes which wrote in memory before the read of process $i$ occurs. So, $s' \times t' = \pi(s) \times \pi(t)$ for some projection $\pi$. Also, observe that the third instruction of the algorithm is computable: process $i$ can decide whether $t' \in \Delta(s')$ as we are dealing solely with input-output relations $\Delta$ that are sequentially computable.

We now prove the correctness of the algorithm. Assume first that $t \in \Delta(s)$. Then

$$t' = \pi(t) \in \pi(\Delta(s)) \subseteq \Delta(\pi(s)) = \Delta(s')$$

where the inclusion is due to the fact that $T$ is projection-closed. Therefore, process $i$ outputs "yes". Hence, all participating processes which decide output "yes", that is, the global output of the set of non-faulty participating processes belongs to $\overline{Y}[J]$, as desired. Assume now that $t \notin \Delta(s)$. If $k < |J|$ processes decide, then whatever are the outputs of these processes, the corresponding output simplex is in $S^J \setminus Y[J]$, as desired. If all the processes in $J$ decide, that is if all participating processes experienced no crashes, then let $i$ be the identity of the process that reads memory the last among all participating processes. Process $i$ reads the entire input $(s, t)$, that is $(s', t') = (s, t)$. Therefore, this process detects that $s' \notin \Delta(t')$, and thus it outputs "no", as desired. Therefore, a projection-closed task is wait-free checkable.

Let us now consider a wait-free checkable task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, and assume, for the purpose of contradiction, that $T$ is not projection-closed. So, there must exist a simplex $s \in \mathcal{I}$, and a projection $\pi$ such that $\pi(\Delta(s)) \not\subseteq \Delta(\pi(s))$. Hence, let $t' \in \pi(\Delta(s)) \setminus \Delta(\pi(s))$, and let $t \in \Delta(s)$ such that $\pi(t) = t'$. Let $s' = \pi(s)$. We have $s' \times t' = \pi(s \times t)$ in the complex $\mathcal{I} \times \mathcal{O}$. Let $J = \text{ID}(s \times t)$, and $J' = \text{ID}(s' \times t')$. On the one hand, we have

$$\Delta_c(\pi(s \times t)) = \Delta_c(s' \times t') = S^{J'} \setminus Y[J']$$

because $t' \notin \Delta(s')$. On the other hand, we have

$$\Delta_c(s \times t) = \overline{Y}[J]$$

because $t \in \Delta(s)$. Therefore, $\Delta_c(\pi(s \times t)) \not\subseteq \Delta_c(s \times t)$, and thus $\Delta_c(\pi(s \times t)) \not\subseteq \pi(\Delta_c(s \times t))$. We do not necessarily have $\Delta_c(\pi(s \times t)) \cap \pi(\Delta_c(s \times t)) = \emptyset$ though, because $S^{J'} \setminus Y[J']$ contains $(|J'| - 2)$-dimensional simplexes with values 1. Nevertheless, if one restricts our consideration to the skeleton $Q$ of $S^{J'} \setminus Y[J']$ consisting of all its simplexes of dimension $|J'| - 1$, we do have $Q \cap \overline{Y}[J] = \emptyset$. Thus, by a reasoning identical to the one showing that the intersection-closeness property is a necessary condition for wait-free solvability, we get that $T_c$ is not wait-free solvable, which contradicts the fact that $T$ is wait-free checkable. □

**Proof of Theorem 3.4** We first show that, for every task $T$, there exists a task $T'$ that is wait-free checkable and $T' \sim T$. Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ be a task. We construct a wait-free checkable task $T' = (\mathcal{I}', \mathcal{O}', \Delta')$ and show that $T \sim T'$. The construction proceeds in two phases, the *completion* phase and the *projection* phase.

The completion phase proceeds as follows. We constructs two intermediate sets $\mathcal{I}_{med}$, $\mathcal{O}_{med}$ and a relation $\Delta_{med} \subseteq \mathcal{I}_{med} \times \mathcal{O}_{med}$. Let $V$ a set of values such that each element of $v$ is not a valid input for $T$ and $|V| = |\mathcal{I}|$.

More formally, for each $s \in \mathcal{I}$ and each $(i, x_i) \in s$, $x_i \notin V$. Let $v^* : \mathcal{I} \to V$ a one-to-one function. For $s \in \mathcal{I}$ (resp. $\mathcal{O}$) and $v \in V$, let $c(s, v) = s \cup \{(i, v), i \in [n] \setminus \mathrm{ID}(s)\}$ the set obtained by replacing each missing entry of $s$ with the value $v$. We first "complete" each input set $s \in \mathcal{I}$ with the value $v^*(s)$. Two distinct input sets $s, s' \in \mathcal{I}$ are thus completed with distinct values $v^*(s) \neq v^*(s')$. Let $\mathcal{I}_{med}$ be the set of input sets completed in this way, i.e.,

$$\mathcal{I}_{med} = \{c(s, v^*(s)), s \in \mathcal{I}\}.$$

Output sets are completed in a similar way. Given an output set $t \in \mathcal{O}$, multiple completed sets may however be formed. More precisely, for each input set $s \in \mathcal{I}$ such that $t \in \Delta(s)$, $\mathcal{O}_{med}$ includes the completed set $c(t, v^*(s))$ whenever $t$ is a valid output for input $s$ and $\mathrm{ID}(s) = \mathrm{ID}(t)$:

$$\mathcal{O}_{med} = \{c(t, v^*(s)), t \in \Delta(s) \text{ and } \mathrm{ID}(s) = \mathrm{ID}(t)\}.$$

Let $\sigma, \tau$ two completed input and output sets respectively. We set $\tau \in \Delta_{med}(\sigma)$ if and only if their respective antecedent $s$ and $t$ before completion form an input-output pair for $T$. That is, for every $\sigma \times \tau \in \mathcal{I}_{med} \times \mathcal{O}_{med}$ we set $\tau \in \Delta_{med}(\sigma)$ if and only if:

$$\exists s \times t \in \mathcal{I} \times \mathcal{O} \text{ with } t \in \Delta(s) \text{ and } \mathrm{ID}(s) = \mathrm{ID}(t) \text{ such that } \sigma \times \tau = c(s, v^*(s)) \times c(t, v^*(s)).$$

We now describe the projection phase. The task $(\mathcal{I}_{med}, \mathcal{O}_{med}, \Delta_{med})$ defined so far is clearly not wait-free solvable as it is even not defined when not every process participates. For each input set $\sigma \in \mathcal{I}_{med}$ (resp. $\in \mathcal{O}_{med}$), and each projection map $\pi$, the set $\pi(\sigma)$ is included in $\mathcal{I}'$ (resp. $\mathcal{O}'$):

$$\mathcal{I}' = \bigcup_{\pi} \pi(\mathcal{I}_{med}) \text{ and } \mathcal{O}' = \bigcup_{\pi} \pi(\mathcal{O}_{med})$$

Similarly, $\Delta'$ is the union over every projection map of the relation $\Delta_{med}$. For every $s' \in \mathcal{I}'$, we define

$$\Delta'(s') = \bigcup_{\pi, \sigma : \pi(\sigma) \subseteq s'} \pi(\Delta_{med}(\sigma)).$$

Let us show that $T' = (\mathcal{I}', \mathcal{O}', \Delta')$ is wait-free checkable. Let $s \in \mathcal{I}'$ and $\mu$ a projection. By definition of $\Delta'$, we get

$$\mu(\Delta'(s)) = \bigcup_{\pi, \sigma : \pi(\sigma) \subseteq s} \mu(\pi(\Delta_{med}(\sigma))) = \bigcup_{\pi', \sigma : \pi'(\sigma) \subseteq \mu(s)} \pi'(\Delta_{med}(\sigma)) = \Delta'(\mu(s)).$$

Therefore, $T'$ is projection-closed, and thus it is wait-free checkable in view of Theorem 3.3.

We now prove that $T \sim T'$. Let us first prove that $T' \preceq T$. Suppose that a black box $\mathcal{A}$ that solves $T$ is given. We describe a wait-free algorithm $\mathcal{A}'$ to solve $T'$. A process $p_i$ that wakes up with input $x_i$ first checks whether there exists an input set $s \in \mathcal{I}$ such that $(i, x_i) \in s$. If such an input set is found, $p_i$ invokes $\mathcal{A}$ with input $x_i$ and decides the value returned. Otherwise, $p_i$ decides $x_i$. Consider an execution of $\mathcal{A}'$ with input $s' \in \mathcal{I}'$ and output $t'$. $s'$ is the projection of a set $s \in \mathcal{I}$ after it has been completed, i.e., $s' = \pi(c(s, v^*(s)))$ where $\pi$ is a projection map. As the value $v^*(s)$ is not included in any input set of $\mathcal{I}$, the inputs of the processes that invoke algorithm $\mathcal{A}$ form a set $s_1 \subseteq s$. Each other process fails or has $v^*(s)$ as input. As $\mathcal{I}$ is closed under containment, $s_1 \in \mathcal{I}$. Therefore, as $\mathcal{A}$ solves $T$, the outputs of the processes that invoke $\mathcal{A}$ form a set $t_1 \in \Delta(s_1)$. By the monotony condition, there exits an output set $t \supseteq t_1$ such that $t \in \Delta(s)$ and $\mathrm{ID}(s) = \mathrm{ID}(t)$. Therefore, in the completing phase of the construction, $\sigma \times \tau$, with $\sigma = c(s, v^*(s)), \tau = c(t, v^*(s))$ is added as an input-output pair for $\Delta_{med}$. Let $p_i$ be a process that decides and $y_i$ its decided value. If $p_i$ decides after invoking $\mathcal{A}$, $(i, y_i) \in t$. Otherwise, $p_i$ decides $v^*(s)$ and $(i, y_i) \notin t$. Therefore, $t' = \mu(\tau)$ for some projection map $\mu$. Recall that we also have $s' = \pi(\sigma)$, where $\pi$ is a projection map. Clearly, $\mathrm{ID}(t') \subseteq \mathrm{ID}(s')$, and thus $\mu(\sigma) \subseteq s'$. To summarize, $\tau \in \Delta_{med}(\sigma)$ and hence $t' = \mu(\tau) \in \mu(\Delta_{med}(\sigma))$. As $\mu(\sigma) \subseteq s'$, it follows from the definition of $\Delta'$ that $t' \in \Delta'(s')$.

We now prove that $T \preceq T'$. Suppose now that we are given a black box algorithm $\mathcal{A}'$ that solves $T'$. We show that $\mathcal{A}'$ also solves $T$. Fix an execution $e$ of $\mathcal{A}'$ with input $s \in \mathcal{I}$. Let $t \in \Delta'(s)$ the output set of this execution. Let $e_1$ a finite prefix of $e$ in which every process that decides in $e$ has decided. Note that the identities of these processes is $\mathrm{ID}(t)$. Execution $e_2$ extends $e_1$. The input set of $e_2$ is $\sigma = c(s, v^*(s))$: after each process $p_i, i \in \mathrm{ID}(t)$ has decided, every process $p_j, j \in [n] \setminus \mathrm{ID}(t)$ wakes up with input $v^*(s)$. Moreover, every process decides in $e_2$. Let $\tau \in \Delta'(\sigma)$ the output set. $e_2$ is a legal extension of $e_1$, as processes may be arbitrarily slow. Also, when the processes $p_j, j \in \mathrm{ID}(t)$ decide in $e_2$, the execution is indistinguishable from $e$, which implies that $\tau \supseteq t$. As $\tau \in \Delta'(\sigma)$ and $\mathrm{ID}(\sigma) = \mathrm{ID}(\tau) = [n]$, $\sigma \times \tau$ is an input-output pair of $\Delta_{med}$. Therefore, $\tau$ is the completion of some set $t' \in \mathcal{O}$ with value $v^*(s)$ ($\tau = c(t', v^*(s))$) and we have $\mathrm{ID}(t') = \mathrm{ID}(s)$ and $t' \in \Delta(s)$. To conclude, we note that $t' \supseteq t$. Since $t \in \Delta(s)$, this implies that $t' \in \Delta(s)$.

This completes the proof of the fact that for every task, there exists an equivalent task that is wait-free checkable. Since determining whether a task is wait-free solvable is undecidable [22, 26], the theorem follows. □

# B Missing proofs of Section 4

**Proof of Theorem 4.3** Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ be a projection-closed, one-to-one and strongly monotone task. We define a map $p : \mathcal{O} \to \mathcal{I}$ and show that $p$ is a covering map. Let $\tilde{v} \in V(\mathcal{O})$. As we assume that $\mathcal{O}$ has no irrelevant simplex, there exists $v \in V(\mathcal{I})$ such that $\{\tilde{v}\} \in \Delta(\{v\})$. Moreover, $v$ is unique. Indeed, if $\{\tilde{v}\} \in \Delta(\{v'\})$ for some $v' \in V(\mathcal{I})$, we get $\Delta(\{v\}) \cap \Delta(\{v'\}) \supseteq \{\tilde{v}\} \neq \emptyset$, which contradicts the hypothesis that $T$ is one-to-one. Therefore, we define for each $\tilde{v} \in \mathcal{O}$ $p(\tilde{v}) = v$, where $v$ is the unique vertex in $V(\mathcal{I})$ such that $\{\tilde{v}\} \in \Delta(\{v\})$.

To see that $p$ is a map, let $\tilde{s} = \{\tilde{v}_1, \ldots, \tilde{v}_m\} \in \mathcal{O}$. By the no irrelevant output simplex assumption, $\tilde{s} \in \Delta(s)$ for some simplex $s = \{v_1, \ldots, v_m\} \in \mathcal{I}$. Since task $T$ is projection-closed, $\{\tilde{v}_i\} \in \Delta(\{v_i\})$. We know however that for every $\tilde{v} \in V(\mathcal{O})$, there is a unique $v = p(\tilde{v}) \in \mathcal{I}$ such that $\{\tilde{v}\} \in \Delta(\{v\})$. Therefore, $p(\tilde{s}) = \{p(\tilde{v}_1), \ldots, p(\tilde{v}_m)\} = s \in \mathcal{I}$ and $p$ is a map. This also shows that $p$ agrees with $\Delta$, since $\tilde{s} \in \Delta(p(\tilde{s}))$. This establishes that $p$ satisfies Definition 4.1(1). As $\mathcal{O}$ is connected by assumption, Definition 4.1(2) is satisfied as well.

If $s = p(\tilde{s})$ for some simplex $\tilde{s} \in \mathcal{O}$, clearly $p|\tilde{s} : \tilde{s} \to s$ is a one-one correspondence because $p$, agreeing with $\Delta$, preserves colors. To prove that the inverse $p^{-1}(s)$ is a union of pairwise disjoint simplexes, let $\tilde{s}_1, \tilde{s}_2$ two simplexes $\in p^{-1}(s)$. Assume that $\tilde{s}_1$ and $\tilde{s}_2$ have one vertex $\tilde{v}$ in common. Let $v = p(\tilde{v})$ and $\pi$ a projection such that $\pi(s) = \{v\}$. By definition of $p$, $\{\tilde{v}\} \in \Delta(\{v\})$. Because $T$ is strongly monotone, there exists a unique $\tilde{s} \in \Delta(s)$ such that $\pi(\tilde{s}) = \{\tilde{v}\}$. As $p$ agrees with $\Delta$, and $p(\tilde{s_1}) = s$, $\tilde{s_1} \in \Delta(s)$. Hence, as $\pi(\tilde{s}_1) = \{\tilde{v}\}$, $\tilde{s}_1 = \tilde{s}$. Similarly, $\tilde{s}_2 = \tilde{s}$. Therefore $\tilde{s}_2 = \tilde{s}_1$ and we conclude that $p^{-1}(s)$ is a disjoint union of pairwise simplexes, as required by Definition 4.1(3).

Reciprocally, assume $T$ is locality-preserving. Let $p : \mathcal{O} \to \mathcal{I}$ be a covering map that agrees with $\Delta$. The fact that $T$ is one-to-one, strongly monotone and projection-closed mechanically follows from the properties of the covering map $p$. Let $t \in \mathcal{O}$ and $s \in \mathcal{I}$.

- Projection-closed. Let $\pi$ be a projection map, and suppose that $t \in \Delta(s)$. As $p$ agrees with $\Delta$, $p(t) = s$, and thus $p(\pi(t))) = \pi(s)$. Using again the fact that $p$ agrees with $\Delta$, we have $\pi(t) \in \Delta(\pi(s))$. Therefore $\pi(\Delta(s)) \subseteq \Delta(\pi(s))$.

- $T$ is one-to-one. Let $v \in V(\mathcal{I})$. The fact that $p$ agrees with $\Delta$ implies that $\Delta(\{v\}) = p^{-1}(v)$. Therefore, for every $v \neq v' \in V(\mathcal{I}), \Delta(\{v\}) \cap \Delta(\{v'\}) = \emptyset$.

- Strong monotony. Let $t, t_1, t_2 \in \mathcal{O}$ and $\pi$ a projection satisfying $t = \pi(t_1) = \pi(t_2)$. Assume that $t \in \Delta(\pi(s))$ and $t_1, t_2 \in \Delta(s)$. As $p$ agrees with $\Delta$, $p(t_1) = p(t_2) = s$. Therefore, as $p^{-1}(s)$ is a disjoint union of simplexes, and $t_1 \cap t_2 \neq \emptyset$, $t_1 = t_2$. So there exists a unique $t', \pi(t') = t$ and $t' \in \Delta(s)$.

The fact that $T$ satisfies the above three properties completes the proof. □

**Proof of Corollary 4.5** Let $T$ be a wait-free solvable and locality-preserving task, with covering map $p$. By the Asynchronous Computability Theorem [28], there exists subdivision $\mathcal{I}^*$ of $\mathcal{I}$ and a color preserving map $\delta : \mathcal{I}^* \to \mathcal{O}$ that agrees with $\Delta$. More precisely, a simplex $s^* \in \mathcal{I}^*$ is obtained by subdividing a simplex $s \in \mathcal{I}$ with $\mathrm{ID}(s) = \mathrm{ID}(s^*)$, called the *carrier* of $s^*$. $\delta$ agrees with $\Delta$ means that for every $s^* \in \mathcal{I}^*$, $\delta(s^*) \in \Delta(carrier(s^*))$.

We show that every simplex in $\mathcal{I}^*$ with the same carrier are mapped by $\delta$ to the same simplex. Let $s^*, s^{*\prime} \in \mathcal{I}^*$ two intersecting simplexes with the same carrier $s$. We have $\delta(s^*) \cap \delta(s^{*\prime}) \neq \emptyset$. As $\delta$ agrees with $\Delta$, and $s^*, s^{*\prime}$ have the same carrier $s$, $\delta(s^*)$ and $\delta(s^{*\prime}) \in \Delta(s)$. Moreover, the set of simplexes $t \in \Delta(s)$ with $\mathrm{ID}(t) = \mathrm{ID}(s)$ coincides with $p^{-1}(s)$. This is because the covering map $p$ of $T$ agrees with $\Delta$. By definition 4.1(3), $p^{-1}(s)$ is a disjoint union of simplexes. Therefore, $\delta(s^*) = \delta(s^{*\prime})$ since they intersect. Two non-intersecting simplexes $s^*$ and $s^{*\prime}$ with the same carrier $s$ are connected by a chain $s_1^*(= s^*), \ldots, s_m^*(= s^{*\prime})$ where $carrier(s_i^*) = s$ and every two consecutive simplexes intersect. By repeating the same reasoning over the chain, we obtain also that $\delta(s^*) = \delta(s^{*\prime})$.

Consequently, we can consider without loss of generality that the subdivision is trivial and $\delta$ is thus a map from $\mathcal{I}$ to $\mathcal{O}$ that agrees with $\Delta$: for each $s \in \mathcal{I}, \delta(s) \in \Delta(s)$. Therefore, the trivial algorithm in which each process decides the value return by applying $\delta$ to its output solves $T$, which proves that $T$ is an identity task. □

$$\begin{tikzcd}
\mathcal{K} \arrow[dd, "p"'] \arrow[dr, "r"] & \\
& \mathcal{L} \arrow[dl, "q"] \\
\mathcal{I} &
\end{tikzcd}$$

**Proof of Theorem 4.8** We denote $p : \mathcal{K} \to \mathcal{I}$ (resp. $q : \mathcal{L} \to \mathcal{I}$) the covering map corresponding to $T_\mathcal{K}$ (resp. $T_\mathcal{L}$).

Let us assume that $H_\mathcal{K} = p_\#(G(\mathcal{K}, \tilde{u})) \subseteq q_\#(G(\mathcal{L}, \tilde{v})) = H_\mathcal{L}$ for some vertexes $\tilde{u} \in V(\mathcal{K}), \tilde{v} \in V(\mathcal{L})$. We show that $T_L \preceq T_K$. By applying Lemma 4.7, we get that there exists a covering map $r : \mathcal{K} \to \mathcal{L}$ such that $qr = p$. Given a black box $\mathcal{A}_\mathcal{K}$ that solves $T_\mathcal{K}$, $T_\mathcal{L}$ can be solved by the following protocol. Each process $p_i$ first invokes $\mathcal{A}_\mathcal{K}$ on its input and gets back an output $o_i$. $p_i$ then decides $r(o_i)$. Fix an execution of the protocol. Let $s_1, s_2$ and $s_3$ the simplexes formed by the initial inputs of the process, the outputs resulting of the invocation of $\mathcal{A}_\mathcal{K}$ and the decision values respectively. As $\mathcal{A}_\mathcal{K}$ solves the locality-preserving task $T_\mathcal{K}$, $p(s_2) = s_1$. Note that we also have $s_3 = r(s_2)$, and hence $q(s_3) = q(r(s_2))$ . Therefore, as $qr = p$, $q(s_3) = p(s_2) = s_1$. By definition of locality-preserving tasks, $q$ agrees with $\Delta_\mathcal{L}$, from which we conclude that the output simplex in the execution $s_3 \in \Delta_\mathcal{L}(s_1)$ as desired.

Reciprocally, let us assume that $T_\mathcal{L} \leq T_\mathcal{K}$. Thus, there is a wait-free protocol that solves $T_\mathcal{L}$ invoking a black box that solves $T_\mathcal{K}$ once, and then executing read-write operations. That is, starting with any input simplex $s_1 \in \mathcal{I}$, processes invoke the protocol for $T_\mathcal{K}$, and get back as output a simplex $s_2 \in \mathcal{K}$ such that $s_2 \in \Delta_\mathcal{K}(s_1)$, or equivalently $p(s_2) = s_1$. Then the processes performs read/write operations, with $s_2$ as input before deciding on some output simplex $s_3 \in \mathcal{L}$.

By the Asynchronous Computability Theorem [28], the read/write operations subdivide the input complex $\mathcal{K}$, and the decisions define a map that goes from the subdivided complex to $\mathcal{L}$. More precisely, $T_\mathcal{K} \leq T_\mathcal{L}$ implies the existence of a chromatic subdivision $\sigma$ of $\mathcal{K}$ and a color-preserving map $\mu : \sigma(\mathcal{K}) \to \mathcal{L}$. Moreover, for every simplex $s \in \sigma(s_2)$ with $ID(s) = ID(s_2)$, we have $\mu(s) \in \Delta_\mathcal{L}(s_1)$ whenever $s_2 \in \Delta_\mathcal{K}(s_1)$.

We prove that we can assume without loss of generality that the subdivision $\sigma$ is trivial. Let $s_2 \in \mathcal{K}$. By the no irrelevant output simplex assumption, $s_2 \in \Delta_\mathcal{K}(s_1)$ for some simplex $s_1 \in \mathcal{I}$. We show that there exists $s_3 \in \mathcal{L}$ such that for every $s \in \sigma(s_2)$ with $ID(s_2) = ID(s)$, $\mu(s) = s_3$. Therefore, we can assume that $\mu$ maps directly each simplex in $\mathcal{K}$ to simplexes in $\mathcal{L}$. Let $s, s' \in \sigma(s_2)$ with $ID(s) = ID(s') = ID(s_2)$. Assume that $s \cap s' \neq \emptyset$. Because $\mu$ is a map, $\mu(s) \cap \mu(s') \neq \emptyset$. Moreover, $\mu(s), \mu(s') \in \Delta_\mathcal{L}(s_1)$, or equivalently, $\mu(s), \mu(s') \in q^{-1}(s_1)$. However, $q^{-1}(s_1)$ is a disjoint union of simplexes (Definition 4.1(3)). Therefore, $\mu(s) = \mu(s')$. If $s, s'$ do not intersect, they are connected by a chain $s^1(= s), \ldots, s^m(= s')$ of simplexes in $\sigma(s_2)$ in which every two consecutive simplexes intersect. Therefore, the same reasoning repeated over the chain implies that $\mu(s) = \mu(s')$.

So let us assume that $\sigma$ is the trivial subdivision. Let $s_2 \in \mathcal{K}$. As mentioned above, $s_2 \in \Delta_\mathcal{K}(s_1)$, or equivalently, $p(s_2) = s_1$. Also, applying $\mu$ to $s_2$ results in a simplex $\in \Delta_\mathcal{L}(s_1)$. Hence, $q(\mu(s_2)) = s_1 = p(s_2)$,

from which we have $q\mu = p$.

We now prove that tasks $T_{\mathcal{K}}$ and $T_{\mathcal{L}}$ induce two subgroups related by containment. Let $\tilde{u} \in \mathcal{K}$ such that $p(\tilde{u}) = v_*$ and $\mu(\tilde{u}) = \tilde{v} \in \mathcal{L}$. Note that $q(\tilde{v}) = q(\mu(\tilde{u})) = p(\tilde{u}) = v_*$. Let $H_{\mathcal{K}}$ and $H_{\mathcal{L}}$ the subgroups of $G(\mathcal{I}, v_*)$ defined by $H_{\mathcal{K}} = p_{\#}(G(\mathcal{K}, \tilde{u}))$ and $H_{\mathcal{L}} = q_{\#}(G(\mathcal{L}, \tilde{v}))$. $H_{\mathcal{K}}$ and $H_{\mathcal{L}}$ are thus induced by $T_{\mathcal{K}}$ and $T_{\mathcal{L}}$ in the sense of Lemma 4.6. Let $[\alpha] \in H_{\mathcal{K}}$. $[\alpha] = p_{\#}([\tilde{\alpha}])$ for some $[\tilde{\alpha}] \in G(\mathcal{K}, \tilde{u})$. Hence, $[\alpha] = q_{\#}(\mu_{\#}([\tilde{\alpha}]))$, from which we get $[\alpha] \in H_{\mathcal{L}}$ and hence $H_{\mathcal{K}} \subseteq H_{\mathcal{L}}$. $\square$

# C Figures

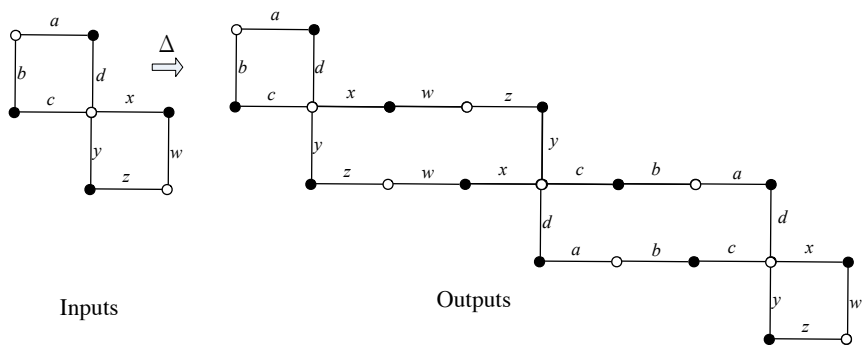
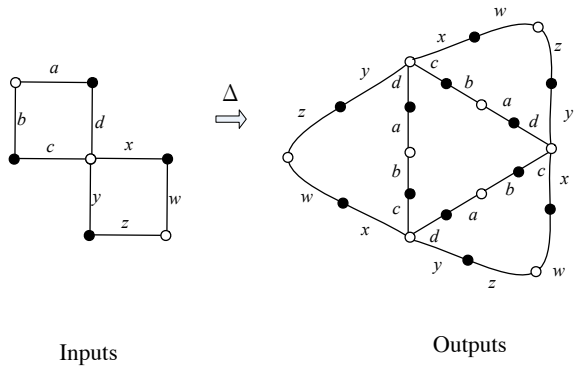
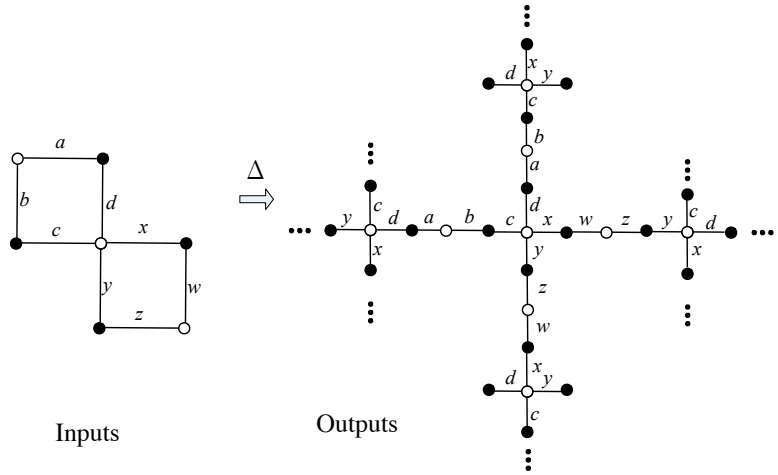



Figure 5: Four cover task




(a) Triangle cover task




(b) Universal cover task

Figure 6: Locality-preserving tasks

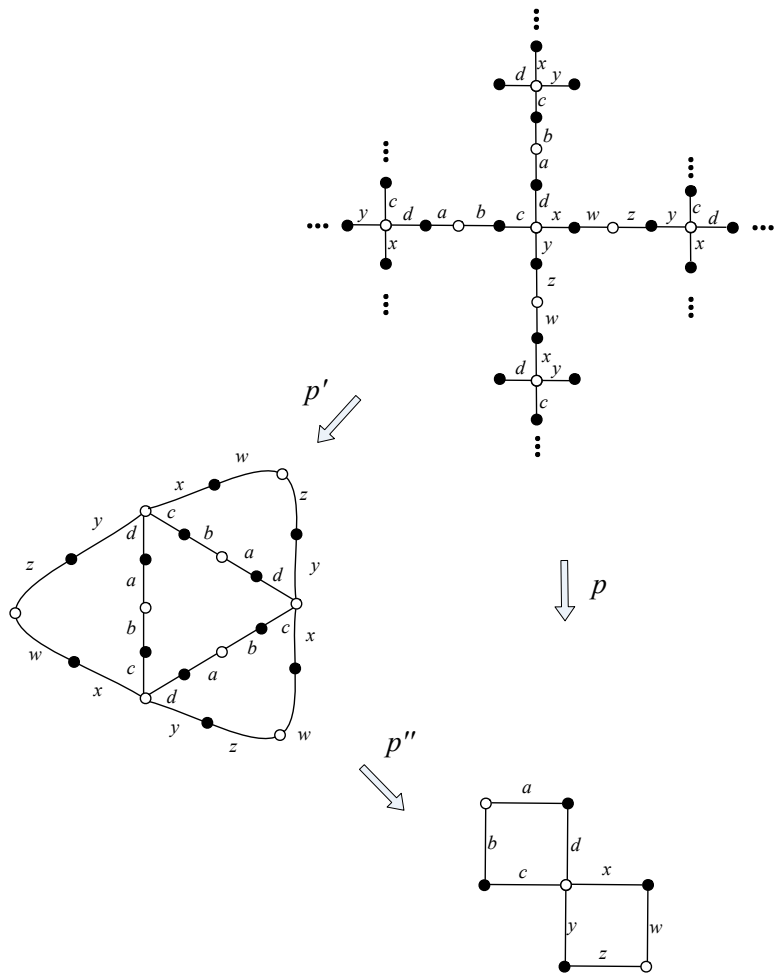



Figure 7: A universal covering covers any other covering