Locality and checkability in wait-free computing

Pierre Fraigniaud[1]⋆, Sergio Rajsbaum[2]⋆⋆, and Corentin Travers[3]⋆⋆⋆

[1] CNRS and U. Paris Diderot, France. pierre.fraigniaud@liafa.jussieu.fr
[2] Instituto de Matemáticas, U. Nacional Autónoma de México, Mexico. rajsbaum@math.unam.mx
[3] LaBRI, U. of Bordeaux and CNRS, France. travers@labri.fr

Abstract. This paper studies several notions of locality that are inherent to the specification of distributed tasks and independent of the computing environment, and investigates the ability of a shared memory wait-free system to solve tasks satisfying various forms of locality.
First, we define a task to be *projection-closed* if every partial output $\pi(t)$ for a full input s is also a valid output for the partial input $\pi(s)$ and prove that projection-closed tasks are precisely those tasks that are wait-free *checkable*.
Our second main contribution is dealing with a stronger notion of locality of topological nature. 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, each simplex s of $\mathcal{I}$ is mapped by Δ to a set of simplexes of $\mathcal{O}$ each isomorphic to s. This topological property yields obstacles for wait-free solvability different in nature from the classical agreement impossibility results. On the other hand, locality-preserving tasks are projection-closed and therefore always 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.

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

⋆ Additional supports from the ANR projects ALADDIN and PROSE, and from the INRIA project GANG.
⋆⋆ Additional supports from UNAM-PAPIIT and PAPIME.
⋆⋆⋆ Work done in part in U. Pierre et Marie Curie. Additional supports from ANR project SPREAD and a CONACYT-CNRS grant.

the set of output configurations, and Δ 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 π. Then, the map Δ 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., [6, 29]) 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 Δ 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 π, can be extended to a $(d+1)$-coloring $\Delta(s)$ for s. In other words, the $(d+1)$-coloring specification Δ satisfies the *monotony* condition

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

for every set of processes s, and every projection π. 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 [10]) whereas 2-coloring rings (of even size) requires $\Omega(n)$ rounds (see [30]).

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.

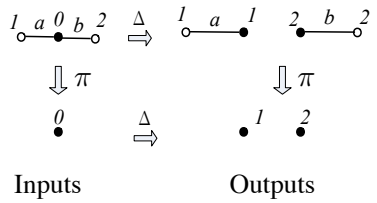


Fig. 1: An intersection-closed task, which is not monotone, and not wait-free solvable. The projection π depicted is for the black process. The black process always starts with 0. When both run, they must agree on the input of the white process, either 1 or 2. If the white process runs solo, it must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

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 whereas monotony is not a 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 π. 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 [30]). Neither it is, in the wait-free setting, a guaranty for a task to be solvable (cf. the FLP impossibility result for consensus [14]). 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.

Our results. The objective of this paper is to investigate the ability of a shared memory system to solve tasks satisfying various forms of locality. Our investiga-

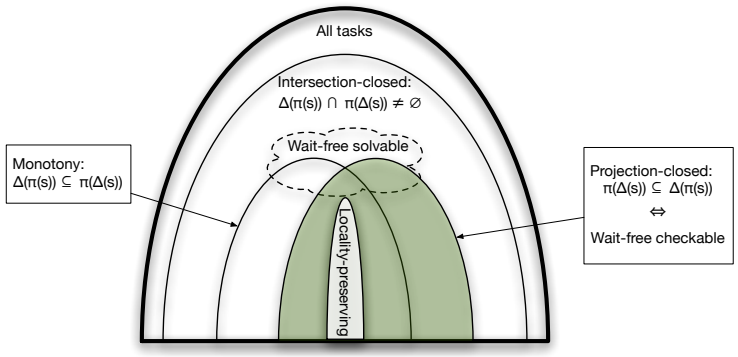


Fig. 2: The universe of tasks

tion 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 Δ 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 Δ, 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 ith 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 1). 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 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 2). 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 Δ, 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 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 [25], we use topology techniques both to prove impossibility results, and to show when one task can implement another.

Due to space limitation, proofs are presented in a companion technical report [17].

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* [2], are of a different nature. Indeed, any wait-free protocol is actually a mapping from a subdivision of the input complex to the output complex [26]. 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 is not decidable [20, 24].

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 [35] have been designed to study communication locality issues. One prominent result in this framework is the $\Omega(\log^* n)$ lower bound [30] 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 [34], on the impact of non-determinism [16], and on the power of oracles providing nodes with information about their environment [15]. The impact on the design of efficient protocols of the absence of a priori knowledge on the global environment has been recently addressed in [28].

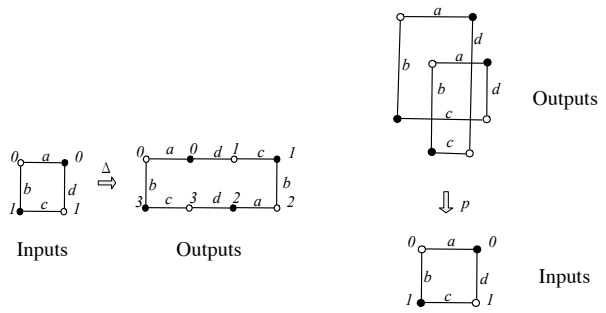


Fig. 3: Two-cover task. The task is for two processes, which start with binary inputs. On the left side of the figure, the specification of the task says that if both start with the same value, they must decide the same value: if they start with 0, decide either 0 or 2; if they start with 1, decide 1 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 Δ 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.

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, 9, 33], under several models of local computation. See [9] 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 [7] 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* [13] 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 [19] and *property testing* [21]. Blum et al. [8] introduced the notion of program *testers* and *correctors*, see also [22, 31].

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 [37]. In the synchronous model, distributed self-testing and correcting protocols that tolerate crash failures are presented for the byzantine generals task in [18]. Self-testing/correcting is reminiscent of the

notion of checking as a means of making a distributed algorithm *self-stabilizing*, as explored in [4, 5] in the synchronous setting. In the framework of network computing, distributed verification has been addressed only recently (see, e.g., [11]), though previous research on proof labeling schemes [27] already gave some insights on the ability of checking global predicates locally (see also the recent paper [23]).

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 [3, 26, 32] 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 step, 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 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* π, 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[4]. In distributed computing, one refers to *colored* simplexes (and complexes), since each vertex v of a simplex is labeled with a distinct process identity $i \in [n]$.

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 $\mathrm{ID}(t) \subset \mathrm{ID}(s)$ 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 Δ 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, Δ 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 Δ 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 $\Delta(s)$.

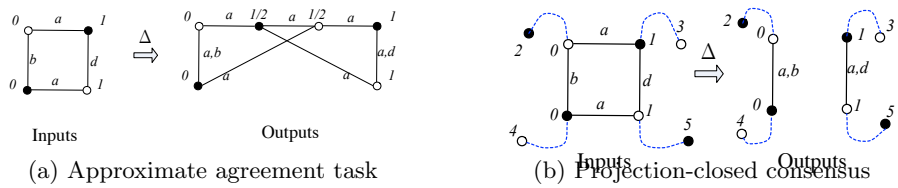
3 Projection-closeness and wait-free checkability

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 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 π, 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 [12] 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 π 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.

[4] 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.



(a) Approximate agreement task (b) Projection-closed consensus

Fig. 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 Δ_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 π 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 Δ_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 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 1. *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 2. *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 [25], 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[5]. We write $T \leq T'$ to emphasize that T' is at least as powerful as T. In Section 4 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. *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, one can define the following equivalence relation $\sim$ between tasks

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

This equivalence notion is the key ingredient in the proof of Theorem 2 below. First consider as an example consensus, which as mentioned above, is 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).

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 [17].

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 a 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

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

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* α 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 α 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 [36] 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 [36]):

Definition 4. *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* Δ 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 5. *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* Δ.

Example of locality-preserving tasks is depicted in Figure 3, 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 Δ, 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 Δ 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 Δ to the two opposite corners that look the same locally.

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 π. 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 π, 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 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 1), we get the following.

Corollary 1. *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 2. *The identity tasks are the only locality-preserving task that are wait-free solvable.*

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). 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.

Edgepath groups and locality-preserving tasks. Our exposition follows Rotman [36]. 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 α' that can be deformed to α along 2-simplexes. More precisely, the paths α' and α 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 ([36], 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 ([36], 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 $\tilde{u} \in V(\tilde{K})$ ([36], 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 ([36], 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 $p_\#(K_H, \tilde{v}_*) = H$ ([36], 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. 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 1. *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 [36] discussed earlier implies the following result.

Lemma 2. *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 $p' = pp''$.*

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. By Lemma 1(2), every locality-preserving task induces a conjugacy class.

Theorem 4. *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). Theorem 4 implies that the set of locality-preserving tasks with input complex $\mathcal{I}$ has a universal task, which is the 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 $\mathcal{I}$.

References

1. Angluin D., Local and Global Properties in Networks of Processors (Extended Abstract). *12th ACM Symp. on Theory of Computing* (STOC), pp. 82–93, 1980.
2. 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.
3. Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations, and Advanced Topics, *Wiley*, 2004.
4. Awerbuch B., Patt-Shamir B. and Varghese G., Self-stabilization by Local Checking and Correction. *32nd IEEE Symp. on Foundations of Computer Science* (FOCS), pp. 268–277, 1991.
5. Awerbuch B. and Varghese G., Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. *32nd IEEE Symp. on Foundations of Computer Science* (FOCS), pp. 258–267, 1991.
6. Barenboim L. and Elkin M., Deterministic distributed vertex coloring in polylogarithmic time. *29th ACM Symp. on Principles of Distributed Computing* (PODC), pp 410-419, 2010.
7. Blum M. and Kannan S., Designing Programs that Check Their Work. *J. ACM* 42(1): 269–291 (1995).
8. Blum M., Luby M. and Rubinfeld R., Self-Testing/Correcting with Applications to Numerical Problems. *J. Comput. Syst. Sci.* 47(3): 549–595 (1993).
9. Chalopin J. and Métivier Y., On the Power of Synchronization Between two Adjacent Processes. *Distributed Computing* 23(3): 177-196 (2010).
10. Cole R. and Vishkin U., Deterministic coin tossing with applications to optimal parallel list ranking *Information and Control* 70(1): 3253 (1986).
11. Das Sarma A., Holzer S., Kor L., Korman A., Nanongkai D., Pandurangan G., Peleg D., and Wattenhofer R., Distributed Verification and Hardness of Distributed Approximation *43rd ACM Symp. on Theory of Computing* (STOC), 2011.
12. 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).

13. Ergün F., Kannan S., Kumar R., Rubinfeld R. and Viswanathan M., Spot-Checkers. *J. Comput. Syst. Sci.* 60(3): 717-751 (2000).
14. Fischer M., Lynch N. and Paterson M., Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.
15. Fraigniaud P., Ilcinkas D., and Pelc A., Communication algorithms with advice. *J. Comput. Syst. Sci.* 76(3-4): 222-232 (2010).
16. Fraigniaud P., Korman A. and Peleg D., Local Distributed Decision. *arXiv:1011.2152*.
17. Fraigniaud P., Rajsbaum S. and Travers C., Locality and Checkability in Wait-free Computing. *Technical report* XXXX
18. Franklin M., Garay J.A. and Yung M., Self-Testing/Correcting Protocols. *13th International Symp. on Distributed Computing* (DISC), pp. 269–284, 1999.
19. 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).
20. Gafni E. and Koutsoupias E., Three-Processor Tasks Are Undecidable. *SIAM J. Comput.* 28(3): 970–983 (1999).
21. 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).
22. Goldwasser S., Gutfreund D., Healy A., Kaufman T. and Rothblum G., A (De)constructive Approach to Program Checking. *40th ACM Symp. on Theory of Computing* (STOC), pp. 143–152, 2008.
23. Goos M. and Suomela J., Locally checkable proofs. *30th ACM Symp. on Principles of Distributed Computing* (PODC), 2011.
24. Herlihy M. and Rajsbaum S., The Decidability of Distributed Decision Tasks. *29th ACM Symp. on the Theory of Computing* (STOC), pp. 589–598, 1997.
25. Herlihy M. and Rajsbaum S., A Classification of Wait-Free Loop Agreement Tasks. *Theor. Comput. Sci.*, 291(1): 55–77, 2003.
26. Herlihy M. and Shavit N., The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, 1999.
27. Korman A., Kutten S. and Peleg D., Proof Labeling Schemes. *Distributed Computing* 22, (2010), 215–233.
28. Korman A., Sereni J.-S., and Viennot L, Toward more Localized Local Algorithms: Removing Assumptions concerning Global Knowledge. *30th ACM Symp. on Principles of Distributed Computing* (PODC), 2011.
29. Kuhn F. and Wattenhofer R., On the complexity of distributed graph coloring. *25th ACM Symp. on Principles of Distributed Computing* (PODC), pp 7-15, 2006.
30. Linial N., Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.
31. Lipton R., New Directions in Testing. *DIMACS workshop on distributed computing and cryptography*, 2:191–202, 1991.
32. Lynch N., *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
33. Mazurkiewicz A., Distributed Enumeration. *Inf. Process. Lett.* 61, 233–239 (1997).
34. Naor M. and Stockmeyer L., What can be Computed Locally? *SIAM J. Comput.* 24(6): 1259–1277 (1995).
35. Peleg D., Distributed Computing: A Locality-Sensitive Approach. *SIAM*, 2000.
36. Rotman J., Covering Complexes with Applications to Algebra. *Rocky Mountain J. of Mathematics,* 3(4): 641–674 (1973).
37. Rubinfeld R., Designing Checkers for Programs that Run in Parallel. *Algorithmica* 15(4): 287–301 (1996).