Topological Structure of Asynchronous Computing I

Maruth Goyal

UT Austin

Spring 2021

Table of Contents

Happy Holi!

Introduction

- In real systems it is very desirable, almost necessary to process multiple things at the same time.

Introduction

- In real systems it is very desirable, almost necessary to process multiple things at the same time.
- Happens in the form of programs which are concurrent, or parallel, or both.
  - **Concurrency**: Multiplexing 1000s of tasks over a single thread. eg: servers, just about any Go program
  - **Parallelism**: Multiplexing threads over multiple cores, truly executing multiple operations at the same time. eg: multithreaded OSs

Introduction

- In real systems it is very desirable, almost necessary to process multiple things at the same time.
- Happens in the form of programs which are concurrent, or parallel, or both.
    - **Concurrency**: Multiplexing 1000s of tasks over a single thread. eg: servers, just about any Go program
    - **Parallelism**: Multiplexing threads over multiple cores, truly executing multiple operations at the same time. eg: multithreaded OSs
- However, such programs often require synchronization among tasks/threads for correctness.
- Synchronizing using primitives such as Mutexes, Locks, Semaphores, etc is susceptible to deadlock, livelock, thread starvation etc.

Introduction

- In real systems it is very desirable, almost necessary to process multiple things at the same time.
- Happens in the form of programs which are concurrent, or parallel, or both.
    - **Concurrency**: Multiplexing 1000s of tasks over a single thread. eg: servers, just about any Go program
    - **Parallelism**: Multiplexing threads over multiple cores, truly executing multiple operations at the same time. eg: multithreaded OSs
- However, such programs often require synchronization among tasks/threads for correctness.
- Synchronizing using primitives such as Mutexes, Locks, Semaphores, etc is susceptible to deadlock, livelock, thread starvation etc.
- This motivates non-blocking programs. We will focus on wait-free programs.

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

- Wait-free algorithms guarantee that every thread will always make progress.
- This also guarantees system-wide progress.
- Well studied class of algorithms [Attiya et al., 1994, Hunt et al., 2010, Herlihy, 1988, Kogan and Petrank, 2012]
- Wait-free data structures:
  - Queue: [Kogan and Petrank, 2011]
  - Hash Table: [Laborde et al., 2017]
  - Linked List: [Timnat et al., 2012]

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

- **Question we're interested in:** *which problems have wait-free algorithms?*

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

- **Question we're interested in:** *which problems have wait-free algorithms?*
- We will look at seminal work from Maurice Herlihy and Nir Shavit from 1999: *The Topological Structure of Asynchronous Computability* [Herlihy and Shavit, 1999]
    - Awarded 2004 Gödel prize for this work.
- They provide necessary and sufficient conditions for problems to have wait-free algorithms using techniques from Algebraic, and Combinatorial Topology.

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

- They proved the following problems **do not** have wait-free algorithms
  - **Renaming** [Attiya et al., 1990]: Suppose you have n processors, each with a unique ID in $[M]$. Now want the processors to choose *unique* names in $[N]$ where $n \leq N < M$.

Introduction

Definition (Wait-free algorithm)

An asynchronous algorithm wherein for each thread of execution, each operation is guaranteed to finish in a bounded number of steps is known as a **wait-free** algorithm

- They proved the following problems **do not** have wait-free algorithms
  - **Renaming** [Attiya et al., 1990]: Suppose you have n processors, each with a unique ID in $[M]$. Now want the processors to choose *unique* names in $[N]$ where $n \leq N < M$.
  - k-**set agreement** [Chaudhuri, 1990]: Each processor has a starting value, and must choose the value of any of the processors as its final value. The processors may choose at most k distinct values.

Prior Work

- Independently, [Saks and Zaharoglou, 1993] proved impossibility of k-set agreement using techniques from Topology. However, their method (according to Herlihy and Shavit), seem “specific” to set agreement, while their method generalizes to arbitrary problems.

Theorem Statement

- The following is the statement of the main theorem.

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision* σ *of* $\mathcal{I}$ *and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\mathrm{carrier}(S, \mathcal{I}))$.

Theorem Statement

- The following is the statement of the main theorem.

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision σ of $\mathcal{I}$ and a color-preserving simplicial map

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S in $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\text{carrier}(S, \mathcal{I}))$.

- In the rest of this talk we will break down and define every subpart of this theorem.
- In the proceeding sessions, we will (1) study the application of this theorem to different problems, and then finally (2) study the proof of this theorem.

Table of Contents

Computational Model

Theorem (Asynchronous Computability Theorem)

A **decision task** $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *has a* **wait-free protocol** *using* **read-write memory** *if and only if there exists a chromatic subdivision* σ *of* $\mathcal{I}$ *and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\mathrm{carrier}(S, \mathcal{I}))$.

Computational Model

1. **Decision Tasks**
2. Wait-Free Protocols
    1. Protocols
    2. Read-Write memory

Computational Model

- "Decision Tasks" allow us to appropriately formalize problems in the Asynchronous setting.

Computational Model

- "Decision Tasks" allow us to appropriately formalize problems in the Asynchronous setting.
- In particular, a decision task is defined using a 3-tuple, $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$. Here, $\mathcal{I}$ is the "input vector", $\mathcal{O}$ the "output vector", and Δ a "task specification"

Definition (I/O Vectors)

An input vector I (resp output vector O) is a vector of length n, where there are n processors, such that each entry is either a value of type D_I (resp. D_O), or $\perp$. At least one entry must not be $\perp$.

Computational Model

- "Decision Tasks" allow us to appropriately formalize problems in the Asynchronous setting.
- In particular, a decision task is defined using a 3-tuple, $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$. Here, $\mathcal{I}$ is the "input vector", $\mathcal{O}$ the "output vector", and Δ a "task specification"

Definition (I/O Vectors)

An input vector I (resp output vector O) is a vector of length n, where there are n processors, such that each entry is either a value of type D_I (resp. D_O), or $\perp$. At least one entry must not be $\perp$.

- If entry $I[x] = \perp$, it means processor x will not participate in the execution.
- Similarly, if $O[x] = \perp$, then it means processor x did not choose an output in the execution.

Computational Model

- It is useful to relate input/output vectors which are basically the same, except perhaps some processors failed / don't participate.

Computational Model

- It is useful to relate input/output vectors which are basically the same, except perhaps some processors failed / don't participate.

Definition

A vector $\vec{U}$ is said to be a prefix of $\vec{V}$ is for $0 \leq i \leq n$, either $\vec{U}[i] = \vec{V}[i]$, or $\vec{U}[i] = \perp$.

Computational Model

- It is useful to relate input/output vectors which are basically the same, except perhaps some processors failed / don't participate.

Definition

A vector $\vec{U}$ is said to be a prefix of $\vec{V}$ is for $0 \leq i \leq n$, either $\vec{U}[i] = \vec{V}[i]$, or $\vec{U}[i] = \perp$.

- Since we are considering wait-free algorithms, we are interested in modelling scenarios where even in the case some processors fail, all others will still produce output. Thus, this motivates the following definition:

Computational Model

- It is useful to relate input/output vectors which are basically the same, except perhaps some processors failed / don't participate.

Definition

A vector $\vec{U}$ is said to be a prefix of $\vec{V}$ is for $0 \leq i \leq n$, either $\vec{U}[i] = \vec{V}[i]$, or $\vec{U}[i] = \perp$.

- Since we are considering wait-free algorithms, we are interested in modelling scenarios where even in the case some processors fail, all others will still produce output. Thus, this motivates the following definition:

Definition

A set V of vectors is *prefix-closed* if for all $\vec{V} \in V$, every prefix $\vec{U}$ of $\vec{V}$ is in V.

Computational Model

- We may now define the “task specification”, which induces a map from inputs to valid sets of outputs. Thus, defining the task to be solved.

Computational Model

- We may now define the "task specification", which induces a map from inputs to valid sets of outputs. Thus, defining the task to be solved.

Definition

A task specification is a relation $\Delta \subset I \times O$ where I, O are prefix-closed input, and output vectors respectively. Moreover, for each $\vec{I} \in I$, there exists at least one $\vec{O} \in O$ such that $(\vec{I}, \vec{O}) \in \Delta$. We will use $\Delta(\vec{I})$ to denote the set $\{\vec{O} \mid (\vec{I}, \vec{O}) \in \Delta\}$.

Computational Model

1. Decision Tasks ✓
2. Wait-Free Protocols
    1. **Protocols**
    2. Read-Write memory

Computational Model

Definition (I/O automaton)

An I/O automaton is a nondeterministic automaton with a (not necessarily finite) set of states, a set of input events, output events, and a transition relation. An execution is an alternating sequence of states and events, given some initial state.

Computational Model

Definition (I/O automaton)

An I/O automaton is a nondeterministic automaton with a (not necessarily finite) set of states, a set of input events, output events, and a transition relation. An execution is an alternating sequence of states and events, given some initial state.

Definition (Process)

A *process* P is an automaton with output events $\textsc{call}(P, v, X, T)$, and $\textsc{Finish}(P, v)$, and input events $\textsc{Start}(P, v)$, and $\textsc{Return}(P, v, X, T)$ where P is a process id, v is a value, X an object, and T is a type.

Computational Model

Definition (I/O automaton)

An I/O automaton is a nondeterministic automaton with a (not necessarily finite) set of states, a set of input events, output events, and a transition relation. An execution is an alternating sequence of states and events, given some initial state.

Definition (Process)

A *process* P is an automaton with output events $\textsc{Call}(P, v, X, T)$, and $\textsc{Finish}(P, v)$, and input events $\textsc{Start}(P, v)$, and $\textsc{Return}(P, v, X, T)$ where P is a process id, v is a value, X an object, and T is a type.

- Intuitively: a process receives $\textsc{Start}$ is it's the entry-point. The $\textsc{Return}$ event models composition with a previous "subroutine".
- Thus, when a process finishes it can either terminate with $\textsc{Finish}$ or it can call the next subroutine with $\textsc{Call}$.

Computational Model

Definition (Object)

An object X is an automaton with input events $\textsc{Call}(P, v, X, T)$, and output event $\textsc{Return}(P, v, X, T)$.

Definition (Read/Write Memory Object)

A *read/write* memory object M is an automaton with input event $\textsc{Call}(P, \textsc{Read}, M, a)$ (also written as $\textsc{Read}(P, a)$), and a corresponding $\textsc{Call}(P, (\textsc{Write}, v), M, a)$ (also written as $\textsc{Write}(P, a, v)$).

Memory Model

- We assume memory is *atomic snapshot memory*.
- There is an array a of length n, where there are n processes.

Memory Model

- We assume memory is *atomic snapshot memory*.
- There is an array a of length n, where there are n processes.
- **reads:** A read atomically returns the entire array.
- **writes:** A write updates the entry corresponding to the processor.

Memory Model

- We assume memory is *atomic snapshot memory*.
- There is an array a of length n, where there are n processes.
- **reads:** A read atomically returns the entire array.
- **writes:** A write updates the entry corresponding to the processor.
- **commutativity:** Reads commute with each other, and writes commute with each other.
- **linearizability:** Atomic snapshot memory is linearizable. i.e., for any sequence of potentially concurrent reads and writes, there's an equivalent sequential execution which preserves relative ordering of events.

Wait Free Protocols

Definition (Wait-Free solving)

A protocol P wait-free solves a decision task, if given an input vector $\vec{I}$ at least one processor produces a $\textsc{Finish}$ event in a finite number of steps independent of the whether the other processors finish, and the output vector $\vec{O}$ produced by the processors is a prefix of some vector in $\Delta(\vec{I})$.

Table of Contents

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ has a wait-free protocol using read-write memory if and only if there exists a **chromatic subdivision** σ *of* $\mathcal{I}$ *and a* **color-preserving simplicial map**

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each **simplex** S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\text{carrier}(S, \mathcal{I}))$.

- Geometric/Abstract simplex
- Geomtric/Abstract complex
- simplicial Map
- Colored complex
- Subdivision of a complex
- Color preserving maps, chromatic subdivision
- Carrier of simplex in subdivision

Topology Background

Definition (Geometric n-simplex)

Given a set of points $\{v_0, \ldots, v_n\}$ in some Euclidean space (say $\mathbb{R}^d$), the geometric n-simplex on these points is the set

$$S = \{x \mid x = \sum_{i=0}^{n} t_i \cdot v_i,\ \sum_{i=0}^{n} t_i = 1,\ 0 \leq t_i \leq 1\} \equiv (v_0, \ldots, v_n)$$

The dimension of a simplex with $n+1$ vertices is n, written as $\dim(S)$.

Topology Background

Definition (Geometric n-simplex)

Given a set of points $\{v_0, \ldots, v_n\}$ in some Euclidean space (say $\mathbb{R}^d$), the geometric n-simplex on these points is the set

$$S = \{x \mid x = \sum_{i=0}^{n} t_i \cdot v_i,\ \sum_{i=0}^{n} t_i = 1,\ 0 \leq t_i \leq 1\} \equiv (v_0, \ldots, v_n)$$

The dimension of a simplex with $n+1$ vertices is n, written as $\dim(S)$.

- For 2 points, (i.e., a 1-simplex) it is a line
- For 3 points, (i.e., a 2-simplex) it is a solid triangle
- For 4 points, (i.e., a 3-simplex) it is a solid tetrahedron

Topology Background

Definition (Geometric n-simplex)

Given a set of points $\{v_0, \ldots, v_n\}$ in some Euclidean space (say $\mathbb{R}^d$), the geometric n-simplex on these points is the set

$$S = \{x \mid x = \sum_{i=0}^{n} t_i \cdot v_i,\ \sum_{i=0}^{n} t_i = 1,\ 0 \le t_i \le 1\} \equiv (v_0, \ldots, v_n)$$

The dimension of a simplex with $n+1$ vertices is n, written as $\dim(S)$.

- For 2 points, (i.e., a 1-simplex) it is a line
- For 3 points, (i.e., a 2-simplex) it is a solid triangle
- For 4 points, (i.e., a 3-simplex) it is a solid tetrahedron

Definition (Face)

Any simplex spanned by a proper subset of $\{v_0, \ldots, v_n\}$ is called a proper face of S.

Topology Background

Definition (Geometric n-simplex)

Given a set of points $\{v_0, \ldots, v_n\}$ in some Euclidean space (say $\mathbb{R}^d$), the geometric n-simplex on these points is the set

$$S = \{x \mid x = \sum_{i=0}^{n} t_i \cdot v_i,\ \sum_{i=0}^{n} t_i = 1,\ 0 \leq t_i \leq 1\} \equiv (v_0, \ldots, v_n)$$

The dimension of a simplex with $n+1$ vertices is n, written as $\dim(S)$.

- Eventually, the vertices of the simplex will be used to identify the states of the various processors, and the simplices will model the consistent state of multiple processors involved in solving a task.

Topology Background

Definition (Geometric n-simplex)

Given a set of points $\{v_0, \ldots, v_n\}$ in some Euclidean space (say $\mathbb{R}^d$), the geometric n-simplex on these points is the set

$$S = \{x \mid x = \sum_{i=0}^{n} t_i \cdot v_i,\ \sum_{i=0}^{n} t_i = 1,\ 0 \leq t_i \leq 1\} \equiv (v_0, \ldots, v_n)$$

The dimension of a simplex with $n+1$ vertices is n, written as $\dim(S)$.

- Eventually, the vertices of the simplex will be used to identify the states of the various processors, and the simplices will model the consistent state of multiple processors involved in solving a task.
- However, we want to reason about multiple possible sets of input states, corresponding to different input vectors $\vec{I}$.

Topology Background

Definition (Geomtric simplicial n-complex)

A geometric simplicial complex $\mathcal{K}$ in a Euclidean space is a collection of geometric simplices such that

- Every face of every simplex of $\mathcal{K}$ is also a simplex of $\mathcal{K}$
- The intersection of any two simplices of $\mathcal{K}$ is also a simplex of $\mathcal{K}$.

The dimension of $\mathcal{K}$, $\dim(\mathcal{K}) = \max_{S \in \mathcal{K}} \dim S$.

Topology Background

Definition (Geomtric simplicial n-complex)

A geometric simplicial complex $\mathscr{K}$ in a Euclidean space is a collection of geometric simplices such that

- Every face of every simplex of $\mathscr{K}$ is also a simplex of $\mathscr{K}$
- The intersection of any two simplices of $\mathscr{K}$ is also a simplex of $\mathscr{K}$.

The dimension of $\mathscr{K}$, $\dim(\mathscr{K}) = \max_{S \in \mathscr{K}} \dim S$.

Definition

A subset $\mathscr{L}$ of a complex $\mathscr{K}$, is called a subcomplex if it's closed under containment and intersection.

Topology Background

Definition (Geomtric simplicial n-complex)

A geometric simplicial complex $\mathscr{K}$ in a Euclidean space is a collection of geometric simplices such that

- Every face of every simplex of $\mathscr{K}$ is also a simplex of $\mathscr{K}$
- The intersection of any two simplices of $\mathscr{K}$ is also a simplex of $\mathscr{K}$.

The dimension of $\mathscr{K}$, $\dim(\mathscr{K}) = \max_{S \in \mathscr{K}} \dim S$.

Definition

A subset $\mathscr{L}$ of a complex $\mathscr{K}$, is called a subcomplex if it's closed under containment and intersection.

Definition

The ℓ-skeleton of a complex $\mathscr{K}$, denoted $\mathrm{skel}^{\ell}(\mathscr{K})$ is the subcomplex consisting of all simplices of dimension at most ℓ.

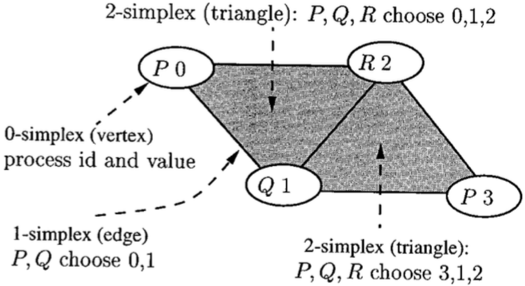

FIG. 4. Vertexes and simplexes.

- A construction that will be most important is maps between complexes which map the simplices of one to simplices of the other. We will refer to these as simplicial maps.

- A construction that will be most important is maps between complexes which map the simplices of one to simplices of the other. We will refer to these as simplicial maps.
- Intuitively, this will be useful to study the interaction of the input, and output simplices for a decision task.

Definition

Let $\mathcal{K}$ and $\mathcal{L}$ be complexes, possibly of different dimensions. A vertex map $\mu : \mathrm{skel}^0(\mathcal{K}) \to \mathrm{skel}^0(\mathcal{L})$ carries vertices of $\mathcal{K}$ to vertices of $\mathcal{L}$. If this in addition carries simplices of $\mathcal{K}$ to simplices of $\mathcal{L}$ it is called a simplicial map.

- A construction that will be most important is maps between complexes which map the simplices of one to simplices of the other. We will refer to these as simplicial maps.
- Intuitively, this will be useful to study the interaction of the input, and output simplices for a decision task.

Definition

Let $\mathscr{K}$ and $\mathscr{L}$ be complexes, possibly of different dimensions. A vertex map $\mu : \mathrm{skel}^0(\mathscr{K}) \to \mathrm{skel}^0(\mathscr{L})$ carries vertices of $\mathscr{K}$ to vertices of $\mathscr{L}$. If this in addition carries simplices of $\mathscr{K}$ to simplices of $\mathscr{L}$ it is called a simplicial map.

Definition

A coloring χ of a complex $\mathscr{K}$ assigns every vertex of $\mathscr{K}$ a color, such that no two vertices connected by a 1-simplex (line) have the same color. The coloring can be thought of as a dimension-preserving simplicial map $\chi : \mathrm{skel}^0(\mathscr{K}) \to \mathrm{skel}^0(S)$ where S is the complex induced by the faces of a n-dimensional simplex S. Intuitively, the coloring condition is enforced by such a map since it is dimension preserving.

- We will now consider a structure which further slices up a complex into potentially smaller simplices.

Topology Background

- We will now consider a structure which further slices up a complex into potentially smaller simplices.
- Intuitively, this helps model certain susbets of processor states, and/or evolution of intermediate states.

Definition

Let $\mathcal{K}$ be a complex in $\mathbb{R}^\ell$. A complex $\sigma(\mathcal{K})$ is a subdivision of $\mathcal{K}$ if

- Each simplex in $\sigma(\mathcal{K})$ is contained in a simplex in $\mathcal{K}$
- Each simplex of $\mathcal{K}$ is the union of finitely many simplices in $\sigma(K)$.

Topology Background

- We will now consider a structure which further slices up a complex into potentially smaller simplices.
- Intuitively, this helps model certain susbets of processor states, and/or evolution of intermediate states.

Definition

Let $\mathcal{K}$ be a complex in $\mathbb{R}^\ell$. A complex $\sigma(\mathcal{K})$ is a subdivision of $\mathcal{K}$ if

- Each simplex in $\sigma(\mathcal{K})$ is contained in a simplex in $\mathcal{K}$
- Each simplex of $\mathcal{K}$ is the union of finitely many simplices in $\sigma(K)$.

Definition

If S is a simplex if $\sigma(\mathcal{K})$, the **carrier** of S denoted $\mathrm{carrier}(S, \mathcal{K})$ is the unique smallest $T \in \mathcal{K}$ such that $S \subset T$.
Intuitively, it is the simplex in the original complex which was subdivided to create S in $\sigma(\mathcal{K})$.

Topology Background

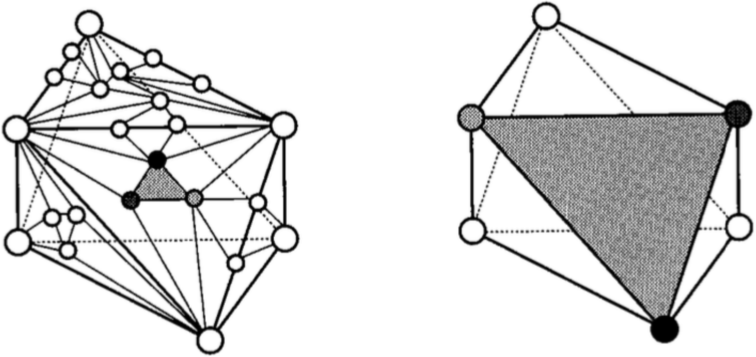

FIG. 7. A simplex and its carrier.

Topology Background

Definition

A *chromatic subdivision* of $(\mathcal{K}, \chi_{\mathcal{K}})$ is a chromatic complex $(\sigma(\mathcal{K}), \chi_{\sigma(\mathcal{K})})$ such that $\sigma(\mathcal{K})$ is a subdivision of $\mathcal{K}$, and for all S in $\sigma(\mathcal{K})$, we have $\chi_{\sigma(\mathcal{K})}(S) \subseteq \chi_{\mathcal{K}}(\mathrm{carrier}(S, \mathcal{K}))$

Table of Contents

Combining Topology and Computation

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision* σ *of* $\mathcal{I}$ *and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\text{carrier}(S, \mathcal{I}))$.

- We have all these topological constructions, but how do we embed our decision task to work with these constructions?

Combining Topology and Computation

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision* σ *of* $\mathcal{I}$ *and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\mathrm{carrier}(S, \mathcal{I}))$.

- We have all these topological constructions, but how do we embed our decision task to work with these constructions?
- Need to (1) Represent the Input/Output sets $\mathcal{I}$ and $\mathcal{O}$ using complexes, and (2) lift Δ to a topological specification.

Combining Topology and Computation

- We first need to introduce a generalization of the geometric simplices from earlier in order to embed our tasks.
- However, conveniently there is provably a correspondence between this generalization and a geometric representation.

Combining Topology and Computation

- We first need to introduce a generalization of the geometric simplices from earlier in order to embed our tasks.
- However, conveniently there is provably a correspondence between this generalization and a geometric representation.

Definition

An *abstract simplex* is simply a non-empty set.

Definition

An *abstract complex* $\mathfrak{K}$ is a collection of abstract simplices closed under containment. i.e., if $S \in \mathfrak{K}$ then so is any face of S.

Combining Topology and Computation

Definition

Let $\vec{I} \in I$ be an input vector. The *input simplex* corresponding to $\vec{I}$, denoted $\mathfrak{I}(I)$, is the abstract colored simplex whose vertices $\langle P_i, v_i \rangle$ correspond to the participating entries in $\vec{I}$, for which $\vec{I}[i] = v_i \neq \perp$. Output simplices defined similarly.

Combining Topology and Computation

Definition

Let $\vec{I} \in I$ be an input vector. The *input simplex* corresponding to $\vec{I}$, denoted $\mathfrak{T}(I)$, is the abstract colored simplex whose vertices $\langle P_i, v_i \rangle$ correspond to the participating entries in $\vec{I}$, for which $\vec{I}[i] = v_i \neq \perp$. Output simplices defined similarly.

Definition

The *input complex* corresponding to I, dneoted by $\mathcal{I}$ is the collection of input simplices $\mathfrak{T}(I)$ corresponding to the input vectors of I. Output complex $\mathfrak{O}$ defined similarly.

Combining Topology and Computation

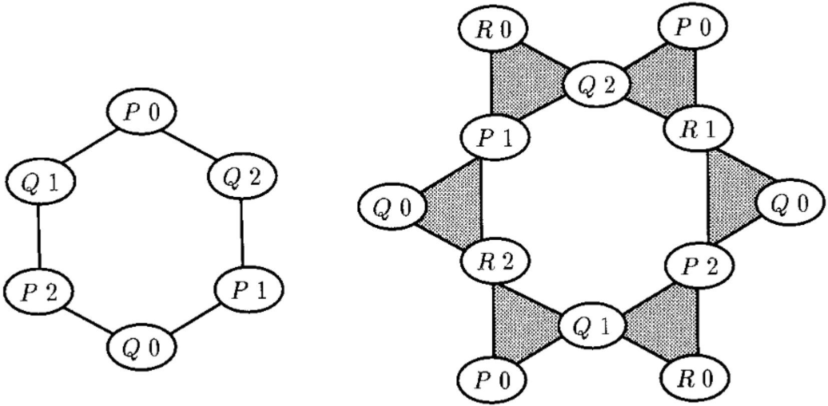

FIG. 8. Some output complexes for the renaming task.

Definition

The *topological task specification* corresponding to the task specification Δ, denoted $\Delta \subseteq \mathcal{I} \times \mathcal{O}$, is defined to contain all pairs $(\mathfrak{T}(\vec{I}), \mathfrak{T}(\vec{O}))$ where $(\vec{I}, \vec{O})$ is in the task specification Δ.

Putting it all Together I

Theorem (Asynchronous Computability Theorem)

A decision task $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *has a wait-free protocol using read-write memory if and only if there exists a chromatic subdivision* σ *of* $\mathcal{I}$ *and a color-preserving simplicial map*

$$\mu : \sigma(\mathcal{I}) \to \mathcal{O}$$

such that for each simplex S *in* $\sigma(\mathcal{I})$, $\mu(S) \in \Delta(\mathrm{carrier}(S, \mathcal{I}))$.

My intuition:

- Last condition $\mu(S) \in \Delta(\mathrm{carrier}(S, \mathcal{I}))$ enforces that μ is mapping to valid output simplexes (i.e., protocol actually solves the task)
- The coloring enforces some notion of "independence" among tasks as desired in a wait-free protocol

Putting it all Together II

- The subdivision allows considering more fine-grained / intermediate states of processors, and the color-preserving map says that these states can be mapped to a valid output state which still preserves that independence.

References I

Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., and Reischuk, R. (1990).
Renaming in an asynchronous environment.
Journal of the ACM (JACM), 37(3):524–548.

Attiya, H., Lynch, N., and Shavit, N. (1994).
Are wait-free algorithms fast?
Journal of the ACM (JACM), 41(4):725–763.

Chaudhuri, S. (1990).
Agreement is harder than consensus: Set consensus problems in totally asynchronous systems.
In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 311–324.

References II

Herlihy, M. and Shavit, N. (1999).
The topological structure of asynchronous computability.
Journal of the ACM (JACM), 46(6):858–923.

Herlihy, M. P. (1988).
Impossibility and universality results for wait-free synchronization.
In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010).
Zookeeper: Wait-free coordination for internet-scale systems.
In *USENIX annual technical conference*, volume 8.

Kogan, A. and Petrank, E. (2011).
Wait-free queues with multiple enqueuers and dequeuers.
ACM SIGPLAN Notices, 46(8):223–234.

References III

Kogan, A. and Petrank, E. (2012).
A methodology for creating fast wait-free data structures.
ACM SIGPLAN Notices, 47(8):141–150.

Laborde, P., Feldman, S., and Dechev, D. (2017).
A wait-free hash map.
International Journal of Parallel Programming, 45(3):421–448.

Saks, M. and Zaharoglou, F. (1993).
Wait-free k-set agreement is impossible: The topology of public knowledge.
In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 101–110.

References IV

Timnat, S., Braginsky, A., Kogan, A., and Petrank, E. (2012).
Wait-free linked-lists.
In *International Conference On Principles Of Distributed Systems*, pages 330–344. Springer.

Questions?

Thank You!