

# Snowflake: A Distributed Streaming Decoder

Tim Chan<sup>1,2</sup>

<sup>1</sup>Department of Materials, University of Oxford, Parks Road, Oxford OX1 3PH, United Kingdom

<sup>2</sup>Center for Quantum Information and Quantum Biology, The University of Osaka, 1-2 Machikaneyama, Toyonaka 560-0043, Japan

We design *Snowflake*, a quantum error correction decoder that, for the surface code under circuit-level noise, is roughly 25% more accurate than the Union-Find decoder, with a better mean runtime scaling: subquadratic as opposed to cubic in the code distance. Our decoder runs in a streaming fashion and has a distributed, local implementation. In designing *Snowflake*, we propose a new method for general stream decoding that eliminates the processing overhead due to window overlap in existing windowing methods.

## 1 Introduction

Decoders are crucial to the quantum error correction that enables fault-tolerant quantum computing. These classical algorithms must decode (i.e. infer errors from measurements and fix them) accurately and extremely quickly to prevent the quantum computation from corrupting. It is difficult to design a decoder sufficiently accurate, fast, and practical to implement in hardware [1–3]. This last requirement has led to interest in decoders that are local i.e. running on a grid of identical processors, each communicating only with their nearest neighbours [4–15]. Such decoders benefit from parallelism [16, §III.D], repairability [17, §3], reduced signal losses [18, pp 6–7], and lower latency [1, §VI] [19, §4.2] when used with local (requiring no long-range qubit–qubit interactions) codes.

A relatively new candidate gaining in popularity [2, 20–30] is the Union-Find decoder (UF) [31, 32]. Recent literature has explored UF implementations of varying levels of locality [17, 33–39], the first being *Helios* [40]. The closely related *Macar* [17] exhibits very similar runtime behaviour to *Helios*, and is the implementation we will repeatedly refer to for comparison. These implementations, save *Helios*, tackled only the batch decoding problem: a simplification of the stream decoding problem that actually occurs in practice. Relatively little work has gone into the latter: since 2002, two methods have been proposed allowing most batch decoders to be used as stream decoders. These are the commonly used *forward method* [41, §VI.B] and the newer *sandwich method* [42, 43], using the terminology of [43].

Tim Chan: [timothy.chan@materials.ox.ac.uk](mailto:timothy.chan@materials.ox.ac.uk)

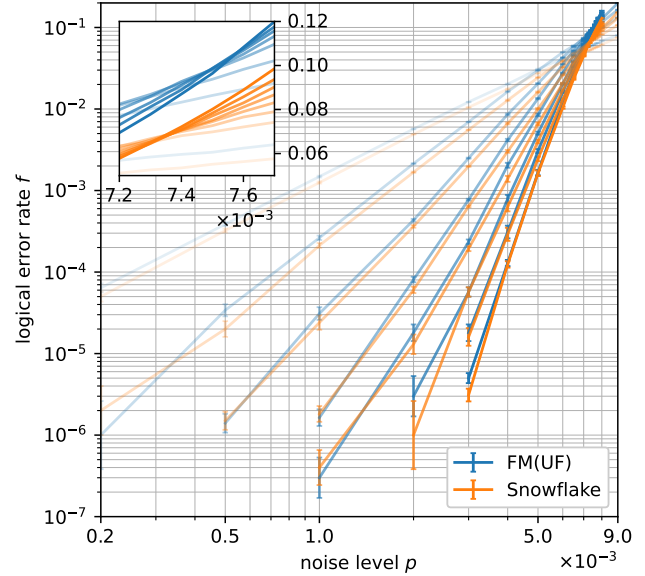


Figure 1: Threshold plots for the surface code under circuit-level noise. Each line corresponds to a fixed distance  $d = 3, 5, \dots, 17$  from shallowest to steepest. Each datapoint is the mean of  $10^4$ – $10^7$  lots of  $d$  measurement rounds; errorbars show standard error. The inset zooms in on both thresholds whose coordinate  $(p, f)_{\text{th}}$  is around  $(0.755, 10.5) \cdot 10^{-2}$  for UF adapted with the forward method, and  $(0.735, 6.8) \cdot 10^{-2}$  for *Snowflake*. Here, the standard error is smaller than the line thickness.

We design *Snowflake*, a generalisation of the local decoders in [17] from the batch to the streaming case. *Snowflake* comprises a simple set of local rules with very little central management, and works for any error-correcting code with a decoding graph embeddable in  $\mathbb{R}^3$  without ‘long-range’ edges. Examples include the repetition code and the surface code under circuit-level noise. The pseudocode for *Snowflake* is summarised in [Algorithm 5](#).

*Snowflake* relies on the new *frugal method* we develop for general stream decoding. This is similar to the forward method, but differs in that it discards none of the computation – it is more frugal with resources. This halves both the power draw and the size of the decoder architecture. We hope the proposal of this method enables the development of more streaming decoders.

Figure 1 shows that, when tested on the surface code under circuit-level noise, *Snowflake* recovers  $\approx 97\%$  of the threshold of UF adapted with the for-

ward method [which we abbreviate as FM(UF)], but in absolute terms is 24.9(5)% more accurate. Moreover, Snowflake’s mean runtime scaling with the code distance improves upon original UF. Snowflake has a distributed implementation that is modular and scalable. This is a 3D lattice of processors though we note an alternative architecture that ‘flattens’ this lattice onto a 2D grid would be even more practical given current chip design capabilities, at a potentially marginal cost in speed.

Section 2 covers the prerequisite theory. We describe the frugal method in Section 3, explain Snowflake in Section 4, then numerically test its accuracy and runtime in Section 5. Section 6 concludes. In Figs. 2 to 7 and Figs. 15 to 17, the time axis points upward. Our emulation code is on GitHub at [44].

## 2 Background

Sections 2.1 and 2.2 introduce widely used formalisms for modelling noisy circuits. These two subsections can be skimmed for readers familiar with error correction. Section 2.3 recaps the batch decoding problem. In Section 2.4 we generalise to the streaming case and in Section 2.5 we provide one solution that already exists in literature. Throughout the paper, we denote the node and edge sets of any graph  $\Gamma$  as  $V_\Gamma$  and  $E_\Gamma$ , respectively.

### 2.1 The Detector Error Model

We can model noise affecting the quantum error-correction code using a *detector error model* (DEM) [45, 46]. Constructing a DEM starts with defining a set of *detectors* and *logical observables*, each of which represents a set of measurements whose parity is deterministic in the absence of noise; e.g. a detector could represent two consecutive measurements of the same stabiliser at a given point in time. Noise is then modelled by a set of independent events called *error mechanisms*, each of which flips a unique subset of detectors and logical observables. For stabiliser measurement under the phenomenological noise model, error mechanisms either a data qubit bitflip or a measurement outcome flip; under a more realistic circuit-level noise model, they represent sets of error events within the physical circuit. Appendix A describes the circuit-level noise model used in our numerics.

A DEM of  $m$  detectors,  $l$  logical observables, and  $n$  error mechanisms is specified by a triple  $(H, L, \mathbf{p})$ , where the matrix  $H \in \mathbb{F}_2^{m \times n}$  ( $L \in \mathbb{F}_2^{l \times n}$ ) defines which error mechanism flips which detector (logical observable), and the vector  $\mathbf{p} \in ]0, 1]^n$  defines the probability each error mechanism occurs; this is usually a function of some universal noise level  $p$ .

In a decoding cycle, the error mechanisms occur with probability according to  $\mathbf{p}$ ; the occurred set is

represented by a vector  $\mathbf{e} \in \mathbb{F}_2^n$ . This flips a set of detectors and logical observables indicated by the support of  $\mathbf{s} = H\mathbf{e}$  and  $\boldsymbol{\lambda} = L\mathbf{e}$ , respectively. The task of the decoder is to infer  $\boldsymbol{\lambda}$  given  $\mathbf{s}$ . More generally, we will use the following definition.

**Definition 1.** A *decoding cycle* is one iteration of the process from when the decoder receives a new (or updated)  $\mathbf{s}$  to when it outputs a new (or updated)  $\boldsymbol{\lambda}$ .

In this paper we focus on the  $l = 1$  case; specifically, the Z memory experiment i.e. inferring whether the  $\hat{Z}$  observable of one logical qubit has flipped after some number of stabiliser measurement rounds. We consider decoders that output a specific correction vector  $\mathbf{c} \in \mathbb{F}_2^n$  satisfying

$$H\mathbf{c} = \mathbf{s}; \quad (1)$$

the  $\boldsymbol{\lambda}$  inferred is then  $\hat{\boldsymbol{\lambda}} = L\mathbf{c} \in \mathbb{F}_2^1$ .

### 2.2 The Decoding Graph

Additionally, we focus on DEMs in which each error mechanism flips at most two detectors (or DEMs that can be approximated as such). In this special case, we can construct a *decoding graph*  $G$  from the DEM. We start by representing each detector as a node. Then for each error mechanism: if it flips exactly two detectors  $\{u, v\}$ , we add an edge  $uv$ ; if it flips exactly one detector  $u$ , we add an edge between  $u$  and a dedicated *boundary node* which represents nothing physical, but exists solely to allow the error mechanism to be represented by an edge.

Thus,  $V_G$  is partitioned into two sets: *detectors*  $V_d$  and *boundary nodes*  $V_b$ . Each node is assigned a unique spacetime coordinate such that for codes with local stabilisers (e.g. the surface code), there are no ‘long-range’ edges. This locality is useful for local decoders that process information between nearby detectors, like Snowflake.  $V_b$  is further partitioned into the *west boundary* and the *east boundary* sets, chosen such that an edge flips the logical observable iff it is incident to the west boundary.

Since  $e \in E_G$  represents an error mechanism which can occur or not, we assign  $e$  a bit value  $x \in \{0, 1\}$  and may refer to it as a *bit value- $x$  edge*. The bit value  $x$  is not known to the decoder, but it *can* flip  $x$ : we call this *flipping the edge*. Of course, the decoder can keep track of which edges it has flipped. A *defect* is a detector incident to an odd number of bit value-1 edges. The *syndrome*  $\mathbb{S} \subseteq V_d$  is the set of defects. The decoder can modify  $\mathbb{S}$ : flipping an edge  $uv$  updates  $\mathbb{S} \leftarrow \mathbb{S} \Delta \{u, v\}$  where  $\Delta$  denotes symmetric difference.

**Definition 2.** In the case  $u \in \mathbb{S}$  and  $v \notin \mathbb{S}$ , we say flipping  $uv$  will *push* the defect at  $u$  along  $uv$  to  $v$ . In the case  $u, v \in \mathbb{S}$ , we say flipping  $uv$  will *annihilate* the defect at  $u$  with the defect at  $v$ . We extend this to the case of a path  $P$  of edges connecting a defect at  $s$  to another defect at  $t$  (or to a boundary node): flipping

the edges in  $P$  will push the defect at  $s$  along  $P$  to annihilate with the defect at  $t$  (or with the boundary node).

**Definition 3.** A set  $F \subseteq E_G$  *annihilates* syndrome  $\mathbb{S}$  if  $F$  constitutes paths that pair each defect with either another defect or a boundary node.

### 2.3 The Batch Decoding Problem

In terms of the decoding graph, the decoding cycle is as follows. Each edge is assigned bit value 1 with probability according to  $\mathbf{p}$ . Knowing only  $\mathbb{S}$ , the decoder must flip edges until  $\mathbb{S} = \emptyset$ .

**Definition 4.** The *correction*  $\mathbb{C} \subseteq E_G$  is the final set of edges the decoder chooses to flip.

Flipping edges until  $\mathbb{S} = \emptyset$  means  $\mathbb{C}$  must annihilate  $\mathbb{S}$ ; this is equivalent to Eq. (1), where  $\mathbb{S}$  and  $\mathbb{C}$  are analogous to  $\mathbf{s}$  and  $\mathbf{c}$ . After applying  $\mathbb{C}$ , the remaining bit value-1 edges can only constitute either cycles, or paths that end at boundaries. Such a path will flip the logical observable if it ends at the west boundary exactly once; this motivates the following.

**Definition 5.** A *logical flip* is path of bit value-1 edges from the west to the east boundary. The *logical error rate* is the logical flip count per  $d$  measurement rounds [47, §VI], where  $d$  is the code distance.

The shortest logical flip has  $d$  edges. We evaluate decoder accuracy by calculating the logical error rate.

### 2.4 The Stream Decoding Problem

We now generalise the batch decoding problem to the streaming case, relevant for very long memory experiments. Figure 2a,b illustrates the following definitions: a *sheet* is a subset of  $V_G$  all with the same  $t$  coordinate, and a *layer* is the periodic unit subgraph of  $G$ . For the stream decoding problem, there are an *indefinite* number of measurement rounds; this means new layers are constantly being added to the top of  $G$  as in Fig. 2c,d. Note the addition of a *future boundary*: a third set of boundary nodes representing detectors whose parities are yet to be determined. When a new layer is added, its detectors replace the future boundary of the previous layer. In general, some edges in the new layer are assigned bit value 1 (specifically, the probability that  $x = 1$  for each edge equals that of the analogous edge in the layer below), creating defects in the new layer. The decoder must constantly add to  $\mathbb{C}$  to annihilate these new defects.

At the end of the Z memory experiment, all data qubits are destructively measured in the Z basis. This corresponds the final layer that is added onto  $G$ , and it has no future boundary. Since the final form of  $G$  has only a west and east boundary, we can once again evaluate decoder accuracy by Definition 5.

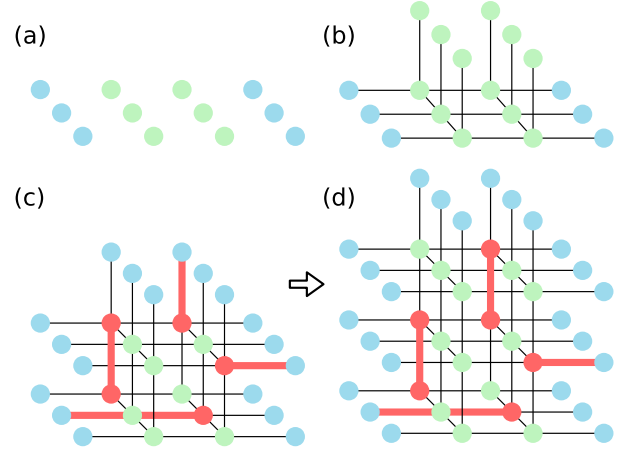


Figure 2: Defects are red nodes; other detectors, green; boundary nodes, blue. Bit value-1 edges are thick and red; bit value-0 edges, thin and black. (a) A sheet and (b) a layer of (c) the decoding graph for the surface code under phenomenological noise. The west, east, and future boundaries are the set of blue nodes on the left, right, and upper faces of the decoding graph, respectively. (d) How said graph changes after one measurement round: one layer is added on top. This is the streaming generalisation of [17, Figure 3b].

For clarity, all figures in the rest of this paper, save Figs. 7 to 14, use a planar  $G$  corresponding to the repetition code under phenomenological noise. However, the same concepts apply to other graphs such as for the surface code under circuit-level noise. In the next subsection we discuss an existing approach to solve the stream decoding problem.

### 2.5 The Forward Method

This method was first proposed in [41, §VI.B] as the ‘overlapping recovery method’ but a more detailed explanation can be found in [42, pp 2–3] where it is called ‘sliding window decoding’. It has been applied to the general class of *quantum low-density parity-check codes* (LDPC, which includes the repetition and surface codes) [48–51]. It reduces the stream decoding problem to batch decoding by chunking the syndrome, decoding each chunk, then stitching together all the corrections. The chunks must overlap, else a small defect pair could straddle two consecutive chunks hence be decoded inaccurately.

More concretely, assume  $G$  already comprises many layers, and define the following.

**Definition 6.** The *decoding window* is at any given time a graph  $W = (C \cup B) \subseteq G$  comprising  $c + b$  consecutive layers of  $G$ , wherein the highest sheet of detectors is replaced by a future boundary. The lowest  $c$  layers of  $W$  is the *commit region*  $C$ ; the highest  $b$  layers of  $W$  is the *buffer region*  $B$ .

**Definition 7.** The *tentative correction*  $\mathbb{T} \subseteq E_W$  is the decoder’s output for a given decoding window  $W$ .

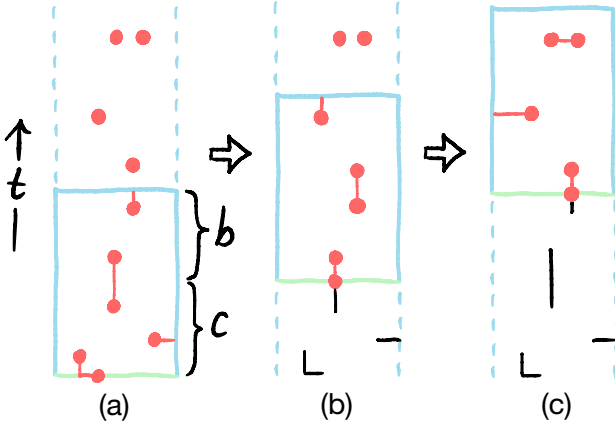


Figure 3: Two decoding cycles of the forward method. The *decoding window* (Definition 6) is the solidly outlined rectangle with boundaries in blue. Defects are red dots. The *tentative correction* (Definition 7) comprises the red lines; edges flipped by the decoder, black lines. The commit and buffer region heights are  $c$  and  $b$ , respectively. (a) The tentative correction contains a path that spans both regions. (b) This path is partially committed which makes an ‘artificial defect’ at the bottom of the new window that must also be annihilated. (c) The rest of the path is committed.

This differs from Definition 4 in that  $\mathbb{T}$  may not be final. We say we *commit* edges from  $\mathbb{T}$  when we transfer them to  $\mathbb{C}$ .

The forward method is illustrated in Fig. 3.  $W$  starts at the bottom, i.e. as the lowest  $c+b$  layers, of  $G$ , and  $\mathbb{C} = \emptyset$  initially. The method then indefinitely repeats the following (skipping step 1 in the first iteration).

---

**Algorithm 1** The forward method decoding cycle.

---

- 1: Raise  $W$  by  $c$  layers i.e. redefine it as the subgraph of  $G$  that is  $c$  layers higher.
  - 2: Use any batch decoder to find a tentative correction  $\mathbb{T}$  that annihilates  $\mathbb{S} \cap V_W$ .
  - 3: Update  $\mathbb{C} \leftarrow \mathbb{C} \cup (\mathbb{T} \cap E_C)$  and forget  $\mathbb{T} \cap E_B$ .
- 

*Remark 1.* Since  $b$  ‘layers of computation’ are forgotten per decoding cycle, decoding  $\tau$  sheets of syndrome data discards a total of  $\approx b\tau/c$  layers of computation.

*Remark 2.* Assuming all probabilities in  $\mathbf{p}$  are of the same order, we need the window overlap  $b \gtrsim d$  to ensure no loss in accuracy [52, §IV.D]. For  $c$  on the other hand, there is no such lower limit imposed i.e. we could set  $c = 1$  with no loss in accuracy (so long as  $b \gtrsim d$ ). However, there are other reasons why  $c$  cannot be too small or large: if too small, then many layers of computation are discarded; if too large, then  $W$  becomes too tall leading to slow decoding cycles or in the case of a lookup-table decoder, high memory requirements.

The most common choice thus is  $c = b = d$  though others have been explored e.g.  $0 \leq b < d$  [3, §3] [43, §B.2] [52, §VII.C] [53, §5.2] or  $c = 1$  [54, §5.1] [49, 50].

In almost all cases, a considerable amount of computation is discarded. This is true for the sandwich method too as it also requires overlapping windows.

*Remark 3.* The future boundary in  $W$  improves decoding accuracy [43, §B.3] compared to if it were absent, as it allows the decoder to account for small defect pairs that straddle this boundary like the one in Fig. 3a. By tentatively pairing the known defect to the future boundary, the decoder delays the decision on how to properly annihilate it until it discovers the other defect just above in Fig. 3b. The same occurs with another defect near the top of this window, but this time it turns out there are no nearby defects above, so it is instead paired to the west boundary in Fig. 3c.

*Remark 4.* In *hard* real-time decoding, the decoder is given a hard time limit for each decoding cycle, which should be set to no longer than  $c$  ancilla qubit measurement intervals. If it exceeds this limit, it is interrupted and the logical qubit is assumed to fail. On the other hand, *soft* real-time decoding demands less by introducing a buffer (in the conventional sense) between the newest measurement data and the data being decoded, in which case only an *average* decoding cycle must complete faster than  $c$  intervals; any slower then the backlog of syndrome data grows indefinitely [16, p 326]. In practice, decoding will likely be a hard real-time system [19, §2.1].

This remark applies also to the frugal method discussed in the next section. We analyse both individual and mean runtime for Snowflake in Section 5.2.

### 3 The Frugal Method

In this section we describe our alternative to the forward method. The frugal method can be thought of as eagerly raising the decoding window *during* the computation of the correction, meaning it need only be complete within the commit region. This idea is made more precise below.

---

**Algorithm 2** The frugal method decoding cycle.

---

- 1: Raise  $W$  by  $c$  layers, same as in Algorithm 1.
  - 2: Find a tentative correction  $\mathbb{T}$  that satisfies Condition 1. This condition is weaker than that in Algorithm 1.
  - 3: Update  $\mathbb{C} \leftarrow \mathbb{C} \cup (\mathbb{T} \cap E_C)$ , same as in Algorithm 1. The difference is that  $\mathbb{T} \cap E_B$  and the computation (i.e. the internal state of the decoder) associated with  $B$  is not forgotten but used as a starting point for, and further developed in, the next decoding cycle.
- 

**Condition 1.**  $\mathbb{T}$  annihilates  $\mathbb{S} \cap V_C$ .

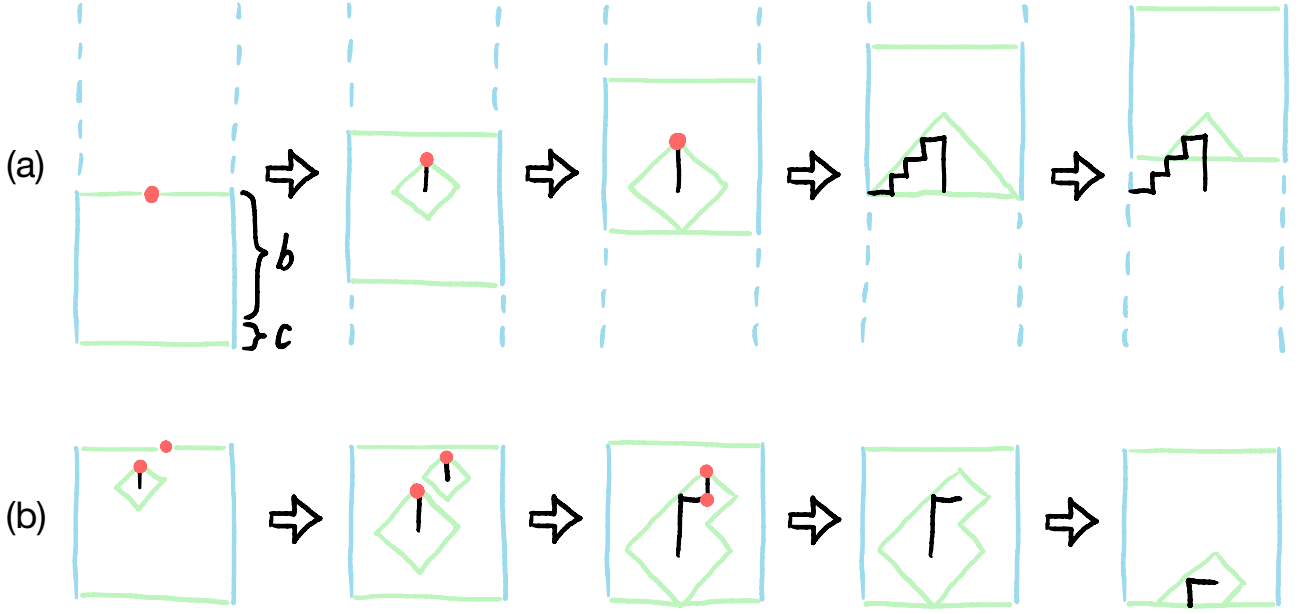


Figure 4: Two examples of how Snowflake finds its tentative correction. Features are shown as in Fig. 3 and additionally, clusters (Definition 8) are outlined in green. (a) A cluster grows around one defect as the decoding window is raised, while the defect is always pushed (Definition 2) to the highest point. As soon as the cluster touches a boundary, the defect is instead pushed to that boundary (in this case, the cluster touches the west and east boundaries at the same time and arbitrarily chooses to push westward). This inactivates (Definition 9) the cluster so it stops growing. It gradually disappears from view. The defect's path is staircase-like as it comprises edges from the decoding graph. (b) Two defects separated in time and space (this time viewed from the perspective of the window so clusters appear to fall). The lower cluster is older, hence larger, when both clusters merge. Frame 3 shows the lower defect on its way to the highest point to annihilate with the other defect, inactivating the resultant cluster.

*Remark 5.* The reuse of computation in step 3 means zero layers of decoder computation are discarded, regardless of  $b$  or  $c$  (c.f. Remark 1).

As computation is never discarded, the power draw and thus heat generated by the device is minimised. This is vital for an on-chip, local decoder integrated with the qubits, as most qubit platforms demand low temperatures. Compared to the forward method for  $c = b$ , where 1 layer of computation is discarded per sheet of syndrome data, the power draw is halved.

Given Remark 5, for the rest of the paper we set  $c = 1$  in the frugal method. This has no effect on the decoding accuracy, but it means the method becomes local (provided the decoder itself is local) as, from the perspective of the decoding window, data shifts down by one layer when  $W$  is raised. The commit region  $C$  is also one layer thick, so  $C \cap E_C$  can be offloaded from the bottom of the decoding window locally. Conceptually, this makes the method much more granular in nature.

We note the frugal method is similar to another recently proposed stream decoding method called *fusion* [55, §IV]. Further, the special case of the frugal method with  $c = 1$  is very similar to a special case of fusion called *round-wise fusion* [56, §6], in which the decoder updates its solution every time a new sheet of syndrome data arrives. Like the frugal method, fusion also avoids redundant computation, but does so

by dividing the decoding graph into windows that do not overlap, then relying on the properties of MWPM to guarantee no loss in accuracy when combining solutions from neighbouring windows. Since fusion is designed specifically for use with MWPM, it is unclear if and how much accuracy is lost when used with UF.

## 4 Snowflake

In this section, we present our decoder. Section 4.1 describes Snowflake's main idea of modifying UF for the gradual window-raising of the frugal method. Section 4.2 discusses choices in cluster growth that drastically affect decoding accuracy. Snowflake uses a modified decoding window *without* a future boundary (so in Definition 6, the highest sheet of detectors and its incident edges is simply absent); this will be explained in Section 4.3. We conclude in Section 4.4 by describing a distributed implementation of our decoder.

In Snowflake, we imagine flipping edges (thus altering  $S$ ) during the computation of  $T$  in Algorithm 2 step 2. Snowflake then performs steps 1 and 3 simultaneously, committing said flips in  $E_C$  as  $W$  is raised. This has no physical consequence on the qubits, as  $C$  is ultimately only tracked by classical electronics in the so-called *Pauli frame* [57, 58].

## 4.1 Main Idea

The algorithm is inspired by UF and explained by analogy to snowfall. As snowflakes fall they gradually grow. If two snowflakes touch they merge i.e. become one. In the same way, an active cluster nucleates at each new defect at the top of the decoding window and grows as it falls. If two clusters touch they merge and become one. Figure 4 illustrates these ideas, and below we make them more precise.

**Definition 8.** A *cluster*  $K$  is a connected subgraph of  $W$  with the additional property that edges can be half-included in it. Connectivity is defined by the fully included edges. It *touches* another cluster  $L$  (or a boundary  $V_b$ ) if  $V_K \cap V_L \neq \emptyset$  (or  $V_K \cap V_b \neq \emptyset$ ).

Each edge  $e \in E_W$  has a *growth value*,  $e.\text{growth} \in \{0, \frac{1}{2}, 1\}$  indicating how much of it is included in *any* cluster. When a cluster grows, it does so in all possible directions (defined by the edges of  $G$ ) by half an edge; this is concretely achieved through Algorithm 3 GROW.

---

**Algorithm 3** Cluster procedures in Snowflake.

---

```

procedure GROW( $K$ )
2:   for all  $e \in E_W : e$  is incident to  $V_K$  do
   |    $e.\text{growth} \leftarrow \min\{1, e.\text{growth} + \frac{1}{2}\}$ 

4: procedure MERGE( $K$ )
   | if  $V_K$  has a boundary node  $v$  then
6:   |    $\text{root} := v$ 
   | else
8:   |    $\text{root} := v \in V_K$  with highest  $t$  coordinate
   |   for each defect  $v$  in  $K$  do
10:  |    $P :=$  a path in  $K$  from  $v$  to  $\text{root}$ 
   |   flip the edges in  $P$ 

```

---

When two clusters become connected by a fully grown edge, they become a single cluster by Definition 8.

**Definition 9.** A cluster is *inactive* iff all its defects can be annihilated by flipping some subset of its fully included edges; else, it is *active*.

Active clusters will eventually inactivate if they grow large enough; this is the general idea of UF: to minimally grow each active cluster until it inactivates. Note the following lemma from [17, §A.2].

**Lemma 1.** A cluster is inactive iff it has an even defect count or touches a boundary.

Snowflake determines the first condition by pushing all defects to a unique point in the cluster called the *root*, determined in Algorithm 3 MERGE. When two defects meet they annihilate, so eventually zero or one defect will remain at the root, indicating the cluster's defect count parity. In Line 11, defects are pushed to the root by flipping edges, as per Definition 2. In this way,  $\mathbb{T}$  is the symmetric difference of all paths traced

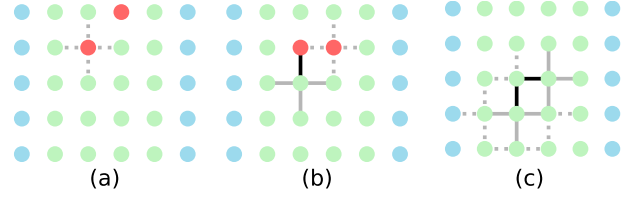


Figure 5: A concrete depiction of Fig. 4b for a distance-5 code to show how the 1:1 cluster growth schedule violates Lemma 2. Ungrown edges are invisible; half-grown, dotted; fully grown, solid. Edges flipped by the decoder are in black.

by defects and moreover is guaranteed to annihilate the defects in any inactive cluster. The idea then is to inactivate all clusters once they have fallen far enough. Note it is not necessary for the path a defect takes to reach the root to be of minimum length.

## 4.2 Cluster Growth Schedules

Here we consider two different *cluster growth schedules* i.e. rules on how active clusters grow. Section 4.2.1 introduces the simplest possible one, 1:1, and identifies an issue that hinders its accuracy. We fix this in Section 4.2.2 with an improved schedule, 2:1.

### 4.2.1 1:1 Schedule

In the most straightforward schedule, Snowflake performs one growth round per decoding cycle, as the example in Fig. 5 shows. A *growth round* here comprises growing each active cluster once, redefining touching clusters, and pushing defects to roots. Algorithm 4 summarises this.

---

**Algorithm 4** A decoding cycle of Snowflake with the 1:1 schedule, using procedures from Algorithm 3.

---

```

raise  $W$  by 1 layer
for each active cluster  $K$  do
  | GROW( $K$ )
for each cluster  $K$  do
  | MERGE( $K$ )

```

---

To see the issue with this schedule, first note the following lemma for [32, Theorem 1] that is satisfied by original UF.

**Lemma 2.** Whenever a cluster grows, the number of initial bit value-1 edges (i.e. whose bit values are initially 1) covered by the union of all clusters increases by  $\geq 1/2$ .

This relies on the impossibility for two active clusters to be only half an edge apart. The theorem leads to the following for original UF.

**Corollary 1.** Any combination of  $s < d/2$  bit value-1 edges can be corrected.

*Proof.* [Lemma 2](#) guarantees that after  $2s$  growth rounds, all initial bit value-1 edges are covered by the union of all clusters, and no more growth occurs. The largest possible cluster diameter after this many growth rounds is  $2s < d$  so no logical flip is possible.  $\square$

For Snowflake with the 1:1 schedule, it is possible for two active clusters to be only half an edge apart; see e.g. [Fig. 5b](#). Then, during growth, they will overgrow ‘into each other’ rather than stop growing as soon they touch. This violates [Lemma 2](#) and leads to unnecessarily large clusters. E.g. assume the defect pair in [Fig. 5a](#) resulted from  $s = 2$  initial bit value-1 edges; in c, two clusters grow but only  $1/2$  more of the initial bit value-1 edges are covered. The resultant cluster has diameter  $9/2 > s$ .

#### 4.2.2 2:1 Schedule

To fix this, we can grow clusters more cautiously. Note in [Fig. 5b](#) it is enough for just one of the clusters to grow for the two to merge, and the reason they are half an edge apart is that one has grown an even, and the other an odd, number of times. So, define a *whole (half) cluster* as one that has grown an even (odd) number of times, and a *mixed cluster* as one that has resulted from merging a whole with a half cluster. Visually, a whole (half) cluster is, when isolated (always), surrounded by ungrown (half-grown) edges. A mixed cluster, e.g. in [Fig. 5c](#), is always surrounded by both types of edge.

In this schedule, Snowflake performs two different growth rounds per decoding cycle: first growing the whole clusters, then the half clusters, as [Fig. 6](#) shows. After each round, all clusters reevaluate their activity. Mixed clusters need not be considered due to the following.

**Proposition 1.** *Mixed clusters never occur with the 2:1 schedule.*

We prove this in [Appendix B](#). Lastly, for balance, whole clusters that grow in the first round (thus becoming half clusters) do not grow again in the same decoding cycle. [Algorithm 5](#) summarises this.

**Algorithm 5** A decoding cycle of Snowflake with the 2:1 schedule, using procedures from [Algorithm 3](#).

---

```

raise  $W$  by 1 layer
2: for each active whole cluster  $K$  do
    | GROW( $K$ )
4: for each cluster  $K$  do
    | MERGE( $K$ )
6: for each active half cluster  $K$  do
    | if no part of  $K$  grew in Line 3 then
8: | | GROW( $K$ )
    | for each cluster  $K$  do
10: | | MERGE( $K$ )

```

---

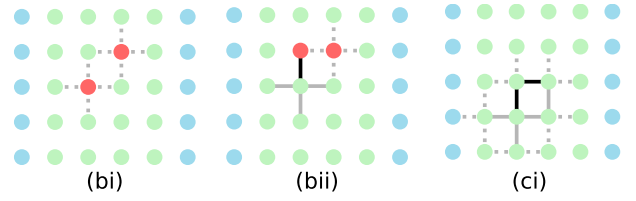


Figure 6: Continuing from [Fig. 5a](#) but with the 2:1 cluster growth schedule, which shows [Lemma 2](#) is no longer violated. ‘i’ labels the first growth round of the decoding cycle; ‘ii’, the second.

The 2:1 schedule satisfies [Lemma 2](#) as it prevents the overgrowth issue: if two active clusters are half an edge apart, only one of them will grow. E.g. in [Fig. 6ci](#), only the whole cluster (lower west) grows and the resultant half cluster has diameter  $4 = 2s$ . Our numerics in [Appendix C](#) show the 2:1 schedule almost doubles the threshold of 1:1, so for the rest of this paper, we consider Snowflake with the 2:1 schedule.

### 4.3 Constructing the Decoding Window

We now discuss the boundaries and required height of Snowflake’s decoding window. Despite [Remark 3](#), Snowflake uses a modified decoding window without a future boundary because in either schedule, clusters fall twice as fast as they grow so this boundary would never be touched.

The height of the buffer region is lower-bounded by the following.

**Proposition 2.** *The minimum layer count of the buffer region that guarantees [Condition 1](#), is  $b_{\min} := 2\lfloor d/2 \rfloor$ .*

*Proof.* Defects are annihilated by either meeting another defect or being pushed to a boundary. So a longest-living defect is one like that in [Fig. 4a](#): never meeting another defect, and created as far as possible, i.e.  $\lfloor d/2 \rfloor$  edges, from any boundary. The cluster containing this defect grows outward by half an edge per decoding cycle so will touch a boundary after  $b_{\min}$  decoding cycles, at the  $(b_{\min} + 1)^{\text{th}}$  layer from the top. So  $\mathbb{T}$  always annihilates all defects in this layer and below, but not above.  $\square$

We set  $b = b_{\min}$  for our distributed implementation. Minimising  $c$  and  $b$  in the frugal method roughly halves the height of  $W$  compared to the forward method for  $c = b = d$ . As [Fig. 7](#) shows, this more than halves the processor count of our implementation described in the next subsection.

### 4.4 Distributed Implementation

Here we describe how Snowflake can run on a distributed implementation. We use the same local architecture of Macar and Actis in [\[17\]](#): a processor for each node in  $V_W$ , a communication link for each edge

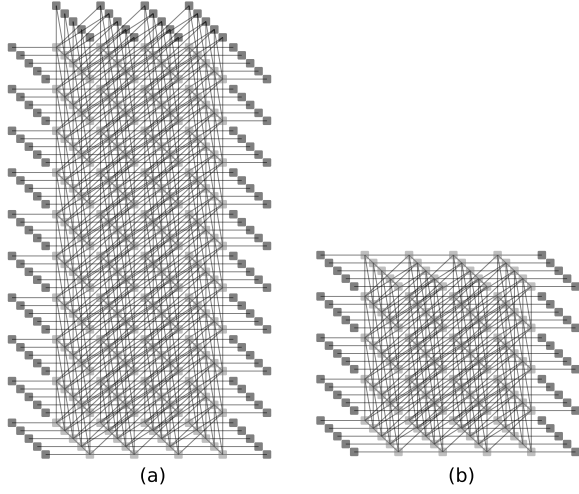


Figure 7: Distributed implementation of the decoders benchmarked in Section 5; the two shown here are for the distance-5 unrotated surface code. Each square is a classical processor; each line is a communication link. Darker squares correspond to boundary nodes in the decoding graph. (a) Macar [17] adapted with the forward method. (b) Snowflake; note the absence of a future boundary.

and one controller which manages the global variables of the algorithm. For the surface code, the processors and links form a 3D lattice oriented so the time axis points upward, as in Fig. 7b. Our decoder is pipelined as successively lower layers in the lattice process increasingly mature stages of decoding.

Snowflake is compatible with any choice of network for controller–processor communication, be it the hierarchical tree used in Macar or the strictly local staging and signalling tree in Actis, with synchronous or asynchronous logic. We thus leave this as an implementation choice.

To rudimentarily measure the speed of our implementation, we consider a simple computational model similar to cellular automata. Each component (i.e. controller or processor) updates its state based on its previous state and that of its neighbours. These updates occur synchronously across the whole implementation, allowing us to define the following.

**Definition 10.** A *timestep* is the duration between consecutive updates of a component’s state.

The clock cycle of a hardware implementation would roughly be proportional to a timestep. Definition 10 limits how fast information propagates through the 3D lattice: messages travel one communication link per timestep. For example, Algorithm 3 GROW lasts 1 timestep, since information need only travel from a node to its incident edges. On the other hand, MERGE lasts a variable number of timesteps dependent on the diameter of the cluster, because information must propagate through the whole cluster to establish its root, and to push defects to this root. Appendix D

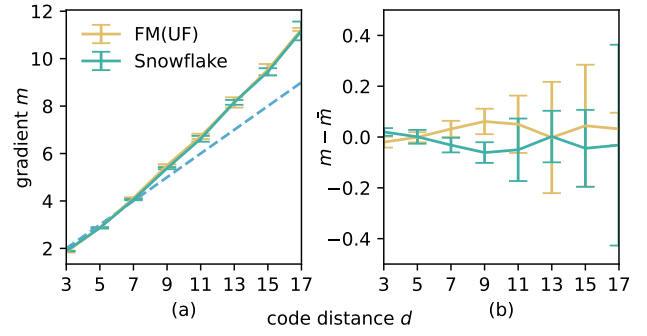


Figure 8: Errorbars show standard error. (a) Gradient  $m$  of the lines in the  $p \leq 6 \cdot 10^{-3}$  region of Fig. 1, estimated using Weighted Least Squares. The dashed blue line shows the asymptotic ( $p \rightarrow 0$ ) value  $m = (d + 1)/2$  from theory [59, §5]. (b) Deviation of said gradient from the mean  $\bar{m} := \frac{1}{2}(m_{\text{FM(UF)}} + m_{\text{Snowflake}})$  of both decoders.

provides the rest of the details of our distributed implementation.

## 5 Numerics

Here, we numerically analyse the accuracy, throughput, and latency of Snowflake in Sections 5.1, 5.2 and 5.3, respectively. We wrote a Python package [44] to emulate the distributed implementation of Snowflake that follows the computational model from the previous subsection. We use it to run the Z memory experiment using the surface code under the circuit-level noise model detailed in Appendix A. Specifically, we decode the syndrome from  $n \gg d$  stabiliser measurement rounds, then count one of two features depending on if we are benchmarking accuracy or throughput. To benchmark latency, we emulate and decode the end of the memory experiment.

For comparison we also benchmark FM(UF). We fix its decoding window to  $c = b = d$  as we found that smaller  $c, b$  values led to both worse accuracies and overall runtimes, consistent with Remark 2. The distributed implementation of FM(UF) i.e. FM(Macar), and of Snowflake, is shown in Fig. 7. For both, the controller–processor communication time cost is the same, so we exclude it by assuming the controller connects directly to each processor.

### 5.1 Accuracy

We evaluate accuracy by Definition 5: after decoding, we count the number  $l$  of logical flips, identifying such paths using the method described in [60]. We then estimate the logical error rate  $\hat{f} = ld/n$ .

Figure 1 compares the accuracy of FM(UF) and Snowflake, for practical noise levels and code distances. Below threshold, we assume the power law  $f/f_{\text{th}} = (p/p_{\text{th}})^m$ . Figure 8 shows the subthreshold gradients  $m(d)$  are indistinguishable for the two de-

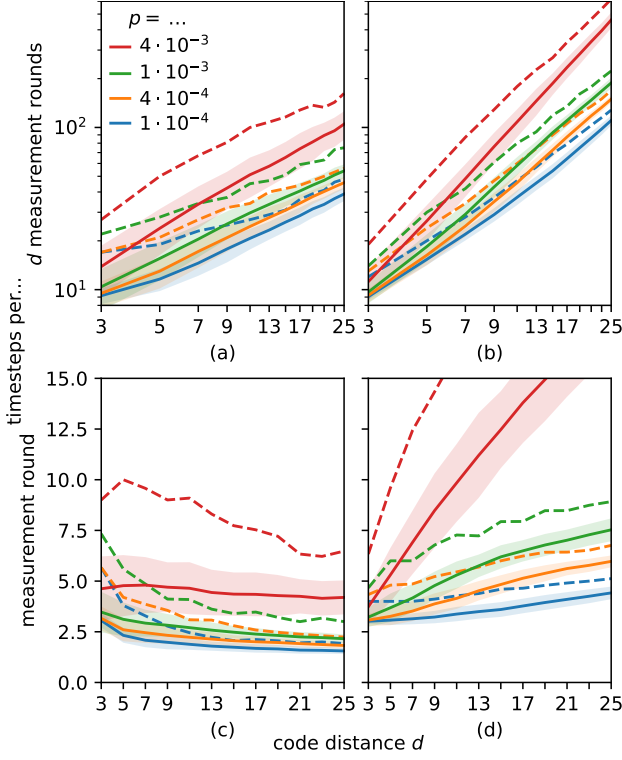


Figure 9: Decoder throughput against surface code distance  $d$ , for various circuit-level noise levels  $p$  below threshold. A *timestep* is defined in Definition 10. Each solid (dashed) line shows the mean (99<sup>th</sup> percentile) of  $10^3$  lots of  $d$  measurement rounds; shading shows standard deviation. The standard error of the mean is smaller than the solid line thickness. (a) Macar [17] adapted with the forward method. (b) Snowflake. (c,d) Same as a,b but rescaled.

coders so we can characterise their accuracy difference solely by their thresholds.

While FM(UF) has a slightly higher threshold, Snowflake leads to a logical qubit lifetime  $1/f$  that is 24.9(5)% longer on average over the Fig. 1 data. We are unsure why Snowflake is more accurate but suspect it may relate to its smoother treatment of  $G$ . The most obvious difference between the outputs of UF and Snowflake is in the relative cluster sizes e.g. were UF used in Fig. 6, both clusters would be the same size when they merge. In Appendix E we rule out another possible explanation related to how each decoder resolves a cluster touching both boundaries.

## 5.2 Throughput

For this we count the number of timesteps (Definition 10) to decode. We consider this per  $d$  measurement rounds as each logical operation lasts  $\mathcal{O}(d)$  rounds using lattice surgery [61], the best known technique for universal computation with the surface code when the quantum device has local connectivity [62, 63]. We also consider the timestep count *per stabiliser measurement round* because for real-time decoding, the decoder’s throughput requirement is im-

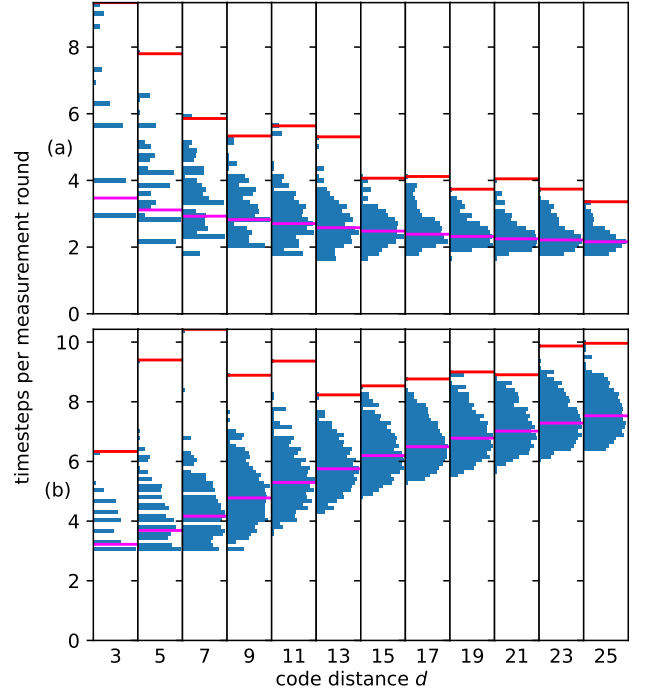


Figure 10: A more detailed view of the  $p = 10^{-3}$  throughput data in Fig. 9c and d, respectively. Within each histogram: the horizontal magenta (red) line shows the mean (maximum) of  $10^3$  lots of  $d$  measurement rounds, and the  $x$ -axis scale is logarithmic.

posed by the quantum device’s stabiliser measurement round duration [19, §1] e.g.  $\approx 1 \mu\text{s}$  for superconducting qubits [64, Figure 1] [65, p 924].

Figure 9 compares the runtime of FM(Macar) and Snowflake. To adapt Macar with the forward method, we measure its runtime to decode a window, then add  $c$  timesteps due to step 1 of the forward method. Using this data, Table 1 estimates the scaling  $m$ , assuming mean runtime  $\propto d^m$ .

Table 1: Gradient  $m$  of the solid lines in Fig. 9a,b estimated using Weighted Least Squares.

| $p$               | FM(Macar) | Snowflake |
|-------------------|-----------|-----------|
| $4 \cdot 10^{-3}$ | 0.94(1)   | 1.770(4)  |
| $10^{-3}$         | 0.778(8)  | 1.42(1)   |
| $4 \cdot 10^{-4}$ | 0.761(7)  | 1.34(2)   |
| $10^{-4}$         | 0.69(2)   | 1.16(2)   |

It shows that FM(Macar)’s scaling is sublinear ( $m < 1$ ) whereas Snowflake’s is subquadratic ( $m < 2$ ).

*Remark 6.* We suspect this difference in mean runtime scaling is because for FM(Macar), active clusters in the same batch all grow at the same time regardless of their vertical coordinate. In Snowflake, this process is somewhat serialised: older active clusters start growing before newer ones do, as they enter  $W$  earlier; see e.g. Fig. 4b.

Despite this discrepancy, subquadratic is still better

than the slightly-higher-than-cubic scaling [40, §II.B] of original UF.

To investigate the practical regime further, Fig. 10 plots the distribution of runtimes for both decoders at a practical noise level of  $10^{-3}$ , where Snowflake is  $\approx 0.9$ – $3.5$  times slower than FM(Macar). Neither distribution has a long tail; such consistent runtimes are beneficial for hard real-time decoding: the time limit can be set lower for a given target timeout failure rate, accelerating the quantum program and increasing qubit coherence. Note that Snowflake uses fewer than half the number of processors compared to FM(Macar) (see Fig. 7). Snowflake may therefore be advantageous in architectures where decoder footprint is more of a limitation than decoding throughput.

### 5.3 Latency

Latency is the duration between the decoder receiving the final sheet of syndrome data (corresponding to the end of the memory experiment) to when the decoder commits its final tentative correction. Low latency is crucial for gate teleportation, where the correction to the logical observable can no longer be tracked in the Pauli frame [66].

Figure 11 plots the distribution of latencies for both decoders at a practical noise level of  $10^{-3}$ . As before, we adapt Macar with the forward method by measuring its runtime to decode a window  $W$  but this time add  $b + c$  timesteps because the final commit region comprises all  $b + c$  layers of  $W$ . The figure shows that Snowflake has a lower latency than FM(Macar); this is mainly due to the height  $d$  of Snowflake’s decoding window, which is half that of FM(Macar).

## 6 Conclusion

We propose the frugal method for stream decoding of quantum error-correcting codes for which there exists a decoding graph embeddable in  $\mathbb{R}^3$  without ‘long-range’ edges. As [50, §V] points out, this paradigm of reusing decoder information from previous windows has been explored already for *classical* LDPC codes e.g. in [67, §III.B.2]. We also design Snowflake, a simple decoder compatible with the frugal method and numerically benchmark its distributed implementation. It is more accurate than original UF and its runtime scales more favourably. There are various avenues of further investigation for the frugal method and Snowflake:

**2D Architecture** The serialisation in the vertical direction mentioned in Remark 6 suggests the viability of an alternative architecture: ‘flattening’ the 3D lattice down onto a 2D grid so there is one processor for each *set of nodes* of the same  $(x, y)$  coordinate i.e. each ancilla qubit. This comes at the cost of serialising the operations within each set, but if these opera-

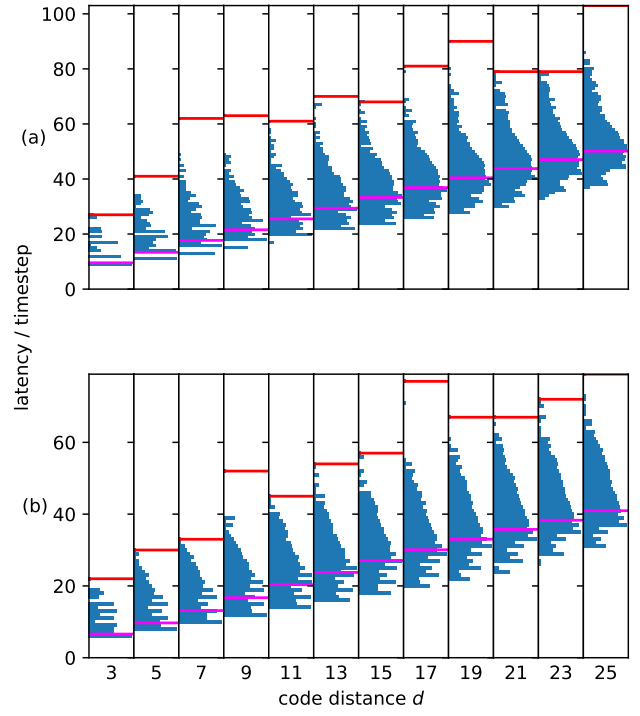


Figure 11: Decoder latency against surface code distance for circuit-level noise level  $10^{-3}$ . A *timestep* is defined in Definition 10. Within each histogram: the horizontal magenta (red) line shows the mean (maximum) of  $10^4$  shots, and the  $x$ -axis scale is logarithmic. (a) Macar [17] adapted with the forward method. (b) Snowflake.

tions are inherently serial, the cost could be marginal. There are two benefits: first, it is more practical as a 2D chip is far easier to build than a 3D lattice of processors; second, messages passed vertically would be instantaneous within a timestep. Future work would involve emulation and numerical analysis of this flattened implementation.

**Asynchronous Logic** As discussed in [17, §6], controller–processor communication could run by asynchronous (instead of synchronous) logic, which would speed up Snowflake. One could explore which other processes in Snowflake could be sped up by asynchronous logic.

**Weighted Edges** Clusters could grow along edges at different speeds to account for their different probabilities in  $\mathbf{p}$ , which would improve accuracy [21]. These speeds could even be configurable in the hardware to fine-tune the noise model during a calibration stage; this however would alter Proposition 2, requiring  $b_{\min}$  to also be configurable, which is nontrivial.

**Computation** Recently, the sandwich method has been generalised from quantum memory to lattice surgery-based computation [52, 53, 68]. We propose an alternative approach to decoding lattice surgery that exploits the granular and local nature

of Snowflake. Imagine the decoder as a homogeneous substrate positioned just above the entire 2D lattice of qubits. Regions of this substrate are turned on (to decode) wherever the below qubits are involved in surgery. Snowflake would work best with so-called *twist-free* lattice surgery [69, §IV] which minimises the use of nonlocal stabilisers. These elongated stabilisers introduce in  $G$  additional detectors [70, Appendix A.3] and accounting for them in our scheme would be nontrivial. Some details of the distributed implementation in Appendix D.1 would need to be generalised. E.g. since boundaries vary during lattice surgery, the ID of each processor would no longer be a constant integer; rather, a combination of a constant 3D coordinate and a (relative) variable boolean indicating whether the processor represents a boundary node. This preserves Remark 7. For controller–processor communication, a hierarchical tree like that used in Macar would be best, as communication cost would then scale logarithmically with the length of the qubit lattice, which may be much larger than  $d$ .

## Acknowledgments

I thank Simon Benjamin, Matt Cassie, and Armands Strikis for useful discussions, and Luka Skoric, Alexandru Paler, and Timo Hillmann for helpful comments. I acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility [71] in carrying out this work and specifically the facilities made available from the EPSRC QCS Hub grant (agreement No. EP/T001062/1). I also acknowledge support from an EPSRC DTP studentship, two EPSRC projects RoaRQ (EP/W032635/1) and SEEQA (EP/Y004655/1), and JST ASPIRE Japan Grant Number JPMJAP2319.

## Competing Interests

The author declares a relevant patent application pending: PCT/EP2025/063485, published as WO2025238191A1 on 2025-11-20.

## References

- [1] D. J. Reilly. “Challenges in scaling-up the control interface of a quantum computer”. In 2019 IEEE International Electron Devices Meeting (IEDM). Pages 31.7.1–31.7.6. (2019).
- [2] Poulami Das, Christopher A. Pattison, Srilatha Manne, Douglas M. Carmean, Krysta M. Svore, Moinuddin Qureshi, and Nicolas Delfosse. “AFS: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers”. In 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). Pages 259–273. (2022).
- [3] Nicolas Delfosse, Andres Paz, Alexander Vaschillo, and Krysta M. Svore. “How to choose a decoder for a fault-tolerant quantum computer? the speed vs accuracy trade-off” (2023). [arXiv:2310.15313](#).
- [4] James William Harrington. “Analysis of quantum error-correcting codes: symplectic lattice codes and toric codes”. *PhD thesis*. California Institute of Technology. (2004).
- [5] Austin G. Fowler. “Minimum weight perfect matching of fault-tolerant topological quantum error correction in average  $O(1)$  parallel time”. *Quantum Information & Computation* **15**, 145–158 (2015).
- [6] Michael Herold, Earl T. Campbell, Jens Eisert, and Michael J. Kastoryano. “Cellular-automaton decoders for topological quantum memories”. *npj Quantum Information* **1**, 15010 (2015).
- [7] Nikolas P. Breuckmann, Kasper Duivenvoorden, Dominik Michels, and Barbara M. Terhal. “Local decoders for the 2D and 4D toric code”. *Quantum Information & Computation* **17**, 181–208 (2017).
- [8] Michael Herold, Michael J. Kastoryano, Earl T. Campbell, and Jens Eisert. “Cellular automaton decoders of topological quantum memories in the fault tolerant setting”. *New Journal of Physics* **19**, 063012 (2017).
- [9] Nicolai Lang and Hans Peter Büchler. “Strictly local one-dimensional topological quantum error correction with symmetry-constrained cellular automata”. *SciPost Physics* **4**, 007 (2018).
- [10] Aleksander Kubica and John Preskill. “Cellular-automaton decoders with provable thresholds for topological codes”. *Physical Review Letters* **123**, 020501 (2019).
- [11] Nicolas Delfosse. “Hierarchical decoding to reduce hardware requirements for quantum computing” (2020). [arXiv:2001.11427](#).
- [12] Samuel C. Smith, Benjamin J. Brown, and Stephen D. Bartlett. “Local predecoder to reduce the bandwidth and latency of quantum error correction”. *Physical Review Applied* **19**, 034050 (2023).
- [13] Ethan Lake. “Fast offline decoding with local message-passing automata” (2025). [arXiv:2506.03266](#).
- [14] Ethan Lake. “Local active error correction from simulated confinement” (2025). [arXiv:2510.08056](#).
- [15] Sam McArdle, Alexander M. Dalzell, Aleksander Kubica, and Fernando G. S. L. Brandão. “The fast for the curious: How to accelerate fault-tolerant quantum applications” (2025). [arXiv:2510.26078](#).
- [16] Barbara M. Terhal. “Quantum error correction for quantum memories”. *Reviews of Modern Physics* **87**, 307–346 (2015).

- [17] Tim Chan and Simon C. Benjamin. “Actis: A strictly local Union-Find decoder”. *Quantum* **7**, 1183 (2023).
- [18] L. M. K. Vandersypen, H. Bluhm, J. S. Clarke, A. S. Dzurak, R. Ishihara, A. Morello, D. J. Reilly, L. R. Schreiber, and M. Veldhorst. “Interfacing spin qubits in quantum dots and donors—hot, dense, and coherent”. *npj Quantum Information* **3**, 34 (2017).
- [19] Francesco Battistel, Christopher Chamberland, Kauser Johar, Ramon W. J. Overwater, Fabio Sebastiano, Luka Skoric, Yosuke Ueno, and Muhammad Usman. “Real-time decoding for fault-tolerant quantum computing: progress, challenges and outlook”. *Nano Futures* **7**, 032003 (2023).
- [20] Muyuan Li, Daniel Miller, Michael Newman, Yukai Wu, and Kenneth R. Brown. “2D compass codes”. *Physical Review X* **9**, 021041 (2019).
- [21] Shilin Huang, Michael Newman, and Kenneth R. Brown. “Fault-tolerant weighted union-find decoding on the toric code”. *Physical Review A* **102**, 012419 (2020).
- [22] Shui Hu. “Quasilinear time decoding algorithm for topological codes with high error threshold”. *Master’s thesis*. Delft University of Technology. (2020).
- [23] Nicolas Delfosse and Matthew B. Hastings. “Union-find decoders for homological product codes”. *Quantum* **5**, 406 (2021).
- [24] Christopher A. Pattison, Michael E. Beverland, Marcus P. da Silva, and Nicolas Delfosse. “Improved quantum error correction using soft information” (2021). [arXiv:2107.13589](#).
- [25] Nicolas Delfosse, Vivien Londe, and Michael E. Beverland. “Toward a union-find decoder for quantum LDPC codes”. *IEEE Transactions on Information Theory* **68**, 3187–3199 (2022).
- [26] Lucas Berent, Lukas Burgholzer, and Robert Wille. “Software tools for decoding quantum low-density parity-check codes”. In Proceedings of the 28th Asia and South Pacific Design Automation Conference. Pages 709–714. ASPDAC ’23 New York, NY, USA (2023). Association for Computing Machinery.
- [27] Ben Barber, Kenton M. Barnes, Tomasz Bialas, Okan Buğdaycı, Earl T. Campbell, Neil I. Gillespie, Kauser Johar, Ram Rajan, Adam W. Richardson, Luka Skoric, Canberk Topal, Mark L. Turner, and Abbas B. Ziad. “A real-time, scalable, fast and resource-efficient decoder for a quantum computer”. *Nature Electronics* **8**, 84–91 (2025).
- [28] Sam J. Griffiths and Dan E. Browne. “Union-find quantum decoding without union-find”. *Physical Review Research* **6**, 013154 (2024).
- [29] Matthias C. Löbl, Susan X. Chen, Stefano Paesani, and Anders S. Sørensen. “Breadth-first graph traversal union-find decoder” (2024). [arXiv:2407.15988](#).
- [30] Nadine Meister, Christopher A. Pattison, and John Preskill. “Efficient soft-output decoders for the surface code” (2024). [arXiv:2405.07433](#).
- [31] Nicolas Delfosse and Gilles Zémor. “Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel”. *Physical Review Research* **2**, 033042 (2020).
- [32] Nicolas Delfosse and Naomi H. Nickerson. “Almost-linear time decoding algorithm for topological codes”. *Quantum* **5**, 595 (2021).
- [33] Namitha Liyanage, Yue Wu, Siona Tagare, and Lin Zhong. “FPGA-based distributed union-find decoder for surface codes”. *IEEE Transactions on Quantum Engineering* **5**, 1–18 (2024).
- [34] Maximilian Jakob Heer, Emanuele Del Sozzo, Keisuke Fujii, and Kentaro Sano. “Novel union-find-based decoders for scalable quantum error correction on systolic arrays”. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Pages 524–533. (2023).
- [35] Maximilian Jakob Heer, Jan-Erik R. Wichmann, and Kentaro Sano. “Achieving scalable quantum error correction with union-find on systolic arrays by using multi-context processing elements”. In 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). Volume 02, pages 242–243. (2023).
- [36] Abbas B. Ziad, Ankit Zalawadiya, Canberk Topal, Joan Camps, György P. Gehér, Matthew P. Stafford, and Mark L. Turner. “Local clustering decoder as a fast and adaptive hardware decoder for the surface code”. *Nature Communications* **16**, 11048 (2025).
- [37] Federico Valentino, Beatrice Branchini, Davide Conficconi, Donatella Sciuto, and Marco D. Santambrogio. “An accurate union find decoder for quantum error correction on the toric code”. In 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Pages 99–105. (2024).
- [38] Yuhang Gu and He Li. “NNUF: Nearest neighbor union-find decoder for quantum error correction”. In 2025 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). Pages 1–4. (2025).
- [39] Kaito Kishi, Riki Toshio, Jun Fujisaki, Hiro-taka Oshima, Shintaro Sato, and Keisuke Fujii. “Even more efficient soft-output decoding with extra-cluster growth and early stopping” (2026). [arXiv:2602.03336](#).
- [40] Namitha Liyanage, Yue Wu, Alexander Deters, and Lin Zhong. “Scalable quantum error correction for surface codes using FPGA”. In 2023 IEEE International Conference on Quan-

- tum Computing and Engineering (QCE). Volume 01, pages 916–927. (2023).
- [41] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. “Topological quantum memory”. *Journal of Mathematical Physics* **43**, 4452–4505 (2002).
  - [42] Luka Skoric, Dan E. Browne, Kenton M. Barnes, Neil I. Gillespie, and Earl T. Campbell. “Parallel window decoding enables scalable fault tolerant quantum computation”. *Nature Communications* **14**, 7040 (2023).
  - [43] Xinyu Tan, Fang Zhang, Rui Chao, Yaoyun Shi, and Jianxin Chen. “Scalable surface-code decoders with parallelization in time”. *PRX Quantum* **4**, 040344 (2023).
  - [44] Tim Chan (2023). url: <https://github.com/timchan0/localuf>.
  - [45] Oscar Higgott and Craig Gidney. “Sparse Blossom: correcting a million errors per core second with minimum-weight matching”. *Quantum* **9**, 1600 (2025).
  - [46] Christophe Piveteau, Christopher T. Chubb, and Joseph M. Renes. “Tensor-network decoding beyond 2D”. *PRX Quantum* **5**, 040303 (2024).
  - [47] Ashley M. Stephens. “Fault-tolerant thresholds for quantum error correction with the surface code”. *Physical Review A* **89**, 022321 (2014).
  - [48] Lucas Berent, Timo Hillmann, Jens Eisert, Robert Wille, and Joschka Roffe. “Analog information decoding of bosonic quantum low-density parity-check codes”. *PRX Quantum* **5**, 020349 (2024).
  - [49] Shilin Huang and Shruti Puri. “Increasing memory lifetime of quantum low-density parity check codes with sliding-window noisy syndrome decoding”. *Physical Review A* **110**, 012453 (2024).
  - [50] Anqi Gong, Sebastian Cammerer, and Joseph M. Renes. “Toward low-latency iterative decoding of QLDPC codes under circuit-level noise” (2024). [arXiv:2403.18901](https://arxiv.org/abs/2403.18901).
  - [51] Kao-Yueh Kuo and Ching-Yi Lai. “Fault-tolerant belief propagation for practical quantum memory” (2024). [arXiv:2409.18689](https://arxiv.org/abs/2409.18689).
  - [52] Héctor Bombín, Chris Dawson, Ye-Hua Liu, Naomi Nickerson, Fernando Pastawski, and Sam Roberts. “Modular decoding: parallelizable real-time decoding for quantum computers” (2023). [arXiv:2303.04846](https://arxiv.org/abs/2303.04846).
  - [53] Sophia Fuhui Lin, Eric C. Peterson, Krishanu Sankar, and Prasanth Sivarajah. “Spatially parallel decoding for multi-qubit lattice surgery”. *Quantum Science and Technology* **10**, 035007 (2025).
  - [54] Poulami Das, Aditya Locharla, and Cody Jones. “LILLIPUT: a lightweight low-latency lookup-table decoder for near-term quantum error correction”. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Pages 541–553. New York, NY, USA (2022). Association for Computing Machinery.
  - [55] Yue Wu and Lin Zhong. “Fusion Blossom: Fast MWPM decoders for QEC”. In 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). Volume 01, pages 928–938. (2023).
  - [56] Yue Wu, Namitha Liyanage, and Lin Zhong. “Micro Blossom: Accelerated minimum-weight perfect matching decoding for quantum error correction”. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. Pages 639–654. New York, NY, USA (2025). Association for Computing Machinery.
  - [57] Emanuel Knill. “Quantum computing with realistically noisy devices”. *Nature* **434**, 39–44 (2005).
  - [58] L. Rieseboos, X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels. “Pauli frames for quantum computer architectures”. In Proceedings of the 54th Annual Design Automation Conference 2017. Pages 1–6. New York, NY, USA (2017). Association for Computing Machinery.
  - [59] Austin G. Fowler, Adam C. Whiteside, and Lloyd C. L. Hollenberg. “Towards practical classical processing for the surface code: Timing analysis”. *Physical Review A* **86**, 042313 (2012).
  - [60] Tim Chan. “Benchmarking accuracy in an emulated memory experiment” (2024). [arXiv:2411.02505](https://arxiv.org/abs/2411.02505).
  - [61] Dominic Horsman, Austin G. Fowler, Simon Devitt, and Rodney Van Meter. “Surface code quantum computing by lattice surgery”. *New Journal of Physics* **14**, 123011 (2012).
  - [62] Austin G. Fowler and Craig Gidney. “Low overhead quantum computation using lattice surgery” (2019). [arXiv:1808.06709](https://arxiv.org/abs/1808.06709).
  - [63] Daniel Litinski. “A game of surface codes: Large-scale quantum computing with lattice surgery”. *Quantum* **3**, 128 (2019).
  - [64] Google Quantum AI. “Suppressing quantum errors by scaling a surface code logical qubit”. *Nature* **614**, 676–681 (2023).
  - [65] Google Quantum AI and collaborators. “Quantum error correction below the surface code threshold”. *Nature* **638**, 920–926 (2025).
  - [66] Abdullah Khalid, Allyson Silva, Gebremedhin A. Dagnew, Tom Dvir, Oded Wertheim, Motty Gruda, Xiangzhou Kong, Mia Kramer, Zak Webb, Artur Scherer, Masoud Mohseni, Yonatan Cohen, and Pooya Ronagh. “Impacts of decoder latency on utility-scale quantum computer architectures” (2025). [arXiv:2511.10633](https://arxiv.org/abs/2511.10633).
  - [67] Inayat Ali, Jong-Hwan Kim, Sang-Hyo Kim,

Heeyoul Kwak, and Jong-Seon No. “Improving windowed decoding of SC LDPC codes by effective decoding termination, message reuse, and amplification”. *IEEE Access* **6**, 9336–9346 (2018).

- [68] Craig Gidney. “Stability experiments: The overlooked dual of memory experiments”. *Quantum* **6**, 786 (2022).
- [69] Christopher Chamberland and Earl T. Campbell. “Universal quantum computing with twist-free and temporally encoded lattice surgery”. *PRX Quantum* **3**, 010331 (2022).
- [70] György P. Gehér, Ophelia Crawford, and Earl T. Campbell. “Tangling schedules eases hardware connectivity requirements for quantum error correction”. *PRX Quantum* **5**, 010348 (2024).
- [71] Andrew Richards. “University of Oxford Advanced Research Computing”. (2015).
- [72] David S. Wang, Austin G. Fowler, and Lloyd C. L. Hollenberg. “Surface code quantum computing with error rates over 1%”. *Physical Review A* **83**, 020302 (2011).

## A Noise Model

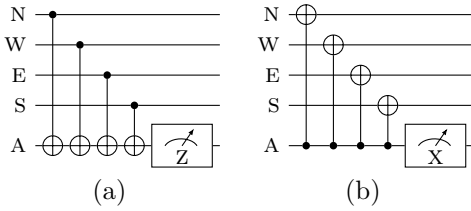


Figure 12: The (a) Z- and (b) X-stabiliser measurement circuits for the surface code. N, W, E, S are the data qubits north, west, east, and south of ancilla qubit A, respectively. Each data qubit interacts with a different ancilla qubit every step for the first four steps but only one interaction is shown. The initial state of A is the measurement outcome from the previous round i.e.  $|0\rangle$  or  $|1\rangle$  for (a), and  $|+\rangle$  or  $|-\rangle$  for (b).

| Probability | Error Process                                                                                                                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| $p_M$       | qubit measurement reports the opposite outcome                                                                                                |
| $p_1$       | qubit idle is followed by an error drawn randomly from $\{\hat{X}, \hat{Y}, \hat{Z}\}$                                                        |
| $p_2$       | $C\hat{X}$ is followed by an error drawn randomly from $\{\hat{I}, \hat{X}, \hat{Y}, \hat{Z}\}^{\otimes 2} \setminus \{\hat{I}^{\otimes 2}\}$ |

Table 2: Error processes for the circuit-level depolarising noise model. Each process can occur once in each of the 5 steps in Fig. 12.

Our numerics were performed using the same noise model as in [17, §B]. Namely, we assume the stabiliser measurement circuits in Fig. 12 and apply the error

processes in Table 2, using the balanced parametrisation from [72, p 1]:

$$(p_M, p_1, p_2) = \left(\frac{2}{3}, \frac{4}{5}, \frac{4}{5}, 1\right)p, \quad (2)$$

where  $p$  is the noise level.

## B Proof of Proposition 1

We first define three functions. The *wholeness* of a non-mixed cluster  $K$  is

$$\text{wh } K := \begin{cases} 1 & K \text{ a whole cluster,} \\ \frac{1}{2} & K \text{ a half cluster.} \end{cases} \quad (3)$$

The *ungrown length* of a path  $P$  in  $W$  that comprises  $f$  fully grown edges,  $h$  half-grown edges, and  $z$  un-grown edges, is

$$l(P) := \frac{1}{2}h + z. \quad (4)$$

The *ungrown distance* between clusters  $K, L$  is the minimum un-grown length of a path from a node in  $K$  to a node in  $L$ :

$$d(K, L) := \min\{l(uv, \dots, wx) : ux \in V_K \times V_L\}. \quad (5)$$

Now consider the following.

**Lemma 3.** *After each growth round of Snowflake with the 2:1 schedule, we have*

$$d(K, L) \in \mathbb{Z} + \text{wh } K + \text{wh } L \quad (6)$$

for any two non-mixed clusters  $K, L$ .

*Proof.* By induction on the number  $r$  of growth rounds elapsed.

Consider base case  $r = 0$ . All  $e \in E_W$  are un-grown so  $l(P) \in \mathbb{Z}$  for any path  $P$  in  $W$ , and so  $d(K, L) \in \mathbb{Z}$ . All clusters are whole so  $\text{wh } K + \text{wh } L = 2$ , and so Eq. (6) is true for  $r = 0$ .

Now the inductive step; assume true after  $r$  growth rounds and consider growth round  $r + 1$ . If...

- neither  $K$  nor  $L$  grows, all terms in Eq. (6) are unchanged;
- $K$  grows but not  $L$ ,  $d(K, L)$  decreases by  $\frac{1}{2}$  and  $\text{wh } K$  changes by  $\frac{1}{2}$ ;
- both  $K$  and  $L$  grow, we must have  $\text{wh } K + \text{wh } L \in \{1, 2\}$ , so  $K$  and  $L$  cannot be half an edge apart before growth, so  $d(K, L)$  decreases by 1. Also,  $\text{wh } K$  and  $\text{wh } L$  each change by  $\frac{1}{2}$ .

In each case, Eq. (6) remains true.  $\square$

Now we can prove the desired statement.

*Proof of Proposition 1.* There are no mixed clusters initially. By Lemma 3, a whole cluster never merges with a half cluster, so mixed clusters never form.  $\square$

## C Schedule Accuracy Comparison

Section 4.2 discussed 1:1 and 2:1 as two possible cluster growth schedules for Snowflake. Figure 13 shows how Snowflake performs with each schedule: 2:1 almost doubles the threshold  $p_{th}$  of 1:1.

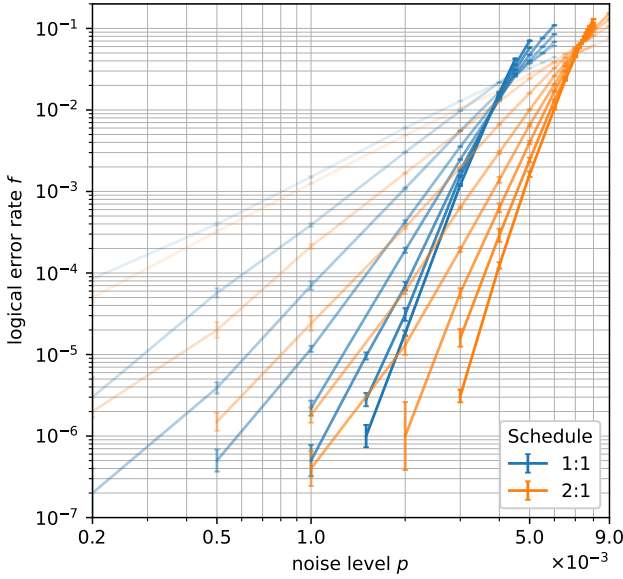


Figure 13: Same as Fig. 1 but the blue lines instead show data for Snowflake with the 1:1 cluster growth schedule. The coordinate  $(p, f)_{th}$  of the threshold is around  $(0.375, 9) \cdot 10^{-2}$  for 1:1, and  $(0.735, 6.8) \cdot 10^{-2}$  for 2:1.

## D Distributed Implementation of Snowflake

In this section we refer to processors as nodes and communication links as edges. The controller can broadcast the same boolean to, and receive (without knowing the sender) booleans from, every node.

In Appendix D.1 we list the variables each component stores and in Appendix D.2 we explain how they are manipulated in order to effect Snowflake. Appendices D.3 and D.4 explain the variable-duration stages in more detail. Appendix D.5 describes how we minimise decoding latency after all data qubits are measured out.

### D.1 Setup

Each node  $v \in V_W$  is assigned a unique integer ID and stores the following variables:

1. **active** is a boolean indicating whether  $v$  is in an active cluster. If  $v.active = \text{true}$  we say  $v$  is active.
2. **whole** is a boolean indicating whether  $v$  is in a whole cluster. If  $v.whole = \text{true}$  we say  $v$  is whole.

3. **CID** is an integer indicating the cluster that  $v$  belongs to. Clusters are identified by the lowest ID of all its nodes. The node of this ID is the root. CID can also adopt the value **reset** to indicate  $v$  should *unroot* (defined in Appendix D.4.2) in the next timestep.
4. **defect** is a boolean indicating whether  $v$  has a defect. This can be thought of as a particle which is passed between nodes. When a node receives two defects in a timestep, they annihilate.
5. **pointer** is the direction  $v$  should push the defect (by flipping the corresponding edge). Possible values (under circuit-level noise) are C, N, W, E, S, D, U, NU, WD, EU, SD, NWD, SEU, representing respectively centre, north, west, east, south, down, up, and combinations thereof. Following the pointers starting from  $v$  should eventually lead to the root of the cluster that  $v$  is in. Only roots have **pointer** = C as they do not push defects but accumulate them.
6. **grown** is a boolean indicating whether the cluster that  $v$  is in has grown in the first growth round of the current decoding cycle.
7. **unrooted** is a boolean indicating whether  $v$  has unrooted in the current decoding cycle.
8. **busy** is a boolean indicating whether  $v$  is busy.
9. **stage** is the current stage of Snowflake. This follows the flowchart in Fig. 14.

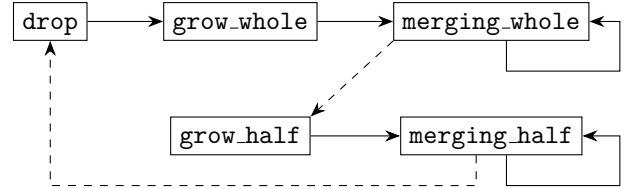


Figure 14: Flowchart for the stages of Snowflake: drop, grow\_whole, and grow\_half each last one timestep whereas the other two last a variable number, hence their loops. The dashed (solid) arrows indicate advances by the global controller (each node locally). Snowflake cycles through all five stages once per decoding cycle.

We refer to the first five variables as *relative* and the last four as *absolute*. Each edge  $e \in E_W$  has the following variables:

1. **growth** is its growth value which is in  $\{0, \frac{1}{2}, 1\}$ .
2. **correction** is a boolean indicating if  $e \in T$ .

These are stored as additional relative variables at the endpoint with the lower ID. It helps to define the *owned edges* as those whose variables a node stores:

$$v.owned := \{uv : uv \in E_W \text{ and } v.ID < u.ID\}. \quad (7)$$

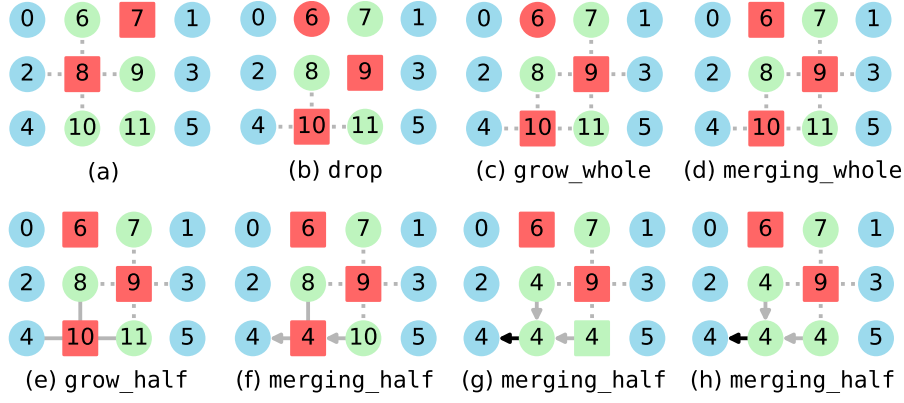


Figure 15: A decoding cycle example for a distance-3 code. Edges are shown as in Fig. 5. Active nodes are squares; inactive nodes, circles. CID is shown as a label. Pointers are shown by arrows on edges. (a) Each node is in its own cluster so has CID = ID. Two clusters are active. (b) All relative variables fall by one layer, including CID which is adjusted to refer to the node below. A new defect appears at the top. (c) Node 9 grows its incident edges by  $\frac{1}{2}$  as it is active and whole. (d) Node 6 recognises it is a root with a defect, so becomes active. (e) Node 10 grows its incident edges by  $\frac{1}{2}$  as it is active and half. Nodes 10 and 11 now each see a lower CID than their own when they look at their neighbours along fully grown edges. (f) They adopt that CID and point toward that neighbour; node 8 thus sees a lower CID than its own. (g) A defect is pushed along a pointer to the root which flips an edge. Node 8 updates its CID and pointer, so now the 4-node cluster is fully flooded but not yet fully synced. (h) Throughout this stage, each non-root node was adopting the active variable of the node it was pointing to. This activity propagation now completes, so the 4-node cluster is fully synced.

## D.2 Procedures

Though the pseudocode in this subsection explains exactly what each component does, it is difficult to infer from this the *emergent effect*. We describe the latter in the accompanying prose.

**Algorithm 6** Run by controller every timestep.

```

if not (waiting or any  $v$ .busy) then
  for  $v \in V_W$  do
     $v$ .stage  $\leftarrow$  next stage
    if  $v$ .stage was merging_whole then
      wait 1 timestep
    else
      wait 2 timesteps

```

In each timestep the controller runs Algorithm 6 which, during either merging stage, tells all nodes to advance to the next stage if none of them are busy. This is the only time the controller must communicate with the nodes, as the merging stages are the only ones of variable duration.

**Algorithm 7** Run by node  $v$  every timestep.

```

call the procedure in Algorithm 8 matching  $v$ .stage
if  $v$ .stage  $\neq$  merging then
   $v$ .stage  $\leftarrow$  next stage

```

At the same time, each node  $v$  runs Algorithm 7, calling procedures from Algorithm 8. Stage **drop** adds to  $\mathbb{C}$  and raises  $W$  by one layer, as in Fig. 15b. Each node  $v$  in the top sheet of  $W$  starts in its own cluster, thus sets its relative variables to reflect this (Lines 10

and 11).  $v$ .whole,  $v$ .CID, and  $v$ .pointer need not be explicitly reset as they never change from their correct values of **true**,  $v$ .ID, and  $\mathbb{C}$ , respectively.

Stage **grow\_whole** grows all edges around each active whole cluster by  $\frac{1}{2}$ , as in Fig. 15c. Stage **grow\_half** does the same but for active half clusters that have not already grown in the current decoding cycle, as in Fig. 15e.

Figure 15d shows a merging example that lasts just one timestep, and f–h shows one that needs three timesteps. The **merging\_whole** stage is the same as **merging\_half** but with two extra processes, so we first explain the latter: it comprises two processes, *syncing* and *flooding*, which occur independently. We thus split MERGINGHALF into two subroutines, given in Algorithm 9, effecting each process.

## D.3 Processes Common to Both Merging Stages

In flooding, newly touching clusters adopt the lowest CID among them. This CID propagates as a flood starting from where they touched: each node looks at its neighbours along fully grown edges. If it sees a lower CID than its own, it adopts that CID and points toward that neighbour (Lines 44 to 47). After this process, all nodes in a given cluster will have the same CID, and following the pointers will always lead to the root promised by CID.

*Remark 7.* Crucial to the algorithm is that boundary nodes are of lower ID than detectors. Each root can thus determine if its cluster touches a boundary by checking if itself is a boundary node.

In syncing, all defects are pushed along pointers to-

---

**Algorithm 8** The five procedures that can be called in [Algorithm 7](#).

---

```

procedure DROP( $v$ )
2:    $v.\text{grown} \leftarrow \text{false}$ 
   for  $e \in v.\text{owned} \cap \text{commit region}$  do
4:     if  $e.\text{correction}$  then                                      $\triangleright$  Commit  $e$ .
        $\mathbb{C} \leftarrow \mathbb{C} \Delta \{e\}$ 
6:     if  $\exists$  node  $u$  below  $v$  then                                    $\triangleright$  Shift relative data down one layer.
        $v.\text{CID} \leftarrow \text{ID of node below current root}$ 
       send all relative variables to  $u$ 
       if  $v \in \text{top sheet of } W$  then                                $\triangleright$  Inherit the new measurement round results.
10:       $v.\text{defect} \leftarrow \text{whether the measurement of the corresponding ancilla qubit differs from the last round}$ 
       $v.\text{active} \leftarrow \text{false}$ 
12: procedure GROWWHOLE( $v$ )
14:   if  $v.\text{active}$  and  $v.\text{whole}$  then
       GROW( $v$ )
16:    $v.\text{grown} \leftarrow \text{true}$ 
       if  $v \in \text{bottom sheet of } W$  and  $v.\text{pointer} \ni D$  then        $\triangleright$  Start unrooting  $v$ .
18:       $v.\text{CID} \leftarrow \text{reset}$ 
       $v.\text{pointer} \leftarrow C$ 
20: procedure GROWHALF( $v$ )
22:    $v.\text{unrooted} \leftarrow \text{false}$ 
       if  $v.\text{active}$  and not  $v.\text{whole}$  and not  $v.\text{grown}$  then
24:      GROW( $v$ )
26: procedure MERGINGWHOLE( $v$ )
        $v.\text{busy} \leftarrow \text{false}$ 
28:   SYNCING( $v$ )
       FLOODINGWHOLE( $v$ )
30: procedure MERGINGHALF( $v$ )
32:    $v.\text{busy} \leftarrow \text{false}$ 
       SYNCING( $v$ )
34:   FLOODINGHALF( $v$ )

```

---

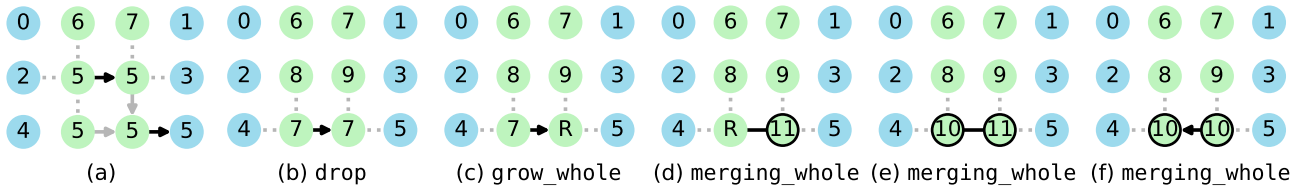


Figure 16: A decoding cycle which demonstrates unrooting. Relative variables are shown as in [Fig. 15](#). CID value reset is labelled 'R'. Nodes with unrooted as true are outlined in black. (a) There is a 5-node cluster. (b) A downward pointer is dropped to the bottom east detector. (c) Thus, this detector starts unrooting. (d) The bottom west detector sees this signal to unroot so starts unrooting. The bottom east detector finishes unrooting. (e) The bottom west detector finishes unrooting. (f) merging\_whole completes.

ward roots, one edge per timestep (Lines 13 to 15). Eventually, each root will know using Lemma 1 whether its cluster is active (Line 9). At the same time, this activity knowledge propagates in the opposite direction from root to the rest of the cluster (Line 12). After this process, a node is in an active cluster iff it is active.

---

**Algorithm 9** Subroutines for Algorithm 8.

---

```

procedure GROW( $v$ )
2:   for each edge  $e$  incident to  $v$  do
   |    $e.\text{growth} \leftarrow \min\{1, e.\text{growth} + \frac{1}{2}\}$ 
4:   flip  $v.\text{whole}$ 

6:   procedure SYNCING( $v$ )
   |    $x := (v \text{ is a detector}) \text{ and } v.\text{defect}$ 
8:   if  $v.\text{pointer} = C$  then  $\triangleright v \text{ is a root.}$ 
   |   |    $v.\text{active} \leftarrow x$ 
10:  else
   |   |    $u := \text{node toward } v.\text{pointer}$ 
12:  |   |    $v.\text{active} \leftarrow u.\text{active}$ 
   |   |   if  $x$  then
14:  |   |   |    $v.\text{busy} \leftarrow \text{true}$ 
   |   |   |    $\text{flip } uv.\text{correction}$  to push defect to  $u$ 
16:  |   |   if  $v.\text{active}$  changed then
   |   |   |    $v.\text{busy} \leftarrow \text{true}$ 

18:  procedure FLOODINGWHOLE( $v$ )
20:  if  $v.\text{CID} = \text{reset}$  then  $\triangleright \text{Finish unrooting } v.$ 
   |    $v.\text{busy} \leftarrow \text{true}$ 
22:  |    $v.\text{CID} \leftarrow v.\text{ID}$ 
   |    $v.\text{unrooted} \leftarrow \text{true}$ 
24:  else
   |   for all  $u : uv \in E_W \wedge uv.\text{growth} = 1$  do
26:  |   |   if  $u.\text{CID} = \text{reset}$  then
   |   |   |   if not  $v.\text{unrooted}$  then
28:  |   |   |   |    $\triangleright \text{Start unrooting } v.$   $\triangleleft$ 
   |   |   |   |    $v.\text{busy} \leftarrow \text{true}$ 
30:  |   |   |   |    $v.\text{CID} \leftarrow \text{reset}$ 
   |   |   |   |    $v.\text{pointer} \leftarrow C$ 
32:  |   |   |   |   break
   |   |   |   else
34:  |   |   |   |    $\text{COMPARE}(u, v)$ 
   |   |   |   |   if  $u.\text{grown}$  and not  $v.\text{grown}$  then
36:  |   |   |   |   |    $v.\text{busy} \leftarrow \text{true}$ 
   |   |   |   |   |    $v.\text{grown} \leftarrow \text{true}$ 

38:  procedure FLOODINGHALF( $v$ )
40:  for all  $u : uv \in E_W \wedge uv.\text{growth} = 1$  do
   |    $\text{COMPARE}(u, v)$ 

42:  procedure COMPARE( $u, v$ )
44:  if  $u.\text{CID} < v.\text{CID}$  then
   |    $v.\text{busy} \leftarrow \text{true}$ 
46:  |    $v.\text{pointer} \leftarrow \text{toward } u$ 
   |    $v.\text{CID} \leftarrow u.\text{CID}$ 

```

---

Note that swapping the order of the two subroutines in MERGINGHALF would speed up the stage,

as pointers would be updated *before* being used to push defects and backpropagate activity. However, the practicality of this depends on the hardware so in this paper we assume the worse case and perform SYNCING using pointers from the *previous* timestep.

## D.4 Processes Unique to merging\_whole

This subsection will refer to both Algorithms 8 and 9.

### D.4.1 Tracking Which Clusters Have Grown

The variable **grown** floods in the same fashion as **cid** but with no connection to **pointer** (Algorithm 9 Lines 35 to 37). This ensures all nodes in a given cluster will have the same boolean for **grown**. The memory of whether a cluster has grown resets after each decoding cycle (Algorithm 8 Line 2).

### D.4.2 Unrooting

*Unrooting* is a process unique to Snowflake and is needed when a downward pointer is dropped to the bottom sheet of  $W$  (can happen while a cluster falls out of view). This breaks the pointer tree structure established in the previous decoding cycle, as  $\exists$  nonempty  $S \subset V_W$  from which following the pointers will lead past the bottom of  $W$ . To fix it, each node in  $S$  resets its CID and **pointer** so that the structure can be rebuilt from scratch, via further **merging\_whole** timesteps. Figure 16 shows an example.

The breaking point is found immediately after **drop** (Algorithm 8 Line 17) whence the signal to unroot,  $\text{CID} = \text{reset}$ , propagates during **merging\_whole** as a flood through the cluster; each node unroots itself as soon as it receives this signal (Algorithm 9 Lines 25 to 32). It takes 2 timesteps for a node to unroot, after which it cannot unroot again in the same decoding cycle (Algorithm 9 Line 27). This ensures the flood does not backpropagate. The memory of whether a node has unrooted resets after this stage finishes (Algorithm 8 Line 22).

The higher up a node is in  $W$ , the lower its ID; this is for two reasons. First, it makes unrooting less likely as pointer paths tend to lead upward rather than downward. Second, it ensures defects in active clusters are pushed to a highest point so they can annihilate with defects in newer, smaller clusters as early as possible.

## D.5 After Data Qubit Measurement

As Section 5.3 describes, data qubit measurement (e.g. at the end of a memory experiment) means there will be a final sheet of syndrome data. In the worst case, Snowflake needs  $b + 1$  decoding cycles between receiving this final sheet and committing its final tentative correction, but on average, most of these growing and merging stages are unnecessary (when there

are no more defects in the decoding window). To minimise latency, the distributed implementation bypasses these unnecessary stages by, upon receiving the final sheet of syndrome data, modifying Algorithm 6 as follows: the controller additionally checks after each decoding cycle for any defects in the decoding window; if there are none, the controller repeatedly sets the stage of all nodes to **drop** so that only drop stages are performed.

## E When a Cluster Touches Both Boundaries

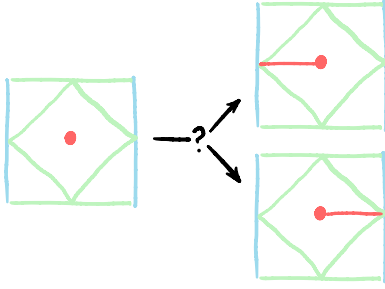


Figure 17: A cluster that touches both west and east boundaries. If it contains an odd number of defects, the correction must contain a path that connects to one of them.

In either UF or Snowflake, whenever a cluster has grown large enough to touch both the west and east boundaries, there is an arbitrary choice of which boundary to push defects to, as Fig. 17 illustrates. Whether the decoder chooses the same boundary or a random boundary every time may impact the overall logical error rate. Snowflake always chooses the west boundary, whereas our implementation of FM(UF) may not necessarily. To see if this explains the accuracy discrepancy between FM(UF) and Snowflake, we reran emulation with a version of FM(UF) that always chooses the west boundary. As Fig. 18 shows, we observed a slight decrease in logical error rate but it was not enough to explain the accuracy discrepancy.

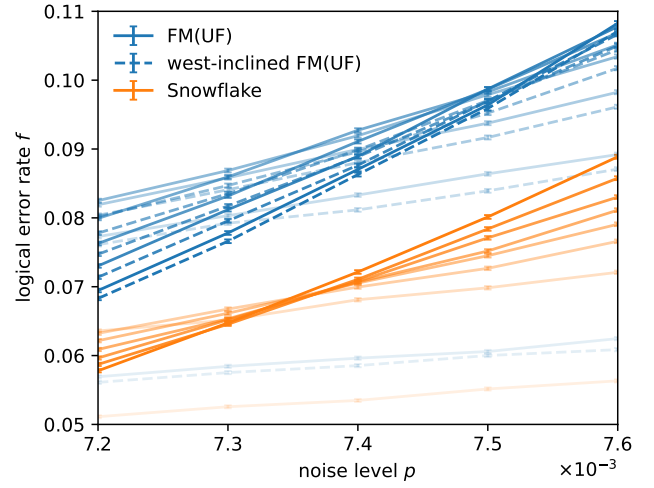


Figure 18: A zoomed-in view of Fig. 1. The additional decoder, west-inclined FM(UF), is the same as FM(UF) but whenever a cluster touches both boundaries, it always pushes defects within that cluster to the west boundary, mimicking the behaviour of Snowflake.