Towards practical classical processing for the surface code: Timing analysis

Austin G. Fowler, Adam C. Whiteside, and Lloyd C. L. Hollenberg
Centre for Quantum Computation and Communication Technology, School of Physics, The University of Melbourne, Victoria 3010, Australia

(Received 24 February 2012; published 12 October 2012)

Topological quantum error-correction codes have high thresholds and are well suited to physical implementation. The minimum-weight perfect-matching algorithm can be used to efficiently handle errors in such codes. We perform a timing analysis of our current implementation of the minimum-weight perfect-matching algorithm. Our implementation performs the classical processing associated with an $n \times n$ lattice of qubits realizing a square surface code storing a single logical qubit of information in a fault-tolerant manner. We empirically demonstrate that our implementation requires only $O(n^2)$ average time per round of error correction for code distances ranging from 4 to 512 and a range of depolarizing error rates. We also describe tests we have performed to verify that it always obtains a true minimum-weight perfect matching.

DOI: 10.1103/PhysRevA.86.042313 PACS number(s): 03.67.Pp

I. INTRODUCTION

Quantum computers promise efficient factoring [1], efficient simulation of quantum systems [2], and the efficient solution of many other classically intractable problems [3]. The primary barrier to the realization of a quantum computer is the physical realization of quantum gates with sufficiently low error to enable quantum error correction to be used. Topological quantum error-correction (TQEC) codes can tolerate error rates of order 1% [4,5] and require only two-dimensional (2D) nearest-neighbor interactions, both physically reasonable targets; however, the classical processing associated with the error correction is highly nontrivial. Without significant future effort, the classical processing will almost certainly limit the speed of any quantum computer, particularly one with intrinsically fast quantum gates.

In this work, we present a timing analysis of our software performing the classical processing associated with TQEC. This software is by orders of magnitude the fastest currently available. We will review the necessary aspects of the surface code [6,7], fault-tolerant schemes built on the surface code [8–10], and our classical processing algorithm [5] as required. Our goal is to analyze in detail the performance and correctness of our implementation of this algorithm. This implementation is contained in a library match.c and called by our tool AUTOTUNE [11], which is designed to prepare a graph problem tailored to arbitrary hardware running a surface-code-family TQEC scheme.

The discussion is organized as follows. In Sec. II, the basic structure and functionality of our software are described. The library match.c, which performs minimum-weight perfect matching [12,13] is described in more detail in Sec. III. Two versions are discussed. An example of the faster version of the algorithm in action is provided in Sec. IV. The probability of logical errors in the surface code as a function of the physical error rate p is discussed in Sec. V. Formatted timing data are presented in Sec. VI. Complete raw timing data can be found in the Supplemental Material. Section VII summarizes and points to further work.

II. OVERVIEW

Our simulation suite of software is designed to handle arbitrary hardware with arbitrary stochastic error models; however, we shall focus on a simple 2D square lattice of qubits and a standard depolarizing channel for each quantum gate for the purposes of benchmarking and demonstrating correctness. Specifically, we shall study the case of no initialization surface-code error detection [4]. A small section of the 2D array of data and syndrome qubits of the surface code and the required cyclic sequence of controlled-NOT (CNOT) gates to simultaneously measure all stabilizers [14] is shown in Fig. 1. At the end of each cycle, all syndrome qubits are measured in the X or Z basis according to whether they are being used to measure X or Z stabilizers, respectively.

Random Pauli errors are generated and propagated using a Pauli frame. When errors lead to syndrome measurement value changes, graph vertices are generated at these space-time locations. By preanalyzing all possible single-error processes [4,11], an underlying lattice of dots and lines is also prepared with dots at every location where a vertex could potentially be generated and lines between every pair of locations that could have vertices generated by a single error. The first-order probability p_{line} of each line is calculated and a weight $w = -\ln(p_{\text{line}})$ stored in each line. This is done so that a large positive weight is associated with any line of low probability, ensuring that an algorithm matching vertices in pairs using paths of lines with minimum total weight will tend to avoid using low-probability lines. Furthermore, a multiple-line path will have a weight related to the product of probabilities of its constituent lines. A lattice of dots and lines and stochastically generated vertices (from surface-code simulation) is shown in Fig. 2.

In many ways, a lattice plus vertices can be considered an implicit complete graph with an edge between any pair of vertices having weight equal to a minimum-weight path between those vertices. The task is to match all vertices in pairs or to neighboring boundaries such that the total weight of all matched paths is minimal. The basic algorithm that efficiently solves this problem given a standard graph is the minimum-weight perfect-matching algorithm [12,13]. We have extended this algorithm to include the concept of boundaries and permit new vertices to be dynamically added to the graph.

III. MATCHING

We have two operational versions of extended minimum-weight perfect matching—complete match [4] which first

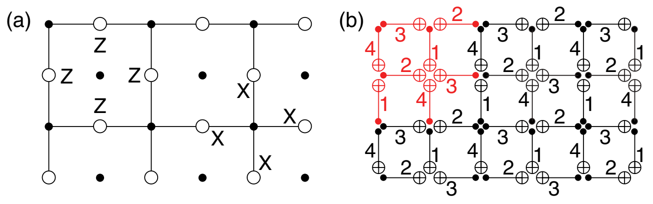

FIG. 1. (Color online) (a) 2D lattice of data qubits (circles) and syndrome qubits (dots) and examples of the data qubit stabilizers. (b) Sequence of CNOT gates permitting simultaneous measurement of all stabilizers. Numbers indicate the relative timing of gates. The highlighted gates can be tiled to fill the plane.

constructs explicit edges between all pairs of vertices no more than approximately d rounds of error correction apart, and edges on demand match [5] which constructs only a small number of local edges and adds further edges to the problem as required. The graphs and matchings generated by complete match (cmatch) and edges on demand match (eodmatch) given Fig. 2 as input are shown in Figs. 3 and 4, respectively. The total weight of matched edges in both cases is identical and in this case the matchings themselves are identical. We have tested cmatch and eodmatch on millions of varied problems, large and small, and always observed identical total weights, strongly implying that both implementations are correct.

Cmatch obtains the true minimum-weight perfect matching despite only including edges between vertices separated by a finite number of rounds. Vertices separated by a very large number of rounds are always cheaper to match to their nearest boundaries than to one another. By using the weights of the lines in the lattice, we calculate the minimum span of

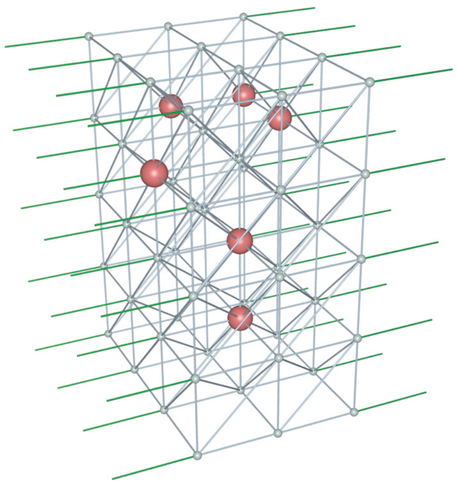

FIG. 2. (Color online) Distance-4 example of a lattice of dots and lines with stochastically generated vertices. The distance of a surface code is the length in lines of the shortest topologically nontrivial path, in this case any path connecting opposing boundaries. Dots (small balls) correspond to space-time locations where the end points of error chains could potentially be detected. Vertices (large balls) correspond to space-time locations where error-chain end points have been detected. Light cylinders link pairs of dots where a pair of vertices could be generated by a single error. Dark cylinders link spatial boundaries to a single dot where a single vertex could be generated by a single error.

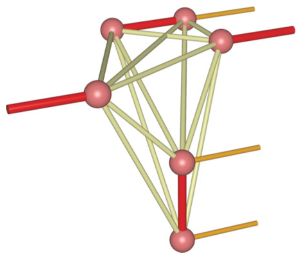

FIG. 3. (Color) Output of cmatch when given Fig. 2 as input. The underlying lattice is used to construct a complete graph and then discarded. Edges in the matching are shown in red.

rounds to connect with edges to guarantee a minimum weight matching. Eodmatch also obtains a true minimum-weight perfect matching as any required edge will eventually be included during execution.

We now describe the eodmatch algorithm. Some definitions are required. Let G be a graph with vertices $\{v_i\}$, edges $\{e_{ij}\}$, and edge weights $\{w_{ij}\}$. The graphs we use in eodmatch are implicitly complete, with the weight of an edge between any given pair of vertices defined to be the weight of a minimum-weight path between those vertices, and the weight of any edge connecting a vertex to a nearby boundary defined similarly. As such, we shall describe the algorithm as though we have a complete graph. The process of dynamically adding the required edges is just a technical detail.

Associate with each vertex v_i a variable y_i, which can be thought of as the radius of a ball centered at v_i. Odd sets of vertices can also be made into blossoms B_k that have their own variables Y_k, which can be thought of as the width of shell around every object in B_k. If a pair of blossoms intersect, then one is contained in the other. Define an edge e_{ij} to be tight if $w_{ij} - y_i - y_j - \sum Y_k = 0$, where the sum is over k such that exclusively v_i or v_j is in B_k. This condition is pictorially depicted in Fig. 5.

Define a node to be a vertex or blossom. Allow edges to possess a label matched or unmatched. Define a blossom to be unmatched if it contains a vertex not incident on a matched

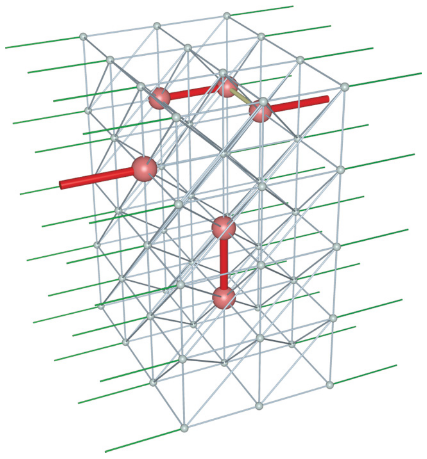

FIG. 4. (Color) Output of eodmatch when given Fig. 2 as input. Note that only one edge other than those ultimately included in the matching has been created (just visible between the top right two vertices).

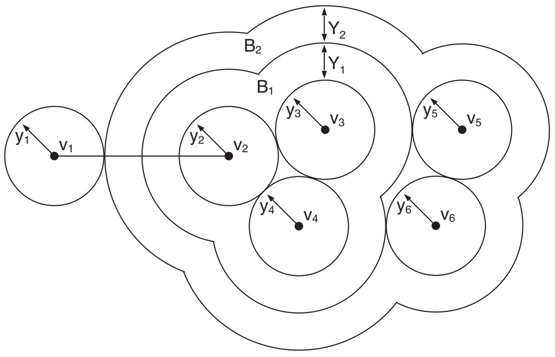

FIG. 5. An example of a tight edge. Edge e_{12} has the property that $w_{12} - y_1 - y_2 - Y_1 - Y_2 = 0$.

edge. An alternating tree is a tree of nodes rooted on an unmatched node such that every path of edges from the root node to a leaf node consists of alternating unmatched and matched edges. Alternating trees can branch only from the root and every second node from the root. Define branching nodes to be outer. Define all other nodes in the alternating tree to be inner. Figure 6 shows all necessary alternating tree manipulations.

Given a weighted graph G, the following algorithm finds a minimum-weight perfect matching.

(1) If there are no unmatched vertices, return the list of matched edges.

(2) Choose an unmatched vertex v to be the root of an alternating tree.

(3) If no edges emanating from the outer nodes of the alternating tree are tight, henceforth called O-tight edges, increase the value of y or Y associated with each outer node while simultaneously decreasing the value of y or Y associated with each inner node until an edge becomes O tight, or an inner blossom node Y variable becomes 0 [Fig. 6(a)].

(4) If an inner blossom node Y variable becomes 0 and there are still no O-tight edges, expand that blossom and return to 3 [Fig. 6(b)].

(5) Choose an O-tight edge e.

(6) If e leads to a matched node not already in the alternating tree, add the relevant unmatched and matched edges and associated nodes to the alternating tree and return to step 3 [Fig. 6(c)].

(7) If e leads to an inner node, mark e so it is not considered again during the growth of this alternating tree and return to step 3 [Fig. 6(d)].

(8) If e leads to an outer node, add the unmatched edge to the alternating tree. There will now be a cycle of odd length. Collapse this cycle into a new blossom and associate a new variable $Y = 0$ [Fig. 6(e)]. Return to step 3.

(9) If e leads to an unmatched vertex or boundary, add e to the alternating tree and augment the path (unmatched $\leftrightarrow$ matched) from the unmatched vertex within the root node to the end of e [Fig. 6(f)]. Destroy the alternating tree, keeping any newly formed blossoms. Return to step 1.

On average, the algorithm needs to consider only a small local region around each vertex to find another unmatched vertex to pair with. This is a property of the graphs associated with topological QEC only, as the probability of needing to consider an edge of length l decreases exponentially with l. This ensures that the runtime is $O(n^2)$, and that the algorithm can be parallelized to achieve $O(1)$ processing per round.

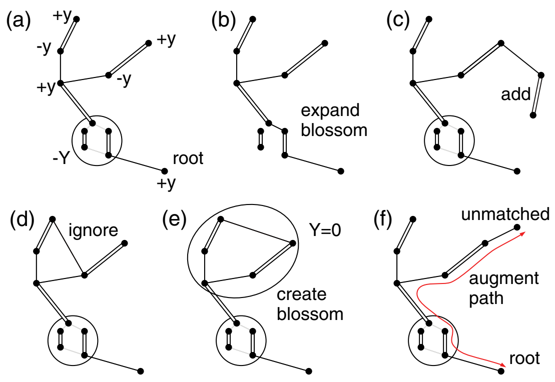

FIG. 6. (Color online) All required alternating tree manipulations. (a) Increase outer-node and decrease inner-node y values (or Y if the node is a blossom), which will maintain the tightness of all tree edges and potentially create new tight edges connected to at least one outer node. (b) Inner blossoms with $Y = 0$ can be expanded into multiple inner and outer nodes and potentially some nodes that are no longer part of the tree. (c) Outer-matched tight edges can be used to grow the alternating tree. (d) Outer-inner tight edges can be ignored as they never grow tighter. (e) Outer-outer tight edges make cycles that can be used to form blossoms. (f) When another unmatched vertex v is found, or an edge to a boundary b, the path from the unmatched vertex within the root node through the alternating tree to v or b is augmented, meaning matched edges become unmatched and unmatched edges become matched. This strictly increases the total number of matched vertices.

IV. EODMATCH EXAMPLE

The rules of the previous section are far from intuitive. Let us consider a simple 1D chain of qubits suffering errors and generating vertices in space and time. Assume the underlying lattice is square. Figure 7(a) shows a possible current state of the matching algorithm, with matched vertices far in the past and unmatched vertices in the present and recent past. The goal is to match as many vertices as possible in the active region (between the horizontal dashed lines) without using any data that are too new. The window that defines the active region rolls forward as additional vertices are generated by the quantum computer. The first vertex chosen for matching is indicated with an arrow. It does not matter which vertex is chosen in the active region; however, our algorithm has a preference for vertices further in the past.

Figure 7(b) shows a shaded exploratory region around the chosen vertex. This is constructed by performing a breadth-first search through the lattice local to the vertex. When any other object is encountered, whether it be a boundary, another exploratory region, or another vertex, expansion is halted. In this case, two unmatched vertices and one exploratory region simultaneously terminate expansion. One of these vertices is chosen to be matched to as shown in Fig. 7(c). It does not matter which vertex is chosen; both are valid choices that would lead to a minimum weight perfect matching being obtained. The next vertex is chosen.

In Fig. 7(d), when exploration around the chosen vertex terminates, a matched vertex is encountered and no unmatched

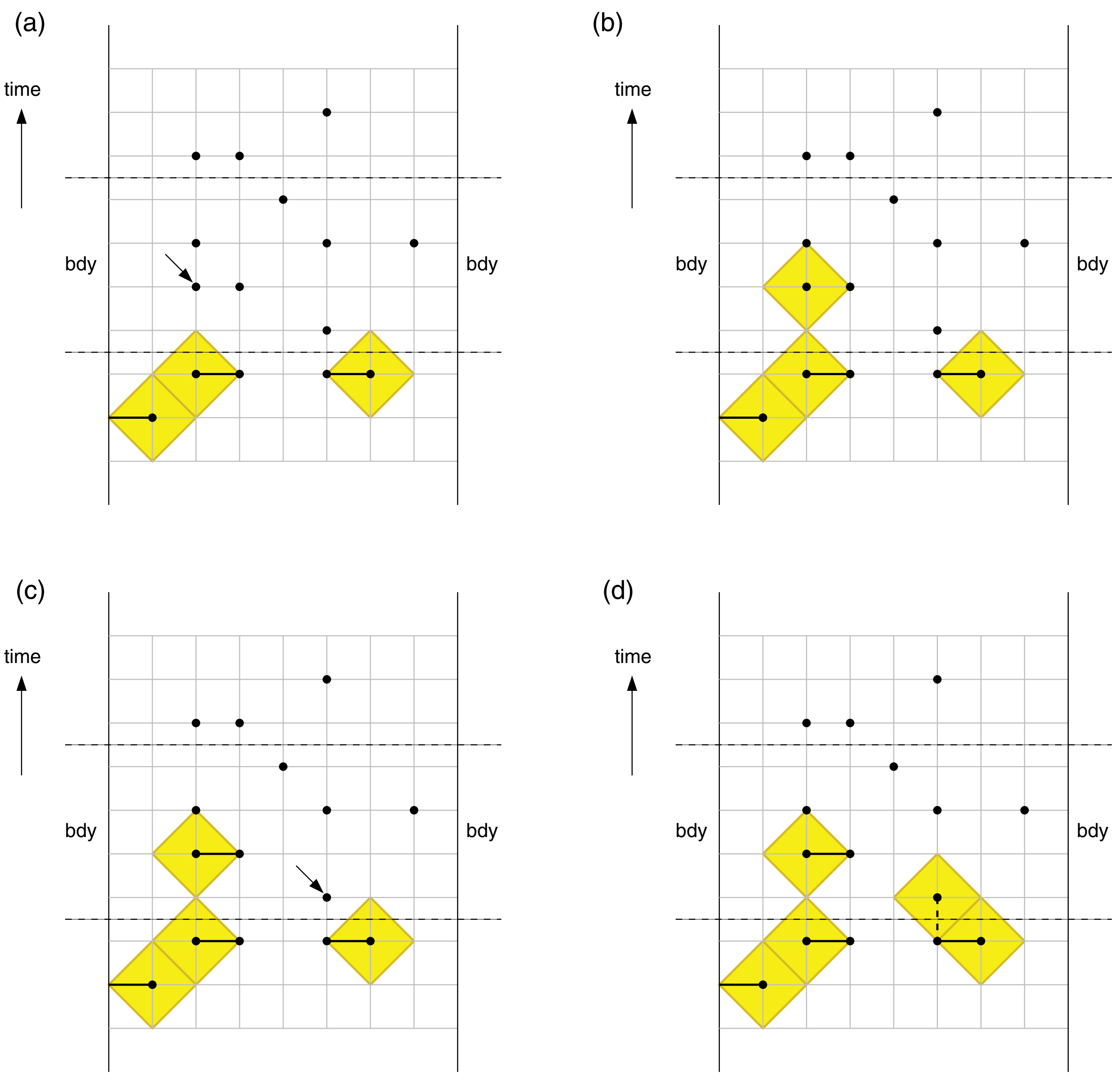

FIG. 7. (Color online) (a) Choose an unmatched vertex. (b) Expand exploratory region until other objects are encountered. (c) Unmatched vertices are encountered; choose one to match to. Choose another unmatched vertex. (d) Expand exploratory region until other objects are encountered. Build an alternating tree.

vertices. This necessitates the construction of an alternating tree. An alternating tree is a tree with alternating unmatched and matched edges. Alternating trees are allowed to branch only at the root and every second node from the root. Branching nodes are called outer nodes; nonbranching nodes are called inner nodes. The alternating tree constructed in Fig. 7(d) consists of three nodes, all of which are simple vertices. We will encounter more complex alternating trees shortly.

Our algorithm attempts to expand the exploratory regions around each outer node and contract the exploratory regions around each inner node. This is impossible in this case as the two outer nodes are touching. Instead, a cycle is formed as shown in Fig. 8(a). This cycle is collapsed to form a blossom, leaving an alternating tree with a single outer node that is a blossom containing three vertices.

The exploratory region around the sole outer node in the alternating tree is expanded until other objects are encountered [Fig. 8(b)]. An unmatched vertex and a boundary are encountered. Two options are available. We could match the edge from the original root vertex to the vertex below it, unmatch the existing matched edge, and then match the resultant unmatched vertex to the nearby boundary. Alternatively, we can match the original root vertex to the newly encountered unmatched vertex. Since this is simpler, we choose this option, the execution of which is shown in Fig. 8(c). The next unmatched vertex chosen is indicated by an arrow. In this case, no expansion of the exploratory region around the vertex is possible. One must instead immediately form an alternating tree consisting of three vertices [Fig. 8(d)].

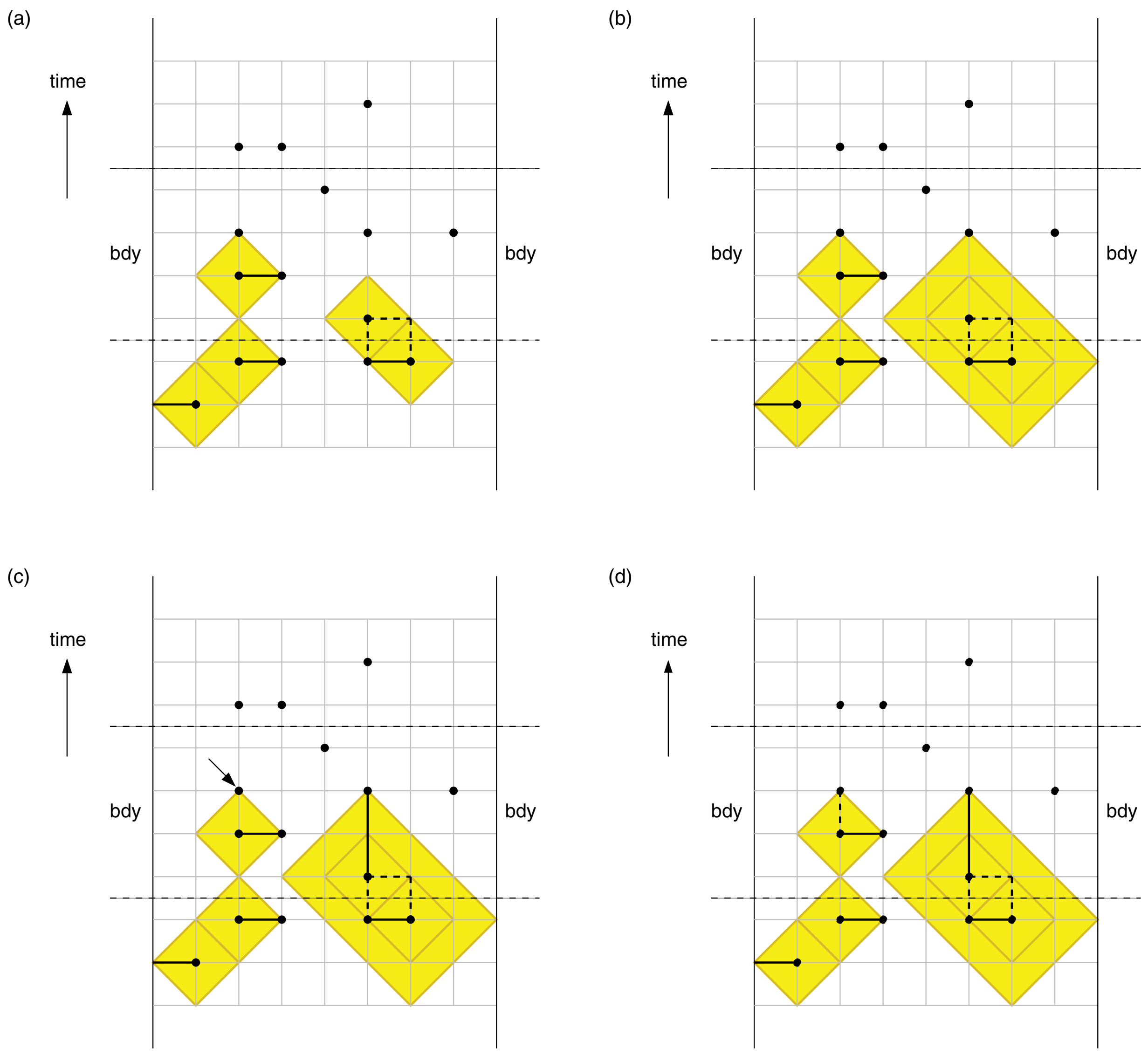

FIG. 8. (Color online) (a) Form blossom. (b) Expand exploratory region around blossom until other objects are encountered. (c) Match to unmatched vertex. Choose another unmatched vertex. (d) Form alternating tree.

The outer-node exploratory regions are expanded while the inner-node exploratory region is contracted [Fig. 9(a)]. This results in the outer-node exploratory regions touching, forming a cycle and thus a blossom [Fig. 9(b)]. This collapses the alternating tree to a single outer node consisting of a blossom containing three vertices.

The exploratory region around the single blossom outer node cannot be expanded, necessitating the creation of another alternating tree consisting of a blossom outer node, then a blossom inner node, then a vertex outer node [Fig. 9(c)]. The two outer exploratory regions can be expanded while the blossom inner-node exploratory region is contracted; however, this leads to exploration outside the active region [Fig. 9(d)]. When this happens, we run our algorithm backwards to the beginning of the current matching attempt, which in this case is Fig. 8(c).

This example hopefully gives a flavor of the algorithm. The salient features we wish to convey to the reader are the algorithm's space-time locality and continuous-processing nature. These features enable one to understand the parallelization of the algorithm. We shall explain this by analogy.

Imagine a box being filled with sand using a 2D array of tubes. Each tube represents a processor. Imagine that the rate of sand coming out of each tube represents the difficulty of the matching problem locally. A slower rate of flow implies higher local difficulty. The sand itself represents vertices that have been matched. The rate of flow of all tubes is set below the maximum possible—pauses are inserted in the algorithm such that it is possible for a tube to be run at greater than the standard rate should it be required. When the problem is locally hard, the rate of flow decreases and a hollow forms locally. When the difficulty of the problem returns to normal, which it must do on average, the rate of flow is increased above the standard rate to fill in this hollow. When the hollow is filled, the rate of flow is brought below the maximum possible again. Local difficulty does not result in global slowdown. Furthermore, surrounding tubes can assist in filling in the hollow. This simple picture explains how one can obtain a globally optimal solution of

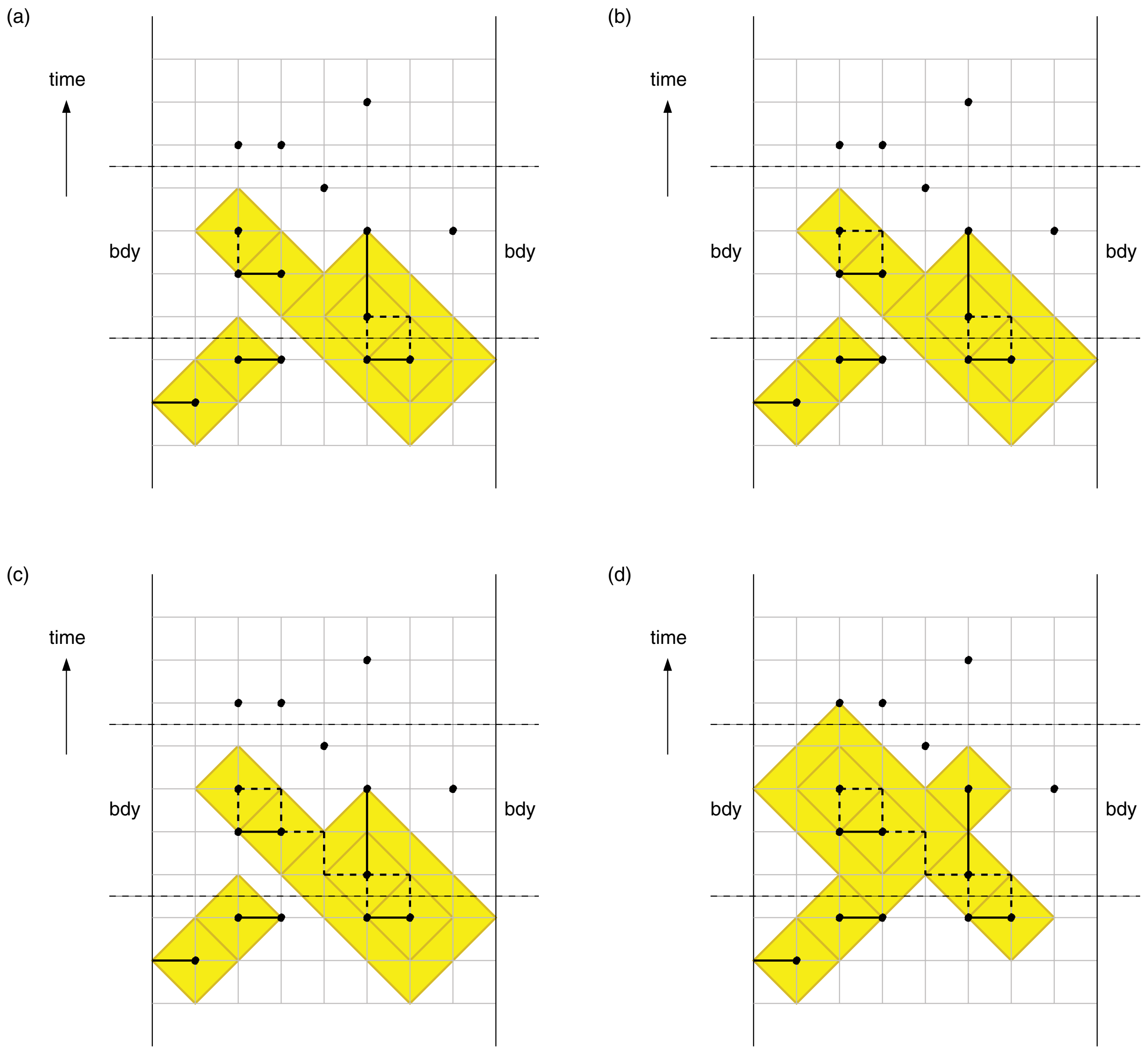

FIG. 9. (Color online) (a) Expand outer-node exploratory regions; contract inner-node exploratory region. (b) Form blossom. (c) Form alternating tree. (d) Expand outer-node exploratory regions; contract inner-node exploratory region. When forbidden region is entered, reverse algorithm execution back to Fig. 8(c). Wait for additional data.

an infinite-size problem in constant average time per round of processing, which is optimal.

Two other techniques for correcting errors in surface codes are being investigated, renormalization [15] and the Metropolis technique [16]. However, neither approach has been successfully applied to a realistic fault-tolerant case. Indeed, in the latest work of the authors of the renormalization approach, minimum weight perfect matching has been used to handle the fault-tolerant case [17]. We are not hopeful that any technique other than matching can be comparably fast and effective in the fault-tolerant case.

V. LOGICAL ERRORS

Strong evidence of the correctness of eodmatch comes from studying the probability of logical error per round of error correction (p_L) at depolarizing probabilities p well below threshold. We calculate p_L by simulating t_{check} rounds of faulty quantum computer operation, then turning off errors, capping the matching problem with a perfect round of error correction, applying corrections, checking whether we have an odd or even number of errors along one of the boundaries, and recording whether this is different from the previous time we checked. The perfect round of error correction is then undone and another t_{check} faulty rounds simulated and the process repeated.

It may seem that the ideal value of t_{check} is 1 to ensure that no logical errors are missed; however, this is not the case. We have observed that many combinations of errors lead to the observation of a logical error if a perfect round of error correction is inserted halfway through it, but no logical error if the perfect round of correction is sufficiently distant. With frequent checking this can mean that a benign pattern of errors is counted as several logical errors. Instead, we typically use a value of t_{check} such that a change in the parity of the number of errors observed along a boundary

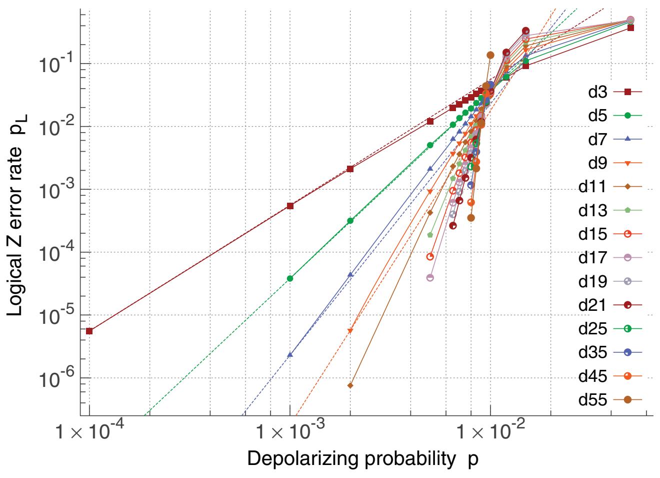

FIG. 10. (Color online) Logical Z error rate per round of error correction for surface-code distances d and depolarizing noise probabilities p. Dashed lines indicate expected low-p asymptotic curves for $d = 3$, 5, 7, and 9.

occurs approximately 10% of the time. We have empirically found that this leads to a logical-error-rate estimate robust to wide variations of t_{check} about this value. The probability of a change per check is equal to the probability of an odd number of logical errors in t_{check} rounds, enabling p_L to be easily calculated.

A distance-d code can reliably correct $\lfloor (d-1)/2 \rfloor$ errors. At low error rates p, clusters of errors are rare and well separated. The probability of suffering a logical error inducing a cluster of $n_d = \lfloor (d+1)/2 \rfloor$ errors should therefore be $O(p^{n_d})$ if the full distance of the code is being realized. Figures 10 and 11 show the complete set of data we have collected for the square-surface code. Polynomials $A_d p^{n_d}$ are drawn through the lowest data point we were able to obtain for distances 3, 5, 7, and 9.

It is computationally expensive to obtain statistics at very low error rates and high distances as very few logical state changes are observed. It is also computationally expensive to obtain data at high error rates and high distances as the minimum-weight perfect-matching problem becomes more difficult around and above the threshold error rate (0.9% [5]). The raw data used to generate Figs. 10 and 11, including timing information, can be found in the Supplemental Material [18].

The distance-3 and -5 dashed asymptotic curves in Figs. 10 and 11 agree very well with the data. For higher distances, it is not currently possible to simulate a sufficiently large number of rounds of error correction to obtain sufficient information at low enough probabilities to achieve such tight agreement. Note that the high-distance-data curves approach the asymptotic curves with a steeper gradient, implying that the surface code is capable of regularly correcting temporal clusters of errors containing more errors than the maximum guaranteed to be correctable. This is a generic feature of topological quantum error correction, as a large cluster of errors widely scattered across the code is not dangerous provided the cluster poorly resembles a topologically nontrivial chain of errors connecting distinct boundaries.

VI. TIMING

The timing information in the Supplemental Material [18] includes everything—initial booting up of the simulation, the simulation of the underlying quantum computer, problem generation, matching, perfect rounds of error correction to enable logical-state-change detection, and maintenance of an appropriate Pauli frame. Figure 12 shows the amount of time devoted to each round of matching alone at three different error rates for distances $d = 4, 8, 16, \ldots, 512$. The quadratic scaling of required time with distance is well demonstrated. At small d nearby boundaries prevent the growth of large blossoms, leading to increased performance. At very high d memory access effects lead to a slight slowdown. Note that real computer systems are too complex to provide perfectly smooth graphs of time scaling even with long time-averaging, as the interplay of different levels of cache and random-access memory leads to measurable deviations from the ideal scaling.

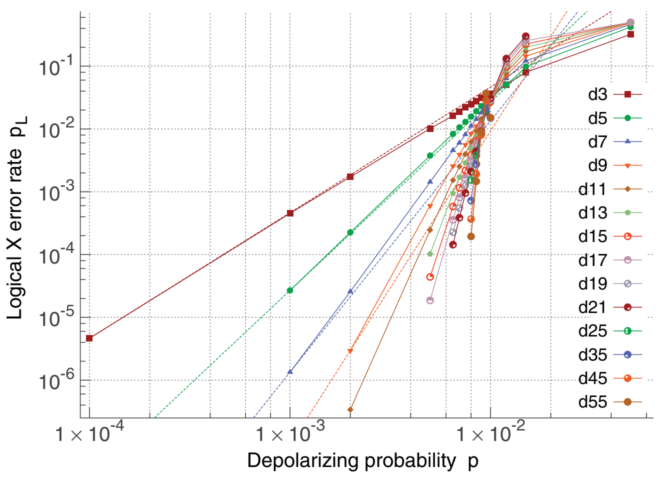

FIG. 11. (Color online) Logical X error rate per round of error correction for surface-code distances d and depolarizing noise probabilities p. Dashed lines indicate expected low-p asymptotic curves for $d = 3$, 5, 7, and 9.

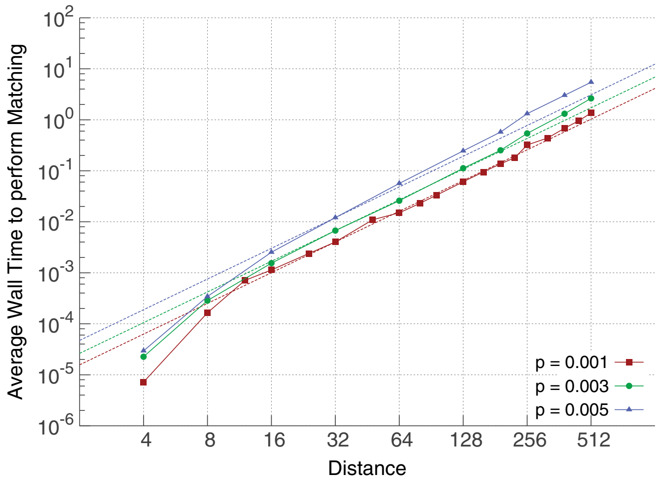

FIG. 12. (Color online) Amount of time in seconds devoted to each round of matching when simulating a distance-d single-logical-qubit square-surface code for depolarizing error rates p. Quadratic curves have been included for reference.

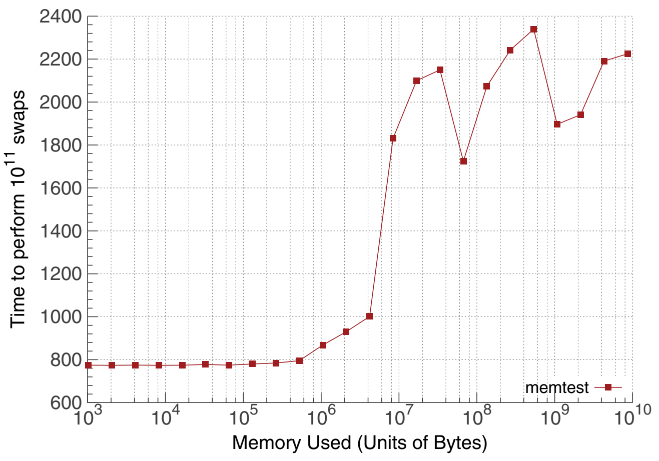

FIG. 13. (Color online) Average time in seconds required to perform 10^{11} swaps of randomly chosen pairs of integers in arrays of increasing size.

To illustrate the complexity of modern computer memory systems, we have generated increasingly large arrays of random integers and calculated the time required to swap a constant large number (10^{11}) of randomly chosen pairs of integers. The results are shown in Fig. 13. Ideally, a swap operation should be $O(1)$ independent of the array size. In practice, it can be seen that larger data sets lead to lower performance as the CPU cache is exceeded. The data in Fig. 13 were generated by 16 core Intel Xeon 3.33 GHz CPUs with 12 Mbytes of cache. Our matching code is more complex than this simple swap demonstration, with gradual delocalization of data as the data set increases in size. This leads to a gradual reduction of the probability of a single memory page load containing additional useful data.

VII. CONCLUSION

After accounting for low-distance nearby boundaries which limit the complexity of matching (making matching significantly faster) and high-distance slower memory access (leading to a slight reduction in performance), Fig. 12 provides strong evidence supporting the claimed $O(d^2)$ runtime of our implementation of the algorithm described in Ref. [5]. A major future goal is to parallelize the algorithm and demonstrate an average processing time per round of error correction independent of the code size, using constant computing resources per unit area.

ACKNOWLEDGMENTS

We acknowledge support from the Australian Research Council Centre of Excellence for Quantum Computation and Communication Technology (Project No. CE110001027), the US National Security Agency (NSA), and the Army Research Office (ARO) under Contract No. W911NF-08-1-0527.

[1] P. W. Shor, in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (IEEE Computer Society Press, Los Alamitos, CA, 1994), pp. 124–134; SIAM J. Sci. Stat. Comput. **26**, 1484 (1997).

[2] S. Lloyd, Science **273**, 1073 (1996).

[3] S. Jordan, http://math.nist.gov/quantum/zoo/.

[4] D. S. Wang, A. G. Fowler, and L. C. L. Hollenberg, Phys. Rev. A **83**, 020302(R) (2011).

[5] A. G. Fowler, A. C. Whiteside, and L. C. L. Hollenberg, Phys. Rev. Lett. **108**, 180501 (2012).

[6] S. B. Bravyi and A. Y. Kitaev, arXiv:quant-ph/9811052.

[7] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, J. Math. Phys. **43**, 4452 (2002).

[8] R. Raussendorf and J. Harrington, Phys. Rev. Lett. **98**, 190504 (2007).

[9] R. Raussendorf, J. Harrington, and K. Goyal, New J. Phys. **9**, 199 (2007).

[10] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Phys. Rev. A **86**, 032324 (2012).

[11] A. G. Fowler, A. C. Whiteside, A. L. McInnes, and A. Rabbani, arXiv:1202.6111 [Phys. Rev. X (to be published)].

[12] J. Edmonds, Can. J. Math. **17**, 449 (1965).

[13] J. Edmonds, J. Res. Natl. Bur. Stand. B **69**, 125 (1965).

[14] D. Gottesman, Ph.D. thesis, Caltech, 1997, arXiv: quant-ph/9705052.

[15] G. Duclos-Cianci and D. Poulin, Phys. Rev. Lett. **104**, 050504 (2010).

[16] J. R. Wootton and D. Loss, arXiv:1202.4316 [Phys. Rev. Lett. (to be published)].

[17] S. Bravyi, G. Duclos-Cianci, D. Poulin, and M. Suchara, arXiv:1207.1443.

[18] See Supplemental Material at http://link.aps.org/supplemental/10.1103/PhysRevA.86.042313 for the raw data used to generate Figs. 10 and 11.