Braidflash: Efficient Braid Simulation for Surface Code Quantum Error Correction

Ali Javadi-Abhari

January 9, 2018

This document describes the Braidflash software, which is a tool for efficient simulation of braids in the context of surface error correction of quantum applications. For further explanation and to cite this tool, please refer to the following publication:

A. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. R. Martonosi, F. T. Chong, **"Optimized Surface Code Communication in Superconducting Quantum Computers,"** IEEE/ACM MICRO, Cambridge, MA, 2017

1 Installation

1. Make sure you have the following dependencies installed:
```
Clang/Clang++ 3.9 (or later)
Boost 1.61 (or later)
Python 2.7.9 (or later)
Metis 5.1 (or later)
libgmp 6.1 (or later)
libffi 3.2 (or later)
```

2. Then from a terminal:
```
cd braidflash
make
```

The Braidflash simulator is now built and ready to use.

2 Overview

The braidflash software performs a physical-level simulation of the surface code operations, given information about a quantum application's logical-level characteristics.

2.1 Inputs:

The program requires the following inputs:

- Per-module logical schedule (`.lpfs` file)
- Coarse-grain module composition (`.cg` file)
- Module frequencies (`.freq` file):

These are all easily obtained by running `scripts/gen_lpfs.sh` on a given `.scaffold` application.

2.2 Outputs:

The outputs will be mainly reported in two files ending with `.kq` and `.br`. The following will be reported.

- Code distance used
- Num logical qubits
- Num physical qubits
- Num logical gates
- Num physical cycles
- Avg qubit manhattan cost.
- Network utilization factor.
- Histogram of braid lengths.
- Histogram of braid criticalities.
- Visualization of network state during various time slices.

2.3 Options:

- `--opt:` optimize qubit layout
- `--p:` physical error rate (10ˆ-p) [int] (default: 5)
- `--yx:` stall threshold to switch DOR routing from xy to yx [int] (default: 8)
- `--drop:` stall threshold to drop entire operation and reinject [int] (default: 20)
- `--tech:` technology [sup, ion, qdot] (default: sup)
- `--pri:` braid priority policy [0-6] (default: 0)

- `--visualize`: show network state at each cycle [Warning: only use on small circuits] (default: none)
- `--help`: display this help and exit
- `--version`: output version information and exit

3 Quick Start (Example)

Running a simulation is a two-step process. First, a logical schedule and logical-level information must be gathered. Then, the simulator can be called.

```
cd braidflash
# Generate logical-level schedules and information
../scripts/gen-lpfs.sh ../Algorithms/Square_Root/square_root.n10.scaffold
# Generate physical-level simulation metrics
./braidflash ./square_root.n10/square_root.n10.flat100k --p 5 --yx 8
--drop 18 --tech sup --pri 3
```

The simulation outputs will be written to the directory: `square_root.n10/braid_simulation`.

4 Code Explanation

The source code for the Braidflash simulator is written from scratch in C++ and Python. It is thoroughly commented and understandable by simple inspection. Below is a basic sketch of how it works.

The logical schedules are read to create a trace of the program for each leaf module. The qubit interactions specify an interaction graph which can be used to optimize qubit placements (this step is done through the `arrange.py` script). Similarly, the operation dependencies create a dependency graph which can be used to create a list of gate dependencies to simulate in order.

Each logical gate is broken down into multiple events. For example, a logical CNOT constitutes the following events:

1. Event $cnot_1$: opening ancilla nodes/link to initialize
2. Event $cnot_2$: closing ancilla link after 1 cycle
3. Event $cnot_3$: opening $route_1$ from source to destination after 1 cycle
4. Event $cnot_4$: closing $route_1$ after 1 cycle
5. Event $cnot_5$: opening $route_2$ from destination to source after minimum d-1 cycles
6. Event $cnot_6$: closing $route_2$ after 1 cycle

7. Event $cnot_7$: closing ancillas after minimum d-1 cycles

Each gate has its own queue of such events which need to be executed one by one. Globally, events may be interleaved (in effect, gates are not atomic—this breakdown of gates allows more efficient use of the lattice).

Two timers are dedicated to ensuring that braids don't get delayed for very long times: `attempt_th_yx`, `attempt_th_drop`. In these cases, the `resolve_cnot` function is called. This changes the route of the CNOT to (hopefully) allow it to complete without further delay.

Priority policies are based on how to select a braid for execution among many eligible ones (those that have dependencies met). These policies are as follows:

1. $Policy_0$: no priorities. in program order.
2. $Policy_1$: criticality only.
3. $Policy_2$: braid length only. short2long.
4. $Policy_3$: braid length only. long2short.
5. $Policy_4$: close2open only.
6. $Policy_5$: crticiality + short2long + close2open
7. $Policy_6$: criticality + short2long (highest crit) + long2short (lower crit) + close2open

5 Visualizing Braids

Tip: For better visualization, you can configure your editor to color `.br` files, showing the actual braid occupancies on the network at each timestep. Below are steps to do this in the Vim editor:

1. Add the following to the your `.vimrc` file:

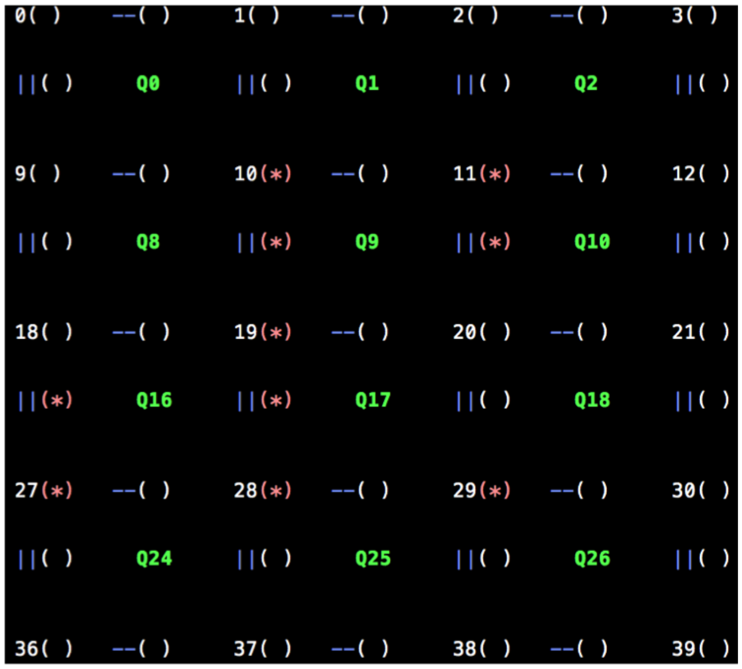
   ```
   autocmd BufRead,BufNewFile *.br set filetype=br
   ```

2. Create `.vim/syntax/br.vim` with the following content:

   ```
   " Vim syntax file
   " Language:     Braids on Quantum Surface Code

   if exists("b:current_syntax")
     finish
   endif

   syn region braidLink start="-" end="-" keepend
   syn region braidLink start="|" end="|" keepend
   ```

```
syn region braidBusy start="(\*" end=")" keepend
syn region  braidQbit start="Q" end="\t"  keepend
syn region  braidClock start="CLOCK" end="\n" keepend

let b:current_syntax = "br"

hi def link braidLink Comment
hi def link braidBusy String
hi def link braidClock Label
hi def link braidQbit Type
}
```

Below is a sample screenshot to illustrate:



```
0( )    --( )   1( )    --( )   2( )    --( )   3( )

||( )   Q0      ||( )   Q1      ||( )   Q2      ||( )


9( )    --( )   10(*)   --( )   11(*)   --( )   12( )

||( )   Q8      ||(*)   Q9      ||(*)   Q10     ||( )


18( )   --( )   19(*)   --( )   20( )   --( )   21( )

||(*)   Q16     ||(*)   Q17     ||( )   Q18     ||( )


27(*)   --( )   28(*)   --( )   29(*)   --( )   30( )

||( )   Q24     ||( )   Q25     ||( )   Q26     ||( )


36( )   --( )   37( )   --( )   38( )   --( )   39( )
```

Figure 1: Visualization of braid occupancies on the network at a given timestep. Green (Qxx) indicates the locations of logical qubits; each logical qubit is implicitly a double hole. White numbers are network routers (qubit corners) and blue lines are network links. Red asterisks indicate occupied links and nodes, thus the path of a braid.