CircuitSimulator Documentation

Patrick Rall, Iskren Vankov, David Gosset

June 20, 2017

Simulation of quantum circuits via the conventional matrix-multiplication approach suffers from an exponential blowup in the number of qubits. Medium size quantum circuits with 20 qubits or more are nearly impossible to simulate this way. On the other hand the Gottesmann-Knill theorem gives a polynomial-time algorithm for simulation of quantum circuits composed entirely of the Clifford operations:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \quad S = \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} \quad CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix} \tag{1}$$

In January 2017, Sergei Bravyi and David Gosset augmented the algorithm from the Gottesmann-Knill theorem to also permit simulation of the T gate:

$$T = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix} \tag{2}$$

Their modification retains the polynomial scaling in both the number of qubits and the number of Clifford operations, allowing it to simulate large quantum circuits. The only exponential parameter is the total number of T gates used. Clifford+T is a well-studied universal set of gates, and many interesting quantum circuits are written in Clifford+T.

For a full explanation of the algorithm please see the original paper by Bravyi and Gosset at `http://arxiv.org/abs/1601.07601`. The purpose of this document is to illustrate how to use the implementation of the algorithm available at `https://github.com/patrickrall/CircuitSimulator`. The goal is that a full understanding of the algorithm is not required to use the software.

Contents

1 Installation

The application is written in Python. Python is easy to read even for non-programmers and is available on all platforms. A full implementation of the algorithm in Python should be easy to install on all operating systems, and matches the pseudo-code in `http://arxiv.org/abs/1601.07601` almost line-by-line.

The most time consuming part of the algorithm is the calculation of inner products $|\Pi \, |H^{\otimes t}\rangle \, |^2$, where Π is a projector that depends on the circuit, and $|H^{\otimes t}\rangle$ is a particular quantum state. This highly parallelizable task also has an implementation in C, designed to run on computer clusters using OpenMPI.

1.1 Python Implementation

The application was developed in Python 3, although tests showed that Python 2 worked also. Download and install the following, either using the links provided or your favorite package manager:

Python 3: `https://www.python.org/downloads/`

numpy: `https://pypi.python.org/pypi/numpy`

matplotlib: `http://matplotlib.org/users/installing.html`

Matplotlib is not strictly required for the algorithm itself, but several testing scripts use matplotlib to visualize results.

1.2 OpenMPI Implementation

OpenMPI is an implementation of Message Passing Interface, a tool that allows many processes to communicate on a cluster. Small numbers of T-gates can be simulated using Python, but for larger numbers the OpenMPI version of the algorithm will be required, either on a single machine or on several.

Download OpenMPI here: `https://www.open-mpi.org/`
Once OpenMPI is installed, C binaries can be compiled using *mpicc* and executed using *mpirun*. To compile the implementation simply change into the root directory and type *make*.

1.3 Usage

2 Performance

3 Approximations

To achieve the fast scaling, the algorithm uses several complicated approximation tricks. You do not need to understand these in order to use the application. The default options make a reasonable compromise between accuracy and performance.

However, the approximations are fully configurable. If you have specific accuracy or performance requirements this section will be helpful.

3.1 Algorithm background

The most of the work in the algorithm is to calculate $|\Pi\,|H^{\otimes t}\rangle\,|^2$, where Π is a projector that depends on the circuit. $|H\rangle = \cos(\pi/8)\,|0\rangle + \sin(\pi/8)$ is the eigenstate of the Hadamard gate, and is a 'magic' state that is used to implement T-gates.

The CircuitSimulator application implements efficient manipulations of so-called stabilizer states. $|H\rangle$ is not a stabilizer state, but we can write $|H^{\otimes 2}\rangle$ as a linear combination of two stabilizer states. To write $|H^{\otimes t}\rangle$ we therefore require a linear combination of $\chi = 2^{\lceil t/2\rceil}$ stabilizer states: $|H^{\otimes t}\rangle \propto \sum_{i=1}^{\chi} |\phi_i\rangle$. We don't need to worry about normalization since we are calculating the ratio of two such inner products. Expand $|\Pi\,|H^{\otimes t}\rangle\,|^2$:

$$|H^{\otimes t}\rangle = \sum_{i=1}^{\chi} |\phi_i\rangle \quad \rightarrow \quad |\Pi\,|H^{\otimes t}\rangle\,|^2 = |\,\langle H^{\otimes t}|\,\Pi\,|H^{\otimes t}\rangle\,| \propto \left|\sum_{i=1}^{\chi}\sum_{j=1}^{\chi} \langle\phi_i|\,\Pi\,|\phi_j\rangle\right|.$$

There are $\chi^2 = 2^t$ terms in this sum, so this algorithm runs in $O(2^t)$. This calculation is so slow that Bravyi and Gosset did not include it in their paper. Instead, they developed several approximate techniques that can calculate output probabilities to arbitrarily small error.

3.2 Inner product approximation

To calculate inner products in $O(\chi)$ time rather than $O(\chi^2)$, the algorithm samples $\langle\theta|\,\Pi\,|H^{\otimes t}\rangle$, where $|\theta\rangle$ is a random stabilizer state. Define a random variable ξ which is the average of N samples from this distribution:

$$\xi = \frac{2^t}{N}\sum_{i=1}^{N} \left|\langle\theta_i|\,\Pi\,|H^{\otimes t}\rangle\right| \quad \rightarrow \quad \mathbb{E}(\xi) = |\Pi\,|H^{\otimes t}\rangle\,|^2, \quad \sigma = \sqrt{\frac{2^t-1}{2^t+1}}\frac{1}{\sqrt{N}}|\Pi\,|H^{\otimes t}\rangle\,|^2$$

The Chebychev inequality guarantees a multiplicative error of at most ε with failure probability p, provided N is large enough:

$$N = \sqrt{\frac{2^t-1}{2^t+1}}\frac{1}{\varepsilon p^2} \quad \rightarrow \quad \left|\xi - |\Pi\,|H^{\otimes t}\rangle\,|^2\right| \leq \varepsilon|\Pi\,|H^{\otimes t}\rangle\,|^2$$

This can be further improved via the median-of-means trick. The software can take the median several bins, each a sample from ξ, lowering the failure probability p without needing to increase L. Median-of-means can only decrease p and not decrease ε, and is only worth it for $p < 0.0076$. See the comment in *libcirc/innerprod.py* for details.

The number of samples and bins can be set manually. However, you can also set ε and p and have the software make the best choice for you.

Configuration

- This approximation is done by default. Turn it off using the -NOSAMPLING flag (In the code, this switch is called *noapprox*).
- Set the number of samples L directly using the SAMPLES= option. $L = 2000$ is the default and guarantees $\varepsilon = 0.2$ with $p = 5\%$ for large t.
- Set the number of bins directly using the BINS= option. Default is 1 bin.
- Set the multiplicative error ε with ERROR=. Overrides SAMPLES= and BINS=.
- Set the failure probability p with FAILPROB=. Overrides SAMPLES= and BINS=.

3.3 $\mathcal{L}$ approximation

The state $|H^{\otimes t}\rangle$ can be exactly decomposed into $2^{\lceil t/2 \rceil}$ stabilizer states. It can also be approximately decomposed into 2^k stabilizer states, where k can be smaller than $t/2$. The trick is to replace $|H^{\otimes t}\rangle$ with a similar state $|\mathcal{L}\rangle$, which is a function of a $t \times k$ binary matrix $\mathcal{L}$ of rank k.

$\mathcal{L}$ is sampled at random. The approximation has fidelity $|\langle H^{\otimes t}|\mathcal{L}\rangle|^2 = 1 - \delta$, where δ is a function of $\mathcal{L}$. k can be set manually, or calculated for a given expected value of δ. For fixed δ and large t, the scaling is $k \approx 0.23t$. But for $t < 65$ or less and $\delta = 10^{-5}$, we have $k > t/2$ so this approximation is only worth it for large t.

The approximation incurs a constant error $|P_{\text{out}} - P_{\text{correct}}| < \sqrt{\delta}$. This is fine when sampling from the circuit output distribution, but may be unsuitable for calculating probabilities directly.

Calculating δ is expensive, so it is not done by default. Calculating the rank of $\mathcal{L}$ is expensive, so it is not done by default. A random binary matrix usually has full rank.

Configuration

- This approximation is on by default when sampling from the circuit output distribution. It is off by default when calculating probabilities. Force it off using the -EXACTSTATE flag (In the code, this switch is called *exact*). Force it on by setting k or δ as described below.

- Set k directly using the K= option. Software will not override if $k > t/2$, but will override if $k > t$ because then it is impossible for $\mathcal{L}$ to be full rank.
- Set expected δ using the FIDBOUND= option. Default is $\delta = 10^{-5}$. Overridden by K=. If the automatically chosen k is larger than $t/2$, the approximation is disabled.
- Calculate and print δ with the -FIDELITY flag. If FIDBOUND= is set, $\mathcal{L}$ is repeatedly sampled until δ is good enough.
- Calculate the rank of $\mathcal{L}$ with the -RANK flag. Will repeatedly sample $\mathcal{L}$ until it has rank k.

4 Code Map

Here is an outline of all files and functions in the code.

4.1 Front end

These functions are written in Python only.

main (main.py) Command line interface.
Compiles circuit from file using **compileCircuit** (libcirc/compile/compilecirc.py).
Parses arguments and feeds them to **probability** (libcirc/probability.py)
or **sampleQubits** (libcirc/sample.py).

sampleQubits (libcirc/sample.py) Samples from a circuit's output distribution.
Repeatedly calls **probability** (libcirc/probability.py).

4.2 Compilation and Gadgetization

Written in Python only. Together these functions take a circuit file as described in section 6, and calculate projectors Π'. Then all that is left to do is calculate $|\Pi' |H^{\otimes t}\rangle|^2$.

compileCircuit (libcirc/compile/compilecirc.py) Compiles "main" in a given file.
Calls **circuitSymbols** (same file) to extract "main" and all of its dependencies. Then calls **compile** (same file) to return a self-contained circuit.

circuitSymbols (libcirc/compile/compilecirc.py) Extracts functions from a circuit file.
Recursively iterates over included files. Yields a dictionary of functions, noting their dependencies. Dictionary is complete in the sense that no dependency is left out. Calls **commentParse** (same file) to remove comments.

compile (libcirc/compile/compilecirc.py) Takes a dictionary of symbols as outputted by **circuitSymbols** (same file). Takes a symbol from that dictionary and compiles it to a completely self-contained circuit using only the gates in **gateset** (same file). Uses **standardGate** (same file) to help parse lines of code.

projectors (libcirc/compile/projectors.py) Given bitstrings x and y and a circuit C, calculates the two projectors $\Pi_G = C(|x\rangle \langle x| \otimes |y\rangle \langle y|)C^\dagger$ and $\Pi_H = C(\mathbb{I} \otimes |y\rangle \langle y|)C^\dagger$. Calls **standardGate** (libcirc/compile/compile.py) and **countY** (libcirc/compile/gadgetize.py) to count the number of bits needed for x, y. If x, y are unspecified it chooses them at random. Calls **gadgetize** (libcirc/compile/gadgetize.py) to actually do the work.

gadgetize (libcirc/compile/gadgetize.py) Given bitstrings x and y and a circuit C, calculates the projector $\Pi_{x,y} = C(|x\rangle \langle x| \otimes |y\rangle \langle y|)C^\dagger$.

truncate (libcirc/compile/projectors.py) Given a projector Π with support on both n circuit qubits and t magic state qubits, truncates the circuit qubits by calculating $\Pi' = (\langle 0^{\otimes n}| \otimes \mathbb{I})\ \Pi\ (|0^{\otimes n}\rangle \otimes \mathbb{I})$. The output projector Π' only has support on the magic state qubits.

4.3 Back end

These functions are written in Python and in C. File names and function names are identical. '.*' denotes '.c' or '.py', and sometimes '.h'.

probability (libcirc/probability.*) Calculates the probability of making a given measurement from a compiled circuit. Takes *config* dictionary with many options.

1. The Python version calculates projectors Π'_G and Π'_H using **projectors** and **truncate** (libcirc/compile/projectors.py).
2. Given the inner product approximation parameters (section 3.2), the number of samples and bins are calculated.
3. If using the C back end, the Python version passes the projectors and options to the C executable via the *subprocess* module. The C implementation parses the projectors and inputs. Now either the C implementation or the Python implementation proceeds.
4. **decompose** (same file) calculates $\mathcal{L}$ if needed. If $\chi = 2^k$ or $2^{\lceil t/2 \rceil}$ is smaller than the total number of samples, inner product sampling (section 3.2) is disabled.
5. $|\Pi'_G |H^{\otimes t}\rangle|^2$ and $|\Pi'_H |H^{\otimes t}\rangle|^2$ are calculated. If using the inner product approximation (section 3.2), it uses **multiSampledProjector** (libcirc/innerprod.*). Otherwise it uses **exactProjector** (libcirc/innerprod.*).
6. Finally it outputs the probability $P = |\Pi'_G |H^{\otimes t}\rangle|^2 / |\Pi'_H |H^{\otimes t}\rangle|^2$, returning control to Python if necessary.

decompose (libcirc/probability.*) Decomposes $|H^{\otimes t}\rangle$ into $|\mathcal{L}\rangle$ as needed in the $\mathcal{L}$ approximation (section 3.3). Calculates k given δ and t if needed. Can also decide to disable the approximation if there is no benefit. Calculates the matrix $\mathcal{L}$ if needed and checks rank and δ if requested.

multiSampledProjector (libcirc/innerprod.*) Given a projector Π, calculates $|\Pi\,|H^{\otimes t}\rangle\,|^2$ using the inner product approximation (section 3.2). Takes the median of several ξ bins, using **sampledProjector** (same file) to calculate ξ.

sampledProjector (libcirc/innerprod.*) Calculates ξ (section 3.2), by averaging many samples of $|\,\langle\phi|\,\Pi\,|H^{\otimes t}\rangle\,|$ where $|\phi\rangle$ is a random stabilizer state. Runs in parallel over the samples using **singleProjectorSample** (same file) to sample $|\,\langle\phi|\,\Pi\,|H^{\otimes t}\rangle\,|$.

singleProjectorSample (libcirc/innerprod.*) Samples $|\,\langle\phi|\,\Pi\,|H^{\otimes t}\rangle\,|$ where $|\phi\rangle$ is a random stabilizer state. Uses the stabilizer library (libcirc/stabilizer/stabilizer.*). $|H^{\otimes t}\rangle$ is decomposed into $2^{t/2}$ stabilizer states, which are prepared using **prepH** (libcirc/stateprep.*).
If using the $\mathcal{L}$ approximation (section 3.3), it calculates $|\,\langle\phi|\,\Pi\,|\mathcal{L}\rangle\,|$. $|L\rangle$ is decomposed into 2^k stabilizer states, which are prepared using **prepL** (libcirc/stateprep.*).

exactProjector (libcirc/innerprod.*) Given a projector Π, calculates $|\Pi\,|H^{\otimes t}\rangle\,|^2$ without sampling. Continuing from the equation in section 3.1, now using zero-indexing:

$$|\Pi\,|H^{\otimes t}\rangle\,|^2 = \left|\sum_{i=0}^{\chi-1}\sum_{j=0}^{\chi-1}\langle\phi_i|\,\Pi\,|\phi_j\rangle\right| = \left|\sum_{i=0}^{\chi-1}\left(\langle\phi_i|\,\Pi\,|\phi_i\rangle + \sum_{j=i+1}^{\chi-1} 2\mathrm{Re}(\langle\phi_j|\,\Pi\,|\phi_i\rangle)\right)\right|$$

Runs in parallel over the j sum, using **exactProjectorWork** (same file) to calculate the term in big parentheses.

exactProjectorWork (libcirc/innerprod.*) Given an index i, calculates a term in the i sum above. If the $\mathcal{L}$ approximation (section 3.3) is used, $|\phi_{i/j}\rangle$ are prepared using **prepL** (libcirc/stateprep.*). Otherwise **prepH** (libcirc/stateprep.*) is used.

prepH (libcirc/stateprep.*) Given an index $0 \leq i < 2^{\lceil t/2\rceil}$, prepares a state $|\phi_i\rangle$ such that $|H^{\otimes t}\rangle \propto \sum_{i=0}^{2^{\lceil t/2\rceil}-1}|\phi_i\rangle$. It achieves this using the decomposition $|H^{\otimes 2}\rangle \propto |\alpha\rangle + |\beta\rangle$ where $|\alpha\rangle, |\beta\rangle$ are stabilizer states. $|\phi_i\rangle$ is a tensor product of several $|\alpha\rangle$ and $|\beta\rangle$.

prepL (libcirc/stateprep.*)
Given $\mathcal{L}$ and an index $0 \leq i < 2^k$, prepares a state $|\phi_i\rangle$ such that $|\mathcal{L}\rangle \propto \sum_{i=0}^{2^k-1}|\phi_i\rangle$.

4.4 Stabilizer library

The files libcirc/stabilizer/stabilizer.* contain several functions for manipulating stabilizer states. These are explained in the appendix of `http://arxiv.org/abs/1601.07601`, but are listed here for completeness. Instead of using a tableau representation, it uses an affine subspace $\mathcal{K} \subset \mathbb{F}_2^n$ of dimension k and a quadratic form q such that $|\phi\rangle = |\mathcal{K}|^{-1/2} \sum_{x \in \mathcal{K}} e^{i \frac{\pi}{4} q(x)} |x\rangle$. In Python, a stabilizer state is a class. In C, a stabilizer state is a struct with a set of functions.

printStabilizerState (C function) and **display** (Python method) print Python code that initializes the state. Very useful for debugging.

unpack (Python method only) prints a state vector representation of the state.

exponentialSum $O(k^3)$. Calculates the sum $\sum_{x \in \mathbb{F}_2^k} e^{i \frac{\pi}{4} q(x)}$. Needed by **innerproduct**. Output is a complex number that can always be represented exactly by three integers ε, p, m, so an exact version outputs these integers instead to avoid numerical error.

shrink $O(kn)$. Given a vector ξ and bit α, shrinks the space $\mathcal{K}$ so $x \cdot \xi = \alpha$ for all $x \in \mathcal{K}$.

innerProduct $O(n^3)$. Calculates the inner product between two stabilizer states. The complex output can also be written in terms of integers ε, p, m.

randomStabilizerState Average case $O(n^2)$, worst case $O(n^3)$. Samples a random stabilizer state by starting with $\mathcal{K} = \mathbb{F}_2^n$ and randomly calling **shrink**.

measurePauli $O(n^2)$. Given a Pauli matrix $P = i^m X(\xi) Z(\zeta)$ in the form of an integer m and two bit vectors ξ, ζ, applies the projector onto the positive eigenspace $(1+P)/2$ to the state and returns the new state and its normalization. Uses a subroutine **extend** to add new vectors to $\mathcal{K}$.

4.5 Utilities

These files are written in C only.

- libcirc/utils/matrix.h is a custom matrix library for manipulating bit vectors. matrix.c implements these by hand using bitwise operations, and matrix-blas.c implements them using the blas library.

- libcirc/utils/comms.h implements wrappers around MPI calls to pass data between threads. It also implements a struct for Pauli projectors.

comms.h includes matrix.h, and stabilizer.h includes comms.h. C files include stabilizer.h when stabilizer operations are needed, and include comms.h otherwise.

5 Measurement

6 The Circuit Language

Lorem ipsum.