University of Sheffield

Masters Project

Simulation of Qubit Cluster Creation

Author:
Josh Kettlewell
090155185

Supervisor:
Dr. Pieter Kok

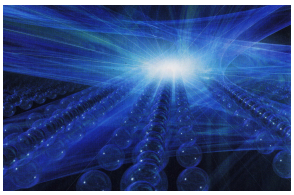

Figure 1: ©New Scientist, Vol 2544, 25.03.2006.

Spring 2012

Abstract

Cluster states, first introduced by Raussendorf and Briegel [**?**] and initially conceived for optical lattices and linear optical computing, are essential entanglement resources which can be used for quantum computing through the use of linear operators. They are created via entangling operations on qubits with an arbitrary probability of success. As there is a lower limit placed upon the lifetime of qubits, and thus cluster states, we can represent a threshold for the qubit decay time in various dimension cluster states [**?**].

1 Introduction

hello

Quantum computers have attracted interest since [**?**] for there ability to find prime factors [**?**] for using in breaking common encryption methods such as RSA [**?**]. A new and interesting method for making building quantum computer is the one way model, designed around measurement based quantum computing [**?**] wherein measurements are made on a single qubit which is part of a group of entangled qubits known as a cluster state. These are regular graphs in at least two dimensions and can be created using a variety of entanglement gates [**?**], and though most gates are probabilistic it has been shown that universal quantum computing can be achieved with an arbitrarily low entanglement probability gate. Although the theory of using the created clusters is well established [**?**], the creation of the states in an efficient way presents challenges due to the aging of the qubits during the computation.

As the qubits will be aging during the construction of the cluster state and the computation, an efficient way to compute is the "just-in-time" method, shown in [**?**] . In each time step the qubits on the far left are measured in each time step of Δt, and thus removed from the cluster, to pass information in the computation, meanwhile mini-clusters of length m are added in order to maintain the cluster at a consistent size. Each time step of Δt is of the magnitude of time required to preform an entanglement procedure with any two arbitrary qubits. It is presumed that this procedure can be undertaken in parallel. N is the "buffer size" which, if depleted, will cause the computation to crash as there are no longer any qubits to which the information may be passed. ΔN is the variance of the buffer zone and M is the number of logical qubits used in the computation. This creates a temporal flow of information from the left to the right. As the entanglement of the mini clusters of length m to the buffer is probabilistic, the size of m is dependent on the entanglement probability. M is the depth of the cluster state and equal to the number of logical qubits.

As all qubits decohere in time regardless of type, we have a limit on the time from the qubit initialisation to its measurement which drives the computation. This decoherence time is given as T_2. In this paper we give a lower bound value for T_2 required for the implementation of a measurement based quantum computer given general applications of the just-in-time method.

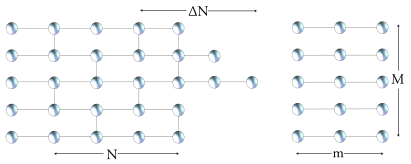

Figure 2: *Just-in-time* computing[5]

Before computation may take place it is first required that the cluster state be built.

2 Theory

Optical quantum computing is reliant on the production of indistinguishable photons from discrete qubits for the production of two photon interference by by entanglement is achieved.. This is a challenge however has been shown possible. [**?**]

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \ , Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \ , Z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \ .$$

$$U_{cz} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \text{ and } U_{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \ .$$

The one way model of quantum computing, birthed by [**?**] is dependent on entanglement of quantum states. In linear optical quantum computing, the this regime relies on the two-photon interference.

Single photons ideal carries of quatum information as they do not decohere as readily as other qubits (such as NV centres) and travel much quicker. But as the do not interact directly, sagacious schemes must be untertaken to produce a quantum computer, the most well known of which being the KLM approach.
The KLM (Knill, Laflamme and Milburn) is a feed forward wherein we are able achieve entanglement by rejecting or accepting certain terms in an output state. In the KLM scheme, a probablistic gate may be used to entangle two qubits without destroying them. This is due to the *teleportation trick* [cite Gottesman and Chuang] which allows entanglement gates to be "teleported" into the circuit using a previously entangled state. This is used to make gates which act on single photons as two qubit gates []. However, as it is not possible to make a Bell measurement require in the teleportation trick on single photons, the protol only suceeds with a sucess of $\frac{n}{n+1}]^2$. Should the gate fail this is analogous to a measurement in a Bell state which may be protected against using error correction methods.

The Hong-Ou-Mandel effect [] is the result of two identical photons each entering a 50:50 splitter. We would expect that the each photon independently "chooses" its own exit and this half the time both photons would come out of the same output and half the time they would exit from different outputs. However, quantum mechanically this is what not we observe. We instead note that the both exit from the same port all of the time due to a suppression of the other terms. This is a pillar of KLM optical computing.

Although the HOM effect is used with a beam splitter, it may also be prudent to use a the polarization of qubits to form entanglement. This may be achieved using a polarizing beam splitter using the two methods of Type I and Type II fusion. Here we make single photon detection and erase polarization information using a 45 degree rotation.

In type I fusion only one of the output ports is detected whereas in type II both are detected. Both require an Bell state $|H,V\rangle + |V,H\rangle$.

A type I fusion gate is a polarisation beam splitter with a horizontal and vertical modes followed by a 45 degree rotation on one of the output modes and finally a detection in the $|H,V\rangle$ basis. The type II fusion gate consists of a diagonal polarization beam splitter and both outputs are detected.

Fusion gates differ from normal entanglement gates in that it it is not possible to input a separable state and receive and entangled state. As type II fusion gates do not generate any output they can infact only be used for the one way model of quantum computing.

Double-heralding protocol This is an entanglement operation wherein qubits may couple out of the quantum memory with a high enough probability to pass the fault tolerance threshold. This is achieved via a projective parity measurement proceedure similar two type II fusion gates.
The measurement of the qubits allows the transfer of information due to their entanglements. Here the red and blue bands represent a single logical and a single-qubit measurement respectively. The information is *pushed* to the right by the measurement which takes place from the top to the bottom of the leftmost column. In order to apply an arbitrary rotation $r(\theta)$ to a qubit eigenvalue, we require measurement;
where $|\psi_{out}\rangle = (X^m HZ(\gamma))(X^l HZ(\beta))(X^k HZ(\alpha))\,|\psi\rangle$. X^k, X^l and X^m are dependent on the realised values and commuting them through the Pauli and Hadamard gates will remove them. This leads to an adjustment of the

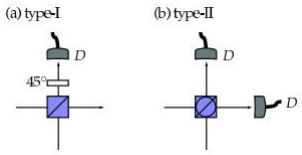

Figure 3: DJRBG

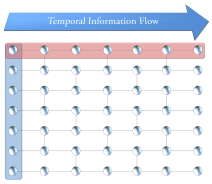

Figure 4: Information flow in a universal cluster state[3]. Circles present qubits in state $|+\rangle$ and lines represent entanglement

measurement bases contingent to the previous measurement outcomes, thus a direction of information flow.

There are several kinds of entangling gate, all of which have different entanglement methods and reduce the number of unentangled qubits in a successful of unsuccessful entanglement.

Entangling Procedure	c_1	c_2
Type-I fusion	2	2
Type-II fusion	3	1
Double-heralding fusion	1	1
Repeat-until-success	1	0
Broker-client model	0	0

Table 1: The entangling procedures[6] and there respective values of c_1 and c_2

For quantum computing to become a viable technology, we need a method by which a system can store the qubit value of a photon. This is because, although we can post-selection and feed forward to modify interferometry, we need to keep the qubits from when they are first entangled to their measurement. It is possible to store a qubit in a fibre-optical delay line, the losses of the fibre remove this for being a solution in a complete computer and the memory time is likely to be greater than Δt. Thus to use single photons another method must be utilised. It would be possible to couple the photon with a cavity however this would remove the advantage of single photon resistance to decoherance and they will decohere in the moemry. Instead we need to use *matter qubits* where single qubit operations can be preformed.

2.1 One Dimensional Clusters

We state that creating a minicluster of size m takes, on average, a time τ simply for ease. The lifetime of the qubit at the time of measurement is thus

$$\tau_{average} = \tau + (\langle N \rangle + m)\Delta t. \tag{2.1}$$

Where $\tau_{average}$ is the average age of a qubit at the time of its computation.

A failed entanglement will require us to re-purify the cluster by measuring a subset of qubits, whereas a successful entanglement will either create a *cherry* (a qubit which attaches to a single qubit on the cluster chain), or add a

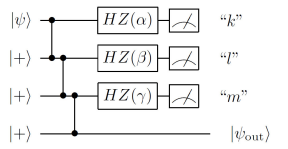

Figure 5: One way model[4]

minicluster to the cluster. c_1 is the number of qubits that do not add to the buffer in a successful entanglement, and c_2 is the number of qubits that are removed from the cluster in the case of an unsuccessful entanglement. Thus the *rate of growth* is

$$R = p(m - c_1) - (1 - p)c_2 - 1, \tag{2.2}$$

where p is the probability of a successful entanglement. For zero growth we require $R = 0$. $\tau_{average}$ must be much less than T_2, the age of the leftmost qubit, so

$$T_2 = \alpha(\tau + (\langle N \rangle + m)\Delta t), \tag{2.3}$$

where α is the *fault tolerance factor*, with $\alpha \geq 1$ and determined by the fault tolerance threshold of a given quantum computer. This is the maximum error which can be allowed, above which it is no possible to use error correction code to save the computation. The value of R is probabilistic, thus the buffer must be sufficiently long as not to disappear in a run of unsuccessful entanglements. The *buffer factor* β is given by

$$\langle N \rangle = \beta \Delta N. \tag{2.4}$$

The fluctuation size of the buffer can be calculated using a 1 dimensional random walk argument due to the probabilistic aspects;

$$(\Delta N^2) = \frac{(1 + c_2)^2}{p} - c_2(2 - c_2). \tag{2.5}$$

Buffer factor β is dependent on the number of logical qubits and the logarithm of the computation time. Thus

$$\begin{aligned} T_2 &= \alpha(\tau + \beta\Delta N + m) \\ &= \alpha(\tau + \sqrt{\frac{(1 + c_2)^2}{p} - c_2(2 - c_2)} + \frac{1 + pc_1(1 - p)c_2}{p}) \\ &= \alpha(\tau + \beta\sqrt{\frac{4 - p}{p}} + \frac{2}{p}) \quad \text{for Double-Heralding gates.} \end{aligned} \tag{2.6}$$

Figure 5 shows, for a conservative estimate of τ, the threshold of T_2 for the broker-client model (lower curve) and double heralding (upper curve) entanglement methods. Shaded areas define where qubit decay forbids quantum computation utilising cluster states created via entangling gates with p probability of success. No lower limit of success probability p exists.

This model assumes qubits are cheap, to the extent that it is simple to create a (very) large number of qubits, and that arbitrarily entangling two qubits can be achieved quickly. The optimal entangling gate and procedure is something that the project has yet to conclude and is one of the objectives of the project.
We will also compare current experimental values of T_2 times to our results.

2.2 more theory

3 The code

The purpose of this project was to create a program that will simulate to entanglement of single qubits into a cluster, and the entanglement of clusters of qubits into larger clusters. The aspiration of this is to evaluate what probability of entanglement an entanglement gate must have, and what number of qubits will be needed, in order to build a one way model quantum computer. The initial goal of the project is to construct the clusters in both 1, 2 and 3 dimensions.

4 The program

The system works by using entangling procedures, representing entanglement gates, to entangle qubits into "clusters". These clusters consist of several entangled qubits in possibly several dimensions of space. As the entanglement gates act in a probabilistic method there are several entanglement strategies which may be employed, all of which leading to different size qubits in a given time. This follows the "just-in-time" creation of cluster states[] whereby the qubits are used in an efficient fashion a short time before they are expected to decay.

Although the creation of cluster states can be calculated analytically to a certain extent, the qubits also decay from their entangled states in a probabilistic manner. Thus it is simpler and more accurate to create a simulation of the system which may be used to represent the qubit states.

We can thus derive an analytical estimation for the T_2 boundary given the entanglement probability.

The main function of the program is computing the average time taken to create a single cluster of length m given a qubit pool of size N and probability of entanglement p. This was done for only 1 dimensional cluster states due to simplicity.

In each time step Δt a global counter t is incremented by one and the average length of the qubit clusters is calculated.

The degradation of qubits occurs regardless of the type, whether it be NV centres, single photons, quatum dots or ions and is caused by the mixing with the enviroment.

The qubit decohereance is modelled using an internal timer one each qubit which is started once said qubit is successfully antangled to anther. This timer values increases the probability of the qubit dechohereing and thus unentanglingand is via an exponentially increasing probability or a simple cut off (kill time) in different versions of the code. The cut off was used initially as this made debugging significantly easier, however, this was replaced by a exponential increase following the the successful implementation due to comparison with real interaction with the environment.

insert picture of the decoherence probabilities vs time here.

4.1 Preliminary work

overview The program is written in the C++ language using a linux based compiler. A linux compiler was chosen as this allow the inclusion of gdb (which has been invaluable in debugging), the use of flags (such as "-0" for optimisation) and it was noticed that the Mac compiler was "doing things behind the scenes", most notably by automatically correcting errors. The final point would have been disastrous when running the simulation on Iceberg (the university supercomputer cluster), as this is linux based and would not have successfully run the program.

The C++ language was chosen as this allows the use of classes, object and vectors. The later has made storing attributes of the clusters and qubits, and dynamically allocating memory during computation significantly easier than if arrays had been implemented. it also allow the system to run faster and more efficiently and the vectors package is pre-installed and optimised. Another reason for using C++ is that it allows the use of the CUDA programming

language as an extension; this will, in theory, allow the program to be preformed in a highly parallel manner by setting different threads to work on an NVidia graphics card.

When the project was officially started in October 2012, some preliminary work had already been undertaken by Joshua Waddington in a summer project funded by the EPSRC. This work gave the outline of the project and what was trying to be achieved by the simulation, which was not initially clear, however the was no code fragment that successfully compiled in October.

Using the program The program is run by navigating to the relevant folder "cluster_simulation_A" through the terminal, were A is the generation of the simulation you wish to use, and typing " Cluster.cpp ClusterCounting.cpp Main.cpp Global.cpp Statistics.cpp " to compile the files into an executable. Then type and enter "./name XY" and the system will run [1]. In pre version 2.2 programs you will notice a long series of matrices and lines of number and entanglement statement being printed to screen. These are simply for debugging processes. The input of the two numbers X and Y are the size of the qubit pool and the entanglement probability respectively. These values are read into the program and passed to the relevant functions. The program initially had these values set in the global header file, however, this meant that the program must be repeatedly edited and recompiled for different values of N and P, which thus excluded the use of bash scripts.

The program contains a great number of files and header files and it may be immediately obvious that these could all be combined into a fewer, larger file. However, for clarity they have been left as separate files as each contains functions which act on different aspects of the program. This has saved confusion during the production of the program and has remained as such considering that the project will likely undertake expansion in the future to accommodate different entanglement regimes and different entanglement gates.

As the program runs in will create a file named "results.csv" to which all the relevant data will be written. This file will be created in the same directory as the executable. If a file named "results.csv" is already present then it shall be amended to include the data from then most recent run - it will not overwite data.

How the program works The system iterates over several loops, defined by the value $NCMS$, wherein the cluster simulation takes place a great number of times for the same value of N and p where each time the number of clusters of length m is calculated. Then the *average* number of clusters of length m is calculated by taking the average from each run (at each timestep), adding them, and dividing by $NCMS$. This is achieved using a pre-written function in the statistics class. The program essentially a monte carlo simulation and produces a very clear trend in the average length of the qubit clusters with time for given values of N and P. Although N and p are the only values which are changed during the running of the program, there are other variables as well. These include the type of algorithm being used to entangle the qubits, the type of gates (thus affecting the values of c_1 and c_2) and the rate of decoherence of the qubits (either from their entangled states or disappearance from the system entirely). The number of time steps undertaken in each case is formed by taking $timesteps = \frac{1}{p}$ as, clearly, higher probability entanglement gates will require few timesteps to complete. A probability of $p = 1$ would obviously complete in only 2 steps. The function $timesteps = \frac{1}{p}$ was chosen as the number of time steps must be an iteger number - thus this resricted the program to working with only values of p given by $\frac{1}{i}$ where i is an integer number. m is given as $m = \frac{1}{p}$

At the end of each loop, the total average number of clusters of length m for each value of N and p at time defined by $timesteps = \frac{1}{p}$ is printed to file.

Batch files and Bash Scripts The batch files (for windows) and bash scripts (for linux) simply run the program several times using nested loops (and/or, in the case of bash scripts, are used to submit the program to run on Iceberg). These are used in versions where N and p must be passed to the file; outer loop increments N, the

[1] 1. The input of X and Y values in not require for all versions of the program and something will have to be changed in main.cpp before recompiling the program. 2. Of course "name" here is simply an arbitrary name and you may call the compiled executable whatever you wish.

qubit pool value, and the inner loop increments p, the entanglement probability. The entanglement probability is incremented in values of 0.1 from 0.1 to 1 while N runs from 10 to a very large value. The main issue the use of scripts was initially designed to solve is that large qubit pools take a very long time to compute (especially when the probability of entanglement is low so the qubit never form a single cluster); thus is can be important to split the loops into several smaller loops so that the processes can be run on different CPU nodes at the same time to decrease the run time requires. This is effect parallelises the system.

Issue; this method presumes that we have a number of entanglement gates equal to half the amount of initial qubits and that qubits can be arbitrarily entangled. Issue; Is there a function which stops the process when all of the qubits are in a single cluster?

The importance of parallelism The program will run much more quickly when it is parrallelised as we will be able to run the program for fixed values of N and p and then form the average cluster length on difference CPU (or GPU threads) at the same time. This is opposed to the programs being run sequentially on a single CPU - this is despite the fact single threads run slower (generally speaking) GPU. This method is especially important when a large qubit pool is used. By letting a node only have to compute a system with a large qubit pool, while allowing lots of systems with smaller qubit pool to be run sequentially on another node, the total time can be reduced.
It was initially hoped that each single system could be run on a different thread on the GPU (thus one thread for every value of $NMCSS$ but this has not yet been implemented.

Building clusters in several dimensions Although the initial goal of he project was show results for 2 and 3 dimensional clusters aswell as a 1 dimensional cluster, that has unfortunately not been possible. The reason is simply that the processes of designing code to sort and order qubit clusters is not a simple a problem as first thought. It would not require to extensive a expansion to allow the system to function in three dimensions - and the program has been written with this expansion in mind by allowing everything to be stored in vectors and all memory to be dynamically allocated. Method for expanding the project to run in three dimensions shall be discussed later.

5 The files

Description of functions in each file

5.1 main.cpp

This file is main file from which all subroutines are called in the main function (main()). The system first declares a value names $NMCS$ (the system may be changed so that this value is read in when the program is executed) which decides how many times the system shall be run (for the same P and N) for average results to be obtained. A higher $NMCS$ is always more desirable as this will give more accurate averages - however, increasing this value beyond necessity is incredibly detrimental as this number is of course directly proportional to the run time (in sequential step up). It is also true that $NMCS$ doe not need to be scaled as the number of qubits in the initial qubit pool increases as may be initially thought. This is simply because the number of qubits produces and averaging affect in itself in the formation of the clusters.

Next, the system calls average length (to show that the initial clustersize is zero - as would be expected as non e of the qubits are entangled), and also create the array using testcluster(). The program then opens the output file and begins to run the simulation.

In the running of the simulation there is first a outer loop over $NMCS$, for the averaging, within which there is an inner loop over the number of timesteps- this is the number of entanglement steps the system will undertake before quiting - (otherwise the system could run infinately for low entanglement probabilities). Inside the inner loop the function then calls print_system(), print(), order_entanglements(), sort_entanglements(), Greed()/Modesty (depending on the method being used), print_system(), print(), average_length()[2]. The purpose of these functions is

[2]Note that print_system() and print() may not be present in some versions as these are quite intensive processes

explained in the classes containing them.

It may be note worthy that the system will continue to run even if all of the qubits are in a single cluster (although is there not a qubit decay in place then nothing will happen). The changes that would likely be made to this, namely stopping the system when a cluster of size m is created or all of the qubits are in a single cluster, is discussed later.

Finally the main function preforms a second loop within the $NCMS$ loop which prints the average length values for each timestep to file, before closing all the loops and closing the file.

5.2 The Qubit class

The qubit class contains the properties of each qubit. These include the original qubit number (the original position of the qubit at the beginning of the computation), the current position it is in and its entanglements with other qubits. Each qubit object holds and internal 1 dimensional binary array of size N used to store information about entanglements with other qubits.

- Public
  - Qbit(). The constructor. This creates the original qbit postion that defines the qubit number.
  - entanglement_value. The holds a value of 0 if the qubit is not entangled to any other qubit and 1 if it entangled.
  - Attach(). This performs a type 1 fusion operation on the qubit and modifies its array.
  - set_position(). This sets the new position of the qubit according to the order it appears (counting from left to right).
  - position_value(). Gives the position of the qubit inside the cluster
  - Detach(int x). Removes a qubit from the cluster.
- Private
  - entanglements[N]. This is the array which stores the entanglements that this qubit has with other qubits.
  - position() This is the current position of the qubit in the cluster
  - time(). This is a counter which stores how much time (how many timesteps have passed since the qubit was entangled). This is used to govern the probability of spontaneous disentanglement.

5.3 qubit.cpp

This file contains the following functions;

- Qbit(). The constructor. This function creates the array which stores which qubits each qubit is entangled to.
- entanglement _value(). This function returns the interger value of how many other qubits the given qubit is entangled to. Note that in a 1 deminsional simulation this value will be either 0,1 or 2.
- Attach(). This creates an entanglement value of one qubit relative to another qubit.
- Detach(). This removes an entanglement value of one qubit relative to another qubit.
- set_position(). This sets the new position of the qubit.
- position_value(). This returns the current position value of the qubit.

5.4 Cluster Class

Then cluster class contains all information which is attributed to the clusters of qubits which exist as separate objects to the qubits themselves. The following attributes are used on the clusters in the sorting and ordering process.

- Public
  - Cluster(). This creates the system.
  - void Greed(). This is the routine which follows the "greed" algorithm.
  - void Modesty(). This is the routine which follows the "modesty" algorithm.
  - void TypeIfusion(). This function entangles the two qubit handed to it with a probability p.
  - void print(). Prints out matrix of qbits showing there entanglements to other qubits.
  - void count_entanglements(). Counts the number of entanglements a single qubit has.
  - void order_entanglements(). Orders the clusters according to thier size
  - void sort_entanglements(). Rearranges the clusters according to thier order.
  - void print_system(). This prints the system to console.
  - double average_length(). This finds the average length of the clusters.
  - void Break(). This removes a qubit from a cluster and resets it.
- Private
  - vector <Qbit> a. This is the vectors structure containing qubit data
  - vector <ClusterCounting> b. This is the vector structure containing cluster data.

5.5 Cluster.cpp

This file contains;

- cluster(). The constructor. This function simply creates the qubit pool and sets the position atributes of each qubit. This is done running through a loop of size N within which position values are declared.

- Count_entanglements(). This function counts the number of entanglements that a given qubit has. This is done by first finding the qubit which is in the position given to the function using a do while loop and then searching to find what qubits this qubit is entangled to. Note that at present the function can count a maximum of two - which is correct as according to the 1 dimensional algorithms that are being used, each qubit can only entangle to a maximum of tow other qubits (the left and the right).
- Order_entanglements(). The purpose of the ordering method is to look through the entanglment matrix and sort the entangled "bricks" into vectors of type clustercounting. from there they can be sorted by the sorting function.
- ClusterCountingSortPredicate(). This returns a true of false value is further used to order the clusters. The function sees whether one function is larger than, small than, or equal in size to another cluster it is handed. If the qubits are both of an equal size then the one which has a the lower qubit number of its letmost is placed to the left.
- sort_entanglements(). This function sorts the clusters into the correct order (from longest to smallest) using the length of the clusters updates all of the qubits with their new positions. It also prints the length of each cluster to screen so the results can be checked.
- print(). This function prints out a matrix showing which qubits are attached to which other qubits. It is purely used to see whether the program is behaving correctly
- Greed(). The is the algorithm which entangles the clusters by entangling them in pairs from smallest to largest.

- TypeIfusion(). This function entangles two qubits handed to it with a probability p. It the entanglement fails then Break() is called.
- Break(). this function removes the qubits handed to it from their various clusters and resets them as single qubits.
- print_system(). This function prints the system to the console. It displays all of the qubits in the pool (by displaying each ones qubit number) and shows the entanglements between them by displaying a "-". The qubits are displayed in the order they are held in the system (in descending order of clusters) and thus this gives and intuitive way to view the state of the system and is very useful in debugging.
- average_length(). This function calculates the average length of the clusters at the time when it is called. It does this by adding the length of all the clusters and dividing by the total number of clusters. Thus this value runs between 1 (when none of the qbits are entangled and thus each qubit is in its own cluster of length 1) and N (when all the qubits are in a single cluster).

5.6 ClusterCounting Class

- Public
  - ClusterCounting(); The constructor.
  - void add_qbit(int add);
  - bool memberqbit_entanglementvalue(int j);
  - int get_length();
  - int length;
  - list<int> memberqbits2;
- Private
  - bool memberqbits[N];

5.7 ClusterCounting.cpp

This file contains the following functions;

- ClusterCounting(). The constructor.
- add_qbit(). Adds a qubit to another cluster (my adding it the the vector list). Presently not used.
- get_length; this function finds the length of a cluster by retrieving the value from the class using memberqits2.size(). The use of vectors clearly simplifies this function.

5.8 Statistics.cpp

This contains a series of statistical functions for calculating potentially important properties such as the variance and averages. Although this has not yet been fully utilised as of yet, it will be used in the final system so results may be displayed with the variance, std error bars, etc. Presently the only function used is the average function.

5.9 Other Files

Matlab We have not yet written any files for the production of graphs. This would make graphing the results much easier as it would mean simply running a Matlab sript instead of using "root" or GNUplot. It would also allow nested loops wherein the first point loops over all of the files and the inner loop prints the file results for each data point specified in the file making plotting result from several files simpler. The outer loop may infact be two loop as it will have to run over all values of both N and P in order to read every file.

6 Misc

7 issues

Problems during build What follows is a discussion of various bugs and anomalies that were present during the building of the project. This should help to highlight some of the intricacies of the program for a clearer understanding.

One of the most confusing errors was that the order of the qubits would spontaneously be reversed when a qubit pool of more than 16 was used. It was thought that this may be a memory allocation error due to the number 16, however it transpired that the issue was instead because the sorting algorithm did not take into account how to order two clusters of the same size. Initially the qubit would only organise for the case that one cluster in larger/smaller than another. This lead to the inclusion of another condition in entangle_predicate.

optimisation The use of the $-o$ flag was used to compile the code, and in some cases the code would fail to compile without it. This is due to missing possible missing library/linker error. The $-o$ flag enforces that only the required libraries and links are used.

runtime or break error If this occurs there is probably a folder missing (if the file has been changes to print to a file in a different directory to the executable). The easiest solution is to check the file system set up against the latest version set-up, or simply copy the entire folder and use this copy to run simulations. The program may also run but not create or amend the results file. This is due to the wrong directory being used in the print command in main.cpp. It should be noted that this will have to be changed - the directory to the results file on Iceberg is different to the results file on a Windows/Mac, even if you are trying to access a results file in the same place as the executable you may need to type the whole address not just ".//".

The implementation of CUDA It was initially hoped CUDA could be included to increase the speed of the program to the parralism. This however proved difficult to implement due to the extensive use of vectors whereas all previous experience in using CUDA included arrays. Thus, as of version 2.2 there is not CUDA extension enabled in the program.

shared memory If CUDA is implemented then shared memory can be used to speed up the simulation be allowing blocks to shared data between threads. This may cause problems if a very large system is being simulated as the shared array may become larger than the blocks memory. This will quickly be realised as the system will produce errors on the first run or crash.

Other Errors and crashes Although the program should work error free some problems may be encountered. Occasionally the system will give errors or crash. This is most probably a compilation error on the part of the user (by either missing a file, referencing a single multiple times, being in the wrong directory, spelling error, etc) or a execution error (check that the execution command has the format "./program XY" if the version in use requires this). If this is not the case then a little further investigation will be needed. Should the program abort with no output (most probably after the second run) then there is possibly a memory error due to the program trying to read from an incorrect address. If this is the case then this report can not help you as this is the main issue that was designed to be fixed by the implementation of vectors. If a version of the program is being used which involved CUDA then there are also some additional factors which may cause a crash; the program may be using more shared memory which is larger than the blocks allocated memory or single threads may be "hanging" (simply remaining static).

The system may also crash after some time and cite a memory error. This was previously encountered and was suspected to be due to the host not keeping tabs on memory being freed at a fast enough pace) and thus presuming all the memory is taken. This was initially solved by using Iceberg but could also be sidestepped by using batch/bash to complete the iterations of p and N; this would mean the program would finish and restart between each instance and thus the host could free memory.

The computer may also suddenly shut down. This is a problem can occur with CUDA and is presumed to be because of the intense workload on the GPU. If the GPU is too busy it will not be able to communicate effectively with the host then the operating system will presume the GPU has crashed and will restart it. This triggered when the internal timer which measures the time since last response between the host and GPU expires. This timer can be increased which may remove this issue, but this has not been attempted.

8 Higher Dimensional Clusters

Although it is possible to create 1 dimensional clusters chains, these are not sufficient for universal quantum computation as this would represent only 1 logical qubit. We will need to alter the composition of the 1 dimensional clusters in order create 2 and 3 dimensional structures required. However, we must be careful in undertaking this as not to destroy the clusters we have taken the time to build.

It is possible to do this using the pauli gates σ_x σ_y σ_z in the X Y and Z respectively. By making an X measurement of a qubit it is possible to form a cherry as shown in figure []. This is can be applied to two linear clusters whose cherries can be entangled via a normal CZ operation. Should the entanglement fail then the two qubits will have to be discarded and reinitialised but the clusters will remain unbroken. A failed gate operation will require a Z-measurement on both cherries to repurify the linear clusters. Thus each failed entanglement of the linear clusters are reduced in size by two qubits and each sucessful entanglement will reduce the length of each by one. This places two restrictions on the system. Firstly it means that the entangling must be undertaken from left to right in a sequential manner, and secondly in means entanglement gates where $c_2 \leq 1$ must be used - this disqualifies fusion gates. The first point is a serious issue and will prove very costly due to the increase in time and it increase the time by an amount proportional to the linear cluster length. The final conclusion of this scheme is a honeycomb structure with both horizontal and vertical entanglements as shown in fig [].

The question of how much more costly this is to make than a 1 dimensional chain is given thusly; as each failed entanglement will shorten each linear chain by 2 qubits, and that we must apply the entanglement procedure $\frac{1}{p}$ times before being successful. The total buffer length is thus increased to;

$$\langle N \rangle_{2D} = \frac{2\beta\Delta N}{p} \tag{8.1}$$

Thus the threshold value T_2 is given as

$$T_2 = \alpha(\tau + \frac{2\beta}{p}\sqrt{\frac{(1+c_2)^2}{p} - c_2(2-c_2)} + \frac{1 + pc_1 + (1-p)c_2}{p} \tag{8.2}$$

$$= \alpha(\tau + \beta\sqrt{\frac{16-4p}{p^3}} + \frac{2}{p}) \text{ for } c_1 = c_2 = 1. \tag{8.3}$$

This is represented in fig [here $\alpha = \beta = 10$ for broker-client (lower) and double heralding (upper)] for a Broker-client entanglement gate and clearly shows a dramatic increase in the T_2 threshold for small values of p. However, these are still within achievable limits. This stategy does not work if $c_2 > 1$ and so a more intricate and costly method must be used.

This method of making 2D cluster states can be expanded to create 3D cluster states simply by creating two 2 Dimensional clusters, or "sheets", and using the same algorithm of creating cherries on either sheet and entangling them as previously. This method allows an arbitrary dimensionality of cluster state to be created.

9 Conclusions

9.1 improvements

Although the program appears to be reasonably successful there are still more functions that could be added.

- A function could be added to give the total run time and the percentage of the run which has been completed. This would be created by calculating how many entanglement procedures will be attempted in each step (approximately $\frac{N}{2}$, multiplying this by the number of timesteps (given by $\frac{1}{p}$), multiply this by the number of different values of p (given as $i =$ and each p is $\frac{1}{i}$ when i is an integer), repeat for all N and times by the time of each entanglement attempt (τ_T). Thus the total time would be given;

$$runtime = \sum_N \tau_T (NMCSS) i \frac{N}{2} \frac{1}{p} \tag{9.1}$$

- CUDA could be implemented to run each instance of $NMCSS$ for the same value of N and p on a different tread on a GPU. This would **vastly** reduce the runtime to

$$runtime = \sum_N \tau_T i \frac{N}{2} \frac{1}{p} \tag{9.2}$$

  Which is a decrease by a factor of 10,000.

10 Further Applications

Clusters are formed by entangling multiple qubits using CZ operations. These may then be used to process information.

11 Preliminary data

Initial functioning verisons of the program allowed average length as a function of p for a qubit pool of size $N = 1 \to 20$ using $NCMS = 10,000$ to be graphed. This showed the interesting result that for probabilities greater than 0.7 the system will form a single cluster. However, for lower probabilities, the average cluster length does not tend to N due to the destructive entanglements.

Insert diagram of prelimary results.

By examining number of m vs p we saw

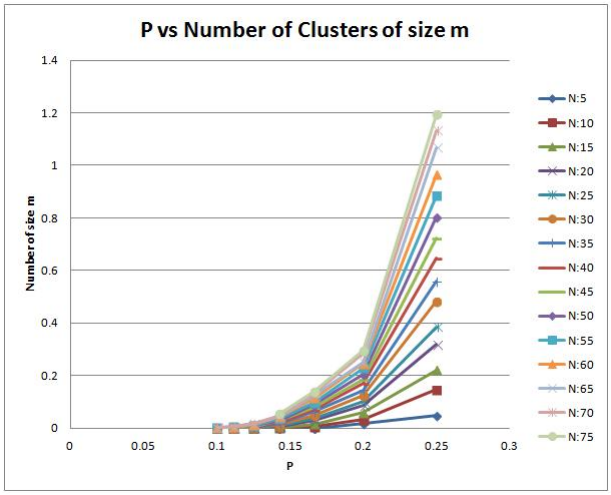

Figure 6: prelim2

We note as expected that the average number of clusters length m increases as a function of N. This agrees with the increasing number of sucessfull entanglements when more are attempted.

Due to the require definition of m and t as integers the constraint on p is degressing number of points as it increases. This is acceptable as the "region of interest" is at the high N low p limit, which, combining with a $log(p)$ x-axis graph gives a passable resolution.

Resetting the program to output only the initial values of p for a given N which achieve number of $m = 1$ in τ we fromed the desired result. As expected the curve is asymptotic about both axis. Using log plot we see ;

12 Conclusions

In this report we have moved through the topic...

13 Acknowledgements

I would like to thank my Project partner Joshua Waddington who has diligently worked with me on this project and my supervisor Dr Pieter Kok.

References