A multi-core CPU implementation of the classical Boson Sampling algorithm

Manan Vaswani

Level M
40cp project

Supervisor: Dr. Raphaël Clifford
Date: 6th May, 2019

Acknowledgement of Sources

Contents

1 Background

The advances made in the field of quantum algorithms in the past 25 years has given rise to questions about whether a number of classical hypotheses still hold true in light of modern quantum research The most notable example is the Extended Church-Turing Thesis which in simple terms says that all computational problems that are efficiently solvable by realistic physical devices are efficiently solvable by a probabilistic Turing machine [1]. However, Shor's algorithm [2] potentially disproves this. Informally, Shor's discovery stated that if a classical computer could accurately simulate a quantum-mechanical experiment in probabilisitic polynomial time, then it must be able to factor integers in polynomial time as well. The problem of factoring integers is known to have no polynomial-time algorithm on a classical computer. It is so widely accepted that factoring integers is a hard problem that a number of cryptographic systems are built on its hardness, including the famous RSA algorithm [3]. Hence, Shor's algorithm has a strong implication: A classical computer that is able to simulate quantum experiments efficiently, it would be able to break widely used cryptosystems such as RSA.

The topic of classically simulating quantum systems is a central idea in the field of quantum supremacy. Quantum supremacy is the phenomena that there exist quantum experiments that cannot be accurately and efficiently simulated using classical systems. Numerous attempts have been made to demonstrate quantum supremacy but the current physical limitations of quantum systems has made that an extremely challenging problem. Additionally, what makes it so hard to demonstrate is that it is not defined in terms of the ability of a quantum system to solve a particular problem, but rather it requires showing that classical computers cannot solve a problem. In a way, it could be likened to classifying computational problems into complexity classes [4]. In the same way that a number of complexity theory related theorems and conjectures rely on a number of theoretic assumptions, it is necessary for quantum supremacy to rely on similar such assumptions as well, as it would be nearly impossible to unconditionally prove such statements [5].

Linking back to Shor's algorithm and its potential to demonstrate quantum supremacy and simultaneously disprove the Extended-Church Turing thesis, there are two significant drawbacks which prevent it from doing so.

Firstly, the problem of factoring integers, while widely believed to be an $\mathsf{NP}-$hard problem, its hardness has not been formally proved. The assumption is not strong enough to verify that Shor's algorithm does achieve quantum supremacy.

The second, and more obvious drawback with Shor's algorithm is that a large-scale physical implementation of a quantum computer to run it is beyond the current reach of technology. While there have been several experiments that successfully factor small numbers like 15 [6] and 21 [7] with high accuracy, we are not at risk of RSA breaking just yet!

This has motivated the search for other such quantum experiments that can be run physically, but not simulated classically. However, and important requirement is that it should be possible to model these experiments as computational problems in order to compare the hardness.

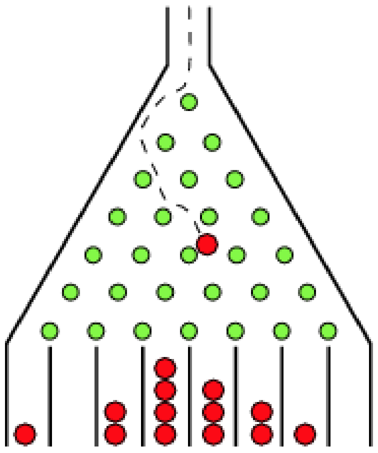


Figure 1: A Galton board, used to demonstrate sampling from a binomial distribution [12]

There are a handful of problems that satisfy this criteria, such as the problem of sampling the output distribution of random quantum circuits, which has been one of the most popular problems in the field of quantum supremacy in the past few years due to a lot of interest from big companies like Google and IBM. The best known classical algorithm for simulating the task of sampling bitstrings from the output of random quantum circuits is exponential in the number of qubits in the bitstring, leading one group of researchers in 2016 to estimate that a physical experiment with ≈ 50 qubits could be enough to demonstrate quantum supremacy [8]. However, more recently a classical simulation of the problem with 56 qubits was carried out successfully [9]. This pushed the imminence of quantum supremacy for the circuit sampling problem even further away.

The other famous problem in the field of quantum supremacy, of a similar flavour, is the Boson Sampling problem, which is the main focus of this paper. While the problem reduces down to a sampling problem like circuit sampling, as the name suggests, it actually belongs to a paradigm of Quantum Computation different to the ones discussed before, called Linear Optical Quantum Computation (or LOQC). The protocol by Knill, Laflamme and Milburn in 2001 [10] demonstrated that a scalable quantum computer with linear optical elements could be robustly implemented. In LOQC, a photon is used to represent a single qubit.

Aaronson and Arkhipov in their paper on the classical complexity of linear optics showed that it was possible to build a quantum computation model within the LOQC paradigm which could not be efficiently simulated by classical computers [11]. They define a model for 'Boson Sampling' in which n identical photons are passed through a linear optical network and then measured to determine their location. This linear optical network consists of a collection of simple optical elements called beam splitters and phase shifters.

The model could be thought of as being a parallel to a Galtons Board, shown in figure

1, in which n balls are dropped into a vertical lattice of wooden pegs, each of which randomly scatters an incoming ball to one of two other pegs with equal probability. The input to this system is the exact arrangement of the pegs, while the output is the number of balls that have landed at each location on the bottom. The output could alternatively be thought of as a sample from the distribution of all possible output arrangements (which is the binomial distribution in this case).

In the Aaronson and Arkhipovs Boson Sampling model, the balls are replaced with identical photons, and the pegs arranged in a lattice are replaced with an arbitrary arrangement of optical elements. Also, the photons can be dropped from different starting locations as opposed to just a single position. The Boson-Sampling problem involves sampling the output distribution using photon-number discriminating detectors. They argued that if there is a polynomial time classical algorithm that exactly samples from the Boson Sampling distribution, the complexity class $\mathsf{P}^{\#\mathsf{P}} = \mathsf{BPP}^{\mathsf{NP}}$ which would collapse the polynomial hierarchy to the third level by Toda's theorem [13].

In order to actually compare the physical model with a classical computational model, the paper reduces the Boson Sampling problem to a problem in purely mathematical terms. Photons are a type of boson, which is one of the two main types of particles in the world. The link between bosons and permanents of matrices has been known since 1953, from the work by Caianiello [14] which showed that the amplitudes of n-boson process can be written as the permanents of $n \times n$ matrices. Hence it was possible to represent the Boson Sampling distribution in terms of the permanents of matrices, and this was shown and proved in the paper.

Carrying out a physical experiment for Boson Sampling is not a trivial task, and increasing the number of photons or the number of output modes to large enough values is extremely difficult. The current experimental record is with 5 photons and 9 output modes. However, this does show the possibility of potentially having larger-scale implementations in the future.

The Aaronson and Arkhipov paper provoked much interest in the classical computation algorithm for Boson Sampling. Much of the work is focussed around fast implementations for the calculation of permanents of large matrices, with recent works showing efficient results for matrices as large as 54×54 [15, 16]. There have been a few other approaches to simulating the Boson sampling problem such as classically modelling the Linear optical network itself, and sampling from the output as opposed to computing the purely mathematical equivalent of the problem [17]. There have also been attempts at using Markov Chain Monte Carlo (MCMC) methods to sample from the Boson Sampling distribution by trying to identify it as the equilibrium distribution of a Markov chain, following the method by Hastings [18]. However, estimating how accurately these methods approximate to the true Boson Sampling experiment has been difficult. Nonetheless, the MCMC method proposed by Neville et. al. [19] gave strong numerical evidence that classical Boson Sampling may be feasible for large input sizes.

In terms of actual implementations of the Boson Sampling algorithm, there have been few, with the most significant benchmark being the test of the simulation run on the Tianhe-2 supercomputer [20] which managed to simulate Boson Sampling for 50 photons with a runtime of approximately 100 minutes.

However, in the groundbreaking paper by Clifford and Clifford [21], a new significantly faster algorithm was proposed which showed promise of carrying out classical Boson Sampling in an even shorter time. The algorithm proposed in this paper is central to our work. We give a highly optimised implementation of this algorithm with the aim of setting a new benchmark for the classical Boson Sampling problem, and in the process pushing away the imminence of quantum supremacy in the near future.

2 Technologies used

2.1 Blue Crystal 4

2.2 OpenMP

2.3 Intel Vtune

3 Preliminaries

3.1 Binary Gray Code

3.2 Permanent of a matrix

Computing the permanent of large matrices is one of the key parts of the Boson Sampling problem, as will be discussed in detail while explaining the problem in the subsequent sections of the paper.

3.2.1 Definition

Definition 3.1. *[22] The permanent of an $n \times n$ matrix $A = (a_{ij})$ is defined as*

$$\operatorname{Per} A = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{1}$$

where the sum is over all elements of the symmetric group S_n i.e. over all permutations of the numbers in $[n] = \{1, 2, ..., n\}$

Example 3.2. For a 2×2 matrix, the permanent is calculated as follows

$$\operatorname{Per} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad + bc \tag{2}$$

Example 3.3. For a 3×3 matrix, the permanent is calculated as follows

$$\operatorname{Per} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aei + bfg + cdh + ceg + bdi + afh \tag{3}$$

One can observe that the definition of the permanent is similar to the more commonly used determinant function, differing in the fact that the permanent definition lacks the alternating signs. An important property to note about the permanent function is that it is invariant to transposition i.e. $\operatorname{Per} A = \operatorname{Per} A^T$ [23].

3.2.2 Computing the permanent

Valiant showed that the problem of computing the permanent of a matrix is in the class #P-complete which implies that it is unlikely to have a polynomial time algorithm which implies that it is unlikely to have a polynomial-time algorithm [24]. The naive algorithm obtained by directly translating the formula into an algorithm would run in $\mathcal{O}(n!n)$ time.

A significant improvement on the naive approach, Rysers algorithm uses a variant of the inclusion-exclusion principle and can be evaluated in $\mathcal{O}(n^2 2^n)$ time [23]. Nijenhuis and Wilf sped this up to $\mathcal{O}(n2^n)$ time by iterating over the sum in Gray Code order [25].

Another formula that is as fast as Rysers was independently derived by Balasubramanian[26], Bax[27], Franklin and Bax[28], and Glynn[29], all using different methods. We shall henceforth refer to this as Glynn's formula and it is described as follows.

Let $M = (m_{ij})$ be an $n \times n$ matrix with $m_{ij} \in \mathbb{C}$, then

$$\operatorname{Per} M = \frac{1}{2^{n-1}} \sum_{\delta} \left(\prod_{k=1}^{n} \delta_k \right) \prod_{j=1}^{n} \sum_{i=1}^{n} \delta_i m_{ij} \tag{4}$$

where $\delta \in \{-1, 1\}^n$ with $\delta_1 = 1$. Hence there are 2^{n-1} such values for δ.

Implementing the formula as is would require $\mathcal{O}(2^n n^2)$ time. However, iterating over the δ arrays in Gray code order reduces it to $\mathcal{O}(n2^n)$ time.

In order to exploit this trick, let v_j be the symbol used to denote the innermost sum i.e. $v_j = \sum_{i=1}^{n} \delta_i m_{ij}$. If δ is iterated over in Gray code order, then one can notice that the terms $\{v_j, j \in [n]\}$ can actually be computed in $\mathcal{O}(n)$ time for a given value of δ rather than $\mathcal{O}(n^2)$ which is how long it would take if done naively. This is because successive elements of δ differ in only one position, so each successive v_j differs only by the addition or subtraction of some m_{ij} where i is the position at which δ was last changed. Hence, the product of v_j terms can now be calculated in $\mathcal{O}(n)$ time. Note that $\delta \in \{-1, 1\}^n$ but elements of the Gray code of size n are in $\{0, 1\}^n$. To resolve this, map each 0 in the Gray code to a 1 of the δ array, and each 1 in the Gray Code to a -1 of the δ array.

4 Describing the paper and specifying the problem

4.1 Explaining the problem

A summary of the problem in purely mathematical terms is as follows.

Let m and n be positive integers. Consider all possible multisets[1] of size n with elements in $[m]$, where $[m] = \{1, ..., m\}$. Let $z = [z_1, z_2, ..., z_n]$ be an array representation of such a multiset, with its elements in non-decreasing order. In other words, z is an array of n integers taken from $[m]$ (with repetition) and arranged in non-decreasing order. Define $\Phi_{m,n}$ to be the set of all distinct values that z can take. Define $\mu(z) =$

[1]A multiset is a special kind of set in which elements can be repeated

$\prod_{j=1}^{m} s_j!$ where s_j is the multiplicity of j in the array z i.e. the number of times it appears in z.

$A = (a_{ij})$ is a complex-valued $m \times n$ matrix constructed by taking the first n columns of a given $m \times m$ Haar random unitary matrix. For each z, build an $n \times n$ matrix A_z where the k^{th} row of A_z is the z_k^{th} row in A, for $k = 1, ..., n$. Finally, define a probability mass function over $\Phi_{m,n}$ as

$$q(z) = \frac{1}{\mu(z)} \left|\operatorname{Per} A_z\right|^2 = \frac{1}{\mu(z)} \left| \sum_{\sigma} \prod_{k=1}^{n} a_{z_k \sigma_k} \right|^2, \quad z \in \Phi_{m,n} \tag{5}$$

where $\operatorname{Per} A_z$ is the permanent of A_z and $\pi[n]$ is the set of all permutations of $[n]$.

The computing task is to simulate random samples from the above pmf $q(z)$.

4.1.1 Calculating the size of the sample space

The size of the sample space $\Phi_{m,n}$ can be calculated using the 'stars and bars' technique from combinatorics [30]. In our problem, we have n 'stars' representing the elements of the array z, and m 'buckets' representing all the values in $[m]$. Recall that z is a sorted array representation of a multiset, so multiple 'stars' can be in one 'bucket'. Since there are m 'buckets', we need $m - 1$ 'bars' to divide the 'stars' into 'buckets'. Therefore, from a total of $m - 1 + n$ objects, we need to pick n of these to be the 'stars'. There are $\binom{m+n-1}{n}$ ways to do this. Hence, there are $\binom{m+n-1}{n}$ possible values of z.

5 The Boson Sampling Algorithm

5.1 The naive approach

Translating the formula above (5) directly to an algorithm would require $\mathcal{O}(\binom{m+n-1}{n} n2^n)$ time to evaluate. The $\binom{m+n-1}{n}$ term comes from the size of the sample space of the pmf, and calculating the value of the permanent using the fastest known methods takes $\mathcal{O}(n2^n)$ time (Section 3.2.2). With $m = \mathcal{O}(n^2)$ as suggested in Section????, the total running time is $\mathcal{O}(\binom{n^2+n-1}{n} n2^n) =$???. Hence, even for relatively small values of n ((give some actual numbers), computing the Boson Sampling problem would be intractable for even very powerful supercomputers.

In the literature [21], two new algorithms are proposed for exact Boson Sampling. These are referred to as Algorithm A and Algorithm B, and both provide a significant speed up on the naive algorithm. The following subsections summarise the approach taken to obtain them.

5.2 Algorithm A

The approach to Algorithm A starts by expanding the sample space to a much larger size, which seems counterintuitive at first, but it allows us to express the pmf (Equation 5) in a form that is much easier to compute. The sample space is expanded to the space of all arrays $\mathbf{r} = (r_1, r_2, ..., r_n)$ where each element r_k is in $[m]$, which implies that we are considering a distribution on the product space $[m]^n$. It is stated and proved that sampling from $q(\mathbf{z})$ is equivalent to sampling from the pmf

$$p(\mathbf{r}) = \frac{1}{n!} \left|\text{Per } A_\mathbf{r}\right|^2 = \frac{1}{n!} \left| \sum_\sigma \prod_{i=1}^{n} a_{r_i \sigma_i} \right|^2, \quad \mathbf{r} \in [m]^n \tag{6}$$

where as before, σ is the set of all permutations of $[n]$.

This method requires $p(\mathbf{r}) = p(r_1, ..., r_n)$ to be rewritten as a product of conditional probabilities using the chain rule i.e.

$$p(\mathbf{r}) = p(r_1)p(r_2|r_1)p(r_3|r_1, r_2)...p(r_n|r_1, r_2, ..., r_{n-1}) \tag{7}$$

We first sample r_1 from $p(r_1), r_1 \in [m]$. Then for $k = 2, ..n$, we sample r_k from the conditional pmf $p(r_k|r_1, r_2, ..., r_{k-1})$ with $r_1, ..., r_{k-1}$ fixed. After sampling all values of r_k, sort $(r_1, r_2, ..., r_n)$ in non-decreasing order, and that results in the array representation of a multiset sampled from the Boson Sampling distribution $q(\mathbf{z})$. In order to obtain the conditional probabilities, a formula for the joint pmf of the leading subsequences of $(r_1, ..., r_k)$ is given in Lemma 1 of the paper, and proved using arithmetic techniques and facts about probability measures. The formula in Lemma 1 is as follows:

$$p(r_1, ..., r_k) = \frac{(n-k)!}{n!} \sum_{c \in \mathcal{C}_k} \left|\text{Per } A^c_{r_1, ..., r_k}\right|^2, \quad k = 1, ..., n \tag{8}$$

where $\mathcal{C}_k$ is the set of $k-$combinations taken without replacement from $[n]$ and $A^c_{r_1,...,r_k}$ is the matrix formed by taking only columns $c \in \mathcal{C}_k$ of the rows $(r_1, ..., r_k)$ of A.

Notice in equation 7, we need to sample r_k from the conditional probability distribution $p(r_k|r_1, ..., r_{k-1})$. This can be rewritten as $p(r_1, ..., r_k)/p(r_1...r_{k-1})$. Since r_k does not appear in the denominator, in order to sample r_k from the conditional pmf, we can equivalently sample from the pmf proportional to the numerator, as $(r_1, ...r_{k-1})$ are fixed, known values at this stage in the algorithm. Therefore, the formula is Lemma 1 is used to calculate the conditional pmfs at each stage, giving way to the following algorithm.

The correctness of the algorithm is clear as it is simply an evaluation of the required pmf using the chain rule of probability. The literature gives a mathematical proof to derive the runtime, however here we shall give an informal explanation to obtain the runtime using the algorithm above. The runtime is dominated by the **for** loop in lines 2 to 6, as the sorting step in line 7 takes only $\mathcal{O}(n \log n)$ using the fastest methods for sorting (insert reference). In each iteration of the **for** loop, we first construct the conditional pmf on the sample space $[m]$. This is represented by a weighted array $\mathbf{w}$. This involves calculating the permanents of $|\mathcal{C}_k|$ $k \times k$ matrices for each $i \in [m]$, where k is the loop index variable. Sampling from this distribution (line 4) takes $\mathcal{O}(m)$ time [31], but this too is dominated by the calculations in line 3, so we can ignore it. Therefore, the total time taken in the k^{th} iteration is $\mathcal{O}(m\binom{n}{k}k2^k)$, as $|\mathcal{C}_k| = \binom{n}{k}$ and calculating

Algorithm 1 Boson Sampler: Single sample $\mathbf{z}$ from $q(\mathbf{z})$ in $\mathcal{O}(mn3^n)$ time

```
Require: $m, n \in \mathbb{Z}_+$; $A$ formed by first $n$ columns of $m \times m$ Haar random unitary matrix
1: $\mathbf{r} \leftarrow \emptyset$
2: for $k \leftarrow 1 \to n$ do
3:     $w_i \leftarrow \sum_{c \in \mathcal{C}_k} \left| \mathrm{Per}\, A^c_{(\mathbf{r},i)} \right|^2, i \in [m]$
4:     $x \leftarrow \mathrm{Sample}(w)$
5:     $\mathbf{r} \leftarrow (\mathbf{r}, x)$
6: $\mathbf{z} \leftarrow \mathrm{IncSort}(\mathbf{r})$
7: return $\mathbf{z}$
```

the permanent of a $k \times k$ matrix takes $\mathcal{O}(k2^k)$ using the fastest methods. So for n iterations,

$$\sum_{k=1}^{n} mk2^k \binom{n}{k} = m\frac{2}{3}n3^n = \mathcal{O}(mn3^n) \tag{9}$$

Therefore the total time taken is $\mathcal{O}(mn3^n)$. In terms of space complexity, we only need to store one permanent calculation at a time, and the weight array for each pmf is of size m. These can also be stored only one at a time, so $\mathcal{O}(m)$ additional space is required.

5.3 Algorithm B

The main theorem of the paper(Theorem 1) states that the time complexity of the Boson Sampling problem is $\mathcal{O}(n2^n + \mathrm{poly}(m, n))$ where $\mathrm{poly}(m, n) = \mathcal{O}(mn^2)$, with $\mathcal{O}(m)$ additional space required. This is achieved using the second algorithm proposed in the paper i.e. Algorithm B.

Algorithm B makes use of the Laplace Expansion [22] of permanents to obtain a significant speed up. The Laplace expansion allows us to calculate the permanent of a matrix using its permanent minors (the permanent of a submatrix with a column and a row removed). For any $k \times k$ matrix $B = (b_{i,j})$,

$$\mathrm{Per}\, B = \sum_{l=1}^{k} b_{k,l}\, \mathrm{Per}\, B^{\diamond}_{k,l}, \tag{10}$$

where $B^{\diamond}_{k,l}$ is the submatrix of B with row k and column l removed. Hence the permanent of B can be calculated in $\mathcal{O}(k)$ steps provided the permanent minors i.e. the values $\{\mathrm{Per}\, B^{\diamond}_{k,l}\}$ are known.

Computing $\{\mathrm{Per}\, B^{\diamond}_{k,l}\}$ by calculating each of the permanents independently would collectively take $\mathcal{O}(k^2 2^k)$ time as there are k permanents to compute and to compute the value of each one takes $\mathcal{O}(k2^k)$ time. Lemma 2 of the paper states that the collection $\{\mathrm{Per}\, B^{\diamond}_{k,l}\}$ can be collectively evaluated in $\mathcal{O}(k2^k)$ time and $\mathcal{O}(k)$ extra space. using

Glynn's formula to evaluate a permanent minor for a given value of l, we have

$$\operatorname{Per} B^{\diamond}_{k,l} = \frac{1}{2^{k-2}} \sum_{\delta} \left(\prod_{i=1}^{k-1} \delta_i \right) \prod_{j \in [k] \setminus l} v_j(\delta), \quad l \in [k], \tag{11}$$

where $\delta \in \{-1, 1\}^{k-1}$ with $\delta_1 = 1$ and $v_j(\delta) = \sum_{i=1}^{k-1} \delta_i b_{ij}$. The usual trick of iterating over the values of δ in Gray code order is required to compute $\operatorname{Per} B^{\diamond}_{k,l}$ in $\mathcal{O}(k2^k)$ time for each value of l, but this would still have to be repeated k times to cover all $l \in [k]$. The second and more novel trick used to achieve an additional speed up is used while calculating the products $\prod_{j \in [k] \setminus l} v_j(\delta)$. Observe that for each l, $\prod_{j \in [k] \setminus l} v_j(\delta) = \prod_{j \in [k]} v_j(\delta)/v_l(\delta)$. However, we cannot compute this method to compute the partial product as it would not work for cases when $v_l(\delta) = 0$. Instead, we first precompute two arrays $\mathbf{f}$ and $\mathbf{b}$ containing the forwards and backwards cumulative products of $\mathbf{v}(\delta) = (v_j(\delta), j \in [k])$ respectively as follows,

$$f_i = \prod_{j=1}^{i} v_j(\delta), \quad f_0 = 1, \quad b_i = \prod_{j=i}^{k} v_j(\delta), \quad b_{k+1} = 1 \quad i \in [k]. \tag{12}$$

Then each partial product $\prod_{j \in [k] \setminus l} v_j(\delta)$ can be expressed as a product of two terms from $\mathbf{f}$ and $\mathbf{b}$, $f_{l-1} b_{l+1}$ for any $l \in [k]$. Since it takes $\mathcal{O}(k)$ time to compute the forward and backward cumulative product arrays and $\mathcal{O}(1)$ time to obtain each partial product $\prod_{j \in [k] \setminus l} v_j(\delta)$, for each value of δ, the total time taken to simultaneously calculate the values of $\{\operatorname{Per} B^{\diamond}_{k,l}\}$ is $\mathcal{O}(k2^k)$. Additional space required to store the partial products is $\mathcal{O}(k)$.

Coming back to formulating the second Boson Sampling algorithm, the sample space is now expanded even further with an auxiliary array $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ where $\alpha \in \pi(n)$, which is the set of permutation of $[n]$. The approach is similar to that of Algorithm A in that the goal is to create a succession of pmfs for leading subsequences of $\mathbf{r}$ and then progressively sampling r_i from conditional pmfs on $(r_1, ..., r_{i-1})$. The conditioning variable α helps us reformulate the pmf in such a way that we can use the permanent minors that we are able to calculate efficiently, while still ensuring that it is equivalent to sampling from the pmf $p(\mathbf{r})$ in equation 6. Define

$$\phi(r_1, ...r_k | \alpha) = \frac{1}{k!} \left| \operatorname{Per} A^{[n] \setminus \{\alpha_{k+1}, ..., \alpha_n\}}_{r_1, ...r_k} \right|^2, \quad k = 1, ..., n-1. \tag{13}$$

In Lemma 3 of the paper, it is shown that $p(\mathbf{r}) = \mathbb{E}_{\alpha}\{\phi(\mathbf{r}|\alpha)\}$, which is the expectation taken over α, uniformly distributed over $\pi(n)$ for fixed $\mathbf{r}$. The chain rule of expectation is used to prove the equality, and this full proof is provided in the literature. As in Algorithm A, a chain of conditional pmfs as follows is used,

$$\phi(\mathbf{r}|\alpha) = \phi(r_1|\alpha)\phi(r_2|r_1, \alpha)\phi(r_3|r_1, r_2, \alpha)...\phi(r_n|r_1, ...r_{n-1}, \alpha), \tag{14}$$

where $\phi(r_k|r_1, ...r_{k-1}, \alpha) = \phi(r_1, r_2, ...r_k|\alpha) / \sum_{r_k} \phi(r_1, ...r_k|\alpha)$, using conditional probabilities and the law of total probability. The algorithm starts off by sampling r_1 from the pmf $\phi(r_1|\alpha)$. Then for stages $k = 2, ..., n$, r_k is sampled from $\phi(r_k|r_1, ...r_{k-1}, \alpha)$ with $(r_1, ..., r_{k-1})$ fixed. Since the terms in the denominator are fixed, known values, sampling from $\phi(r_k|r_1, ...r_{k-1}$ is equivalent to sampling from the numerator, which can be evaluated by taking advantage of the Laplace Expansion of permanents. At stage n, the array $(r_1, ..., r_n)$ will have been sampled from $p(\mathbf{r})$. Sorting $(r_1, ..., r_n)$ in non-decreasing order gives $\mathbf{z}$, the array representation of the multiset sampled from the Boson Sampling distribution $q(\mathbf{z})$.

Algorithm 2 Boson Sampler: Single sample $\mathbf{z}$ from $q(\mathbf{z})$ in $\mathcal{O}(n2^n + \text{poly}(m, n))$ time

```
Require: $m, n \in \mathbb{Z}_+$; $A$ formed by first $n$ columns of $m \times m$ Haar random unitary matrix
 1: $\mathbf{r} \leftarrow \emptyset$                                        ▷ Empty array
 2: $A \leftarrow \text{Permute}(A)$                          ▷ Randomly permute columns of $A$
 3: $w_i \leftarrow |a_{i,1}|^2, i \in [m]$                   ▷ Make indexed weighted array $w$
 4: $x \leftarrow \text{Sample}(w)$                              ▷ Sample index $x$ from $w$
 5: $\mathbf{r} \leftarrow (\mathbf{r}, x)$                                  ▷ Append $x$ to $\mathbf{r}$
 6: for $k \leftarrow 2$ to $n$ do
 7:     $B_k^\diamond \leftarrow A_{\mathbf{r}}^{[k]}$
 8:     Compute $\{\text{Per}\, B_{k,l}^\diamond, l \in [k]\}$                   ▷ From Lemma 2
 9:     $w_i \leftarrow \left|\sum_{l=1}^{k} \text{Per}\, B_{k,l}^\diamond\right|^2, i \in [m]$       ▷ Using Laplace Expansion
10:     $x \leftarrow \text{Sample}(w)$
11:     $\mathbf{r} \leftarrow (\mathbf{r}, x)$
12: $\mathbf{z} \leftarrow \text{IncSort}(\mathbf{r})$                   ▷ Sort $\mathbf{r}$ in non-decreasing order
13: return $\mathbf{z}$
```

Correctness holds as Algorithm B samples from successive conditional pmfs, which is equivalent to sampling from $\phi(r|\alpha)$ due to the chain rule of probability (Equation 14), for a given α. Since α is uniformly distributed over the sample space $\pi(n)$, the algorithm equivalently samples from $\mathbb{E}_\alpha\{\phi(\mathbf{r}|\alpha)\}$ which is shown to be equal to $p(r)$ in Lemma 3.

Once again, we will analyse the time complexity of the algorithm by going through the actual steps of the algorithm rather than the more mathematical explanation given in [21]. The time complexity of the algorithm is dominated by the **for** loop as expected. Permuting the columns of A (Line 2) can be carried out in $\mathcal{O}(n)$ time using the Knuth shuffling algorithm [32]. Lines 3 and 4 take $\mathcal{O}(m)$ time, in order to generate the weight array representing the pmf and sample from it. In the **for** loop from lines 6 to 11, computing the permanent minors (Line 8) takes $\mathcal{O}(k2^k)$ time in the k^{th} iteration, as discussed previously. Subsequently, the weight array in Line 9 is computed in $\mathcal{O}(mk)$ time as it is an array of size m, with each element taking $\mathcal{O}(k)$ time to compute as it is a sum of k known terms. Sampling from this pmf takes $\mathcal{O}(m)$ time. The final sorting step outside the loop, in Line 12 takes $\mathcal{O}(n \log n)$ time, so it is dominated by the loop. Each iteration of the loop takes $\mathcal{O}(k2^k) + \mathcal{O}(mk)$ time in total. So for all iterations, this takes

$$\sum_{k=2}^{n} \mathcal{O}(k2^k) + \mathcal{O}(mk) = \mathcal{O}(n2^n) + \mathcal{O}(mn^2) \tag{15}$$

time.

The space is dominated by the weight array in Line 9, which is of size $\mathcal{O}(m)$.

Hence, the Boson Sampling Problem can be run in $\mathcal{O}(n2^n + \text{poly}(m,n))$ time using Algorithm B.

6 Implementation

In this section, we will discuss the actual implementation of Algorithm B described above. The paper by Clifford and Clifford which demonstrated the two new algorithms is still relatively new, which explains why there have not been any open-source large scale benchmark tests for actual implementations of the algorithm.

6.1 Original R Implementation

At the time of writing this paper, the only open-source implementation is at [33], by the authors of the paper. The implementation is in written as a package written in R and is freely available to download and use from CRAN. This package, named 'Boson Sampling' contains an implementation of Algorithm B, along with the required functions that it depends on. The relevant functions are described below.

`bosonSampler`

The package provides a function '`bosonSampler(A, sampleSize, perm=FALSE)`', which implements the Boson Sampling Algorithm B from [21]. As the function definition suggests, it takes three arguments

- `A`: The first n columns of an $m \times m$ random unitary matrix.
- `sampleSize`: The number of independent sample values to be taken from the distribution, where each value is a vector of size n.
- `perm`: Takes the value `TRUE` if the permanents and pmfs associated to each sample are required to be returned. By default, `perm=FALSE` unless specified otherwise.

Note that we do not need to explicitly provide the values m and n as the algorithm suggests since those values are obtained by simply checking the dimensions of the input $m \times n$ matrix using the `ncol(A)` and `nrow(A)` functions from R. This implementation allows us to obtain multiple samples from the Boson Sampling distribution using the same input matrix A by repeatedly looping over the algorithm the desired number of times. One notable difference is that the input matrix A is transposed right at the start, with the reason being that R stores matrices in column order so transposing the matrix makes operations easier. When accessing elements of A, the indices have been adjusted accordingly. Semantically, the function implements the algorithm exactly as it is, which ensures its' correctness (provided the inputs received and additional functions used are also correct).

`cxPermMinors`

The package provides a set of functions for computing permanents of matrices: `cxPerm(A)`, `rePerm(B)`, `cxPermMinors(C)`. Algorithm B requires the values $\{\operatorname{Per} B^{\diamond}_{k,l}, l \in [k]\}$ to be computed in each iteration of the loop, and in order to do this, the `bosonSampler` function makes a call to the `cxPermMinors` function. This function takes an input matrix A of size $k \times k-1$. Note that this is actually the transpose of the $B^{\diamond}_{k}$ in Line 7 of the algorithm, since the original matrix in `bosonSampler` was transposed. `cxPermMinors` is written in C++ and is integrated into the R package using 'RCpp', which is another package in R [34]. The package facilitates mapping R datatypes to C++ equivalents and vice-versa. The function also extensively uses the Armadillo package from C++[35], which is a linear algebra library providing efficient functions and classes for vectors, matrices, and operations on them. This function calculates the permanent minors using Glynn's algorithm [29] using the technique of iterating over Gray codes, by Nijenhuis and Wilf[25] to iterate over Gray codes , and the trick of calculating forward and backward cumulative products described in Section 5.3.

`randomUnitary`

The function `randomUnitary(size)` takes in an integer `size` as an input, and returns a size $\times$ size complex-valued random unitary matrix.

Example

The function was run in the R console to produce 5 samples with $n = 10$ and $m = 100$.

```
> library(BosonSampling)        # load the BosonSampling package
> set.seed(7)         # set the random seed in R
> n <- 10
> m <- 100
> A <- randomUnitary(m)[,1:n]        #construct random unitary matrix
> valueList <- bosonSampler(A, sampleSize = 5)$values
> #run Boson Sampling algorithm
> apply(valueList, 2, sort) #sort each sample
      [,1] [,2] [,3] [,4] [,5]
 [1,]    6    2    6   15    5
 [2,]    6    5    8   15   15
 [3,]   24    5    8   24   23
 [4,]   36   15   30   45   24
 [5,]   54   16   32   60   44
 [6,]   61   52   35   65   50
 [7,]   77   68   41   77   62
 [8,]   78   69   46   80   77
 [9,]   78   81   50   80   89
[10,]   79   99   52   96   95
```

6.2 Implementation in C++

While choosing a programming language to implement a more efficient version of the Boson Sampling algorithm, performance was the key deciding factor. The language chosen was C++ due to its reputation for speed in mathematical problems, ease of use especially for multi-threaded programs, and the availability of highly efficient linear algebra libraries such as Armadillo. As with the *R* implementation, three core functions are required for the algorithm: a Boson Sampler (which runs the actual algorithm), a random unitary matrix generator, and a function to calculate the permanent minors of a matrix.

Armadillo Library: In the implementation, we make extensive use of the C++ Armadillo library[35]. Armadillo is a high quality linear algebra library that provides us with efficient implementations of matrix and vector operations. It greatly simplifies operations such as adding vectors, calculating cumulative products, and accessing matrix columns to name a few. Armadillo supports complex numbers as well, which was a necessary requirement for Boson Sampling. In addition, it supports the use of OpenMP for parallelisation, and automatically uses it in some cases to speed up computationally expensive operations.

6.2.1 Generating a random unitary matrix

The first step carried out was to create a function `randomUnitary(int m)` to generate a complex-valued random unitary matrix A which will be passed as an argument to the Boson Sampler. In order to generate a Haar random unitary matrix, the following algorithm from [36] was used. QR Decomposition decomposes a given matrix A into an

Algorithm 3 Random Unitary: Generate an $m \times m$ complex-valued random unitary matrix

```
Require: $m \in \mathbb{Z}_+$
1: $A \leftarrow$ createRandomMatrix($m$)
2: $Q, R \leftarrow$ QRDecomposition($A$)
3: $R_{\text{diag}} \leftarrow$ Sign (Real (Diagonal ($R$)))
4: $U \leftarrow Q * R_{\text{diag}}$
5: return $U$
```

orthogonal matrix Q and a right triangular matrix R such that $Q * R = A$. Armadillo provides the all the required supplementary functions used in the implementation, which made the C++ implementation straightforward. Since this function is run only once as a preprocessing step before the Boson Sampling, extra steps to optimise or parallelise it were not taken. Firstly, because the primary aim of the project is to optimise the implementation of the Boson Sampling algorithm itself, which receives the random unitary matrix as an input, and secondly, it is already a highly efficient implementation since all the functions used are from Armadillo so have already been optimised, as mentioned before. The object returned by `randomUnitary(int m)` is an $m \times m$ random unitary armadillo matrix.

6.2.2 Computing the Permanent Minors

Line 8 of the algorithm requires the permanent minors of a given $k-1 \times k$ matrix to be computed. We use a separate function `cxPermMinors(arma::cx_mat C)` which does exactly this for us. Once again, Armadillo data types and functions and data classes have been used extensively. Since the original R package (Section 6.1) already had an implementation of this particular function in C++, we used this in our implementation to begin with and then optimised it as required. In our code, `m` denotes number of rows and `n` denotes number of columns of the input matrix. Since the transpose of the matrix is being used from the Boson Sampling function, `m == n+1`. Using this notation, recall the formula for calculating the permanent minors,

$$\operatorname{Per} B^{\diamond}_{m,l} = \frac{1}{2^{m-2}} \sum_{\delta} \left(\prod_{s=1}^{m-1} \delta_s \right) \prod_{i \in [m] \setminus l} v_i(\delta), \quad l \in [m], \tag{16}$$

where $\delta \in \{-1,1\}^{m-1}$ with $\delta_1 = 1$ and $v_i(\delta) = \sum_{j=1}^{m-1} \delta_j b_{ji}$. The two tricks to speed up the naive implementation were to iterate the values of δ over the Gray code, and to compute the innermost product $\prod_{i\in[m]\setminus l} v_i(\delta), l \in [m]$ with the help of the forward and backward cumulative products (Section 5.3). Since there is a lot going on in this single formula, we divide the operations being performed into 5 parts for a clearer explanation:

a) Iterating δ over Gray code: In order to iterate over values of $\delta \in \{-1,1\}^{m-1}$ with $\delta_1 = 1$ in Gray code order (Section 3.1), the function uses the help of two additional variables, `int j` and `arma::ivec g`: an integer, and a vector of integers from Armadillo respectively. `j` is the 'active index' of the Gray code. In order to go from one value in the Gray code to the next, the j^{th} bit is flipped. Additionally, the original code uses an auxiliary array `g` to change from one value of `j` to the next. The condition of the `while` loop ensures that all the Gray codes of size $n-2$ are iterated over. The δ array is represented by a boolean valued vector `d`, with `true` being equivalent to 1, and `false` equivalent to -1. The relevant snippets of code from the function are shown below.

```
int j = 0, k;
arma::uvec d(n); d.fill(true);
arma::ivec g = arma::regspace< arma::ivec>(0,(n-1));
...
while(j < n-1){
   ...
```

```
    d[j] = !d[j];
    // Iterate Gray code: j is active index
    if( j > 0){
        k = j + 1; g[j] = g[k]; g[k] = k; j = 0;
    } else {
        j = g[1]; g[1] = 1;
    }
}
```

b) Computing $\mathbf{v}(\delta) = \{v_i(\delta), i \in [m]\}$ efficiently for each δ: At the start of the function, $\mathbf{v}(\delta)$ is initialised to the row-sums of the input matrix `C`, divided by two. This is because δ is initialised to an all-1 (or equivalently all-`true`) vector, so $v_i(\delta) = \sum_{j=1}^{m-1} \delta_j b_{ji} = \sum_{j=1}^{m-1} b_{ji}$. The actual code does a row sum instead of a column sum like the formula suggests due to the matrix being transposed. The reason for dividing by 2 is explained later. In each iteration of the loop over δ, the active index j is used to check the value of δ_j, which is the element of δ about to be flipped, and accordingly adds or subtracts the j^{th} column to $\mathbf{v}(\delta)$. These operations are performed in the following lines of code.

```
...
v = arma::sum(C,1)/2;
...
while( ... ){
    if(d[j]) v -= C.col(j); else v += C.col(j);
    ...
}
```

c) Computing the partial products $\prod_{i\in[m]\setminus l} v_i(\delta), \quad l \in [m]$: In order to exploit the trick of calculating the partial products of $\mathbf{v}(\delta)$ quickly, we use a vector `p` to store the accumulated result of calculating the permanent minors, and directly add or subtract the partial products to this vector. `p` is initialised with the partial products of the initial values of `v` which can be seen in the lines of code that follow

```
...
arma::cx_vec p(m), q(m);
arma::cx_double t;
...
q = arma::cumprod(v);

t = v[m-1];         // last element of v
p[m-1] = q[m-2];
```

```
for(i = m-2; i > 0; i--) {
    p[i] = t*q[i-1];
    t *= v[i];
}
p[0] = t;
...
}
```

In this code, `q` is the vector of forward cumulative products, and `t` is a single variable being used to successively store values of the backward cumulative product without having to precompute and store all of them. The same method as above is used while adding or subtracting these partial products in each iteration of the loop.

d) Computing the value $\left(\prod_{s=1}^{m-1} \delta_s\right)$: Another result of iterating over δ in Gray code order is that the product $\prod_{s=1}^{m-1} \delta_s$ does not need to be calculated in each iteration of the loop. The reason being that exactly one element of δ is flipped in each iteration, so $\prod_{s=1}^{m-1} \delta_s$ alternates between the values $+1$ and -1 in every successive iteration. Equivalently, it means that we alternate between adding and subtracting the partial products to the accumulator in successive iterations of the loop. Therefore, this product is represented by a single boolean variable `s` that is negated at the end of each loop iteration.

```
...
s = true;
...
while(j < n-1){
   ...
   if(s){
        //Subtract partial products of v from p
    } else {
        //Subtract partial products from p
    }
    s = !s;
    ...
}
```

e) Multiplying by $\frac{1}{2^{m-2}}$ term: Finally, the sum in the formula is divided by 2^{m-2}, but rather than evaluating this separately, and dividing all the elements of the result by it, we initially divide each element of $\mathbf{v}(\delta)$ by 2. Then, when the partial products of $\mathbf{v}(\delta)$ are being computed, each partial product is divided by a factor 2^{m-1}, and then assigned to the accumulator `p`. Before returning `p` at the end of the function, it is multiplied by 2 in order to give the desired value of 2^{m-2} in the denominator.

The object `p` returned is a vector containing the set of permanent minors of the input matrix `C`.

6.2.3 Simulating exact Boson Sampling using Algorithm B

The function `bosonSampler(arma::cx_mat A, int n, int m)` is an implementation of Algorithm B (Section 5.3) of the exact Boson Sampling problem in C++. We followed the algorithm exactly as it is, and describe a line-by-line explanation below (excluding trivial steps). The function takes as arguments a complex-valued Armadillo matrix `A` which is a random unitary matrix as required by the algorithm, an integer `n` representing input size, and an integer `m` representing output modes.

Preprocessing steps Before we start the actual algorithm, A is required to be an $m \times n$ matrix as we require only the first n columns. A random seed is generated using a Mersenne twister engine which is a uniform fast pseudo-random number generator [37]. The random seed is required later for the sampling steps in the algorithm.

```
// Take first n columns of A
A.set_size(m, n);

// Generate random seed
random_device rd;
mt19937 gen(rd());
```

Line 2 requires the columns of A to be randomly permuted. In order to do this, we have implemented a simple Knuth-shuffle algorithm [32].

```
// Line 2 : Permute columns
for (int i = 0; i <= n-2; i++) {
    uniform_int_distribution<int> uni(i, n-1);
    int j = uni(gen);
    A.swap_cols(i, j);
}
```

Line 3 and 4 involves making a weighted array $\mathbf{w}$ and sampling a value x from it. We created a C++ standard vector of doubles to represent the weighted array using the formula provided in Line 3 of the algorithm. C++ provides a class for creating creating discrete distributions using indexed weight arrays or vectors. This was used to generate the distribution and the random seed generated earlier was used to sample x from it. 1 is added to the value sampled, as indices of C++ vectors start at 0, but the problem requires the indices to start at 1. This sampling was carried out in the code snippet below.

```
// Line 3
vector<double> w;
for (int i = 1; i <= m; i++) {
    w.push_back(norm(A(i-1, 1)));
}

// Line 4
discrete_distribution<> d(w.begin(), w.end());
int x = d(gen) + 1;
```

Lines 6 to 11 contains the loop for repeatedly generating the marginal distributions and sampling values from them. Firstly, the permanent minors are computed of the submatrix of A and stored in an Armadillo vector, by making a call to `cxPermMinors(arma::cx_mat)`:

```
// Line 8
arma::cx_vec perms;
perms = cxPermMinors(B_k);
```

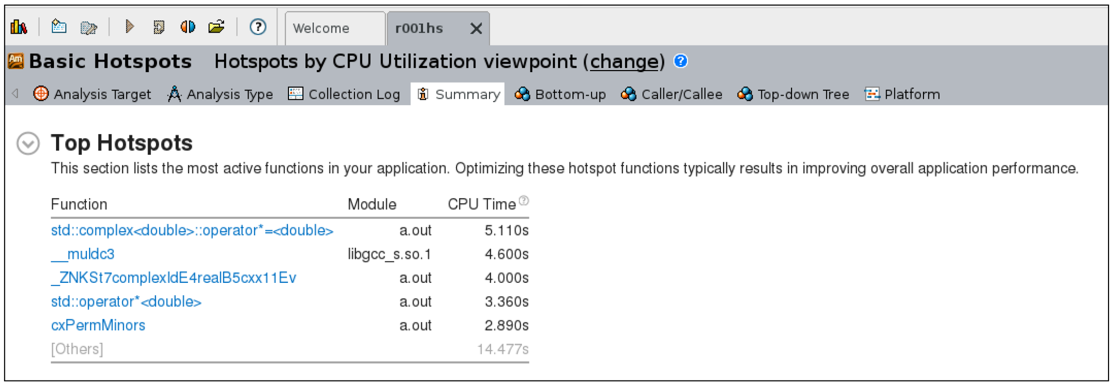
The permanent minors are used to create a weight vector using the formula in Line 9. As before, an integer x is sampled from the discrete distribution represented by a weight vector $\mathbf{w}$, and then appended to the vector $\mathbf{r}$.

Line 12 is the final step of sorting the vector of sampled values r in non-decreasing order. The `sort()` function in the standard C++ library does exactly this for vectors.

```
// Line 12
vector<int> z;
sort(r.begin(), r.end());
z = r;
```

6.3 Running the code

The files containing the functions were initially compiled on a personal computer, using the g++ compiler with no extra flags added for optimisation. The `main()` function required the user to enter a value of n when running the function, and then set $m = n^2$. An $m \times m$ complex valued random unitary matrix is generated using the `randomUnitary(int)` function. Subsequently, the `bosonSampler(arma::cx_mat, int, int)`



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
std::complex<double>::operator*=<double>	a.out	5.110s
__muldc3	libgcc_s.so.1	4.600s
_ZNKSt7complexIdE4realB5cxx11Ev	a.out	4.000s
std::operator*<double>	a.out	3.360s
cxPermMinors	a.out	2.890s
[Others]		14.477s

Figure 2: A summary of ‘hotspots’ in the code identified by the Intel Vtune Profiler

function is called, and a single sample from the Boson Sampling distribution with input size n and m output modes is obtained. Finally, this sample is printed out to the command line. An example of this process with $n = 10$ is shown below. The following instructions are run in the same directory as the C++ files.

```
> g++-8 bosonSampling_b_arma.cpp cxPermMinors.cpp randomUnitary.cpp \
>         -o a.out -std=c++11 -larmadillo
> ./a.out 25
[ 7 21 28 37 38 47 63 67 91 97 ]
```

6.4 Profiling the code

Once the code was running as expected, we could begin working on its efficiency and reducing the observed runtime. In order to gain a better understanding of the how the runtime of the program was divided across the functions, the Intel Vtune profiler[38] was used. Code profilers dynamically analyse executable code that is running, and identify bottlenecks by giving a detailed break down of information about CPU usage, and time taken to run individual lines of the code. The profiler was run on the compiled code, and a screenshot of the results observed is shown in figure 2. The top 4 ‘hotspots’ identified were all related to the operation of multiplying two objects of type `std::complex<double>`. The Bottom-Up representation within the profiler gives information on where exactly in the code the hotspots can be found. This representation showed that the top 4 hotspots were all within the `for` loop of `cxPermMinors( ... )`, which is exactly as expected since the runtime of the algorithm is dominated by the loop required for computing the permanent minors.

Note that the second hotspot ‘__muldc3’ is an internal library routine, which is run by the C++ compiler when two complex numbers are multiplied. To see this, we used an online tool called GodBolt [39] which converts high-level code in some programming language (C++ in our case) to low-level, compiler-dependent assembly code that. This showed that in assembly-level language, the g++ compiler essentially replaced lines containing the multiplication of two complex numbers with a list of instructions, one of which was '__muldc3'. Complex multiplication is a fairly straightforward operation that can be simplified as follows,

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i \tag{17}$$

which is an operation that one would expect to be calculated very efficiently in C++ as it is reduced to a few basic arithmetic calculations on real numbers. However, an open-source implementation of '__muldc3' [40] shows that the function is much larger than expected, due to a number of checks being made for edge cases on complex numbers. This is another bottleneck that could potentially be resolved. The third hotspot is simply a mangled name for an internal library routine that deals with complex numbers in C++.

To avoid having to run the profiler every single time, a timer function was added into the code to keep track of how long the entire program takes to run, as well as the cumulative runtime of the `cxPermMinors(...)` function.

6.5 Parallelising the code

6.5.1 Parallelisation technique

The most obvious solution to speed up the process of calculating the permanent minors is to parallelise Lines 6 to 11 of the algorithm which corresponds to the `for` loop in the code which iterates over the values of δ.For an input matrix of size $n \times n - 1$, there are $(2^{n-1} - 1)$ iterations of the loop. As n grows, this number grows exponentially, so making the iterations run in parallel would have a significant impact. The reason that the loop is in fact parallelisable is that its purpose is essentially to generate some values in each iteration and add them to an accumulator. In order to parallelise it, each thread would have to maintain its own local accumulator, and the total number of iterations could be divided equally among the threads. After each thread completes its assigned set of iterations, it would add the values in its local accumulator to a global accumulator to a global accumulator, which would finally hold the required end result. However, this was not as trivial as it sounds since a number of variables within the loop were dependent on their values in the previous iteration. Hence, we needed to find a way to initialise these variables to the correct values for any iteration number so that a thread could be given these values to start with, and then start its set of iterations. The modifications made for this parallelisation to work are explained as follows,

a) Getting the value of the active index j in the k^{th} iteration: The loop in the original code iterates over δ in Gray Code order by keeping track of the 'active index' j and then flipping the j^{th} bit of δ at the end of each loop iteration. This active index j is also required in calculating successive values of the vector $\mathbf{v}$, so it was also essential to derive a method for finding the value of j in the k^{th} iteration. To see how the value of j relates to the iteration number k, refer to Table 1. Notice that j is equal to the position of the least significant bit set to 0 in the binary representation for k. Equivalently, it is the number of of trailing 0-bits in $j+1$, starting at the least significant bit position. There exists an intrinsic function (or built-in function) which does this exact operation for us. Intrinsics are a special collection of functions which are handled directly by the compiler, and are mapped directly to x86 SIMD(Single instruction, multiple data) instructions. Since these intrinsics are nearly equivalent to directly using actual assembly code, they are also highly efficient. For example, the corresponding intrinsic in the g++ compiler is `__builtin_ctzll (unsigned long long)`. Using this function in each iteration of the loop also allows us to get rid of the auxiliary array `g` in the original code. it is used in the following way:

```
int getActiveIndex(long long ctr) {
        return __builtin_ctzll(ctr);
        // _mm_tzcnt_64 for intel compiler
}
...
for (...) {
    ...
    j = getActiveIndex(ctr+1);
}
```

b) Getting the value of k^{th} element of the Gray code: Once we have the correct values of j in each iteration of the loop, we can use it as before to iterate from one element in the Gray code to the next in the same way as in the original implementation. However, if a thread is made to start from the k^{th} iteration, it also needs to be assigned an initial value for δ. Once again, refer to the example in Table 1 to see how elements of the Gray code are related to iteration number k. Notice that the Gray code corresponding to index j is actually equal to j XOR $j >> 1$ where $j >> 1$ represents a single bit shift to the right. A proof of this formula by induction is shown in the appendix. Additionally, integer representation of the Gray code element is required to be converted to the δ representation, where 1s in the Gray code are represented by 0s in δ and 0s in the Gray code are represented by 1s. The functions for these operations are called by each thread right before it starts its loop iterations.

```
int getKthGrayCode(long long k) {
    return k ^ (k >> 1);
}
```

c) Getting the value of `s`: Recall that `s` was a boolean variable which decided whether partial products would be added or subtracted to the accumulator. Since in the original serial code, the value of s alternated between true and false across iterations,

Table 1: Elements of Gray code and active indices for $n = 4$

Iteration number k	k **in binary**	**Active index** j	**Gray code elements** δ
0	000	0	000
1	001	1	001
2	010	0	011
3	011	2	010
4	100	0	110
5	101	1	100
6	110	0	101
7	111	3	111

we use the fact that for an even iteration number, `s` was set to `false` and for odd iterations, `true`. This needs to be done only once for each thread, as the value of `s` could simply be flipped at the end of each iteration to update the value correctly.

```
//my_start is the starting index for a thread
if (my_start%2 == 0) s = false;
                else s = true;
```

d) Computing the v array for iteration k: Computing the value of **v** for a given δ requires a direct interpretation of the formula $v_i(\delta) = \sum_{j=1}^{m-1} \delta_j b_{ji}$ from the equation for calculating permanent minors. Once again, this only needs to be done once for each thread since we use a different and more efficient method to iterate from one value of **v** to the next, which was described in section 6.2.2.

```
arma::cx_vec getV(arma::uvec d, int j, int n, long long ctr, arma::cx_mat C) {
    arma::cx_vec v(n);
    v = arma::sum(C,1)/2;
    for (int i = 0; i < n; i++) {
        if(d[i] == 0) v -= C.col(i);
    }
    ...
    return v;
}
```

6.5.2 Using OpenMP to modify the code for use with multiple threads

The code was parallelised using OpenMP pragmas (Section ??). The loop was converted from a ‘while’ loop to a ‘for’ loop. In addition, instead of carrying out the first iteration before the loop and then running the loop for $2^{n-1} - 1$ iterations, the loop is made to run for a collective total of 2^n iterations.

Consider the case where the number of threads is equal to a multiple of 2. In this case, dividing the iterations between the threads is fairly straightforward. The threads are indexed $0...t-1$, if there are t threads being used. The starting loop index for each thread is defined by the following formula:

$$\text{Start Index} = \frac{2^{n-1}}{t} * \text{threadIndex} \tag{18}$$

and similarly, the ending loop index for a thread is calculated in the following way

$$\text{End Index} = \frac{2^{n-1}}{t} * (\text{threadIndex} + 1). \tag{19}$$

Hence, each thread runs $2^{n-1}/t$ loop iterations.

However, in the case that the number of threads is not a power of 2, the iterations are not divided equally among all threads but we do ensure that they are divided in such a way that each thread has no more than 1 extra iteration compared to any other thread. This is done with a slight modification on the formulas above, where the if there are x extra iterations, the first x threads get $\text{floor}(2^{n-1}/t) + 1$ iterations.

Threads are created using the `pragma omp parallel` construct, and then the start and end index of the loop is defined for each thread, Along with this, the number of threads, and scope of variables is specified in the `pragma`.

```
#pragma omp parallel num_threads(numThreads) private(d, v, s, q, j) shared(C, p)
{
    ...
}
```

As the code shows, only the matrix `C` and global accumulator `p` are shared across threads. The rest of the variables are made thread local as they are modified by each thread individually at the same time. The variables local to the loop are initialised to the appropriate values, and then the loop is run. Finally, the local accumulator is added to the global accumulator using the `pragma omp critical` construct.

```
#pragma omp critical
{
    p+= p_local;
}
```

Using the techniques discussed above, we were able to parallelise the code. Checks were run on the code and it was tested against known values of permanents of matrices to ensure correctness.

6.6 Other Oprtmisations

6.6.1 Compiler choice

The execution time of the code is determined by the number of instructions that the CPU executes. Since the compiler is responsible for translating our high-level C++ code to a set of low-level assembly instructions, the choice of compiler can have a significant effect on performance. We test our code with two different compilers: the GCC compiler, and the Intel C++ compiler, both of which provide support for OpenMP.

GCC compiler: GCC (or GNU Compiler Collection) actually refers to a collection of compilers for different programming languages, and is widely used especially for C++. The version we use is GCC 7.2. GCC is a portable, multi-platform compiler and can produce outputs for most types of processors. It is a free software, distributed under the GNU General Public License [41].

Intel C++ compiler: The Intel C++ compiler is a highly optimised compiler for systems using processors that support Intel architectures. The Intel compiler automatically performs operations such as Single Instruction Multiple Data vectorisation, which speeds up standard operations by performing them simultaneously using the processors hardware. In addition, it also carries out loop and memory transformations to speed up a program. The only caveat is that it may not carry out these intensive optimisations for systems without Intel architectures. It is freely available for students, educators and open-source developers [42].

6.6.2 Compiler options

Both compilers offer useful optimisation options, which we test on our program as well. Compiler options are not required to compile the program, but they do provide the ability to change how the code is processed by the compiler. The options we tested are summarised in the following tables [43].

7 Critical Evaluation

We tested out the above implementation at each stage on the Blue Crystal supercomputer and recorded the timings. The relevant graphs were generated using the `matplotlib` library from Python.

Option	**Notes**
`-Ofast`	Flags beginning with `-O` enable a number of different optimisation options. `-Ofast` actually combines `-O3`, which provides the highest level of optimisations and `-ffast-math`, which tells the compiler to ignore a number of IEEE or ISO rules/specifications for some math functions. Hence, this may speed up code greatly, but is not always safe to use.
`-march=native`	This tells the compiler what exact code it should produce for the systems processor architecture, since different CPUs support different instruction sets.
`-funroll-loops`	Unrolls loops whose number of iterations can be determined at compile time or entry to the loop. This option does make the code larger, and may or may not speed up the code.
`-pipe`	Has no effect on the generated code, but it speeds up the compilation process.

Table 2: Compiler options used with the GCC compiler

Option	**Notes**
`-O3`	Flags beginning with `-O` enable a number of different optimisation options. `-O3` provides the highest level of optimisation by enabling compiler vectorisation, and aggressive loop transformations among other techniques.
`-xHOST`	This tells the compiler what exact code it should produce for the systems processor architecture, since different CPUs support different instruction sets.

Table 3: Compiler options used with the Intel C++ compiler

7.1 Initial implementation

We first tested out the C++ code written by us without any optimisations made, compiled with both the Intel and gcc compilers. This has been plotted as a graph shown in figure 3. For each value of n from 2 to 30, 10 results were recorded, with their timing taken. The mean and standard deviation of the code runtimes for each of these values was calculated and is depicted in the graph. The error bars represent one standard deviation in the data. Notice that the y-axis of the graph uses a log-scale, which is why the lines appear to be roughly linear. We expected the time taken to grow exponentially, and this was the observed result. These results are as expected, with the Intel compiled code being significantly faster than the gcc compiled code. The average time for the gcc compiled code to complete the algorithm was roughly 1100 seconds, whereas for the intel compiled code, it was 100 seconds. The reason that the Intel compiled code is over 10 times faster is that it is highly optimised by the compiler even without specifying any additional optimisation options. Additionally, we can see that

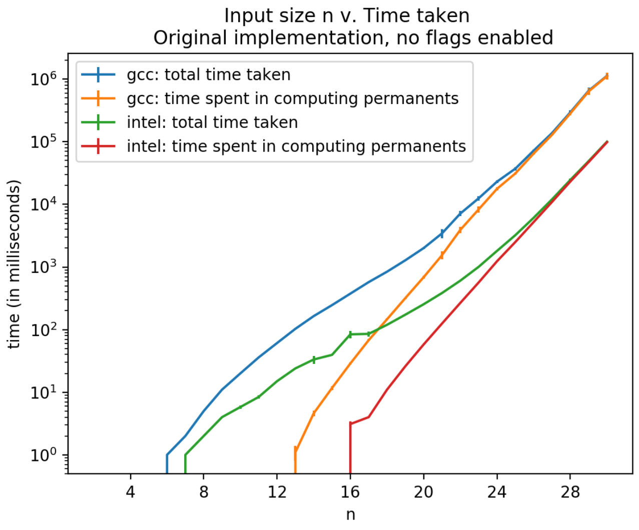


Figure 3: Original implementations

the time taken for computation of permanents is consistently only a bit less than the total time taken, except for smaller values of n, in which case the amount of time taken to compute permanents was 0. This is because for small sizes of matrices, calculating the permanents requires only a few simple mathematical calculations. The overheads of running the rest of the algorithm appears to dominate for these small values of n.

It was also interesting to note that the timings to run the Intel compiled code were a lot more consistent than the timings for the gcc compiled code. The standard deviation for the gcc code with higher values of n was roughly 130, whereas for intel, it never exceeded 0.8, giving the coefficients of variation 11% and 0.8% respectively.

7.2 Testing the different compiler option/flags

After parallelising the code, in order to compare the different compiler options, the code was run on 16 cores with different options enabled each time. As before, 10 samples were taken for each value of n, and the values plotted in the graphs are the means

7.2.1 gcc

We had four different compiler options to test with gcc, as explained in section 6.6.2. The `-funroll` option makes almost no difference, which we anticipated since it is known to not improve timings in every case. Specifically, in the case of our loop which was run an exponential number of times for permanent calculation, unrolling loops would potentially make the process of going from one iteration to the next faster, but a lot more memory would be used as a cost, which would slow it back down. What was more surprising was that the `-march=native` flag had only a minute effect (1 to 2 seconds for n=30) in speeding up the time. On the other hand, the `-Ofast` flag provided a

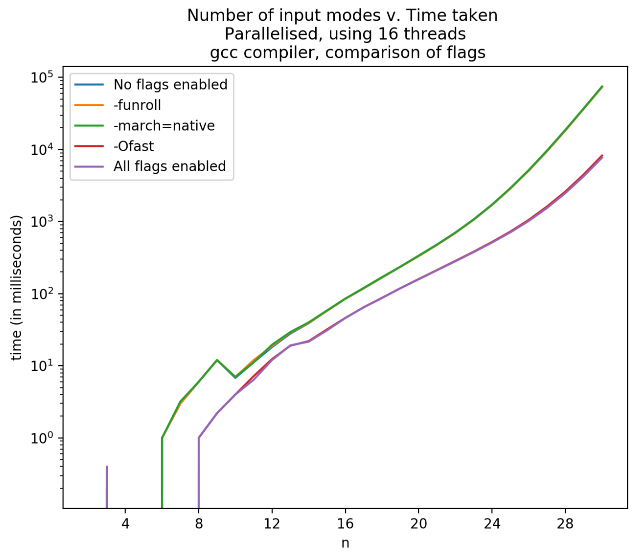


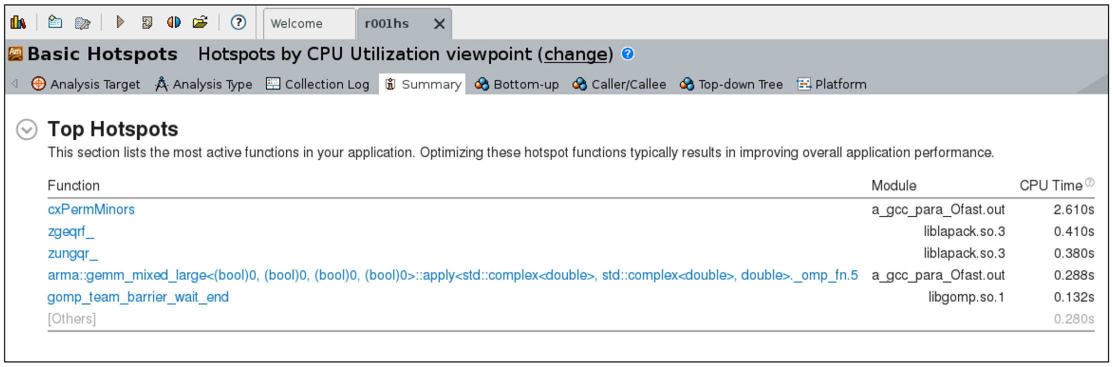
Figure 4: Comparing different gcc compiler options

major speed up, making the code over 10 times faster to run than with no compiler options enabled. The code was profiled again at this stage to understand how such a significant speed up was made. The results are shown in figure 5. The difference is that the top few hotspots are no longer dominated by the inbuilt function '__muldc3', and other specific operations related to multiplying complex numbers. The reason this has happened is that using `-Ofast` tells the compiler to ignore strict ISO standards, and instead of running the complex number multiplication function with a lot of checks, it simply performs the bare operations. While it does make the code susceptible to errors in edge cases (specifically involving operations where the real part of the number is set to infinity), we expect to never encounter these cases in our code, and can ensure that we don't by adding in some checks of our own.

7.2.2 Intel compiler

The two chosen flags were tested on the Intel compiler and surprisingly, neither of the two made a big difference to the timing. This is shown in figure 6. Considering how much faster the Intel compiled code was even before any options were added, we presume that the Intel compiler available on Blue Crystal is maximally optimised for speed by default.

After enabling optimisation flags, the gcc and Intel compiled code have similar timings for $n = 30$, with the gcc compiled code occasionally being 1-2 seconds faster.



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
cxPermMinors	a_gcc_para_Ofast.out	2.610s
zgeqrf_	liblapack.so.3	0.410s
zungqr_	liblapack.so.3	0.380s
arma::gemm_mixed_large<(bool)0, (bool)0, (bool)0, (bool)0>::apply<std::complex<double>, std::complex<double>, double>._omp_fn.5	a_gcc_para_Ofast.out	0.288s
gomp_team_barrier_wait_end	libgomp.so.1	0.132s
[Others]		0.280s

Figure 5: A summary of hotspots found in the code identified by the Intel Vtune Profiler after enabling the -Ofast flag

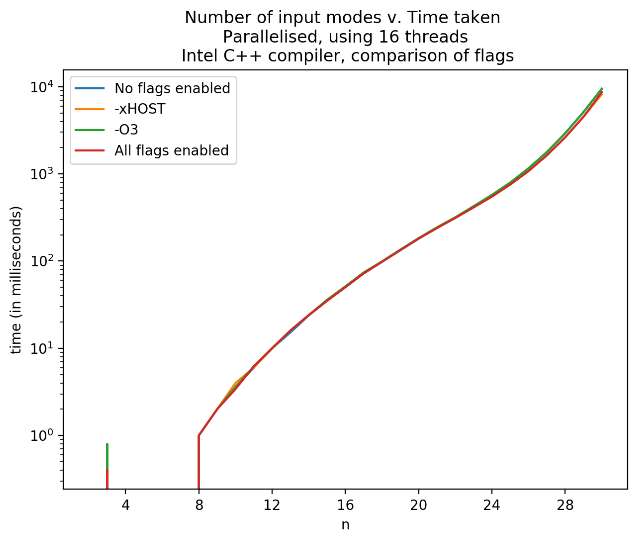


Figure 6: Comparing different Intel compiler options

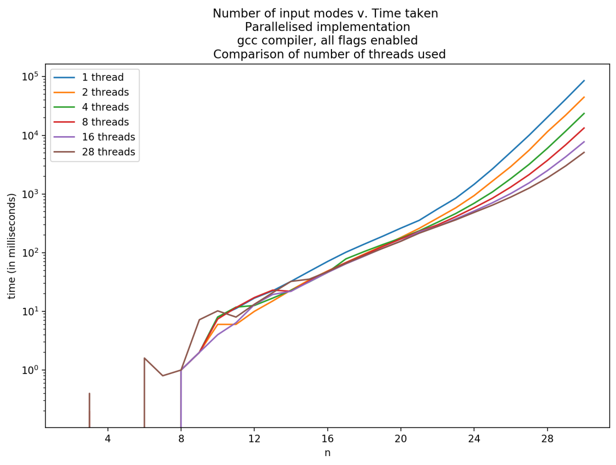


Figure 7: Comparing different number of threads used with gcc compiled code

Table 4: Timings to run code (in seconds), as number of threads is changed, for $n = 30$

No. of cores	**Time for gcc**	**Time for Intel**
1	85	88
2	44	46
4	23	24
8	13	14
16	8	8
28	5	5

7.2.3 Multiple threads

One of the main results of our paper was to parallelise the algorithm, and the results obtained by doing so were as expected. The code was run after being compiled with both compilers and all listed flags enabled, with a different number of threads available each time for the sake of comparison. The results we obtained showed a speed up proportional to the number of threads used, shown in figures 7 and 8. In a few cases for values between $n = 8$ and $n = 12$, we can see that in figure 9 there are some inconsistencies, and using more threads might take a few milliseconds more than with less threads. This is because multithreading does have overheads with assigning values to threads as well as with creating threads.

Another set of timings were taken for running the Boson Sampling code with $n = 30$, and successively increasing the number of threads, shown in figure 9. The results show that the timing reduces in steps as the number of cores used is increased. One can also notice the trend that as the number of threads is doubled, the timing is roughly halved. A summary of the actual timings are also shown in table 4.

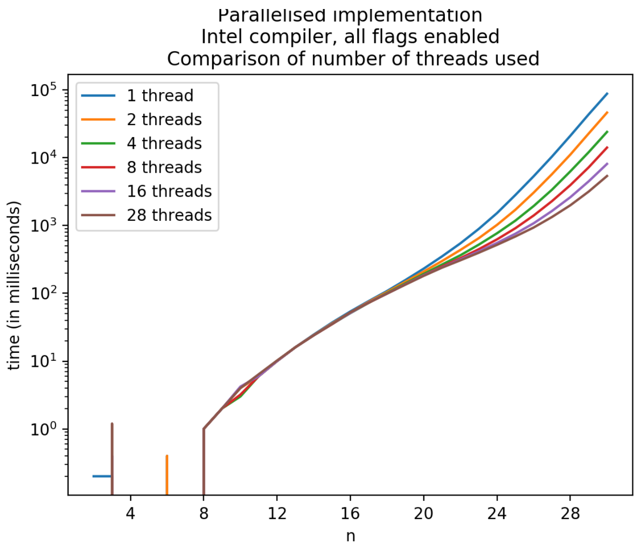


Figure 8: Comparing different number of threads used with intel compiled code

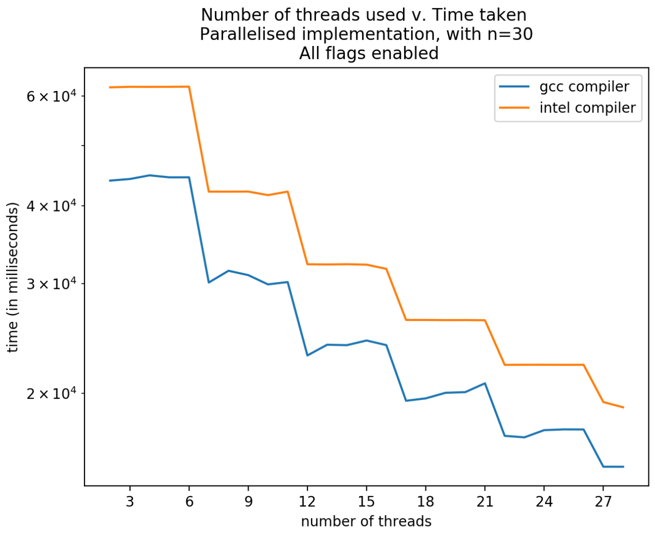


Figure 9: Comparing different number of threads used

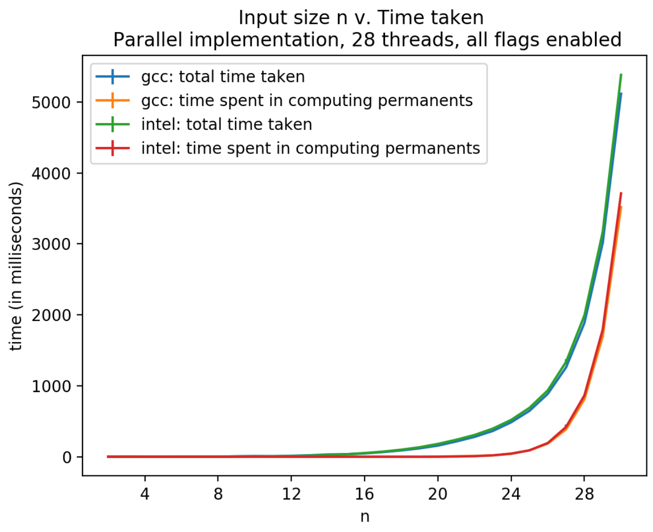


Figure 10: Comparing final results

7.2.4 Final result

Using all the optimisations we have made, we ran the code on Blue Crystal, using 28 cores, and with all mentioned compiler-flags enabled for optimisation. The results are shown in 10. The gcc compiled code gives marginally faster results as compared to the Intel compiled code, and the times taken to simulate sampling bosons for $n = 30$ are 5.1 seconds and 5.3 seconds respectively.

8 Conclusion

The final results produced by us showed a speed up of around 200 times with the gcc compiler, and 20 times for the Intel compiler, as compared to our initial measurments. We also ran a few tests for $n = 35$, and it took roughly 320 seconds to run. While these numbers may not seem impressive compared to the benchmark on the Tianhe-2 supercomputer [20], our implementation uses a maximum of 28 cores, whereas the benchmark for $n = 50$ computed in 600 minutes was made using 312?,000 cores. We project that in order to break that benchmark, we would need access to a supercomputer with only $\approx 10,000$ cores.

8.0.1 Further Works

References

[1] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing.* Oxford University Press, Inc., New York, NY, USA, 2007.

[2] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *arXiv e-prints*, pages quant–ph/9508027, Aug 1995.

[3] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[4] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK, 2003.

[5] Aram W. Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, Sep 2017.

[6] Thomas Monz, Daniel Nigg, Esteban A. Martinez, Matthias F. Brandl, Philipp Schindler, Richard Rines, Shannon X. Wang, Isaac L. Chuang, and Rainer Blatt. Realization of a scalable shor algorithm. *Science*, 351(6277):1068–1070, 2016.

[7] Juha J. Vartiainen, Antti O. Niskanen, Mikio Nakahara, and Martti M. Salomaa. Implementing Shor's algorithm on Josephson charge qubits. *Phys. Rev. A*, 70(1):012319, Jul 2004.

[8] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing Quantum Supremacy in Near-Term Devices. *arXiv e-prints*, page arXiv:1608.00263, Jul 2016.

[9] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W. Draeger, Eric T. Holland, and Robert Wisnieff. Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits. *arXiv e-prints*, page arXiv:1710.05867, Oct 2017.

[10] E. Knill, R. Laflamme, and G. J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46–52, 2001.

[11] Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 333–342, New York, NY, USA, 2011. ACM.

[12] Margherita Barile and Eric W Weisstein. Galton board. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GaltonBoard.html.

[13] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20:865–877, 10 1991.

[14] E. R. Caianiello. On quantum field theory — i: explicit solution of dyson's equation in electrodynamics without use of feynman graphs. *Il Nuovo Cimento (1943-1954)*, 10(12):1634–1652, Dec 1953.

[15] Wojciech Roga and Masahiro Takeoka. Classical simulation of boson sampling with sparse output. *arXiv e-prints*, page arXiv:1904.05494, Apr 2019.

[16] P. H. Lundow and K. Markström. Efficient computation of permanents, with applications to boson sampling and random matrices. *arXiv e-prints*, page arXiv:1904.06229, Apr 2019.

[17] Saleh Rahimi-Keshari, Timothy C. Ralph, and Carlton M. Caves. Sufficient Conditions for Efficient Classical Simulation of Quantum Optics. *Physical Review X*, 6(2):021039, Apr 2016.

[18] W. Larry Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[19] Alex Neville, Chris Sparrow, Raphaël Clifford, Eric Johnston, Patrick M. Birchall, Ashley Montanaro, and Anthony Laing. Classical boson sampling algorithms with superior performance to near-term experiments. *Nature Physics*, 13:1153 EP –, 10 2017.

[20] Junjie Wu, Yong Liu, Baida Zhang, Xianmin Jin, Yang Wang, Huiquan Wang, and Xuejun Yang. A benchmark test of boson sampling on Tianhe-2 supercomputer. *National Science Review*, 5(5):715–720, 07 2018.

[21] Peter Clifford and Raphaël Clifford. The classical complexity of boson sampling. *CoRR*, abs/1706.01260, 2017.

[22] Marvin Marcus and Henryk Minc. Permanents. *The American Mathematical Monthly*, vol. 72:577–591, 1965.

[23] Herbert John Ryser. *Combinatorial Mathematics*. Mathematical Association of America, 1963.

[24] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.

[25] Albert Nijenhuis and Herbert S. Wilf. 23 - the permanent function (perman). In *Combinatorial Algorithms (Second Edition)*, pages 217 – 225. Academic Press, second edition edition, 1978.

[26] K Balasubramanian. *Combinatorics and diagonals of matrices*. PhD thesis, Indian Statistical Institute, Calcutta, Dec 1980.

[27] Eric Bax. *Finite-difference algorithms for counting problems*. PhD thesis, California Institute of Technology, February 1998.

[28] Eric Bax and Joel Franklin. A finite-difference sieve to count paths and cycles by length. *Information Processing Letters*, 60(4):171 – 176, 1996.

[29] David G. Glynn. The permanent of a square matrix. *European Journal of Combinatorics*, 31(7):1887 – 1891, 2010.

[30] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 1968.

[31] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, April 1974.

[32] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[33] Peter Clifford and Raphaël Clifford. *BosonSampling: Classical Boson Sampling*, 2017. R package version 0.1.1.

[34] Dirk Eddelbuettel, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Douglas Bates, and John Chambers. *RCpp*, 2017. R package version 0.12.12.

[35] Conrad Sanderson. Armadillo c++ linear algebra library, June 2016.

[36] Maris Ozols. How to generate a random unitary matrix. Technical report, 2009.

[37] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[38] Intel Software. Intel vtune amplifier 2018. Software available at https://software.intel.com/en-us/vtune, 2018.

[39] God-bolt.

[40] muldc3.c - implement __muldc3.

[41] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection.* GNU Press.

[42]

[43]