Project: Implementing a Parallel Version of Grover's Algorithm in a Quantum Simulator

Heye Vöcking 376154

Abstract—We are using a simulated quantum computer to compare the performance of Grover's search algorithm [1] with a faster version proposed by Ozhigov [2]. Grover's algorithm allows to search through a domain of cardinality N for k targets by $\frac{k\pi\sqrt{N}}{4}$ simultaneous queries to an oracle. Ozhigov's version improves the speed by a factor of $\sqrt{2}$ for the case $k \geq 2$. We will discuss Ozhigov's algorithm, provide an implementation, and analyze its expected running times to confirm the claimed speedup.

Index Terms—Grover's algorithm, iterated search, quantum computing simulator, Python, QuTiP

1 Introduction

Quantum computing has been an exciting field of research ever since Feynman proposed quantum computation in 1982 [3] and Deutsch proved in 1985 [4] that entangled quantum bits—in short *qubits*—can be used to achieve a polynomial speedup over calculation with classical bits.

However, quantum computers are a delicate piece of hardware and their implementation still poses some challenges that have to be overcome [5]. The currently most powerful quantum system applicable to our problem, has 72 qubits. It is the quantum computer developed by Google in 2018 [6]. However, the actual power of a quantum system, the quantum volume, cannot solely be described by the number of physical qubits. It also depends on the connectivity of the device, the number of parallel operations, and the number of gates that can be applied before errors cause the device to essentially behave classically [7]. Therefore, a physical quantum system will not perform as reliable as a system that simulates the same number of qubits.

Despite the limited power of physical quantum systems available today, research on quantum algorithms is still a very relevant topic, because once the hardware has caught up, the algorithms developed and tested on quantum simulators can be executed on physical quantum computers as soon as they become available. The currently largest quantum computing simulator can simulate up to 121 qubits [8].

With our hardware, we can simulate a system of 18 qubits, which is sufficient to run Ozhigov's algorithm for a domain described by 6 bits.

2 Background

In this section, we are describing Grover's search algorithm, Ozhigov's parallelized version of it, and introduce the simulator we are using.

2.1 Quantum Simulator

We decided to use the Python programming language due to its simplicity and the availability of libraries, like NumPy [9], for scientific computing. As quantum simulator we are using the 4.5.0 version of the *Quantum Toolbox in Python* (QuTiP) developed by Johansson et al. [10], [11]. We mainly make use of its implementation of tensor multiplication, as well as the CNOT and the SNOT (Hadamard) gate. It uses NumPys sparse complex matrices, which allows for the intuitive handling of quantum objects. They can be added, subtracted, multiplied, and divided (by a scalar) with standard Python operators.

2.2 Grover's Algorithm

Given an oracle (a black box function) the quantum search algorithm described by Lov Grover finds, with high probability, the unique input that causes the oracle to return a particular value. The oracle is a function f that takes a binary number x as the input and returns the value 1 in a single point x^0 and the value 0 in any other point. We call the input x^0, where $f(x^0) = 1$, the *solution* or the *target*. As the oracle is a black box function it is impossible to determine *how close* an input x is to the target x^0.

An example function is:

$$f(x) = \begin{cases} 1 & \text{if } x = x^0, \\ 0 & \text{if } x \neq x^0; \end{cases}$$

To find the value x^0, a classical computer needs to call the function sequentially with every possible input, which leads to a complexity of $\mathcal{O}(N)$, where $N = 2^n$ with $n = |x^0|$, the length of the word x^0.

An example application is brute-forcing the 256-bit key of a symmetric encryption algorithm. Using a classical computer, we would need to try $N = 2^{256}$ possible inputs sequentially, in the worst case, which is infeasible with current hardware. This algorithm has a complexity of $\mathcal{O}(2^n)$, where n is the bit count of the symmetric key.

Grover's algorithm reduces the complexity to $\mathcal{O}(2^{\frac{n}{2}})$. In the above example, it reduces the number of queries from 2^{256} when using a classical algorithm, to 2^{128}.

However, the speedup provided by Grover's algorithm for this task can be trivially mitigated by increasing the key length from 256 bits to 512 bits. Thus, the protection offered by symmetric encryption against Grover's algorithm can be restored by doubling the length of the symmetric key to achieve the same protection compared to classical algorithms [12].

The complexity $\mathcal{O}(\sqrt{N})$ of Grover's algorithm is optimal [13], with a lower bound estimated at $\frac{\pi\sqrt{N}}{4}$ and a probability of error at about $\frac{1}{N}$ [14]. This means, there is no substantially faster algorithm for this problem.

Ozhigov shows that, despite the lower bound of $\frac{k\pi\sqrt{N}}{4}$ in the case of $k = 1$, it is possible to improve upon this limit by a factor of $\sqrt{2}$, when the number of targets $k \geq 2$, which we will discuss below [2].

2.3 Ozhigov's Algorithm

Ozhigov defines three different types of searches: *Iterated Search* (IS), *Structured Search* (SS), and *Repeated Search* (RS).

The IS-problem is a sequence of similar searches $S_1, S_2, ..., S_k$ where S_i is the problem of finding the unique solution x_i^0 for the equation $f_i(x_i) = 1$. f_i is a Boolean function that is only accessible if we know all previous solutions to x_i, that we will call x_j, where $j < i$. Furthermore, the lenght of a word $|x_i| = n$, so the domain is $N = 2^n$. The goal is to find x_k^0, with $k \geq 2$ and $k \ll N$.

Sequential applications of Grover's search algorithm for $x_1^0, x_2^0, ..., x_k^0$ give a result after $\frac{k\pi\sqrt{N}}{4}$ applications with an error probability of $\frac{k}{N}$ [14]. The oracles $f_1, f_2, ..., f_k$ are dependent in a way that the oracle f_i depends on all oracles $f_1, f_2, ..., f_j$ where $j < i$. Such that f_i can be assumed to have the form $f_i(x_1, x_2, ..., x_i)$ and each equality $f_i(x_1, x_2, ..., x_i) = 1$ has the unique solution $x_1^0, x_2^0, ..., x_i^0$, $i = 1, 2, ..., k$. It is required that all oracles f_i can be executed simultaneously.

This allows us to take advantage of the interference between the results of their actions caused by the leak of amplitude from one step to the next in the sequential search. The amplitudes of all solutions $x_i \neq x_i^0$ decrease, while the amplitude of x_i^0 in search number i increases from one step to the next, such that it becomes approximately $\frac{2l+1}{\sqrt{N}}$ after the first l steps. This leak of the amplitude can be used for the next $i+1$-th search.

Ozhigov shows that this effect can be used to solve the problem with $\frac{k\pi\sqrt{N}}{4\sqrt{2}}$ queries, which is $\sqrt{2}$ faster than k sequential applications of Grover's algorithm. Ozhigov defines the special case $k = 2$ of IS as the *Repeated Search* (RS) problem. [2]

RS is in itself a special case of the *Structured Search* (SS) with the cardinality $M = 1$. Fahri and Gutmann [15] investigated the case $1 \ll M \ll N$ and developed a quantum algorithm in SS with a complexity of $\mathcal{O}(\sqrt{MN})$. They wrote that the best-known strategy at the time for the case $M = 1$ is the sequential application of Grover's algorithm. Later Ozhigov showed that by using the evolution of amplitudes, instead of algebraic properties, the runtime can be improved by a constant factor of $\sqrt{2}$ [2].

We want to confirm this improvement by applying Ozhigov's approach in a quantum simulator.

3 Approach

Ozhigov describes two different representations: an implementation using differential equations, and a purely matrix-based implementation. We will focus on the latter approach, as it is simpler to work with matrices in the simulator.

The described approach allows for two targets, x^0 and y^0. An operation indicates with the subscripted character x or y on which target it operates. The main execution step in the algorithm is the following equation:

$$Z = \{-W_x R_{0x} W_x \mathcal{F}_1\}\{-W_y R_{0y} W_y F_2\}, \qquad (1)$$

or in its expanded form:

$$Z = -[(I \otimes W_x R_{0x} W_x \otimes I) P F_1 P][-(I \otimes W_y R_{0y} W_y \otimes I) F_2],$$

where I is the identity, W is the Walsh-Hadamard transformation on n_q pairs of qubits

$$W = \underbrace{J \otimes ... \otimes J}_{n_q}, \qquad (2)$$

J is the standard Hadamard transform

$$J = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}, \qquad (3)$$

and R_0 is the rotation of the target state

$$R_0 |t\rangle = \begin{cases} |t\rangle & \text{if } t \neq \overline{0}, \\ -|0\rangle & \text{if } t = \overline{0}; \end{cases} \qquad (4)$$

$\mathcal{F}_1$ is basically F_1 wrapped with P. It is the crucial step, that allows for the improvement because it temporarily saves the result of F_1 in the register u. P is defined like this:

$$P |u, x, y, a\rangle = |u \oplus x, x, y, a\rangle. \qquad (5)$$

The oracle calls are denoted with F_1 and F_2:

$$F_1 |u, x, y, a\rangle = \begin{cases} |u,x,y,a\rangle & \text{if } x \neq x^0, \\ -|u,x,y,a\rangle & \text{if } x = x^0; \end{cases} \qquad (6)$$

$$F_2 |u, x, y, a\rangle = \begin{cases} |u,x,y,a\rangle & \text{if } |x,y\rangle \neq |x^0,y^0\rangle, \\ -|u,x,y,a\rangle & \text{if } |x,y\rangle = |x^0,y^0\rangle; \end{cases} \qquad (7)$$

where u,, x, and y are variables with values of n_q qubits each from three different copies of $\mathcal{H}_0 = C^N$. The ancillary qubits are $a = a_1 \otimes a_2 \in C^4$, with $a_1 = a_2 = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Furthermore, we use the notations $f_1(x)$ and $f_2(x, y)$ for two oracles in the repeated quantum search where x^0 and y^0 represent the values of x and y which are unique solutions, such that $f_1(x^0) = 1$ and $f_2(x^0, y^0) = 1$.

4 Implementation

To demonstrate the difference between Grover's algorithm and Ozhigov's version, we are first explaining the crucial parts of an implementation of Grover's algorithm and then look at the parts modified by Ozhigov.

4.1 Grover's Algorithm in Python Code

The implementation of Grover's algorithm consists of an entanglement step using Hadamard matrices and two diffusion operators enclosing the call to the oracle. These are applied $\frac{\pi\sqrt{N}}{4}$ times in order to yield the maximum probability. N is the possible number of outcomes.

After initializing the variables, we calculate the count of steps that yield the highest chance of returning the correct result:

```
N = 2**n_q
steps = ceil(pi/4*sqrt(N))
I_N = I([2]*n_q)
```

n_q is the number of qubits used, *I_N* is the identity matrix and the functions *ceil*, *pi*, and *sqrt* are imported from the Python *math* package.

The variable *psi* is our quantum register. Representing gates as matrices allows us to multiply the matrix of the gate with *psi* and write the result back to *psi*.

Between steps, we can save the current state of *psi* for later inspection.

As step 0 we apply the Walsh-Hadamard transformation W to entangle the qubits in the register:

```
psi = W(n_q)*psi
```

W(n_q) creates the Walsh-Hadamard transformation matrix, as described in equation 2. It is implemented in the following function, taking the desired size (denoted with n_q in equation 2) as the parameter *s*:

```
def W(s):
  return MUL(
    [snot(s, i) for i in range(s)],
  )
```

The resulting values of a call to *W(2)* and their further development by applying other matrices are shown in figure 1.

The call to *snot* is creating the matrix corresponding to an SNOT-gate. We first create the SNOT-matrices and then multiply them with each other. The *snot* function is the SNOT-gate imported from QuTiP, and *MUL* is a helper function to multiply all elements of a given list:

```
def MUL(items):
  return reduce(mul, items)
```

mul is imported from the Python *operators* package. It is similar to applying the operator $\otimes$ from eg. equation 2.

Z represents the complete circuit for a single step of Grover's algorithm. Building this matrix is a lot more time-consuming than executing a single step. Table 1 lists the measured running times. Z is built in the following line:

```
Z = W(n_q)*R(n_q)*W(n_q)*oracle
```

The *oracle* is a tensor-multiplied identity matrix where one position on the diagonal has the value -1. It is built in *F* for $k = 1$. For $k > 1$, *multi_target_F* can be used:

```
def multi_target_F(targets):
  return MUL(
    [F(t) for t in targets]
  )
def F(target):
  t = [int(b) for b in target]
  ket = tensor(
    *[basis(2, b) for b in t]
  )
  return I_N - 2*ket*ket.dag()
```

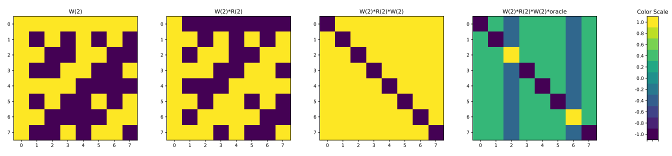

Fig. 1: Visualization of the real-valued parts during the construction of the Z matrix for Grover's algorithm. In the visualization of *"W(2)*R(2)*W(2)*oracle"* we can clearly see the two targets $x^0 = 0b110$ (at 6,6) and $y^0 = 0b010$ (at 2,2) encoded in the matrix.

	Grover		Ozhigov			
n	Z	Step	Z_x	Z_y	Z	Step
1	3.61	0.11	5.39	4.30	10.04	0.14
2	4.87	0.10	8.82	5.52	14.50	0.14
3	6.56	0.11	15.31	7.04	23.68	0.51
4	8.79	0.11	49.04	25.27	119.47	6.96
5	11.73	0.12	665.19	407.66	2,615.12	158.26
6	16.71	0.22	14,831.57	10,355.53	78,565.33	4,670.43

TABLE 1: Run time per matrix build ("Z", "Z_x", "Z_y") and per execution of a single step ("Step") measured in milliseconds, averaged over 100 repetitions, for each Domain (n). The time measured for Ozhigov's Z-matrix does not include the time for building *Z_x* and *Z_y*. It is purely the operation of *Z_x* $*$ *Z_y*.

The function R returns a matrix of size s to rotate the target state, similar to equation 4:

```
def R(s):
  ket = tensor(*[basis(2, 0)]*s)
  r = 2*ket*ket.dag()
  return r - identity([2]*s)
```

where *tensor*, *basis*, and *identity* are imported from QuTiP.

After we have built Z once, we can apply it repeatedly to *psi*. One multiplication simulates executing one query to the oracle and the application of all steps in a single iteration of Grover's algorithm:

```
for i in range(steps):
  psi = Z*psi
```

To observe how the amplitudes evolve, we save a copy of *psi* after each step. A visualization can be seen in figure 6.

4.2 Modifications according to Ozhigov

For simplicity reasons, we are only considering the case $k = 2$ for Ozhigov's algorithm, such that we have 2 targets: x^0 and y^0.

During the execution, we need to apply some gates only to part of our quantum register *psi*.

Thus we have to tensor-multiply them with the identity matrices of the other register parts. Therefore, we first create these identity matrices to have them available for handy access:

```
I_U = identity([2]*n)
I_X = identity([2]*n)
I_Y = identity([2]*n)
I_N = tensor(I_U, I_X, I_Y)
```

n is again the size of the domain. *I_U* stands for the identity matrix of the ancillary register, *I_X* for the identity matrix of the registers of the x target, and *I_Y* for the identity matrix of the registers of the y target. *I_N* is the identity matrix for all registers combined. The *identity* and *tensor* functions are imported from the QuTiP library.

Next, we have to create the Z matrix described in equation 1. We first build two separate matrices, *Z_x* for the x target and *Z_y* for the y target and combine them at the end:

```
def build_Z():
  return build_Z_x()*build_Z_y()
```

The development of values in the matrices is visualized in figure 2.

The *build_Z_y* function is similar to the way we built Z in the implementation of Grover's algorithm.

```
def build_Z_y():
  return (
    W_Y()*R_Y()*W_Y()*F_2()
  )
```

W_Y() and *R_Y()* are similar to *W()* and *R()* in the implementation of Grover's algorithm. The suffix *_Y* indicates that they are only applied to the quantum registers of the y target.

The *W_Y* and *W_X* functions are creating a Walsh-Hadamard transform matrix with the size of the quantum register. They are applied and are then tensor-multiplied with the identity matrix of the other registers.

```
W_Y = lambda: tensor(I_U,I_X,W(Y))
W_X = lambda: tensor(I_U,W(X),I_Y)
```

By comparing *W_Y* and *W_X*, we see that they have to be tensor-multiplied in the correct order so that they only affect their corresponding target qubits.

The rotation step *R()* is changed to only be applied to the corresponding register as well. To set all other positions in the matrix to 0, we subtract the identity matrix at the end as can be seen in the following listing:

```
def R_X():
  ket = tensor(*[basis(2, 0)]*X)
  r = 2*ket*ket.dag()
  return tensor(I_U, r, I_Y) - I_N
def R_Y():
  ket = tensor(*[basis(2, 0)]*Y)
  r = 2*ket*ket.dag()
  return tensor(I_U, I_X, r) - I_N
```

The *tensor* and *basis* are imported from QuTiP.

The y part of the oracle is defined in *F_2* like in equation 7:

```
def F_2():
  f = F(target_y)
  return I_N - tensor(I_U, I_X, f)
```

Where *target_y* is a string of 1s and 0s describing the target y^0 and F the same as in the previous section.

The x part of Z is built similarly to the y part:

```
def build_Z_x():
  return (
    W_X()*R_X()*W_X()*P()*F_1()*P()
  )
```

F_1 is the x part of the oracle, that is implemented similarly to the *F_2* function, as described in equation 6:

```
def F_1():
  f = F(target_x)
  return I_N - tensor(I_U, f, I_Y)
```

The differences and similarities are obvious when looking at the order in which the u, x, and y matrices are multiplied and how they are built into a matrix that can be applied to the quantum registers.

The last missing part is the P function described in equation 5. It is responsible for XORing the qubits corresponding to the x target and the ancillary qubits:

```
def P():
  z = zip(qx, qu)
  return MUL(
    [xor(n_q,c,t) for c,t in z],
  )
```

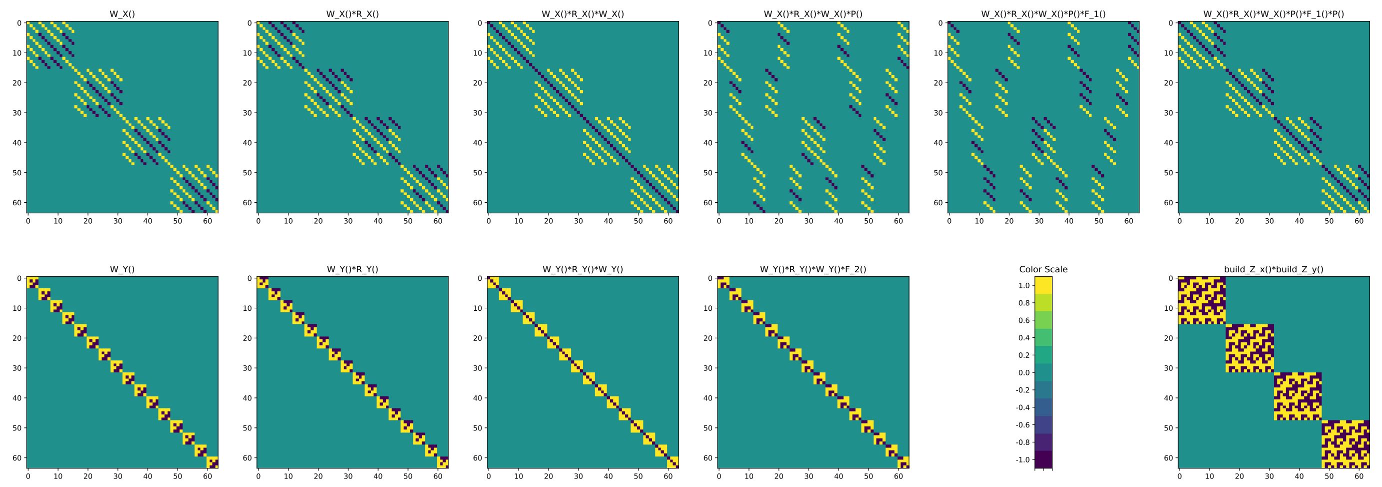

Fig. 2: Real values during the construction of the Z matrix for Ozhigov's algorithm. By comparing the matrix of *"W_X()*R_X()*W_X()"* to *"W_X()*R_X()*W_X()*P()*F_1()*P()"* we can see how the target $x^0 = 0b11$ has been encoded in the matrix. Similarly we can see that encoding of the target $y^0 = 0b01$, when comparing *"W_Y()*R_Y()*W_Y()"* to *"W_Y()*R_Y()*W_Y()*F_2()"*.

Where *xor* is the CNOT-gate from the QuTiP library and the variables *qx* and *qu* are arrays holding the indices of the quantum registers used for the qubits of x and u.

5 Evaluation

For evaluation, we compare the probability development of Grover's and Ozhigov's algorithm for targets of different lengths. However, there are some limitations and peculiarities we have to consider.

5.1 Simulator Peculiarities

As can be seen in table 1, the measured execution times for Ozhigov's algorithm are substantially higher than for Grover's algorithm. The reason behind this is, that Ozhigov's algorithm uses more qubits and the simulator cannot benefit from quantum parallelism as it has to do all calculations explicitly. Therefore comparing the execution times is not subject to this evaluation. However, since all underlying amplitudes have to be calculated in the simulator, we are able to observe them and therefore the resulting probabilities, unlike on a real quantum computer.

5.2 Visualization Compression

The matrix-histogram plots of the probability development have too many values to be displayed in their entirety. To focus on the crucial parts they have been compressed using run-length encoding along the axis of outcomes. It is applied up to two times. Firstly, if a repeating pattern over a stable period is identified, its first occurrence is plotted normally and all the following occurrences are replaced by a symbolic flat line with a label like this *"repeats 705 times..."*. Secondly, whenever the probabilities of more than 3 consecutive entries are the same, they are replaced with 3 symbolic entries and labeled with the first label, the last label, and a label in the middle indicating the number of skipped outcomes, eg. *"... 22 ..."*.

5.3 RAM Limitations

The number of values we need to store for the quantum register is 2^{n_q}, with n_q being the number of qubits. Each value is a complex number, so we need 128 bit or 16 byte per value. We need $(2^{n_q})^2 \cdot 16$ byte of RAM to perform an operation on an n_q-qubit vector. Our machine has 32 GiB of RAM, this allows us to simulate a system with up to 16 qubits if the values are stored in dense matrices, see table 2. However, it is possible to simulate systems with a higher number of qubits if the matrix can be stored in sparse matrices. Ozhigov's algorithm requires $3 \cdot |x^0|$ qubits, but since the matrices representing the gates can benefit greatly from the space-saving through the use of sparse matrices, we can simulate it for a domain described by up

n_q	RAM	n_q	RAM	n_q	RAM	n_q	RAM
		4	1 KiB	9	1 MiB	14	1 GiB
		5	4 KiB	10	4 MiB	15	4 GiB
1	16 B	6	16 KiB	11	16 MiB	16	16 GiB
2	64 B	7	64 KiB	12	64 MiB	17	64 GiB
3	256 B	8	256 KiB	13	256 MiB	18	256 GiB

TABLE 2: RAM required for operating on n_q qubits with dense matrices. Depending on the nature of the data, using sparse matrices can greatly lower the RAM requirement, as can be seen in table 3.

Domain	Grover			Ozhigov		
n	n_q	Expected	Measured	n_q	Expected	Measured
1 bit	1	16 B	94 MiB	3	256 B	94 MiB
2 bit	2	64 B	94 MiB	6	16 KiB	94 MiB
3 bit	3	256 B	94 MiB	9	1 MiB	95 MiB
4 bit	4	1 KiB	94 MiB	12	64 Mib	117 MiB
5 bit	5	4 KiB	94 MiB	15	4 GiB	777 MiB
6 bit	6	256 KiB	94 MiB	18	256 GiB	20 GiB

TABLE 3: Amount of RAM used for different number n_q of qubits. Comparing the *Expected* (using dense matrices) and *Measured* (using sparse matrices) results for Grover's and Ozhigov's algorithm. The measured results were averaged over 100 runs.

to 6 bits. The measured memory requirements for both, Grover's and Ozhigov's algorithm are shown in table 3. Note that the required RAM does not seem to increase for a small number of qubits, as it is shadowed by the amount of RAM used by the python process itself.

5.4 Grover's Algorithm With a Single Target

Figure 3, 4, and 5 visualize the development of probabilities during a run of Grover's algorithm. We get the probability by squaring the amplitudes of the recorded intermediate states.

The input is the following oracle:

$$f(x) = \begin{cases} 1 & \text{if } x = 24, \\ 0 & \text{otherwise;} \end{cases}$$

Simply put, the target outcome is 24. In binary representation $0b11000$. Figure 3 visualizes the probabilities after the initial entanglement of all qubits before any Grover operation has been performed (step=0). All outcomes have the same amplitude and therefore the same probability of being returned as the result if we would perform a measurement at this step. After the first Grover operation (step=1), the probability of the target outcome is higher than the probability of any other outcome. The target outcome of $0b11000$ would only be observed

n	$\frac{\text{Grover } (k=1)}{\text{Ozhigov } (k=2)}$	$\frac{\text{Grover } (k=2)}{\text{Ozhigov } (k=2)}$
3	$\frac{2.56}{1.64} \approx 1.10\sqrt{2}$	$\frac{4.00}{1.64} \approx 1.73\sqrt{2}$
4	$\frac{4.23}{2.42} \approx 1.23\sqrt{2}$	$\frac{5.12}{2.42} \approx 1.49\sqrt{2}$
5	$\frac{6.64}{3.73} \approx 1.26\sqrt{2}$	$\frac{8.46}{3.73} \approx 1.60\sqrt{2}$
6	$\frac{9.80}{5.39} \approx 1.29\sqrt{2}$	$\frac{13.28}{5.39} \approx 1.74\sqrt{2}$

TABLE 4: Speedup-factor calculated by dividing the average number of expected queries for Grover's algorithm by that number for Ozhigov's algorithm. The result is given as a factor of the expected lower bound of $\sqrt{2}$. The values can also be seen in figure 14 and 15.

in 68.75% of cases at this point. The probability of the target outcome increases with each step until it reaches its maximum after 3 iterations of the Grover algorithm. Step number 3 is also the step closest to the expected number of steps to achieve the highest probability for the correct result $\left\lfloor \frac{\pi\sqrt{16}}{4} \right\rceil = 3$. Therefore, the results of our implementation of Grover's algorithm confirms the expected runtime of $\mathcal{O}(\sqrt{N})$.

5.5 Grover's Algorithm With Two Targets

To reconstruct the claim made by Ozhigov, we have to evaluate the behavior when the oracle has two possible solutions and we want to find both. We have two different cases: the first is one oracle with two different targets and the second is two independent oracles with each having a single target.

For example taking 8 and 24 as target outcomes, or in binary representation $0b01000$ and $0b11000$. For the first case we design our oracles to look like this:

$$f(x) = \begin{cases} 1 & \text{if } x = 8 \text{ or } x = 24, \\ 0 & \text{otherwise;} \end{cases}$$

For the second case, the two independent oracles would look like this:

$$f_x(x) = \begin{cases} 1 & \text{if } x = 8, \\ 0 & \text{if } x \neq 8; \end{cases} \quad f_y(x) = \begin{cases} 1 & \text{if } x = 24, \\ 0 & \text{if } x \neq 24; \end{cases}$$

5.6 Ozhigov's Algorithm With Two Targets

Now we are applying the algorithm described by Ozhigov to check if it offers a speedup of at least $\sqrt{2}$ over the algorithm described by Grover. The number of iterations for the highest probability of success $\frac{k}{N}$ can be calculated

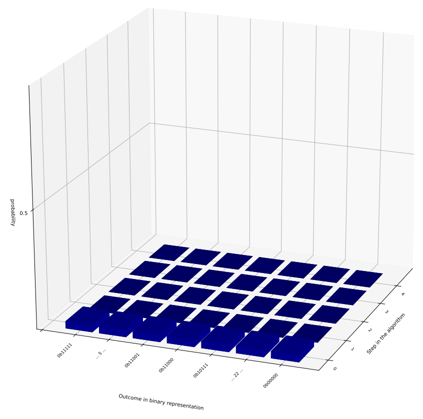

Fig. 3: The probabilities after performing the initial entanglement.

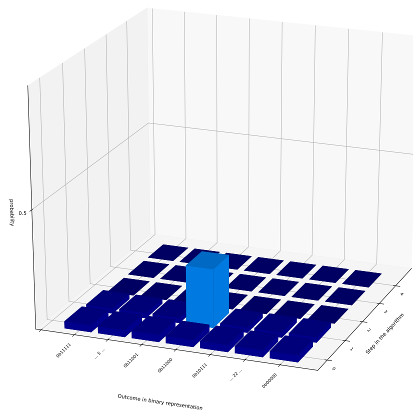

Fig. 4: The probabilities after the first operation of Grover's algorithm.

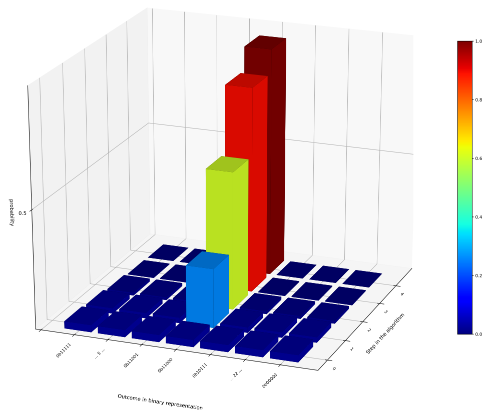

Fig. 5: The probabilities after the fourth iteration of Grover's algorithm.

Fig. 6: Probability development during Grover's algorithm finding the target $x^0 = 0b11000$. By squaring the value of the amplitude we can get the probability that a measurement performed at this step returns as the result the label at that position (from $0b00000$ to $0b11111$).

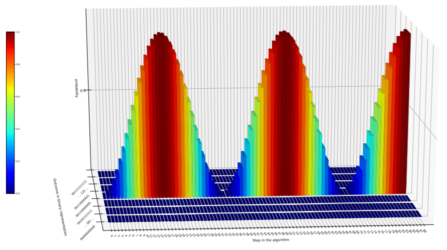

Fig. 7: Development of the probabilities.

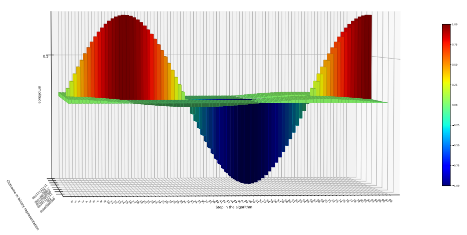

Fig. 8: Development of the amplitudes.

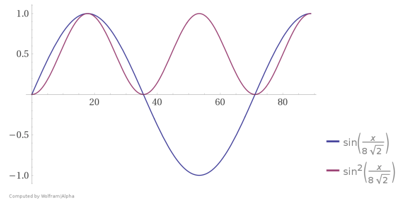

Fig. 9: sin and sin^2 on $[0, \frac{5\pi\sqrt{2^9}}{4}]$.

Fig. 10: A comparison of how the probabilities compare to the amplitudes when running Grover's algorithm for five times the optimal number of steps. The amplitudes decline after the optimal number of steps, following a curve similar to that of the sinus function. Since $probability = amplitude^2$ they never go below 0.

In figure 9 the denominatior $d = 8\sqrt{2}$ was chosen, as it is a solution for the equation $sin(x\frac{5\pi\sqrt{2^9}}{4}) = sin^2(x\frac{5\pi\sqrt{2^9}}{4}) = 1$.

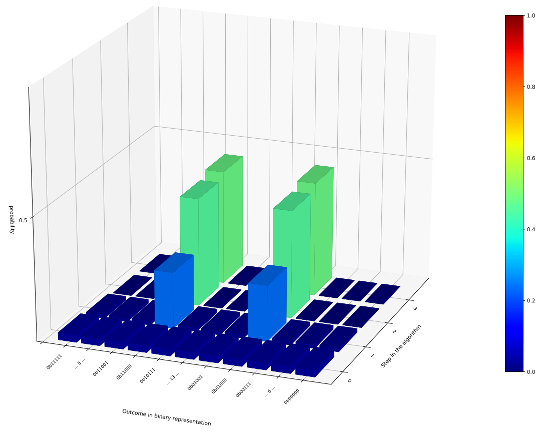

Fig. 11: The probabilities after the third iteration of Grover's algorithm with the targets $0b11000$ and $0b01000$. When performing a measurement we will only get one possible target. Therefore, we have to run the algorithm again and perform another measurement until we have measured both targets.

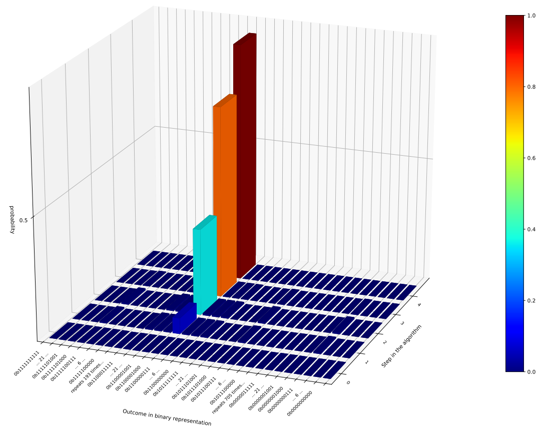

Fig. 12: The probabilities after the fourth iteration of Ozhigov's algorithm with the targets $0b11000$ and $0b01000$, embedded into a single outcome $0b1100001000$. Because we are getting both targets encoded in the result, we will not have to run the algorithm again, if the result is correct.

with $\frac{k\pi\sqrt{\frac{N}{k}}}{4}$. As an example, we will examine the case $k = 1$ and $N = 2^5 = 32$ with a perfect oracle, which always returns the same output for the same input. Namely 1 if the input is equal to x^0 and 0 if the input is not equal

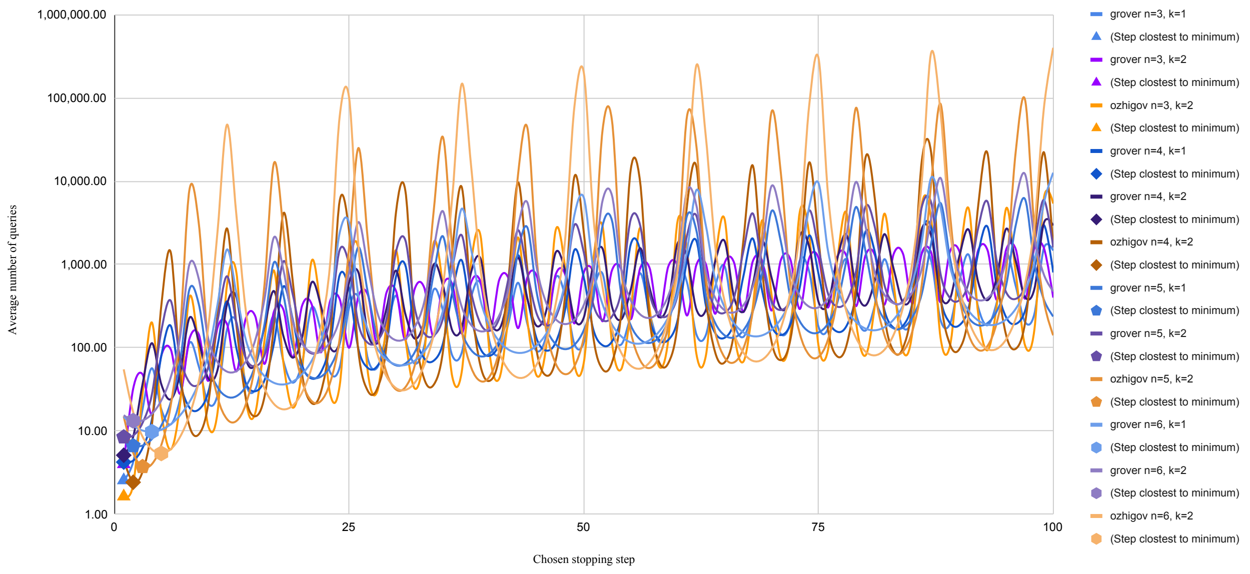

Fig. 13: Overview: steps 1–100, Grover (k=1 and k=2) vs Ozhigov (k=2) for n=3–6.

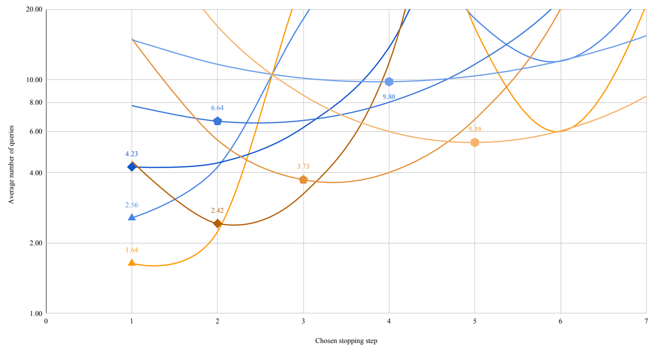

Fig. 14: Zoomed-in: steps 1–20 and Grover (k=1) vs Ozhigov.

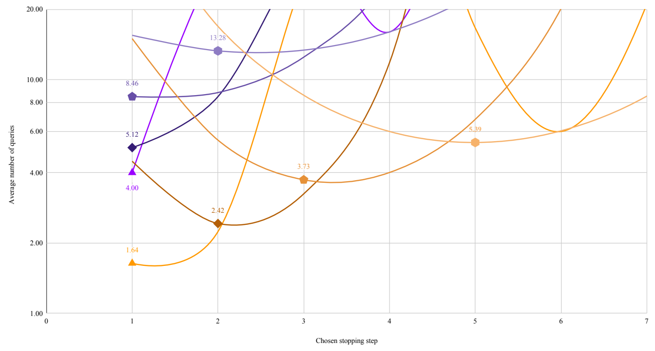

Fig. 15: Zoomed-in: steps 1–20 and Grover (k=2) vs Ozhigov.

Fig. 16: Figure 13 shows the average expected number q of total queries (x-axis), if execution of a run is stopped after l steps (y-axis). Figure 14 and 15 show a zoomed-in version of the case Grover (k=1 and k=2) separately. These values and the resulting speedup-factor are listed in table 4. The reason n=1 and n=2 have been excluded is that we need $k \ll N$, because the behavior for small values of n (and therefore small values of $N = 2^n$), is not on par with larger values. A simple example is the case of n=1 and k=2, where the whole domain is made up of two values: $0b0$ and $0b1$, of which both would be targets.

to x^0. After $\frac{\pi\sqrt{N}}{4}$ or about 4.442 iterations, the chance for error is the lowest with $\frac{1}{N} = 3.125\%$. However, in the case that we are unlucky the algorithm returns a result $x \neq x^0$. Fortunately, we can confirm whether the result is correct by giving it as input to the oracle and check if it returns the value 1. In case it returns the value 0 we know that we were unlucky and have to run the algorithm again.

Figure 5 shows that the probability for measuring the correct result is rising with every step j for $1 \leq j \leq 4$. Therefore, we can stop early, query the oracle with the returned result, and run the algorithm again if the oracle returns 0. This strategy was described by Boyer et al. [14]. We use it to calculate the optimal number of steps to get the minimum expected number of queries until success. This number, if we have to re-run the algorithm, is $k{\cdot}l\sqrt{\frac{N}{k}}$, with l being the step we chose to stop at.

After each step, we determine the amplitude of the target register by executing the algorithm in the simulator and then calculate the expected number of queries. To get the speedup Ozhigov's algorithm provides, we can determine the expected minimum number of queries for different domain sizes and calculate the speedup-factor.

Table 4 and figures 13, 14, and 15 show the results for the expected number of queries to perform, if we stop each run of the algorithm after a specific step and re-run it if the result is not a solution for the oracle. The smallest count of steps is marked for each algorithm. The resulting speedup-factors relative to the expected factor of $\sqrt{2}$ is listed in table 4.

As can be seen in figure 14 and 15, for each evaluated domain, described by 3 or more, bits we measure a speedup-factor greater than the lower bound of $\mathcal{O}(\sqrt{2})$ claimed by Ozhigov. Therefore, we confirm, that the expected speedup was achieved.

6 Related Work

In contrast to our implementation on a quantum simulator there are also implementations on real devices, which are done eg. by Jones et al. who implemented a quantum search algorithm on a nuclear magnetic resonance quantum computer [16] and by Mandviwalla et al., who have demonstrated that it is possible to execute Grover's search algorithm on the quantum computer offered by IBM [17].

Ross et al. modified Grover's algorithm for a fast database search [18], Farhi et al. investigated the square root speedup in SS-problems [19], and Hogg et al. developed a framework for structured quantum search [20].

7 Conclusion

7.1 Summary

We implemented Ozhigov's algorithm, which speeds up Grover's algorithm for the specific case of $k = 2$ by searching for both targets in parallel. We measured the running time, RAM requirements for up to 18 qubits, and determined the minimum number of queries. We calculated the speedup-factor and confirmed that Ozhigov's algorithm achieves the claimed speedup by a factor of at least $\sqrt{2}$.

7.2 Outlook and Future Work

Instead of using a quantum simulator, the next step could be to use an actual quantum computer. Several institutions offer access to their quantum systems. However, Mandviwalla et al. have observed that a high error rate is causing difficulties to execute Grover's algorithm for 4 qubits effectively [17]. Since Ozhigov's algorithm needs 3 times the number of qubits the technology will most likely need more improvements, regarding the error rate, until we can confirm the results that we observed in the simulator on a real quantum computer.

References

[1] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. ACM, pp. 212–219.

[2] Y. Ozhigov, "Speedup of Iterated Quantum Search by Parallel Performance."

[3] R. P. Feynman, "Simulating Physics with Computers," vol. 21, no. 6, pp. 467–488.

[4] D. Deutsch, "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer," vol. 400, no. 1818, pp. 97–117.

[5] D. P. DiVincenzo and Others, "The Physical Implementation of Quantum Computation."

[6] J. Kelly. A Preview of Bristlecone, Google's New Quantum Processor. [Online]. Available: https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html

[7] L. S. Bishop, S. Bravyi, A. Cross, J. M. Gambetta, and J. Smolin, "Quantum Volume."

[8] B. Villalonga, D. Lyakh, S. Boixo, H. Neven, T. S. Humble, R. Biswas, E. G. Rieffel, A. Ho, and S. Mandrà, "Establishing the Quantum Supremacy Frontier with a 281 Pflop/s Simulation."

[9] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[10] J. R. Johansson, P. D. Nation, and F. Nori, "QuTiP: An Open-Source Python Framework for the Dynamics of Open Quantum Systems," vol. 183, no. 8, pp. 1760–1772.

[11] ——, "QuTiP 2: A Python Framework for the Dynamics of Open Quantum Systems," vol. 184, no. 4, pp. 1234–1240.

[12] D. J. Bernstein, "Grover vs. McEliece," in *Post-Quantum Cryptography*, N. Sendrier, Ed. Springer Berlin Heidelberg, vol. 6061, pp. 73–80.

[13] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, "Strengths and Weaknesses of Quantum Computing," vol. 26, no. 5, pp. 1510–1523.

[14] M. Boyer, G. Brassard, P. Hoeyer, and A. Tapp, "Tight bounds on quantum searching," may 1996.

[15] E. Farhi and S. Gutmann, "Quantum Mechanical Square Root Speedup in a Structured Search Problem," nov 1997.

[16] J. A. Jones, M. Mosca, and R. H. Hansen, "Implementation of a Quantum Search Algorithm on a Nuclear Magnetic Resonance Quantum Computer," *Nature*, vol. 393, no. 6683, pp. 344–346, may 1998.

[17] A. Mandviwalla, K. Ohshiro, and B. Ji, "Implementing Grover's Algorithm on the IBM Quantum Computers," in *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*. Institute of Electrical and Electronics Engineers Inc., jan 2019, pp. 2531–2537.

[18] D. A. Ross, "A Modification of Grover's Algorithm as a Fast Database Search," jul 1998.

[19] E. Farhi and S. Gutmann, "Quantum Mechanical Square Root Speedup in a Structured Search Problem," nov 1997.

[20] T. Hogg, "A Framework for Structured Quantum Search," *Physica D: Nonlinear Phenomena*, vol. 120, no. 1-2, pp. 102–116, jan 1997.