mathematics

MDPI

Article

Functional Matrices on Quantum Computing Simulation

Hernán Indíbil de la Cruz Calvo [1,*,†], Fernando Cuartero Gómez [1,†], José Javier Paulet González [1,2,†], Mauro Mezzini [3,†] and Fernando López Pelayo [1,*,†]

[1] Departmento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Campus Universitario, 02071 Albacete, Spain; fernando.cuartero@uclm.es (F.C.G.); jose.paulet@uclm.es (J.J.P.G.)
[2] Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Plaza de las Ciencias 3, 28040 Madrid, Spain
[3] Educational Sciences Department, Roma Tre University, Via Ostiense, 00154 Rome, Italy; mauro.mezzini@uniroma3.it
* Correspondence: hernanindibil.cruz@uclm.es (H.I.d.l.C.C.); fernandol.pelayo@uclm.es (F.L.P.)
† These authors contributed equally to this work.

Abstract: In simulating Quantum Computing by using the circuit model the size of the matrices to deal with, together with the number of products and additions required to apply every quantum gate becomes a really hard computational restriction. This paper presents a data structure, called Functional Matrices, which is the most representative feature of QSimov quantum computing simulator which is also provided and tested. A comparative study of the performance of Functional Matrices with respect to the other two most commonly used matrix data structures, dense and sparse ones, is also performed and summarized within this work.

Keywords: quantum simulator; circuit model; functional matrices; computational efficiency

MSC: 81P68

check for updates

Citation: de la Cruz Calvo, H.I.; Cuartero Gómez, F.; Paulet González, J.J.; Mezzini, M.; Pelayo, F.L. Functional Matrices on Quantum Computing Simulation. *Mathematics* **2023**, *11*, 3742. https://doi.org/10.3390/math11173742

Academic Editor: Emmanuel Lorin

Received: 28 June 2023
Revised: 14 August 2023
Accepted: 22 August 2023
Published: 31 August 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Classical computing has a firmly established model, based on Boolean logic, known as Von Neumann architecture. By contrast, quantum computing can be implemented in different ways, including adiabatic quantum computing, cluster state quantum computing, or topological quantum computing, among others. However, the most firmly established model to be used as a standard is the quantum circuit model introduced by Deutsch [1], which is described as follows. First, the quantum computer is in a state described in a quantum register. Then, the state evolves by applying operations specified in an algorithm. Finally, the information about the state of the quantum register is obtained by a special operation, called measurement. Thus, a design of logic circuits operating on logic gates and acting on classical bits is extended to the design of another type of circuit that operates by means of quantum gates acting on qubits that are finally required to be measured.

Currently, many resources are needed to research and develop quantum computing. On the one hand, the number of simulable qubits able to be entangled is kind of low, as we will discuss in the following paragraphs. On the other hand, not everyone has access to a real quantum computer with both enough qubits as well as entangling options in order to test their algorithms; beyond this, they also have to check whether some qubits are allowed to be entangled within the current architecture at their disposal. As an example of a public access quantum computer, we have those provided by IBM in their Quantum Experience initiative. A large study has been completed about how to map quantum circuits to the different architectures they provide [2].

The lack of standardization on the set of gates provided by real and simulated quantum computers makes even more difficult the quantum algorithms design task. This arises

for example when comparing OpenQASM [3] and Quil [4]. Furthermore, current implementations of Quil do not include any CX gate (also known as C-NOT), but include CZ gate instead. All these features, with special attention to the small number of available qubits (ancillary qubits included), make that having as many disposable qubits as possible in order to be stored and operated becomes a key factor in quantum computing, either actual or simulated.

There is a huge variety of software tools that can be used for quantum computing simulation, almost all of them defined for different purposes. The essential facilities expected to be provided by a quantum simulator are an expressive programming language that allows the programmer to implement all possible computations, together with a good performance, making it possible to simulate a remarkable number of qubits.

Among the first simulators, ref. [5] explored a huge number of concepts about quantum computer simulation. At the time the paper was written, simulators were not able to handle a large number of qubits, i.e., they were mostly developed for educational purposes rather than for high-performance simulation.

Currently, a complete list of tools is available from Quantiki [6]. There are many kinds of simulators when classified by the data structure they use to store quantum information. The most common simulator category is the full-state vector one, usually used to simulate general-purpose quantum machines. Other kinds of simulators capable of simulating universal quantum computation are tensor networks based [7] and decision diagrams based [8] (unmaintained) [9]. Tensor network simulators require less space than the former, in exchange for fidelity. The more accurate results you need, the more memory usage. Decision diagrams have the same flaw, as they are stochastic and they will not have the same accuracy as state-vector simulators. There are also density matrix simulators, capable of doing anything full-state vector simulators are able to do, trading being able to simulate noise with being more inefficient. There are very efficient simulators capable of reproducing other quantum machines with more restrictions. These restrictions range from permitting a highly restricted set of gates to even blocking entanglement. We will focus on full-state vector simulators. Their common characteristic, in essence, is that the amount of memory needed to store a register of n entangled qubits corresponding to 2^n complex values. Besides, simulating the effect of applying a quantum gate to an n qubits system requires operating a matrix that represents the operation by the state vector in column form. This means that, in the worst-case scenario, we will need a matrix of size $2^n \times 2^n$. Those reasons, especially the latter, heavily bound the number of qubits that can be handled. Let us notice that one of the quantum computer simulators most widely used in education environments, mainly due to its easy-to-use drag-and-drop interface, Quirk [10], suffers from this problem.

Several techniques have been followed in order to implement simulators with as many entangled qubits as possible, and support as many operations as possible. Among them, we can mention the use of sparse matrices, as [11] does in the QX simulator, or decomposing a quantum circuit into smaller circuits to execute distributed calculations, either in a parallel architecture [12], or sequentially on the same platform [13]. Research has been completed on how to efficiently simulate quantum circuits, yet it is still inefficient without losing universality when it only contains a subset of operations, such as not allowing controlled gates (storing each qubit in a separate state vector or density matrix) or with stabilizer circuits [14]. Our focus is on simulating universal quantum circuits, and therefore, we are not able to use the same improvements.

QX simulator [11] can be taken as an example of one of the currently most efficient tools for this purpose. QX needs almost 270 GB of memory just to store a 34 fully entangled qubits registry. QSimov requires a similar, but slightly lower, storage capacity than QX; nevertheless, at applying some quantum gates, QX pays an substantially more expensive computational price, especially when dealing with matrices with a low number of zeros (referred to as a low degree of sparsity). This scenario is completely independent of QSimov memory requirements.

Another way to cope with the requirements of the gate application operation is to avoid calculating the referred big matrix. This approach is used by some simulators such as Intel Quantum Simulator [15], formerly known as QHiPSTER [16]. Instead of calculating and storing the whole matrix, just the elements of the full-state vector that will be modified are calculated and operated accordingly. This approach is widely used by lots of currently available quantum computing simulators such as QuEST [17] (developed at the University of Oxford) and Qiskit [18] (developed by IBM), both of them offering a function for each gate they offer.

We provide the quantum computing research community with a quantum computer simulator named QSimov, whose main purposes are, on the one hand, being able to deal with a big number of qubits and, on the other hand, providing a set of basic gates as well as an easy way to build more complex ones.

The simulator uses the circuit model, similar to many other simulators [19] or even real quantum computers. This model was chosen because a lot of well-known algorithms, such as Deutsch's algorithm or Shor's, are described using it. QSimov handles the problem of both memory allocating and operating by using functional matrices instead of the most common option conformed by either dense matrices with some sort of parallelization or sparse matrices, such as some of the aforementioned simulators. These functional matrices are a data structure that will be presented and described within this paper.

A comparison between the functional matrices implemented in QSimov versus the other two implementations, i.e., NumPy (dense matrices) and SciPy (sparse matrices) has been performed.

The paper is structured as follows, Section 2 presents the data structure over which QSimov operates. Section 3 describes the experimental setup we have implemented with the aim of checking empirically our intuition about the advantages of our proposal. Section 4 summarizes the results of the experiments comparing our simulator running over the most commonly used data structure for these purposes vs. our proposal of functional matrices. Section 5 ends the paper.

2. Functional Matrices

The size of the matrices representing gates for Quantum computing simulation, using the circuit model, is a crucial issue to be considered as the state vector of these systems grows at a rate of 2^n, where n = *number of qubits in the system*; therefore, the matrix representing any gate grows at a rate of 2^{2n}, which becomes a storage problem.

In order to cope with this, we have defined a data structure, inspired by both functional programming features and lazy evaluation, the characteristics of which have been cooperatively used with the aim of overcoming this limitation.

The data structure functional matrix provides a function $f(i, j, r, c, argv)$ such that, for a given $r \cdot c-$ sized bi-dimensional matrix A, we obtain the $A_{i,j}$ element within the matrix A, where i is the row and j is the column of this element. Parameters r and c stand for the total number of rows and columns in A, respectively. $argv$ is a pointer to a tentative set that could be required in some cases in order to compute $A_{i,j}$. The functional matrix data structure lets you handle a matrix with f, r, c, and $argv$ (and therefore, matrices cannot be reshaped, and $argv$ must be constant).

These functional matrices domain supports the following operations over matrices:

- Addition;
- Subtraction;
- Product/multiplication;
- Entity-wise multiplication;
- Transposition;
- Hermitian transposition;
- Kronecker product;
- Trace;
- Partial trace (a generalization of trace function).

Most of these operations are used in quantum computing. According to [20], any quantum gate (and its extension to a quantum circuit) can be written in terms of universal quantum gates that in the end are all either unitary or unitary controlled. They are going to be considered as base elements for the comparison.

The way of operating functional matrices is somehow lazy (by demand), i.e., it only computes an element of the corresponding functional matrix when it is required for any further operation. Using a functional matrix instead of an ordinary matrix (made of arrays) has some advantages as well as some disadvantages regarding space and time complexities. In a nutshell, we are interested in both space (size in memory to store and handle elements) and time complexities (time to get an element)

Size in memory:

- Array matrices: $r \cdot c$, where $r = \#(rows)$ and $c = \#(columns)$;
- Functional matrices: The size of the function + six integer numbers (corresponding to the number of rows, number of columns, code for a previous operation, Boolean for transpose matrix, Boolean for conjugate, Boolean for whether previous operation) + one complex number + three pointers (to $argv$ and to the matrices just in case of an existing previous operation).

Time to get an element:

- Array matrices: Time to read it from memory;
- Functional matrices: Depends on the time complexity of the function f.

Bearing this in mind, functional matrices should be used instead of an array matrix provided that you are able to define such function f in a computationally efficient fashion.

Neither when the space needed to compute f is bigger than the one needed by the array matrix $r \cdot c$-sized nor when the time complexity of f is unaffordable is it worth using functional matrices data structure.

3. Quantum Simulating in Practice

The last version of QSimov is currently available at https://github.com/Mowstyl/QSimov/ (accessed on 1 June 2023) and in PyPI (Python Package Index) under the name "QSimov-Mowstyl".

Three Python functions have been defined in this demo of execution on QSimov.

The first one returns the circuit defined by the Deutsch–Jozsa algorithm. It requires two parameters: size (number of qubits affected by the oracle) and U_f (oracle associated with f).

```
def DJAlgCircuit(size, U_f):
"""Return Deutsch-Josza algorithm circuit.

$U_f$ is the oracle, having x1..xn and y as
input qubits. x1..xn and y take only values
either 0 or 1.
Once applied it returns x1..xn and y',
where y' = f(x1..xn) XOR y.
Size is n + 1, where n is the number of
input bits of f."""

# The last qubit is defined as an ancilla
# qubit formerly set to 1
c = qj.QCircuit("Deutsch-Josza Algorithm",
ancilla=[1])

# We apply a Hadamard gate to all the qubits
c.add_line(*["H" for i in range(size)])
c.add_line(U_f)  # We apply the oracle

# We apply a Hadamard gate to all qubits but
# the last one
c.add_line(*("H" for i in range(size-1)), "I")

# We measure x1..xn qubits.
# If all of them are 0, $f$ is constant,
# otherwise $f$ is balanced.

c.add_line(qj.Measure([1 for i in range(size - 1)]
+ [0]))

return c
```

The second function returns U_f oracle we are going to test. The function used by the oracle is defined as $f(x) = x \geq 0$, a balanced function where x is an integer number encoded by $size - 1$ bits in which two's complement stands for negative numbers. Therefore, we only need to test the most significant bit of x. This can be achieved by applying a NOT gate to the output qubit y anti-controlled by the most significant qubit.

```
def geq_zero(size):
"""DJ Oracle for f(x) = x >= 0. Balanced"""
gate = qj.QGate("U_(x>=0)")

gate.add_line(*[None for i in range(size-1)],
["X", None, [size-2]])

return gate
```

The last function defined is the main one whose input parameter is expected to be the total number of qubits we are going to use. In order to work with a worst-case scenario, we disable optimizations other than functional matrices (this circuit could be simulated with only two entangled qubits). This means all qubits will be ready to be entangled, with a state of 2^{size} complex numbers. After the execution of the circuit, the output (whether f is balanced or constant) and the running time (measured in seconds) taken by the algorithm, will be printed.

```
def main():
"""Execute DJ with specified qubits."""
argv = sys.argv[1:]
# We only have one argument
if 1 != len(argv) or int(argv[0]) < 2:
print("Syntax: " + sys.argv[0] +
" <number of qubits (min 2)>")
return
# The number of qubits (x1..xn, y)
nq = int(argv[0])
gate = geq_zero(nq)  # The U_f oracle
# The Deutsch-Jozsa algorithm circuit
circuit = DJAlgCircuit(nq, gate)
# We specify useSystem = False to disable
# optimizations. That means we will use a
# state vector of size 2^n even if we do
# not need it, making the simulation a
# lot slower than it should be.
init = t.time()
_, mes = circuit.execute([0
for i in range(nq - 1)],
args={"useSystem": False}
)
end = t.time()
mes = mes[0]
print("Is balanced?:", any(mes[:-1]))
print("Elapsed time:", end - init, "s")
```

For this example, the maximum number of entanglement-ready qubits supported by QSimov v3.1.2 has been used, i.e., 30 so far.

As all optimizations have been disabled for this simulation, it has used more resources than it could for a function that only checks the sign bit of a binary number. This way, we will be able to see how the entanglement impacts the resources needed for a simulation.

The result of the execution was *f is balanced*, as expected. The whole execution of the example took 1 h plus 4.159 s, 3604.159 s. In total, 16 GB of memory was used to store the state as an array of 2^n complex double numbers. Another 16 GB was used in creating another state array to store the result (applying gate operation), for a peak of 32 GB.

4. A Comparative Study

Let us start by describing the platforms supporting the experiments as well as the tasks to be performed.

4.1. Experimental Setups

For the sake of obtaining the most valuable information with the computing architectures at our disposal, we have performed the same series of experiments over the following two platforms, as they are considered quite representative:

- AMD PC:
  - Number of nodes: 1;
  - Processor: 1 × AMD Ryzen 7 3700X, 8 Cores, 3600 MHz;
  - RAM: 2 × 8 GB, DDR4, 3600 MHz;
  - SO: Windows 10 Education 64-bit.
- HP GALGO:
  - Number of nodes: 25;
  - Processor: 2 × Intel Xeon E5-2650, 8 Cores, 2000 MHz;

- RAM: 16 × 4 GB, DDR2, 667 MHz;
- SO: CentOS 6.x 64-bit.

Two experiments have been carried out over each of these platforms in order to empirically test the benefits of functional matrices vs. both dense and sparse matrices as representative of the most commonly used types of matrices for this purpose nowadays.

Dense matrices will be stored using NumPy's multidimensional arrays and operated using NumPy (which relies on BLAS and LAPACK for linear algebra operations).

Sparse matrices will be stored by those with the same name within SciPy's Dictionary of Keys. It has been chosen in order to obtain an easy way both to build and to access them. Finally, SciPy will make them become another type of matrix during its operation.

From now on, the lines representing the performance of the three different domains of matrices at comparison are depicted in the following way:

- Solid lines are used for dense matrices;
- Dashed lines distinguish sparse matrices;
- Functional matrices are identified by dotted lines.

At quantum computing simulations, applying a n qubits gate to a n qubits registry requires allocating in memory 2^{2n} times the size required for a single qubit, and this amount is far bigger than the 2^n qubits required to store the registry itself. Therefore, in the end, quantum gates application becomes the weak point when simulating a quantum computer (provided that we are concerned neither with decoherence nor with error issues)

In applying a m qubits gate, $m < n$, in order to be actually capable of multiplying the gate matrix by the key of the registry, we have to use the Kronecker product involving that gate and identity matrices. The outcome matrix' size is 2^{2n} again.

This is the formula that we implement in order to compute an element of the matrix resulting from the tensor product of two matrices A and B:

$$A \in \mathbb{C}^{r_A \times c_A}$$
$$B \in \mathbb{C}^{r_B \times c_B}$$
$$(A \otimes B)_{i,j} = A_{\lfloor i/r_B \rfloor, \lfloor j/c_B \rfloor} \cdot B_{i \bmod r_B, j \bmod c_B}$$

The core function f provided by functional matrix structure is remarkably appropriate in this case since they allow handling quantum gates just paying constant computational complexity in space, thus avoiding the space bottleneck. To be honest, there is a very slight increase (linearly increasing) in the time needed to get (access/read) an element of the matrix.

When working with functional matrices, it is best to avoid multiplying all the gate matrices and afterwards apply the resultant matrix to the registry. Instead, QSimov operates quantum gates one by one so avoiding the elements to be stored, which would generate a lot of computations being repeated when getting the elements of the outcome matrix (the number of computations performed more than once depends on the matrix multiplication algorithm implemented, which in the naïve case belongs to $O(n^3)$ time complexity).

This formula is the formula that we implement in order to calculate an element of the matrix resulting from the product of two given matrices A and B:

$$A \in \mathbb{C}^{r_A \times c_A}$$
$$B \in \mathbb{C}^{r_B \times c_B}$$
$$AB_{i,j} = \sum_{k=0}^{c_A - 1} A_{i,k} \cdot B_{k,j}$$

Apart from this improvement regarding space issues, we must keep in mind that it is still needed to multiply a $2^n \times 2^n$ matrix by a 2^n column vector. In a nutshell, we moved the bottleneck of the simulation problem from space to time.

4.2. Core Experiment: Quantum Gates

As quantum gate application is assumed to be the main problem in quantum computing simulations, we focus this first experiment in comparing our proposal when running over a representative set of quantum gates.

To begin with, we have considered a Hadamard gate for n qubits, as it is maybe the most commonly used. Afterwards, we have taken a composite gate composed of other quantum gates randomly chosen from a wide set of them, so trying to sample all the rest:

- 3 qubits: $[R_z(\frac{\pi}{3}), R_y(\frac{\pi}{3}), H]$;
- 4 qubits: $[CX, R_x(\frac{\pi}{4}), R_x(\frac{\pi}{4})]$;
- 5 qubits: $[R_x(\frac{\pi}{5}), CY, R_y(\frac{\pi}{5}), R_z(\frac{\pi}{5})]$;
- 6 qubits: $[CZ, R_y(\frac{\pi}{6}), CZ, R_x(\frac{\pi}{6})]$;
- 7 qubits: $[CZ, R_z(\frac{\pi}{7}), R_x(\frac{\pi}{7}), CZ, R_z(\frac{\pi}{7})]$;
- 8 qubits: $[CY, CZ, R_y(\frac{\pi}{8}), R_x(\frac{\pi}{8}), H, R_y(\frac{\pi}{8})]$;
- 9 qubits: $[CZ, CZ, CY, R_y(\frac{\pi}{9}), CX]$;
- 10 qubits: $[CX, R_y(\frac{\pi}{10}), R_z(\frac{\pi}{10}), CY, R_x(\frac{\pi}{10}), H, R_x(\frac{\pi}{10}), R_z(\frac{\pi}{10})]$;
- 11 qubits: $[R_z(\frac{\pi}{11}), R_y(\frac{\pi}{11}), R_x(\frac{\pi}{11}), R_y(\frac{\pi}{11}), CY, R_x(\frac{\pi}{11}), CY, R_y(\frac{\pi}{11}), R_z(\frac{\pi}{11})]$;
- 12 qubits: $[CZ, R_y(\frac{\pi}{12}), CZ, CY, R_y(\frac{\pi}{12}), CX, CY]$;
- 13 qubits: $[R_z(\frac{\pi}{13}), R_y(\frac{\pi}{13}), R_x(\frac{\pi}{13}), R_y(\frac{\pi}{13}), CY, R_x(\frac{\pi}{13}), CY, R_y(\frac{\pi}{13}), CY, R_z(\frac{\pi}{13})]$.

These are the detailed set of gates used for each number of qubits in the second case. The corresponding matrices for these gates are shown in Appendix A.

The memory usage (how many MBs have been allocated to store the data structure) figures summarizing a number of experiments are captured in Figures 1 and 2.

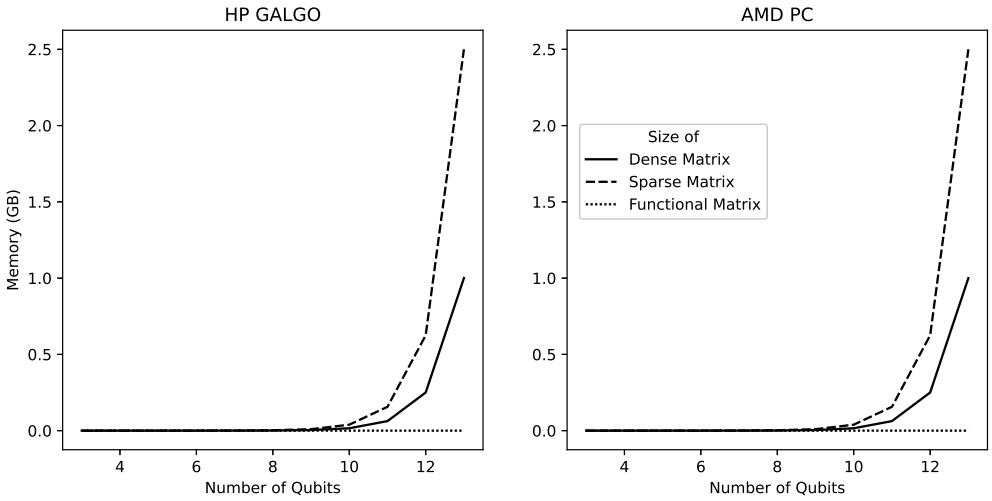

Figure 1. Memory usage with Hadamard gate from 3 to 13 qubits.

As expected, functional matrices are far better than the other two options, since they only need to use memory on demand so nothing is required to be stored. Nevertheless, a drawback appears when elements have to be accessed. Even in this case, only dense matrices improve functional ones. We should bear in mind that time is not the main problem when simulating quantum.

The access time (average number of seconds needed to access all the elements of the matrix divided by the number of elements in the matrix) has been computed and depicted for both methods in Figures 3 and 4.

For composite gates the time needed to "create" the gate, i.e., computing the Kronecker product of one or two qubit gates to obtain the corresponding matrix, has also been recorded; see Figure 5. Again, here, the advantage when creating some representative gates appears by the side of functional matrices. Moreover, it seems that as the number of qubits grows this advantage increases too.

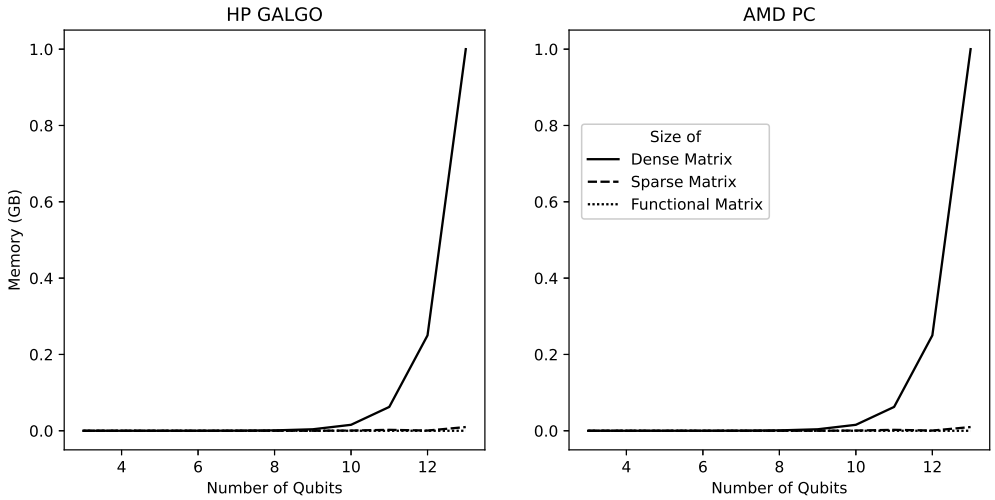

Figure 2. Memory usage with random gates from 3 to 13 qubits.

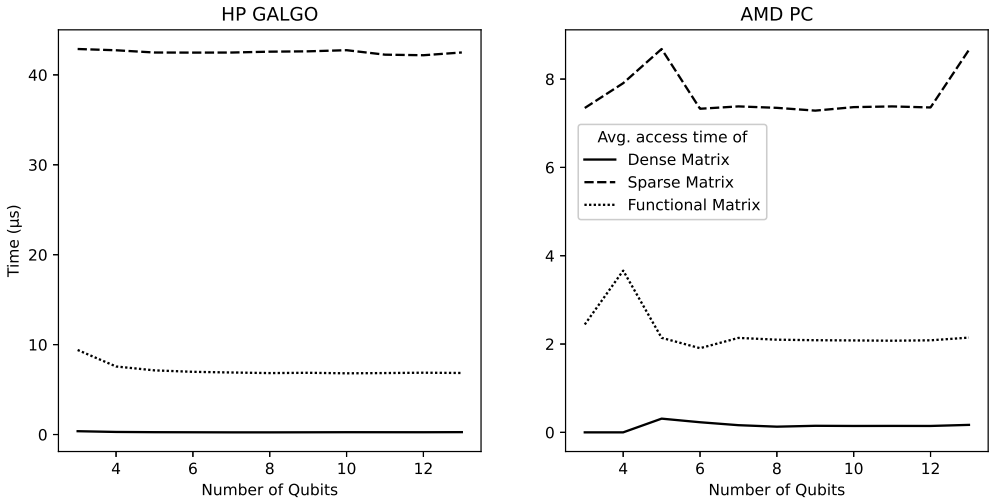

Figure 3. Average time to access an element of a Hadamard gate from 3 to 13 qubits.

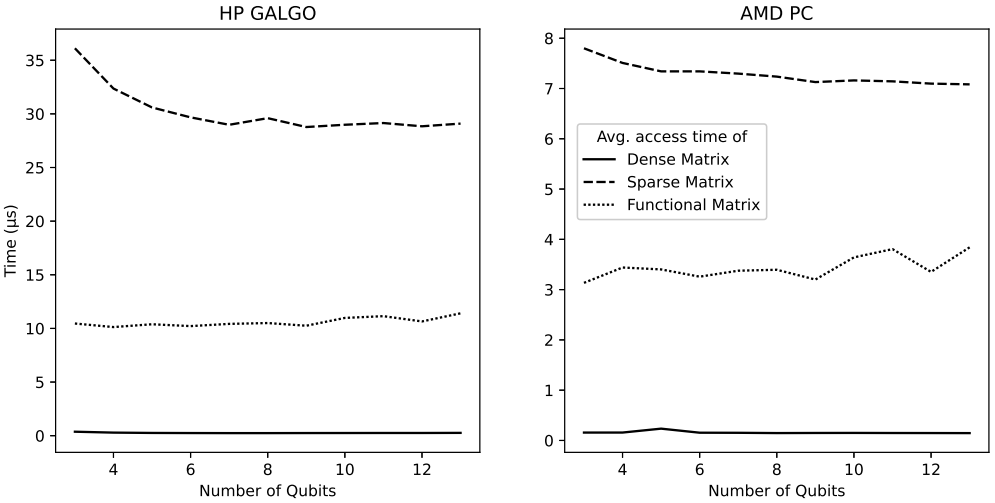

Figure 4. Average time to access an element of a random quantum gate from 3 to 13 qubits.

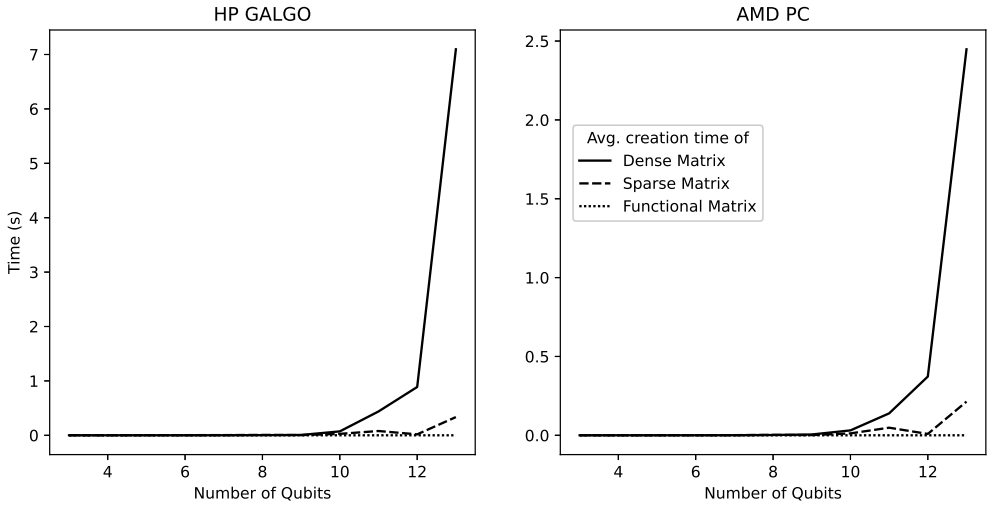

Figure 5. Average time to create a random quantum gate from 3 to 13 qubits.

4.3. Second Experiment: Concurrence

The concurrence of a quantum system, as defined in [21], is a value widely used in order to measure the degree of entanglement of a bipartite system. In [22], this concept was extended to multipartite systems. Finally, ref. [23] did the same for arbitrary dimensional bipartite systems. In our case, we have computed only the concurrence of the system after removing the first qubit from it.

In order to get this value, we require:

1. Computing the density matrix of the system ρ;
2. Tracing that first qubit, obtaining a reduced density matrix $\rho_{Q-\{q_0\}}$, the so-called partial trace computation;
3. Computing its square $\rho^2_{Q-\{q_0\}}$.

The trace of that matrix is the last step before performing some more calculations to finally obtain this formula:

$$C(\psi) = \sqrt{2(1 - Tr(\rho^2_{Q-\{q_0\}}))}$$

where

- Q is the set of qubits in the quantum system;
- ψ is the vector of state amplitudes of the system;
- ρ is the density matrix defined as $\rho = |\psi\rangle \cdot \langle\psi|$;
- $\rho_{Q-\{q_0\}}$ is the reduced density matrix after tracing out the qubit q_0 from the system.

More precisely, in order to compute the concurrence of the density matrix of the system ψ, as well as, in the rest of the tasks to be performed for the sake of testing *functional matrices* with the most common options, we have considered the following scenarios to be compared:

- By using dense matrices, we would need to calculate $|\psi\rangle \cdot \langle\psi|$, a matrix of 2^{2n} complex numbers, so obtaining the reduced density matrix of size $2^{2(n-1)}$ by tracing out a qubit from the system, calculating its square, calculating the trace of that matrix, and finally operating with that value;
- By using sparse matrices using them the same way we have used dense matrices;
- By using functional matrices with such function f that uses the registry as its *argv* argument, then you can get the elements you need of the density matrix without storing them, i.e., using them just to compute the elements of the diagonal and adding them one by one to calculate the trace.

In order to study the concurrence of systems of entangled qubits, the following quantum processes are executed:

1. Setting all qubits to 0;
2. Hadamard gate is applied over the first qubit;
3. For i ranging from 0 to $n-1$ (n is the number of qubits of the system), a rotation of $\frac{\pi}{i+1}$ rads around the X axis controlled by qubit i is applied over qubit $i+1$.

For each kind of matrix (dense, sparse, and functional), both the maximum memory requirements, Figure 6 and the time needed to compute it have been recorded and depicted in Figure 7.

In the case of memory requirements, both sparse and functional matrices are the best options with identical results.

When measuring the time required to compute concurrency, both dense and functional matrices are the best options.

Furthermore, in this second experiment, the figures indicate that functional matrices perform as well as the best in both measures.

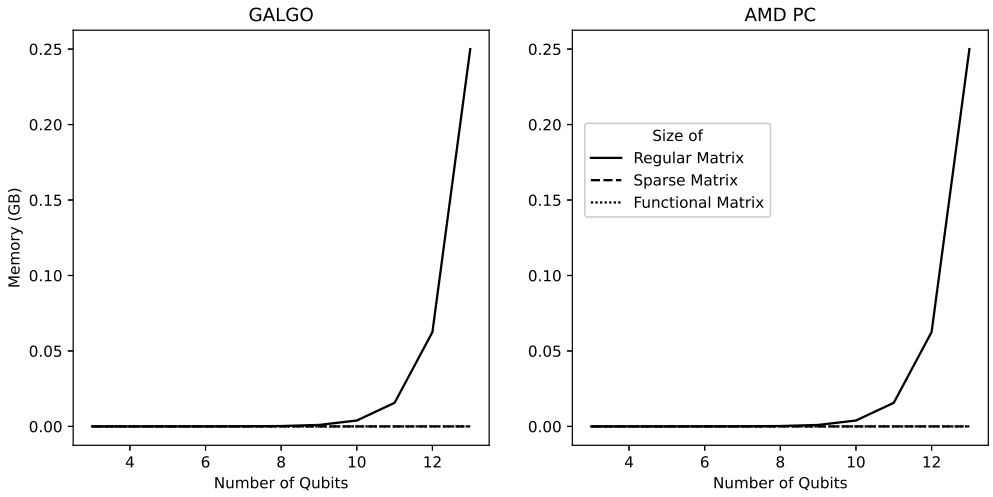

Figure 6. Maximum memory needed to calculate the concurrence of a system from 3 to 13 qubits.

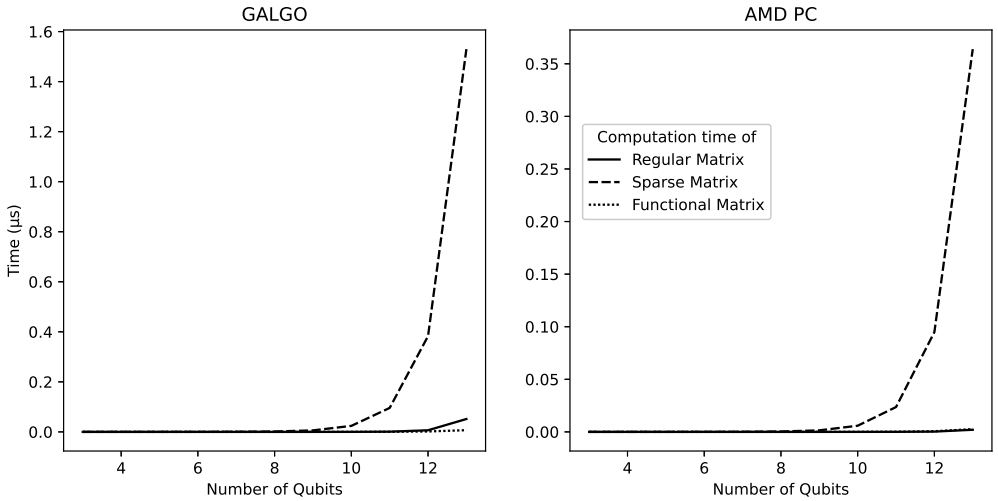

Figure 7. Average time needed to calculate the concurrence of a system from 3 to 13 qubits.

4.4. Third Experiment: Algorithms

For the last experiment, we developed a quantum computer simulator that only makes use of functional matrices. With it, three well-known algorithms have been executed and several graphs showing the memory usage and the time needed have been drawn.

This experiment has been executed using the AMD PC described in Section 4.1. All circuits below have been implemented for both functional matrix and full state vector (dense) simulation. In the first experiment, we compared execution time and memory usage when multiplying a big matrix representing the operation by the state column vector. For this instance, another, more competitive approach has been used, which is to never calculate said matrix. This way of simulating quantum computing is the one used by some Qiskit Aer backends, i.e., QuEST and QSimov by default, since it does not suffer from the bottleneck of having a $2^n \times 2^n$ matrix while still having to store a state vector of size 2^n. The simulation with full state vectors has been parallelized using OpenMP, with 16 threads and shared memory. The execution with functional matrixes has been parallelized using MPI, with 16 nodes to make use of all logic threads available in said setup.

An experiment does not end until all the nodes have sent their results to the root node and it has answered with the result of the measurement. Furthermore, all nodes must have the same data available, sharing the whole application code. That is why the time and the memory usage shown in both graphs is the one the root node (id = 0) returned.

It is important to note that Hadamard gates applied in parallel are not being applied either one by one or by calculating their tensor product. That is because a functional matrix representing $H^{\otimes n}$ has been defined.

$$f(i,j,\#(rows)) = \frac{\sqrt{\#(rows)}}{\#(rows)} \cdot g(i,j,\#(rows))$$

$$g(i,j,r) = \begin{cases} (-1)^{i\&j}, & \text{if } r = 2 \\ (-1)^{(i \geq r/2)\&(j \geq r/2)} \cdot g(i \bmod (r/2), j \bmod (r/2), r/2), & \text{otherwise} \end{cases}$$

f is the function that returns an element of $H^{\otimes n}$ matrix, for any positive value of $n =$ number of qubits. It can be proven that $f(i,j,2^n) = H^{\otimes n}[i,j]$. By using this functional matrix, any number of parallel Hadamard gates will take the same amount of memory and will perform better than calculating the Kronecker product of 2×2 Hadamard gates. & is meant to be the bitwise "and" operation.

4.4.1. Deutsch–Jozsa Algorithm

Let f be a Boolean function that is guaranteed to be constant or balanced:

$$f : \{0,1\}^n \to \{0,1\}$$

Let U_f be a quantum oracle for $n+1$ qubits. The first n qubits are treated as the inputs parameters $\{x_0, \ldots, x_{n-1}\}$ for the f function, and the last qubit y will be used to do the output, with $y \leftarrow y \oplus f(x_0, \ldots, x_{n-1})$. When working with classical values, all the input qubits will be left as they were before applying the gate, only the output y will have been modified according to the result of evaluating f.

We have used the quantum circuit implementing the Deutsch–Jozsa algorithm for these experiments, which can be seen in Figure 8. If f is constant, the algorithm guarantees that all measurements will be zero. Otherwise, at least one of them will be one.

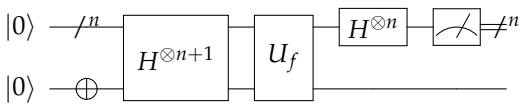

Figure 8. Deutsch–Jozsa algorithm implementation.

During each iteration of the experiment, a function has been picked between two: a constant and a balanced one:

- Constant function: $f(x_0, \ldots, x_{n-1}) = 1$;
- Balanced function: $f(x_0, \ldots, x_{n-1}) = x_{n-1}$.

4.4.2. Bernstein–Vazirani Algorithm

Let s be a secret string of bits of size n.

Let f be a Boolean function $f : \{0,1\}^n \to \{0,1\}$ defined as follows:

$$f(x_0, \ldots, x_{n-1}) = \bigoplus_{i=0}^{n-1} x_i \wedge s_i$$

Let U_f be a quantum oracle for $n+1$ qubits with the same definition as that in the Deutsch–Jozsa algorithm.

The quantum circuit that implements Bernstein–Vazirani algorithm is the same that implements Deutsch–Jozsa, the only change is the way we interpret the measurements. This time, the measurement of qubit x_i will give us the bit s_i of the secret string. It can be seen in Figure 8.

During each iteration of the experiment, a random s bit string has been generated. The oracle is generated by adding a C-NOT gate targeting y qubit and controlled by the x_i qubit if and only if $s_i = 1$, for each input qubit. A 4-qubit example can be seen in Figure 9.

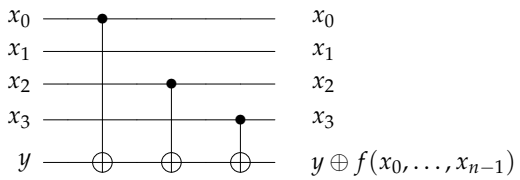

Figure 9. Bernstein–Vazirani U_f oracle with $s = 1101$.

4.4.3. Grover's Algorithm

Let f be a function $f : \mathbb{N}_{<M} \to \{0,1\}$. If and only if $f(x) = 1$, then x is a solution. There are no restrictions regarding the number of solutions, ranging from 0 to M.

Let $m = \lceil \log_2(M) \rceil$ be the minimum number of bits needed to represent any possible input.

Let U_ω be a quantum oracle for at least $m+1$ qubits. The first m qubits encode the input, being the binary representation of x. The next qubit q_m will be used to store $q_m \oplus f(x)$. The rest of the qubits, if any, will be ancillary qubits needed to calculate f. Their number n depends on the way f has been implemented. This way of arranging the qubits may vary between implementations, sometimes using the last qubit as the output one, and the ones between it and the input qubits as the ancillary qubits. There are no reasons to prefer one to the other, neither of them affecting the result.

Let IAM be the quantum gate that implements the "Inversion About the Mean" operator. Its circuit can be seen in Figure 10. This operator is applied once at the end of each Grover iteration. For the sake of efficiency, the IAM used at the last Grover iteration has one less gate, since it does not affect the results of the measurements that will be performed just after its application. It just affects the phase, which will be lost regardless. The circuit of this IAM' gate is the one in Figure 11. This gate cannot be used in any other iteration, for this phase change will affect the result otherwise.

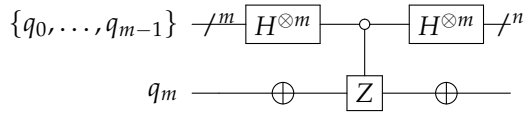

Figure 10. Inversion of the mean gate.

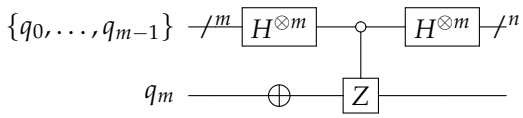

Figure 11. Inversion of the mean gate only used in the last iteration.

The quantum circuit implementing Grover's algorithm that we have used for these experiments can be seen in Figure 12. The squared section may have to be repeated $g - 1$ times, with g being the number of Grover iterations needed in this problem instance. If the right number of iterations are executed (depending on the number of solutions), there is a high chance that what we have measured is a solution. Any variation in the number of iterations or leaving garbage values in ancillary qubits can greatly decrease the odds of finding the answer. To obtain all possible solutions for the function f, Grover's algorithm may need to be executed multiple times. It is not exact like the previous two algorithms; it is probabilistic.

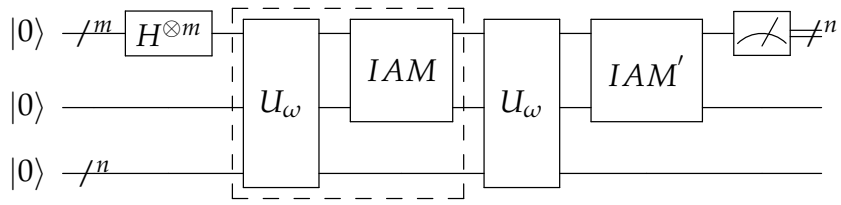

Figure 12. Grover's algorithm implementation.

A different function has been defined for the specific number of qubits of each experiment, from 2 to 10. This number of qubits is not $m + 1$, but $m + n + 1$. Since we are trying to measure how the simulator performs, it would be unfair to make it depend on how well did we define the U_ω oracle. That is the reasoning behind this decision, to take into account the number of ancillary qubits. The logic functions we have defined are as follows:

1. $f_2(a) = \neg a$;
2. $f_3(a, b) = a \vee \neg b$;
3. $f_4(a, b, c) = (a \wedge \neg b) \veebar (\neg a \wedge c)$;
4. $f_5(a, b, c) = (a \wedge \neg b) \vee \neg c$;
5. $f_6(a, b, c, d) = (a \wedge \neg b \wedge c) \vee d$;
6. $f_7(a, b, c) = (a \wedge \neg b) \vee (a \wedge \neg c) \vee (b \wedge \neg c)$;
7. $f_8(a, b, c, d) = (a \wedge \neg c) \vee (d \wedge \neg c) \vee (a \wedge \neg b)$;
8. $f_9(a, b, c, d) = ((a \vee \neg c) \wedge (d \vee \neg c)) \vee (b \wedge a)$;
9. $f_{10}(a, b, c, d, e) = ((a \wedge \neg b \wedge c) \vee \neg(a \wedge d \wedge e) \vee \neg(b \veebar c) \vee \neg(b \vee e)) \wedge c$.

The circuits that implement these functions can be found in Appendix B. After calculating the number of Grover iterations, we will only need one per problem instance.

4.4.4. Results

Memory usage goes from exponential with dense vectors to linear with functional vectors, as shown in Figure 13. What has been measured is the peak memory used. The exponential nature of the problem is not avoided by the first approach, having to store 2^n complex numbers in memory. Meanwhile, when using functional matrices, the linear growth is due to the linearly increasing number of operations performed.

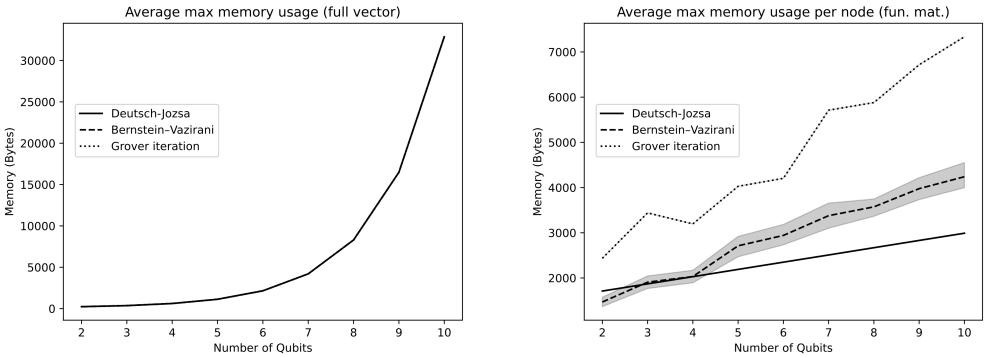

Figure 13. Algorithm average max memory usage. Dense vector vs. functional vector.

The lowest growth is that of the Deutsch–Jozsa algorithm execution, since each qubit only adds a measurement, that can be decomposed into multiplying by a projection matrix and by a scalar. We do not take into account the two extra Hadamard gates because of our $H^{\otimes n}$ functional matrix. The number of gates applied in the Bernstein–Vazirani algorithm ranges from 0 to n. On average, increasing the number of qubits by one will increase the number of operations by two, doubling the growth rate of the Deutsch–Jozsa algorithm execution. In spite of this, it is still linear. Finally, for Grover's algorithm, the number of gates depends on the function to be implemented. Since we have tried to use the least possible number of qubits for each logic function, that means that adding one qubit increases the complexity of the oracle we have built. That is why it shows the highest memory usage of all algorithms.

Take into account that the memory specified in the rightmost graph will be used up on every computation node. It is not the sum of the memory used on each one. We have decided to show this because usually in supercomputing, you may have distributed memory instead of shared memory, so you are more interested in how much memory a node needs at most. To get the total memory used, multiply it by 16. It would still be linear.

As for the execution time in Figure 14, it is shown to be growing exponentially as expected in both charts. The Bernstein–Vazirani algorithm is shown to be the slowest because of how we implemented the gate application functional matrix. It greatly benefits from the number of controls or anti-controls. Bernstein–Vazirani's oracles are composed of gates with only one control, while Grover's have a greater number of them.

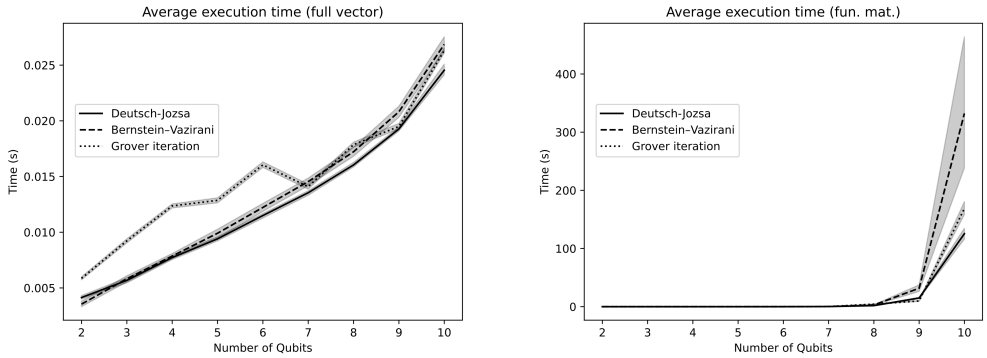

Figure 14. Algorithm average execution time using. Dense vector vs. functional vector.

The implementation for both the functional matrices and the full state vector simulator is available at Doki repository, QSimov's simulation core, written in C. https://github.com/Mowstyl/Doki/ (accessed on 1 June 2023).

The time needed to run the simulations is clearly lower when using full vectors rather than using functional ones. This is due to the fact that our functional matrix implementation repeats a lot of calculations, and shows that there is still room for improvement. Despite this difference, both approaches still have exponential asymptotic complexities.

These results show that algorithms that run over a universal quantum computer simulator can be executed with a very little amount of memory (with the same growth rate as the number of gates in the algorithm). They can also be easily parallelized to reduce their execution time with just a few messages between nodes (number of nodes + 2 or 3 messages per measurement). While functional matrices have been able to beat any other approach in terms of memory usage, lots of improvements and optimizations have to be made in order to be competitive in terms of execution time.

5. Conclusions and Further Work

Working with big-size matrices is a very common scenario when performing Quantum Computing Simulations. In the field of simulation using the circuit model, applying a gate is translated into a matrix product: the gate is represented by a matrix and the state of the system is represented by a column vector. Both of them grow exponentially which, therefore, drastically decreases the efficiency of the simulation.

In this paper, we make a proposal that reduces both the time and memory consumption of the stated problem using functional matrices that have been proven to outperform ordinary matrices to achieve a linear space complexity for quantum gates. In particular, a comparative study between the most commonly used mathematical structures and functional matrices has been performed. They have been tested under the scenarios and over the types of platforms that we understand to be the most representative.

When comparing these results to the ones achieved by simulators that directly apply the gates without calculating the tensor product, we find greater memory requirements but with smaller execution times, favoring the latter simulators. The exponentially growing nature of the matrices we are working with, along with the computational cost of the performed operations, leads to this result, making our first approach unable to match the speed of these simulators. This does not mean that functional matrices will never lead to better results than theirs. Since the development performed is just a proof of concept to test the capabilities of the functional matrices, the algorithms that implement matrix and tensor product used in this article are the naïve ones. Consequently, further work should search and implement algorithms with a smaller computational cost, leading to better results.

Apart from this, functional matrices can also be used to perform some hard calculus (in computational terms) as the partial trace of a system. This fact generates a general improvement in space complexity in any computation in which you could make use of them, for instance, whatever operation involving the density matrix calculated from the full-state vector.

QSimov quantum computing simulator is open-source and available at its GitHub repository: https://github.com/Mowstyl/QSimov/ (accessed on 1 June 2023).

Author Contributions: Conceptualization, H.I.d.l.C.C., F.C.G. and F.L.P.; Formal analysis, H.I.d.l.C.C. and F.L.P.; Investigation, H.I.d.l.C.C., F.C.G., J.J.P.G., M.M. and F.L.P.; Writing—original draft, H.I.d.l.C.C., F.C.G., J.J.P.G., M.M. and F.L.P.; Writing—reviewing and editing, H.I.d.l.C.C. and F.L.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Spanish MINECO/FEDER project AwESOMe (PID2021-122215NB-C31) and the Region of Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union and the QSimov Quantum Computing project 'Ampliación de la plataforma QSIMOV aumentando su versatilidad y conectividad' (220426UCTR).

Institutional Review Board Statement: Not applicable.

Data Availability Statement: For those readers that could be deeply interested in experiment results, the source for reproducing them can be found here https://github.com/Mowstyl/FunMatExperiments (accessed on 1 June 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Basic Quantum Gates under Consideration

Lots of different quantum gates have been used in the comparative study made in Section 4. The matrix form of all these gates is given in this appendix.

I: Identity gate for n qubits:

$$I(n) = \bigotimes_{i=1}^{n} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{A1}$$

H: Hadamard gate for n qubits:

$$H(n) = \bigotimes_{i=1}^{n} \frac{\sqrt{2}}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{A2}$$

R_x: 1-qubit gate that rotates ϕ rads around the X axis of the Bloch sphere:

$$R_x(\phi) = \begin{pmatrix} cos(\frac{\phi}{2}) & -i\, sin(\frac{\phi}{2}) \\ -i\, sin(\frac{\phi}{2}) & cos(\frac{\phi}{2}) \end{pmatrix} \tag{A3}$$

R_y: 1-qubit gate that rotates ϕ rads around the Y axis of the Bloch sphere:

$$R_y(\phi) = \begin{pmatrix} cos(\frac{\phi}{2}) & -sin(\frac{\phi}{2}) \\ sin(\frac{\phi}{2}) & cos(\frac{\phi}{2}) \end{pmatrix} \tag{A4}$$

R_z: 1-qubit gate that rotates ϕ rads around the Z axis of the Bloch sphere:

$$R_z(\phi) = \begin{pmatrix} e^{-i\frac{\phi}{2}} & 0 \\ 0 & e^{i\frac{\phi}{2}} \end{pmatrix} \tag{A5}$$

CX: 2-qubit controlled gate that rotates π rads around the X axis of the Bloch sphere of the second qubit if the first one's value is 1:

$$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{A6}$$

CY: 2-qubit controlled gate that rotates π rads around the Y axis of the Bloch sphere of the second qubit if the first one's value is 1:

$$CY = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix} \tag{A7}$$

CZ: 2-qubit controlled gate that rotates π rads around the Z axis of the Bloch sphere of the second qubit if the first one's value is 1:

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{A8}$$

Appendix B. Oracles Used in Grover's Algorithm Expermients

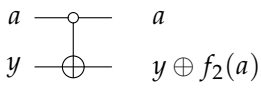

Figure A1. U_ω gate for f_2.

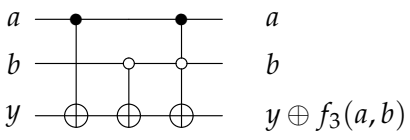

Figure A2. U_ω gate for f_3.

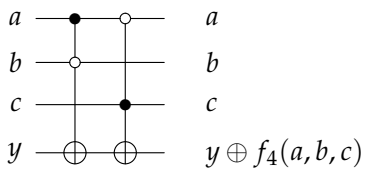

Figure A3. U_ω gate for f_4.

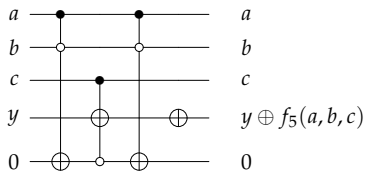

Figure A4. U_ω gate for f_5.

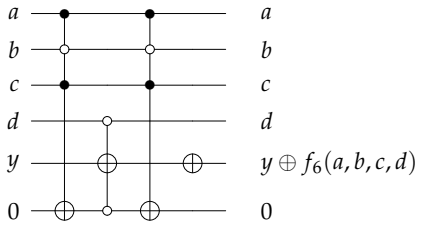

Figure A5. U_ω gate for f_6.

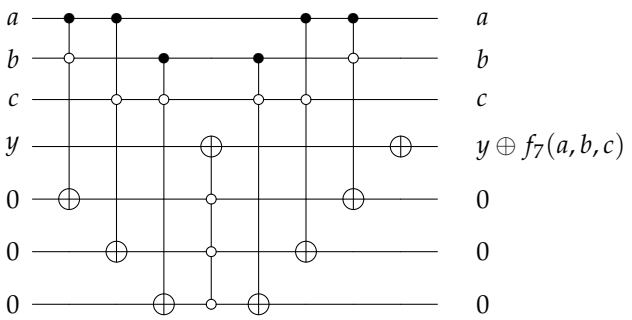

Figure A6. U_ω gate for f_7.

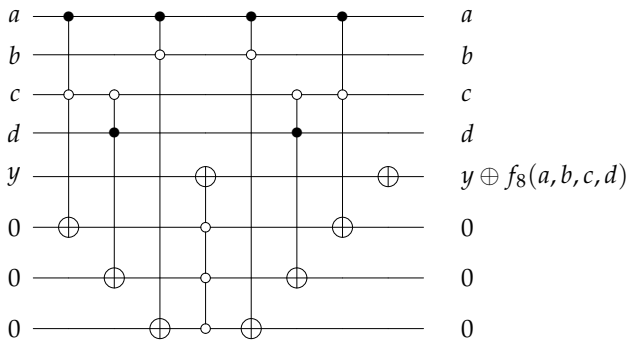

Figure A7. U_ω gate for f_8.

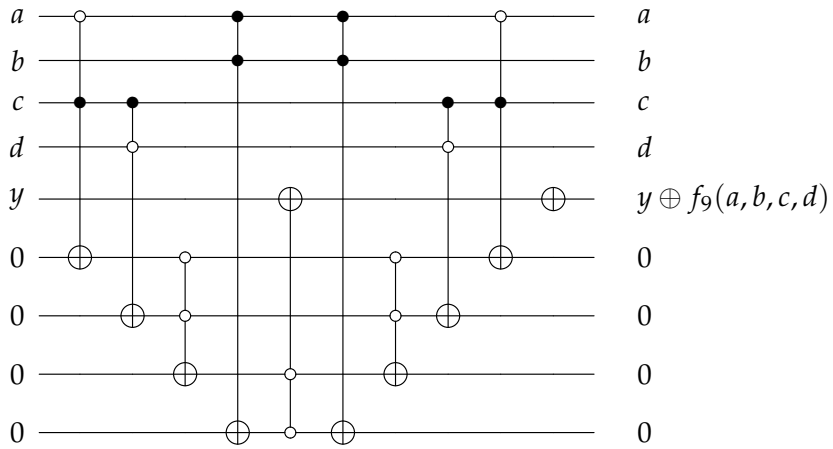

Figure A8. U_ω gate for f_9.

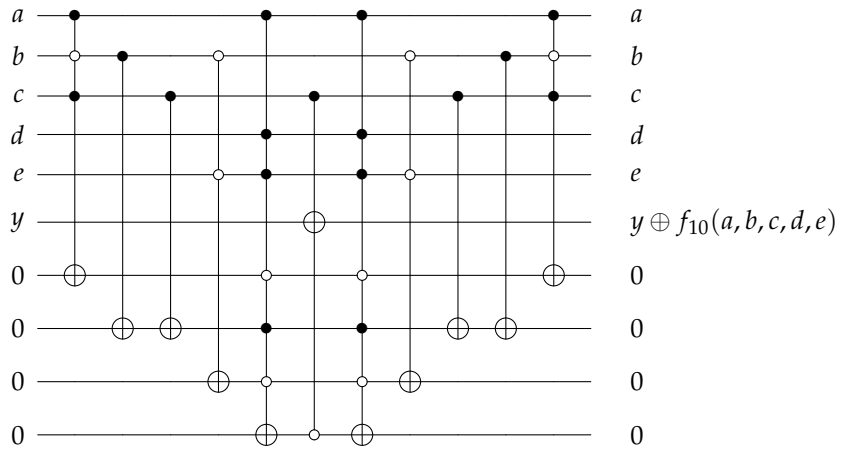

Figure A9. U_ω gate for f_{10}.

References

1. Deutsch, D.E. Quantum computational networks. *Proc. R. Soc. Lond.* **1989**, *425*, 73–90.
2. Zulehner, A.; Paler, A.; Wille, R. Efficient mapping of quantum circuits to the IBM QX architectures. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1135–1138. [CrossRef]
3. Cross, A.W.; Bishop, L.S.; Smolin, J.A.; Gambetta, J.M. Open Quantum Assembly Language. *arXiv* **2017**, arXiv:1707.03429.
4. Smith, R.S.; Curtis, M.J.; Zeng, W.J. A Practical Quantum Instruction Set Architecture. *arXiv* **2016**, arXiv:1608.03355.
5. Raedt, H.; Michielsen, K. Computational Methods for Simulating Quantum Computers. In *Host Publication*; American Scientific Publishers: Valencia, CA, USA, 2006.
6. Quantiki Wiki. List of QC Simulators. 2020. Available online: https://quantiki.org/wiki/list-qc-simulators (accessed on 20 December 2020).

7. Zhao, Y.Q.; Li, R.G.; Jiang, J.Z.; Li, C.; Li, H.Z.; Wang, E.D.; Gong, W.F.; Zhang, X.; Wei, Z.Q. Simulation of quantum computing on classical supercomputers with tensor-network edge cutting. *Phys. Rev. A* **2021**, *104*, 032603. [CrossRef]
8. Miller, D.; Thornton, M.; Goodman, D. A Decision Diagram Package for Reversible and Quantum Circuit Simulation. In Proceedings of the 2006 IEEE International Conference on Evolutionary Computation, Vancouver, BC, Canada, 16–21 July 2006; pp. 2428–2435. [CrossRef]
9. Hillmich, S.; Zulehner, A.; Kueng, R.; Markov, I.L.; Wille, R. Approximating Decision Diagrams for Quantum Circuit Simulation. *ACM Trans. Quantum Comput.* **2022**, *3*, 1–21. [CrossRef]
10. Gidney, C. My Quantum Circuit Simulator: Quirk. 2016. Available online: https://algassert.com/2016/05/22/quirk.html (accessed on 1 December 2022).
11. Khammassi, N.; Ashraf, I.; Fu, X.; Almudever, C.G.; Bertels, K. QX: A high-performance quantum computer simulation platform. In Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 464–469.
12. Barratt, F.; Dborin, J.; Bal, M.; Stojevic, V.; Pollmann, F.; Green, A.G. Parallel Quantum Simulation of Large Systems on Small Quantum Computers. *arXiv* **2020**, arXiv:2003.12087.
13. Peng, T.; Harrow, A.W.; Ozols, M.; Wu, X. Simulating Large Quantum Circuits on a Small Quantum Computer. *Phys. Rev. Lett.* **2020**, *125*, 150504. [CrossRef] [PubMed]
14. Aaronson, S.; Gottesman, D. Improved simulation of stabilizer circuits. *Phys. Rev. A* **2004**, *70*, 052328. [CrossRef]
15. Guerreschi, G.G.; Hogaboam, J.; Baruffa, F.; Sawaya, N.P.D. Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits. *Quantum Sci. Technol.* **2020**, *5*, 034007. [CrossRef]
16. Smelyanskiy, M.; Sawaya, N.P.D.; Aspuru-Guzik, A. qHiPSTER: The Quantum High Performance Software Testing Environment. *arXiv* **2016**, arXiv:1601.07195.
17. Jones, T.; Brown, A.; Bush, I.; Benjamin, S.C. QuEST and High Performance Simulation of Quantum Computers. *Sci. Rep.* **2019**, *9*, 10736. [CrossRef] [PubMed]
18. Anis, M.S.; Abraham, H.; AduOffei, R.A.; Agliardi, G.; Aharoni, M.; Akhalwaya, I.Y.; Aleksandrowicz, G.; Alexander, T.; Amy, M.; Anagolum, S. Qiskit: An Open-source Framework for Quantum Computing. 2021. Available online: https://zenodo.org/record/2562111 (accessed on 20 December 2020).
19. Karafyllidis, I.G. Quantum computer simulator based on the circuit model of quantum computation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2005**, *52*, 1590–1596. [CrossRef]
20. Barenco, A.; Bennett, C.H.; Cleve, R.; DiVincenzo, D.P.; Margolus, N.; Shor, P.; Sleator, T.; Smolin, J.A.; Weinfurter, H. Elementary gates for quantum computation. *Phys. Rev. A* **1995**, *52*, 3457–3467. [CrossRef] [PubMed]
21. Fan, H.; Matsumoto, K.; Imai, H. Quantify entanglement by concurrence hierarchy. *J. Phys. A Math. Gen.* **2003**, *36*, 4151–4158. [CrossRef]
22. Mintert, F.; Kuś, M.; Buchleitner, A. Concurrence of mixed multipartite quantum states. *Phys. Rev. Lett.* **2005**, *95*, 260502. [CrossRef]
23. Chen, K.; Albeverio, S.; Fei, S.M. Concurrence of arbitrary dimensional bipartite quantum states. *Phys. Rev. Lett.* **2005**, *95*, 040504. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.