A tree-approach Pauli decomposition algorithm with application to quantum computing

1st Océane Koska
LMF and Eviden Quantum Lab
Université Paris-Saclay and ATOS/Eviden
Les Clayes-sous-Bois, France
oceane.koska@eviden.com

2nd Marc Baboulin
Laboratoire Méthodes Formelles (LMF)
Université Paris-Saclay
Orsay, France
marc.baboulin@universite-paris-saclay.fr

3rd Arnaud Gazda
Eviden Quantum Lab
ATOS/Eviden
Les Clayes-sous-Bois, France
arnaud.gazda@eviden.com

***Abstract*—The Pauli matrices are 2-by-2 matrices that are very useful in quantum computing. They can be used as elementary gates in quantum circuits but also to decompose any matrix of $\mathbb{C}^{2^n \times 2^n}$ as a linear combination of tensor products of the Pauli matrices. However, the computational cost of this decomposition is potentially very expensive since it can be exponential in n. In this paper, we propose an algorithm with a parallel implementation that optimizes this decomposition using a tree approach to avoid redundancy in the computation while using a limited memory footprint. We also explain how some particular matrix structures can be exploited to reduce the number of operations. We provide numerical experiments to evaluate the sequential and parallel performance of our decomposition algorithm and we illustrate how this algorithm can be applied to encode matrices in a quantum memory.**

***Index Terms*—Quantum computing, Matrix decomposition, Pauli matrices, Tree exploration, Block-encoding**

I. Introduction

The Pauli matrices [19, p. 65] are four 2-by-2 matrices that are commonly used in quantum physics and quantum computing. They constitute the so-called Pauli group and are given below:

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

Namely, these matrices correspond respectively to the identity matrix, the NOT operator, and two rotations. Note that in some textbooks, I is not included in the Pauli group. The Pauli matrices form a basis of $\mathbb{C}^{2\times 2}$ and when we combine them using n tensor products we obtain 4^n matrices that we will call *Pauli operators* in the remainder and which form a basis of $\mathbb{C}^{2^n \times 2^n}$.

In quantum mechanics, these matrices are related to the observables describing the spin of a spin-$\frac{1}{2}$ particle [8]. When combined using tensor products, the Pauli matrices can be used to describe multi-qubit Hamiltonians [17] and in quantum error-correcting codes [11]. The decomposition of matrices in the Pauli operator basis has a wide range of applications. For instance, it is used in the construction of several quantum algorithms in physics, chemistry, or machine-learning problems (for example in the Hamiltonian decomposition to describe many-body spin glasses [13]). The Pauli decomposition is performed on a classical computer and is part of so-called hybrid quantum-classical algorithms. It can be for instance used in variational algorithms [2], [25], to build observables from arbitrary matrices in many simulation frameworks [7], [20], [22] or to encode data in the quantum memory of a quantum computer [10].

Due to the exponential cost in the number of qubits (for generic matrices), it is necessary to reduce this cost to its minimum. In this work, we propose an algorithm to decompose any matrix of $\mathbb{C}^{2^n \times 2^n}$ in the Pauli operator basis. This method, that we name Pauli Tree Decomposition Routine (PTDR), exploits the specific form of Pauli operators and uses a tree approach to avoid redundancy in the computation of the decomposition. We also take advantage of some specific structures of the input matrices. We propose a parallel (multi-threaded) version of our algorithm targeting one computational node and present a strong scaling analysis. Due to the exponential cost in time and memory, we also anticipate a future distributed multi-node version by extrapolating scalability results on larger problems.

This paper is organized as follows: In Section II we recall (and demonstrate) some results about the Pauli decomposition and we present recent related work. Then in Section III we describe our decomposition algorithm along with its complexity analysis. In Section IV we adapt the algorithm to special cases of matrix structures (diagonal, tridiagonal, ...) and we explain how we can obtain a decomposition from existing ones for several matrix combinations. In Section V we present performance results where we compare our algorithm with existing implementations and we propose a multi-threaded version. Then in Section VI we present an application of this decomposition to encode matrices in quantum computers via the block-encoding technique. Finally, concluding remarks are given in Section VII.

In the remainder we will use the following notations:

- We will work in complex Euclidean spaces with canonical basis. On such vector spaces, the tensor product and the Kronecker product coincide [18] and we use the term *tensor product* (denoted as $\otimes$) throughout this paper.
- A^* denotes the conjugate transpose of a matrix A.
- The notation for a bitstring in base 2 is illustrated by the following example: $\overline{1010}_2 = 10$.
- For an array k and some indices a and b, $k[a : b]$ corresponds to the sub-array extracted from k between indices a included and b not included.

II. Background and related work

A. Pauli decomposition

a) Pauli operator basis: Let $n \in \mathbb{N}^*$, the Pauli operator basis of size n corresponds to the set

$$\mathcal{P}_n = \left\{ \bigotimes_{i=1}^{n} M_i, \;\; M_i \in \{I, X, Y, Z\} \right\},$$

where I, X, Y and Z are the Pauli matrices.

Theorem 1. *P_n is a basis of $\mathbb{C}^{2^n \times 2^n}$.*

Proof. $\mathcal{P}_n$ has 4^n elements in a 4^n-dimensional space then we need to show that $\mathcal{P}_n$ is linearly independent. Let us consider the standard inner product for matrices defined as $\langle A|B\rangle = Tr(A^*B)$. It can be easily verified that the 2-by-2 Pauli matrices are mutually orthogonal with respect to this inner product. If now we have two distinct Pauli operators $A, B \in \mathcal{P}_n$ such that $A = A_1 \otimes A_2 \otimes \cdots \otimes A_n$ and $B = B_1 \otimes B_2 \otimes \cdots \otimes B_n$. Then

$$\begin{aligned}\langle A|B\rangle = Tr(A^*B) &= Tr\left(\bigotimes_{i=1}^{n} A_i^* \bigotimes_{i=1}^{n} B_i\right) \\ &= Tr\left(\bigotimes_{i=1}^{n} A_i^* B_i\right) = \prod_{i=1}^{n} Tr(A_i^* B_i).\end{aligned}$$

Since $A \neq B$, there exists i such that $A_i \neq B_i$ and then $Tr(A_i^* B_i) = 0$ (because the Pauli matrices A_i and B_i are orthogonal) and thus $Tr(A^*B) = 0$.

As a result $\mathcal{P}_n$ is an orthogonal (and even orthonormal) set and is linearly independent. □

b) Decomposition in the Pauli operator basis: Following Theorem 1, any matrix $A \in \mathbb{C}^{2^n \times 2^n}$ can be decomposed in the Pauli operator basis of size n (we can use a zero padding to make any size of matrix fits this constraint) and then can be expressed as

$$A = \sum_{P_i \in \mathcal{P}_n} \alpha_i P_i, \text{ with } \alpha_i \in \mathbb{C}.$$

Theorem 2. *If A is Hermitian ($A^* = A$) then the α_i's in the Pauli decomposition are real numbers.*

Proof. Since A and the Pauli matrices are Hermitian we get

$$\sum_{P_i \in \mathcal{P}_n} \alpha_i P_i = \left(\sum_{P_i \in \mathcal{P}_n} \alpha_i P_i\right)^* = \sum_{P_i \in \mathcal{P}_n} (\alpha_i P_i)^* = \sum_{P_i \in \mathcal{P}_n} \bar{\alpha_i} P_i,$$

thus $\forall i, \;\; \alpha_i = \bar{\alpha_i}$ and $\alpha_i \in \mathbb{R}$.

□

c) Straightforward decomposition method: To decompose A in the corresponding Pauli operator basis $\mathcal{P}_n$ the idea is to compute each coefficient separately, similarly to [6].

For example, if $A \in \mathbb{C}^4 \times \mathbb{C}^4$ then

$$A = \alpha_{II}(I \otimes I) + \alpha_{IX}(I \otimes X) + \alpha_{IY}(I \otimes Y) + \ldots$$

To compute a given coefficient $\alpha_{M_1 M_2}$, where $M_1, M_2 \in \{I, X, Y, Z\}$ we use the fact that

$$\alpha_{M_1 M_2} = \frac{1}{4} Tr\big((M_1 \otimes M_2)A\big).$$

This can be generalized to a matrix A of size $2^n \times 2^n$ where we have

$$\alpha_{M_1 M_1 \ldots M_n} = \frac{1}{2^n} Tr\big((\bigotimes_i M_i)A\big).$$

Therefore, for each of the 4^n coefficients we need to compute:

- the tensor product of n Pauli matrices,
- the trace of this product multiplied by the matrix A.

If these tasks are achieved without any consideration of the matrix structure the total cost of the algorithm would be $\mathcal{O}(2^{4n})$ (complex flops) because of the computational cost of the tensor product.

d) First optimization of the trace computation: The trace computation can be easily optimized with three ideas:

- Pauli operator matrices only contain $1, -1, i, -i$ values so the multiplications involved in the tensor product are simpler.
- The Pauli operators P_i are sparse since they have only one entry in each row and column. Consequently, computing an element of the product between a Pauli operator and a matrix A requires only one multiplication.
- To compute the trace, we only need to compute the diagonal entries of the product between the Pauli operator and A.

By using these three basic ideas, the trace computation can be performed in $\mathcal{O}(2^n)$ arithmetical operations instead of $\mathcal{O}(2^{2n})$. However, the cost of the tensor products mentioned above still dominates the computation.

B. Related work

A technique is presented in [21] to decompose a square real symmetric matrix H of any arbitrary size in the corresponding Pauli basis. This technique relies on solving a linear system of equations. A Python implementation is provided. However, this method is not optimal because it relies on the straightforward generation of all the tensor products and their storage in a dictionary. Therefore it is expensive in computational time.

We can also find decomposition routines in quantum simulation tools like the open source software framework PennyLane [20] (*pauli_decompose routine*).

In a recent work [23], an algorithm called *PauliComposer* is introduced to compute tensor products of Pauli matrices. The authors use this algorithm to decompose a Hamiltonian (Hermitian matrix) in the Pauli basis, which is an application similar to the one described in Section VI. The algorithm exploits the particular structure of the Pauli operators to avoid a huge part of the computation. Below are more details about this work.

a) PauliComposer - notations: Let $P \in \mathcal{P}_n$ such that $P = \sigma_{n-1} \otimes \cdots \otimes \sigma_0$ where $\forall i,\ \sigma_i \in \mathcal{P}_1 = \{I, X, Y, Z\}$. P being a Pauli operator, for each row there will be only one column with a non-zero element. Consequently, we can use a sparse matrix structure to store the matrix with two arrays k and m of size 2^n. Given a row j, the column index of the non-zero element is noted $k[j]$, and its value is noted $m[j]$. Formulated differently, for each j, the j-th non-zero coefficient of the P matrix is denoted

$$m[j] = P_{j,k[j]}$$

.

Another important thing to notice is that a Pauli operator is either real (with values 0, $+1$, -1) or complex (with values 0, $+i$, $-i$). Therefore we can switch from the set $\mathcal{P}_1 = \{I, X, Y, Z\}$ to $\tilde{\mathcal{P}}_1 = \{I, X, iY, Z\}$ and construct

$$\tilde{P} := \tilde{\sigma}_{n-1} \otimes \cdots \otimes \tilde{\sigma}_0$$

where $\forall i, \tilde{\sigma_i} \in \tilde{\mathcal{P}}_1$. By counting the number of iY in $\tilde{P}$, denoted n_Y, one can recover

$$P = (-i)^{n_Y \mathrm{mod} 4} \tilde{P}.$$

In the following, a diagonality function d is defined to track the diagonality of a matrix $M \in \mathcal{P}_1$,

$$d(M) = \begin{cases} 0 & \text{if } M = I \text{ or } M = Z, \\ 1 & \text{if } M = X \text{ or } M = Y \end{cases}$$

b) PauliComposer - algorithm: The *PauliComposer* algorithm is an iterative algorithm:

- For the first row ($j = 0$) the nonzero element can be found in the column
$$k[j = 0] = \overline{d(\sigma_{n-1}) \ldots d(\sigma_0)}_2.$$
- For the following entries, when 2^l elements have already been computed, the next 2^l elements can be found using these entries. For the column indices, we have
$$k[j + 2^l] = k[j] + (-1)^{d(\sigma_l)} 2^l, \quad j \in [\![0, 2^l - 1]\!],$$
and for the values of these nonzero elements we have
$$m[j + 2^l] = P_{j+2^l, k[j+2^l]} = \zeta_l P_{j,k[j]}, \quad j \in [\![0, 2^l - 1]\!],$$
where $\zeta_l = 1$ if $\sigma_l \in \{I, X\}$ and $\zeta_l = -1$ otherwise.

With the *PauliComposer* algorithm, if we consider the worst-case scenarios, we need to perform $\mathcal{O}(2^n)$ sums and $\mathcal{O}(2^n)$ changes of sign (see [23]). This algorithm improves the state of the art in computing tensor products of Pauli matrices. However, to compute a decomposition in the Pauli basis, the authors just iterate over all the elements, without taking into account the similarities between a Pauli operator to another. In the next section, in which we aim to propose a faster algorithm, we will keep the same notations as those used in describing the *PauliComposer* algorithm.

A very recent algorithm called *Tensorized Pauli Decomposition* [12] has been proposed for Pauli decomposition, faster than previously existing sequential solutions. This algorithm (with Python sequential implementation) uses matrix partitioning to accelerate the computation. We will also compare our sequential code with this algorithm in the numerical experiments.

III. Pauli Tree Decomposition

A. Description of the algorithm

Our algorithm referred to as *Pauli Tree Decomposition Routine* is given in Algorithm 1. It uses a tree exploration to reduce significantly the number of elementary operations needed to perform the decomposition in the Pauli basis of a given matrix in $\mathbb{C}^{2^n \times 2^n}$. We exploit the redundancy of information from one Pauli operator to another. For instance, I and X contain the same values but not in the same locations. On the contrary, considering X and iY, the non-zero values are at the same place in both matrices but the values are different.

The tree depicted in Figure 1 (that we call *Pauli tree*) represents all the possible Pauli operators for a given depth equal to the size of the operators. In this example, the path in red corresponds to all the Pauli operators ending with $\cdots \otimes X \otimes I$. The tree starts with a root associated with no Pauli matrix then each node of the tree has 4 children, one for each matrix $I, X, Y$and Z until the depth of n is obtained, then the final nodes $I, X, Y$and Z are just becoming leaves. To iterate over all elements of the Pauli basis $\mathcal{P}_n$ we use an

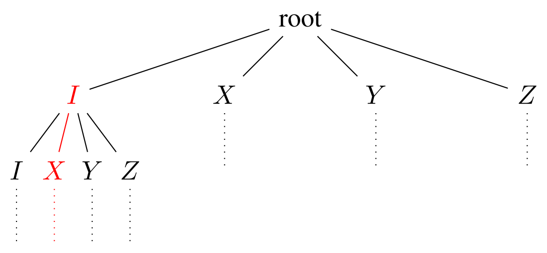


Fig. 1. Example of Pauli tree.

in-order tree exploration as shown in Figure 2.

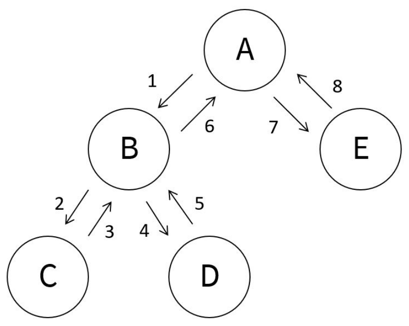


Fig. 2. Inorder walk through a tree. The walk starts from A and follows the arrows from 1 to 8. The node exploration is the following: ABCBDBAEA

The tree has its own arrays k and m that respectively track the column indices and non-zero values. In addition, we save memory space by not storing the array j because we already know that $j = [\![0, 2^n - 1]\!]$. Arrays k and m are initialized with 0 except for their first elements.

- $k[0] = \overline{d(x_n - 1) \dots d(x_0)}_2$,
- $m[0] = 1$.

Then these arrays are updated through the tree exploration (see Algorithm 4) to get at any time the current sparse matrix representation. The update rules to get from one node to another are simple, at depth $l \neq 0$ ($l = 0$ being the root, $l = n$ being the leaves):

- The current node is I: we only need to copy the 2^l top elements in the bottom part by shifting them to 2^l to the right. Therefore, we update both k and m
  - $k[2^l : 2^{l+1}] = k[0 : 2^l] + 2^l$
  - $m[2^l : 2^{l+1}] = m[0 : 2^l]$
- The current node is X: we already have computed the tree update for I but know we want to get X. The values are going to be the same however we need to shift the element's position so we only update in place k
  - $k[2^l : 2^{l+1}] = k[2^l : 2^{l+1}] - 2^{l+1}$
- The current node is Y: similarly, by comparing X we already have and iY we want, we notice that the positions are the same but the values are changing, so we only update in place m
  - $m[2^l : 2^{l+1}] = -m[2^l : 2^{l+1}]$
- The current node is Z: by comparing iY we already have and Z, we notice that this time the values are the same but the positions are changing, so we only update in place k
  - $k[2^l : 2^{l+1}] = k[2^l : 2^{l+1}] + 2^{l+1}$

These rules are used in Algorithm 3 to update the tree environment. Moreover, if $l = n$ then it means we have reached the leaves, therefore we have all the information needed to immediately compute the coefficient of the decomposition of our matrix related to the current Pauli operator in the tree using Algorithm 2. When we have explored the whole tree all the coefficients have been computed and stored.

Algorithm 1 Pauli Tree Decomposition Routine

```
Input: matrix in $\mathbb{C}^{2^n} \times \mathbb{C}^{2^n}$

tree ← Pauli tree of depth $n$
explore_node(tree.root, tree)
```

Algorithm 2 compute_coeff

```
Input: tree with arrays $k$ and $m$, current number of Y $n_Y$ and matrix $A \in \mathbb{C}^{2^n} \times \mathbb{C}^{2^n}$

coeff ← 0
for $j$ in $0 \dots 2^n - 1$ do
    coeff ← coeff + $(-i)^{n_Y \bmod 4} m[j] \times A[\ k[j]\ 545][j]$
end for
add coeff in the list of coefficients of the decomposition
```

Algorithm 3 update_tree

```
Input: current_node ∈ $\{I, X, Y, Z\}$, tree with arrays $k$ and $m$

$l \leftarrow$ tree.depth − current_node.depth − 1
if current_node is I then
    $k[2^l : 2^{l+1}] \leftarrow k[0 : 2^l] + 2^l$
    $m[2^l : 2^{l+1}] \leftarrow m[0 : 2^l]$
else if current_node is X then
    $k[2^l : 2^{l+1}] \leftarrow k[2^l : 2^{l+1}] - 2^{l+1}$
else if current_node is Y then
    $m[2^l : 2^{l+1}] \leftarrow -m[2^l : 2^{l+1}]$
else if current_node is Z then
    $k[2^l : 2^{l+1}] \leftarrow k[2^l : 2^{l+1}] + 2^{l+1}$
end if
if current_node.depth = 0 then
    tree.compute_coefficient
end if
```

Algorithm 4 explore_node

```
Input: current_node ∈ $\{I, X, Y, Z\}$, tree

update_tree(current_node, tree)
if current_node.depth $> 0$ then
    explore_node(current_node.childI, tree)
    explore_node(current_node.childX, tree)
    explore_node(current_node.childY, tree)
    explore_node(current_node.childZ, tree)
end if
```

B. Complexity analysis

Here we are going to compare the number of operations needed to iterate over all the Pauli basis elements to perform a decomposition in this basis, if we use or not a tree approach.

a) Without the tree approach: Without the tree approach, we need to compute each of the 4^n elements of the Pauli basis $\mathcal{P}_n$ using [23]. This task requires filling 2 arrays of size 2^n using elementary operations like addition, multiplication, and memory copy, so 2×2^n elementary operations. So considering all the 4^n possible Pauli operators to compute we have a count of elementary operations of

$$C_{notree} = 2 \times 8^n.$$

b) With the tree approach: With the tree approach, we compute the coefficients step by step, using the redundancy of information. With this approach, we drastically reduce the number of elementary operations needed for the tensor product part of the computation. The number of elementary operations performed at depth $l \neq 0$ for a "sibling" group of nodes is $5 \times 2^{l-1}$. At each depth $l \neq 0$, the number of "sibling" group of nodes is 4^{l-1}. Consequently, we have a total of

$$C_{tree} = 2 + \sum_{l=1}^{n} (5 \times 2^{l-1}) 4^{l-1} = \frac{9}{7} + \frac{5}{7} 8^n$$

operations. Thus for this task, we improve the complexity by a factor $\frac{14}{5}$, in comparision with *PauliComposer*.

IV. Special cases

A. Diagonal and band matrices

1) Diagonal matrices: Let D be a diagonal matrix in $\mathbb{C}^{2^n \times 2^n}$. Let us consider the subset $\mathcal{P}_n^{IZ}$ of $\mathcal{P}_n$ defined as

$$\mathcal{P}_n^{IZ} = \left\{ \bigotimes_{i=1}^{n} M_i | M_i \in \{I, Z\} \right\},$$

where only I and Z matrices are allowed in the tensor products. Then D admits a unique decomposition in $\mathcal{P}_n^{IZ}$.

Proof. Let us consider a diagonal matrix $D \in \mathbb{C}^{2^n \times 2^n}$. We know that D admits a decomposition in $\mathcal{P}_n$

$$D = \sum_{P_i \in \mathcal{P}_n} \alpha_i P_i,$$

where α_i's are complex numbers. P_n is a basis of $\mathbb{C}^{2^n \times 2^n}$ so no diagonal P_i can be obtained with a linear combination of non-diagonal P_i, so non-diagonal P_i leads to non-diagonality in the final result. The P_i that are diagonal are only composed of I and Z. □

Therefore we can adapt the algorithm if we have a diagonal matrix by just removing the X and Y children in the tree. This results in only tracking the values in m because we necessarily have $k = [\![0, 2^n - 1]\!]$. Moreover, by removing the Y matrix the vector m can only contains 1 and -1, we can store using only booleans (0 for 1 and 1 for -1).

For this adaptation of the algorithm, the complexity of the tensor product part using the tree decomposition routine is

$$C_{tree}^{diagonal} = 2 + \sum_{l=1}^{n} (2 \times 2^{l-1}) 2^{l-1} = \frac{4}{3} + \frac{2}{3} 4^n.$$

Remark 1. *For anti-diagonal matrices, we can get a similar result by replacing I and Z with X and Y. This idea is also relevant in the following.*

2) Tridiagonal matrices: Let A be a tridiagonal matrix in $\mathbb{C}^{2^n \times 2^n}$ (i.e. if $i < j - 1$ or $i > j + 1$ then $a_{i,j} = 0$). This special structure of matrices leads to some constraints on the decomposition in the Pauli basis and these constraints can be exploited to reduce the total complexity of the algorithm. We can remove some branches of the tree because they lead to incompatible paths.

In the decomposition are allowed only the terms that:

- either fit the tridiagonal structure; so diagonal or tridiagonal themselves,
- or there exist several other terms that can cancel with each other on the extra terms out of the tridiagonal structure.

Theorem 3. *Let A be a tridiagonal matrix in $\mathbb{C}^{2^n \times 2^n}$ then there are at most $(n + 1)2^n$ non-zero terms in its Pauli decomposition.*

Proof. Given $n \in \mathbb{N}^*$, let us consider the subsets of $\mathcal{P}_n$, $\mathcal{P}_n^{IZ}$ and

$$\mathcal{P}_n^{XY} = \left\{ \bigotimes_{i=1}^{n} M_i | M_i \in \{X, Y\} \right\}.$$

Let $A \in \mathbb{C}^{2^n \times 2^n}$ be a matrix such that

$$A = \sum_j \alpha_j P_j,$$

where $P_j \in \mathcal{P}_n$. Then let us consider the following subset of $\mathcal{P}_n$

$$\Gamma_n = \left\{ \bigotimes_{i=1}^{m} V_i \bigotimes_{i=m+1}^{n} W_i | m \in [\![1, n]\!], V_i \in \{I, Z\}, W_i \in \{X, Y\} \right\}$$

Let us show that $\forall n \in \mathbb{N}^*, \exists i \in [\![1, 2^n]\!], \alpha_i \neq 0, P_i \notin \Gamma_n \implies$ A is not tridiagonal. In other words, only the elements of Γ_n are allowed in the decomposition of a tridiagonal matrix of size $2^n \times 2^n$.

For $n = 1$, $\Gamma_1 = \{I, X, Y, Z\} = \mathcal{P}_1$, so the property is true.

Now we suppose (recurrence hypothesis) that for $n > 0, \exists i \in [\![1, 2^n]\!], \alpha_i \neq 0, P_i \notin \Gamma_n \implies$ A is not tridiagonal. Let us show by recurrence that for $n + 1$ we also have this property. Let us take a matrix $A \in \mathbb{C}^{2^{n+1} \times 2^{n+1}}$, then A can be decomposed in the Pauli basis as

$$A = \sum_{i=1}^{4^{n+1}} \alpha_i P_i,$$

where $P_i \in \mathcal{P}_{n+1}$. This sum can be split

$$A = \sum_{i=1}^{4^n} \alpha_i^I I \otimes Q_i + \sum_{i=1}^{4^n} \alpha_i^X X \otimes Q_i + \sum_{i=1}^{4^n} \alpha_i^Y Y \otimes Q_i + \sum_{i=1}^{4^n} \alpha_i^Z Z \otimes Q_i,$$

where $Q_i \in \mathcal{P}_n$. For the sums about I and Z, the corresponding matrices structure is

$$\begin{bmatrix} . & 0 \\ 0 & . \end{bmatrix},$$

thus the matrices are tridiagonal if and only if their blocks are tridiagonal so we refer to the recurrence hypothesis. For the sums about X and Y the structure of the matrix is the following

$$\begin{bmatrix} 0 & . \\ . & 0 \end{bmatrix},$$

Here the blocks need to be zero except for respectively the bottom left and top right parts. If A is tridiagonal then in the sum

$$\sum_{i=1}^{4^n} \alpha_i^X X \otimes Q_i + \sum_{i=1}^{4^n} \alpha_i^Y Y \otimes Q_i$$

for all $Q_i \in \mathcal{P}_n \setminus \mathcal{P}_n^{XY}, \alpha_i^Y = 0$ and $\alpha_i^X = 0$. In other words, it means that we only keep the elements Q_i that are built with X and Y only (because with a non-zero value only on the bottom left and top right corners). Consequently for $n+1, \exists i \in [\![1, 2^{n+1}]\!], \alpha_i \neq 0, P_i \notin \Gamma_n \implies$ A is not tridiagonal. Thus the property is also true for $n + 1$.

Thus for all $n > 0$, the compatible elements are only the ones in Γ_n, so we have a maximum of $(n + 1)2^n$ non-zero terms in the decomposition. □

To compute the complexity of our algorithm adapted to the tridiagonal case we notice that it corresponds to the diagonal case for the $n-1$ first levels of the trees and the last one is the same as in the general case. For the $n-1$ first levels the complexity is $\frac{4}{3}+\frac{2}{3}4^{n-1}$. Then for the last one, the number of flops is $5 \times 2^{n-1} \times 2^{n-1}$. Therefore, the complexity generated by the tensor products is

$$C_{tree}^{tridiagonal} = \frac{4}{3} + \frac{17}{12}4^n.$$

3) Band-diagonal matrices: We can generalize the previous case to band diagonal matrices of width $(2s+1)$. Let A be a matrix in $\mathbb{C}^{2^n \times 2^n}$. A is a band diagonal matrices of width $(2s+1)$ if $i < j-s$ or $i > j+s$ implies that $a_{i,j}=0$

Theorem 4. *Let $s>0$. Let A be a band diagonal matrix of width $(2s+1)$ in $\mathbb{C}^{2^n \times 2^n}$. Then the Pauli decomposition of A contains at most*

$$(sn - c(s))\, 2^n,$$

where

$$c(s) = s\left(\lfloor \log_2(s) \rfloor + 1\right) - 2^{\lfloor \log_2(s) \rfloor + 1}.$$

Proof. The proof is similar to the tridiagonal, case except we take

$$\Gamma_n = \left\{ \bigotimes_{i=1}^{m} V_i \bigotimes_{i=m+1}^{n-M-1} W_i \bigotimes_{i=n-M}^{n} U_i \;\middle|\; 1 \le m < n-m, \right.$$
$$\left. V_i \in \{I,Z\}, W_i \in \{X,Y\}, U_i \in \{I,X,Y,Z\} \right\},$$

where $M = \lfloor \log_2(s) \rfloor + 1$. The cardinal of this set is $(sn - c(s))\, 2^n$. □

To compute the complexity of our algorithm adapted to the band-diagonal case we notice that it corresponds to the diagonal case for the $n-s$ first levels of the tree and the last s ones are the same as in the general case. For the $n-s$ first levels the complexity is $\frac{4}{3}+\frac{2}{3}4^{n-s}$. Then for the last ones, the number of flops for the depth l is $5 \times 4^l \times 2^{n-s} \times 2^{l-s+n}$. By summing these terms from $i=0$ to $i=s-1$ we get the complexity for the last levels and the total complexity generated by the tensor products is

$$C_{tree}^{band-diagonal} = \frac{4}{3} + \frac{5}{7}\left(8^s - \frac{1}{15}\right)4^{n-s}.$$

Note that for $s=1$ we retrieve the same complexity as for the tridiagonal case.

Regarding the storage, our algorithm requires two arrays of size 2^n, one of unsigned integers (the location, k) and one containing the $+1$ and -1, that can be encoded as boolean (the values, m). Moreover, we need of a dictionary storing the different Pauli string, of length n, associated to their coefficients in the decomposition. We have at most 4^n pairs to store.

4) Summary: For the general and special cases given previously, we summarize in Table I the maximum number of terms in the Pauli decomposition and the computational cost to compute it. Note that all flops involved for the tensor product are in real arithmetic. To be consistent in our formulas, the complex flops required for the Trace computation have been converted into real flops. We have removed the constants from the formulas given previously.

TABLE I
NUMBER OF TERMS AND FLOPS FOR PAULI DECOMPOSITION.

	Max term count	Complexity (flops)
general	4^n	$\mathcal{O}(8^n)$
diagonal	2^n	$\mathcal{O}(4^n)$
tridiagonal	$(n+1)2^n$	$\mathcal{O}(n4^n)$
band-diagonal	$(sn-c(s))2^n$	$\mathcal{O}(sn4^n)$

B. Combinations of matrices

Because of the computational cost of the decomposition in the Pauli basis, it could be useful, when the matrix to decompose is a combination of other matrices, to take advantage of existing Pauli decompositions of these matrices if available. In this section, we express the decomposition resulting from some matrix operations on existing Pauli decompositions.

1) Direct sum: Let $A, B \in \mathbb{C}^{2^n \times 2^n}$, we recall that the direct sum of A and B is

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}.$$

Suppose we have the Pauli decompositions

$$A = \sum_j \alpha_j P_j, \text{ and } B = \sum_j \beta_j P_j,$$

then the decomposition of $C = A \oplus B$ is

$$C = I \otimes \sum_j \frac{\alpha_j + \beta_j}{2} P_j + Z \otimes \sum_j \frac{\alpha_j - \beta_j}{2} Pj.$$

Proof.

$$C = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} = I \otimes \frac{A+B}{2} + Z \otimes \frac{A-B}{2},$$

we have

$$C = I \otimes \sum_j \frac{\alpha_j + \beta_j}{2} P_j + Z \otimes \sum_j \frac{\alpha_j - \beta_j}{2} P_j.$$

□

2) Block-diagonal matrices: Given $m, k > 0$, let us consider $N = 2^m$ matrices $A_i \in C^{2^k \times 2^k}$(it is always possible to use zero padding to reach a compatible amount of matrices). Let us consider the block diagonal matrix

$$\mathcal{A}_N = \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_N \end{bmatrix}.$$

If we already know the Pauli decomposition of the A_i's, then it is possible to construct the decomposition of $\mathcal{A}_N$ using several times the decomposition of a direct sum (see above) by grouping the A_i matrices two by two recursively. This is an example for $N = 8$:

$$\mathcal{A}_N = \Big((A_1 \oplus A_2) \oplus (A_3 \oplus A_4)\Big) \oplus \Big((A_5 \oplus A_6) \oplus (A_7 \oplus A_8)\Big)$$

3) Linear combination: Let $A, B \in \mathbb{C}^{2^n \times 2^n}$ and $\mu \in \mathbb{C}$. If

$$A = \sum_j \alpha_j P_j, \text{ and } B = \sum_j \beta_j P_j,$$

then the Pauli decomposition of $C = \mu A + B$ is trivially

$$C = \sum_j (\mu \alpha_j + \beta_j) P_j.$$

4) Multiplication: Let $A, B \in \mathbb{C}^{2^n \times 2^n}$ with

$$A = \sum_j \alpha_j P_j, \text{ and } B = \sum_j \beta_j P_j,$$

then the decomposition of $C = A \times B$ can be directly deduced from these decompositions, instead of performing the multiplication $A \times B$ and then decomposing it. The resulting decomposition is

$$C = \sum_{j,k} \alpha_j \beta_q (P_j \times P_q),$$

where $P_j \times P_q = \bigotimes_{i=1}^n M_i^j \times \bigotimes_{i=1}^n M_i^q = \bigotimes_{i=1}^n \left(M_i^j \times M_i^q\right)$, with M_i^j and M_i^q being Pauli matrices. The product of Pauli matrices always results in a Pauli matrix (multiplied or not by a complex factor) so one can avoid the computation and just refer to a computation table.

5) Hermitian matrix augmentation: Let $A \in \mathbb{C}^{2^n \times 2^n}$ with Pauli decomposition

$$A = \sum_{P_j \in \mathcal{P}_n} \alpha_j P_j.$$

Let now consider the Hermitian augmented matrix

$$\tilde{A} = \begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix}.$$

Then the decomposition of $\tilde{A}$ in $\mathcal{P}_{n+1}$ can be directly obtained from the decomposition of A in $\mathcal{P}_n$

$$\tilde{A} = X \otimes \sum_{P_j \in \mathcal{P}_n} a_j P_j + Y \otimes \sum_{P_j \in \mathcal{P}_n} b_j P_j,$$

where a_j and b_j are the real and imaginary part of α_j, respectively.

Proof.

$$\begin{aligned}
\tilde{A} = \begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix} &= \begin{bmatrix} 0 & \sum\limits_{P_j \in \mathcal{P}_n} \overline{\alpha_j} P_j \\ \sum\limits_{P_j \in \mathcal{P}_n} \alpha_j P_j & 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & \sum\limits_{P_j \in \mathcal{P}_n} a_j P_j \\ \sum\limits_{P_j \in \mathcal{P}_n} a_j P_j & 0 \end{bmatrix} + \begin{bmatrix} 0 & -i \sum\limits_{P_j \in \mathcal{P}_n} b_j P_j \\ i \sum\limits_{P_j \in \mathcal{P}_n} b_j P_j & \end{bmatrix} \\
&= \quad X \otimes \sum_{P_j \in \mathcal{P}_n} a_j P_j \quad + \quad Y \otimes \sum_{P_j \in \mathcal{P}_n} b_j P_j.
\end{aligned}$$

□

This result is interesting because it means we do not need to compute the decomposition of $\tilde{A}$ if we already know the decomposition of A. This is advantageous because the decomposition algorithm has an exponential behavior. Moreover, the number of nonzero terms in the decomposition of $\tilde{A}$ is at most twice that of A.

V. Numerical experiments

The experiments have been carried out on one node of the QLM (Quantum Learning Machine) located at EVIDEN/BULL. This node is a 16-core (32 threads with hyper-threading) Intel(R) Xeon(R) Platinum 8153 processor at 2.00 GHz. In the following, we consider the decomposition in the Pauli basis of a generic matrix, with no specific structure.

A. Sequential code

We plot in Figure 3 the execution time for computing the decomposition in the Pauli basis using the existing Python implementations *H2zixy* [21], PennyLane [20], *PauliComposer* [23] and *Tensorized Pauli Decomposition* [12] and our tree-based code implemented in Python and C++. Our algorithm enables to address, in the same amount of time, one more qubit in the decomposition than *PauliComposer* (for $n \geq 2$) and outperforms the *H2zixy* and Pennylane implementations. Because of the exponential complexity in n, this improvement is significant in terms of execution time. However, for higher number of qubits the *Tensorized Pauli Decomposition* is faster because it scales better in time than all the other algorithms. Note that for these sequential codes, the plots have been limited to 30 minutes of execution time and $n = 12$ (which corresponds to dense matrices of size 4096×4096 in complex arithmetic) because the time increases exponentially with n.

To be able to address larger problems we have also implemented a C++ version of the algorithm for parallelization purpose. By comparing our Python version to the sequential C++ one, we observe at least a two-qubit advantage, which enables us to work with larger qubit systems. We observe in this graph that this advantage is significantly increased with the muti-threaded version.

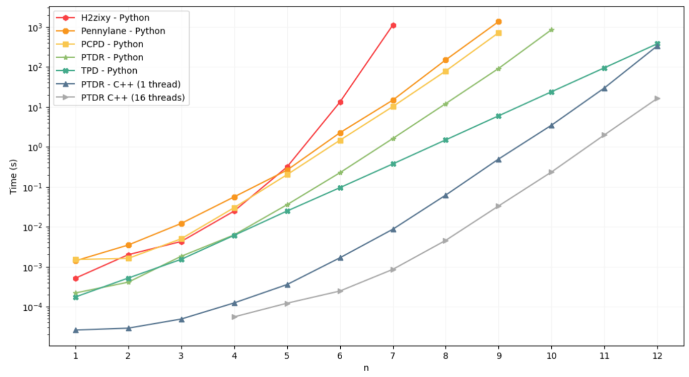


Fig. 3. Execution time for computing the decomposition in the Pauli basis using the serial codes *H2zixy*, Pennylane, *PauliComposer* (PCPD), *Tensorized Pauli Decomposition* (TPD) and the serial and multi-threaded code Pauli Tree Decomposition Routine (PTDR).

B. Multi-threaded code

We have also developed a multi-threaded version of our algorithm to accelerate the Pauli decomposition and address more qubits. In our algorithm, we split the Pauli tree into a forest of several Pauli trees by cutting all the branches at a certain level. Each tree of the forest is then executed on one thread independently from the others. This parallelization does not introduce any additional error and approximation. Each thread handles an independent subpart of the tree and all results are independent. Therefore we can use the same algorithm as for the whole tree for each sub-tree and the accuracy is not impacted. Moreover, there is no need to store the input matrix. If it is possible to guess the value for each entry (for example from a function) then it is still possible to perform the decomposition. The overhead will only depend on the cost of the function, as compared to a memory access. Therefore this method is still compatible with huge matrices that we do not want to store in memory. Figure 4 evaluates

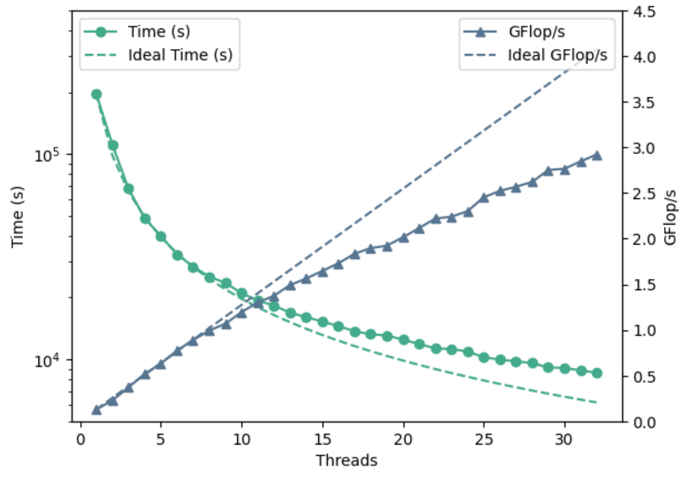


Fig. 4. Execution time and Gflop/s, depending on the number of threads (15 qubits).

the strong scaling of the multi-threaded C++ version of our algorithm. We observe that the speed grows linearly with the number of threads (the dotted lines correspond to an ideal speedup equal to the number of threads). For these experiments, the number of qubits is $n = 15$ (i.e., dense matrices of size 32768×32768 in complex arithmetic) to be able to achieve the computation on a single node, and the number of Pauli trees in the forest is $4^4 = 256$. Note that to our knowledge there are no existing parallel version for the other algorithms mentioned in Figure 3.

C. Comments on memory and possible multi-node version

1) Memory footprint analysis: After numerical experiments, TPD execution time seems to scale in $O(4^n)$ while our algorithm is still in $\mathcal{O}(8^n)$. However, it is interesting to study how TPD and PTDR behave in term of memory needed. In the following we do not take into account the memory needed to store the input matrix and the output results (which are equivalent for both algorithms) and we only focus on the memory needed to compute the decomposition of a matrix of size $2^n \times 2^n$.

a) Pauli Tree Decomposition Routine (PTDR): In the PTDR algorithm the computation is performed in place with two arrays of both size 2^n, as explained in Section III. Therefore the memory needed to perform the decomposition using PTDR is $\mathcal{O}(2^n)$. Note that in practice, one of this array contains integers while the other one contains booleans.

b) Tensorized Pauli Decomposition (TPD): To our knowledge the computation is performed by working recursively on submatrices defined from the input matrix. This leads to use $\mathcal{O}(2^{2n}) = \mathcal{O}(4^n)$ memory-space (see code provided in [12]). Note that in practice all the elements are double complex.

2) Theoretical extrapolation in a multi-node environment: In this paragraph we extrapolate the behavior of the PTDR and TPD algorithms using several nodes with the following hypotheses:

- we suppose that we have a cluster with a fixed RAM per node (64 Go, 128Go, 256Go) and 128 threads per node.
- we suppose that we have a parallel distributed version of PTDR (presently multi-threaded in C++) and TPD (presently sequential in Python).

Note that we do not consider the communication cost but the smaller data structures handled by PTDR (see the section before) should provide an advantage on this side.

The purpose of this analysis is to see in which constrained HPC environments PTDR outperform TPD, and vice-versa. The first constraint is a time budget (where TPD scales better) and the second constraint is the number of nodes accessible (PTDR scales better in memory footprint). We display the result for 64Go, 128Go and 256Go per node (with 128 threads per node) in Figure 5.

In Figure 5 the blue zone corresponds to the situation where PTDR is advantageous and can compute the decomposition for higher number of qubits than TPD. The red zone corresponds to the opposite statement. TPD scales better for short amount of time. However, the PTDR algorithm scales better in terms of

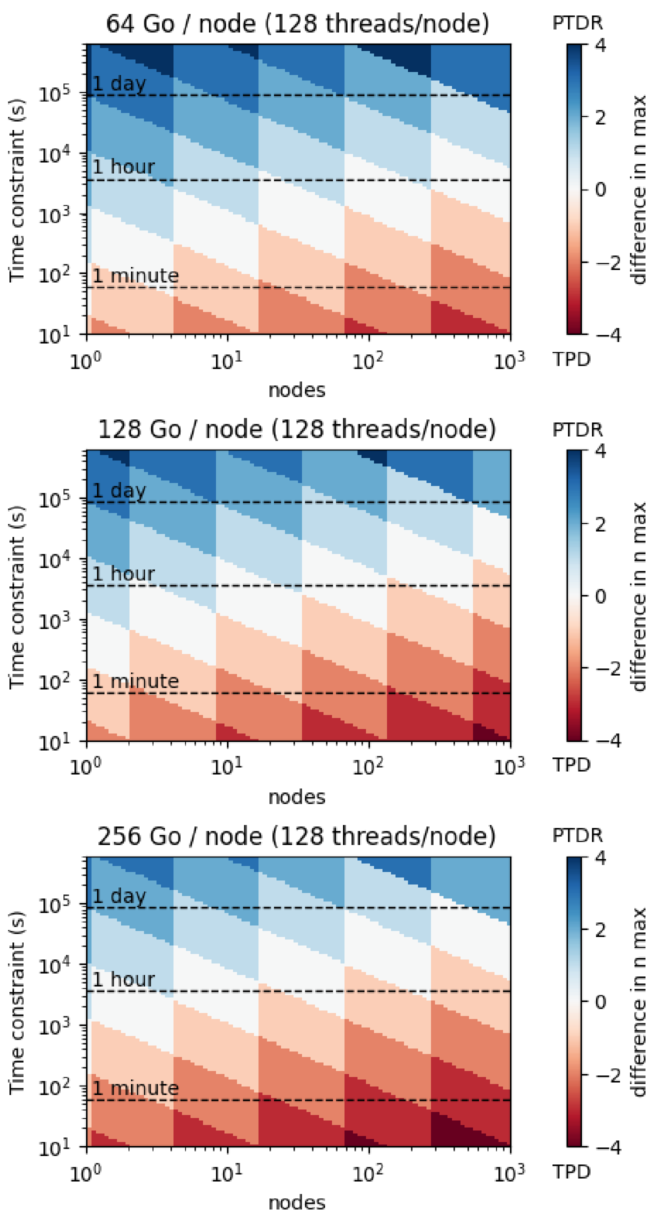


Fig. 5. Difference in the maximum number of qubits n manageable under time and number of nodes constraints for PTDR and TPD. The blue and the red regions correspond to the constraints where PTDR and TPD are advantageous, respectively.

memory footprint and thus can better exploit the multi-node architecture to outperform TPD in longer tasks. We remind that, the plot provided in Figure 5 does not take into account the communication cost, which should be higher for TPD because of the amount of data transferred.

To summarize, the PTDR algorithm seems to be adapted to HPC using multi-node systems thanks to low cost in memory and communication while TPD could prevail for smaller cases because of a smaller time complexity.

VI. Application to quantum computing

A. Preliminary notions

In quantum computing, we handle qubits instead of bits so that we can exploit the properties of quantum systems (superposition, entanglement,...). A 1-qubit quantum state $|\psi\rangle$ (using the Dirac notation) is a unit vector of $\mathbb{C}^2$ that can be expressed as

$$|\psi\rangle = \alpha \begin{pmatrix}1\\0\end{pmatrix} + \beta \begin{pmatrix}0\\1\end{pmatrix} = \alpha |0\rangle + \beta |1\rangle ,$$

with $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.

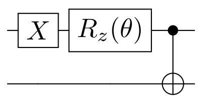
Here, the quantum state $|\psi\rangle$ corresponds to the superposition of the two basis states $|0\rangle$ and $|1\rangle$ of respective amplitude α and β. We can combine single qubits using tensor products to create an n-qubit quantum state which is a unit vector of $(\mathbb{C}^2)^{\otimes n} = \mathbb{C}^{2^n}$ [19].

All operations performed on qubits are unitary (and then linear, norm-preserving, and reversible), except for the operation of measurement that projects the quantum state on a basis state. These operations can be represented as quantum gates in a quantum circuit. There exist 1-qubit gates (acting on one qubit only) such as the Pauli X, Y and Z but also rotation gates like the rotation Z gate (rotation through angle θ around the z-axis).

$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} .$$

There are also multi-qubit gates, such as the controlled-NOT gate (CNOT) which creates entanglement between two qubits. In the CNOT gate, the second qubit is "controlled" by the first one in the sense that we apply a bit-flip (X gate) on the second qubit if the first one is $|1\rangle$ and remains unchanged otherwise. This notion can be generalized to a controlled-U operation where U is an operator acting on a given number of qubits, as we will see in Section VI-D.

The quantum gates can be combined to create quantum circuits acting on one or several qubits. In Figure 6 is given an example of a quantum circuit acting on two qubits. The wires correspond to the different qubits. The time evolves from the left to the right. The first qubit undergoes an X gate then a $R_z(\theta)$. Finally, a $CNOT$ gate acts on both qubits, the first qubit (with the dark circle) controls the application of a NOT operation (the large crossed circle) on the second one. Such quantum circuits can be used to create more complex quantum algorithms.



$$\equiv \quad (CNOT)(R_z(\theta) \otimes I)(X \otimes I)$$

Fig. 6. Example of a 2-qubit quantum circuit.

B. Quantum block-encoding

Manipulating dense or sparse matrices in quantum algorithms is essential to address practical applications. However quantum computers handle only unitary matrices and thus encoding techniques must be provided. The block-encoding [10] technique enables us to load matrices into the quantum memory by embedding a non-unitary matrix into a unitary one to call it in a quantum circuit. Namely, a general matrix $A \in \mathbb{C}^{N \times N}$ (with $N = 2^n$) is encoded in a unitary matrix U_A as

$$U_A = \begin{bmatrix} A & \cdot \\ \cdot & \cdot \end{bmatrix}, \tag{1}$$

When we apply U_A to $|0\rangle_a |\psi\rangle_d$ (as a common notation, we omit the tensor product operator between $|0\rangle_a$ and $|\psi\rangle_d$), where $|0\rangle_a$ corresponds to the ancilla (auxiliary) qubits and $|\psi\rangle_d$ corresponds to the data qubits. Then we get

$$U_A \left(|0\rangle_a |\psi\rangle_d\right) = |0\rangle_a A |\psi\rangle_d + \cdots . \tag{2}$$

So if when we measure the ancilla qubits we obtain $|0\rangle_a$ then we have applied the matrix A to the data state $|\psi\rangle_d$. On the contrary, if something else is measured, then we do not have applied A to the data qubits, so we need to perform the quantum circuit again. There also exists a definition of an approximate block-encoding. Given an n-qubit matrix A (with $N = 2^n$), if we can find $\alpha, \epsilon \in \mathbb{R}_+$ and an $(m+n)$-qubit unitary matrix U_A so that

$$\|A - \alpha\ \left(\langle 0^m| \otimes I_N\right)\ U_A\ \left(|0^m\rangle \otimes I_N\right)\|_2 \leq \epsilon, \tag{3}$$

then U_A is called an (α, m, ϵ)-block-encoding of A. When $\epsilon = 0$ the block encoding is exact and U_A is called an (α, m)-block-encoding of A. The value m corresponds to the number of ancilla qubits needed to block encode A. Note that a unitary matrix is a (1,0,0)-block encoding of itself [3].

C. Block-encoding using Pauli decomposition

In the remainder, we assume that A is Hermitian since we can always obtain a Hermitian matrix using the augmented matrix

$$\tilde{A} = \begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix}.$$

The goal is to get a quantum block-encoding of the matrix A. We use the method given in [1], [5], [10] for implementing linear combinations of unitary operators on a quantum computer. With this method, we create a block-encoding of A from its Pauli decomposition, given that Pauli operators are all unitary.

Let us consider a Hermitian matrix $A \in \mathbb{C}^{2^n \times 2^n}$ decomposed in $M = 2^m$ Pauli operators

$$A = \sum_{i=0}^{M-1} \alpha_i V_i$$

where $V_i \in \mathcal{P}_n$ and the α_i's are non-zero **real** numbers (since H is Hermitian). This task is performed on a classical computer using our PTDR algorithm.

To apply the matrix A to the n-qubit input quantum state $|\psi\rangle_d$, we need to:

1) allocate m ancilla qubits and prepare the state

$$|\alpha\rangle_a = \frac{1}{\sqrt{\sum_i |\alpha_i|}} \sum_i \sqrt{|\alpha_i|}\, |i\rangle_a$$

as a superposition of the basis states $|i\rangle_a$ of $\mathbb{C}^{2^m}$,

2) apply V_i to the data state $|\psi\rangle_d$, controlled by the ancilla qubits in state $|i\rangle_a$,

$$|i\rangle_a |\psi\rangle_d \rightarrow |i\rangle_a V_i |\psi\rangle_d ,$$

3) unprepare step 1.

Proof. Let us prove that the previous instructions lead to a block-encoding of the matrix $A = \sum_{i=0}^{M-1} \alpha_i V_i$, where here we consider without loss of generality that $\alpha_i > 0$ (as the sign of the coefficient α_i can be absorbed by the unitary V_i [4]). For this purpose, we are going to split the computation into two parts: on one side we will look at steps 1 and 2 together, and on the other side step 3, and the projection on $\langle 0|_a$.

- After the data and ancilla registers have been initialized, we have the $(n+m)$-qubit state

$$\frac{1}{\sqrt{\sum_i \alpha_i}} \sum_i \sqrt{\alpha_i}\, |i\rangle_a |\psi\rangle_d .$$

At the end of step 2, we get

$$\frac{1}{\sqrt{\sum_i \alpha_i}} \sum_i \sqrt{\alpha_i}\, |i\rangle_a V_i |\psi\rangle_d .$$

- Projecting the ancilla preparation inverse on $\langle 0|_a$ corresponds to

$$\frac{1}{\sqrt{\sum_i \alpha_i}} \sum_i \sqrt{\alpha_i}\, \langle i|_a .$$

Now, if we combine both previous results we get:

$$\left(\frac{1}{\sqrt{\sum_i \alpha_i}} \sum_i \sqrt{\alpha_i}\, \langle i|_a\right) \left(\frac{1}{\sqrt{\sum_i \alpha_i}} \sum_i \sqrt{\alpha_i}\, |i\rangle_a V_i |\psi\rangle_d\right)$$
$$= \frac{A}{\sum_i \alpha_i} |\psi\rangle_d ,$$

therefore we have block-encoded A. □

D. Quantum circuit and complexity

In Figure 7 is given an example of the quantum circuit that block-encodes a matrix when we have 4 terms in its Pauli decomposition (m=2). The top wire corresponds to the data qubits already initialized in quantum state $|\psi\rangle$. The two other wires correspond to the ancilla qubits used for the encoding. First, the state $|\alpha\rangle_a$ is prepared on the ancilla qubits. Then the application of the Pauli operators V_{00}, V_{01}, V_{10} and V_{11} on the data qubits is controlled by the value of the ancilla qubits. A white circle means we control with $|0\rangle$ while the black one means we control with $|1\rangle$. In the end, the state preparation on the ancilla qubits is undone.

Currently, quantum computing is in its noisy intermediate-scale quantum (NISQ) era. NISQ quantum processors are characterized by a qubit count that does not exceed thousands, coupled with a low tolerance to errors. In such a context, the

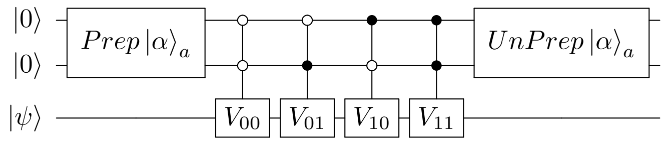


Fig. 7. Quantum circuit for the block-encoding (m=2).

previously introduced block-encoding algorithm is compatible in terms of number of qubits used as it scales logarithmically with the data size. However, it does not align in terms of circuit depth, leading to a low fidelity because of successive errors [26], [27].

Consequently, it seems more relevant to consider such a quantum algorithm from a Large Scale Quantum (LSQ) perspective, where quantum processors operate with error-corrected qubits. In this context, the complexity of the quantum circuit depends mainly on the number of T gates (because the Clifford gates are considered as having no cost on these devices [9], [14]).

By using the Kerenidis-Prakash tree techniques [16] as a state preparation, and pattern-rewriting in Clifford+T circuits [15], [19], [24], this algorithm has a complexity in T-count of $\mathcal{O}\left(2^m(nm + \mathrm{polylog}(1/\epsilon)\right)$, to block-encode a matrix of size $2^n \times 2^n$, where we have 2^m non-zero terms and an ϵ-approximation of the rotations gates. The time complexity of the execution of the block encoding on an LSQ device would be proportional to the T-gate complexity. If we compare the T-gate complexity ($\mathcal{O}\left(2^m(nm + \mathrm{polylog}(1/\epsilon)\right)$) of this block-encoding technique with the complexity of the decomposition in the Pauli basis ($\mathcal{O}(8^n)$), we expect the most expensive task to be the Pauli decomposition on large problem instances. Therefore, reducing the cost of the Pauli decomposition is essential to make this block encoding technique affordable. Our algorithm enables us to block-encode larger matrices in a reasonable amount of time compared to the state-of-the-art.

VII. Conclusion

We have proposed a new algorithm for decomposing a generic matrix into Pauli operators (which are tensor products of Pauli matrices). This algorithm uses a tree-based approach to reduce the number of arithmetical operations. We have developed a scalable multi-threaded C++ code that enables us to address moderate size problems (15 qubits, i.e., dense matrices of size 32768×32768 in complex arithmetic) on a single node using a limited memory footprint. We have also explained in a theoretical scaling analysis that our algorithm is a promising basis for a future multi-node version. As an application, we have described how this decomposition can be used to encode a Hermitian matrix into a quantum memory. In this situation our Pauli decomposition routine is used as a preprocessing phase for the quantum algorithm involving this matrix.

References

[1] Dominic W. Berry, Andrew M. Childs, Richard Cleve, Robin Kothari, and Rolando D. Somma. Simulating hamiltonian dynamics with a truncated taylor series. *Physical Review Letters*, 114(9), 2015.

[2] Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yigit Subasi, Lukasz Cincio, and Patrick J. Coles. Variational quantum linear solver. *arXiv preprint arXiv:1909.05820*, 2020.

[3] Daan Camps, Lin Lin, Roel Van Beeumen, and Chao Yang. Explicit quantum circuits for block encodings of certain sparse matrices. *arXiv preprint arXiv:2203.10236*, 2022.

[4] Shantanav Chakraborty. Implementing any linear combination of unitaries on intermediate-term quantum computers. *arXiv preprint arXiv:2302.13555*, 2023.

[5] Andrew M. Childs, Robin Kothari, and Rolando D. Somma. Quantum algorithm for systems of linear equations with exponentially improved dependence on precision. *SIAM Journal on Computing*, 46(6):1920–1950, 2017.

[6] Alexis De Vos and Stijn De Baerdemacker. The decomposition of an arbitrary $2^w \times 2^w$ unitary matrix into signed permutation matrices. *Linear Algebra and its Applications*, 606:23–40, 2020.

[7] Eviden Quantum Lab. myQLM: Quantum Computing Framework, 2020-2024.

[8] R.P. Feynman, R.B. Leighton, M. Sands, and EM Hafner. *The Feynman Lectures on Physics; Vol. I*, volume 33. AAPT, 1965.

[9] Austin G. Fowler and Craig Gidney. Low overhead quantum computation using lattice surgery. *arXiv preprint arXiv:1808.06709*, 2019.

[10] András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 2019.

[11] D. Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. In *Proceedings of Symposia in Applied Mathematics*, volume 68. American Mathematical Society, Washington, D.C., Oct 2010. arXiv:0904.2557.

[12] Lukas Hantzko, Lennart Binkowski, and Sabhyata Gupta. Tensorized Pauli decomposition algorithm. *arXiv preprint arXiv.2310.13421*, 2023.

[13] W. Heisenberg. Zur Theorie des Ferromagnetismus. *Z. Phys.*, 49(9-10):619–636, 1928.

[14] Dominic Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14(12):123011, 2012.

[15] Hu Jiang, Wang Pengjun, and Zhang Qiaowei. Synthesis of quantum circuits by multiplex rotation gates. *IEICE Electronics Express*, 13, 02 2016.

[16] Iordanis Kerenidis and Anupam Prakash. Quantum recommendation systems. *arXiv preprint arXiv:1603.08675*, 2016.

[17] Seth Lloyd. Universal quantum simulators. *Science*, 273(5278):1073–1078, 1996.

[18] Charles F.Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[19] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[20] PennyLane. https://pennylane.ai/.

[21] Rocco Monteiro Nunes Pesce and Paul D. Stevenson. H2ZIXY: Pauli spin matrix decomposition of real symmetric matrices. *arXiv preprint arXiv:2111.00627*, 2021.

[22] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[23] Sebastián V. Romero and Juan Santos-Suárez. Paulicomposer: Compute tensor products of pauli matrices efficiently. *arXiv preprint arXiv:2301.00560*, 2023.

[24] Neil J. Ross and Peter Selinger. Optimal ancilla-free Clifford+T approximation of z-rotations. *Quantum Information and Computation 16 (11-12)*, 16(11–12):901–953, 2016.

[25] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. The variational quantum eigensolver: A review of methods and best practices. *Physics Reports*, 986:1–128, 2022.

[26] A. Kandala K. X. Wei, S. Srinivasan, E. Magesan, S. Carnevale, G. A. Keefe, D. Klaus, O. Dial, and D. C. McKay. Demonstration of a high-fidelity CNOT for fixed-frequency transmons with engineered ZZ suppression. *Physical Review Letters*, 127(13), 2021.

[27] Tianyu Xie, Zhiyuan Zhao, Shaoyi Xu, Xi Kong, Zhiping Yang, Mengqi Wang, Ya Wang, Fazhan Shi, and Jiangfeng Du. 99.92%-fidelity CNOT gates in solids by filtering time-dependent and quantum noises. *arXiv preprint arXiv:2212.02831*, 2022.