Parallel Algorithms for Inner Product of Stabilizer States

Andi Lin[1] and Ricardo Rivera[2]

[1] *University of Utrecht*
[2] *University of Amsterdam*

March 25, 2021

Abstract

We adapt the ideas of [1] and apply parallel algorithmic techniques to two out of the three algorithms required for the computation of the inner product between two stabilizer states. Our dense algorithms show that it is possible to achieve $3\times$ speedup over a sequential algorithm in the regime where the ratio of non-zero elements in the rectangular matrix is $\gg 1/1000$. We also show that for dense parallel algorithms it is impossible to achieve greater speedups by using more processors. We showcase a sparse implementation of one of the algorithms and show that as long as the number of non-zero elements is $\ll 1/100$ we can achieve a $2.5\times$ speedup over its corresponding dense implementation. Finally, we note that unlike the dense algorithms, sparse implementations, will always incur greater speedups when using a greater number of processors.

1 Introduction

Quantum computers have received much attention over the last 20 years, as they have shown to efficiently solve NP-hard problems like prime factorization [2], or obtaining quadratic speedups when searching through an unordered database [3]. These algorithms require thousands of error-corrected qubits, while, as of today, most quantum computers only possess around 200 unstable qubits. It is therefore currently impossible to experimentally test such algorithms. The alternative is then to create simulators in order to test some aspects of these and other quantum algorithms. Several simulators [4, 5, 6, 7, 8] have already been developed, but there is no one better than the other, as they all face constraints. [1] used the ideas of [8] to develop a set of algorithms capable of computing the inner product between two stabilizer states in time $\mathcal{O}(n^2)$. In this project, we combine our knowledge of parallel algorithms plus the simulator of [8] to accelerate the computation of the inner product.

2 Preliminaries

This section serves as an introduction to the notation and mathematical tools of the field of quantum computation. We assume the reader is already familiar with intermediate linear algebra concepts. The most relevant parts for the understanding of the parallel algorithm are the sentences in the form of definitions, propositions, theorems, and remarks. If the reader requires or desires more information about the subject, we recommend the book *Quantum Computation and Quantum Information* [9].

2.1 Quantum States & Quantum Gates

Definition 1 (Quantum State)**.** *In quantum mechanics, a quantum state is a complex vector* $|\psi\rangle$ *in a Hilbert space* $\mathcal{H}$ *that has magnitude one.*

Here, the symbol $|\cdot\rangle$, denotes a column vector and $\langle\cdot|$, the corresponding row vector. We can obtain one from the other by transposing and conjugating the vector (jointly denoted by the symbol †), so $|\psi\rangle^\dagger = \langle\psi|$.

We usually work with a specific orthonormal basis of this Hilbert space, called the **computational basis**, where $\mathcal{H} = \mathbb{C}^\Sigma$ and the basis vectors are given by $|i\rangle_{i\in\Sigma}$. For example, a d-dimensional Hilbert space has $\Sigma = \{0, \dots, d-1\}$, and so a general state $|\psi\rangle$ in this space can be written as

$$|\psi\rangle = \sum_{i=0}^{d-1} c_i |i\rangle, \quad \sum_{i=0}^{d-1} \|c_i\|^2 = 1, \tag{2.1}$$

where the constraint of the coefficients is made to ensure that the magnitude of the vector is in fact unity.

Just like any other vectors, we can consider the inner product between two states in the same Hilbert space. If $|\psi\rangle = \sum_i b_i |i\rangle$ and $|\phi\rangle = \sum_j c_j |j\rangle$ are two states in $\mathcal{H}$, then the inner product is given by $\langle\phi|\psi\rangle = \langle\psi|\phi\rangle = \sum_i \sum_j b_i^* c_j \langle i|j\rangle$. And since the computational basis is orthonormal, we have that $\langle i|j\rangle = \delta_{i,j}$. Loosely speaking, the inner product can be thought of as a measure of how similar two states are, and also plays an important role on measurements of quantum systems which we do not discuss here. As will be described in more detail later on, our goal is to parallelize some of the algorithms described in [1] which are related to the computation of the inner product between two **stabilizer** or **classical** states (defined in Section 2.2).

Another relevant operation for quantum states (and operators) is the outer or **tensor** product, which helps express composite systems in a bigger Hilbert space. Unlike the inner product, the tensor product can be performed with Hilbert spaces of different size. For example, if we consider $|\psi\rangle \in \mathbb{C}^2$ and $|\phi\rangle \in \mathbb{C}^3$ then the tensor product of these is

$$|\psi\rangle \otimes |\phi\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \otimes \begin{pmatrix}\gamma\\ \delta\\ \epsilon\end{pmatrix} = \begin{pmatrix}\alpha\gamma & \alpha\delta & \alpha\epsilon & \beta\gamma & \beta\delta & \beta\epsilon\end{pmatrix}^\top.$$

For this project we will only work with **qubits**, composite quantum systems where each

individual **qubit** 'lives' in the Hilbert space $\mathcal{H} = \mathbb{C}^2$. If we tensor n of these together, making a system of n qubits, then these together are vectors in the space $\mathbb{C}^{2^n}$ and the statevector is

$$|\psi\rangle = \begin{pmatrix} c_0 \\ \vdots \\ c_{2^n} \end{pmatrix}. \tag{2.2}$$

Remark 1. *The statevector representation consists of 2^n complex coefficients,and so the number of coefficients needed to represent the state increases exponentially in the number of qubits. Clearly, this is a complication when trying to simulate the general behavior of a quantum computer. However, as we discuss in Section 2.2, there exists a certain class of states which we can simulate efficiently in a classical computer.*

Now that we have talked about quantum states, we discuss how to change them and obtain different ones. We are looking for linear operators that transform quantum states to other quantum states, i.e. operators that preserve the norm of vectors. These are the unitary operators.

Definition 2 (Unitary Operators)**.** *A **unitary operator** U is a linear operator $U : \mathcal{H} \to \mathcal{H}$ that respects the relation $U^\dagger U = UU^\dagger = I$.*

One can see that these preserve the norm of vectors by considering the state $|\phi\rangle = U|\psi\rangle$ where $|\psi\rangle$ is any other valid state. This state has norm $\langle\phi|\phi\rangle = \langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle = 1$. From the definition, it is also clear that we are allowed to multiply any of these operators by a complex phase or **global phase** to modify the behavior of a gate.

Just like with classical computers, we can arrange these operators or **gates** into circuits, thus having a compact and pictorial representation on the operators that act on the quantum state. In this report, we make use of 6 different single-qubit gates $\{H,S,I,X,Y,Z\}$, and 2 double-qubit ones $\{CX,CZ\}$. The complete behavior of the gates is not relevant to us, and so we only discuss some important relationships among these. Here, we discuss the general behavior of the **Pauli operators** $P = \{I, X, Y, Z\}$. These follow the relations

$$Y = iXZ \quad \text{and} \quad X^2 = Y^2 = Z^2 = I. \tag{2.3}$$

It is possible to create n-qubit gates out of these by tensoring n of these together in any combination. For example, a 3-qubit Pauli gate (with a global phase) could be $X \otimes Y \otimes -X = -(X \otimes Y \otimes X)$. These operators are key elements of the n-qubit Pauli group, denoted by $\mathcal{P}_n$.

Definition 3 (Pauli Group)**.** *The n-qubit **Pauli Group** is defined as*

$$\mathcal{P}_n \coloneqq \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}^{\otimes n}. \tag{2.4}$$

2.2 Stabilizer States

In the previous section, we mentioned that in order to track the behavior of a general state through a circuit, we are required to store 2^n complex coefficients of the statevector, therefore

requiring an exponential amount of memory depending on the number of qubits of the system. However, [8] found a class of states that can be efficiently simulated in a classical computer using $\mathcal{O}(n^2)$ memory. This result is better known as the **Gottesman-Knill** theorem. In the same paper, they also provide a $\mathcal{O}(n^3)$ time algorithm to simulate such states (the one we use in this report to create the inputs for the parallel algorithms), but it was quickly revised by [7] to a time complexity of $\mathcal{O}(n^2)$.

Theorem 1 (Gottesman-Knill)**.** *Any n-qubit state $|\psi\rangle$ that is generated only through S, H, CX, and CZ gates can be efficiently simulated in a classical computer.*

Gottesman realized that states created only by these gates could be uniquely represented by keeping track of n elements of $\mathcal{P}_n$ that stabilize a state.

Definition 4 (Stabilizer)**.** *We say a unitary U **stabilizes** a state $|\psi\rangle$ if $U|\psi\rangle = |\psi\rangle$.*

The n elements of $\mathcal{P}_n$ that stabilize a single state $|\psi\rangle$ form a subgroup, denoted $S(|\psi\rangle)$, and it has the following properties.

Proposition 1. *The following are true for all elements of $S(|\psi\rangle)$:*

1. *They commute with one another.*
2. *The global phases can only be $+1$ or -1.*

Now we can see that keeping track of $S(|\psi\rangle)$ in a classical computer only takes $\mathcal{O}(n^2)$, as every n-qubit Pauli operator in $S(|\psi\rangle)$ requires $2n+1$ bits (letting $I = 00$, $X = 01$, $Z = 10$, $Y = 11$ and 1 bit for the global phase).

For our algorithm, we want to be able to write $S(|\psi\rangle)$ into a matrix, we can achieve this by using the following propositions.

Proposition 2. *Any n-qubit Pauli operator can be written as*

$$P(z, x) = i^{-z\cdot x} Z^z X^x, \tag{2.5}$$

where z and x are elements of $\mathbb{F}_2^n$, the finite field of two elements (i.e. a binary field) of dimension n. And $z \cdot x = \sum_i^n z_i x_i$.

Thus, we can think of any n-qubit Pauli operator as a vector over $\mathbb{F}_2^n$ with entries z, x. Simply, z and x are binary strings or **bitstrings** that indicate in which positions of the tensor product there is a single-qubit Pauli Z or an X. If both z and x have a 0 on the same position, this refers to the identity operator, while if they both have a 1 we know from Eq. (2.3) that this refers to a Y. For example, $P(011, 010) = i^{-011\cdot 010}(I \otimes Z \otimes Z)(I \otimes X \otimes I) = i^{-1}(I \otimes ZX \otimes Z) = I \otimes Y \otimes Z$.

Proposition 3. *In this notation, multiplication of two Pauli operators is equivalent (up to a phase) to the sum of their z and x bitstrings. The complete equivalence is given by*

$$P(z, x)P(z', x') = i^{z'\cdot x - x'\cdot z} i^{(z'+z)\cdot(x'+x)-y} P(z' + z, x' + x), \tag{2.6}$$

*where $y = (z' + z) \cdot (x' + x)$ but the sum of bitstrings here is not taken over $\mathbb{F}_2^n$. The exponent of the imaginary units is called the **symplectic product**.*

Using the binary representation of the n-qubit Pauli operators as in Eq. (2.5), we can write each element of $S(|\psi\rangle)$ as the binary string $(z|x)$ of length $2n$. Since $S(|\psi\rangle)$ contains n Pauli operators we can stack each string on top of each other yielding a $n \times 2n$ matrix or **tableau**, plus a vector of length n corresponding to the global phase of each Pauli operator (row). For example, the state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is stabilized by the operators $-(Y \otimes Y)$ and $X \otimes X$, and thus its tableau (plus phase vector) is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

3 Dense Algorithms

In this section we discuss the dense implementation of the first two algorithms of [1]. We do not parallelize the last algorithm, as this resembles Algorithm 2 and the results of the inner product computation are only relevant in the scope of quantum computing, not in parallel algorithms. We also note that for the sake of clarity, and space, the vector of global phases will not be incorporated in the pseudo-code, but will be discussed inside the text.

3.1 Algorithm 1: Canonical Form

We begin this section with the following definition.

Definition 5. *Let M be a $n \times 2n$ stabilizer tableau. M' is called a **canonical reduced form** of M if both the right and left halves of M' is in row-reduced echelon form and is obtained from M by row swap and row addition operations.*

The goal of this algorithm is to transform a tableau to its *canonical form*.[1] Before discussing the parallel algorithm, first, we discuss the sequential algorithm on how to achieve this goal. Consider the following $n \times 2n$ matrix, $\mathcal{A}$, split into left and right halves. We will first focus our attention on the right half.

$$\left(\begin{array}{cccc|cccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{array}\right) \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

The first step of the algorithm is to ensure that $\mathcal{A}_{ij}$ is a 1 (if it exist), while the second step is to remove the trailing zeroes across column j by performing row additions. Assuming that column j does not consist entirely of 0's, we increment i and continue the same process on the next iteration. If j consists entirely of $0's$, then we simply move on without increasing i. After completing the right half, we perform the exact same algorithm on the left half, continuing from where index i left off from the right half of the algorithm.

In essence, the sequential algorithm consists of three main operations: searching for index

[1] In quantum computing terms, the canonical form is the simplest stabilizer tableau to represent state $|\psi\rangle$. Recall that $S(|\psi\rangle)$ is an abelian subgroup under multiplication, so row additions and row multiplications do not change the state.

Algorithm 0: Canonical row reduction

```
Input  : A
Output : Canonical reduced form of A
⇒ ROWSWAP(R_i, R_k), swaps row i and k of A
⇒ ROWADD(R_i, R_k), add row i to row k of A
i ← 0
for j ∈ [n, ..., 2n − 1] do
    for l ∈ [i, ..., n − 1] do
        if A_lj = 1 then
            k ← l
            break
    end
    if k exist then
        ROWSWAP(R_i, R_k)
        for m ∈ [0, ..., n − 1], m ≠ i do
            if A_m,j = 1 then
                ROWADD(R_i, R_m)
        end
        i ← i + 1
end
```

k, row-swap and row addition. All these operations entail a natural way for parallelization.

Let $P = NM, N \geq M$ be the number of processors, where we assume N, M divides n, the number of rows in tableau $\mathcal{A}$. We will use the 2d-cyclic distribution, $\phi(i,j) = (\phi_0(i), \phi_1(j)) = (i \mod N, j \mod M)$, for our tableau. The phases are also distributed cyclically but only to the processors that satisfy p.i.d $div\ N = 0$. Next, we define $\mathcal{A}'$ to be the local matrix of $\mathcal{A}$ for each processor P and a mapping $\mu : \mathcal{A} \to \mathcal{A}'$ by $(i,j) \mapsto (i\ div\ N, j\ div\ M) = (\mu_0(i),\ \mu_1(j))$. Finally, we will abuse notation and denote $\mathcal{A}'_{(\mu_0(i),\ \mu_1(j))}$ by $\mathcal{A}'_{[i,j]}$.

The first four supersteps of Algorithm 1 parallelize the search and row-swap operation. Intuitively, we are searching for 1's concurrently along column j, and send the indices of the row where a one was found to the processor $P(\phi_0(i), \phi_1(j))$. This processor then chooses the minimum row index of the elements received to swap it's row with, and alerts all other processors that are involved in the swapping. In superstep 5, the column processors of column j will send their entries of column j across the processor row as *flags*, which indicate whether or not there is a trailing one along column j. Also, processors in column i send all of their entries to the processor column. In superstep 6, the row processors that received a flag perform row addition in order to remove the trailing one. As discussed in Eq. (2.5), to obtain the correct phases after row addition, we calculate the symplectic product. Similar to the sequential version, at the end of superstep 6, we increase the i, j counter by one. However, if no such k was found in superstep 1, then we only increase counter for j.

3.1.1 Cost and complexity

Let us begin by providing a complexity analysis for the sequential version. Since the left-half follows the exact same algorithm, thus it does not contribute to the complexity of the algorithm and hence we only focus on the right half of $\mathcal{A}$.

Following the algorithm, we first search for k (if it exist) and perform a row swap, this

Algorithm 1: Parallel Canonical Form Reduction

Input : $n \times 2n$ matrix, $\mathcal{A}$
Output : Canonical reduced form of $\mathcal{A}$
Data: $m = n \; div \; N, \; m' = 2n \; div \; M$
$s = \text{p.i.d} \mod N, \; t = \text{p.i.d} \; div \; N$

Superstep 1:
$i \leftarrow 0, \; j \leftarrow 0$
for $r \in [0, \ldots, m-1]$ **do**
  if $\mathcal{A}'_{[r,\ j]} = 1$ **then**
    $k_{(r,s)} \leftarrow r \cdot N + s$
    $P(\phi_0(k_{(r,s)}), \phi_1(j))$ **PUT** $k_{(r,s)}$ in $P(\phi_0(i), \phi_1(j)))$
    break
end

Superstep 2:
if $\phi_0(i) = s, \; \phi_1(j) = t$ **then**
  $k = \text{argmin}\{k_{(r,s)} \mid 0 \le r \le m-1, \; 0 \le s \le N-1\}$
  for $a \in [0, ..., P-1]$ **do**
    PUT k in P_a
  end

Superstep 3:
if $\phi_0(k) = s$ **then**
  $R_k \leftarrow \mathcal{A}'_{[k]}$
  Put R_k in $P(\phi_0(i), \star)$
if $\phi_0(i) = s$ **then**
  $R_i \leftarrow \mathcal{A}'_{[i]}$
  Put R_i in $P(\phi_0(k), \star)$

Superstep 4:

if $\phi_0(r) = s$ **then**
  $\mathcal{A}'_r \leftarrow R_i$
if $\phi_0(i) = s$ **then**
  $\mathcal{A}'_i \leftarrow R_k$

Superstep 5:
if $\phi_1(j) = t$ **then**
  for $i' \in [0, \ldots, m-1]$ **do**
    if $i' \cdot N + s \neq i$ **then**
      $flag_{i'} \leftarrow \mathcal{A}'_{[i,j]}$
      PUT $flag_{i'}$ in $P(\phi_0(i'), \star)$
  end
if $\phi_0(i) = s$ **then**
  $R_i \leftarrow \mathcal{A}'_{[i]}$
  PUT R_i in $P(\star, \phi_1(j))$

Superstep 6:
for $i' \in [0, \ldots, m-1]$ **do**
  if *flag*$_{i'}$ *= 1* **then**
    $\mathcal{A}'_{[i']} \leftarrow \mathcal{A}'_{[i']} + R_i$
end

incurs a complexity of $\mathcal{O}(n^2)$. Next, we loop across column j and perform row addition to remove trailing ones, and once again, this incurs a complexity of $\mathcal{O}(n^2)$. Finally, since there are n columns on the right half of $\mathcal{A}$, in total, we obtain a complexity of $\mathcal{O}(n^3)$.

Remark 2. *As we are implementing the algorithm in C, row-swapping for the sequential version is nothing more than a swap of pointers, hence, it is actually constant in complexity. This is important to note because in the parallel version, the processors may not always control the rows they are required to swap with, and so we require inter-processor communication.*

Next, we analyze the cost of our parallel algorithm under the BSP model. For simplicity, we disregard the computation needed for the mapping $\mu_0, \mu_1, \phi_0, \phi_1$. Let g denote time per data word and l the global synchronization time.

- In superstep 1, for each processor, we do m many iterations. In each iteration, we perform two computations for $k_{(r,s)}$ and send it. At the same time, $P(\phi_0(i), \phi_1(j))$ receives at most N many $k_{(r,s)}$. So we have a BSP cost of $2m + Ng + l$.
- In superstep 2, determining k is of order $\mathcal{N}$ and we are sending to P many processors. Resulting in a BSP cost of $N + Pg + l$.
- In superstep 3, the $R's$ is of size m' and we are sending to at most M many other processors. Resulting in a BSP cost of $Mm'g + l$.
- Superstep 4 consists of swapping pointers by Remark 2, and thus essentially 0 cost.
- In the first part of superstep 5, each processor performs 2 computations to determine the flag. And we note that there are at most m many flags for each processor. Finally, we send to at most M many other processors. For the second part, the size of R_i is once again m' and we send to at most N processors. In total, we have a BSP cost of $(2Mm + Nm')g + l$.
- Finally, for superstep 6, row addition incurs a cost of $\mathcal{O}(m')$ and each processor has to perform it at most m times. So we have in total, a BSP cost of $mm' + l$.

Adding up all the cost and factoring the n number of iterations each processor must perform, we have a total BSP cost of:

$$n \cdot ((2Mm + (N + M)m' + N + P)g + m(m' + 2) + N + 6l).$$

We can see that for each iteration, communication cost is of linear order, while computation cost is quadratic. Then, for small qubit systems (small n), it might be more costly to implement the parallel version than the sequential one. However, for large qubit systems (which we are interested in), the bulk of the cost falls onto the computation. As we increase the number of processors $P = NM$, we can see that $m \leq n,\ m' \leq 2n$ decreases, resulting in a cheaper cost when compared to the sequential version.

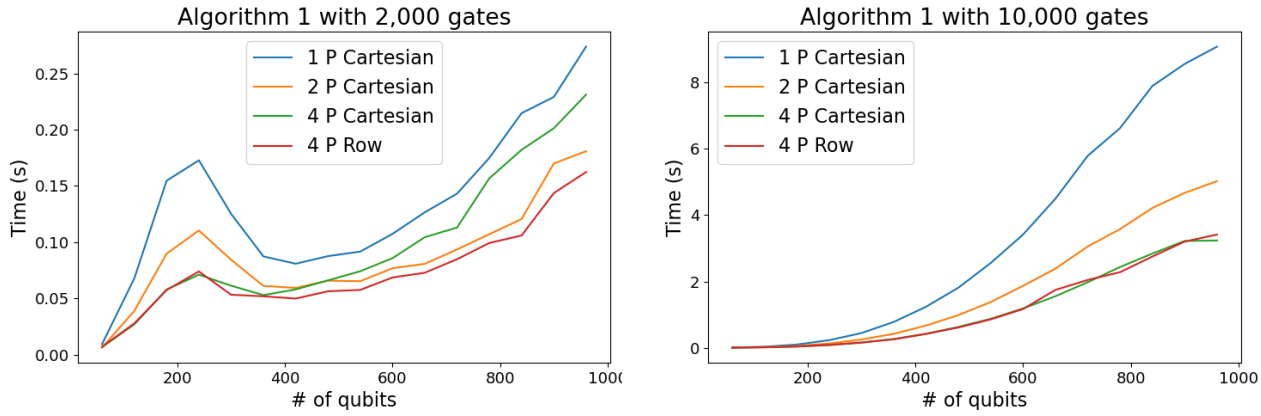

Figure 1: Runtime of Algorithm 1. Each data point was averaged over 40 iterations. See Section C for a benchmark of the computer where the data was taken.

3.1.2 Results & Discussion

Looking at the second graph of Fig. 1, we can see that for large systems, the quadratic factor completely dominates, whereas for small number of qubits, the runtime is essentially linear. Perhaps it is worth pointing out two more facts. First, is that even with just four processors, the graphs already exhibit the expected theoretic behaviour. Second, the advantage of parallelization is more prominent with greater number of qubits. The downside is that in this limit, we are restricted by the number of qubits that Ricardo's simulator[2] can handle. Combining these two facts, leads us to the conclusion that there is no real advantage in running the algorithm with more processors.

It is also important to point out that we verified all of our results with a simple Python implementation we designed.

3.2 Algorithm 2: Circuit Simulation and Basis State Reduction

Our next objective, is to obtain a minimal quantum circuit of the form $H-CX-CZ-S-H$, where each $-\cdot-$ stands for many gates of that type. The output of the previous algorithm serves as an input for this new algorithm, and the canonical form guarantees that we obtain the minimal circuit just described. Since we are now performing column operations, these operations do change the state $|\psi\rangle$ to a **computational basis state**, whose tableau only has ones on the left-hand side, like in Eq. (4.1).

The H, CX, CZ and S gates are unitaries that act on quantum states, but in terms of the tableau, these consist on column operations. The operation of each gate in the tableau can be summarized as follows:

- $CX(i,j) : (z_i, z_j, x_i, x_j) \mapsto (z_i \oplus z_j, z_j, x_i, x_j \oplus x_i)$. And $r = r \oplus x_i z_j \, (x_j \oplus z_i \oplus 1)$.
- $H(i) : (z_i, x_i) \mapsto (x_i, z_i)$. $r = r \oplus x_i z_i$.

[2]Part of Ricardo's master's thesis project involved developing the simulator described in [8], and so we used his simulator to create random tableaus that work as inputs for our parallel algorithm.

- $S(i) : (z_i, x_i) \mapsto (z_i \oplus x_i, x_i)$. $r = r \oplus x_i z_i$.
- $CZ(i, j) : (z_i, z_j, x_i, x_j) \mapsto (z_i \oplus x_j, z_j \oplus x_i, x_j, x_i)$. $r = r \oplus x_i z_i$.

Where i, j indicate the qubits (or columns in this case) to which we apply the gates. r refers to the global phase, and z, x refer to the left and right side of the tableau.

Now we will give the sequential algorithm and provide the ideas behind the parallel algorithm. We refer the readers to Section A for the more in depth detailed parallel algorithm. Finally, given any matrix $\mathcal{A}$, we use $\mathcal{A}_j^\top$ to indicate the j-th column of $\mathcal{A}$.

Algorithm 2: Basis state reduction

INPUT: Canonical reduced matrix $\mathcal{A}$
OUTPUT: Basis state of $\mathcal{A}$, Circuit $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$
$i \leftarrow 0$ ▷ Apply first block of H-gates
for $j \in [n, \ldots, 2n-1]$ **do**
  $k \leftarrow \text{argmin}[i, \ldots, n-1]$ such that $\mathcal{A}_{k,\ j} = 1$
  if k *exist* **then**
    ROWSWAP$(\mathcal{A}_i,\ \mathcal{A}_k)$
  else
    $k_2 \leftarrow \text{argmax}[i, \ldots, n-1]$ such that $\mathcal{A}_{k,\ j-n} = 1$
    if k *exist* **then**
      ROWSWAP$(\mathcal{A}_i,\ \mathcal{A}_{k_2})$
      if $\mathcal{A}_{i,j} = 1$ *or* $\mathcal{A}_{i,j-n} = 1$ **then**
        HCONJUGATE$(\mathcal{A}_j^\top,\ \mathcal{A}_{j-n}^\top)$
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{H_{j-n}\}$
  end
  $i \leftarrow i + 1$
end

We can observe from the sequential algorithm of the first block of H gates that the only new operation introduced is **HCONJUGATE**. By our update rule $(z_i, x_i) \mapsto (x_i, z_i)$, this is equivalent to swapping columns $\mathcal{A}_j^\top$ and $\mathcal{A}_{j-n}^\top$. However, since M divides n, the distribution is symmetric about the center of the matrix. This means that for any processor p, p controls both $\mathcal{A}'_{[i,\ j]}$ and $\mathcal{A}'_{[i,\ j-n]}$. Thus, the column swap operation in this case requires no communication and only incurs a cost of order $\mathcal{O}(m)$.

From the discussion above, one might ask how to perform column operations in general. It might be worthwhile to address this issue before we proceed. Due to the language architecture of C, it is unnatural to attempt to store a vector as a column vector, as it entails using a two-dimensional array. This also creates issues when attempting to send such vector across processors. However, a feasible workaround would be to have each processors update (and read off) the column entries of the global matrix $\mathcal{A}$ in parallel. We will use this scheme of *global matrix update* for all future column operations.
The operation **CXCONJUGATE** is again a column operation, and so we will use the *global matrix update* scheme to implement the mapping $(z_j, z_k, x_j, x_k) \mapsto (z_j \oplus z_k, z_k, x_j, x_k \oplus x_j)$ in parallel as follows:

- For every $i \in [0, \ldots, n-1]$, all processors update $\mathcal{A}_{i,k} \leftarrow \mathcal{A}'_{[i,k]}$ and $\mathcal{A}_{i,k-n} \leftarrow \mathcal{A}'_{[i,k-n]}$.

▷ APPLY CNOT GATES

for $j \in [n, \ldots, 2n-1]$ **do**
  for $k \in [j+1, \ldots, 2n-1]$ **do**
    if $\mathcal{A}_{j,k} = 1$ **then**
      CXCONJUGATE$(\mathcal{A}_j^\top,\ \mathcal{A}_k^\top)$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{CX_{i,j-n}\}$
  end
end

- If $\phi_1(j) = t$, then for every $i \in [0, \ldots, n-1]$, update $\mathcal{A}_{i,j} \leftarrow \mathcal{A}'_{[i,j]}$ and $\mathcal{A}_{i,j-n} \leftarrow \mathcal{A}'_{[i,j-n]}$.
- If $\phi_1(k) = t$, for each $i \in [0, \ldots, n-1]$, we update $\mathcal{A}'_{[i,k]} \leftarrow \mathcal{A}'_{[i,k]} \oplus \mathcal{A}_{i,j}$.

Remark 3. *It is important to note that without the update scheme, for each j, k, we would incur a communication cost of quadratic order just to perform the conjugation, this is extremely costly for large qubit simulation.*

▷ APPLY CZ-GATES

for $j \in [0, \ldots, n-1]$ **do**
  for $k \in [j+1, \ldots, n-1]$ **do**
    if $\mathcal{A}_{j,k} = 1$ *and* $\mathcal{A}_{j,k+n} = 0$ **then**
      CPCONJUGATE$(\mathcal{A}_j^\top,\ \mathcal{A}_k^\top)$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{CP_{i,j-n}\}$
  end
end

The parallel implementation of **CPCONJUGATE** is extremely similar to that of **CXCONJUGATE**, with only a few differences.

- In step 1 and 2, instead of $j-n, k-n$. We instead update the column index $j+n, k+n$.
- If $\phi_1(k) = t$, for each $i \in [0, \ldots, n-1]$, we update $\mathcal{A}'_{[i,j]} \leftarrow \mathcal{A}'_{[i,])} \oplus \mathcal{A}_{i,k+n}$ and $\mathcal{A}'_{[i,k]} \leftarrow \mathcal{A}'_{[i,k]} \oplus \mathcal{A}_{i,j+n}$

▷ APPLY S-GATES

for $j \in [n, \ldots, 2n-1]$ **do**
  if $\mathcal{A}_{j,j} = 1$ *and* $\mathcal{A}_{j,j-n} = 1$ **then**
    SCONJUGATE$(\mathcal{A}_j^\top)$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{P_{j-n}\}$
end

▷ APPLY H-GATES

for $j \in [n, \ldots, 2n-1]$ **do**
  if $\mathcal{A}_{j,j} = 1$ *and* $\mathcal{A}_{j,j-n} = 0$ **then**
    HCONJUGATE$(\mathcal{A}_j^\top)$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{H_{j-n}\}$
end

The **SCONJUGATE** and **HCONJUGATE** operations involve column addition and column swap between $\mathcal{A}'_{[i,\ j]}$ and $\mathcal{A}'_{[i,\ j-n]}$ according to our mapping rule, which we already know how to perform from the beginning of this algorithm. Finally, we implement row addition to remove the trailing zeroes below the diagonal of the left half of the matrix, which is similar to what we discussed in Algorithm 1 to complete the transformation of $\mathcal{A}$ to it's basis state.

3.3 Results & Discussion

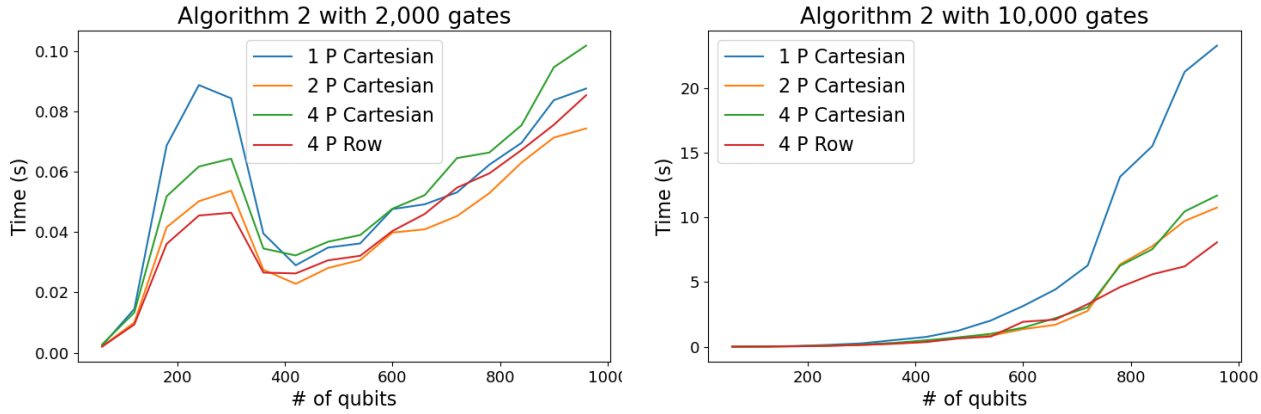

Figure 2: Algorithm 2 runtime.

Compared to Fig. 1, Fig. 2 has a similar shape which is expected as we are using the same parallel algorithm whenever we row swap and row add. The greatest difference between Algorithm 1 and Algorithm 2 is the introduction of column operations. However, due to the architecture of C, column swap or addition cannot be done in constant time but it incurs a $\mathcal{O}(m)$ complexity. That is why for large qubit simulation, Algorithm 2 is slower than Algorithm 1.

4 Sparse Algorithm

This section serves to showcase the advantages of a sparse algorithm for the problem we are trying to study. As this is only a showcase, we only discuss Algorithm 1 (without phases) and let the conclusions of this algorithm motivate future work and development. Too see why a sparse implementation would work, we first have to revisit how the tableaus are generated by the simulator described in [8]. To begin, we start from the state $|0\rangle^{\otimes n}$ that has tableau

$$I_n|0_n, \tag{4.1}$$

where I_n and 0_n are the identity matrix and the matrix of zeros of dimension n. That is, the tableau of the initial state contains n non-zero elements. As we run the randomly generated circuit with this state as the initial state, the tableau evolves according to the rules described in Section 3.2. Unfortunately, there are no scientific result involving the distribution of ones

in a tableau, but we can expect (highly) sparse systems when considering few gates and many qubits. We will assume this is the case for the rest of this section.

4.1 Implementation

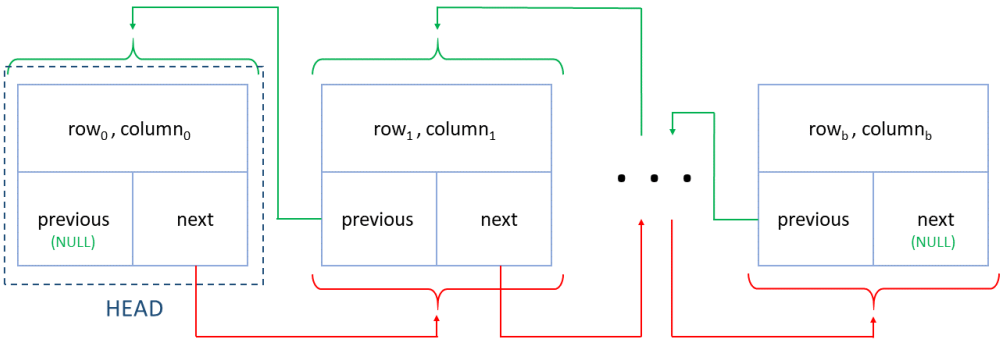

Figure 3: One-dimensional doubly-linked list with b nodes. Each node has two integer values *row* and *column*, and it represents a non-zero element of the global matrix $\mathcal{A}$ at position $row_i, column_i$. The **HEAD** node is the only one in the scope of the parallel function, while the remaining ones 'live' in the heap memory. Therefore, HEAD will require special treatment in the code so we do not accidentally delete it.

The setup for the sparse parallel algorithm is similar to that of Section 3.1, where we again distribute the data in a 2d-cyclic fashion, $(\phi_0(i), \phi_1(j)) = (i \mod N, j \mod M)$. The only difference is that now the processors will not create a local matrix $\mathcal{A}'$, but each processor builds a one-dimensional unordered doubly-linked list $\mathcal{L}$ like that of Fig. 3. By *unordered*, we mean that the elements of this list are not arranged in any way i.e a non-zero element in the first row and first column of the tableau might appear in any position of the list. In contrast to the local matrix, the linked list only takes into consideration the non-negative entries received, and tracks where these elements correspond to in $\mathcal{A}$. Therefore, there is also no need for the mapping $\mu : \mathcal{A} \to \mathcal{A}'$.

The sparse parallel algorithm follows the same superstep structure as the dense algorithm; however, the computations within each superstep are optimized. We will now give a short overview of the differences within each superstep. We let b be the total number of nodes (elements in the linked list) that each processor has, r_u the number of nodes with row value u, and c_v the number of nodes with column value v.

- Superstep 1. Instead of searching through all rows of $\mathcal{A}'$ to find a one in column j, here the processors now search through $\mathcal{L}$ to find any node that has column value j. This is faster, as we expect $b < m$.

- Superstep 2. No changes.
- Superstep 3. Processors $\phi_0(k) = s$ now send r_k elements, instead of m' ($r_k \ll m'$). Similarly, processors $\phi_0(i) = s$ now send r_i elements, instead of m'. The sparsity greatly decreases communication costs.
- Superstep 4. The processors involved in superstep 3 read less elements.
- Superstep 5. Processors $\phi_1(j) = t$ sends the same number of flags as in the dense implementation, but these our now found faster. Processors $\phi_0(i) = s$ again only put r_i elements instead of m'.
- Superstep 6. Since the processors that have to add rows received less data (from superstep 5), they will finish the computation faster.

The reader can find the code on how the data structure is built, and how it implements row swaps and row additions in Section B.

4.2 Cost

Recall that the BSP cost of algorithm one using dense data structure is given by

$$n \cdot ((2Mm + (N + M)m' + N + P)g + m(m' + 2) + N + 6l).$$

Locally, each processor has r_u many row nodes and c_v many column nodes. Since we are not changing the algorithm but merely the way our matrix is stored, our cost analysis remains the same and hence our new BSP cost is simply

$$n \cdot ((2Mr_u + (N + M)c_v + N + P)g + r_u(c_v + 2) + N + 6l).$$

But now $r_u, c_v \ll m, m'$. Thus we can expect to see a significant decrease in runtime.

4.3 Results & Discussion

From the graphs of Fig. 4 we can observe that when the tableau considered is generated by a few number of gates, and the system consists of many qubits, the sparse implementation reduces the computation time by a third. It is not evident from these graphs, but unlike the dense implementation, increasing the number of processors provides a significant computational advantage even for small systems. This follows directly from the structure of sparse systems, as more processors imply smaller linked lists and less time wasted on searching and updating lists. The same cannot be said about dense implementations. Finally, we also observe that the sparse implementation falters when the number of ones in the 400 qubit system reaches 300 (or 1/1000 the size of the tableau). To us, it seems that the system is still extremely sparse, and so we think it is our implementation of the data structure that fails. We thus conclude this section by elaborating on the aspects that could be improved.

Like we mentioned initially, we use a one-dimensional unordered doubly-linked list, but the performance of the algorithm could be greatly improved by using a two-dimensional ordered quadruple-linked list. This particular type of data structure resembles a matrix where

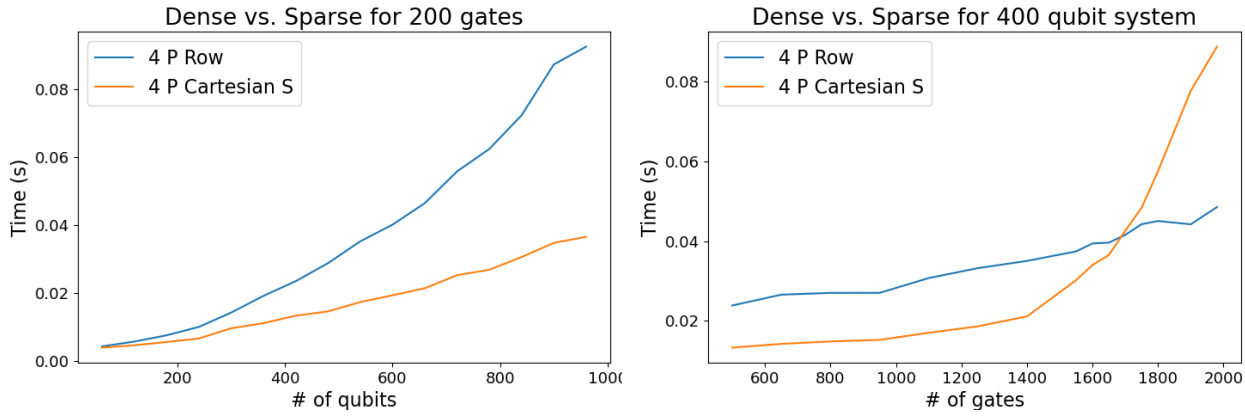

Figure 4: Performance comparison between fastest dense ($P = 4$ with $N = 4$ and $M = 1$) and fastest sparse ($N = 2$ and $M = 2$) implementation of Algorithm 1. Left: The input tableau is generated by applying 200 gates, and we vary the size of the system. Right: We set the size of the system and vary the number of gates in the circuit that creates the input tableau. The lines cross when the tableau is built using 1700 gates, resulting in around 300 non-zero elements.

the zero elements have been removed. Hence, we could search through columns and rows much faster. Another possibility for improvement, is to try other ways of distributing the data to the processors. Our conjecture is that the 2d-cyclic distribution might be causing load imbalances, and perhaps permuting the rows and columns at random or implementing a Mondriaan distribution will yield better results.

5 Conclusion

Transforming the algorithms described by [1] into parallel algorithms, we were able to achieve a dramatic reduction in computation cost. However, as we have shown, there is much more to explore for this particular computation. A better simulator like that of [6] would provide us the ability to simulate bigger systems, and as a result make better use of the higher core supercomputers. Also, a better understanding of the distribution of the non-zero elements of the stabilizer tableau would help in determining the best distribution for sparse implementations.

Combining the fields of parallel computation and quantum simulation is unorthodox, and quite challenging, but we have shown promising results in terms of lower runtime, especially when it comes to simulating large number of qubits. Hence, we are quite optimistic about the future prospect of combining these fields, and improving the already emerging field of hybrid (quantum-classical) algorithms.

References

[1] H. J. Garcia, I. L. Markov, and A. W. Cross, "Efficient inner-product algorithm for stabilizer states," *arXiv preprint arXiv:1210.6646*, 2012.

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[3] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, 1996.

[4] K. M. Obenland and A. M. Despain, "A parallel quantum computer simulator," *arXiv preprint quant-ph/9804039*, 1998.

[5] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Improving gate-level simulation of quantum circuits," *Quantum Information Processing*, vol. 2, no. 5, pp. 347–380, 2003.

[6] S. Bravyi, D. Browne, P. Calpin, E. Campbell, D. Gosset, and M. Howard, "Simulation of quantum circuits by low-rank stabilizer decompositions," *Quantum*, vol. 3, p. 181, 2019.

[7] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, no. 5, p. 052328, 2004.

[8] D. Gottesman, "The heisenberg representation of quantum computers," *arXiv preprint quant-ph/9807006*, 1998.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

Appendices

A Alg 2 Detailed

Algorithm 2: Parallelized basis state reduction

Input : Canonical reduced form of $\mathcal{A}$
Output : Basis state and Quantum Circuit $\mathcal{C}$
$\mathcal{H}_1 \leftarrow \emptyset$
$\mathcal{I} \leftarrow \emptyset$
Data: $m = n \; div \; N, \; m' = 2n \; div \; M$
$s = \text{p.i.d} \mod N, \; t = \text{p.i.d} \; div \; N$
$i \leftarrow 0 < n, \; j \leftarrow n < 2n$

Superstep 1: Apply the first block of H-gates

$k \leftarrow k \in [n, \ldots, 2n-1]$ such that $\mathcal{A}_{kj} = 1$
if k *exist* **then**
  ROWSWAP$(\mathcal{A}_i, \mathcal{A}_k)$; ▷ `The search and swap is parallelized`
  $i \leftarrow i+1, \; j \leftarrow j+1$
else
  $k_2 \leftarrow \text{argmax}[0, \ldots, n-1]$ such that $\mathcal{A}_{kj} = 1$
  if k_2 *exist* **then**
    ROWSWAP$(\mathcal{A}_i, \mathcal{A}_{k_2})$; ▷ `The search and swap is parallelized`

    if $\phi_0(i) = s$ **then**
      for $j' \in [j+1, \ldots, 2n-1]$ **do**
        if $\mathcal{A}'_{(i,j')} = 1$ *and* $i \cdot N + s \geq j'$ **then**
          $\mathcal{I} \leftarrow \mathcal{I} \cup \{j'\}$
      end
    for $j' \in \mathcal{I}$ **do**
      $\mathcal{H}_1 \leftarrow \mathcal{H}_1 \cup \{H_{j'-n}\}$
      COLUMNSWAP$(\mathcal{A}'^{\top}_{(j')}, \; \mathcal{A}'^{\top}_{(j'-n)})$
    end
  $i \leftarrow i+1, \; j \leftarrow j+1$
end

Superstep 2: Apply CNOT-gates

$\mathcal{CX} \leftarrow \emptyset$
$\mathcal{I} \leftarrow \emptyset$
$j \leftarrow 0 < n$
for $k \in [j+n+1, \ldots, 2n-1]$ **do**
  $k' \leftarrow \mu_0(k)$
  if $\phi_0(j) = s$ *and* $\mathcal{A}'_{[j,k]} = 1$ **then**
    if $k' \cdot M + t \geq k$ **then**
      $j' \leftarrow k' \cdot M + t$
      $\mathcal{I} \leftarrow \mathcal{I} \cup \{j'\}$
end
if $\mathcal{I} \neq \emptyset$ **then**
  for $j' \in \mathcal{I}$ **do**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}_{i,\ j'} \leftarrow \mathcal{A}'_{[i,\ j']}$
      $\mathcal{A}_{i,\ j'-n} \leftarrow \mathcal{A}'_{[i,\ j'-n]}$
    end
  end
  if $\phi_1(j) = t$ **then**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}_{i,\ j} \leftarrow \mathcal{A}'_{[i,\ j]}$
      $\mathcal{A}_{i,\ j-n} \leftarrow \mathcal{A}'_{[i,\ j-n]}$
    end
  for $j' \in \mathcal{I}$ **do**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}'_{[i,j']} \leftarrow \mathcal{A}'_{[i,j']} \oplus \mathcal{A}_{ij'}$
    end
    $\mathcal{CX} \leftarrow \mathcal{CX} \cup \{(CX, i, j'-n)\}$
  end
  $j \leftarrow j+1$

Superstep 3: Apply CP-gates

$\mathcal{CP} \leftarrow \emptyset$
$\mathcal{I} \leftarrow \emptyset$
$j \leftarrow 0 < n$
for $k \in [0, \ldots, n-1]$ **do**
  $k' \leftarrow \mu_0(k)$
  if $\phi_0(j) = s$ *and* $\mathcal{A}'_{[j,k]} = 1$ *and* $\mathcal{A}'_{[j,k+n]} = 0$ **then**
    if $k' \cdot M + t \geq k$ **then**
      $j' \leftarrow k' \cdot M + t$
      $\mathcal{I} \leftarrow \mathcal{I} \cup \{j'\}$
end

if $\mathcal{I} \neq \emptyset$ **then**
  for $j' \in \mathcal{I}$ **do**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}_{i,\ j'} \leftarrow \mathcal{A}'_{(i,\ j')}$
      $\mathcal{A}_{i,\ j'+n} \leftarrow \mathcal{A}'_{(i,\ j'+n)}$
    end
  end
  if $\phi_1(j) = t$ **then**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}_{i,\ j} \leftarrow \mathcal{A}'_{(i,\ j)}$
      $\mathcal{A}_{i,\ j+n} \leftarrow \mathcal{A}'_{(i,\ j+n)}$
    end
  for $j' \in \mathcal{I}$ **do**
    for $i' \in [0, \ldots, m]$ **do**
      $i \leftarrow i' \cdot N + s$
      $\mathcal{A}'_{[i,j]} \leftarrow \mathcal{A}'_{[i,j]} \oplus \mathcal{A}_{ij'}$
      $\mathcal{A}'_{[i,j']} \leftarrow \mathcal{A}'_{[i,j']} \oplus \mathcal{A}_{ij+n}$
    end
    $\mathcal{CP} \leftarrow \mathcal{CP} \cup \{(CP, i, j' - n)\}$
  end
  $j \leftarrow j + 1$

Superstep 4: Apply phase-gates

$\mathcal{S} \leftarrow \emptyset$
$j \leftarrow n < 2n$
if $\phi_0(j) = s$ *and* $\phi_1(j) = s$ **then**
  if $\mathcal{A}'_{[i,j]} = 1$ *and* $\mathcal{A}'_{[i,j-n]} = 1$ **then**
    ${\mathcal{A}'_{[j-n]}}^{\top} = {\mathcal{A}'_{[j-n]}}^{\top} \oplus {\mathcal{A}'_{[j]}}^{\top}$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{S_{j-n}\}$

Superstep 5: Apply H-gates

$\mathcal{H}_2 \leftarrow \emptyset$
$j \leftarrow n < 2n$
if $\phi_0(j) = s$ *and* $\phi_1(j) = s$ **then**
  if $\mathcal{A}'_{[i,j]} = 1$ *and* $\mathcal{A}'_{[i,j-n]} = 0$ **then**
    COLUMNSWAP$({\mathcal{A}'_{[j-n]}}^{\top},\ {\mathcal{A}'_{[j]}}^{\top})$
    $\mathcal{H}_2 \leftarrow \mathcal{H}_2 \cup \{H_{j-n}\}$

B Code

```
void parallelalg1_test() // Parallelized version of Algorithm 1
{
  for (int col = num_qubits; col < 2 * num_qubits; col++){
      //Row addition begins here

      //Sending row k to other row processors to facilitate removing of trailing ones later
      if (row_name == k % N){
        for (int l = 0; l < N; l++){
          bsp_put(l + (col_name % M) * N, my_matrix[(int)floor(k / N)], num_to_add, 0, column_dim * sizeof(int));
          if (col_name == 0){
            // Processor in row k with column name 0 send their phases to all other procs with col_name 0.
            bsp_put(l, &my_vec[(int)floor(k / N)], &phase_received, 0, sizeof(int));
          }
        }
      }
      bsp_sync();

      // Right side of matrix and sending the flags
      if (col_name == col % M && row_name == k % N){
        my_matrix[(int)floor(k / N)][(int)floor(col / M)] = -1;
        for (int i = 0; i < row_dim; i++){
          for (int l = 0; l < M; l++){
            bsp_put(row_name + (l % M) * N, &my_matrix[i][(int)floor(col / M)], flag_array, i * sizeof(int), sizeof(int));
          }
        }
        my_matrix[(int)floor(k / N)][(int)floor(col / M)] = 1;
      }
      else if (col_name == col % M){
        for (int i = 0; i < row_dim; i++){
          for (int l = 0; l < M; l++){
            bsp_put(row_name + (l % M) * N, &my_matrix[i][(int)floor(col / M)], flag_array, i * sizeof(int), sizeof(int));
          }
        }
      }
      bsp_sync();

      // Compute the first symplectic inner product in parallel. We call this variable "a".
      // We put the local a, on the processor with column_name 0.
      for (int l = 0; l < row_dim; l++){
        a_loc[l] = our_nan;
        b_loc[l] = our_nan;
      }
      for (int i = 0; i < row_dim; i++){
        if (flag_array[i] == 1){
```

```
45          // z, x = my_matrix[:column_dim/2], my_matrix[coulmn_dim/2:]
46          //z', x' = nums_to_add[:column_dim/2], nums_to_add[column_dim/2:]
47          // Compute a_loc = <z,x'> - <x,z'>
48          a_loc[i] = 0;
49          b_loc[i] = 0;
50          for (int j = 0; j < (int)(column_dim / 2); j++){
51            a_loc[i] += (my_matrix[i][j] * num_to_add[j + (column_dim / 2)]) -
     (my_matrix[i][j + (column_dim / 2)] * num_to_add[j]);
52
53            int v = MOD((my_matrix[i][j] + num_to_add[j]) * (my_matrix[i][j +
    column_dim / 2] + num_to_add[j + column_dim / 2]), 2);
54
55            b_loc[i] += v - ((my_matrix[i][j] + num_to_add[j]) * (my_matrix[i
    ][j + column_dim / 2] + num_to_add[j + column_dim / 2]));
56          }
57        }
58      }
59      bsp_put(row_name, a_loc, partial_as, col_name * row_dim * sizeof(int),
    row_dim * sizeof(int));
60      bsp_put(row_name, b_loc, partial_bs, col_name * row_dim * sizeof(int),
    row_dim * sizeof(int));
61      bsp_sync();
62
63      // The processor with column name 0, will use the partial a's to compute
     a for every row he particates in.
64      if (col_name == 0){
65        for (int i = 0; i < row_dim; i++){
66          if (partial_as[i] != our_nan){
67            for (int ii = i; ii < row_dim * M; ii += row_dim){
68              total_as[i] += partial_as[ii] + partial_bs[ii];
69            }
70            total_as[i] = MOD(total_as[i], 4);
71            my_vec[i] = MOD(total_as[i] / 2 + my_vec[i] + phase_received, 2);
72          }
73        }
74      }
75
76      //Replace our stuff with the things just received depending on the flag
77      for (int i = 0; i < num_qubits; i++){
78        if (row_name == i % N){
79          if (flag_array[(int)floor(i / N)] == 1){
80            //Start replacing horizontally
81            for (int l = 0; l < column_dim; l++){
82              my_matrix[(int)floor(i / N)][l] = (my_matrix[(int)floor(i / N)][
    l] + num_to_add[l]) % 2;
83            }
84          }
85        }
86      }
87      bsp_sync();
88      /*Right side algorithm ends here*/
```

Listing 1: Parallel Algorithm 1

```
// HEAD is the most important node of the list, as it is through this node that we can access the child nodes.
// Therefore, we must never delete it or change its address. If we loose it, then we cannot access the child
// nodes located in heap memory. HEAD is the link that connects different memory locations  and keeps the list
// together.

// We define the doubly-linked list. It has two integer fields "row" and "column".
// "prev" and "next" point to the previous and next nodes.
struct node
{
  int row;
  int column;
  struct node *prev;
  struct node *next;
};

// We create a new node in the following way.
// If there is no relevant node in the list, HEAD is a fake node with entries "-1".
void append_to_list(struct node *head, int i, int j)
{
  // If HEAD is fake (list is empty)
  if (head -> row == -1){
    head -> row = i;
    head -> column = j;
    head -> next = NULL;
    head -> prev = NULL;
  }
  //If there is at least one real node in list.
  else{
    while (head->next != NULL)
    {
      head = head->next;
    }
    // Create child node
    struct node *new = malloc(sizeof(struct node));
    new -> row = i;
    new -> column = j;
    head->next = new;
    new->prev = head;
    new->next = NULL;
  }
}

//Function that deletes any node.
void delete_single(struct node *head, int i, int j){
  struct node *temp = head;
  while(temp != NULL){
    if(temp -> row == i && temp -> column == j){
      if(temp == head){
        delete_node_head(temp); //Special function that "deletes" HEAD.
```

```
50          break;
51        }
52        else{
53          delete_node(temp); //Delete a normal node
54          break;
55        }
56      }
57      temp = temp -> next;
58    }
59  }
60
61  // Function that swaps two rows
62  void swap_two_rows(struct node *head, int row1, int row2){
63    while(head != NULL){
64      if(head -> row == row1){
65        head -> row = row2;
66      }
67      else if(head -> row == row2){
68        head -> row = row1;
69      }
70      head = head -> next;
71    }
72  }
73
74  //Function that adds two rows together. Follows the same logic as superstep 5
      and 6.
75  // "flag_array" contains the rows that participate in the addition.
76  // ""num_to_add  contains the column values we should add to the list.
77  // Both "flag_array" and "num_to_add" may contain elements "-1" that tell the
      processor when to stop.
78  void add_rows(struct node *head, int *flag_array, int *num_to_add, int
      sizeflag, int sizenum){
79    bool flag = false;
80    struct node *temp = head; // Set a temporary addres.
81    for(int i = 0; i < sizeflag; i++){
82      if(flag_array[i] != -1){  //Check if we have to stop.
83        for(int j =0; j < sizenum; j++){
84          if(num_to_add[j] != -1){ // Check if we have to stop
85            flag = false;
86            while(temp != NULL){
87              if(temp -> row == flag_array[i] && temp -> column == num_to_add[j
      ]){ //If element already exists, we delete it.
88                delete_single(head, flag_array[i], num_to_add[j]);
89                flag = true;
90                break;
91              }
92              temp = temp -> next;
93            }
94            //If element does not exist in the list we add it.
95            if(flag == false){
96              append_to_list(head, flag_array[i], num_to_add[j]);
97            }
98            temp = head;
99          }
```

```
100           else{
101             break;
102           }
103         }
104         temp = head;
105       }
106       else{
107         break;
108       }
109     }
110 }
```

Listing 2: Data structure and helper functions of sparse algorithm

```
1  // ------------------ Superstep 5 --------------------------
2
3  // Processors handling row k send to others in their processor column
4  if(row_name == k % N){
5    tot_ones_row = find_ones_row(&head, all_ones_row, k);
6    for (int l = 0; l < N; l++){
7      bsp_put(l + (col_name % M) * N, all_ones_row, num_to_add, 0, tot_ones_row
       * sizeof(int));
8    }
9  }
10
11 if(col_name == col%M){
12   tot_ones_col = find_ones_col_except(&head, all_ones_col, col, k);
13   //Send to the processor row
14   for (int l = 0; l < M; l++){
15     bsp_put(row_name + (l % M) * N, all_ones_col, flag_array, 0, tot_ones_col
       * sizeof(int));
16   }
17 }
18 bsp_sync();
19
20 // ----------------Superstep 6 --------------------------
21
22 // Now we do the updates
23 add_rows(&head, flag_array, num_to_add, row_dim, column_dim);
24
25 k++;
26
27 bsp_sync();
```

Listing 3: Superstep 5 & 6 sparse algorithm

C Benchmark

p	r (Mflop/s)	g (flops)	l (flops)
1	6481.42	172.2	141.7
2	6480.84	192.4	-1040.7
3	6458.9	224.6	304.9
4	6337.4	237.8	-875.1

Table 1: Benchmark for the 4-core 2.5GHz Intel-Core i7 MacBook Pro where all data was gathered. The computer has 8 logical cores, but there was no visible improvement in using more than the 4 physical cores.