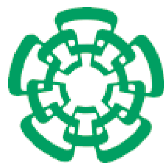

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Computación

Intérprete serial y paralelo para algoritmos de Cómputo Cuántico

Tesis que presenta

Carlos Alberto Valle Garrido

para obtener el grado de

Maestro en Ciencias en

la Especialidad de Ingeniería Eléctrica

Director de Tesis

Dr. Guillermo Morales Luna

México, D.F. Marzo 2008

Resumen

El computo cuántico es un importante paradigma, donde las implementaciones prácticas son un problema complicado y hoy día tienen muchas limitantes en la investigación. Sin embargo, sus algoritmos formales han mostrado avances en eficiencia respecto a su complejidad, y muchos problemas difíciles para el Computo Clásico han sido formalmente resueltos de manera eficiente mediante el uso de algoritmos cuánticos. Se requiere de plataformas computacionales para probar algoritmos cuánticos. En esta tesis se desarrollo un intérprete capaz de transformar pseudocódigos de alto nivel de algoritmos cuánticos mediante el uso de un lenguaje formal el cual está constituido de una serie de composiciones de terminos básicos, i.e. operadores y listas de operandos. Los programas transformados pueden ser ejecutados en una simple computadora de escritorio o en plataformas paralelas las cuales pueden ser un cluster con lenguaje de programación MPI o un rack de multiprocesadores.

Abstract

Quantum Computing is an important paradigm, whose practical implementation is a matter of intense research and up-today is quite limitative. However, its formal algorithms have shown impressive gains in complexity efficiency, and several hard problems for Classical Computing have been formally solved very efficiently using quantum algorithms. Hence, computing platforms to test quantum algorithms are required. Here we develop an interpreter able to transform high level pseudocodes for quamtum algorithms into a formal language consisting of compositions of basic terms, i.e. operators and operands lists. The transformed programs may be run either in a single computer or in a parallel platform which in turn may be a MPI cluster or a multiprocessor rack.

Agradecimientos

Este escrito es el logro de muchas personas e instituciones que han estado a mi lado apoyándome en cada una de las etapas de mi carrera profesional a quienes les dedico estos agradecimientos.

A mis padres les agradezco todo el apoyo que me han brindado durante toda mi vida, dándome consejos y mostrándome el camino correcto a seguir...

A mis hermanos que siempre me han motivado a seguir con las metas propuestas hasta cumplirlas...

A mis profesores en especial a los mas *estrictos* ya que sin su dedicación no tendría los conocimientos necesarios para la realización de está tesis...

Al Dr. Guillermo Morales... sin duda el profesor al que más admiro... por sus conocimientos en las matemáticas y en la computación. Gracias por ser mi asesor...

A mis amigos, todos aquellos con los que compartí desveladas en estos dos años... en especial al Willi, mi compañero de cuarto, que nunca me negó su ayuda en el momento en que la necesite...

A Sofi por ser tan amable y siempre tener tiempo para resolver mis problemas académicos...

Al CINVESTAV por darme la oportunidad de ingresar a sus instalaciones...

Al CONACYT y a la UJAT que sin su apoyo económico no hubiera podido dedicar todo mi tiempo a mis estudios...

A las personas más importantes de mi vida... Edna y mis bebes... gracias por estar a mi lado en todo momento..

Índice general

Índice de tablas **x**

1. Introducción **1**
1.1. Planteamiento del problema 3
1.2. Marco teórico 4
1.2.1. Notación clásica en computación cuántica 4
1.2.2. Multiplicación de matrices 4
1.2.3. Producto tensorial 5
1.2.4. Qubits y quvectores 6
1.2.5. Qu-compuertas 7
1.2.6. Mediciones 10
1.2.7. Gramáticas 11
1.3. Organización de la tesis 12

2. Procesamiento de las compuertas cuánticas **13**
2.1. Representación de las matrices 13
2.2. Hermitiana de una matriz 16
2.2.1. Transpuesta de una matriz 16
2.2.2. Conjugado complejo de una matriz 17
2.3. Suma de matrices 19
2.4. Multiplicación de matrices 19
2.4.1. Multiplicación de matrices utilizando su transpuesta 20
2.4.2. Multiplicación de matrices obteniendo por partes las entradas de la matriz resultante 21
2.4.3. El algoritmo de Winograd para la multiplicación de matrices. 24
2.5. Producto tensorial 25

3. Sintaxis del interprete **27**
3.1. Gramáticas y Lenguajes 27
3.2. Descripción de la solución 29
3.2.1. Diseño de la gramática 30
3.2.2. Fases de la solución 31

4. Semántica procedimental **35**
4.1. Implementación de la solución . . . 35
4.2. Compilación . . . 36
4.2.1. Fase de definición de símbolos . . . 36
4.2.2. Fase de expansión . . . 38
4.2.3. Fase de sustitución . . . 39
4.2.4. Fase de identación . . . 40
4.2.5. Pequeño ejemplo de la compilación . . . 41
4.3. Ejecutor . . . 41
4.3.1. Ejecutor en Fedora en el lenguaje C sobre una PC . . . 42
4.3.2. Ejemplo del ejecutor en Fedora en el lenguaje C sobre una PC 43
4.3.3. Ejecutor en Red Hat en el lenguaje MPI sobre un cluster . . . 46
4.3.4. Ejemplo del ejecutor en Red Hat en el lenguaje MPI sobre un cluster . . . 47

5. Casos de prueba **49**
5.1. Reverso de qubits . . . 50
5.1.1. Desarrollo del algoritmo *reverso* . . . 50
5.1.2. Ejemplo de objeto resultante para el algoritmo reverso . . . 53
5.2. Transformada Discreta de Fourier Cuántica . . . 56
5.2.1. Ejemplo de objeto resultante para el algoritmo TDF cuántica 60
5.2.2. Tiempos de ejecución de la TDF cuántica . . . 64
5.3. Algoritmo de Peter Shor . . . 65
5.3.1. Pequeño recordatorio de teoría de números . . . 67
5.3.2. Algoritmo cuántico para el cálculo de órdenes . . . 68
5.3.3. Elementos con orden potencia de 2 . . . 68
5.3.4. Elementos con orden arbitrario . . . 70

6. Conclusiones y trabajo futuro **75**
6.1. Conclusiones . . . 75
6.2. Trabajo futuro . . . 77

A. Disco de programas, pruebas y desarrollos **79**

Referencias **81**

Índice de figuras

1.1. Compuerta CNOT, prototipo de las compuertas para multiplies qubits. 8

1.2. Compuerta CNOTI, es la compuerta CNOT con los roles cambiados. 9

1.3. Generación de la compuerta CNOTI. 9

2.1. Representación de una matriz en archivos. Esta representación sirve de igual manera para representar a estados cuánticos, ya que estos son matrices de n renglones y una columna. 16

2.2. Diagrama de flujo del algoritmo de Winograd usando Archivos. El modulo *Winograd_Memoria_Dinámica* es el mismo algoritmo pero haciendo las operaciones con memoria dinámica. 25

3.1. Esquema general de la solución. Se divide al problema en dos bloques, la compilación donde se obtiene el código objeto que es libre de plataforma, sistema operativo, lenguaje y la ejecución donde se obtiene el resultado númerico deseado. 34

4.1. Esquema a seguir para distribuir de manera equitativa las instrucciones escritas en el código objeto a ejecutar. En la imagen $i \in \mathbb{N}$, t es el número total de `Adc's`, n es el número de procesadores y $dist = \frac{t}{n}$. . 47

5.1. Circuito que intercambia dos qubits, y su equivalente símbolo esquemático más comunmente usado. 51

5.2. Circuito que intercambia tres qubits. 51

5.3. Esta gŕafica muestra que el tiempo tomado para ejecutar el archivo que describe la TDF con los símbolos definidos por la gramática es superior al tiempo que tarda el programa en ejecutar de manera inmediata la TDF. 64

5.4. Se paralelizo él algoritmo de la TDF cuántica y se ejecutó usando 1 y 3 procesadores. En la gráfica se puede ver que apartir de $k = 3$, es mejor la ejecución del algoritmo usando 3 procesadores. 65

5.5. Se paralelizó el ejecutor de símbolos de la TDF cuántica. Se ejecuto usando 1 y 3 procesadores. En la gráfica se puede ver que apartir de $k = 4$, es mejor la ejecución del algoritmo usando 3 procesadores. . . 66

5.6. La gráfica muestra los tiempos tomados por las implementaciones en paralelo del algoritmo *Ejecutor* así como de la ejecución inmediata de la TDF, en ambos casos se utilizo 3 procesadores. 66

Índice de tablas

2.1. Se muestra el tamaño y la cantidad de memoria necesarios para las qucompuertas, que actúan en el espacio $\mathbb{H}^n$. Las entradas de las matrices son números complejos, cada uno de ellos está representado por dos `doubles` que ocupan 8 bytes cada uno, por lo que la cantidad de memoria necesaria para representar al operador crece de forma exponencial con respecto a n. 14

2.2. Algoritmo para leer un vector de datos en M_i. La variable x representa la cantidad de datos que el algoritmo puede leer sin afectar la ejecución del programa, la variable y es la cantidad de complejos por leer. Si $y < x$ se leen y datos si no x datos de la matriz M_i. 17

2.3. Algoritmo para obtener la transpuesta de una matriz. Hay que tener en cuenta que las lecturas evitan tomar los dos primero enteros que indican la dimensión de la matriz. 18

2.4. Algoritmo para obtener el conjugado de las entradas de una matriz. La palabra *conj* representa a la función conjugado de un número complejo. 18

2.5. Algoritmo para la suma de matrices. La entrada *oper* indica si la operación a realizar es suma o resta. 20

2.6. Algoritmo para la multiplicación de matrices usando la transpuesta de la segunda matriz para el acceso rápido de los datos. 22

2.7. Algoritmo para la multiplicación de matrices. Se suman resultados parciales hasta obtener cada uno de los renglones de la matriz deseada. . . 23

2.8. Algoritmo para realizar el producto tensorial de matrices. Se multiplica cada elemento de $M_{i,\alpha}$ por $M_{j,\beta}$ y se escribe en M_k, esto para todos los renglones de las matrices. 26

3.1. Reglas sintácticas. 32

4.1. La primera columna representa la producción que genera símbolos terminales, la segunda el símbolo terminal utilizado, la tercera nos indica qué parámetros son necesarios para que la producción puede ser ejecutada. 37

4.2. Algoritmo que ejecuta el código objeto resultante de la compilación. `P1` y `P2` representan la primera y segunda pila respectivamente. La función `Push` inserta un elemento en la pila `P1` o `P2` según el caso. Si la pila a insertar es `P2` entonces también actualiza el campo `NumParLeidos` del tope de `P1`. La función `Ejec` ejecuta la operación guardada en el tope de P1 usando los primeros `P1.NumPar` elementos de la pila `P2`. 44

5.1. Subgrupo formado por las compuertas C, D y la operación de composición. 50
5.2. Algoritmo para obtener la matriz que aplicada a n qubits, obtiene su reverso 53
5.3. Algoritmo para calcular la transformada rápida de Fourier cuántica . 60
5.4. Algoritmo para localizar divisores de ordenes de elementos. 70
5.5. Algoritmo para calcular órdenes de elementos 71
5.6. Algoritmo para calcular fracciones continuadas. 72
5.7. Algoritmo para localizar divisores de órdenes de elementos 73

Capítulo 1

Introducción

En Ciencias de la Computación existe un análogo al *Premio Nobel*, el cual lleva por nombre *el Premio Turing* y se otorga cada año por la *Association for Computing Machinery* (ACM). Lleva el nombre de *Turing* en honor al matemático Alan Mathison Turing (1912 - 1954), al cual se le considera uno de los padres de la Computación. Una de las aportaciones más importantes que hizo fue la formalización del concepto de *algoritmo* y con ello la formalización misma de la computación. Esto lo logró gracias a su trabajo "*Los números computables, con una aplicación al problema Entscheidungs*"publicado en 1936 [1, 2], en este documento Turing diseñó unas máquinas llamadas de Turing (MT), las cuales él mismo demostró que eran capaces de resolver cualquier problema matemático que pudiera ser representado mediante un algoritmo. Una característica que tienen estas máquinas es que los lenguajes que reconocen son los generados por las gramáticas formales.

Alan Turing y Alonzo Church (1903 - 1995), propusieron lo que hoy se conoce como la *Tesis Church-Turing*, en la cual se demostró que para cualquier programa de computadora es posible crear una máquina de Turing equivalente.

Las máquinas de Turing y la Tesis Church-Turing, ayudaron a dividir a los problemas en clases. Estas clases están formadas por problemas con la misma complejidad, donde la complejidad es la cantidad de recursos que necesitan los problemas para ser resueltos. De entre las clases, unas de las más importantes son las clases de problemas P y NP. Los problemas de la clase P, tienen la característica de que tienen una complejidad polinomial, es decir, existe al menos un algoritmo capaz de darle solución en tiempo polinomial, mientras que en los problemas de la clase NP no existe dicho algoritmo, sin embargo, existe al menos un algoritmo capaz de verificar si una solución propuesta para este tipo de problema es correcta en tiempo polinomial.

La tesis Church-Turing asume de que el dispositivo que la implemente se base en una máquina de Turing, pero cabe la posibilidad de que exista o se diseñe una máquina que haga computación empleando otros principios y que pueda contradecir la tesis Church-Turing. Lo que significa que quizá para algún problema en la clase NP, existe un algoritmo que puede solucionar al problema en tiempo polinomial utilizando un dispositivo de cómputo diferente a las máquinas de Turing clásicas.

Una alternativa para hacer computación es lo que actualmente se conoce como

computación cuántica que inicialmente fue propuesta por Richard Feynman [3]. Él quería resolver sistemas cuánticos llevando a cabo simulaciones en una computadora clásica, pero resultaba muy costosa su ejecución (de manera exponencial en tiempo), por lo que propuso la creación de una computadora que se rigiera por las leyes de la mecánica cuántica.

La idea de Feynman, aun cuando era muy novedosa no fue considerada hasta que David Deutsch estudio algunos aspectos que debería tener dicha computadora y tiempo después, trabajando en conjunto con David Deutsch y Paul Benioff, propusieron las bases de la computación cuántica y con ello el concepto de *computadoras cuánticas.*

Aun cuando ya se habían sentado las bases de la computación cuántica, no se creía que algún día llegara a ser de utilidad, ya que no existía ningún procedimiento importante para las computadoras cuánticas, hasta que en 1994 Peter Shor presenta un algoritmo capaz de resolver un problema en la clase NP de manera eficiente [4, 5, 6]. El problema que resolvío fue el de factorizar un número grande en tiempo polinomial. A este algoritmo se le considera como el más importante en el estado del arte, debido a las consecuencias que tendría si se le llegara a implementar en una verdadera computadora cuántica, principalmente a los sistemas criptográficos como el RSA que basan su seguridad en la dificultad de factorizar un número grande.

A partir del algoritmo de Shor la comunidad científica enfocó más recursos al área de la computación cuántica, teniendo como resultado algunos otros algoritmos que son capaces de resolver ciertos problemas en la clase NP en tiempo polinomial, utilizando las operaciones que presuntamente debería tener una computadora cuántica. Todos los resultados que actualmente existen son puramente teóricos pues asumen la existencia de una computadora cuántica ideal; debido a esto y al hecho de que no existe tal implementación se ha recurrido a la simulación de las operaciones que debería de realizar tal computadora.

De las pruebas que se han llevado a cabo en algunos algoritmos cuánticos, se ha estimado que simular un algoritmo con una entrada de 20 bits cuánticos requiere alrededor de un día de cómputo en las computadoras modernas. Como los problemas que normalmente atacan estos algoritmos pertenecen a la clase NP (crecen en complejidad exponencial), se puede ver que cuando la cantidad de bits cuánticos crece, el tiempo y los recursos computacionales para llevar a cabo la simulación crecen de forma exponencial.

Al problema de la simulación cuántica se le puede pseudo-resolver de maneras muy distintas y variadas. Una solución la propuso [7] donde se implementan y empaquetan las primitivas cuánticas en una biblioteca bajo una plataforma de hardware. Un enfoque diferente se propone en [8] donde se plantea crear un *Procesador Cuántico* utilizando también dispositivos físicos como los FPGA. Pero no todas las simulaciones han sido en el área del hardware, en software se cuenta con el trabajo de [9], el cual es una demostración de que es posible implementar de manera eficiente el algoritmo de Grover [10] usando una representación gráfica del producto tensorial de matrices. Una aportación más del lado del software es la biblioteca de primitivas cuánticas Qulib

[11], la cual tiene las siguientes características:

- Los vectores base pueden ser de tamaño arbitrario.
- Cuenta con una representación eficiente de los estados cuánticos.
- Se puede combinar sub-estados de qubits.
- Se puede hacer composiciones de operadores cuánticos.
- Es fácil de extender debido a la herencia de las clases.

con la desventaja de que es limitada por el sistema operativo, sólo se encuentra para Linux. Como se puede ver, tratar de visualizar el comportamiento de un algoritmo cuántico resulta una tarea complicada tanto en tiempo de creación como en tiempo de ejecución.

Se han planteado traductores de lenguajes como en [12], donde se propone un procedimiento para diseñar un traductor de C a VHDL. Se genera un *parser* y un analizador léxico. Plantean una traducción de dos pasadas, en la primera pasada se crean las variables y el flujo de datos, todo esto almacenado en una tabla, mientras que en al segunda pasada, se produce el código en VHDL que es la salida final.

También se han propuesto metodologías como en [**?**] para traducir del código C al VHDL, en el cual se dan consejos a seguir para una buena elección de qué parte del código debe ser traducido y qué parte es conveniente mantener en C. Toma la idea de [12] al dividir en 2 partes la traducción, pero sin ser posible la optimización a código paralelo, cuyo problema lo resuelven en [13].

Existen programas comerciales que ya hacen esta traducción y optimización, es el caso del Handle C. En [14] se crea un una variante del lenguaje C, cuya ventaja es que ya traduce automáticamente de este lenguaje a código que es posible grabar en un FPGA.

1.1. Planteamiento del problema

Existen problemas para los cuales el mejor algoritmo clásico es de orden exponencial, estos problemas se encuentran en la clase *NP*; es decir, no existe un procedimiento que resuelva el problema en tiempo polinomial.

Sin embargo, la teoría de la computación cuántica nos ofrece algoritmos que sean capaces de reducir los tiempos de ejecución de algunos problemas de la clase P e incluso de la clase *NP*. Estos algoritmos se basan en la suposición de que se cuenta con una computadora cuántica, la cual necesita tecnología que no está disponible hoy en día. Estas computadoras deben de cumplir con ciertas características como: las operaciones que pueden realizar, la representación de la información y el escalamiento de la misma, entre otras.

Para observar el comportamiento y obtener pruebas estadísticas de estos algoritmos se recurre a la simulación con la tecnología disponible de las operaciones de una

computadora cuántica. Esto requiere escoger la plataforma, el sistema operativo y/o el lenguaje correcto pero esta simulación necesita de cierto análisis, muchas pruebas e incluso, en algunas ocasiones, la única manera de saber cuál de las opciones es mejor para cierto problema, es implementar la solución en todas las opciones que se tengan, lo que al final se traduce a un costo en el tiempo de implementación.

Este es precisamente el problema que se quiere abordar en esta tesis, diseñar un procedimiento que sea capaz de simular de manera eficiente lo que una computadora cuántica haría, así como poder cambiar de plataforma, sistema operativo y/o lenguaje, sin afectar de manera significativa el tiempo de implementación.

1.2. Marco teórico

Las computadoras actuales basan su funcionamiento en una serie de operaciones, como la aplicación de compuertas, el almacenamiento de datos, la inicialización de registros, entre otros; sin las cuales no se podría realizar cómputo alguno. En el caso de la computación cuántica y su computadora, existe un análogo a estos términos. Es por tal motivo que en las siguientes sub-secciones se definirán las operaciones básicas de las que debe constar una computadora cuántica, así como los términos y definiciones que nos servirán para entender correctamente el contenido de la tesis.

1.2.1. Notación clásica en computación cuántica

La notación de Dirac es la más comúnmente usada en mecánica cuántica. Esta notación consta de dos elementos, los "*kets*" y los "*bra*", que están representados por $|x\rangle$ y $\langle x|$ respectivamente. Los kets denotan vectores columna y los bra representan al conjugado transpuesto complejo de $|x\rangle$.

En la notación de Dirac se pueden realizar operaciones como el producto interior y el producto exterior. El producto interior está denotado por la multiplicación de "*bra-kets*", $\langle y| \cdot |x\rangle$ o $\langle y|x\rangle$, lo que da como resultado un valor númerico. El producto exterior está denotado por la multiplicación "*kets-bra*", $|x\rangle\langle y|$, resultando una matriz cuadrada. En particular si $|x\rangle = [1, 0]^T$ y $|y\rangle = [0, 1]^T$ entonces, $\langle x|x\rangle = 1$, $\langle x|y\rangle = 0$ y su producto exterior es el siguiente:

$$
\begin{aligned}
|x\rangle\langle x| &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \\
|x\rangle\langle y| &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}
\end{aligned}
$$

1.2.2. Multiplicación de matrices

Una de las operaciones más importantes en el paradigma de la computación cuántica es la aplicación de Qu-compuertas. Esta aplicación se traduce generalmente a la multiplicación de una matriz por un vector, lo cual se puede llevar a cabo de diferentes

maneras. Es de nuestro interés aplicar esta operación eficientemente, por lo que se han analizado los algoritmos más representativos en el estado del arte.

El algoritmo clásico para multiplicar 2 matrices cuadradas de 2×2 realiza 8 multiplicaciones y 4 sumas. Si tomamos a esta multiplicación como la base de la recurrencia para multiplicar matrices cuadradas, entonces el orden de esta operación será $O(n^3)$.

Strassen, en su tesis doctoral atacó este problema y propuso un algoritmo que realiza 3 multiplicaciones a cambio de realizar 18 sumas, pero como la operación básica más costosa es la multiplicación, esto se tradujo en la disminución en el orden de la multiplicación de matrices cuadradas a $O(n^{\log_2 7})$. El algoritmo de Strassen fue mejorado por *Winograd*, el cual sigue la misma idea de Strassen, dividir las matrices a multiplicar y realizar sumas en vez de multiplicaciones. El cambio se encuentra en que Winograd une términos semejantes, lo que reduce en 3 la cantidad de sumas a realizar.

En [15] se demuestra que son necesarias y suficientes las 15 sumas y 7 multiplicaciones recursivas del algoritmo de Winograd para obtener la multiplicación de dos matrices cuadradas. Este resultado es importante ya que nos dice implícitamente que el orden de la multiplicación de una matriz por un vector es de $O(n^2)$. Por lo tanto, la multiplicación de una compuerta cuántica por un quregistro es de orden $O(n^2)$ ya que ésta se puede ver como la multiplicación de una matriz por un vector.

Estos algoritmos se han implementado en cluster's como en [16, 17], donde se compara el desempeño de cada uno de ellos. Estas implementaciones en paralelo tienen una cierta dificultad dependiendo de la cantidad de memoria y la forma de la recursión de los algoritmos, en [18] se estudia la manera en que el algoritmo clásico de multiplicación, de Strassen y de Winograd acceden a los datos, dando una idea de la dificultad al ser paralelizados. Se concluye que el algoritmo estándar es mucho más fácil de llevar a la paralelización que los algoritmos de Strassen y Winograd. También se observa que el error obtenido en estos algoritmos es mayor que en el procedimiento clásico.

Se han diseñado algoritmos de órdenes más bajos para la multiplicación de matrices como en [19], en el cual se dan algoritmos de $O(n)$, $O(1)$, $O(log_2 n)$ pero teniendo la desventaja de que necesitan exactamente n^2, n^3 y $n^{log_2 7}$ procesadores respectivamente, para su correcto funcionamiento.

1.2.3. Producto tensorial

La computación cuántica trabaja con espacios vectoriales. De aquí se puede ver la importancia del producto tensorial, ya que éste combina dos espacios vectoriales de dimensiones arbitrarios y los une, dejando un espacio vectorial de dimensión igual a la suma de las dimensiones individuales. Formalmente, suponga que V y W son espacios vectoriales complejos de dimensión m y n respectivamente. Sea $V \otimes W$ el producto tensorial de V y W un espacio vectorial complejo de dimensión mn. Los elementos de $V \otimes W$ son una combinación lineal de *productos tensoriales* $|v\rangle \otimes |w\rangle$ con

$|v\rangle$ en V y $|w\rangle$ en W. En particular, si $(|i\rangle)_{i\in I}$ y $(|j\rangle)_{j\in J}$ son bases ortonormales para los espacios V y W entonces $(|i\rangle \otimes |j\rangle)_{i\in I, j\in J}$ es una base para $V\otimes W$.

Gráficamente:

$$V \otimes W \equiv \begin{bmatrix} V_{11}W & V_{12}W & \dots & V_{1n}W \\ V_{21}W & V_{22}W & \dots & V_{2n}W \\ \vdots & \vdots & \ddots & \vdots \\ V_{m1}W & V_{m2}W & \dots & V_{mn}W \end{bmatrix}$$

Para todo escalar $z \in \mathbb{C}$, $|v_1\rangle, |v_2\rangle \in V$ y $|w_1\rangle, |w_2\rangle \in W$ el producto tensorial satisface las siguientes condiciones:

1. $z(|v_1\rangle \otimes |w_1\rangle) = (z|v_1\rangle) \otimes |w_1\rangle = |v_1\rangle \otimes (z|w_1\rangle)$

2. $(|v_1\rangle + |v_2\rangle) \otimes |w_1\rangle = |v_1\rangle \otimes |w_1\rangle + |v_2\rangle \otimes |w_1\rangle$

3. $|v_1\rangle \otimes (|w_1\rangle + |w_2\rangle) = |v_1\rangle \otimes |w_1\rangle + |v_1\rangle \otimes |w_2\rangle$

1.2.4. Qubits y quvectores

Una de las características que diferencian a la computación cuántica con respecto a la computación clásica es el uso de diferente representación en su información. En computación clásica la información se representa con 1's y 0's también llamados bits, mientras que en computación cuántica esto se hace con Qubits. Una de las características del qubit es que su valor puede ser 0 o 1 con una cierta probabilidad.

Formalmente, sea $\mathbb{C}$ el campo de los números complejos y sea $\mathbb{H}$ un espacio de Hilbert, entonces la base canónica del espacio $\mathbb{H}_1 = \mathbb{C}^2$ consta de los vectores $\mathbf{e}_0 = [1\ 0]^T$ y $\mathbf{e}_1 = [0\ 1]^T$, donde T denota la transpuesta. Si $z_0, z_1 \in \mathbb{C}$ son complejos tales que $|z_0|^2 + |z_1|^2 = 1$, entonces $z_0\mathbf{e}_0 + z_1\mathbf{e}_1$ es un estado, llamado bit cuántico o *qubit*.

Identificamos al primer vector básico $\mathbf{e}_0$ con el valor de verdad *falso*, o *cero*, y al segundo $\mathbf{e}_1$ con el valor de verdad *verdadero*, o *uno*. Así pues, cada estado es una *"superposición"* de ambos valores cero y uno.

Para cada $n > 1$, definimos recursivamente $\mathbb{H}_n = \mathbb{H}_{n-1} \otimes \mathbb{H}_1$. De aquí resulta que $\dim(\mathbb{H}_n) = 2^n$ y una base de este espacio es $B_{\mathbb{H}_n} = (\mathbf{e}_{\varepsilon_{n-1}\cdots\varepsilon_1\varepsilon_0})_{\varepsilon_{n-1},\cdots,\varepsilon_1\varepsilon_0\in\{0,1\}}$, donde, puesto de manera recursiva, $\mathbf{e}_{\varepsilon_{n-1}\cdots\varepsilon_1\varepsilon_0} = \mathbf{e}_{\varepsilon_{n-1}\cdots\varepsilon_1} \otimes \mathbf{e}_{\varepsilon_0}$.

Para $i < j, [\![i, j]\!]$ denotará el conjunto de enteros $\{i, i+1, \dots, j-1, j\}$. Cada índice $i \in [\![0, 2^n - 1]\!]$ puede escribirse en binario como una cadena de bits de longitud $n : i = (\varepsilon_{n-1}\cdots\varepsilon_1\varepsilon_0)_2$. Así pues identificaremos a cada índice con la cadena que lo representa: $i \leftrightarrow \varepsilon = \varepsilon_{n-1}\cdots\varepsilon_1\varepsilon_0$. Mediante esta identificación: $[\![0, 2^n - 1]\!] \approx \{0,1\}^n$.

Si $\mathbf{z} \in \{\mathbf{v} \in \mathbb{C}^m \mid 1 = \mathbf{v}^H\mathbf{v} =: \langle \mathbf{v} \mid \mathbf{v}\rangle \ \&\ m = 2^n\}$ es un vector en la esfera unitaria euclidiana en $\mathbb{H}_n$ entonces $\sum_{\varepsilon\in\{0,1\}^n} z_\varepsilon \mathbf{e}_\varepsilon$ es un estado correspondiente a una palabra de información de longitud n, es decir, es la concatenación de n qubits o *quvector* de dimensión n es un *n-quvector*.

1.2.5. Qu-compuertas

Los operadores en computación cuántica son llamados *Quantum gates* o Qu-compuertas, los cuales son los análogos de las compuertas clásicas AND, OR, etc. Una de las principales diferencias es que las qu-compuertas pertenecen a espaciós vectoriales y por tal motivo se les puede representar como matrices.

Formalmente, sea $\mathbb{C}$ el campo de los números complejos, y para cada m, n sea $\mathbb{C}^{m\times n}$ el espacio de matrices de orden $m\times n$, es decir, de matrices con m renglones y n columnas, con entradas números complejos. Para cada matriz $M=(m_{ij})_{i,j}\in\mathbb{C}^{m\times n}$ su *transpuesta hermitiana* es $M^H=(m^H_{ji})_{ji}\in\mathbb{C}^{n\times m}$ donde para cada pareja de índices $(i,j)\in[\![0,m-1]\!]\times[\![0,n-1]\!]$, $m^H_{ji}=\overline{m_{ij}}$ (si $z=a+ib\in\mathbb{C}$ es un número complejo, naturalmente $\overline{z}=a-ib\in\mathbb{C}$ es su *conjugado*). Una matriz $M=(m_{ij})_{i,j}\in\mathbb{C}^{m\times n}$ se dice ser *unitaria* si $M^HM=\mathbf{1}_{nn}$, donde $\mathbf{1}_{nn}$ denota a la matriz *identidad* de orden $n\times n$.

Al subconjunto consistente de los vectores columnas unitarias en $\mathbb{C}^{m\times 1}$ (es decir, el espacio de vectores columnas de dimensión m) se le llama *conjunto de estados* de un sistema físico cerrado y la dimensión m se conoce como el *grado de libertad* del sistema. En $\mathbb{C}^{m\times 1}$ se tiene que cada estado es un vector en la esfera euclidiana unitaria de $\mathbb{C}^m$. Sea pues $E_m=\{\mathbf{v}\in\mathbb{C}^m \mid 1=\mathbf{v}^H\mathbf{v}=:\langle\mathbf{v}\mid\mathbf{v}\rangle\}$ el conjunto de estados.

Ahora, sea $U\in\mathbb{C}^{m\times n}$ una matriz unitaria cuadrada de orden $m\times n$, U determina una transformación ortogonal $\mathbb{C}^m\to\mathbb{C}^m:\mathbf{v}\mapsto U\mathbf{v}$. De hecho, al restringirla a E_m se tiene una transformación $E_m\to E_m$. Entonces U se dice ser una *compuerta cuántica*. Un *algoritmo cuántico* es la composición de un número finito de compuertas cuánticas.

Compuertas cuánticas básicas

Sea $\mathbb{H}_1$ el espacio donde viven las *compuertas básicas*, llamadas también *operadores cuánticos*, las cuales son:

Identidad. $I=\begin{bmatrix}1&0\\0&1\end{bmatrix}$. $I:\mathbb{H}_1\to\mathbb{H}_1$ es el operador identidad.

Pauli-X. $X=\begin{bmatrix}0&1\\1&0\end{bmatrix}$. Se tiene $X:\begin{bmatrix}z_0\\z_1\end{bmatrix}\mapsto\begin{bmatrix}z_1\\z_0\end{bmatrix}$. X es unitaria y tiene como función permutar señales, es de hecho “una reflexión a lo largo de la diagonal principal”. También conocida como *negación*.

Pauli-Y. $Y=\begin{bmatrix}0&-i\\i&0\end{bmatrix}$. Se tiene $Y:\begin{bmatrix}z_0\\z_1\end{bmatrix}\mapsto\begin{bmatrix}-iz_1\\iz_0\end{bmatrix}$. Y es unitaria y tiene como función intercambiar las probabilidades, rotar 90 grados en sentido de las manecillas del reloj al primer elemento y 90 grados en sentido opuesto de las manecillas del reloj al segundo elemento.

Pauli-Z. $Z=\begin{bmatrix}1&0\\0&-1\end{bmatrix}$. Se tiene $Z:\begin{bmatrix}z_0\\z_1\end{bmatrix}\mapsto\begin{bmatrix}z_0\\-z_1\end{bmatrix}$. Z es unitaria y tiene como función cambiar el signo del segundo elemento del vector de entrada. También

conocida como *cambio de signo.*

Hadamard. $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. Se tiene $N : \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} \mapsto \frac{1}{\sqrt{2}}\begin{bmatrix} z_0 + z_1 \\ z_0 - z_1 \end{bmatrix}$. H es unitaria y tiene como función "reflejar el plano respecto al eje x y rotar luego un ángulo de $\frac{\pi}{4}$ radianes en sentido opuesto a las manecillas del reloj".

Compuertas cuánticas para multiples qubits

Sea M una compuerta básica, entonces $M^{\otimes n}$ es el producto tensorial de la compuerta básica M consigo misma n veces, es una compuerta en $\mathbb{H}_n$. Esta es una manera de construir compuertas cuánticas para multiples qubits.

El prototipo para las compuertas cuánticas de múltiples qubits es el *Not-controlado* o *CNOT*. Esta compuerta tiene dos qubits de entrada, conocidos como el qubit de *control* y *objetivo*, respectivamente. El circuito que representa a la compuerta CNOT es mostrada en la Figura 1.1.

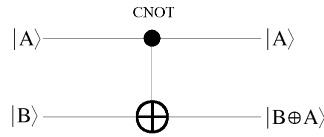

Figura 1.1: Compuerta CNOT, prototipo de las compuertas para multiplies qubits.

La línea de arriba representa al qubit de control, mientras que la línea de abajo representa al qubit objetivo. La acción de la compuerta es descrita como:

Sea **CNOT** : $\mathbb{H}_2 \to \mathbb{H}_2$ la transformación lineal que sobre los vectores básicos actúa $\mathbf{e}_x \otimes \mathbf{e}_y \mapsto \mathbf{e}_x \otimes \mathbf{e}_{x\oplus y}$, es decir, si el qubit de control es 0 entonces el qubit objetivo es dejado igual, si el qubit de control es 1, entonces el qubit objetivo es sumado al qubit de control modulo 2.

En símbolos, la función CNOT actúa como:

$$|00\rangle \mapsto |00\rangle; |01\rangle \mapsto |01\rangle; |10\rangle \mapsto |11\rangle; |11\rangle \mapsto |10\rangle; \tag{1.1}$$

La compuerta CNOT debe cumplir con ser *unitaria*, lo cual se puede verificar rápidamente de su representación matricial, respecto a la base canónica de $\mathbb{H}_2$ mostrada en la ecuación 1.2.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.2}$$

La compuerta CNOTI también pertenece al campo $\mathbb{H}_2$, es decir, es una compuerta para dos qubits. Esta compuerta es semejante al CNOT ya que tiene dos qubits de

entrada, pero estos son conocidos como el qubit *objetivo* y *control*, respectivamente; es decir, es el CNOT pero con los “roles cambiados”. El circuito que representa a la compuerta CNOTI es mostrada en la Figura 1.2.

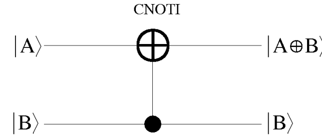

Figura 1.2: Compuerta CNOTI, es la compuerta CNOT con los roles cambiados.

La línea de arriba representa al qubit objetivo, mientras que la línea de abajo representa al qubit de control. La acción de la compuerta se describe como:

Sea **CNOTI** : $\mathbb{H}_2 \rightarrow \mathbb{H}_2$ la transformación lineal que sobre los vectores básicos actúa $\mathbf{e}_x \otimes \mathbf{e}_y \mapsto \mathbf{e}_{x \oplus y} \otimes \mathbf{e}_y$, es decir, si el qubit de control es 0 entonces el qubit objetivo es dejado igual, si el qubit de control es 1, entonces el qubit objetivo es sumado al qubit de control módulo 2.

En símbolos la función CNOTI es como:

$$|00\rangle \rightarrow |00\rangle; |01\rangle \rightarrow |11\rangle; |10\rangle \rightarrow |10\rangle; |11\rangle \rightarrow |01\rangle; \tag{1.3}$$

Así como la compuerta CNOT, la compuerta CNOTI debe cumplir con ser *unitaria*, lo cual se puede verificar rápidamente de su representación matricial, respecto a la base canónica de $\mathbb{H}_2$, mostrada en la ecuación 1.4.

$$CNOTI = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{1.4}$$

La compuerta CNOTI se genera conjugando las entradas y salidas de la compuerta CNOT con la compuerta *Hadamard*, es decir, haciendo el producto tensorial de la compuerta *Hadamard* y multiplicándolo por las entradas y salidas de la compuerta CNOT. Esto se ve gráficamente en la Figura 1.3.

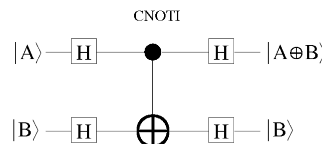

Figura 1.3: Generación de la compuerta CNOTI.

Como se ve, con las compuertas básicas y la compuerta CNOT, se pueden generar compuertas unitarias en $\mathbb{H}_2$, más aun, estas compuertas son llamadas compuertas

universales en computación cuántica, es decir *"cualquier compuerta unitaria en $\mathbb{H}_n$ puede ser generada por las compuertas básicas y la compuerta CNOT"* [20].

Note que $N^{\otimes n}$ y $H^{\otimes n}$ son compuertas en $\mathbb{H}_n$ y que $N^{\otimes n}$ actúa como el complemento a $2^n - 1$ es decir en los vectores básicos se tiene

$$N^{\otimes n}(\mathbf{e}_{\varepsilon_{n-1}\dots\varepsilon_1\varepsilon_0}) = \mathbf{e}_{\delta_{n-1}\dots\delta_1\delta_0} \tag{1.5}$$

donde $(\varepsilon_{n-1}\dots\varepsilon_1\varepsilon_0)_2 + (\delta_{n-1}\dots\delta_1\delta_0)_2 = 2^n - 1$.

Observemos también que

$$\begin{aligned} \mathbf{H}^{\otimes 1}(\mathbf{e}_0) &= \frac{1}{\sqrt{2}}(\mathbf{e}_0 + \mathbf{e}_1) \\ \mathbf{H}^{\otimes 2}(\mathbf{e}_{00}) &= \frac{1}{(\sqrt{2})^2}(\mathbf{e}_{00} + \mathbf{e}_{01} + \mathbf{e}_{10} + \mathbf{e}11) \end{aligned}$$

y de manera general

$$H^{\otimes n}(\mathbf{e}_{0\dots0}) = \frac{1}{(\sqrt{2})^n}\left(\sum_{\varepsilon\in\{0,1\}^n} \mathbf{e}_\varepsilon\right) \tag{1.6}$$

es decir, el operador $H^{\otimes n}$ aplicado al primer vector básico $\mathbf{e}_{0\dots0}$ produce el estado que promedia a todos los demás con pesos uniformes.

1.2.6. Mediciones

Para saber el valor de una variable en una computadora clásica basta con leer ese valor e imprimirlo. Este no es el caso en las computadoras cuánticas, la medición implica más que sólo ver el valor de la variable que se desee. Si después de aplicar ciertas transformaciones sobre un quregistro y medirlo, queda determinado y no es posible revertir esta operación, es decir, que si se quiere volver al estado indeterminado, la única manera es volviendo a ejecutar todo el procedimiento.

Una medición cuántica sobre un estado en $\mathbb{H}_n$ es descrito por un conjunto de operadores de medición $\{M_i\}$, el índice i se refiere a la medición del estado base $\mathbf{e}_i \in \mathbb{H}_n$, donde

$$M_i = |i\rangle\langle i|,\ i \in [\![0, 2^n - 1]\!] \tag{1.7}$$

cumpliendo con la ecuación

$$\sum_{i=0}^{2^n-1} M_i^H M_i = \mathbf{1}_{nn} \tag{1.8}$$

Si el estado del sistema cuántico es ψ la probabilidad de que el estado medido sea $\mathbf{e}_i \in \mathbb{H}_n$ está dado por

$$p(i) = \langle \psi | M_i^H M_i | \psi \rangle \tag{1.9}$$

el nuevo estado queda determinado por

$$\frac{M_i|\psi\rangle}{\sqrt{\langle \psi | M_i^H M_i | \psi \rangle}} \tag{1.10}$$

se cumple que las probabilidades suman 1:

$$1 = \sum_i p(i) = \sum_i \langle \psi | M_i^H M_i | \psi \rangle \tag{1.11}$$

De hecho el proceso de medición se realiza al final de cualquier algoritmo cuántico, así que el último estado actual después de una medición es el estado *final*.

Mediciones parciales

Si el estado $\psi = \psi_1 \otimes \ldots \otimes \psi_j \otimes \ldots \otimes \psi_n$, donde la dimensión de ψ_i es k_i, $i \in [\![0, 2^n - 1]\!]$, es posible hacer una medición en uno solo de sus factores. Supóngase que se quiere medir el factor ψ_j, entonces basta con modificar el operador de medición por la ecuación

$$M = I^{\otimes x} \otimes M_j \otimes I^{\otimes y}$$

donde se cumple que

$$\begin{aligned} M_j &= |i\rangle\langle i|, i \in [\![0, k_j - 1]\!] \\ 2^x &= \prod_{i=0}^{j-1} k_i \\ 2^y &= \prod_{i=j+1}^{n} k_i \end{aligned} \tag{1.12}$$

Este tipo de mediciones es útil cuando sólo nos interesa, una parte del estado actual. En el algoritmo de factorización de números grandes que propuso Peter Shor, se utiliza esta técnica de medición.

1.2.7. Gramáticas

La base de la solución que se propone en esta tesis para el problema planteado, es una gramática. Las gramáticas formales son sistemas de manipulación simbólica que permiten generar cadenas de símbolos, llamadas por esto bien formadas, o bien reconocen cuándo una cadena dada está, en efecto, bien formada.

Formalmente, sea T un conjunto de símbolos terminales y sea V un conjunto de símbolos variables. La unión de ellos, $A = V \cup T$, es un alfabeto de la gramática.

A^* es el diccionario sobre A y consta de todas las palabras de longitud finita, con símbolos en A. A^+ coincide con A^*, excepto que no contiene la palabra vacía *nill*.

Una *regla de producción* es un elemento del producto cartesiano $A^+ \times A^*$. Si $(a, c) \in A^+ \times A^*$ escribimos $a \rightarrow c$ y decimos que a es el *antecedente* y c es el *consecuente* de la regla $a \rightarrow c$. Sea $P \subset A^+ \times A^*$ un conjunto de reglas de producción. Sea $S \in V$ un símbolo variable distinguido, llamado *inicial*. El sistema $G = (V, T, P, S)$, se dice ser una *gramática formal*. Las reglas de producción transforman palabras en otras:

$$\forall c, y \in A^* : x \ da \ y \Leftrightarrow \exists p, q \in A^*, (a \rightarrow c) \in P : (x = paq) \& (y = pcq). \tag{1.13}$$

La *cerradura reflexivo-transitiva* de la relación "da" define la relación de derivación:

$$\forall x, y \in A^* : x \ deriva \ y \ si \ x(da)^* y \tag{1.14}$$

El *lenguaje generado* por la gramática consta de todas las palabras que se derivan del símbolo inicial y que solo contienen símbolos terminales:

$$Lenguaje(G) = \{x \in T^* | S \ deriva \ x\} \tag{1.15}$$

1.3. Organización de la tesis

El documento de tesis está organizado de la forma siguiente: en el capítulo 2 se estudia la forma de multiplicar dos matrices cuadradas de manera eficiente así como los algoritmos con los que se hicieron las operaciones básicas de computación cuántica, en el capítulo 3 se diseña el esquema general de nuestra solución así como una gramática capaz de generar a los algoritmos cuánticos, en el capitulo 4 se muestra la forma en que fue implementada nuestra solución usando las operaciones desarrolladas en el capitulo 2, en el capítulo 5 se presentan tres casos de estudio, los cuales son el *reverso de un quvector*, la *Transformada Discreta de Fourier* y el algoritmo de *Shor*, y finalmente en el capítulo 6 se presentan las conclusiones y trabajo futuro.

Capítulo 2

Procesamiento de las compuertas cuánticas

La forma de representar a las qucompuertas y a los estados es a través de matrices y vectores respectivamente, por lo que hacer *evolucionar* a un estado ψ_1 a un estado ψ_2 debido a una qucompuerta Q_i, se traduce a multiplicar la matriz que representa a la qucompuerta Q_i por el vector columna que representa al estado ψ_1. De aquí la importancia que tiene la multiplicación de matrices en computación cuántica, pero esta operación no es la única, ya que para generar las qucompuertas se necesitan las siguientes operaciones:

1. Hermitiana de una matriz.

2. Suma de matrices.

3. Multiplicación de matrices.

4. Producto tensorial de matrices y vectores.

las cuales llamaremos *operaciones básicas*, por lo que si se les implementan de manera óptima obtendremos una reducción en el costo de ejecución del algoritmo cuántico.

2.1. Representación de las matrices

Las qucompuertas trabajan en el mismo espacio que los quvectores, por lo que una qucompuerta para un estado $\psi_i \in \mathbb{H}^n$, estará representada por una matriz cuadrada de orden $(2^n \times 2^n)$. Si observamos la Tabla 2.1, la cual tiene los primeros valores de n, notaremos que si se implementan las operaciones básicas de suma y multiplicación de matrices utilizando memoria dinámica, tendremos el problema de insuficiencia de memoria debido al tamaño de las matrices, lo que puede significar que la ejecución sea bastante deficiente, la no terminación e incluso la terminación súbita dependiendo del sistema operativo.

Valor de n	Dimensión de la qucompuerta	Memoria necesaria (Mb)
3	8×8	1/1024
4	16×16	1/256
5	32×32	1/64
6	64×64	1/16
7	128×128	1/4
8	256×256	1
9	512×512	4
10	1024×1024	16
11	2048×2048	64
12	4096×4096	256
13	8192×8192	1024

Tabla 2.1: Se muestra el tamaño y la cantidad de memoria necesarios para las qucompuertas, que actúan en el espacio $\mathbb{H}^n$. Las entradas de las matrices son números complejos, cada uno de ellos está representado por dos `doubles` que ocupan 8 bytes cada uno, por lo que la cantidad de memoria necesaria para representar al operador crece de forma exponencial con respecto a n.

Una posible solución es obtener más recursos, pero el problema se soluciona parcialmente, ya que estas matrices crecen de manera exponencial, por lo cual con unos cuantos valores más de n se tendría el mismo problema.

Debido a esto se decidió implementar todos los operadores y los estados cuánticos utilizando un sistema de almacenamiento más grande, la opción lógica es utilizar archivos, ya que éstos poseen capacidades mucho mayores que la memoria RAM. Con lo cual el primer problema a resolver sería el cómo representar a una matriz en un archivo de manera eficiente.

Una matriz puede ser representada de diferentes formas en un archivo ya que existen ciertos parámetros los cuales dependiendo de la elección del usuario pueden dar origen a ciertas representaciones diferentes una de la otra. Las matrices con las que se trabaja en computación cuántica tienen como entradas números complejos, por lo que es importante considerar un número adecuado de dígitos por flotante que se van a guardar en el archivo.

Si se quisiera una gran precisión en los números que componen a la matriz, entonces se necesitaría guardar una gran cantidad de dígitos, pero cabe la posibilidad de que algún compilador, de años atrás, utilice una cantidad pequeña de bits para representar a sus flotantes, es decir, que la precisión con la que utiliza los números sea menor, lo que causaría un error de desbordamiento, por ejemplo puede ocurrir que se escojan 15 dígitos para representar a los flotantes y el compilador solo utilice 8 bits, lo cual claramente tiene menos precisión que la se pide. Para evitar este problema se puede elegir una pequeña cantidad de dígitos, pero esto también es un problema, ya que hay operaciones, como la transformada de Fourier, que requieren la mayor

precisión posible, lo que es muy frecuente en computación cuántica.

La solución que propusimos es muy sencilla, guardar el número con la máxima precisión según el compilador utilizado, lo cual se logra guardando los bits que utiliza el compilador para representar estos números, en vez de los caracteres que los representan, por lo que los archivos donde se encuentran las matrices con las que hemos trabajado son archivos binarios. Esta forma de guardar a los números en los archivos nos ofrece otras ventajas aparte de la precisión, las cuales son:

- La fácil y rápida lectura/escritura de los números
- Evita escribir un símbolo especial para el signo de punto flotante.
- Nos evita escribir un símbolo especial que represente la separacion entre números, debido a que se conoce la cantidad de bits de los que consta cada número.
- No requiere un caracter especial para representar el signo del número utilizado.

Otro problema que se presenta al utilizar archivos, es decidir cómo van a estar ordenados los números. Se podría guardar a la matriz de manera similar a como se muestra tradicionalmente, pero esto implica que se tengan que escribir saltos y retornos de lńeas con la única función de identificar donde empieza y donde termina cada renglón de las matrices. Este esquema sería muy ineficiente ya que las matrices constan de muchos datos y por consiguiente los archivos donde se guarden las matrices tendrían una gran cantidad de caracteres que sólo servirían para separar a los renglones de las matrices guardadas.

Para evitar el problema de los caracteres que no son *útiles*, decidimos almacenar en el encabezado primeramente en los archivos dos enteros, el primero de ellos representa la cantidad de renglones de los que consta la matriz y el segundo las columnas, seguido de los valores de los renglones de la matriz escritos de manera continua como si de un vector se tratara. Con este esquema es posible reconstruir a la matriz sin ambigüedad. Esta representación de una matriz en un archivo se ve graficamente en la Figura 2.1. Para el caso particular de los estados cuánticos se recurre al mismo esquema, ya que un estado ψ_i puede representarse como una matriz de una sola columna, es decir, el segundo entero guardado en el esquema es 1.

Una vez solucionado el problema de la representación de las qucompuertas y de los estados cuánticos, se necesita tener las operaciones básicas diseñadas de tal manera que puedan trabajar con matrices que tengan el formato descrito anteriormente.

Las operaciones básicas, en general, necesitan diferentes datos para poder funcionar de manera correcta, por lo que no basta implementar un sólo lector de archivos para acceder a los datos y subirlos a memoria RAM ya que, para algunas operaciones, se tardaría mucho tiempo en acceder a los valores correctos. Es por este motivo que hemos diseñado algoritmos que optimizan la recuperación de los datos en los archivos para cada una de las funciones básicas.

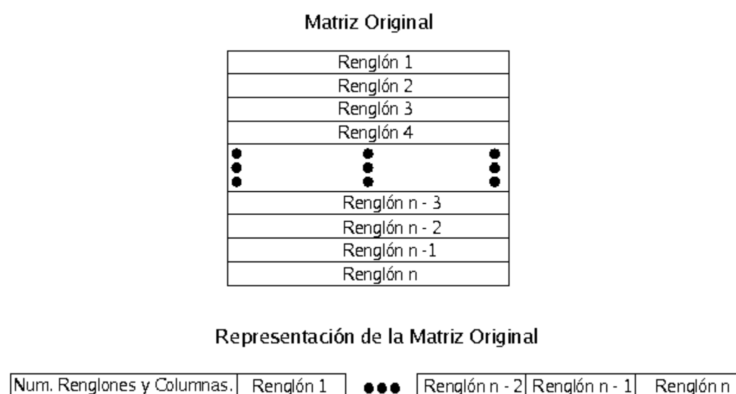

Figura 2.1: Representación de una matriz en archivos. Esta representación sirve de igual manera para representar a estados cuánticos, ya que estos son matrices de n renglones y una columna.

2.2. Hermitiana de una matriz

Una operación que es muy utilizada en computación cuántica es la *Hermitiana* de una matriz que se define como la transpuesta conjugada compleja de una matriz, es decir, se deben aplicar dos operaciones: la *transpuesta de la matriz argumento* y su *conjugado complejo*. Es por tal motivo que diseñamos un algoritmo para realizar la operación transpuesta y uno más para la operación conjugado complejo haciendo que los accesos a disco se efectúen de manera óptima.

2.2.1. Transpuesta de una matriz

La matriz de la que se quiere obtener la transpuesta tiene el formato definido en la Figura 2.1. Si las matrices con las que se van a afectuar los cálculos tienen una dimensión en bytes mayor a la cantidad de memoria RAM, entonces tratar de implementar la transpuesta usando solo memoria dinámica no sería eficiente debido a la cantidad de memoria necesaria que tendría que utilizar la computadora, lo cual fue planteado en la subseción anterior, por lo que se tiene que proceder de tal manera que se lea del archivo una cantidad de datos que no sobrepase el máximo de RAM.

Debido a esto, implementamos el algoritmo de la Tabla 2.3. Donde las variables M_i, M_j, representan a los nombres de los archivos donde se encuentran las matrices M_i, M_j. La función `RenCol` (M_i) devuelve el número de renglones y columnas de la matriz M_i, `Leer`(x, M_i) lee x complejos de la matriz M_i, siempre renglones completos, `Esc`$(dato, M_i)$ escribe el valor almacenado en la variable dato en el archivo con el nombre M_i, `Jmp`(k, M_i) salta k complejos en la matriz M_i a partir de la posición actual, `Ubica`(k, M_i) se ubica en la posición k de la matriz M_i, esta operación es *rápida* debido a que la matriz está representada como un vector. El algoritmo lee los datos de M_i y los escribe en M_j, desplazándose dentro del archivo de tal manera que

Entrada: dim, x, y, M_i.
Salida: $x \mid y$ datos $\in M_i$
Procedimiento Vdatos
  Si $y > x$
    $v1 = leer(x, M_i)$
    $dim = x$, $y = y - x$
  Si no
    $v1 = leer(y, M_i)$
    $dim = y$, $y = 0$
  Devolver $v1$
  Terminar

Tabla 2.2: Algoritmo para leer un vector de datos en M_i. La variable x representa la cantidad de datos que el algoritmo puede leer sin afectar la ejecución del programa, la variable y es la cantidad de complejos por leer. Si $y < x$ se leen y datos si no x datos de la matriz M_i.

la reconstrucción nos muestre la transpuesta de M_i.

Si pensamos en las operaciones de suma, multiplicación y producto tensorial de archivos se puede ver que se necesita trabajar con tres matrices al mismo tiempo, las matrices argumentos y resultante, por lo que se recomienda que la máxima cantidad de datos leídos a la vez sea x al rededor de $\frac{RAM}{3*sizeof(complejo)}$, lo que significa dividir a la memoria RAM en partes iguales entre las matrices que se van a tener en memoria al mismo tiempo.

2.2.2. Conjugado complejo de una matriz

Si $z = a + ib \in \mathbb{C}$ es un número complejo, entonces $\overline{z} = a - ib$ es su conjugado. El algoritmo que diseñamos para obtener el conjugado complejo de una matriz se muestra en la Tabla 2.4. Se lee cada entrada de la matriz original, se le aplica la función *conj* que cambia de signo al valor imaginario del número complejo y se reescribe en el archivo de salida.

Debido a la forma en que se representan las matrices en los archivos, se puede realizar la operación conjugado complejo de manera óptima, ya que los datos leídos no contienen caracteres *inútiles* como el punto flotante, además de que no se tiene que decidir que parte es la real y cual la imaginaria del número, debido a nuestra representación. La escritura también ha sido optimizada debido a que no hay que tener un indicador de donde termina o empieza un renglón, por lo que se puede leer y escribir bloques completos de dimensión x en los archivos.

Entrada: M_i, M_j.
Salida: $M_j = M_i^T$
Procedimiento Transpuesta_de_Matrices
  $\{Ren, Col\} = RenCol(M_i)$
  $Esc(Col, M_j),\ Esc(Ren, M_j)$
  $Tam = Ren \times Col$
  $Ini = cnt = 0$
  Mientras $Tam > 0$
    $v1 = Vdatos(leido, x, Tam, M_i)$
    $i = 0$
    Mientras $leido > i$
      $Esc(v1\left[i\right], M_j)$
      $i++, cnt++$
      $Jmp(Col, M_j)$
      Si $cnt \% Col == 0$
        $Ini++$
        $Ubica(Ini, M_j)$
  Terminar

Tabla 2.3: Algoritmo para obtener la transpuesta de una matriz. Hay que tener en cuenta que las lecturas evitan tomar los dos primero enteros que indican la dimensión de la matriz.

Entrada: M_i, M_j.
Salida: $M_j = \overline{M}_i$
Procedimiento Conjugado_de_Matrices
  $\{Ren, Col\} = RenCol(M_i)$
  $Esc(Ren, M_j),\ Esc(Col, M_j)$
  $Tam = Ren \times Col$
  Mientras $Tam > 0$
    $v1 = Vdatos(leido, x, Tam, M_i)$
    $i = 0$
    Mientras $leido > i$
      $Esc(conj(valor\left[i\right]), M_j)$
      $i++$
  Terminar

Tabla 2.4: Algoritmo para obtener el conjugado de las entradas de una matriz. La palabra *conj* representa a la función conjugado de un número complejo.

2.3. Suma de matrices

Las matrices que se trabajan en computación cuántica pertenecen a espacios de Hilbert y la operación *suma de matrices* preserva el espacio de sus operadores, es decir, se conserva la dimensión de las matrices, es por eso que la aplicación de esta operación debe hacerse de manera eficiente.

Supóngase que se tienen las qucompuertas $Q_i, Q_j \in \mathbb{H}_n$ y se quiere aplicar la suma de ellas para obtener la qucompuerta Q_k. Obsérvese que para sumar las qucompuertas se necesitan que los valores de los que constan las matrices que las caracterizan, así como la matriz resultante se encuentren en memoria, lo que nos lleva al primer problema, la cantidad de memoria que se debe tener en una computadora que utilice qucompuertas como operadores debe ser suficiente para realizar la de suma de matrices de dimensión 2^n, por lo que la memoria de la computadora con la que se trabaje debe constar de $3 * 2^n * \texttt{sizeof}(complejo)$ de RAM.

Con frecuencia no se cuenta con la memoria suficiente para almacenar las matrices, por lo que implementamos un algoritmo que se adapta a la cantidad máxima de memoria RAM de la que dispone la computadora (ver tabla 2.5). El algoritmo lee por fragmentos a los dos operandos M_i y M_j, suma las partes de la matrices leídas y escribe el resultado en el archivo de salida, es decir, obtiene a M_k sumando por partes a las dos matrices de las qucompuertas. Si se cambia la operación *suma* por *resta* se obtiene un algoritmo para restar matrices, esta es la función del entero *oper* en el algoritmo. Si $oper = 1$ entonces el algoritmo suma, en otro caso resta.

La eficiencia del algoritmo radica en el hecho de que no se necesitan caracteres especiales para la representación de las matrices (ya que se guardan por renglones) y llevar a cabo la suma de matrices sólo requiere sumar los elementos referenciados por los apuntadores a los archivos argumento. Por otro lado, si las matrices se guardaran de forma renglón-columna, es necesario introducir caracteres especiales, por ejemplo saltos de línea y/o retornos de carro. Entonces al realizar la suma se tendrían que considerar los caracteres especiales sumando pasos en la ejecución del algoritmo.

2.4. Multiplicación de matrices

Como se mencionó, aplicar una qucompuerta a un estado, se traduce a la multiplicación de una matriz por un vector columna. Pero la multiplicación no sólo se encarga de la aplicación de qucompuertas, también es responsable, junto a las otras operaciones básicas de generar dichos operadores. Por lo que esta operación resulta ser una de las más importantes en el ambiente de computación cuántica.

Existen diversos algoritmos para multiplicar matrices cuadradas, entre ellos el más conocido es el método tradicional, que multiplica el renglón i de la primera matriz por la columna j de la segunda matriz, generando la entrada (i, j) de la matriz resultante. Si $A, B \in \mathbb{C}^{2\times 2}$ (son matrices cuadradas de 2×2), entonces el algoritmo clásico de multiplicación se puede representar con las siguientes ecuaciones:

Entrada: $M_i, M_j, M_k, oper.$
Salida: $M_k = M_i + / - M_j$
Procedimiento Suma_de_Matrices
  $\{Ren, Col\} = RenCol(M_i)$
  $Esc(Ren, M_k)$, $Esc(Col, M_k)$
  $Tam = Tam2 = Ren \times Col$
  Mientras $Tam > 0$
    $v1 = Vdatos(leido, x, Tam, M_i)$
    $v2 = Vdatos(leido, x, Tam2, M_j)$
    $i = 0$
    Mientras $leido > i$
      $Esc(v1[i]\ + / -\ v2[i], M_k)$
      $i++$
  Terminar

Tabla 2.5: Algoritmo para la suma de matrices. La entrada *oper* indica si la operación a realizar es suma o resta.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

donde:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Si tomamos en cuenta que las matrices son *grandes* y se encuentran en archivos siguiendo el formato definido anteriormente, entonces, la implementación de la multiplicación de matrices no se podría hacer con memoria dinámica, debido a los problemas de dimensión mencionados anteriormente. Por lo que para esta operación se diseñaron tres algoritmos, de los cuales dos se basan en la multiplicación clásica y uno en el algoritmo de Winograd, sabiendo que el origen y destino de las matrices son archivos.

Los algoritmos diseñados tratan de acceder al disco duro el menor número de veces ya sea para leer o para escribir. Estos algoritmos son los siguientes:

2.4.1. Multiplicación de matrices utilizando su transpuesta

El algoritmo que diseñamos se basa en la multiplicación clásica de matrices, donde se optimiza el acceso al disco duro. Como el formato diseñado para guardar los datos de la matriz asemejan a un vector renglón y si suponemos que $M_i \in \mathbb{C}^{m \times n}$ entonces

acceder a los datos como columnas implica leer un valor y saltar dentro del archivo n datos y volver a leer, haciendo esto m veces por columna, lo cual costaría mucho tiempo de cómputo.

Una manera de solucionar este problema es generando la transpuesta de la segunda matriz, utilizando el procedimiento de la tabla 2.3, para cambiar las columnas por renglones y así leer completamente los datos necesarios para obtener la entrada (i, j) de la matriz resultante sin tener que realizar saltos dentro de los archivos. El procedimiento que diseñamos se puede ver en la Tabla 2.6.

La representación elegida de las matrices evita realizar la identificación de caracteres especiales y el procedimiento de la tabla 2.3 aplicado a la segunda matriz argumento en al algoritmo de la Tabla 2.6, la lectura y escritura de los datos para las operaciones suma y multiplicación son muy similares. Por los mismos argumentos dados en la suma, la lectura y escritura de datos en la multiplicación de matrices es eficiente.

2.4.2. Multiplicación de matrices obteniendo por partes las entradas de la matriz resultante

Debido al formato en que están almacenadas las matrices, se le puede dar otra solución al problema de no acceder muchas veces al disco duro. Sea $M_{i,\alpha}$ el renglón α de la matriz M_i y $M_{i,\{\alpha,\beta\}}$ la entrada con posición (α, β) de la matriz M_i. Si $M_j \in \mathbb{C}^{n,m}$ y $M_k \in \mathbb{C}^{m,n}$ dos matrices, entonces $M_i = M_j \times M_k$ es

$$M_{i,\alpha} = \sum_{\beta=0}^{m-1} M_{j,\{\alpha,\beta\}} \times M_{k,\{\beta,0\}}, \ldots, \sum_{\beta=0}^{m-1} M_{j,\{\alpha,\beta\}} \times M_{k,\{\beta,n-1\}}, \; \alpha \in \{0, n-1\} \quad (2.1)$$

De la ecuación (2.1), si se tienen los renglones $M_{j,\alpha}, M_{k,\beta}$ solamente es posible obtener una fracción del renglón $M_{i,\alpha}$, pero si se dispone de toda la matriz M_k, entonces $M_{i,\alpha}$ es la multiplicación de $M_{j,\alpha}$ por toda la matriz M_k. Tomando como base este hecho, se desarrolló el procedimiento de la tabla 2.7. El algoritmo va generando las sumas que van a formar la entrada $M_{i,\{\alpha,\beta\}}$ y las va guardando en memoria hasta que se ha multiplicado todo el renglón $M_{j,\alpha}$ por la matriz M_k con lo que se generan las sumas necesarias para obtener los renglones completos de la matriz M_i por lo que no hay que realizar saltos dentro de los archivos.

La ventaja de este procedimiento radica en que se reduce el número de veces en que se accede a los archivos y esto se ve reflejado en el tiempo de ejecución del algoritmo. Debido a nuestra representación de las matrices en archivo este procedimiento es más eficiente debido a que no se tienen caracteres que no son datos númericos, como en el caso descrito para la suma, por lo que la lectura de los valores en la matriz M_k se realiza con la única condición de que el archivo no se haya terminado, ahorrando tiempo en la ejecución del algoritmo.

Entrada: M_i, M_j, M_k.
Salida: $M_k = M_i \cdot M_j$
Procedimiento Multiplica_Matrices_Transpuesta

$\{Ren1, Col1\} = RenCol(M_i)$
$\{Ren2, Col2\} = RenCol(M_j)$
$Esc(Ren1, M_k)$, $Esc(Col2, M_k)$
$Tam1 = Ren1 \times Col1$
$x = y = 0$
$Transpuesta_de_Matrices(M_j, M_t)$
Mientras $Tam1 > 0$
  $v1 = Vdatos(leido1, x, Tam1, M_i)$
  $Tam2 = Ren2 \times Col2$
  Mientras $Tam2 > 0$
    $v2 = Vdatos(leido2, x, Tam2, M_t)$
    $i = j = 0$
    Mientras $leido1 > i$
      $v3 = v3 + v1[i] * v2[j]$
      $i++, j++$
      Si $i \% Col1 == 0$
        $i = i - Col1$
        $Esc(v3, M_k)$
      Si $i \% Col1 == 0$ & $leido2 < j$
        $i = i + Col1, j = 0$
        $Jmp(Col2 - leido2/Ren2, M_k)$
    $Ubica(x = x + leido2/Ren2, M_k)$
  $Ubica(x = y = leido1/Col1 * Col2 + y, M_k)$
Terminar

Tabla 2.6: Algoritmo para la multiplicación de matrices usando la transpuesta de la segunda matriz para el acceso rápido de los datos.

Entrada: M_i, M_j, M_k.
Salida: $M_k = M_i \cdot M_j$
Procedimiento Multiplica_Matrices_Parciales
  $\{Ren1, Col1\} = RenCol(M_i)$
  $\{Ren2, Col2\} = RenCol(M_j)$
  $Esc(Ren1, M_k)$, $Esc(Col2, M_k)$
  $Tam1 = Ren1 \times Col1$
  Mientras $Tam1 > 0$
    $v_1 = Vdatos(leido1, x, Tam1, M_i)$
    $i = 0$
    Mientras $leido1 < i$
      $Tam2 = Ren2 \times Col2$, $j = 0$
      Mientras $Tam2 > 0$
        $v_2 = Vdatos(leido2, x, Tam2, M_j)$
        $k = 0$
        Mientras $leido2 < k$
          $v_3[j] = v_3[j] + v_1[i] \times v_2[k]$
          $k++$, $j++$
          Si $k\,\%Col2 == 0$
            $i++$, $j = 0$
      Mientras $Col2 > j$
        $Esc(v_3[j], M_k)$
  Terminar

Tabla 2.7: Algoritmo para la multiplicación de matrices. Se suman resultados parciales hasta obtener cada uno de los renglones de la matriz deseada.

2.4.3. El algoritmo de Winograd para la multiplicación de matrices.

Strassen propuso un algoritmo para multiplicar matrices, el cual divide en 4 partes a la matriz original realizando 3 multiplicaciones recursivas, siendo la multiplicación de matrices cuadradas de 2×2 el caso base y 18 sumas. Este algoritmo fue mejorado por Winograd quien propuso un algoritmo que realiza la misma cantidad de multiplicaciones pero sólo 15 sumas. El procedimiento se describe a continuación

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{2.2}$$

donde:

$$\begin{array}{lll} S_1 = A_{21} + A_{22} & P_1 = S_2 S_6 & T_1 = P_1 + P_2 \\ S_2 = S_1 - A_{11} & P_2 = A_{11} B_{11} & T_2 = T_1 + P_4 \\ S_3 = A_{11} - A_{21} & P_3 = A_{12} B_{21} & T_3 = P_5 + P_6 \\ S_4 = A_{12} - S_2 & P_4 = S_3 S_7 & \\ S_5 = B_{12} - B_{11} & P_5 = S_1 S_5 & C_{11} = P_3 + P_2 \\ S_6 = B_{22} - S_5 & P_6 = S_4 B_{22} & C_{12} = T_1 + T_3 \\ S_7 = B_{22} - B_{12} & P_6 = S_4 B_{22} & C_{21} = T_2 - P_7 \\ S_8 = S_6 - B_{21} & & C_{22} = T_2 + P_5 \end{array} \tag{2.3}$$

Como los algoritmos antes presentados, éste tiene que trabajar con matrices que siguen el formato descrito previamente, por lo que se podría realizar la división de las matrices copiando cada parte en un archivo, esto hasta llegar a las matrices base, pero no es necesario, ya que se pueden tener dos algoritmos de Winograd; uno que trabaje con todos los datos en memoria y otro que trabaje de manera similar a los algoritmos de multiplicación antes descritos, es decir, que trabaje con algunas partes en memoria y otras en archivos.

El primer algoritmo que debe ejecutarse es el que divide a la matriz en archivos hasta que el tamaño de la matriz no sobrepase a $x \sim \frac{RAM}{3 * sizeof(complejo)}$ y es aquí donde debe ejecutarse el algoritmo de Winograd pero implementado con memoria dinámica, el cual debe dar como resultado un archivo con la multiplicación de las matrices más pequeñas, regresando así de la recursión sin necesidad de dividir a la matriz original en archivos hasta la base.

Este algoritmo cuando se implementa con memoria dináica es muy eficiente debido a que su complejidad es menor que la de la multiplicación clásica, pero debido a que se trabajó con archivos, se tuvo que adaptar para que se dividiera la matriz en archivos con matrices de menor tamaño, lo que implica una mayor cantidad de accesos al disco duro, reduciendo su efectividad de manera significativa.

El diagrama de flujo que representa el procedimiento anteriormente descrito, se puede ver en la Figura 2.2.

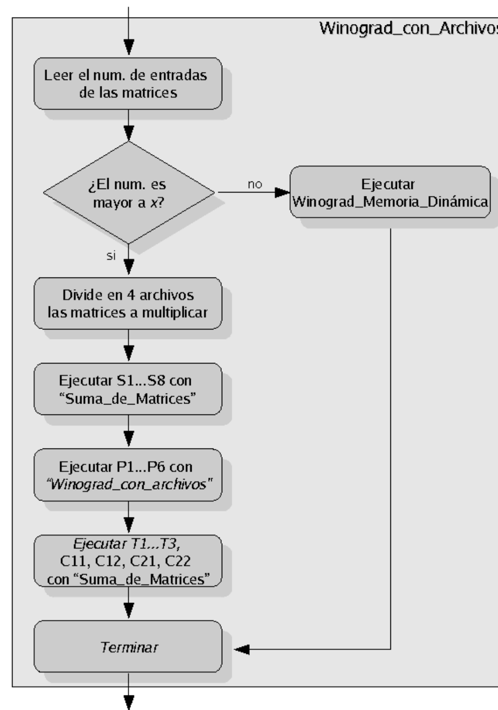

Figura 2.2: Diagrama de flujo del algoritmo de Winograd usando Archivos. El modulo *Winograd_Memoria_Dinámica* es el mismo algoritmo pero haciendo las operaciones con memoria dinámica.

2.5. Producto tensorial

Su función principal es la de formar sistemas cuánticos de dimensión mayor a sus operandos. Sea M y N dos matrices de n y m renglones respectivamente, para obtener $M_k = M \otimes N$ se multiplica cada elemento en $M_{i,j}$ por la matriz N, o más claramente, cada renglón N_i por un elemento en $M_{i,j}$. Este procedimiento se muestra en la Tabla 2.8.

Como en los algoritmos descritos anteriormente, la lectura de los datos se ve favorecida por la representación elegida, debido a que los archivos que contienen a las matrices constan exclusivamente de datos númericos por lo que se ahorra en el tiempo de ejecución debido a que no se tiene que realizar comparaciones para encontrar a los símbolos especiales.

Entrada: M_i, M_j, M_k.
Salida: $M_k = M_i \otimes M_j$
Procedimiento Producto_Tensorial_Matrices

```
{Ren1, Col1} = RenCol(M_i)
{Ren2, Col2} = RenCol(M_j)
Esc(Ren1 × Ren2, M_k), Esc(Col1 × Col2, M_k)
Tam1 = Ren1 × Col1
Mientras Tam1 > 0
    v_1 = Vdatos(leido1, x, Tam1, M_i)
    i = 0
    Mientras leido1 > i
        Tam2 = Ren2 × Col2, j = 0
        Mientras Tam2 > 0
            v_2 = Vdatos(leido2, x, Tam2, M_j)
            k = cnt = 0
            Mientras leido2 > k
                v_3[j] = v_1[i] * v_2[k]
                k++, j++
                Si k %Col2 == 0
                    i++
                    Si i ≠ 0 & i %Col1 == 0
                        j = 0, cnt = cnt + Col2
                        Mientras Col1 × Col2 > j
                            Esc(v_3[j], M_k), j++
                        j = 0
                    k = cnt
Terminar
```

Tabla 2.8: Algoritmo para realizar el producto tensorial de matrices. Se multiplica cada elemento de $M_{i,\alpha}$ por $M_{j,\beta}$ y se escribe en M_k, esto para todos los renglones de las matrices.

Capítulo 3

Sintaxis del interprete

En este capítulo hablaremos de la idea principal en la que se basa nuestra solución, para esto definiremos primero algunos conceptos necesarios para poder ver la importancia que tienen los lenguajes y las gramáticas en el área de las Ciencias de la computación. Esto nos dará las bases necesarias para construir el núcleo de nuestra solución, ya que se basa en una gramática que es capaz de generar el lenguaje de todos los algoritmos cuánticos.

3.1. Gramáticas y Lenguajes

Un lenguaje se puede definir como un grupo de palabras y métodos que son combinados y entendidos por una comunidad considerable. Esta definición es suficiente para describir la mayoría de los lenguajes *hablados o naturales* en la sociedad, pero no es suficiente para generar un lenguaje matemático. En matemáticas formales un *alfabeto* o *vocabulario* es un conjunto de símbolos, los cuales pueden ser dígitos o símbolos especiales.

Una *sentencia* es una cadena de símbolos definidos en un alfabeto de longitud finita. Esta cadena puede ser de longitud 0, a la cual llamaremos ϵ. Si $\mathcal{V}$ es un alfabeto entonces $\mathcal{V}^*$ denota al conjunto de todas las sentencias compuestas de símbolos de $\mathcal{V}$, incluyendo la sentencia ϵ al cual llamaremos un diccionario. Y sea $\mathcal{V}^\dagger$ al conjunto formado por $\mathbb{V}^* - \{\epsilon\}$, es decir, el diccionario que consta de todas las palabras no vacías. Un *lenguaje* es un grupo de sentencias sobre un alfabeto ([21]). La mayoría de los lenguajes contienen un número finito de sentencias. Una manera de representar a los lenguajes es dar un algoritmo que determine si una sentencia se encuentra o no en el lenguaje (para mayor detalles ver la referencia [21]). Otra forma de representar a los lenguajes es describiendo a la gramática que la genera.

Las gramáticas fueron formalizadas primero por la lingüística al estudiar a los lenguajes naturales. Formalmente una gramática G se define por (V_n, V_t, P, S), donde V_n son las variables, V_t los símbolos terminales, P las producciones y S el símbolo inicial. Los conjuntos V_n, V_t y P se asume que son finitos y que (V_n, V_t) no contienen elementos en común. Convencionalmente se denota por $V = V_n \bigcup V_t$. El grupo de

producciones P consiste de expresiones de la forma $\alpha \rightarrow \beta$, donde α es una cadena en $V^{\dagger}$ y β es una cadena en V^*. Finalmente, S es siempre un símbolo en V_n.

Las gramáticas se clasifican en cuatro tipos, las cuales son: las gramáticas de tipo 0 o *sin restricciones*, las gramáticas tipo 1 o *gramáticas sensibles al contexto*, las gramáticas tipo 2 o *gramáticas libres de contexto* y las gramáticas tipo 3 o *gramáticas regulares*. A esta clasificación se le conoce como la *jerarquía de Chomsky*. Las gramáticas de tipo 0 pueden verse como una serie de reglas que se tienen que seguir para generar un resultado correcto, es decir, se pueden ver como *procedimientos*. Una manera de ver este funcionamiento de manera formal es a través de las máquinas de Turing, ya que estas fueron propuestas como un modelo matemático para describir procedimientos.

Las máquinas de Turing fueron propuestas por Alan Turing y en conjunto con Church propusieron lo que se conoce como la *Tesis de Church-Turing*, en la cual se demuestra que las máquinas de Turing pueden hacer cómputo universal (ver capítulo 1). Existen otros formalismos que son equivalentes a la máquinas de Turing. Uno de ellos es el *Cálculo Lambda* propuesto por Church. Que las máquinas de Turing realicen cómputo universal tiene gran importancia ya que los lenguajes de programación se basan en la teoría de lenguajes formales que a su vez son generadas por las gramáticas formales, es decir los lenguajes de programación son sintácticamente equivalentes a las máquinas de Turing o al cálculo Lambda, por lo que a lo sumo tienen la misma capacidad de expresión que las máquinas de Turing o que el cálculo Lambda. *Como se puede ver los lenguajes, las gramáticas y las máquinas de Turing están estrechamente ligadas.*

Los lenguajes de programación se pueden clasificar según el nivel de abstracción, la forma de ejecución o el paradigma que utilicen. Según la forma de ejecución los lenguajes de programación se pueden clasificar en *Interpretados* o *Compilados*. Un interprete lee línea por línea un programa dado y lo va ejecutando, si en algún punto el programa tiene error, este es detenido e informado a su usuario. Un compilador consta de tres partes principales: analizadores *sintáctico*, *léxico* y *semántico*. El compilador revisa que el programa se encuentre bien escrito, que no tenga errores de asignaciones y por último genera un objeto que es la traducción del lenguaje de alto nivel a código máquina que puede ser ejecutado por una computadora.

Un ejemplo claro de un lenguaje de programación compilado es *Fortran* el cual empezó a desarrollarse en 1955 por la división de investigación de IBM para facilitar la programación, ya que en esa época se programaba directamente en el lenguaje máquina. Un factor que influenció el desarrollo de Fortran fue la economía de la programación, ya que en ese entonces era más caro pagar a los programadores que lo que costaba en sí las computadoras y el tiempo de uso de estas máquinas se perdía en gran parte en el trabajo de *depurado* de los programas [22]. Una de las aportaciones más importantes que hizo Fortran al área de la computación fue la introducción de los *Mnemónicos*. Los mnemónicos son palabras que representan una serie de operaciones en bajo nivel que son más fáciles de entender para los programadores que el código máquina.

La clasificación de los lenguajes según su paradigma es bastante más amplia. Dentro de los paradigmas más citados se encuentran los *lenguajes imperativos* como lo es *C*, Java y Basic. Los que se basan en la *programación lógica*, donde uno de los mayores representantes es *Prolog* que fue desarrollado por la Universidad de Marselle como una herramienta práctica para programación lógica. Prolog es un simple pero potente lenguaje de programación fundamentado en la lógica símbolica; el mecanismo básico de computación es un proceso de reconocimiento de patrones llamado *Unificación* que opera en los términos de las estructuras guardadas. Las reglas de Prolog tienen una remarcable similaridad a las gramáticas tipo 0, ya que tiene un elemento en el lado izquierdo de la proposición y algunos o ningún elemento del lado derecho. La existencia de parámetros en las reglas de Prolog refuerzan la similaridad con las gramáticas tipo 0 y con ello la posibilidad de especificar un número infinito de reglas. Una característica importante que tiene el lenguaje Prolog es el poder trabajar con datos en ambas direcciones, es decir, dada una estructura es capaz de producir una salida correcta y de una salida correcta es capaz de encontrar la entrada correspondiente [23].

Una tercera clasificación de los lenguajes de programación debido a su paradigma son los *lenguajes funcionales.* Los lenguajes de programación funcional empezaron con John McCarthy que es el inventor de *Lisp* en 1950. Peter Landin y Christopher Strachey identificaron la importancia fundamental del cálculo Lambda para el modelado de lenguajes de programación y pusieron los fundamentos de la operación semántica, a través de las máquinas abstractas y la denotación semántica. El cálculo lambda también llamado *evaluación Lazy*, tiene la propiedad de poder representar y manipular estructuras de datos infinitas ([24]). Lisp es un lenguaje con características similares a Prolog. La diferencia principal entre estos dos lenguajes es que Lisp trabaja con el cálculo Lambda de Church y Prolog con un subconjunto de la lógica clásica. En realidad, Lisp se puede ver como una especialización de Prolog [25].

Otro ejemplo importante de esta clasificación es *Haskell* que lleva este nombre en honor a Haskell B. Curry que fue desarrollado en la universidad de Yale. Así como Lisp, Haskell utiliza la teoría del cálculo Lambda para ejecutar sus programas, por lo tanto tiene todas las características de la evalución Lazy.

Nuestra solución consta de una gramática de tipo 0 que es capaz de generar al *Lenguaje de los algoritmos cuánticos.* El lenguaje de programación escogido para ejemplificar nuestro lenguaje formal fue C, es decir, escogimos un lenguaje que cae en el paradigma de los lenguajes imperativos y que utiliza un compilador para ejecutar, pero como veremos en la siguiente sección, nuestro lenguaje formal bien se pudo haber escrito en un lenguaje lógico o en algún otro que utilice un interprete para su ejecución.

3.2. Descripción de la solución

El problema que se quiere resolver es poder ejecutar cualquier algoritmo cuántico en cualquier plataforma o lenguaje que se quiera, sin tener como consecuencia una

pérdida significativa de tiempo en la traducción a otro lenguaje. Para esto, primero debemos definir qué es un algoritmo cuántico.
Se dice que un algoritmo es cuántico *si todos los procedimientos que se apliquen durante la ejecución del algoritmo se encuentran definidos en computación cuántica, es decir, que aplica únicamente compuertas cuánticas en su construcción.*

Esta definición es suficiente para *entender* las bases de un algoritmo cuántico, pero no es suficiente para definirlo de manera formal. Si analizamos la definición anterior se puede ver que tiene partes similares a la definición de un lenguaje formal y como hemos visto, un lenguaje formal es generado por las gramáticas formales, por lo tanto, diseñar una gramática que sea capaz de generar al *lenguaje de los algoritmos cuánticos* sería la manera más natural de definir formalmente a un algoritmo cuántico.

3.2.1. Diseño de la gramática

Nos dimos a la tarea de diseñar dicha gramática siguiendo la regla de que todas las operaciones que se pueden realizar deben estar definidas en computación cuántica.

Nuestra gramática consta de las siguientes categorías sintácticas:

$$\begin{aligned}\langle \text{Ct} \rangle &: \text{Constantes}\\ \langle \text{Qb} \rangle &: \text{Qubits}\\ \langle \text{Qv} \rangle &: \text{Quvectores}\\ \langle \text{Qr} \rangle &: \text{Quregistros}\\ \langle \text{Qm} \rangle &: \text{Qumatrices}\\ \langle \text{Qg} \rangle &: \text{Qucompuertas}\end{aligned}$$

dentro de los meta-símbolos Qv, Qr, Qm, Qg se le va a agregar el prefijo L, para denotar que los objetos de las correspondientes categorías son etiquetados según su longitud. Vamos a denotar por $\xi : \langle$ Categoria $\rangle$ al hecho de que el objeto sintáctico pertenece a la categoría ⟨Categoria⟩. Las producciones o reglas sintácticas que se muestran en la Tabla 3.1, son las utilizadas en nuestra gramática. La explicación de cada una de ellas es la siguiente:

(3.1) Los números complejos y los bits clásicos son constantes.

(3.2) Los bits clásicos son qubits escritos con la notación de Dirac.

(3.3) Los qubits son una superposición de los bits clásicos escritos en la notación de Dirac.

(3.4) Todo quvector de longitud n tiene dimensión 2^n.

(3.5) Todo qubit es un quvector de longitud 1.

(3.6) La multiplicación escalar de quvectores es un quvector.

(3.7) El producto tensorial de quvectores es un quvector (la longitud es aditiva).

(3.8) Todo quregistro es un quvector con norma Euclidiana 1.

(3.9) El cambio de fase de un quregistros es un quregistro.

(3.10) La j-ésima proyección de un quregistro produce el j-ésimo qubit del quregistro.

(3.11) Una qumatriz de longitud n es una matriz de $2^n \times 2^n$ entradas complejas.

(3.12) El Hermitiano conjugado de una qumatriz es una qumatriz.

(3.13) El producto tensorial de qumatrices es una qumatriz (la longitud es aditiva).

(3.14) Las qucompuertas son qumatrices unitarias.

(3.15) Las columnas de las qucompuertas son bases ortonormales.

(3.16) Las matrices de Pauli y todas las R_n son qucompuertas de paridad 1.

(3.17) CNot es una compuerta de paridad 2.

(3.18) El operador Toffoli es una qucompuerta de paridad 3.

(3.19) La composición de qucompuertas es una qucompuerta.

(3.20) El producto tensorial de qucompuertas es una qucompuerta (la longitud es aditiva).

(3.21) La aplicación de una qucompuerta sobre un quregistro produce un quregistro.

(3.22) Suprimiendo la etiqueta de la longitud, los objetos de una categoría dada son escogidos de *Qv*, *Qr*, *Qm*, Qg.

(3.23) Para un quvector y un índice la medición de la entrada dada por el índice produce un bit.

3.2.2. Fases de la solución

Ya definido de manera formal lo que es para nosotros un algoritmo cuántico, podemos proceder a diseñar una solución al problema. La idea consta de dos partes principales, la compilación y la ejecución del algoritmo. La compilación del algoritmo cuántico se va a encargar de traducir el algoritmo a un código objeto ejecutable, lo que sería generar el código objeto de una máquina con la diferencia que nuestro código máquina será el de una *computadora cuántica*, es decir, nuestra compilación simula la compilación de una computadora cuántica. Para lograr esto se siguen las siguientes etapas:

- Traducir el algoritmo cuántico que se quiera simular a expresiones válidas en nuestra gramática.

$$\xi \in \mathbb{C} \cup \mathbb{B} \implies \xi : \langle \mathrm{Ct} \rangle \tag{3.1}$$
$$\xi \in \mathbb{B} \implies |\xi\rangle : \langle \mathrm{Qb} \rangle \tag{3.2}$$
$$\alpha_0, \alpha_1 \in \mathbb{C} \ \& \ |\alpha_0|^2 + |\alpha_1|^2 = 1 \implies \alpha_0 \,|0\rangle + \alpha_1 \,|1\rangle : \langle \mathrm{Qb} \rangle \tag{3.3}$$
$$x_0, \ldots, x_{2^n-1} \in \mathbb{C} \implies \left([x_0, \ldots, x_{2^n-1}]^T, n \right) : \langle \mathrm{LQv} \rangle \tag{3.4}$$
$$\xi : \langle \mathrm{Qb} \rangle \implies (\xi, 1) : \langle \mathrm{LQv} \rangle \tag{3.5}$$
$$(\xi, n) : \langle \mathrm{LQv} \rangle \ \& \ \alpha \in \mathbb{C} \implies (\cdot \alpha \xi, n) : \langle \mathrm{LQv} \rangle \tag{3.6}$$
$$(\xi_0, n_0), (\xi_1, n_1) : \langle \mathrm{LQv} \rangle \implies (\otimes \xi_0 \xi_1, n_0 + n_1) : \langle \mathrm{LQv} \rangle \tag{3.7}$$
$$(\xi, n) : \langle \mathrm{LQv} \rangle \ \& \ \|\xi\|_2 = 1 \implies (\xi, n) : \langle \mathrm{LQr} \rangle \tag{3.8}$$
$$(\xi, n) : \langle \mathrm{LQr} \rangle \ \& \ \alpha \in \mathbb{C} \ \& \ |\alpha| = 1 \implies (\cdot \alpha \xi, n) : \langle \mathrm{LQr} \rangle \tag{3.9}$$
$$(\xi, n) : \langle \mathrm{LQr} \rangle \ \& \ 0 \leq j \leq n-1 \implies \Lambda_j \xi : \langle \mathrm{Qb} \rangle \tag{3.10}$$
$$(\forall i, j : \ 0 \leq i, j \leq 2^n - 1 \ \Rightarrow \ x_{ij} \in \mathbb{C}) \implies \left([x_{ij}]_{0 \leq i,j \leq 2^n-1}, n \right) : \langle \mathrm{LQm} \rangle \tag{3.11}$$
$$(\xi, n) : \langle \mathrm{LQm} \rangle \implies (H\xi, n) : \langle \mathrm{LQm} \rangle \tag{3.12}$$
$$(\xi_0, n_0), (\xi_1, n_1) : \langle \mathrm{LQm} \rangle \implies (\otimes \xi_0 \xi_1, n_0 + n_1) : \langle \mathrm{LQm} \rangle \tag{3.13}$$
$$(\xi, n) : \langle \mathrm{LQm} \rangle \ \& \ \xi^H \xi = \mathbf{1}_n \implies (\xi, n) : \langle \mathrm{LQg} \rangle \tag{3.14}$$
$$\{x_j\}_{0 \leq j \leq 2^n-1} \text{ una báse ortonormal de } \mathbb{C}^{2^n} \implies ([x_0 \ \cdots \ x_{2^n-1}], n) : \langle \mathrm{LQg} \rangle \tag{3.15}$$
$$\xi \in \{\sigma_0, \sigma_x, \sigma_y, \sigma_z\} \cup \{R_n\}_n \implies (\xi, 1) : \langle \mathrm{LQg} \rangle \tag{3.16}$$
$$(\mathrm{CNot}, 2) : \langle \mathrm{LQg} \rangle \tag{3.17}$$
$$(\mathrm{Toff}, 3) : \langle \mathrm{LQg} \rangle \tag{3.18}$$
$$(\xi_0, n), (\xi_1, n) : \langle \mathrm{LQg} \rangle \implies (\circ \xi_0 \xi_1, n) : \langle \mathrm{LQg} \rangle \tag{3.19}$$
$$(\xi_0, n_0), (\xi_1, n_1) : \langle \mathrm{LQg} \rangle \implies (\otimes \xi_0 \xi_1, n_0 + n_1) : \langle \mathrm{LQg} \rangle \tag{3.20}$$
$$(\xi_0, n) : \langle \mathrm{LQg} \rangle \ \& \ (\xi_1, n) : \langle \mathrm{LQr} \rangle \implies (A \xi_0 \xi_1, n) : \langle \mathrm{LQr} \rangle \tag{3.21}$$
$$(\xi, n) : \langle \mathrm{L}\eta \rangle \implies \xi : \langle \eta \rangle, \ \eta = \mathrm{Qv, Qr, Qm, Qg} \tag{3.22}$$
$$(\xi, n) : \langle \mathrm{L}Qv \rangle ; i \in \langle Ct \rangle \implies \mathbb{M}_i(\xi) \in \mathbb{B} \tag{3.23}$$

Tabla 3.1: Reglas sintácticas.

- Escribir el algoritmo resultante seleccionando algún sistema operativo, un lenguaje de programación y la plataforma en la que se desee implementar el algoritmo.
- Ejecutar dicho algoritmo, pero en vez de obtener resultados numéricos obtener una sustitución de cada una de las compuertas utilizadas en el algoritmo, lo que se podría ver como un resultado símbolico del algoritmo cuántico. Los símbolos serían las compuertas que se deben aplicar en una computadora cuántica para obtener el resultado deseado.
- Empaquetar dicho resultado símbolico en un objeto, que podría verse como el ejecutable de una computadora cuántica.

La ejecución del algoritmo cuántico se va a encargar de obtener el resultado numérico. Para hacer esto de la manera más parecida a una computadora cuántica, se realiza ejecutando todas y cada una de las instrucciones escritas en el código objeto, es decir, ejecutando cada una de las compuertas que son necesarias para obtener el resultado deseado en el algoritmo. Para lograr esto, se siguen los siguientes pasos:

- Leer cada una de las compuertas en el código objeto y traducirlas a funciones númericas en la computadora actual, este procedimiento debe estar escrito en el sistema operativo, lenguaje y plataforma en la que se desee sacar estadísticas del algoritmo cuántico.
- Ejecutar de manera numérica las instrucciones, esto se realiza aplicando cada una de las compuertas descritas en el código objeto, las cuales deben tener una representación numérica en la computadora actual.
- Dar el resultado numérico, el cual sería la respuesta de una computadora cuántica si aplicara la serie de compuertas escritas en el código objeto.

Siguiendo estos pasos, se puede pasar de un algoritmo cuántico escrito en alto nivel, a la ejecución de un algoritmo cuántico; independientemente del sistema operativo, lenguaje o plataforma seleccionada. El esquema general de la solución se puede apreciar gráficamente en la figura 3.1. El procedimiento antes descrito puede compararse con la forma en que JAVA ejecuta sus programas. Por lo que si tomamos los conceptos de JAVA, nuestro ejecutor podría llamarse *Máquina Virtual Cuántica*. La gran diferencia que existe entre nuestro esquema y JAVA radica en que nosotros no limitamos el lenguaje en el cual va a ser escrito el algoritmo cuántico, brindando la posibilidad al usuario de elegir todos los detalles de programación y más aun, no limitamos que el lenguaje en que deba ser escrito el ejecutor sea el mismo que el lenguaje en el que se escribió el algoritmo. De esta manera el usuario puede tener un algoritmo cuántico escrito en C y ejecutar el código objeto en Prolog si así lo desea, incluso puede cambiar de plataforma sin preocuparse de adaptar el código original.

Una ventaja de seguir nuestro esquema es que el único tiempo extra necesario para cambiar de sistema operativo, lenguaje de programación o plataforma, será el tiempo

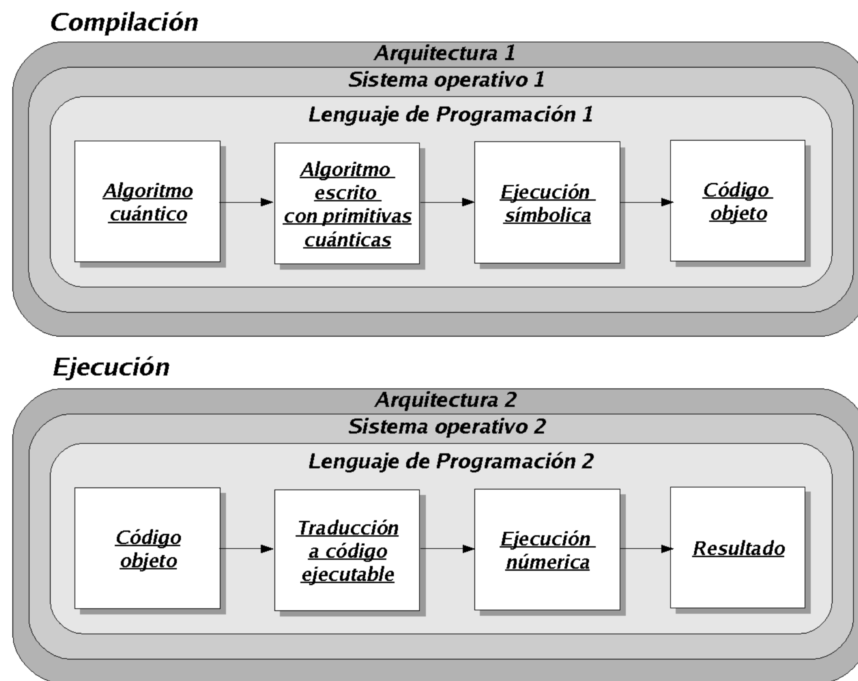

Figura 3.1: Esquema general de la solución. Se divide al problema en dos bloques, la compilación donde se obtiene el código objeto que es libre de plataforma, sistema operativo, lenguaje y la ejecución donde se obtiene el resultado númerico deseado.

necesario para implementar el ejecutor en dicha selección y esto se realizará sólo la primera vez que se quiera ejecutar nuestro código objeto, por lo que para los siguientes algoritmos, el cambio de sistema operativo, lenguaje de programación o plataforma será inmediato.

Capítulo 4

Semántica procedimental

En este capítulo hablaremos de la forma en que fue implementada nuestra solución, basándonos en la gramática y las fases definidas en el capítulo anterior tanto para el compilador como para el ejecutor.

4.1. Implementación de la solución

Para probar que nuestro esquema es factible, implementamos una serie de programas que realizan lo que hemos descrito anteriormente. Para la fase de compilación, el sistema operativo, el lenguaje y la plataforma escogida fueron Fedora, lenguaje C y una PC respectivamente. La parte de compilación la dividimos en cuatro pequeñas fases principales:

- La *definición de los símbolos* que pueden aparecer en nuestro código objeto. En esta parte nos dimos a la tarea de seleccionar las producciones que generán símbolos terminales y les asignamos símbolos válidos en el lenguaje seleccionado.
- La *expansión* de los ciclos que pudiera tener el programa. Como un programa escrito en algun lenguaje de programación imperativa probablemente lleve ciclos, nos dimos a la tarea de expander esos ciclos, como lo haría un compilador.
- La *sustitución* del código del programa por los símbolos correctos. En esta parte el programa sustituye el código escrito en el lenguaje por símbolos definidos en la fase de definición de símbolos.
- La *sangria* de los símbolos terminales. El resultado obtenido de la sustitución es muy difícil de verificar a simple vista, por lo que decidimos escribir la solución de una manera más legible para las personas, esto lo logramos identando la solución de tal manera que se asemejara a un programa.

Para probar que el código objeto obtenido en la parte de compilación puede ser ejecutado en diversos sistemas operativos, implementamos nuestro ejecutor en dos sistemas diferentes. La primera de ellas se implementó en el sistema operativo Fedora

bajo el lenguaje C y una PC como plataforma. Este ejecutor tiene las siguientes características:

- Utiliza dos pilas como estructura de datos central. La primera pila tiene tres campos: el operador, un contador de parámetros necesarios para ese operador y un contador de parámetros que ha leído para ese operador.
- La segunda pila consta de un solo parámetro el cual sirve como apuntador al archivo donde se van guardando los resultados de cada una de las operaciones hechas hasta el momento.

La segunda implementación del ejecutor se realizó en el sistema operativo Red Had bajo el lenguaje MPI y un cluster como plataforma. Este ejecutor tiene las siguientes características:

- Este ejecutor utiliza las mismas dos pilas que fueron implementadas en el primer ejecutor, la principal diferencia radica en el uso del cluster. Para aprovechar mejor el poder de cómputo de cada procesador se agregó un contador el cual tiene la finalidad de dividir de manera equitativa el trabajo de cada procesador.

4.2. Compilación

Para generar el código objeto resultante hemos implementado cuatro fases las cuales en su conjunto las hemos llamado *Compilación*. Esta compilación se encarga de traducir el código escrito en el lenguaje de programación C a un código objeto el cual puede ser ejecutado en cualquier plataforma. Cada una de las cuatro fases de la compilación es descrita a continuación.

4.2.1. Fase de definición de símbolos

Los símbolos que aparecen en el objeto resultante de nuestra compilación deben estar definidos de tal manera que no haya ambigüedad al momento de ser ejecutados. Es por este motivo que se definieron estos símbolos para las producciones de nuestra gramática que lo requieran, es decir, sólo se definieron para las producciones que generan símbolos terminales. Al pasar una gramática que es una definición abstracta de un lenguaje a una implementación en un lenguaje de programación se puede observar que las producciones requieren de parámetros, los cuales son las entradas de cada una de las producciones. La Tabla 4.1 consta de tres columnas las cuales significan lo siguiente: La primera columna es el número de la producción que genera un símbolo terminal. El número hace referencia a las producciones de la Tabla 3.1. La segunda columna consta de los símbolos utilizados para esa produción. La tercera columna nos indica qué parámetros debe recibir la producción para poder ser ejecutada.

Los parámetros de las producciones listadas en la tercera columna de la Tabla 4.1, tienen el siguiente significado durante el tiempo de ejecución de nuestro programa:

Producción	Símbolo	Parámetros
(3.1)	Emb	Ct
(3.2)	Inm	Ct
(3.7)	Vtp	Qv, Qv
(3.10)	Lmb	Ct, LQv, Qv
(3.12)	Her	$Qm \mid Qg$
(3.13, 3.20)	Mtp	$Qm, Qm \mid Qg, Qg$
(3.15)	Adc	$Ct, Ct, Ct, Qv \mid Qm \mid Qg$
(3.16)	I, X, Y, Z	
(3.16)	H, S, T, RN	
(3.17)	Cnt	
(3.18)	Tff	
(3.19)	Prd	$Qm, Qm \mid Qg, Qg$
(3.21)	App	Qg, Qv

Tabla 4.1: La primera columna representa la producción que genera símbolos terminales, la segunda el símbolo terminal utilizado, la tercera nos indica qué parámetros son necesarios para que la producción puede ser ejecutada.

- En **(3.1)**: Ct es un número.
- En **(3.2)**: Ct es un bit.
- En **(3.7)**: Qv, Qv son los quvectores a multiplicar.
- En **(3.10)**, Ct, LQv, Qv los parámetros representan el número del qubit a obtener la longitud y el quvector de donde extraer el qubit.
- En **(3.12)**: $Qm \mid Qg$ es la qumatriz o qucompuerta que se le va a aplicar el Hermitiano.
- En **(3.13)**: $Qm, Qm \mid Qg, Qg$ representan las dos qumatrices o qucompuertas a las cuales se les va a aplicar el producto tensorial.
- En **(3.15)**: $Ct, Ct, Ct, Qv \mid Qm \mid Qg$ los dos primeros parámetros representan la cantidad de renglones y columnas a copiar, el tercer parámetro es un número tal que si es 0 es un archivo nuevo, si es 1 significa anexar datos, el último parámetro dice que los datos a anexar pueden ser tomados de quvectores, qumatrices o qucompuertas.
- En **(3.16)**: los símbolos I, X, Y, Z, H, S, T, RN representan a las matrices identidad, Pauli-X, Pauli-Y, Pauli-Z, Hadamard, Phase, $\frac{\pi}{8}$ y n-ésima raíz unitaria de longitud 1 respectivamente.
- En **(3.17)**: el símbolo Cnt representa la matriz Cnot.

- En **(3.18)**: el símbolo Tff representa la matriz de Toffoli.
- En **(3.19)**: $Qm, Qm \mid Qg, Qg$ representan a las dos qumatrices o qucompuertas que se van a componer.
- En **(3.21)**: Qg, Qv representan a una qucompuerta que se va a aplicar sobre un quvector.

Los símbolos mostrados son los únicos que aparecen en nuestro código objeto así como los parámetros definidos.

4.2.2. Fase de expansión

Para compilar de manera correcta el código de nuestro algoritmo cuántico seguimos ciertas reglas, de entre las más importantes se encuentran las siguientes:

- El algoritmo cuántico que vaya a ser escrito en algún lenguaje de programación debe seguir estrictamente a la gramática definida anteriormente.
- Al momento de implementar el algoritmo cuántico podemos utilizar cualquier cantidad de variables temporales, siempre que éstas no se escriban dentro del código objeto resultante.
- Siempre que se vaya a ejecutar una producción que de como resultado un símbolo terminal, éste debe ser escrito en el código objeto.

En la implementación de nuestra compilación decidimos escribir el algoritmo cuántico de manera tradicional, es decir, generando resultados numéricos con el motivo de asegurarnos que el algoritmo funcionaba correctamente, pero teniendo el cuidado de que las producciones que generan símbolos terminales se encontraran en una biblioteca de funciones aislada del código del algoritmo cuántico que deseemos estudiar.

El motivo de escribir las operaciones que generan símbolos terminales de manera aislada, fue para la fácil sustitución de esta biblioteca por otra que en vez de generar resultados numéricos, nos diera resultados símbolicos así no tuvimos que cambiar ni una linea de código del programa original.

Para obtener el código objeto con nuestra salida simbólica cambiamos la biblioteca que tiene los procedimientos con salidas numéricas por la que realiza la salida simbólica. El programa resultante del algoritmo cuántico que deseemos ejecutar probablemente contendrá ciclos por lo que tenemos que hacer un *desenrrollo* de éstos, llevando una lista de los archivos que se van utilizando para guardar las qumatrices o qucompuertas que se van generando. Hablamos de archivos debido a que nuestras primitivas cuánticas, las cuales definimos en el capitulo 2 de esta tesis, utilizan archivos para guardar los datos que van generando.

El archivo de salida tendrá en su interior renglones con el formato mostrado en la ecuación (4.1). El renglón empezara con el símbolo utilizado para representar la producción que es aplicada, seguido de los archivos donde se encuentran sus parámetros

y el último fragmento del renglón es el nombre del archivo donde se va a guardar el resultado de aplicar la producción utilizando los parámetros dados.

$$< \textit{símbolo} > < \textit{archivo } 1 >< \textit{archivo } 2 > \ldots < \textit{archivo destino} > \tag{4.1}$$

así si se aplicara la producción (3.13) que es la aplicación del producto tensorial de qucompuertas, donde las qucompuertas estuvieran guardadas en los archivos `File1` y `File2` y se quisiera guardar el resultado en el archivo `File3` entonces el resultado de aplicar nuestra expansión sobre el programa sería:

$$\texttt{Mtp File1 File2 File3} \tag{4.2}$$

sucede lo mismo cuando la operación es una asignación, un ejemplo de esto sería la aplicación de la producción (3.13), la cual podría ser la asignación de la compuerta Hadamard al archivo `File4`. Aplicar nuestra expansión nos dejaría la siguiente salida

$$\texttt{H File4} \tag{4.3}$$

4.2.3. Fase de sustitución

Una vez que se tienen todo el programa expandido y con las producciones que deben ejecutarse en orden junto con sus parámetros se debe proceder a sustituir los nombres de los archivos por las compuertas que tienen en su interior.

Para sustituir el nombre de los archivos por cada una de las compuertas que representan utilizamos la ayuda de una lista. Cada nodo de la lista tiene dos campos, el nombre del nodo y una cadena de caracteres. El nombre del nodo es el mismo que el nombre del archivo y la cadena de caracteres va guardando la serie de operaciones que se hicierón antes de asignarsele los datos. Se puede apreciar de mejor manera el procedimiendo del siguiente ejemplo:

1. Se quiere aplicar el producto tensorial de dos compuertas Hadamar

```
  H     File0
  H     File1
Mtp     File0 File1 File2
```

2. La lista evolucionaría de la siguiente manera:

```
Lista:   {{Nom: File0, Cad: H}}
Lista:   {{Nom: File0, Cad: H},{Nom: File1, Cad: H}}
Lista:   {{Nom: File0, Cad: H},{Nom: File1, Cad: H},
          {Nom: File2, Cad: Mtp H H}}
```

   de este ejemplo se puede apreciar que el campo `Nom` tiene el mismo valor que el nombre del archivo al que se le va asignar la operación requerida y el campo `Cad` guarda la operación realizada junto con todos sus parámetros.

3. Extendamos el ejemplo, suponga que ahora se quiere aplicar el producto tensorial del resultado anterior con la compuerta Identidad

```
  I     File0
Mtp     File2 File0 File1
```

4. La lista evolucionaría de la siguiente manera:

```
Lista:   {{Nom: File0, Cad: I},{Nom: File1, Cad: H},
          {Nom: File2, Cad: Mtp H H}
Lista:   {{Nom: File0, Cad: I},{Nom: File1, Cad: Mtp Mtp H H I},
          {Nom: File2, Cad: Mtp H H}}
```

   obsérvese que ahora el nodo con el `Nom: File1`, tiene el resultado de toda la serie de operaciones que fueron aplicadas en su campo `Cad`. Una característica de la cadena guardada en el campo `Cad` es que tiene las operaciones y los parámetros acomodados en posfijo, de esta manera se elimina cualquier ambigüedad que pudiera darse a la hora de aplicar las operaciones sobre las compuertas. El resultado final de todas las operaciones que hayan sido ejecutadas lo tendrá el campo `Cad` del último nodo accesado.

4.2.4. Fase de identación

Al llegar a este punto el código objeto ya tiene todas las operaciones y parámetros que necesita para ser ejecutado, pero al estar el resultado en un solo renglón es muy complicado leer y verificar por una persona que sea correcto, es por eso que decidimos identar la solución.

Este problema se puede comparar con tratar de leer el código de un programa que se encuentra escrito todo en un renglón, lógicamente sería más fácil si éste estuviera identado. Siguiendo la misma idea, decidimos identar la solución de manera similar a que si fuera el código de un programa.

Para hacer esto seguimos las siguientes reglas:

- Se va a suponer que el archivo de salida es un plano cartesiano.
- Cada parámetro de una operación se debe encontrar en un renglón. Así si la operación utiliza dos parametros, tendrá cada uno de ellos en un renglon diferente y consecutivo.
- Todos los parámetros de una operación se debe encontrar en la misma columna.
- Es valido anidar operaciones.

Al final de la identación tenemos el código objeto con la serie de compuertas necesarias para obtener el resultado deseado, con la característica que es libre del sistema en que vaya hacer ejecutado.

4.2.5. Pequeño ejemplo de la compilación

La forma en que va evolucionando un algoritmo cuántico se puede apreciar en el siguiente ejemplo, donde se genera la qucompuerta SWAP, suponiendo que ya fue escrito el código del algoritmo en C:

1. Algoritmo cuántico

$$CNOT \cdot H \otimes H \cdot CNOT \cdot H \otimes H \cdot CNOT$$

2. Fase de expansión:

```
Cnt File0
H File1
H File2
Cnt File3
Mtp File1 File2 File4
Prd File4 File3 File1
Prd File1 File4 File5
Prd File0 File5 File6
Prd File6 File0 Descripcion.txt
```

3. Fase de sustitución

```
Descripcion.txt = Prd Prd Cnt Prd Prd Mtp H H Cnt Mtp H H Cnt
```

4. Fase de sangria

```
Descripcion.txt =
     Prd Prd Cnt
             Prd Prd Mtp H
                         H
                     Cnt
                 Mtp H
                     H
         Cnt
```

4.3. Ejecutor

Una vez se ha ejecutado la compilación se obtiene el código objeto que puede ser ejecutado en diversas plataformas. Para probar esta afirmación hemos desarrollado nuestro ejecutor en dos plataformas diferentes. La primera arquitectura que elegimos para desarrollar nuestro ejecutor tiene las siguientes características:

- Sistema operativo Fedora.
- Lenguaje de programación C.
- Computadora de escritorio como plataforma.

Para la segunda implementación de nuestro ejecutor tratamos de variar en lo mayor posible las características de la arquitectura. Esta segunda arquitectura consta de las siguientes características:

- Sistema operativo Red Hat.
- Lenguaje de programación MPI.
- Cluster como plataforma.

El desarrollo de cada uno de nuestros ejecutores se detalla a continuación.

4.3.1. Ejecutor en Fedora en el lenguaje C sobre una PC

El ejecutor implementado recibe como entrada el objeto resultante de la compilación, el formato de nuestro código objeto no necesitar estar identado pero si lo está no afecta en nada a la ejecución ya que los espacios en blanco son ignorados. La implementación realizada en esta plataforma consta principalmente de dos pilas como estructura de datos para poder realizar la evaluación numérica de nuestro objeto. El funcionamiento de los campos de cada una de nuestras pilas se explica a continuación:

- La primera pila consta de tres campos. El campo *Oper* se encarga de guardar qué operación cuántica es mandada a realizar. El campo *NumPar* es una variable que guarda la cantidad de parámetros necesarios para poder ejecutar la operación guardada en Oper. El campo *NumParLeidos* se inicializa en 0 y se incrementa cada vez que se lee un parámetro de la operación guardada en Oper.
- La segunda pila consta de un solo campo llamado *NameFile* donde se van guardando los nombres de los archivos donde se encuentran los resultados de las operaciones realizadas.

Definimos como un parámetro a las qucompuertas básicas, es decir, a las compuertas *X, Y, Z, H, Cnot* y *Toffoli* y a un operador como un símbolo que no es un parámetro. Para ejecutar leemos de izquierda a derecha cada una de las palabras de las que consta nuestro código objeto.

Si la palabra leída es un operador entonces se inserta en la primera pila la palabra leída, se inicializa el contador de parámetros leídos y se actualiza el valor de parámetros necesarios según el operando leído. Una vez hecho esto se lee la siguiente palabra.

Si la palabra leída es un parámetro se realiza la asignación numérica de la qucompuerta a un archivo y se inserta en la segunda pila el nombre del archivo que contiene dicha asignación. Una vez guardado el operando se entra a un ciclo en el cual se realiza los siguiente pasos:

1. Se incrementa el contador de parámetros leídos del operador en el tope de la primera pila.
2. Se realiza una comparación entre los campos `NumPar` y `NumParLeidos` de la primera pila.
3. Si son iguales entonces se aplica la operación guardada en el campo `Oper` del elemento en el tope de la primera pila usando como parámetros los primeros `NumPar` nombres de los archivos guardados en la segunda pila.
4. Se realiza un `Pop` sobre la primera pila y `NumPar` sobre la segunda.
5. El resultado de la aplicación de la operación se guarda en un archivo, cuyo nombre es insertado en la segunda pila.
6. Se regresa al inicio de este ciclo.

El procedimiento descrito se realiza hasta que el archivo donde se encuentra el código objeto queda vacío. El resultado final de la aplicación de todas las operaciones guardadas en el código objeto se encuentra en el archivo con el nombre guardado en el tope de la segunda pila. El procedimiento descrito anteriormente se sintetiza en el algoritmo mostrato en la Tabla 4.2.

4.3.2. Ejemplo del ejecutor en Fedora en el lenguaje C sobre una PC

La forma en que va evolucionando la ejecución de nuestro código objeto se puede apreciar en el siguiente ejemplo, donde se resuelve la aplicación del producto tensorial de dos qucompuertas Hadamard seguido del producto tensorial de la compuerta identidad:

- El código objeto resultante de la aplicación de la compilación sobre el código escrito en C cuyo operación representa el producto tensorial de dos qucompuertas Hadamard seguido del producto tensorial de la qucompuerta identidad es el siguiente:

  `Mtp Mtp H H I`

- Se lee el símbolo `Mtp` y las pilas evolucionan de la siguiente manera:

```
Pila1:   {{Oper: Mtp, NumPar: 2, NumParLeido: 0}}
Pila2:   { }
```

```
Entrada: Obj = resultado de la compilación
Salida:    Res = nombre del archivo con
                 la solución númerica de Obj
Procedimiento.Ejecutor
   Mientras No feof(Obj)
      Sim = Lee símbolo
      Si Sim es Operador
         Push(Sim) en P1
         Act(P1.NumPar)
         P1.NumParLeidos = 0
      Si Sim es Parámetro
         Push(File.Sim) en P2
      Mientras P1.NumParLeidos = P1.NumPar
         Res = Ejec(P1.Oper)
         Mientras P1.NumPar > 0
            Pop(P2)
            P1.NumPar−−
         Pop(P1)
         Push(Res) en P2
   Terminar
```

Tabla 4.2: Algoritmo que ejecuta el código objeto resultante de la compilación. `P1` y `P2` representan la primera y segunda pila respectivamente. La función `Push` inserta un elemento en la pila `P1` o `P2` según el caso. Si la pila a insertar es `P2` entonces también actualiza el campo `NumParLeidos` del tope de `P1`. La función `Ejec` ejecuta la operación guardada en el tope de P1 usando los primeros `P1.NumPar` elementos de la pila `P2`.

- Se lee el segundo símbolo `Mtp` y las pilas evolucionan de la siguiente manera:

```
Pila1:    {{Oper: Mtp, NumPar: 2, NumParLeido: 0},
           {Oper: Mtp, NumPar: 2, NumParLeido: 0}}
Pila2:    { }
```

- Se lee el símbolo `H` y las pilas evolucionan de la siguiente manera:

```
Pila1:    {{Oper: Mtp, NumPar: 2, NumParLeido: 0},
           {Oper: Mtp, NumPar: 2, NumParLeido: 1}}
Pila2:    {{NameFile: File0}}
```

  en el archivo `File0` se encuentra la matriz Hadamard y el contador `NumParLeido` se incrementa.

- Se lee el segundo símbolo `H` y las pilas evolucionan de la siguiente manera:

```
Pila1:    {{Oper: Mtp, NumPar: 2, NumParLeido: 0},
           {Oper: Mtp, NumPar: 2, NumParLeido: 2}}
Pila2:    {{NameFile: File0},
           {NameFile: File1}}
```

  en el archivo `File1` se encuentra la matriz Hadamard y el contador `NumParLeido` se incrementa.

- Como la comparación se cumple se accede al ciclo descrito anteriormente, por lo que las pilas evolucionan de la siguiente manera:

```
Pila1:    {{Oper: Mtp, NumPar: 2, NumParLeido: 1}}
Pila2:    {{NameFile: File2}}
```

  en el archivo con el nombre `File2` se encuentra la evaluación del producto tensorial de las qucompuertas guardadas en los archivos con nombres `File0` y `File1`. El contador `NumParLeido` del elemento en el tope de la pila se incrementa.

- Se lee el símbolo `I` y las pilas evolucionan de la siguiente manera:

```
Pila1:    {{Oper: Mtp, NumPar: 2, NumParLeido: 2}},
Pila2:    {{NameFile: File2},
           {NameFile: File3}}
```

  en el archivo con el nombre `File3` se encuentra la qucompuerta identidad. El contador `NumParLeido` se incrementa.

- Como la comparación se cumple se accede al ciclo descrito anteriormente, por lo que las pilas evolucionan de la siguiente manera:

  `Pila1:` $\{\}$
  `Pila2:` $\{\{$`NameFile: File4`$\}\}$

  en el archivo con el nombre `File4` se encuentra el resultado de aplicar el producto tensorial de dos qucompuertas Hadamard seguido del producto tensorial de la qucompuerta identidad.

4.3.3. Ejecutor en Red Hat en el lenguaje MPI sobre un cluster

Un cluster se puede ver como un conjunto de computadoras que trabajan de manera simultánea para generar un resultado correcto. Por la misma naturaleza del cluster existen diversas formas de usar cada uno de los procesadores. Dos de los esquemas más utilizados actualmente son los siguientes:

- Generar un solo programa el cual va a ser ejecutado en cada uno de los procesadores con la diferencia de que los datos de entrada para cada uno de ellos es diferente. Así se logra generar la ejecución de un programa n veces, suponiendo que se cuenta con n procesadores, en un solo tiempo de computo.

- Dividir un programa en n partes, donde n es la cantidad de procesadores que se tienen, y cada una de estas partes es ejecutada por un solo procesador, así el tiempo de computo de una corrida de este programa es reducido en una fracción aproximada a $\frac{1}{n}$. Se dice que es aproximada porque hay que tomar en cuenta el tiempo que se toman los procesadores en compartir información entre ellos.

Nosotros hemos decidido por la primera opción, ya que las operaciones que vamos a ejecutar son relativamente pocas y siempre las mismas además de que no queremos que haya tráfico en la red de comunicaciones de los procesadores ya que esto aumentaría de gran manera nuestro tiempo de ejecución. La variante que hemos puesto al primer esquema es que una vez que los procesadores han terminado de ejecutar unimos los datos para generar una sola salida, es decir, como si se tratara del segundo esquema.

La forma en que hemos dividido lo datos de entrada lo relacionamos con la producción (3.15). Esta producción genera el símbolo terminal `Adc` la cual se puede ver como un reacomodo de quvectores para generar una qucompeurta. Si nuestro código objeto contiene muchos de estos símbolos entonces conviene ejecutarlo en un cluster ya que se trata de un computo *grande*, si no lo tuviera no habría necesidad de dividir las operaciones ya que serían pocas y en vez de disminuir nuestro tiempo de ejecución este aumentaría.

La forma en que hemos dividido las operaciones que deben realizar cada operador sigue la siguiente formula:

$$\begin{aligned} x &\in [\![1, n]\!] \\ dist &= \frac{t}{n} \\ x_{i,inicial} &= (i-1) * dist + 1 \\ x_{i,final} &= \begin{cases} t & \text{si } i == n \\ i * dist & \text{en otro caso} \end{cases} \end{aligned} \tag{4.4}$$

donde x_i es el procesador i, n es el número de procesadores y t es el número total de `Adc's` que aparecen en nuestro código objeto. La forma en que hemos dividido la carga de trabajo de cada uno de los procesadores se puede apreciar graficamente en la Figura 4.1.

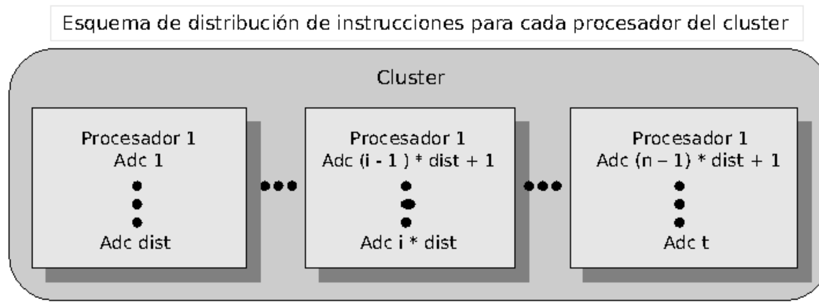

Figura 4.1: Esquema a seguir para distribuir de manera equitativa las instrucciones escritas en el código objeto a ejecutar. En la imagen $i \in \mathbb{N}$, t es el número total de `Adc's`, n es el número de procesadores y $dist = \frac{t}{n}$.

Una vez definido la forma en que dividimos la carga de trabajo de cada procesador decidimos que la forma de ejecución de cada procesador sea igual a como si fuera un PC, es decir, seguimos el mismo algoritmo que diseñamos para el ejecutor que corre en Fedora, en el lenguaje C bajo una PC con una pequeña variante. La diferencia fundamental entre este ejecutor y el anterior radica en la producción (3.15), esta producción la va a realizar el procesador padre, que para nosotros va a ser el primer procesador que recibe el código objeto que queremos ejecutar.

4.3.4. Ejemplo del ejecutor en Red Hat en el lenguaje MPI sobre un cluster

La forma en que va evolucionando la ejecución de nuestro código objeto en el cluster se puede apreciar en el siguiente ejemplo, donde se resuelve la aplicación del producto tensorial de dos qucompuertas Hadamard seguido del producto tensorial de la compuerta identidad en cada uno de nuestros procesadores:

- El código objeto resultante de la aplicación de la compilación sobre el código escrito en C cuyo operación representa el producto tensorial de dos qucompuertas

Hadamard seguido del producto tensorial de la qucompuerta identidad repetido cuatro veces y de los cuales se quiere que los dos primeros quvectores de cada una de las qucompeurtas resultantes formen una sola qucompuerta por lo que el objeto tendria en su interior lo siguiente:

```
Adc 8 2 0 Mtp Mtp H H I
Adc 0 2 1 Mtp Mtp H H I
Adc 0 2 1 Mtp Mtp H H I
Adc 0 2 1 Mtp Mtp H H I
```

aqui el valor 8 del primer renglón significa que se va a crear una qumatriz de dimensión 2^3, el valor 2 indica que se van a tomar 2 quvectores de la qumatriz a su derecha, el valor 0 indica que es un archivo nuevo, es decir, una qumatriz nueva y los siguientes parámetros forman la qumatriz de la que se van a obtener los quvectores. Para los siguientes renglones significan lo mismo con la excepción que el valor cero en el tercer parámetro significa que van a anexar sus datos a una qumatriz existente la cual esta a la izquierda de su valor generado.

- Supongamos que se tienen exactamente cuatro procesadores, entonces la división de las operaciones en cada uno de los procesadores quedaría de la siguiente forma:

$$\begin{aligned}
t = 4, n = 4 \quad &\Rightarrow \quad dist = \frac{t}{n} = \frac{4}{4} = 1 \\
\text{Formula} \quad &: \quad [(i-1) * dist + 1, i * dist] \\
\text{Procesador 1} \quad &: \quad [(1-1) * 1 + 1 = 1, 1 * 1 = 1] = [1, 1] \\
\text{Procesador 2} \quad &: \quad [(2-1) * 1 + 1 = 2, 2 * 1 = 2] = [2, 2] \\
\text{Procesador 3} \quad &: \quad [(3-1) * 1 + 1 = 3, 3 * 1 = 3] = [3, 3] \\
\text{Procesador 4} \quad &: \quad [(4-1) * 1 + 1 = 4, 4 * 1 = 4] = [4, 4]
\end{aligned}$$

  es decir, a cada procesador les va a tocar realizar los Adc entre los intervalos mostrados, en este caso solo les va a tocar realizar uno solo de los Adc.

- Supongamos que se ha ejecutado el algoritmo *Ejecutor* de manera correcta en cada uno de los procesadores, entonces se tienen cuatro qumatrices una en cada procesador. La primera vez que aparece el `Adc` el tercer parámetro vale 0 y eso significa que él es el encargado de buscar la cantidad de datos marcados en su primer y segudo parámetro. Entonces el primer procesador buscaría en su matriz los dos primeros quevectores y los agregaría a un archivo nuevo. El segundo quvector mandaría sus primeros dos quevectores al procesador uno para anexarlos al archivo creado por él y asi para cada uno de los procesadores. Al final el procesador uno tendría en su interior una qumatriz que es igual a la concatenación de los primeros dos quevectores obtenidos por cada procesador.

Capítulo 5

Casos de prueba

En los capítulos anteriores se dieron las bases en las que se fundamentan todos los algoritmos cuánticos así como la definición de la gramática con la que se trabajó en esta tesis. Pero la teoría, necesita ser ilustrada para su mejor entendimiento es por eso que se probó la teoría descrita anteriormente con el algoritmo más representativo del estado del arte, el algoritmo desarrollado por Peter Shor el cual factoriza un número entero en dos menores en caso de poderse o bien indica que es primo.

Para poder entender de manera correcta el algoritmo de factorización de números enteros, se van a mencionar las bases de las que consta dicho algoritmo, empezando por definir la manera de obtener el reverso de un producto tensorial seguido de la transformada discreta de Fourier cuántica y terminando con el algoritmo de Shor.

Estos algoritmos trabajan con el supuesto de que se tiene una computadora cuántica la cual no existe, como se mencionó anteriormente, por lo que nos dimos a la tarea de simularla. Es por eso que se definió una gramática la cual seguiremos para dicha simulación.

El primero de los algoritmos que se ha estudiado y probado es el reverso de productos tensoriales. Este algoritmo lo hemos desarrollado a partir de una compuerta capaz de invertir el orden de las multiplicaciones de dos qu-vectores, dicha compuerta es la compuerta SWAP. Debido a la forma en que están guardados los datos en los archivos, este algoritmo se puede ver como la inversión de vectores pero usando únicamente compuertas cuánticas.

La transformada discreta de Fourier cuántica, se puede decir que es el *corazón* del algoritmo de Factorización de números enteros. La importancia de este algoritmo radica en que es capaz de distribuir y agrupar probabilidades y debido a esto, usando la teoría desarrollada por Peter Shor, reduce de manera considerable la cantidad de intentos necesarios para encontrar el orden de un número, el cual, pensando en tiempo de computo es el pricipal problema a resolver en el algoritmo publicado por Peter Shor.

Por ultimo, el algoritmo estudiado a fondo en esta tesis es el de factorización de enteros publicado en 1994 por Peter Shor, el cual diseñó un algoritmo que es capaz de factorizar un número entero en tiempo polinomial y como sabemos, el mejor algoritmo clásico resuelve este problema en tiempo exponencial computacionalmente

∘	I	C	D	CD	DC	CDC
I	I	C	D	CD	DC	CDC
C	C	I	CD	D	CDC	DC
D	D	DC	I	CDC	C	CD
CD	CD	CDC	C	DC	I	D
DC	DC	D	CDC	I	CD	C
CDC	CDC	CD	DC	C	D	I

Tabla 5.1: Subgrupo formado por las compuertas C, D y la operación de composición.

hablando. Debido a las implicaciones que tendría en la criptografía, ya que muchos de los algoritmos de esta área de la Computación basan su seguridad en lo difícil que es encontrar la factorización de los números enteros, se considera a este algoritmo como el más representativo del estado del arte en Computación Cuántica.

5.1. Reverso de qubits

El resultado de aplicar el producto tensorial de dos qu-vectores es un qu-vector, el cual se representa en nuestra notación como un vector y éste se guarda en un archivo. Como se mencionó anteriormente, el producto tensorial no es conmutativo, por lo que cambiar de orden los factores no generara el mismo resultado.

Este intercambio, que llamaremos *reverso*, se podría hacer de manera muy fácil si se pudiera utilizar las operaciones clásicas y se tratara de vectores comunes, pero como esta operación debe ser soportada por una computadora cuántica, se debe basar absolutamente de operaciones validas en Computación Cuántica. Es por eso que hemos desarrollado un algoritmo que es capaz de realizar esta función usando solamente compuertas cuánticas. El procedimiento seguido para desarrollar dicho algoritmo se detalla acontinuación.

5.1.1. Desarrollo del algoritmo *reverso*

Sea $C = CNOT$ y $D = CNOTI$. En el espacio de transformaciones unitarias, C y D generan un subgrupo con la operación de composición. La tabla de multiplicar puede verse en la Tabla 5.1. Se puede decir que este grupo queda *presentado* por su unidad I, dos generadores C, D y la relación $CDC = DCD$. De hecho este grupo es isomorfo al grupo de permutaciones de 3 elementos, S_3. En efecto, si $\rho = (1, 2)$ es la *reflexión* y $\phi = (1, 2, 3)$ es el ciclo de orden 3, entonces se puede identificar a $C \leftrightarrow \rho$ y a $D \leftrightarrow \rho \circ \phi$.

Aplicando la compuerta formada por $R_2 = CDC$ a las bases de estado $|a, b\rangle$ se tienen las ecuaciones (5.1), donde se puede ver que la función que realiza esta compuerta es tal que $R_2 (\mathbf{e}_i \otimes \mathbf{e}_j) = \mathbf{e}_j \otimes \mathbf{e}_i$, es decir su función es la de intercambiar

las bases de estado. Esta compuerta es llamada comunmente *SWAP*.

$$\begin{aligned} |a,b\rangle &\longrightarrow |a, a\oplus b\rangle \\ &\longrightarrow |a\oplus(a\oplus b), a\oplus b\rangle = |b, a\oplus b\rangle \\ &\longrightarrow |b,(a\oplus b)\oplus b,\rangle = |b,a\rangle \end{aligned} \tag{5.1}$$

El cicuito que representa a la compuerta $SWAP$ se puede observar en la Figura 5.1, hay que aclarar que el circuito representa la compuerta a aplicar con el paso del tiempo, no necesariamente corresponde a una implementación física.

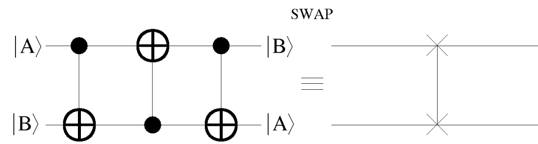

Figura 5.1: Circuito que intercambia dos qubits, y su equivalente símbolo esquemático más comunmente usado.

Ahora, se desea que $R_3 \in \mathbb{H}_3$, sea una compuerta tal que $R_3(\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k) = \mathbf{e}_k \otimes \mathbf{e}_j \otimes \mathbf{e}_i$, es decir que aplique la operación $SWAP$ pero en $\mathbb{H}_3$. En ecuaciones se tiene:

$$\begin{aligned} &|000\rangle \to |000\rangle; |001\rangle \to |001\rangle; |010\rangle \to |010\rangle; |011\rangle \to |011\rangle; \\ &|100\rangle \to |101\rangle; |101\rangle \to |100\rangle; |110\rangle \to |111\rangle; |111\rangle \to |110\rangle; \end{aligned} \tag{5.2}$$

El circuito que representa el funcionamiento de R_3 se puede ver en la Figura 5.2. Como se ve, es un circuito muy similar al de la compuerta $SWAP$, con la diferencia de que entre las compuertas $CNOT$ y $CNOTI$ se encuentra un qubit el cual no sufre ninguna alteración.

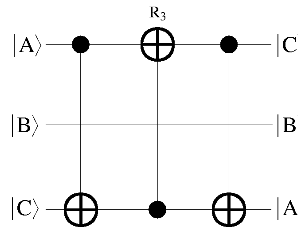

Figura 5.2: Circuito que intercambia tres qubits.

Para generar una compuerta de 3 qubits que aplique el $CNOT$, pero con el qubit de control 1 y el qubit objetivo 3, se puede seguir a $CNOT_{1,3}$ donde

$$CNOT_{1,3} = (SWAP \otimes I) * (I \otimes CNOT) * (SWAP \otimes I) \tag{5.3}$$

realizar el producto tensorial de la identidad por alguna compuerta nos pasa del campo $\mathbb{H}_i$ al $\mathbb{H}_{i+1}$, aplicar $SWAP \otimes I$ por la izquierta de alguna matriz $M \in \mathbb{H}_3$

intercambia los renglones 2 y 3 por 4 y 5 respectivamente, y aplicarla por la derecha realiza el intercambio en las columnas.

La representación matricial de $CNOT_{1,3}$ es mostrada en la ecuacion (5.4), se puede ver que esta compuerta cumple con ser unitaria.

$$CNOT_{1,3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.4}$$

Si se procede de manera similar para la compuerta $CNOTI$, se tendrá

$$CNOTI_{1,3} = (SWAP \otimes I) * (I \otimes CNOTI) * (SWAP \otimes I) \tag{5.5}$$

de la representación matricial de $CNOTI_{1,3}$, mostrada en (5.6), se puede ver que cumple con ser unitaria.

$$CNOTI_{1,3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.6}$$

Por lo tanto R_3 queda definido de la siguiente manera

$$R_3 = CNOT_{1,3} * CNOTI_{1,3} * CNOT_{1,3} \tag{5.7}$$

De R_2 y R_3 se puede ver que se intercambia el primero y el ultimo elemento de los productos tensoriales. Para el caso R_3, no se realiza nada al segundo qubit. Si tomamos a estos elementos como la base de una recursión, se puede generalizar a R_n.

Sea $R_n : \mathbb{H}_n \rightarrow \mathbb{H}_n$ tal que $R_n(\mathbf{e}_1 \otimes \ldots \otimes \mathbf{e}_n) = \mathbf{e}_n \otimes \ldots \otimes \mathbf{e}_1$, donde

$$C_{1,n} = \begin{cases} C & \text{si } n = 2 \\ (S \otimes I_1 \otimes \cdots \otimes I_{n-1}) * (I \otimes C_{1,n-1}) * (S \otimes I_1 \otimes \cdots \otimes I_{n-1}) & \text{en otro caso} \end{cases} \tag{5.8}$$

$$D_{1,n} = \begin{cases} D & \text{si } n = 2 \\ (S \otimes I_1 \otimes \cdots \otimes I_{n-1}) * (I \otimes D_{1,n-1}) * (S \otimes I_1 \otimes \cdots \otimes I_{n-1}) & \text{en otro caso} \end{cases}$$

$$R_n = \begin{cases} C * D * C & \text{si } n = 2 \\ C_{1,3} * D_{1,3} * C_{1,3} & \text{si } n = 3 \\ C_{1,n} * D_{1,n} * C_{1,n} * (I \otimes R_{n-2} \otimes I) & \text{en otro caso} \end{cases}$$

Entrada: n = cantidad de qubits.
Salida: R_n = La matriz que aplicada a n qubits, genera su reverso
Procedimiento.ReversoN
(1) Bandera = mod(n, 2)
(2) Si Bandera
$R_n = Identidad$
(3) Si !Bandera
$R_n = SWAP$
(4) Hacer desde i = 0 hasta n-2
(a) SWAP = $SWAP \otimes Identidad$
(b) CNOT = $Identidad \otimes CNOT$
(c) Aux = $SWAP * CNOT$
(d) CNOT = $Aux * SWAP$
(e) CNOTI = $Identidad \otimes CNOTI$
(f) Aux = $SWAP * CNOTI$
(g) CNOTI = $Aux * SWAP$
(h) Si Bandera
(i) Aux = $CNOT * CNOTI$
(ii) Aux = $Aux * CNOT$
(iii) Aux2 = $Identidad \otimes R_n$
(iv) Aux2 = $Aux2 \otimes Identidad$
(v) $R_n = Aux * Aux2$
(vi) Bandera = -1
(i) Bandera++
(5) Terminar

Tabla 5.2: Algoritmo para obtener la matriz que aplicada a n qubits, obtiene su reverso

donde $S = SWAP$, $C = CNOT$, $D = CNOTI$, $I = identidad$.

De (5.8) se ve que la recurrencia actua primero en los dos renglones más externos, los intercambia y pasa a los siguientes 2 más externos, así hasta llegar a la base de la recurrencia.

Por lo tanto, de lo descrito anteriormente, se ve que la matriz que aplicada a un qu-registro de n qubits y obtiene su reverso, se puede obtener procediendo con el algoritmo de la Tabla 5.2.

5.1.2. Ejemplo de objeto resultante para el algoritmo reverso

El código objeto resultante de aplicar la compilación al algoritmo reverso para un quvector de dimensión 2^2 es el siguiente:

- En la fase de sustitución tendríamos los siguientes símbolos en nuestro objeto:

```
Descripcion.txt = Prd Prd Cnt Prd Prd MTP H H Cnt MTP H H Cnt
```

- Aplicando la fase de sangria al objeto evolucionaría a lo siguiente:

```
Descripcion.txt =
    Prd Prd Cnt
            Prd Prd MTP H
                        H
                    Cnt
                MTP H
                    H
        Cnt
```

  como podemos ver se trata de la compuerta SWAP.

Si el quevector a encontrar su reverso hubiera sido generado por tres qubits entonces el objeto resultante de aplicar la compilación al algoritmo reverso para este quvector se vería de la siguiente forma:

- En la fase de sustitución tendríamos los siguientes símbolos en nuestro objeto:

```
Descripcion.txt = Prd Prd Prd Prd Prd MTP Prd Prd Cnt Prd Prd
        MTP H H Cnt MTP H H Cnt I MTP I Cnt MTP  Prd  Prd Cnt
        Prd Prd MTP H H Cnt MTP H H Cnt I Prd Prd MTP Prd Prd
        Cnt Prd Prd MTP H H Cnt MTP H H Cnt I MTP I  Prd  Prd
        MTP H H Cnt MTP H H MTP Prd Prd Cnt Prd  Prd  MTP H H
        Cnt MTP H H Cnt I Prd Prd MTP Prd Prd Cnt Prd Prd MTP
        H H Cnt MTP H H Cnt I MTP I Cnt MTP Prd Prd  Cnt  Prd
        Prd MTP H H Cnt MTP H H Cnt I MTP MTP I I I
```

- Aplicando la fase de sangria al objeto evolucionaría a lo siguiente:

```
Descripcion.txt =
      Prd Prd Prd Prd Prd MTP Prd Prd Cnt
                                  Prd Prd MTP H
                                              H
                                          Cnt
                                      MTP H
                                          H
                              Cnt
                          I
                      MTP I
                          Cnt
                  MTP Prd Prd Cnt
                              Prd Prd MTP H
```

```
                                    H
                                Cnt
                            MTP H
                                H
                    Cnt
                I
        Prd Prd MTP Prd Prd Cnt
                            Prd Prd MTP H
                                        H
                                    Cnt
                                MTP H
                                    H
                        Cnt
                    I
                MTP I
                    Prd Prd MTP H
                                H
                            Cnt
                        MTP H
                            H
            MTP Prd Prd Cnt
                        Prd Prd MTP H
                                    H
                                Cnt
                            MTP H
                                H
                    Cnt
                I
    Prd Prd MTP Prd Prd Cnt
                        Prd Prd MTP H
                                    H
                                Cnt
                            MTP H
                                H
                    Cnt
                I
            MTP I
                Cnt
        MTP Prd Prd Cnt
                    Prd Prd MTP H
                                H
                            Cnt
                        MTP H
                            H
                Cnt
            I
MTP MTP I
        I
    I
```

5.2. Transformada Discreta de Fourier Cuántica

Sea $f : [\![0, n-1]\!] \to \mathbb{C}$ una función. La *transformada discreta de Fourier* de f es la función $\hat{f} : [\![0, n-1]\!] \to \mathbb{C}$ tal que para cada $j \in [\![0, n-1]\!] : \hat{f} = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \exp\left(\frac{2\pi i j \kappa}{n}\right) f(\kappa)$. Aquí i es la raíz cuadrada de -1.

En $\mathbb{C}^n$ consideremos la base canónica formada por los vectores $\mathbf{e}_j = (\delta_{ij})_{i=0}^{n-1}$, $j = 0, \ldots, n-1$. Para cada vector $\mathbf{f} = \sum_{j=0}^{n-1} f(j)\mathbf{e}_j \in \mathbb{C}^n$, su *transformada discreta de Fourier* es $\mathbf{TDF}(\mathbf{f}) = \hat{\mathbf{f}} = \sum_{j=0}^{n-1} \hat{f}(j)\mathbf{e}_j \in \mathbb{C}^n$. Se puede ver que **TDF** es una transformación lineal y, respecto a la base canonica de $\mathbb{C}^n$, se representa por la matriz $\mathbf{TDF} = \frac{1}{\sqrt{n}} \left(\exp\left(\frac{2\pi i j \kappa}{n}\right)\right)_{j\kappa}$, la cual es en efecto unitaria, de hecho la matriz hermitiana de $\mathbf{TDF}$, $\mathbf{TDF}^H$, tiene la misma estructura que **TDF** salvo que los exponentes en cada entrada tienen el signo "-".

En particular, se tiene

$$\forall j \in [\![0, n-1]\!] : TDF(\mathbf{e}_j) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} exp(\frac{2\pi i j k}{n})\mathbf{e}_k \tag{5.9}$$

es decir:

$$TDF(\mathbf{f}) = \sum_{j=0}^{n-1} f(j) TDF(\mathbf{e}_j) \tag{5.10}$$

Ahora, supongamos que $n = 2^\nu$ es una potencia de 2. En este caso, la **TDF** puede calcularse mediante el procedimiento de la llamada *Transformada rápida de Fourier* **TRF** o **FFT** por sus siglas en inglés: *Fast Fourier Transform.* Este procedimiento es de complejidad en tiempo $O(\nu 2^\nu) = O(n \log n)$. Pero utilizando el paralelismo inherente en la computación cuántica, se le calculará en un tiempo $O(\nu)$.

Observemos, por un lado, que $\mathbb{H}_\nu = \mathbb{C}^n$, y por otro lado, de la ecuacion (5.9), que para los primeros valores de ν se tiene:

$\nu = 1$

$$\begin{aligned} TDF(\mathbf{e}_0) &= \frac{1}{\sqrt{2}} \sum_{k=0}^{1} exp(\frac{2\pi i 0 k}{2}) e_k = \frac{1}{\sqrt{2}} \sum_{k=0}^{1} exp(0)\mathbf{e}_k \\ &= \frac{1}{\sqrt{2}}(e_0 + e_1) = H(\mathbf{e}_0) \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} TDF(\mathbf{e}_1) &= \frac{1}{\sqrt{2}}\sum_{k=0}^{1} exp(\frac{2\pi i1k}{2})e_k = \frac{1}{\sqrt{2}}\sum_{k=0}^{1} exp(\pi ik)e_k \\ &= \frac{1}{\sqrt{2}}(exp(0)e_0 + exp(\pi i)e_1) = \\ &= \frac{1}{\sqrt{2}}(e_0 - e_1) = H(\mathbf{e}_1) \\ &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} \end{aligned}$$

$\nu = 2$

$$\begin{aligned} TDF(\mathbf{e}_{00}) &= \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{2\pi i0k}{4})e_k = \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(0)\mathbf{e}_k \\ &= \frac{1}{\sqrt{4}}(e_{00} + e_{01} + e_{10} + e_{11}) = \frac{1}{\sqrt{2}}(e_0 + e_1) \otimes \frac{1}{\sqrt{2}}(e_0 + e_1) \\ &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} TDF(\mathbf{e}_{01}) &= \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{2\pi i1k}{4})e_k = \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{\pi ik}{2})e_k \\ &= \frac{1}{\sqrt{4}}(exp(0)e_0 + exp(\frac{\pi i}{2})e_1 + exp(\pi i)e_2 + exp(\frac{3\pi i}{2})e_3 \\ &= \frac{1}{\sqrt{4}}(e_0 + ie_1 - e_2 - ie_3) = \frac{1}{\sqrt{4}}(e_{00} + ie_{01} - e_{10} - ie_{11}) \\ &= \frac{1}{\sqrt{2}}(e_0 - e_1) \otimes \frac{1}{\sqrt{2}}(e_0 + ie_1) \\ &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ i \end{bmatrix} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ i \\ -1 \\ -i \end{bmatrix} \end{aligned}$$

$$\begin{aligned}
TDF(\mathbf{e}_{10}) &= \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{2\pi i 2k}{4})e_k = \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\pi i k)e_k \\
&= \frac{1}{\sqrt{4}}(exp(0)e_0 + exp(\pi i)e_1 + exp(2\pi i)e_2 + exp(3\pi i)e_3 \\
&= \frac{1}{\sqrt{4}}(e_0 - e_1 + e_2 - e_3) = \frac{1}{\sqrt{4}}(e_{00} - e_{01} + e_{10} - e_{11}) \\
&= \frac{1}{\sqrt{2}}(e_0 + e_1) \otimes \frac{1}{\sqrt{2}}(e_0 - e_1) \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
TDF(\mathbf{e}_{11}) &= \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{2\pi i 3k}{4})e_k = \frac{1}{\sqrt{4}}\sum_{k=0}^{3} exp(\frac{3\pi i k}{2})e_k \\
&= \frac{1}{\sqrt{4}}(exp(0)e_0 + exp(\frac{3\pi i}{2})e_1 + exp(3\pi i)e_2 + exp(\frac{9\pi i}{2})e_3 \\
&= \frac{1}{\sqrt{4}}(e_0 - ie_1 - e_2 + ie_3) = \frac{1}{\sqrt{4}}(e_{00} - ie_{01} - e_{10} + ie_{11}) \\
&= \frac{1}{\sqrt{2}}(e_0 - e_1) \otimes \frac{1}{\sqrt{2}}(e_0 - ie_1) \\
&= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -i \end{bmatrix} = \frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ -i \\ -1 \\ i \end{bmatrix}
\end{aligned}$$

De manera general, a cada índice $j \in [\![0, 2^\nu - 1]\!]$ lo identificamos con la palabra $\varepsilon_j = \varepsilon_{j,v-1}\cdots\varepsilon_{j,1}\varepsilon_{j,0}$ que lo representa en base 2. Así, el vector básico $\mathbf{e}_j \in \mathbb{H}_\nu$ es el producto tensorial de los vectores básicos $\mathbf{e}_{\varepsilon_{j,k}} \in \mathbb{H}_1$. Tendremos entonces, en $\mathbb{H}_\nu$, que para cada $j = 0, \ldots, 2^\nu - 1$:

$$\begin{aligned}
TDF(e_{\varepsilon_j}) &= \bigotimes_{k=0}^{\nu-1} \frac{1}{\sqrt{2}}\left(e_0 + exp\left(\frac{\pi i j}{2^k}\right)e_1\right) \\
&= \tfrac{1}{\sqrt{2}}\left(e_0 + exp\left(\tfrac{\pi i j}{2^0}e_1\right)\right) \otimes \tfrac{1}{\sqrt{2}}\left(e_0 + exp\left(\tfrac{\pi i j}{2^1}e_1\right)\right) \otimes \cdots \otimes \tfrac{1}{\sqrt{2}}\left(e_0 + exp\left(\tfrac{\pi i j}{2^{\nu-1}}e_1\right)\right) \qquad (5.11)
\end{aligned}$$

La forma de los factores en este produto tensorial sugiere considerar los operadores $Q_k : \mathbb{H}_1 \to \mathbb{H}_1$ con representacion, respecto a la base canónica, dada mediante la matriz unitaria $Q_k = \begin{bmatrix} 1 & 0 \\ 0 & exp\left(\frac{\pi i}{2^k}\right) \end{bmatrix}$. De hecho, como en (5.11) la potencia va cambiando, podemos considerar más bien un correspondiente operador "controlado":

$Q^c_{kj} = \begin{bmatrix} 1 & 0 \\ 0 & exp\left(\pi i \frac{j}{2^k}\right) \end{bmatrix}$. Así por ejemplo, si $j = 1$ entonces $Q^c_{k1} = Q_k$ en tanto que si $j = 0$ entonces $Q^c_{k0} = I$ coincide con la identidad.

Obsérvese que para cada $\mathbf{x}_0 = \frac{1}{\sqrt{2}}(\mathbf{e}_0+\mathbf{e}_1) = H(\mathbf{e}_0)$ se tiene $Q^c_{kj}(\mathbf{x}_0) = \frac{1}{\sqrt{2}}(\mathbf{e}_0 + exp(\pi i \frac{j}{2^k})\mathbf{e}_1)$. Ahora, a cada $j \in [\![0, 2^\nu - 1]\!]$ representémoslo en base-2 mediante la palabra ε_j. Se tiene que para cada $\ell \in [\![0, \nu-1]\!]$, $\frac{\varepsilon_{j,\ell}2^\ell}{2^k} = \frac{\varepsilon_{j,\ell}}{2^{k-\ell}}$. Por tanto, $exp\left(\pi i \frac{j}{2^k}\right) = exp\left(\pi i \frac{\sum_{\ell=0}^{\nu-1}\varepsilon_{j,\ell}2^\ell}{2^k}\right) = \prod_{\ell=0}^{\nu-1} exp\left(\pi i \frac{\varepsilon_{j,\ell}}{2^{k-\ell}}\right)$ y en consecuencia, $Q^c_{kj} = Q^c_{k-\nu+1,\varepsilon_{j,\nu-1}} \circ \cdots \circ Q^c_{k-1,\varepsilon_{j,1}} \circ Q^c_{k,\varepsilon_{j,0}}$. Como k ha de variar entre 0 y $\nu - 1$ vemos que se ha de disponer de $2(2\nu - 1)$ compuertas de la forma $Q^c_{\kappa\varepsilon}, \kappa \in [\![-(\nu-1), \nu-1]\!], \varepsilon \in 0, 1$.

Obsérvese también que si $j < 2^{\nu_1}$, con $\nu_1 \leq \nu$ entonces todos los dígitos, en su representación binaria, con índices entre $\nu_1 - 1$ y $\nu - 1$ son 0, y por lo tanto las correspondientes compuertas controladas actuarán como la identidad. Definamos pues para cada $(j,k) \in [\![0, 2^\nu - 1]\!] \times [\![0, \nu-1]\!]$,

$$P_{jk} = Q^c_{k-\nu_1+1,\varepsilon_{j,\nu_1-1}} \circ \cdots \circ Q^c_{k-1,\varepsilon_{j,1}} \circ Q^c_{k,\varepsilon_{j,0}}, \tag{5.12}$$

donde $\nu_1 = \lceil \log_2 j \rceil + 1$. Tenemos pues: $P_{jk}(\mathbf{x}_0) = \frac{1}{\sqrt{2}}\left(\mathbf{e}_0 + exp\left(\pi i \frac{j}{2^k}\right)\right)\mathbf{e}_1)$.

Fijo $j \in [\![0, 2^\nu - 1]\!]$ para $k = 0, \cdots, \nu - 1$ los términos $P_{jk}(\mathbf{x}_0)$ van dando los de la derecha de la ec. (5.11) y éstos van apareciendo de izquierda a derecha según se les muestra ahí. Sin embargo, para cada $k \in [\![0, \nu-1]\!]$ observamos que en la definición (5.12) se está utilizando una notación algebraica, es decir, los operadores $Q^c_{k-\ell,\varepsilon_{j,\ell}}$ se aplican en orden inverso: de derecha a izquierda. De hecho, haciendo más explicita la definición (5.12) se tiene:

$$\begin{aligned}
Q^c_{0,\varepsilon_{j,0}}(\mathbf{x}_0) &= P_{j0}(\mathbf{x}_0) \\
Q^c_{1,\varepsilon_{j,0}} \circ Q^c_{0,\varepsilon_{j,1}}(\mathbf{x}_0) &= P_{j1}(\mathbf{x}_0) \\
Q^c_{2,\varepsilon_{j,0}} \circ Q^c_{1,\varepsilon_{j,1}} \circ Q^c_{0,\varepsilon_{j,2}}(\mathbf{x}_0) &= P_{j2}(\mathbf{x}_0) \\
\vdots \quad &\quad \vdots \\
Q^c_{\nu-1,\varepsilon_{j,0}} \circ \cdots \circ Q^c_{2,\varepsilon_{j,\nu-3}} \circ Q^c_{1,\varepsilon_{j,\nu-2}} \circ Q^c_{0,\varepsilon_{j,\nu-1}}(\mathbf{x}_0) &= P_{j\nu-1}(\mathbf{x}_0)
\end{aligned}$$

es decir, para cada $k \in [\![0, \nu-1]\!]$ se han de aplicar consecutivamente las compuertas $Q^c_{\ell,\varepsilon_{j,k-\ell}}$, con $\ell = 0, \cdots, k$, la cual selecciona a los dígitos en la representación binaria de j yendo del más significativo hacia el menos significativo.

Así pues, será necesario utilizar los operadores reverso para intercambiar el orden de los bits de cada indice $j \in [\![2^\nu - 1]\!]$.

Por ejemplo, si:

$$\begin{aligned}
j &= 6 \\
(j)_2 &= 1_2 1_1 0_0 \\
Reverso(j)_2 &= 0_2 1_1 1_0
\end{aligned}$$

Entrada: $n = 2^\nu, \mathbf{f} \in \mathbb{C}^n = \mathbb{H}_\nu$. **Salida:** $\hat{\mathbf{f}} = \text{TDF}(\mathbf{f}) \in \mathbb{H}_\nu$ **Procedimiento:** TDF(n, $\mathbf{f}$) 1. Sea $\mathbf{x}_0 := H(\mathbf{e}_0)$ 2. Para cada $j \in [\![0, 2^{\nu-1}]\!]$, o equivalente, para cada $(\varepsilon_{j,\nu-1} \cdots \varepsilon_{j,1}\varepsilon_{j,0}) \in 0, 1^\nu$ hágase en paralelo: (a) Para cada $k \in [\![0, \nu - 1]\!]$ hágase en paralelo: i. Sea $\delta := R_k(\ \varepsilon_j\|_k)$ el reverso de la cadena formada por los $(k + 1)$ bits menos significativos. ii. Sea $\mathbf{y}_{jk} := \mathbf{x}_0$ iii. Para $\ell = 0$ hasta k hágase { $\mathbf{y}_{jk} := Q^{c2}(\mathbf{y}_{jk}, \mathbf{e}_{\delta_{j,\ell}})$ } (b) Sea $\mathbf{y}_j := \mathbf{y}_{j0} \otimes \cdots \otimes \mathbf{y}_{j\nu-1}$ 3. Dése como resultado $\hat{\mathbf{f}} = \sum_{j=0}^{2^\nu-1} f_j \mathbf{y}_j$ 4. Terminar

Tabla 5.3: Algoritmo para calcular la transformada rápida de Fourier cuántica

Ahora bien, cada bit ε se representa por el vector básico $\mathbf{e}_\varepsilon$. Así que cada operador controlado $Q^c_{k,\varepsilon}$, cuyo dominio es $\mathbb{H}_1$ puede identificarse con el operador $\mathbf{x} \mapsto Q^{c2}(\mathbf{x}, \mathbf{e}_\varepsilon)$ donde

$$Q^{c2} = (I \otimes Q_k) \circ C \circ (I \otimes Q_k^H) \circ C \circ (Q_k \otimes I) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & exp\left(\pi i \frac{j}{2^k}\right)^2 \end{bmatrix} \quad (5.13)$$

con

$$Q_k \quad = \quad \begin{bmatrix} 1 & 0 \\ 0 & exp\left(\pi i \frac{j}{2^k}\right) \end{bmatrix}, C = Cnot$$

El algoritmo para calcular la transformada rápida de Fourier cuántica se puede ver en la Tabla 5.3. El cálculo de la transformada inversa discreta de Fourier, $\textbf{TIDF}(n, \mathbf{f})$ se hace de manera similar que el mostrado en la Tabla 5.3, cambiando la matriz Q_k por su hermitiana Q_k^H que sólo involucra el cambio de signo en la potencia de su elemento.

5.2.1. Ejemplo de objeto resultante para el algoritmo TDF cuántica

El código objeto resultante de aplicar la compilación al algoritmo de la *transformada discreta de Fourier* cuántica para un quvector de dimensión 2^2 es el siguiente:

- En la fase de sustitución tendríamos los siguientes símbolos en nuestro objeto:

```
Descripcion.txt =
  AdC 4 1 4 0 VTP VTP Emb 1 Lmb 2 2 App Prd MTP I RN 0 Prd
  Cnt Prd MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App
  H Inm 0 Lmb 2 2 App Prd MTP I RN 1 Prd Cnt Prd MTP I Her
  RN 1 Prd Cnt MTP RN 1 I VTP Inm 0 Lmb 2 2 App Prd  MTP I
  RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I  VTP
  Inm 0 App H Inm 0

  AdC 0 1 4 1 VTP VTP Emb 1 Lmb 2 2 App Prd MTP I RN 0 Prd
  Cnt Prd MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App
  H Inm 0 Lmb 2 2 App Prd MTP I RN 1 Prd Cnt Prd MTP I Her
  RN 1 Prd Cnt MTP RN 1 I VTP Inm 1 Lmb 2 2 App Prd  MTP I
  RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I  VTP
  Inm 0 App H Inm 0

  AdC 0 1 4 1 VTP VTP Emb 1 Lmb 2 2 App Prd MTP I RN 0 Prd
  Cnt Prd MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App
  H Inm 0 Lmb 2 2 App Prd MTP I RN 1 Prd Cnt Prd MTP I Her
  RN 1 Prd Cnt MTP RN 1 I VTP Inm 0 Lmb 2 2 App Prd  MTP I
  RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I  VTP
  Inm 1 App H Inm 0

  AdC 0 1 4 1 VTP VTP Emb 1 Lmb 2 2 App Prd MTP I RN 0 Prd
  Cnt Prd MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App
  H Inm 0 Lmb 2 2 App Prd MTP I RN 1 Prd Cnt Prd MTP I Her
  RN 1 Prd Cnt MTP RN 1 I VTP Inm 1 Lmb 2 2 App Prd  MTP I
  RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I  VTP
  Inm 1 App H Inm 0
```

Si se buscara la matriz que realiza la TDF cuántica pero para un quevector de dimensión 2^3 entonces el objeto resultante de aplicar la compilación al algoritmo de la TDF cuántica para este quvector se vería de la siguiente forma:

- En la fase de sustitución tendríamos los siguientes símbolos en nuestro objeto:

```
Descripcion.txt =
    AdC 8 1 8 0 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
    RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
    VTP Inm 0 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
    Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
    Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
    Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App H  Inm 0
    Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
```

```
2 Prd Cnt MTP RN 2 I VTP Inm 0 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 0  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 1 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 1 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 0  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 0 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 0 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 0  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 1 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 1 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 0  App H
```

```
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 0 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 0 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 1  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 1 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 0  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 1 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 1  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
VTP Inm 0 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 0 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 0 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 1  App H
Inm 0

AdC 0 1 8 1 VTP VTP VTP Emb 1 Lmb 2 2 App Prd MTP I
RN 0 Prd Cnt Prd MTP I Her RN 0 Prd Cnt MTP  RN 0 I
```

```
VTP Inm 1 App H Inm 0 Lmb 2 2 App  Prd  MTP I  RN 1
Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1 I  VTP
Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt Prd  MTP I
Her RN 0 Prd Cnt MTP RN 0 I VTP Inm 1  App H  Inm 0
Lmb 2 2 App Prd MTP I RN 2 Prd Cnt Prd MTP I Her RN
2 Prd Cnt MTP RN 2 I VTP Inm 1 Lmb 2 2 App Prd  MTP
I RN 1 Prd Cnt Prd MTP I Her RN 1 Prd Cnt MTP  RN 1
I VTP Inm 1 Lmb 2 2 App Prd MTP I RN 0 Prd Cnt  Prd
MTP I Her RN 0 Prd Cnt MTP RN 0 I VTP  Inm 1  App H
Inm 0
```

5.2.2. Tiempos de ejecución de la TDF cuántica

Los tiempos tomados de la ejecución de la TDF en una máquina local de manera estándar y después de ser transformada a símbolos es mostrado en la gráfica de la Figura 5.3.

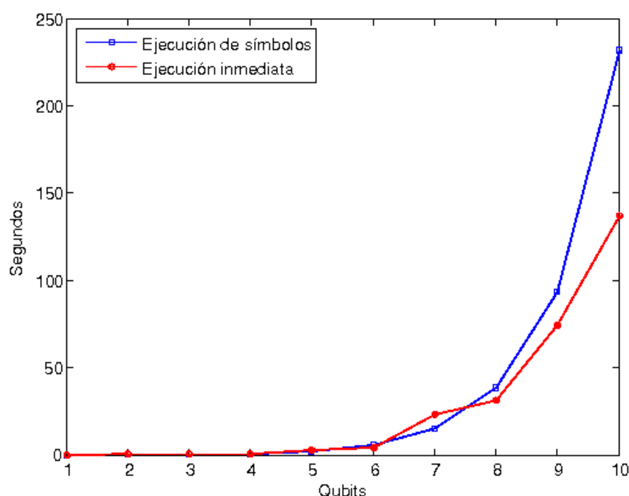

Figura 5.3: Esta gráfica muestra que el tiempo tomado para ejecutar el archivo que describe la TDF con los símbolos definidos por la gramática es superior al tiempo que tarda el programa en ejecutar de manera inmediata la TDF.

Se puede ver que ejecutar de manera inmediata es mejor para el caso de una sola PC, pero la cantidad de tiempo extra que toma la ejecución símbolica con respecto al tiempo de ejecución inmediata no es tan grande. La diferencia de tiempo se debe principalmente a que el programa ejecutor de símbolos realiza una lectura en el archivo de entrada, además de tener que implementar las pilas descritas en el algoritmo de ejecución.

Se implementó la TDF usando una arquitectura de cluster con las siguientes características:

- Cuatro nodos.
- Ocho procesadores AMD Opteron(tm) Processor 248
- Ocho gigabytes de memoria RAM.

Los tiempos tomados de la ejecución de la TDF en el cluster ejecutando de manera inmediata usando uno y tres procesadores se ven en la gráfica de la Figura 5.4.

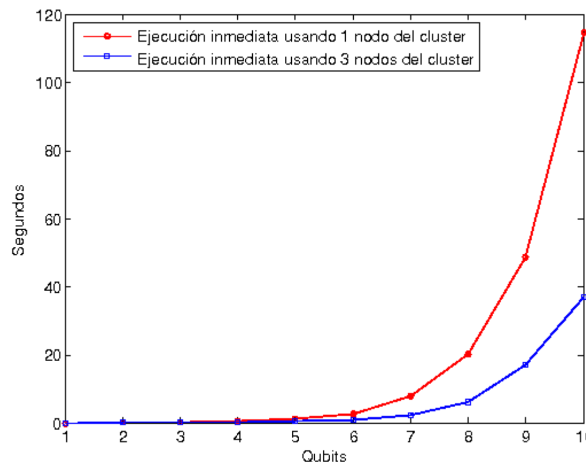

Figura 5.4: Se paralelizo él algoritmo de la TDF cuántica y se ejecutó usando 1 y 3 procesadores. En la gráfica se puede ver que apartir de $k = 3$, es mejor la ejecución del algoritmo usando 3 procesadores.

Se implementó la ejecución de los símbolos generados por el algoritmo de compilación. Los tiempos que se obtuvieron se pueden ver en la gráfica de la Figura 5.5. El programa fue ejecutado usando 1 y 3 procesadores. Apartir de $k = 4$, usar 3 procesadores es mejor que usar solamente 1. Esto se debe a que la transferencia de mensajes en los primeros k son más costosas que la operación a realizar.

En la gráfica de la Figura 5.6, se comparan los tiempos del programa que ejecuta de manera inmediata una instrucción cuántica, contra el que lee de un archivo los símbolos y los ejecuta usando el algoritmo de ejecución. Se puede ver que el primer esquema es más rápido. Esto se debe a que el esquema que utiliza símbolos tiene que leer de un archivo las instrucciones, (*para* $k = 10$ *el archivo tiene un tamaño de 5.4 megabytes*).

5.3. Algoritmo de Peter Shor

Este es un algoritmo de tipo cuántico y tiene el propósito de factorizar a un número entero dado n como el producto de dos enteros menores, si esto es posible, o bien indicar que n es primo, en otro caso.

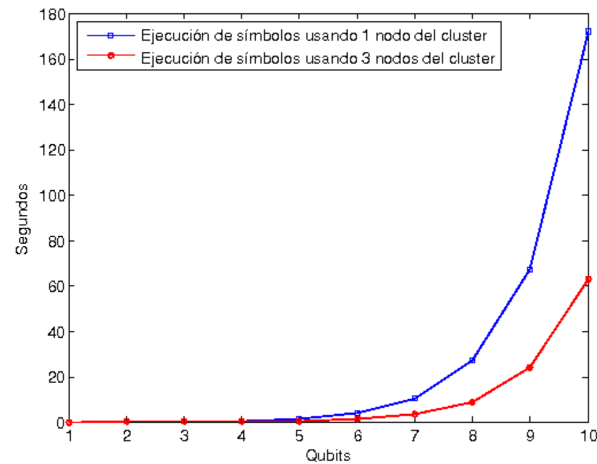

Figura 5.5: Se paralelizó el ejecutor de símbolos de la TDF cuántica. Se ejecuto usando 1 y 3 procesadores. En la gráfica se puede ver que apartir de $k = 4$, es mejor la ejecución del algoritmo usando 3 procesadores.

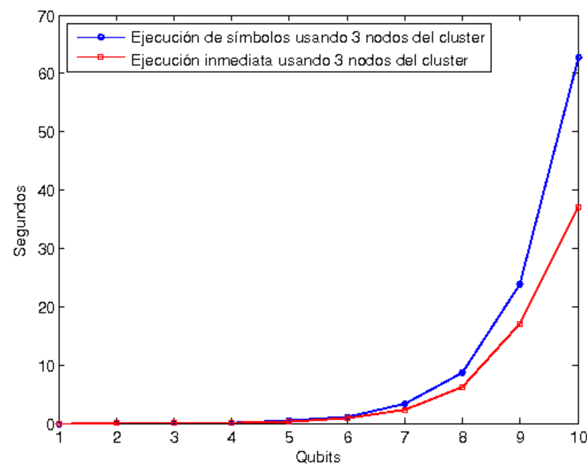

Figura 5.6: La gráfica muestra los tiempos tomados por las implementaciones en paralelo del algoritmo *Ejecutor* así como de la ejecución inmediata de la TDF, en ambos casos se utilizo 3 procesadores.

5.3.1. Pequeño recordatorio de teoría de números

Dado dos enteros n, m, su *máximo común divisor* es $d = \text{mcd}(n, m)$ tal que d divide a ambos n y m, es decir es un divisor común de n y m, y cualquier otro divisor común divide a d también. Se puede ver que d es el menor entero positivo que se puede escribir como una combinación lineal de n y m con coeficientes enteros. El *Algoritmo de Euclides* calcula, para dos enteros n y m dados, $d = \text{mcd}(n, m)$ y lo expresa de la forma $d = an + bm$, con $a, b \in \mathbb{Z}$.

Los enteros n y m son *primos relativos* si $\text{mcd}(n, m) = 1$, es decir, si no poseen un divisor común que no sea trivial. Sea $\Phi(n) = \{m \in [\![1, n]\!] \mid \text{mcd}(c, m) = 1\}$ el conjunto de enteros positivos primos relativos con n, menores que n. Se tiene que el número de elementos en $\Phi(n)$ es el valor de la *función de Euler* ϕ en n. Con la multiplicación módulo n, $\Phi(n)$ es un grupo de orden $\phi(n)$. Asi pues, si $m \in \Phi(n)$ entonces $m^{\phi(n)} = 1 \mod n$. Por tanto, para cada entero $m \in \Phi(n)$ existe un mínimo elemento r, divisor de $\phi(n)$, tal que $m^r = 1 \mod n$. Tal r se dice ser el *orden* de m en el grupo multiplicativo de residuos módulo n.

Sea n un entero para el cual se ha de buscar un factor entero no trivial. Elijamos un entero m tal que $1 < m < n$. Si $\text{mcd}(n, m) = d > 1$, entonces d es un factor no-trivial de n. En otro caso, $\text{mcd}(n, m) = 1$, y se tiene que m quedará en el grupo multiplicativo de residuos de n, i.e. $m \in \Phi(n)$. Si acaso m tuviera ahí un orden par r, entonces $k = m^{\frac{r}{2}}$ es tal que $k^2 = 1 \mod n$. En consecuencia, $(k-1)(k+1) = 0 \mod n$, es decir n divide al producto de dos números menores que él. Por tanto, los factores primos de n han de aparecer como factores de esos números. Así pues al calcular mcd $(n, k-1)$ y mcd $(n, k+1)$ obtendremos factores no-triviales de n.

Un primer problema en este procedimiento de decisión consiste entonces en encontrar un elemento de orden par en el grupo multiplicativo de residuos módulo n. Si se elige m al azar, la probabilidad de que al cabo de k intentos no se haya localizado un tal m es $2^{-\kappa\ell}$ y obviamente esto tiende a cero muy rápidamente al incrementar κ. Así pues, bien vale la pena repetir pruebas arbitrarias de selección de un elemento (impar) menor que n para localizar uno de orden par en el grupo multiplicativo de residuos módulo n.

Sin embargo, desde el punto de vista computacional, el mayor problema que presenta el algoritmo descrito radica en el cálculo del orden del elemento actual m en $\Phi(n)$: el número de potencias de m a calcular es del orden de $\phi(n)$ que a su vez es de orden n.

Sea $\nu = \lceil \log_2 n \rceil$ el número de bits necesarios para escribir a n, es decir, sea ν el *tamaño* de n. Resulta claro que $O(n) = O(2^\nu)$ lo cual indica que el procedimiento anterior es de complejidad exponencial respecto al tamaño de la entrada. El algoritmo de Peter Shor se fundamenta en un procedimiento polinomial en ν para realizar la tarea de calcular el orden de un elemento.

5.3.2. Algoritmo cuántico para el cálculo de órdenes

Supongamos dado $n \in \mathbb{N}$. Sea $\nu = \lceil \log_2 n \rceil$ su tamaño. Sea μ tal que $n^2 \leq 2^\mu < 2n^2$, es decir, $\mu = \lceil 2\log_2 n \rceil$. Se considerá $\mu + \nu$ qubits y se trabajará en el espacio $\mathbb{H}_{\nu+\mu} = \mathbb{H}_\mu \otimes \mathbb{H}_\nu$, de dimensión $2^{\mu+\nu} = 2^\mu \cdot 2^\nu$.

Consideremos la función

$$f_0 : \mathbb{Z} \times \mathbb{Z} \times \Phi(n) \to [\![0, n-1]\!] \subset [\![0, 2^\nu - 1]\!], (i, j, m) \mapsto f_0(i, j, m) = (j + m^i) \bmod n \tag{5.14}$$

Por el Teorema Pequeño de Fermat se tiene que f_0 es periódica respecto a su primer argumento y su período es el orden de m en $\Phi(n)$, el cual es un divisor de $\phi(n) = \text{card}(\Phi(n))$, naturalmente también es periódica respecto a su segundo argumento y su período es n. Así pues, para cualesquiera dos números enteros $r, s \in \mathbb{Z}$, $f_0(i, j, m) = f_0(i + a\phi(n), j + bn, m)$. Definamos

$$\begin{aligned} f_1 : [\![0, 2^\mu - 1]\!] \times [\![0, 2^\nu - 1]\!] \times \Phi(n) &\to [\![0, 2^\nu - 1]\!], \\ (i, j, m) \mapsto f_1(i, j, m) &= \begin{cases} f_0(i, j, m) & \text{si } j < n \\ j & \text{si } j \geq n \end{cases} \end{aligned} \tag{5.15}$$

Observamos que para cada i, m, la restricción $j \mapsto f_1(i, j, m)$ al intervalo de enteros $[\![0, n-1]\!]$ es una permutacion y su restricción a $[\![n, 2^\nu - 1]\!]$ coincide con la identidad.

Para cada $m \in \Phi(n)$ definimos un operador lineal unitario $V_m : \mathbb{H}_{\mu+\nu} \to \mathbb{H}_{\mu+\nu}$ haciéndolo actuar en los vectores básicos como

$$V_m : \mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_j} \mapsto \mathbf{e}_{\varepsilon_j} \otimes \mathbf{e}_{\varepsilon_{f_1(i,j,m)}} \tag{5.16}$$

5.3.3. Elementos con orden potencia de 2

Supongamos dado $m \in \Phi(n)$ y que éste es tal que su orden r es una potencia de 2.

Sea $P_1 = H^{\otimes\mu} \otimes I^{\otimes\nu}$ donde $H, I : \mathbb{H}_1 \to \mathbb{H}_1$ son los operadores de Hadamard e identidad respectivamente. Por la ec. (1.6) $P_1(\mathbf{e}_0 \otimes \mathbf{e}_0) = \frac{1}{\sqrt{2}^\mu} \sum_{\varepsilon \in \{0,1\}^\mu} \mathbf{e}_\varepsilon \otimes \mathbf{e}_0$. Escribamos $s_1 = P_1(\mathbf{e}_0 \otimes \mathbf{e}_0)$. Ahora, aplicando el operador V_m, resulta $V_m(\mathbf{s}_1) = \frac{1}{\sqrt{2}^\mu} \sum_{i=0}^{2^\mu - 1} \mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_{f_1(i,0,m)}}$. Escribamos $s_2 = V_m(s_1)$.

Ya que $f_1(i, 0, m) = m^i \bmod n$, se tiene que f_1 es una función periódica de período r respecto a i. Sea $I_{i_0} = \{i \mid 0 \leq i \leq 2^\mu - 1 : i = i_0 \bmod r\}$ la clase de índices que dejan residuos i_0 al dividírseles entre r. Claramente $[\![0, 2^\mu - 1]\!] = \bigcup_{i_0=0}^{r-1} I_{i_0}$, y la cardinalidad de cada conjunto I_{i_0} es $s = \frac{2^\mu}{r}$, el cual número, para este caso, es entero. Por tanto, es posible reescribir

$$\mathbf{s}_2 = \frac{1}{\sqrt{2}^\mu} \sum_{i_0=0}^{r-1} \left(\sum_{i \in I_{i_0}} \mathbf{e}_{\varepsilon_i} \right) \otimes \mathbf{e}_{\varepsilon_{m^{i_0}}} \tag{5.17}$$

Si aquí se aplica el postulado de medición, entonces se elegirá a un vector de la forma $\mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_m i_0}, i \in I_{i_0}$, para una potencia fija $i_0 \leq r$, con probabilidad $\frac{r}{2^\mu}$. El estado correspondiente a esta situación es de la forma

$$\mathbf{s}_3 = \sum_{i=0}^{2^\mu - 1} g(i) \mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_m i_0} \tag{5.18}$$

donde $g : i \mapsto \begin{cases} \sqrt{\frac{r}{2^\mu}} & \text{si } i \in I_{i_0} \\ 0 & \text{si } i \notin I_{i_0} \end{cases}$

La función g también es periódica, de período r. Ahora, se tiene que la transformada de Fourier de $g, \breve{g}$ será también periódica pero de período proporcional a $\frac{1}{r}$.

Calculemos la transformada inversa discreta de Fourier de $\mathbf{s}_3$ (en lo sucesivo, i denotará a la raíz cuadrada de - 1):

$$\breve{\mathbf{s}_3} = \mathrm{TDF}^H(\mathbf{s}_3) = \sqrt{\frac{r}{2^\mu}} \sum_{k=0}^{s-1} \left(\frac{1}{\sqrt{2^\mu}} \sum_{\ell=0}^{2^\mu - 1} \exp\left(-\frac{2\pi i \ell}{2^\mu} (\kappa r + i_0) \right) \mathbf{e}_\ell \right) \otimes \mathbf{e}_{\varepsilon_m i_0}, \tag{5.19}$$

y al intercambiar el orden de las sumatorias se obtiene:

$$\mathbf{s}_4 = \breve{\mathbf{s}_3} = \frac{1}{\sqrt{r}} \left(\sum_{\ell=0}^{2^\mu - 1} \left(\frac{1}{s} \sum_{k=0}^{s-1} \exp\left(-\frac{2\pi i \ell \kappa}{s} \right) \right) \exp\left(-\frac{2\pi i \ell i_0}{2^\mu} \right) \mathbf{e}_\ell \right) \otimes \mathbf{e}_{\varepsilon_m i_0} \tag{5.20}$$

Ya que $\exp(-\frac{2\pi i \ell}{s})$ es una raíz s-ésima de la unidad, se tiene $\frac{1}{s} \sum_{k=0}^{s-1} \exp(-\frac{2\pi i \ell \kappa}{s})$ será 1 o 0 en función de que ℓ sea o no un múltiplo entero de s, es decir un número de la forma $\ell = ts$, con $t = 0, \ldots, r-1$. Aquí es esencial el hecho de que s es entero. Así pues, de (5.20),

$$s_4 = \frac{1}{\sqrt{r}} \left(\sum_{t=0}^{r-1} \exp\left(-\frac{2\pi i t i_0}{r} \right) \mathbf{e}_{\frac{2^\mu t}{r}} \right) \otimes \mathbf{e}_{\varepsilon_m i_0} \tag{5.21}$$

Las relaciones (5.18) y (5.21), que expresan a $\mathbf{s}_4$ y $\mathbf{s}_4 = \breve{\mathbf{s}_3}$, involucran ambas el orden r. Pero hay una diferencia esencial entre ellas: En (5.18) los índices ahí denotados i del primer qubit correspondiente a coeficientes no-nulos dependen de la potencia "aleatoria"i_0, en tanto que en (5.21) no dependen de ésa, e involucran sin embargo, a r.

Si ahora se aplica el postulado de medición se obtendrá un valor de la forma $\frac{2^\mu t_0}{r}$, con $t_0 \in [\![0, r-1]\!]$, cada uno con probabilidad r^{-1}. Si $t_0 = 0$, entonces no es posible obtener ninguna información acerca de r y se ha de repetir el procedimiento otra vez. En otro caso, al dividir entre 2^μ se tiene el valor racional $\frac{r_0}{r_1} = \frac{t_0}{r}$. Se conoce a r_0 y r_1 mas aún no se conoce a t_0 ni r. Sin embargo, necesariamente r_1 divide a r. Así pues, se puede aplicar de nuevo el algoritmo cuántico a partir de $m_1 = m^{r_1}$. Procediendo

Entrada: $n \in \mathbb{N}, m \in \Phi(n)$ de orden potencia 2.
Salida: r tal que $r \mid o(m)$
Procedimiento: DivisorOrdenPotencia2(n, m)

1. Sea $\nu := \lceil \log_2 n \rceil, \mu := 2\nu$
2. Definase $V_m : \mathbb{H}_{\mu+\nu} \to \mathbb{H}_{\mu+\nu}$ como en la ec. (5.16)
3. Sea $\mathbf{s}_1 := (H^{\otimes\mu} \otimes I^{\otimes\nu})(\mathbf{e}_0 \otimes \mathbf{e}_0)$
4. Sea $\mathbf{s}_2 := V_m(\mathbf{s}_1)$
5. Sea $\mathbf{s}_3 := \sum_{i=0}^{2^\mu-1} g(i)\mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_m i_0}$ el estado equivalente a "tomar una medición.en $\mathbf{s}_2$. Entonces g queda determinada como en la ec. (5.18)
6. Sea $\mathbf{s}_4 :=$ TIDF$(2^\mu, \mathbf{s}_3)$ la transformada inversa discreta de Fourier de $\mathbf{s}_3$
7. Sea $\mathbf{e}_{\varepsilon_\kappa} \otimes \mathbf{e}_{\varepsilon_m i_0}$ una medición de $\mathbf{s}_4$.
8. Si $\kappa == 0$ repítase desde el paso 3.
   En otro caso sea $\frac{r_0}{r_1} = \frac{\kappa}{2^{nu}}$ y dése como resultado r_1.
9. Terminar

Tabla 5.4: Algoritmo para localizar divisores de ordenes de elementos.

recursivamente se obtiene una factorización $r = r_1 r_2 \cdots r_p$ conteniendo a lo mas $\log_2 r$ factores.

En resumen, el algoritmo para localizar divisores de órdenes de elementos se puede ver en la Tabla 5.4 y el algoritmo para calcular órdenes de elementos se puede ver en la Tabla 5.5.

5.3.4. Elementos con orden arbitrario

Dejemos de suponer que el orden r de m sea una potencia de 2 en $\Phi(n)$. Siguiendo la misma línea que en el caso anterior, sea V_m definido como en la ec. (5.16). Sea $\mathbf{s}_1 = (H^{\otimes\mu} \otimes I^{\otimes\nu})(\mathbf{e}_0 \otimes \mathbf{e}_0)$ y $\mathbf{s}_2 = \mathbf{V}m(\mathbf{s}_1)$. Reagrupando los términos según se hizo en la ec. (5.17) se puede escribir

$$\mathbf{s}_2 = \frac{1}{\sqrt{2}^\mu} \sum_{j=0}^{r-1} \left(\sum_{i \in I_{i_0}} \mathbf{e}_{\varepsilon_i} \right) \otimes \mathbf{e}_{\varepsilon_m j} \tag{5.22}$$

donde los conjuntos I_{i_0} son clases de equivalencia, congruentes con j, modulo r, pero ahora no son de la misma cardinalidad. Si $u = 2^\mu \bmod r$ y $s = (2^\mu - u)/r$ entonces u clases tendrán $s+1$ elementos y las restantes tendrán s elementos. Definamos $s_j = s+1$ para $j = 1, \ldots, u$ y $s_j = s$ para $j = u+1, \ldots, r-1, 0$. Entonces el estado que representa el tomar una medición, como en la ec. (5.18), es, para algún $i_0 \in [\![0, r-1]\!]$:

Entrada: $n \in \mathbb{N}, m \in \Phi(n)$ de orden potencia 2.
Salida: r tal que $r \mid o(m)$
Procedimiento: OrdenPotencia2(n, m)
1. Sea inicialmente $r := 1$ y $m_1 := m$
2. Repitase
   (a) sea $r_1 :=$ DivisorOrdenPotencia2(n, m_1)
   (b) actualícese $r := r \cdot r_1$
   (c) actualícese $m_1 := m_1^{r_1} \mathrm{mod} n$
   hasta tener $m_1 == 1$
3. Dése como resultado r
4. Terminar

Tabla 5.5: Algoritmo para calcular órdenes de elementos

$$\mathbf{s}_3 = \sum_{i=0}^{2^\mu - 1} g(i) \mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_m i_0} \tag{5.23}$$

donde $g : i \mapsto \begin{cases} \frac{1}{\sqrt{s_{i_0}}} & \text{si } i \in J_{i_0} \\ 0 & \text{si } i \notin J_{i_0} \end{cases}$

Calculando la transformada inversa discreta de Fourier y reagrupando términos, como en la ec. (5.20), se obtiene

$$\mathbf{s}_4 = \check{\mathbf{s}}_3 = \frac{1}{\sqrt{2^\mu}} \left(\sum_{\ell=0}^{2^\mu - 1} \left(\frac{1}{\sqrt{\mathbf{s}_{i_0}}} \sum_{\kappa=0}^{s_{i_0}-1} \exp\left(-\frac{2\pi i \ell \kappa r}{2^\mu} \right) \right) \exp\left(-\frac{2\pi i \ell i_0}{2^\mu} \right) \mathbf{e}_\ell \right) \otimes \mathbf{e}_{\varepsilon_m i_0} \tag{5.24}$$

pero en este caso el coeficiente que involucra a la suma interior nunca se anula (como r no necesariamente divide a 2^μ, aquí no se está sumando un "juego completo" de raíces s_{i_0}-ésimas de la unidad). Al tomar una medición del primer qubit, la probabilidad de que se elija a $\mathbf{e}_\ell \otimes \mathbf{e}_{\varepsilon_m i_0}$ es entonces

$$P(\ell) = \frac{1}{\sqrt{2^\mu \mathbf{s}_{i_0}}} \left| \sum_{\kappa=0}^{\mathbf{s}_{i_0}-1} \exp\left(-\frac{2\pi i \ell \kappa r}{2^\mu} \right) \right|^2 \tag{5.25}$$

y los máximos de esos valores corresponden a enteros $\ell =$ `EnteroMásProximo`$(\frac{\kappa 2^\mu}{r})$, $\kappa = 0, \ldots, r-1$. Supongamos que tras una medición se haya elegido $\mathbf{e}_{\ell_\kappa} \otimes \mathbf{e}_{\varepsilon_m i_0}$, con $\ell_\kappa =$ `EnteroMásProximo`$(\frac{\kappa 2^\mu}{r})$. Entonces, al dividir ese índice entre 2^μ se obtiene $\frac{\ell_\kappa}{2^\mu} \sim \frac{\kappa}{r}$, y de aquí se quiere conocer r. Para esto hay que recordar la noción de *fracciones continuadas*.

Si $\frac{p}{q}$ es un número racional no-negativo, su *fracción continuada* es

Entrada: $\frac{p}{q} \in \mathbb{Q}$
Salida: $[a_0, a_1, \ldots, a_\nu]$: fracción continuada que representa a $\frac{p}{q} \in \mathbb{Q}$
Procedimiento: `FracciónContinuada`$\left(\frac{p}{q}\right)$
1. Sea inicialmente $lst := []$ (lista vacía) y $xact := \frac{p}{q}$
2. Mientras el denominador de $xact$ sea mayor que 1 hágase
(a) sea $i :=$ `ParteEntera`$(xact)$
(b) escríbase $\frac{p_1}{q_1} = xact$
(c) actualícese $xact := \frac{q_1}{p_1 - iq_1}$
(d) actualícese $lst := lst * [i]$
3. Actualícese $lst := lst * [xact]$
4. Dése como resultado lst

Tabla 5.6: Algoritmo para calcular fracciones continuadas.

$$\frac{p}{q} = a_0 + \frac{1}{a_1 + \frac{1}{\ldots + \frac{1}{a_\nu}}} = [a_0, a_1, \ldots, a_\nu] \tag{5.26}$$

donde $a_0, a_1, \ldots, a_\nu$ son enteros no-negativos. Para cada $w \leq \nu$, la fracción continuada $[a_0, a_1, \ldots, a_w]$ se dice ser el *w-ésimo convergente* de $\frac{p}{q}$, y, en efecto, cada convergente es una aproximación racional a $\frac{p}{q}$. El algoritmo para calcular fracciones continuadas se puede ver en la Tabla 5.6.

Así pues, luego de haber realizado una medición y haber obtenido el valor $\frac{\ell_\kappa}{2^\mu} < 1$, se calcula su fracción continuada $[a_0, a_1, \ldots, a_\nu](a_0 = 0)$ y los correspondientes convergentes $[c_0, c_1, \ldots, c_\nu]$ (también $c_0 = 0$) y entre estos se selecciona a aquellos cuyos denominadores r_j sean menores que n, los cuales han de ser divisores del orden r de m.

En resumen, esta vez el algoritmo para localizar divisores de órdenes de elementos se puede ver en la Tabla 5.7.

Habiendo obtenido divisores de órdenes, se puede proceder a obtener los órdenes de manera similar a como se bosquejó en el procedimiento `OrdenPotencia2`, mas en este caso hay que llevar un recuento de las varias posibilidades de divisores que arroja el procedimiento `DivisorOrden` descrito anteriormente.

Entrada: $n \in \mathbb{N}, m \in \Phi(n)$
Salida: r tal que $r \mid o(m)$
Procedimiento: DivisorOrden(n, m)

1. Sea $\nu := \lceil \log_2 n \rceil, \mu := [\![2\log_2 n]\!]$
2. Definase $V_m : \mathbb{H}_{\mu+\nu} \to \mathbb{H}_{\mu+\nu}$ como en la ec. (5.16)
3. Sea $\mathbf{s}_1 := (H^{\otimes \mu} \otimes I^{\otimes \nu})(\mathbf{e}_0 \otimes \mathbf{e}_0)$
4. Sea $\mathbf{s}_2 := V_m(\mathbf{s}_1)$
5. Sea $\mathbf{s}_3 := \sum_{i=0}^{2^\mu - 1} g(i) \mathbf{e}_{\varepsilon_i} \otimes \mathbf{e}_{\varepsilon_m i_0}$ el estado equivalente a "tomar una medición" en $\mathbf{s}_2$. Entonces g queda determinada como en la ec. (5.18)
6. Sea $\mathbf{s}_4 :=$ TIDF$(2^\mu, \mathbf{s}_3)$ la transformada inversa discreta de Fourier de $\mathbf{s}_3$
7. Sea $\mathbf{e}_{\varepsilon_\kappa} \otimes \mathbf{e}_{\varepsilon_m i_0}$ una medición de $\mathbf{s}_4$.
8. Si $\ell == 0$ repitase desde el paso 3. En otro caso
   (a) sea $[a_0, a_1, \ldots, a_\nu] := \texttt{FracciónContinuada}\left(\frac{\ell_k}{2^\mu}\right)$
   (b) sea $[c_0, c_1, \ldots, c_\nu]$ la lista convergente y
   (c) dése como resultado la lista de denominadores, de los convergentes, que sean menores que n
9. Terminar

Tabla 5.7: Algoritmo para localizar divisores de órdenes de elementos

Capítulo 6

Conclusiones y trabajo futuro

En este capitulo hablaremos primero de las conclusiones a las que llegamos seguido de algunas sugerencias para trabajos futuros que se pueden o deben realizar para extender las aplicaciones de este trabajo de tesis.

6.1. Conclusiones

El tema principal de esta tesis fue generar un esquema que resolviera el problema de ejecutar un algoritmo cuántico escrito en un cierto lenguaje y este mismo pueda ser ejecutado en cualquier otra plataforma.

Para llegar a nuestra meta nos enfocamos en diversos puntos, entre los principales fue el de definir de manera formal qué cosa es un algoritmo cuántico. Este problema lo trasladamos a un campo más familiar para nosotros, las gramáticas formales ya que nuestra formación es como computologos y no como físicos. Nuestra gramática formal está definida de manera que es capaz de generar cualquier algoritmo cuántico existente, al menos en la práctica.

Nuestra gramática nos otorga las restricciones necesarias para generar el lenguaje de los algoritmos cuánticos y con ello podemos restringir al programador sobre qué herramientas dispone para crear su programa.

Con nuestra definición logramos que los programas resultantes de un algoritmo cuántico sea uniforme. De este paso debimos crear un nuevo procedimiento para trasladar nuestro programa a un objeto intermedio el cual debería tener un formato que pueda ser ejecutado sin ambigüedad por otro programa escrito en una arquitectura diferente sin necesidad de hacer grandes modificaciones al programa ejecutor.

Para trasladar nuestro código original a un código objeto intermedio desarrollamos un *compilador* que lo que realiza es traducir de manera eficiente nuestro programa al objeto deseado, utilizando diversas estructuras de datos así como esquemas matemáticos para representar nuestra salida que al final se puede ver como una salida símbolica del algoritmo cuántico.

Esta salida símbolica es la entrada para nuestro ejecutor el cual lee este objeto teniendo la certeza que va a ser correcto gracias a nuestra gramática. Las operaciones

que realiza el ejecutor para leer el objeto son fácilmente implementadas en la mayoría de los lenguajes de programación ya que se trata principalmente de la lectura de archivos.

Después de la lectura de nuestro archivo objeto tenemos que obtener la salida *numerica* del algoritmo cuántico original, para hacer esto optimizamos nuestras operaciones básicas esto debido a que ejecutar un algoritmo cuántico en una computadora clásica requiere una gran cantidad de memoria y de tiempo de cómputo.

La optimización que hicimos sobre nuestras operaciones básicas se basó en el uso de memoria RAM. Observamos que la cantidad de memoria necesaria para ejecutar un algoritmo cuántico era muy superior a la que disponíamos, es por eso que decidimos utilizar el siguiente dispositivo de almacenamiento mas grande, es decir, el disco duro. Nuestras operaciones básicas son eficientes en este sentido debido a que se adaptan a la cantidad de memoria RAM disponible y por lo tanto también son eficientes en velocidad.

Para probar que nuestro esquema es funcional y eficiente decidimos implementar tres algoritmos cuánticos. El algoritmo cuántico más importante desarrollado hasta el momento es al algoritmo de Peter Sort. Se le considera el más importante debido a sus implicaciones en los algoritmos criptográficos ya que estos basan su seguridad en lo difćil que es factorizar un numero entero en sus primos y es justamente lo que ataca el algoritmo Sort, es por eso que elegimos este algoritmo como caso principal de prueba. El corazón de este algoritmo es la *Transformada Discreta de Fourier Cuántica* por lo que éste también pertenece a nuestro conjunto de casos de pruebas.

Implementar un algoritmo existente el cual se ha probado en muchas ocasiones podría no parecer importante es por eso que desarrollamos un algoritmo cuántico propio al cual le llamamos *Reverso*.

Las plataformas que utilizamos para nuestros casos de prueba fueron dos, las cuales tratamos que sean lo más diferentes posibles. La primera elección fue una plataforma clásica de programación. Esta plataforma consta del sistema operativo *Fedora*, el lenguaje de programación *C* y una PC como arquitectura de cómputo. Para este primer esquema observamos los tiempos y las salidas que generaban nuestros algoritmos y los comparamos con los resultados obtenidos en nuestra segunda implementación. La segunda plataforma consta del sistema operativo *Red Hat*, el lenguaje *MPI* y un *cluster* como arquitectura. De las observaciones en los tiempos de ejecución observamos que las características del cluster fueron conservadas, es decir, utilizar una plataforma diferente para ejecutar nuestro algoritmo cuántico mantienen sus características con lo que logramos la meta planteada al principio de esta tesis, generar un esquema que resolviera el problema de ejecutar un algoritmo cuántico escrito en un cierto lenguaje y este mismo pueda ser ejecutado en cualquier otra plataforma.

6.2. Trabajo futuro

Para trabajo futuro planteamos que se desarrolle una mayor cantidad de casos de prueba. Esta propuesta incluye el desarrollar algoritmos cuánticos propios ya que esto es una tarea complicada pero utilizando las herramientas creadas en esta tesis la dificultad se reduce.

Crear herramientas de más alto nivel para poder facilitar la implementación y estudio de los algoritmos cuánticos actuales.

En esta tesis utilizamos software tanto para la generación de nuestro código objeto como para la ejecución, pero desarrollar un ejecutor genérico que reciba este objeto y lo ejecute podría ser implementado en Hardware lo cual, junto con nuestro esquema, llevaría a la estandarización de los algoritmos cuánticos. Podría pensarse en este ejecutor como una primera emulación de una computadora cuántica.

Apéndice A

Disco de programas, pruebas y desarrollos

El disco compacto que acompaña a la presente tesis tiene la siguiente estructura:

1. Documento de tesis

   a) Tesis: Intérprete serial y paralelo para algoritmos de Cómputo Cuántico.

2. Presentaciones y Reportes (Reportes parciales)

   a) Protocolo de Tesis. Documento: Protocolo Tesis.

   b) Demostración del porque no es posible reducir el orden de la multiplicación de un vector por una matriz cuadrada, entregado al Dr. Guillermo Morales Luna. Documento: Demostración 1.

   c) Reporte Entregado en el seminario de Tesis impartido por el Dr. Francisco Rodríguez Henríquez. Documento: Estado del Arte.

   d) Reporte entregado en el Seminario de Tesis impartido por la Dra. Sonia Mendoza Chapa. Documento: Reporte1.

   e) Presentación en el seminario de Tesis impartido por el Dr. Gerardo de la Fraga. Documento: Presentación1.

   f) Prsentación final.

3. Programas

   a) Manual de uso de los programas desarrollados.

   b) Código fuente de las funciones básicas implementadas.

   c) Código fuente del algoritmo reverso de quvectores de ejecución inmediata.

   d) Código fuente del algoritmo reverso de quvectores de ejecución simbólica.

   e) Código fuente del algoritmo la TDF de ejecución inmediata.

f) Código fuente del algoritmo la TDF de ejecución simbólica.

g) Código fuente del algoritmo la TDF de ejecución simbólica en cluster.

h) Código fuente del algoritmo de Peter Shor de ejecución inmediata.

i) Código fuente del algoritmo de Peter Shor de ejecución simbólica.

j) Código fuente del algoritmo de Peter Shor de ejecución simbólica en cluster.

k) Binario del algoritmo de reverso de quvectores de ejecución inmediata.

l) Binario del algoritmo de reverso de quvectores de ejecución simbólica.

m) Binario del algoritmo de la TDF de ejecución inmediata.

n) Binario del algoritmo de la TDF de ejecución simbólica.

ñ) Binario del algoritmo de la TDF de ejecución simbólica en cluster.

o) Binario del algoritmo de Peter Shor de ejecución inmediata.

p) Binario del algoritmo de Peter Shor de ejecución simbólica.

q) Binario del algoritmo de Peter Shor de ejecución simbólica en cluster.

r) Manaual de uso de los programas implementados.

s) Algunas Corridas de los programas implementados.

Referencias

[1] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Read to the Society in 1936, but published in 1937. Correction in volume 43, 544–546.

[2] Alonzo Church. Correction to a note on the entscheidungsproblem. *J. Symbolic Logic*, 1(3):101–102, 1936.

[3] Richard Phillips Feynman. *Quantum computation and quantum information*. Laurie M Brown, World Scientific, Northwestern University, USA, 2005.

[4] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, 41(2):303–332, 1999. http://dx.doi.org/10.1137/S0036144598347011.

[5] C. Lavor, L. R. U. Manssur, and R. Portugal. Shor's algorithm for factoring large integers, 2003. http://www.citebase.org/abstract?id=oai:arXiv.org:quant-ph/0303175.

[6] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *IEEE Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[7] Ahmed Usman Khalid, Zeljko Zilic, and Katarzyna Radecka. Fpga emulation of quantum circuits. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[8] M. Fujishima. Fpga-based high-speed emulator of quantum computing. In *FPT '03: Proceedings. 2003 IEEE International Conference on Field-Programmable Technology*, pages 21–26. IEEE, 2003.

[9] A. Kawaguchi, K. Shimizu, Y. Tokura, and N. Imoto. Classical simulation of quantum algorithms using the tensor product representation, 2004. http://www.citebase.org/abstract?id=oai:arXiv.org:quant-ph/0411205.

[10] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, New York, NY, USA, 1996. ACM. http://doi.acm.org/10.1145/237814.237866.

[11] Dr. Bernhard Ömer. Quantum programming in qcl, 2000. http://tph.tuwien.ac.at/~oemer/qcl.html.

[12] R. L. Sanevelly, V. V. Haggard. A procedure for designing a translator from c to vhdl. *Southeastern Symposium On Sytem Theory*, 34:329–333, 2002.

[13] R.L. Jie Chen Haggard. Extraction of parallel hardware during c to vhdl translation. In *STOC '02: Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pages 334–338. IEEE, 2002.

[14] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, 2003. http://dx.doi.org/10.1109/MC.2003.1220583.

[15] Hulston C.K.J.; Redlich P.J.; Jackson W.R.; Marshall M.; Larkins F.P.; Bshouty N.H. On the additive complexity of 2 x 2 matrix multiplication. *Information Processing Letters*, 56(6):329–335(7), December 1995.

[16] D. Boku T. Sato M. Ohtaki, Y. Takahashi. Parallel implementation of strassen's matrix multiplication algorithm for heterogeneous clusters. In *STOC '04: Proceedings. 18th International Parallel and Distributed Processing Symposium*, page 112. IEEE, 2004.

[17] Frédéric Suter. Mixed parallel implementations of the top level step of strassen and winograd matrix multiplication algorithms. In *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, page 10008.2, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 222–231, New York, NY, USA, 1999. ACM. http://doi.acm.org/10.1145/305619.305645.

[19] Keqin Li, Yi Pan, and Si Qing Zheng. Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system. *IEEE Trans. Parallel Distrib. Syst.*, 9(8):705–720, 1998. http://dx.doi.org/10.1109/71.706044.

[20] M.Ñielsen and I. Chuang. *Quantum computation and quantum information.* Cambridge University Press, New York, NY, USA, 2000.

[21] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[22] John Backus. The history of fortran i, ii, and iii. pages 25–74, 1981. `http://doi.acm.org/10.1145/800025.1198345`.

[23] Jacques Cohen. A view of the origins and development of prolog. *Commun. ACM*, 31(1):26–36, 1988. `http://doi.acm.org/10.1145/35043.35045`.

[24] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM. `http://doi.acm.org/10.1145/1238844.1238856`.

[25] David H D Warren, Luis M. Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with lisp. *SIGART Bull.*, (64):109–115, 1977.