Universidad del Bío-Bío, Chile
Facultad de Ciencias Empresariales
Departamento de Sistemas de Información

Simulación y optimización de múltiples cintas en paralelo de la máquina 'SMART' desde un patrón establecido

PROYECTO DE TÍTULO PRESENTADO POR FERNANDO ANDRÉS BASTÍAS VALDOVINOS
DE LA CARRERA INGENIERÍA CIVIL INFORMÁTICA
CON LA COLABORACIÓN DEL DR. RODRIGO TORRES AVILÉS

2023

Resumen

El presente proyecto tiene como finalidad analizar el comportamiento a largo plazo de distintas cintas provenientes de un único patrón, para analizar las propiedades de una Máquina de Turing particular llamada SMART. De esta forma, es posible identificar su comportamiento observando el conteo de las apariciones de ciertos patrones, y cuánto tiempo utiliza para alcanzarlos. Sin embargo, la máquina requiere tiempos exponenciales por cada cinta simulada, por lo que el estudio de sus propiedades transitivas se ve frenado por el factor tiempo. Llega un momento en el que es poco eficiente identificar patrones, porque la investigación tomaría demasiado tiempo.

El documento a continuación presenta un concepto conocido como **programación en paralelo**, cuyo propósito en esta investigación es poder implementar una metodología para optimizar los tiempos de ejecución de la simulación de la máquina. Para esto, se definirán los conceptos fundamentales para entender por qué se optó por tomar esta alternativa; se presentarán las configuraciones necesarias para su buen funcionamiento, y se realizará una comparativa de resultados con el propósito de demostrar la utilidad de este concepto para el estudio de las Máquinas de Turing.

El contenido del documento se divide en una introducción que presenta el problema a desarrollar, seguido de la metodología que implementará la investigación para resolverlo. Luego, se presentan los conceptos claves necesarios para tener mayor claridad de los aspectos importantes que se utilizarán en este informe. A continuación, se explicará la situación en la que se encuentran varios conceptos relacionados a la programación en paralelo hoy en día, a través del Estado del Arte.

El desarrollo del proyecto se presenta en su propio capítulo, donde se detalla cómo fue llevado a cabo, indicando los requerimientos, condiciones y configuraciones para poder ejecutarlo. Además, se presentan los resultados obtenidos con este procedimiento.

En el último capítulo se hace una reflexión de por qué esta metodología fue efectiva, y qué puede ofrecer para el estudio de las Máquinas de Turing en el futuro, tras presentar los resultados positivos que se obtuvieron a través del desarrollo de este proyecto.

NOMENCLATURA: Máquina de Turing, Programación en Paralelo, Máquina SMART, Backtracking

Abstract

The following project presents an analysis regarding the long-term behavior of multiple tapes with a unique pattern to study the properties of a specific Turing Machine called SMART. This way, it is possible to study its behavior based on the count of those patterns whenever they appear, and how long it takes for them to appear. However, this machine requires exponential times for each simulated tape, so the study of its transitive properties gets slowed down because of the time issues. There's a moment where it's not efficient to identify patterns because the investigation will take a lot of time.

This document will present a concept called parallel programming, whose purpose in this project is to implement a methodology to optimize the execution times of the simulation. Because of this, the fundamental concepts will be defined to understand why this is the chosen method. The right setup will be explained as well, and there will be a comparative chart to prove how useful this concept is for the study of Turing Machines.

An introduction will present what is the problem to solve, followed be the methodology used to solve it. Then, the key concepts will be presented so it's easier to understand the most important aspects that'll be covered in this project. Next, the current state of the studies related to these concepts are presented in the Art's state segment.

Finally, in the last chapter there'll be an argument regarding why this methodology has been effective, and how does it help regarding the studies on Turing Machines, after presenting the positive feedback obtained through this project.

Índice

Índice de Figuras

1. INTRODUCCIÓN

El estudio de los niveles de transitividad de la máquina SMART ha sido realizado en gran medida gracias a las observaciones de su comportamiento. La forma más sencilla de hacerlo a través de simulaciones que logran representar un comportamiento fiel de dicha máquina. Estas simulaciones ya existen, y logran representar bien la metodología para recoger resultados que permiten analizar su comportamiento.

Trabajando con la simulación de la máquina SMART, se puede observar que la entrega de los resultados requiere un tiempo que se vuelve exponencialmente mayor, mientras más grande es el patrón ingresado por consola. Estos resultados son importantes para poder definir algunas propiedades que son clave para definir ciertas facetas que nos demuestran el comportamiento de esta máquina.

Algunas de las propiedades que se pueden deducir a través de la experimentación son los niveles de transitividad. La obtención de estos resultados puede tardar unos cuantos segundos al ingresar un patrón de tamaño pequeño, pero fácilmente puede llegar a durar horas en entregar resultados, incluso días si se sigue intentando con el siguiente patrón más cercano que se puede simular.

El proyecto a continuación, tiene como propósito realizar una optimización en los tiempos de ejecución en la simulación del programa ya mencionado. Para eso, se llevará a cabo un análisis que determinará la viabilidad de la programación en paralelo en este simulador.

La programación en paralelo consiste en segmentar la ejecución de un aspecto específico de un programa en múltiples hilos de los procesadores disponibles. O, en otras palabras, reparte el trabajo que normalmente ocupa un solo procesador en múltiples ramas, las cuales serán repartidas y ejecutadas en simultáneo.

Este proyecto podría significar un gran avance en el estudio de las máquinas de Turing en general, si se logra demostrar que se puede ejecutar la simulación en múltiples hilos de un procesador permitiendo dividir la tarea de la simulación en múltiples procesos en paralelo, y por lo tanto disminuyendo considerablemente el tiempo necesario para obtener resultados conclusivos.

2. OBJETIVOS GENERALES Y ESPECIFICOS

2.1. Objetivo general:

- Optimizar la velocidad de ejecución de la simulación de múltiples cintas de la máquina SMART desde un patrón establecido

2.2. Objetivos específicos:

- Emular eficientemente en secuencial la máquina SMART, a través de una recreación en un programa en C.
- Analizar técnicas de ejecución paralelas.
- Implementar programación en paralelo a la simulación.
- Estudiar el tiempo ahorrado durante la simulación.
- Analizar la eficiencia del método utilizado.

3. METODOLOGÍA DE TRABAJO

El proyecto se fundamenta en gran medida de las investigaciones del profesor Rodrigo Torres, junto con los investigadores franceses Julien Cassaigne y Nicolas Ollinger, donde describen el funcionamiento de la máquina SMART y las distintas propiedades que cumple.

Este funcionamiento está puesto en prueba a través de un software escrito en C++, en el que se ingresa en consola dos enteros: el primero define el tamaño del patrón que se busca encontrar durante la ejecución de la simulación de la máquina SMART, mientras que el segundo establece el límite máximo en que el algoritmo es ejecutado. Este segundo genera un comportamiento interesante en la simulación, pues logra cumplir con su objetivo de forma rápida cuando se ingresa un número igual o menor a 7. Luego, entre 8 y 9, se comienza a notar que toma un poco más de tiempo.

Al ingresar un número igual o mayor a 10, el programa empieza a tener un tiempo de ejecución considerablemente mayor. Tras hacer múltiples mediciones de cuánto tarda el programa en ejecutarse, cada iteración tarda 10 veces más en realizarse al ir aumentando en 1 el dígito que mide el tiempo. Es decir, si al ingresar el número 7 tarda 0,04 segundos, el 8 tardaría 0,4 segundos, el 9 tardaría 4 segundos, y así sucesivamente. Toma una cantidad baja de números para lograr que el programa aumente su tiempo de ejecución considerablemente, llegando a durar horas o incluso días.

El enfoque que se le busca al programa para mejorar su eficiencia es intentar ejecutar una misma función en diferentes hilos que estén disponibles en el procesador, porque así es posible dividir el proceso y hacerlo más eficiente según el hardware a disposición.

A este concepto se le conoce como **programación en paralelo**, y para este proyecto se trabajará con la API OpenMP, el cual incluye librerías que permiten una sencilla implementación. Esto facilita la viabilidad de este concepto para este programa.

Uno de los requerimientos para poder justificar la implementación de la programación en paralelo, es averiguar si existe una función dentro del programa que forma parte del 90% del tiempo total ejecutándose, o más. Para esto, se utilizará la librería Chrono, que está incluida en C++. Lo que hace esta función es crear cronómetros dentro del programa, la cantidad que uno desee y que inicien o terminen en el momento que uno estime conveniente. En este caso, se puede determinar la duración exacta de cada función que se ejecuta al iniciar el programa, lo cual será determinante para definir los siguientes pasos a seguir.

Por último, todo este trabajo se realizará sobre el entorno de desarrollo integrado Microsoft Visual Studio 2022. Así, las configuraciones que se apliquen durante la realización del proyecto podrían ser diferentes si se trabaja bajo un entorno distinto.

4. MARCO TEORICO.

4.1. Introducción a la informática moderna:

A inicios de los años 40, las ciencias de la computación todavía eran un concepto que estaba en desarrollo temprano, pero eso no era motivo para ignorar sus cualidades. Durante esa época, el mundo estaba afectado por los eventos de la Segunda Guerra Mundial, donde se demostró el potencial que tenía la informática para ser desarrollada en el futuro. En este periodo, el poder de la información permitió que los bandos triunfadores tomaran ventaja científica y tecnológica del conflicto, gracias a las máquinas con las que lograron descifrar los mensajes codificados que se enviaban en Alemania, evitando que el enfrentamiento bélico se extendiera. [1]

Uno de los mayores contribuyentes en esta disciplina fue Alan Turing, uno de los primeros investigadores en modelar una máquina que representaría la base de lo que es la informática moderna. Es más, se podría decir que la estructura lógica que utilizan los computadores hoy en día sigue estando basada en este descubrimiento, el cual fue llamado la Máquina de Turing. Fue a partir de aquí donde el desarrollo en los campos de la compilación de códigos, procesamiento de información y algoritmos o desarrollo de autómatas empezó a tener un crecimiento más acelerado.

La máquina de Turing ha sido la invención que permitió definir si un problema es solucionable a través de una propuesta que una máquina pueda leer, procesar y automatizar. Fue de los acercamientos más importantes para definir lo que sería la inteligencia artificial moderna, y su estudio ha perdurado hasta el día de hoy.

4.2. Máquinas de Turing

La primera máquina de Turing se contextualizó bajo la idea de poder resolver problemas, traduciéndolos a través de expresiones matemáticas que después se convierten en operadores lógicos para definir valores de verdadero y falso con números binarios. Así, muchos problemas pueden ser formulados de tal manera que puedan ser automatizados por una máquina capaz de seguir algoritmos.

En 1936, Alan Turing definió la máquina de Turing como un espacio infinito de memoria marcada por cuadros en los que se pueden escribir símbolos, dispuestos en una cinta de tamaño establecido (o bien infinita). La máquina puede desplazarse a través de dichos cuadros, utilizando un cabezal que lee lo almacenado en los cuadros, además de escribir sobre ellos. Finalmente, la máquina está compuesta por un programa, que le indica al cabezal qué es lo que debe hacer cuando se encuentra con algún símbolo en particular. Este programa consta de una lista de instrucciones que indica el comportamiento que tendrá la máquina según lo que está escrito en éste.

La máquina termina su ejecución cuando el programa que le envía instrucciones no tiene más comandos que entregarle, dependiendo del estado en el que se encuentra. El resultado final es representado por la cinta completa con los símbolos guardados en sus cuadros.

Formalmente, la máquina de Turing es compuesta por los siguientes elementos:

- Un conjunto finito de estados, Q = {q1, q2, …, qn}
- Un conjunto inicial, q1
- Un alfabeto C = {s1, s2, …, sn}
- Un conjunto de funciones

El conjunto de instrucciones está definido a través de una tabla finita, en la que se especifica el comportamiento de la máquina al encontrarse con los distintos estados que puede tener la cinta. A estos comportamientos se les llama **funciones**, siendo la función principal la que dará la instrucción inicial que debe seguir la máquina. El momento en que la máquina no encuentre las condiciones para ejecutar alguna de las funciones, se detendrá. [2]

4.3. La máquina SMART:

La máquina SMART es un modelo de máquina de Turing que cumple con las siguientes propiedades:

- Posee solamente 4 estados y 3 símbolos, por lo que es pequeña (Small)
- Es un sistema que no tiene subsistemas (Minimal)
- No posee puntos periódicos (Aperiodic)
- Sus funciones de transición son biyectivas y sobreyectivas (Reversible)
- Está basada en la máquina de Turing (Turing Machine)

Este modelo surgió a raíz de múltiples investigaciones que datan del 2002, donde se descubrió una Máquina de Turing que resultaba ser aperiódica, pero no reversible. Se dice que una máquina es reversible cuando cada configuración que presenta tiene una única configuración que la precede.

En 2008 se teorizó la existencia de una Máquina que cumpliera las propiedades de reversibilidad y aperiodicidad al mismo tiempo, sentando finalmente las bases que darían forma a la primera máquina SMART en 2017, la cual tiene la particularidad de no tener condiciones de detención ni de comienzo. Además, tampoco admite el símbolo blanco en su abecedario.

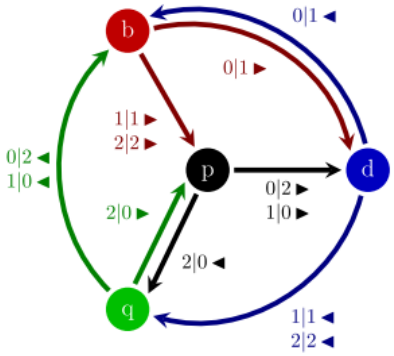

Figura 1 Comportamiento de la Máquina SMART

En la figura 1 se puede apreciar la lista de funciones que procesa la máquina SMART, mostrando los estados que puede tener la máquina, los símbolos que puede utilizar, y las rutas que sigue dependiendo del símbolo leído. [3]

4.4. Programación en paralelo

El concepto de la programación en paralelo ha sido uno de los factores más importantes en lo que a avances de poder en procesamiento respecta. La capacidad hoy en día de los procesadores de ejecutar múltiples hilos a través de múltiples núcleos, ha permitido una mayor accesibilidad para realizar tareas que años atrás habrían sido mucho más difíciles de lograr gracias a la aparición de los procesadores con múltiples núcleos que empezaron a salir al mercado desde el año 2006.

La programación en paralelo consiste en utilizar de forma eficiente los hilos disponibles de un procesador para dividir múltiples tareas disponibles en un mismo programa, para poder así acortar los tiempos de ejecución considerablemente. De esta forma es posible resolver problemas en un tiempo mucho menor, dando la posibilidad de realizar experimentos que requerían largas horas para la obtención de resultados. [4]

Este cambio en arquitectura ocurrió porque hubo una baja significativa de rendimiento en los procesadores: pasó de ir aumentando un 50% por año entre 1986 a 2002, hasta ir aumentando tan solo un 20% en los años siguientes. Así fue como los principales fabricantes de procesadores decidieron que el siguiente rumbo que deberían seguir los procesadores para mejorar su rendimiento, era a través de la ejecución en paralelo con múltiples procesadores en un mismo circuito integrado.

Esto significó un cambio muy importante en la perspectiva de los desarrolladores de software, porque agregar más núcleos a un procesador no acelerará todo por arte de magia, sino que hay que aprender a adaptarlos para que puedan ejecutarse a través del sistema de múltiples procesadores. Los programas que existían antes de este evento tecnológico simplemente desconocen la existencia de la ejecución en paralelo.

5. ESTADO DEL ARTE

5.1. Mixing topológico

Recientemente se han realizados estudios en lo que respecta al Mixing de las máquinas SMART, concepto que ha sido estudiado a través de múltiples disciplinas de la computación. En el caso de las máquinas de Turing, se han realizado estudios en sistemas relativamente sencillos, por lo que el estudio es un tema bastante reciente a través de las máquinas SMART.

El Mixing topológico en las máquinas de Turing corresponde al análisis de dos inputs cualesquiera (estado y descripción finita de parte de la cinta analizada), en el que dentro de un tiempo T, el primer input logra intersectar con el segundo input de manera persistente a través del tiempo. [5]

Se dice que una máquina (φt) para cualquier t perteneciente a los reales, definido sobre un espacio topológico X es topológicamente mixing si para cada par de sistemas abiertos U, V, existe un tiempo $T > 0$ tal que:

$$\varphi t\,(U) \cap V \neq \emptyset, \forall t > T$$

A través de esta definición, se busca estudiar la evolución de la máquina, tomando como referencia dos espacios abiertos U y V. Dependiendo del tamaño, más tiempo toma demostrar la existencia de comportamientos similares.

La máquina SMART tiene propiedades demostradas de Mixing Débil, una propiedad que está incluida dentro del Mixing Topológico. Para demostrar el segundo, es necesario analizar los patrones que se generan en la máquina SMART a través del simulador. [6]

5.2. Usos de la programación en paralelo hoy en día

La programación en paralelo es un tópico que se refuerza constantemente a la hora de optimizar varios softwares. En estos últimos años, los procesadores más económicos tienen como mínimo 2 núcleos y 4 hilos de procesamiento. Pero los procesadores más comunes tienen 4 núcleos y 8 hilos, como es el caso de los Intel i3 de décima generación en adelante, lanzados al mercado en agosto de 2019. Los procesadores Ryzen de tercera generación de AMD también poseen procesadores de 4 núcleos y 8 hilos dentro de su gama de entrada.

Esto es muy importante porque permitió a la comunidad informática poder realizar experimentos y así compartirlos con el resto, a través de plataformas como Github por ejemplo. Hoy

en día es cada vez más accesible utilizar esta tecnología, que ha demostrado ser el camino que se necesita para mejorar el rendimiento de nuestros dispositivos y programas.

Un detalle importante para tener en mente a la hora de programar en paralelo es que son fracciones del programa las que se deben ejecutar en paralelo, no la totalidad del mismo. Es decir, hay que tener claridad respecto al funcionamiento del programa para entender qué aspectos se pueden o no paralelizar. Cuando una función de un programa está activa durante un corto periodo de ejecución, intentar paralelizarlo no traerá resultados sobresalientes.

Si bien siempre existe la posibilidad de paralelizar cualquier aspecto de un programa, es importante tener claro si es efectivamente el mejor método para optimizar su tiempo de ejecución. Un método teórico que puede dar una idea bastante adecuada de cuándo vale la pena optar por la alternativa de la programación en paralelo, es a través de la ley de Amdahl. Esta ley establece una posible relación entre la cantidad de procesadores disponibles y el tiempo optimizado de la ejecución del programa, de acuerdo con el porcentaje del tiempo que toma cada función originalmente.

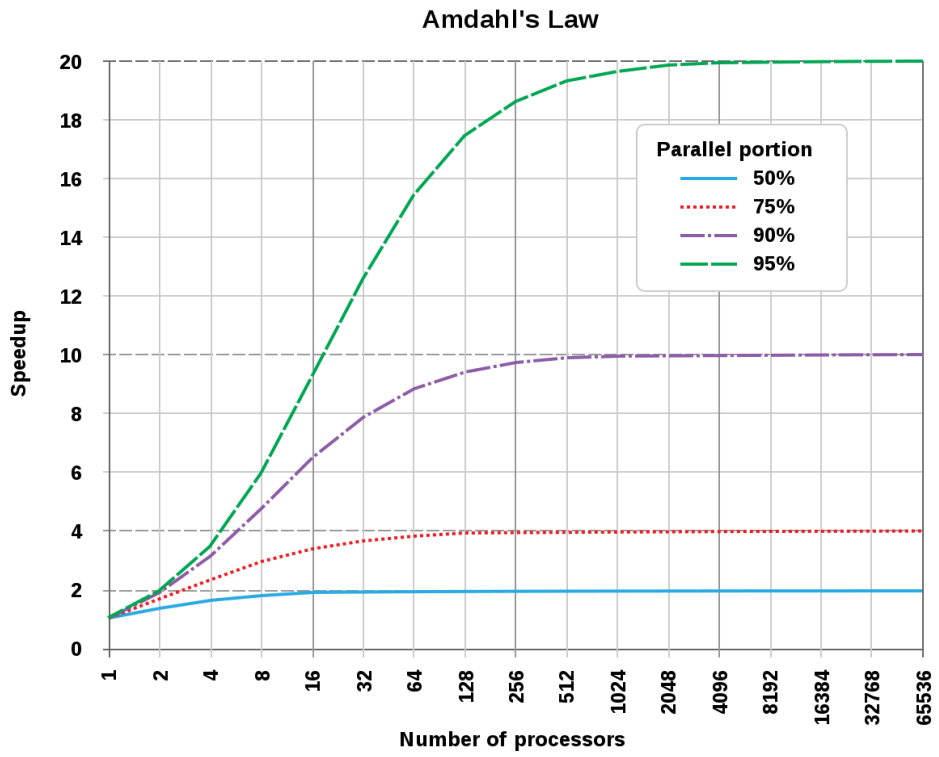

Figura 2 Incremento teórico de velocidad de ejecución según la Ley de Amdahl

Si bien la ley de Amdahl se basa en cálculos que fueron presentados en la conferencia de Spring Joint Computer en 1967, establece una observación que resulta bastante lógica: si una función toma

parte del 90% o más del tiempo de ejecución, el potencial que tiene para que la ejecución en paralelo optimice es mucho mayor respecto a si se ejecuta durante un 75% del tiempo o menor. [7]

Para lograr obtener estas métricas, existe una librería en C++ que se llama Chrono. Esta librería permite determinar el tiempo en el que un programa llega desde un punto A hasta un punto B. Es decir, es posible determinar el tiempo que toma cada función al ejecutarse dentro de un programa, y dichas medidas se pueden ir almacenando en una hoja de cálculo para realizar los correspondientes cálculos, estudios y observaciones.

Otro aspecto a considerar es que los procesadores más poderosos disponibles en el mercado con un valor menor a 600 dólares no tienen más de 24 núcleos. Si bien existen procesadores de 64 núcleos como el AMD Ryzen Threadripper Pro 5995X, este tiene un valor por sobre los 9000 dólares. Esto hace que en la práctica sea difícil que exista una comunidad muy activa de personas desarrollando para trabajar con tal cantidad de núcleos en un procesador. Observando el mismo gráfico para una cantidad menor de núcleos, éste queda de la siguiente manera:

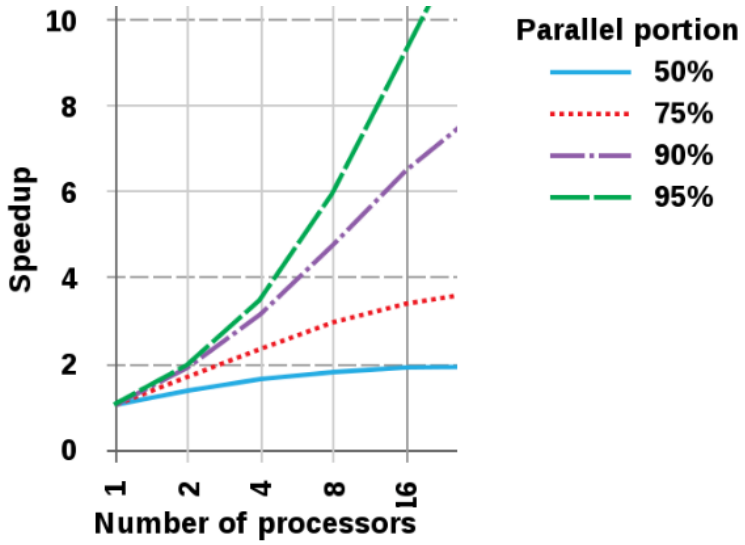

Figura 3 Incremento teórico de velocidad de ejecución según la Ley de Amdahl, para un número reducido de procesadores

Del gráfico se puede observar que incluso con una cantidad decente de procesadores, es posible obtener una mejoría de casi 4 veces respecto al tiempo de ejecución original, utilizando 4 procesadores. Como la gran mayoría de procesadores comerciales trabajan con 4 núcleos como mínimo, es un objetivo realista de alcanzar.

5.3. Simulador Máquina SMART:

El simulador con el que se realiza este proyecto es un programa escrito en C++, el cual cumple con el propósito de recrear el funcionamiento parcial de la máquina SMART. Esto es, porque el propósito del simulador no es derechamente mostrar la cinta que se genera tras un tiempo de ejecución determinado, sino que es encontrar las posiciones de la cinta donde el patrón que uno ingresa en pantalla no se encuentra.

Esto es particularmente importante, porque permite estudiar la consistencia de la aparición de estos patrones. Asimismo, muchas propiedades pueden ser descubiertas de acuerdo al análisis de los patrones encontrados a través de la simulación.

Este simulador trabaja únicamente con el patrón $\binom{0 \quad 0^n}{b}$, porque así se puede analizar el comportamiento de la máquina de forma más directa para determinar si es topológicamente mixing. Esto es posible ya que se ha demostrado que la máquina es minimal, al no poseer subsistemas que puedan intervenir en el análisis.

Al iniciar el simulador, el usuario debe ingresar dos números enteros: el primero indica el patrón que el programa busca a través de la cinta, mientras que el segundo define el tiempo que la máquina se ejecuta, definiendo el tamaño máximo que tendrá la cinta en el proceso.

Las funciones que están incluidas en el simulador son “asignación”, “computar_hasta”, “arbol_rec” y la función “main”. De estas funciones, “arbol_rec” posee un llamado recursivo a sí mismo que se ejecutará las veces que sean necesarias hasta llegar a un límite indicado por el segundo número entero que el usuario debe ingresar para empezar la simulación.

El tiempo que puede durar la ejecución de este programa puede ser de tan solo unos segundos. Pero en el momento que empieza a durar un poco más, la diferencia pasa de ser de segundos a minutos, e inmediatamente después empieza a tomar más de una hora en terminar de procesarse. Esto es una limitante considerable a la hora de realizar análisis con números más grandes, por lo que el propósito del proyecto es averiguar si existe la posibilidad de aplicar programación en paralelo en algún punto del simulador para realizar una optimización en los tiempos de ejecución.

6. DESARROLLO:

6.1. Pruebas iniciales

Para el desarrollo de este proyecto, la simulación de la máquina se realiza a través de un programa en C++ que es compilado a través del entorno de trabajo Visual Studio 2022. Este viene con varias herramientas fáciles de descargar e implementar.

Más adelante se indicarán algunas configuraciones que hay que aplicar para asegurar que todo esté funcionando como debería, pues la librería utilizada para hacer funcionar el proceso no implementa de forma automática los recursos necesarios para llevar el proceso a cabo.

Tras descargar el kit de desarrollo para aplicaciones de escritorio en C++, hay que abrir el archivo que contiene la simulación llamada SmartRW.cpp. Con el propósito de tener una mayor facilidad de análisis, se incluye la librería Chrono que viene incluida en C++, la cual permite cronometrar el tiempo que tarda el programa en ejecutarse.

No solamente muestra el resultado de la ejecución del programa en su totalidad, sino que también da la posibilidad de calcular la duración de las funciones que vienen incluidas de forma separada.

Tal como es mencionado en el apartado 5.3 del documento, el simulador recibe dos números enteros: uno para determinar el valor a encontrar, y el otro para determinar la duración que tomará la simulación.

La función "Arbol_rec" llama particularmente la atención porque posee propiedades recursivas dentro de la misma, por lo que para la siguiente tabla se optó por darle prioridad a la medición de dicha función a través del tiempo.

Valor Tiemp	Arbol_rec	Otras Funciones	Main	Porcentaje arbol_rec
5	1696	3335729	3337425	0,05
6	8837	3750624	3759461	0,24
7	58783	2441446	2500229	2,35
8	489019	3243898	3732917	13,10
9	4436210	7558151	11994361	36,99
10	43788820	2964037	46752857	93,66
11	437112525	4110901	441223426	99,07
12	4694185972	12178408	4706364380	99,74

Figura 4 Tiempo de ejecución de las funciones del simulador para distintos tiempos n, en microsegundos

Para tener una comparación equitativa, se utilizó siempre el valor "1" a través de distintos tiempos de ejecución, los cuales están descritos en la columna "Valor Tiempo" de la tabla recién adjunta. La medición presentada en la tabla está en microsegundos.

6.2. Observaciones respecto a tiempos de ejecución

De acuerdo a la figura 4, la duración total de la simulación depende casi completamente de la duración arbol_rec, ya que la duración de las demás funciones no va más allá del orden de los $\mathbf{10^1}$ **segundos**, mientras que arbol_rec tiene una duración que permanece en el orden de los $\mathbf{10^{n-8}}$ **segundos**, siendo n el entero que define la duración de la ejecución que tendrá la simulación.

Según la información registrada en la tabla, en el tiempo n = 8 la simulación empieza a tener una duración de ejecución más perceptible, la cual empieza a ser considerablemente mayor a partir de este punto. Es más, desde n = 11 hay que esperar cerca de un minuto para ver los resultados, y a partir de n = 12 la simulación tarda más de una hora.

A partir de los resultados observados, se estima que la duración de n = 13 habría sido de casi 12 horas, mientras que con n = 14 habría tardado más de 5 días.

El siguiente gráfico muestra la participación de la función arbol_rec en el tiempo de ejecución de la simulación respecto al resto del programa. La sección azul representa el tiempo en el cual dicha función estuvo activa, mientras que la sección verde es la suma del tiempo de ejecución de las funciones Asignación, Computar_hasta y el resto de la función main.

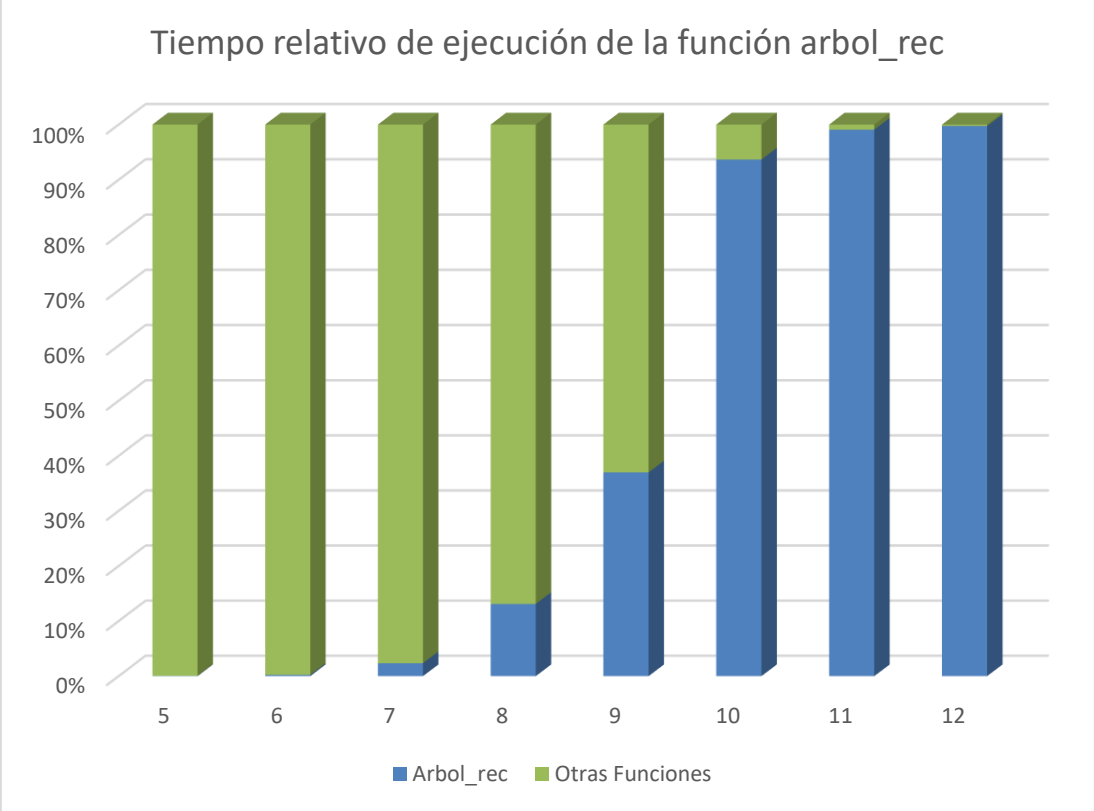

Figura 5: Gráfico de Barra representando el tiempo relativo de ejecución de la función arbol_rec

6.3. ¿Por qué programar en paralelo?

Como se observa en el gráfico anterior, la función arbol_rec empieza a consumir más del 95% del tiempo total de ejecución desde el tiempo n = 11 o superior, por lo que la ley de Amdahl establece que dicha función podría ser una candidata muy fuerte para comprobar si existen técnicas de paralelización que pudieran optimizar su rendimiento. De poder demostrar un método de paralelización, siempre existirá una posibilidad de seguir mejorando su tiempo de ejecución con el paso del tiempo.

El método de paralelización implementado ocupa la API OpenMP, la cual permite una implementación más sencilla y no requiere realizar cambios sustanciales en el código fuente del simulador, facilitando así su uso.

La forma que trabaja OpenMP consiste en ejecutar en simultáneo múltiples instancias de una función que realiza algún tipo de trabajo, y ejecuta todos los subprocesos que pueda realizar físicamente hablando al mismo tiempo.

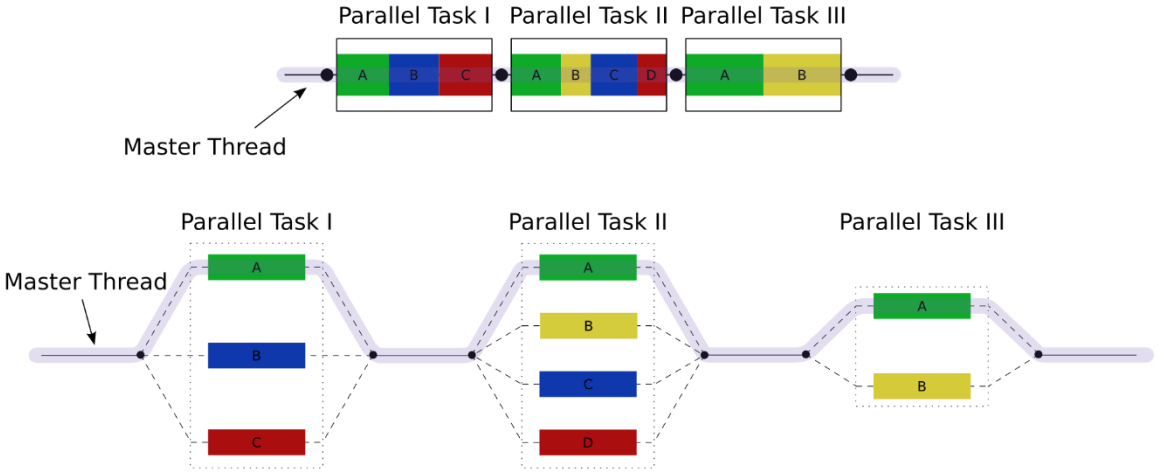

Figura 6: Representación gráfica del funcionamiento de la programación en paralelo

En la figura anterior, se puede apreciar cómo 3 distintos procesos que presentan en total 9 subprocesos se ejecutan de forma tal que no haya que llamar a cada uno de dichos subprocesos por separado. Estos son agrupados y ejecutados al mismo tiempo dentro de sus propios grupos cuando son llamados por el proceso principal. [8]

Otra razón para utilizar OpenMP es que incluye una función para poder paralelizar bucles For de manera automatizada. La función es Arbol_rec, e incluye un bucle For con una llamada recursiva. Así, con la instrucción "**#pragma omp for**" se puede paralelizar dicho bucle sin tener que alterar el código fuente.

Para tener acceso a las funciones de OpenMP, hay que incluir la librería <opm.h> al inicio del código fuente.

6.4. Configuraciones finales

Un problema que puede ocurrir a la hora de realizar las pruebas es que sin importar qué funciones intenta utilizar OpenMP, el programa puede llegar a arrojar los mismos resultados. Normalmente suele ocurrir que al implementar librerías externas a un código fuente, el compilador envía advertencias o emite errores de compilación que impiden su buen funcionamiento.

Una particularidad de OpenMP, es que está escrito de forma tal que el compilador ignora sus líneas de comando en caso de que no pueda interpretar las instrucciones. En el caso de Visual Studio 2022, las configuraciones por defecto no permiten ejecutar las instrucciones de OpenMP, por lo que antes de realizar las pruebas hay que verificar ciertas configuraciones.

Primero, hay que asegurarse que se realizó correctamente la instalación del kit de desarrollo para el escritorio con C++, junto con las Herramientas de Clang en C++ para Windows. Si ya se tienen otras herramientas instaladas en Visual Studio, el menú de instalación de herramientas adicionales está en la pestaña "Tools", en la opción "Get Tools and Features".

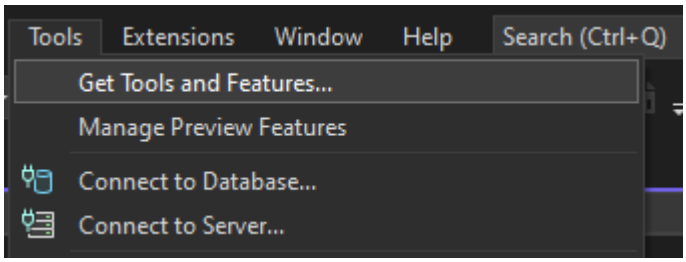

Figura 7: Cómo llegar a la ventana de instalación de componentes de Visual Studio

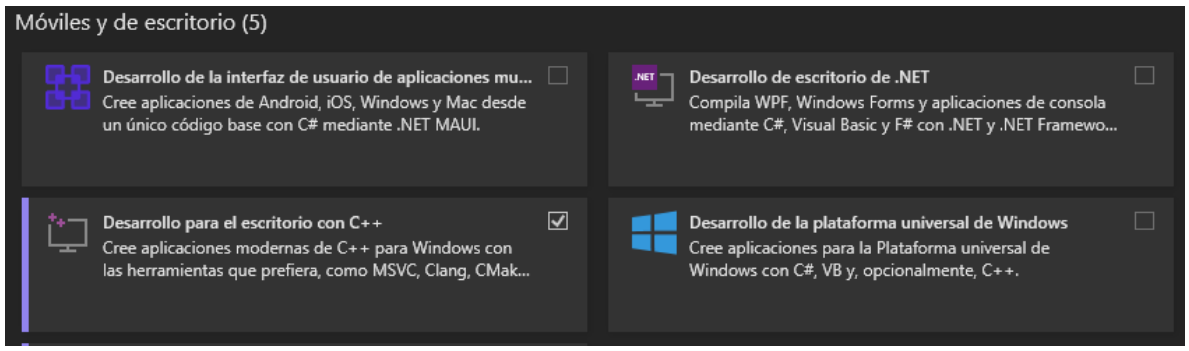

Figura 8: Ventana de instalación de componentes de Visual Studio

Figura 9: Componentes específicos a ser instalados

Luego, hay que entrar en las propiedades del proyecto, en la pestaña "Project".

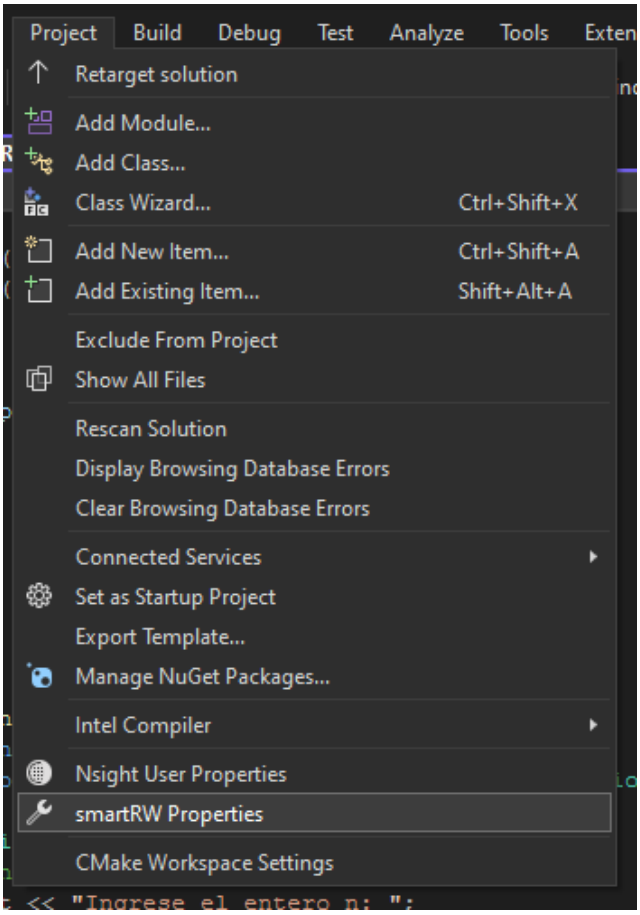

Figura 10: Ubicación de las propiedades del proyecto

Acá, en la pestaña "General" está la opción para cambiar el Toolset de la plataforma. Es aquí donde tenemos que cambiar a "LLVM (clang-cl)", originalmente dice "Visual Studio 2022 (v143)".

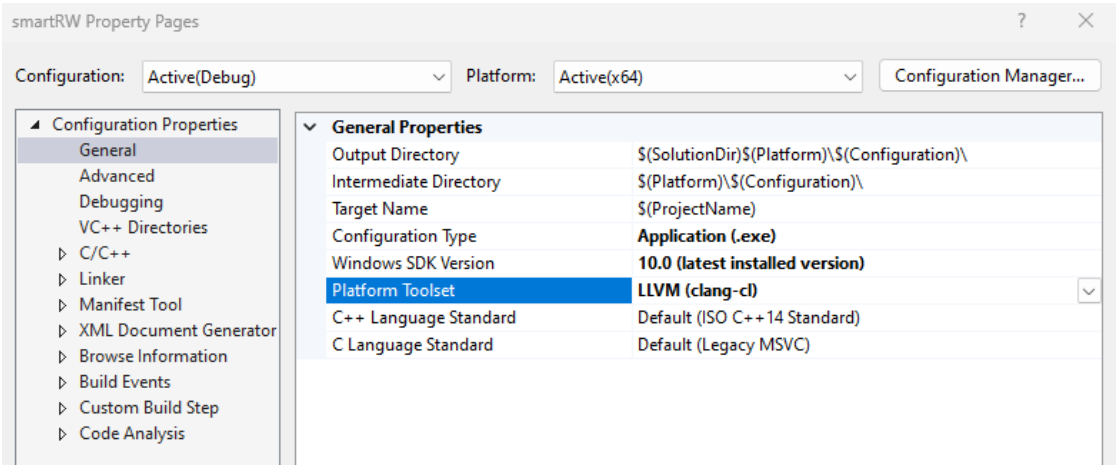

Figura 11: Menú para cambiar el Toolset de la plataforma, indispensable para trabajar con OpenMP

Finalmente, basta con incluir el comando #pragma omp parallel for antes del bucle for que se encuentra en "arbol_rec".

```
void arbol_rec(string cinta, int cont) {
    int gg;
    string temp, temp2;
    gg = computar_hasta(&cinta, &cont);

    if (gg == -1) return;
    if (gg == -2) {
        cout << "ERROR" << endl;
        exit(_Code: 0);
    }
    temp2 = cinta;

#pragma omp parallel for
    for (int i = 0; i < 3; i++) {
        temp = temp2;
        temp[gg] = i + 48;
        arbol_rec(cinta: temp, cont);
    }
}
```

Figura 12: Implementación de OpenMP en el código fuente

Con esto, el programa implementa de forma exitosa una paralelización a un bucle for, dando resultados prometedores.

6.5. Comparación de resultados

Una vez las configuraciones están hechas, estos son los resultados que arroja el simulador con los mismos números ingresados en la figura x.

Tiempo n	Arbol_rec	Arbol_rec	Porcentaje Mejora
5	1696	1421	16,2 %
6	8837	5119	42,1 %
7	58783	38613	34,3 %
8	489019	313017	36,0 %
9	4436210	2871616	35,3 %
10	43788820	27626467	36,9 %
11	437112525	279507853	36,1 %
12	4694185972	3211560780	31,6 %

Figura 13: Comparación de resultados antes y después de paralelizar el código

En el siguiente gráfico se puede apreciar la disminución del tiempo de ejecución para cada tiempo n, facilitando su visualización:

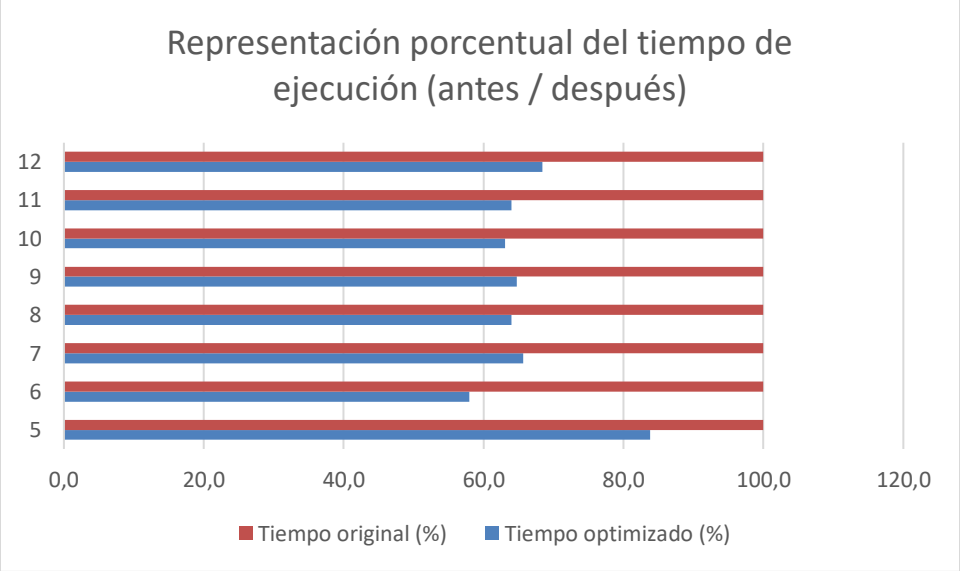

Figura 14: Representación en gráfico de barra del tiempo de ejecución de ambos métodos

Como se puede observar, con esta implementación de OpenMP se puede apreciar una mejora aproximada de un 35% en tiempos de ejecución, una disminución bastante considerable a la hora de obtener resultados.

Este porcentaje se puede traducir en una disminución de 34 minutos para el tiempo n = 12, y una disminución teórica de 4 horas para el tiempo n = 13.

7. CONCLUSIÓN:

A través del desarrollo del proyecto, se demuestra exitosamente la posibilidad de obtener mejores tiempos de ejecución de la simulación de la máquina SMART, al paralelizar un segmento que consume gran parte del tiempo total de ejecución, el cual corresponde a un algoritmo de búsqueda por backtracking.

Una de las razones por la que la programación en paralelo es útil en escenarios de algoritmos de búsqueda, es que la función a paralelizar no trabaja con variables que cambien sus valores e inmediatamente después vuelvan a ser utilizados. Es por esto que la función puede ejecutarse en paralelo, haciendo que cada llamada trabaje de forma independiente y sin afectar al resto de los hilos.

La implementación de OpenMP en este proyecto resultó ser relativamente sencilla con las configuraciones correctas. Como su implementación se basó principalmente en principios teóricos, existe una alta gama de posibilidades que se pueden implementar en el estudio de las máquinas de Turing con la programación en paralelo.

7.1. Trabajo Futuro:

Este proyecto sienta las bases para desarrollar técnicas de ejecución en paralelo para el estudio de las Máquinas de Turing.

La demostración de resultados positivos con la configuración presentada permite cambiar el enfoque con el cual se desarrollan simulaciones de Máquinas de Turing, abriendo paso a métodos que se vean beneficiados con el paralelismo.

Como la tecnología sigue progresando hacia un entorno de procesadores con múltiples núcleos de procesamiento, adaptar el enfoque de investigación hacia la optimización del entorno de trabajo a través de la programación en paralelo va a ser la norma dentro de los próximos años, y permitirán importantes avances científicos en el futuro.

8. ANEXO

8.1. Código fuente del compilador

1. Librerías y variables globales

```
#include <iostream>
#include <cmath>
#include <iterator>
#include <set>
#include <chrono>
#include<omp.h>

using namespace std;

int NN, n, m, liminf, limsup;

set<int, less<int> > s1;

int TR[27];
string PRin[27], PRout[27];
```

2. Función 'asignacion'

```
17  void asignacion(string lag) {
18      TR[0] = 2 * pow(_Left: 3, _Right: n + 1) - 1;
19      PRin[0] = lag + "00";
20      PRout[0] = "1" + lag + "1";
21      TR[1] = pow(_Left: 3, _Right: n + 2);
22      PRin[1] = "1" + lag + "10";
23      PRout[1] = "10" + lag + "1";
24      TR[2] = pow(_Left: 3, _Right: n + 2);
25      PRin[2] = "2" + lag + "10";
26      PRout[2] = "20" + lag + "1";
27      TR[3] = pow(_Left: 3, _Right: n + 2);
```

```
28      PRin[3] = "0" + lag + "01";
29      PRout[3] = lag + "001";
30      TR[4] = pow(_Left: 3, _Right: n + 2);
31      PRin[4] = "0" + lag + "02";
32      PRout[4] = lag + "002";
33      TR[5] = pow(_Left: 3, _Right: n + 1);
34      PRin[5] = "0" + lag + "1";
35      PRout[5] = lag + "01";
36      TR[6] = pow(_Left: 3, _Right: n + 1);
37      PRin[6] = "0" + lag + "2";
38      PRout[6] = lag + "02";
39      TR[7] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1);
40      PRin[7] = "1" + lag + "010";
41      PRout[7] = "100" + lag + "1";
42      TR[8] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1);
43      PRin[8] = "2" + lag + "010";
44      PRout[8] = "200" + lag + "1";
45      TR[9] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1);
46      PRin[9] = "1" + lag + "02";
47      PRout[9] = "1" + lag + "20";
48      TR[10] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1);
49      PRin[10] = "2" + lag + "02";
50      PRout[10] = "2" + lag + "20";
51      TR[11] = pow(_Left: 3, _Right: n + 2) + 1;
52      PRin[11] = "1" + lag + "11";
53      PRout[11] = "1" + lag + "21";
54      TR[12] = pow(_Left: 3, _Right: n + 2) + 1;
55      PRin[12] = "2" + lag + "11";
56      PRout[12] = "2" + lag + "21";
57      TR[13] = pow(_Left: 3, _Right: n + 2) + 1;
58      PRin[13] = "1" + lag + "12";
59      PRout[13] = "1" + lag + "22";
60      TR[14] = pow(_Left: 3, _Right: n + 2) + 1;
61      PRin[14] = "2" + lag + "12";
```

```
62      PRout[14] = "2" + lag + "22";
63      TR[15] = 2 * pow(_Left: 3, _Right: n + 2);
64      PRin[15] = "11" + lag + "2";
65      PRout[15] = "12" + lag + "1";
66      TR[16] = 2 * pow(_Left: 3, _Right: n + 2);
67      PRin[16] = "21" + lag + "2";
68      PRout[16] = "22" + lag + "1";
69      TR[17] = pow(_Left: 3, _Right: n + 3) - 1;
70      PRin[17] = "2" + lag + "210";
```

```
71      PRout[17] = "000" + lag + "1";
72      TR[18] = pow(_Left: 3, _Right: n + 3) - 1;
73      PRin[18] = "2" + lag + "22";
74      PRout[18] = "0" + lag + "20";
75      TR[19] = pow(_Left: 3, _Right: n + 2) + pow(_Left: 3, _Right: n + 1);
76      PRin[19] = "01" + lag + "2";
77      PRout[19] = lag + "000";
78      TR[20] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1) + 1;
79      PRin[20] = "1" + lag + "011";
80      PRout[20] = "10" + lag + "21";
81      TR[21] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1) + 1;
82      PRin[21] = "1" + lag + "012";
83      PRout[21] = "10" + lag + "22";
84      TR[22] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1) + 1;
85      PRin[22] = "2" + lag + "011";
86      PRout[22] = "20" + lag + "21";
87      TR[23] = pow(_Left: 3, _Right: n + 3) - pow(_Left: 3, _Right: n + 1) + 1;
88      PRin[23] = "2" + lag + "012";
89      PRout[23] = "20" + lag + "22";
90      TR[24] = 2 * pow(_Left: 3, _Right: n + 2) - 1;
91      PRin[24] = "2" + lag + "20";
92      PRout[24] = "02" + lag + "1";
93      TR[25] = pow(_Left: 3, _Right: n + 3);
94      PRin[25] = "2" + lag + "211";
95      PRout[25] = "00" + lag + "21";
96      TR[26] = pow(_Left: 3, _Right: n + 3);
97      PRin[26] = "2" + lag + "212";
98      PRout[26] = "00" + lag + "22";
99  }
```

3. Función 'computar_hasta'

```
101  int computar_hasta(string* cinta, int* cont) {
102      int ff;
103      while (*cont < limsup) {
104          if (*cont >= liminf) s1.insert(_Val: *cont);
105
106          //cout << *cont << *cinta << endl;
107          //getchar();
108
109          for (int i = 0; i < 27; i++) {
110              ff = cinta->find(_Right: PRin[i]);
111              if (ff != string::npos) {
112                  *cont = *cont + TR[i];
113                  cinta->replace(_Off: ff, _Nx: PRout[i].length(), _Right: PRout[i]);
114                  break;
115              }
116          }
117
118          if (ff == string::npos) {
119              ff = cinta->find(_Ptr: "b");
120              if (cinta->at(_Off: ff + n + 2) == '_') return ff + n + 2;
121              else if (cinta->at(_Off: ff - 1) == '_') return ff - 1;
122              else if (cinta->at(_Off: ff + n + 3) == '_') return ff + n + 3;
123              else if (cinta->at(_Off: ff - 2) == '_') return ff - 2;
124              else if (cinta->at(_Off: ff + n + 4) == '_') return ff + n + 4;
125              else return -1;
126          }
127      }
128      if (*cont >= limsup) return -1;
129      else return -2;
130  }
131
```

4. Función 'arbol_rec'

```
132  void arbol_rec(string cinta, int cont) {
133      int gg;
134      string temp, temp2;
135      gg = computar_hasta(&cinta, &cont);
136
137      if (gg == -1) return;
138      if (gg == -2) {
139          cout << "ERROR" << endl;
140          exit(_Code: 0);
141      }
142      temp2 = cinta;
143
144  #pragma omp parallel for
145      for (int i = 0; i < 3; i++) {
146          temp = temp2;
147          temp[gg] = i + 48;
148          arbol_rec(cinta: temp, cont);
149      }
150  }
```

5. Función main

```
int main() {
    using namespace std::chrono;
    auto start_main = high_resolution_clock::now();

    string cinta;
    //int mm;
    cout << "Ingrese el entero n: ";
    cin >> n;
    //n=3;
    //cin >> mm;
    cout << "Ingrese el entero m: ";
    cin >> m;
    //NN = n+1+2*m+3;
    NN = 100;
    //liminf = pow(3,n+3)*(pow(3,mm*2)-1)/4-mm;
    liminf = 1;
    //limsup = pow(3,n+3)*(pow(3,m*2)-1)/4-m;
    limsup = pow(_Left: 3, _Right: m);

    string LAG;
    LAG = "b0";
    for (int i = 0; i < n; i++) LAG = LAG + "0";

    auto start_asignacion = high_resolution_clock::now();
    asignacion(LAG);
    auto stop_asignacion = high_resolution_clock::now();
    auto duration_asignacion = duration_cast<microseconds>(_Dur: stop_asignacion - start_asignacion);
```

```
    int cont = 0;
    cinta = "_";
    for (int i = 0; i < 2 * NN + 2; i++) cinta = cinta + "_";
    for (int i = NN - n / 2 + 1; i < NN - n / 2 + n + 2; i++) cinta[i] = '0';
    cinta[NN - n / 2] = 'b';


    auto start_arbol_rec = high_resolution_clock::now();

    arbol_rec(cinta, cont);

    auto stop_arbol_rec = high_resolution_clock::now();
    auto duration_arbol_rec = duration_cast<microseconds>(_Dur: stop_arbol_rec - start_arbol_rec);

    int sum = 0, div = 0, liminf = 0, lagoon = 0, maxl = 0, windo;
    set<int, less<int> >::iterator itr;
```

```
    set<int, less<int> >::iterator itr;
    for (itr = s1.begin(); itr != s1.end(); itr++) {
        if (maxl < lagoon) {
            maxl = lagoon;
            windo = liminf - lagoon - 1;
        }
        lagoon = 0;
        div = 0;
        while (liminf != *itr) {
            if (liminf > pow(_Left: 3, _Right: m - 1))lagoon++;
            if (div == 0) cout << endl;
            div = 1;
            cout << liminf++ << " ";
            sum++;
        }
        liminf++;
    }
```

```
212     if (liminf < limsup) cout << liminf << "+" << endl;
213     cout << endl;
214     cout << maxl << "--" << windo << endl;
215     cout << sum << endl;
216     cout << s1.size() << endl;
217
218     auto std::chrono::...k::time_point stop_main = high_resolution_clock::now();
219     auto std::chrono::microseconds duration_main = duration_cast<microseconds>(_Dur: stop_main - start_main);
220     cout << "Time taken by arbol rec: "
221         << duration_arbol_rec.count() << " microseconds" << endl;
222 }
```

9. BIBLIOGRAFÍA

[1] Lagercrantz, D., Frías, C. M., & Lexell, M. (2016). El enigma Turing (Áncora & Delfín) (Spanish Edition). Ediciones Destino.

[2] Turing Machines (Stanford Encyclopedia of Philosophy). (2018, September 24). http://seop.illc.uva.nl/entries/turing-machine/

[3] Cassaigne, J., Ollinger, N., & Torres-Avilés, R. (2017). A small minimal aperiodic reversible Turing machine. Journal of Computer and System Sciences, 84, 288–301. https://doi.org/10.1016/j.jcss.2016.10.004

[4] Pacheco, P., & Malensek, M. (2020). An Introduction to Parallel Programming (2nd ed.). Morgan Kaufmann.

[5] Vanessa L. Matus de la Parra Rodríguez (2017). Mixing topológico del flujo geodésico en superficies hiperbólicas de volumen finito. P.5. https://people.math.rochester.edu/grads/vmatusde/thesis.pdf

[6] Torres-Avilés, R. (2022). Topological mixing notions on Turing machine dynamical systems. Information and Computation, 285, 104915. https://doi.org/10.1016/j.ic.2022.104915

[7] Hill, M. D., & Marty, M. R. (2008). Amdahl's Law in the Multicore Era. Computer, 41(7), 33–38. https://doi.org/10.1109/mc.2008.209

[8] Rogers, M., Ghafoor, S., & Boshart M. (2008). How to Design a Parallel Program. https://www.csc.tntech.edu/pdcincs/resources/resources/How%20to%20Design%20a%20Parallel%20Program.pdf