Computación Paralela y Distribuida

Proyecto

Parallel Regular Expression Matching

Integrantes:

Alejandro Sebastián Chipayo Flores
Adrian Sandoval Huamaní
Julisa Lapa Romero

Docente:

Jose Antonio Fiestas Iquira

2024-1

Índice

1. Introducción

La coincidencia de secuencias de ADN, la detección de intrusiones en redes y la extracción de información de documentos web requieren autómatas finitos (FA) y coincidencia de expresiones regulares (REM). El número de estados del autómata y el tamaño de la entrada aumentan la complejidad computacional de estas tareas. Los sistemas multicore brindan nuevas oportunidades para mejorar el rendimiento de REM debido al estancamiento de las velocidades de los procesadores. [1]

En esta investigación, presentamos un algoritmo innovador para la coincidencia de expresiones regulares paralelas (PaREM). Este algoritmo está diseñado para escalar de manera efectiva con diferentes tamaños de problemas y números de hilos. PaREM mejora las matrices de adyacencia dispersas y densas al optimizar la especulación sobre los estados iniciales de subentradas distribuidas entre unidades de procesamiento. Además, creamos la herramienta PaREM, que crea automáticamente código paralelo (C++ y OpenMP) a partir de expresión regular o FA, lo que facilita su ejecución en sistemas de memoria compartida.

2. Implementación

2.1. Implementación del automata:

Implementamos dos automatas que aceptan dos expresiones regulares. Estas expresiones regulares utilizadas son 1^*00^*1 y $(0|1)?(0^*)(1+)0$, que utilizan 3 y 5 estados respectivamente.

Código del primer automata:

```
#include <vector>
#include <unordered_map>

int main() {
    int main() {
    // Fase 1: Generacion del automata determinista
    const int numState = 3;
    const int initialState = 0;

    std::vector<bool> F(numState, false);
    F[2] = true;

    std::vector<std::unordered_map<char,int>> Tt;
    Tt.resize(numState);
    Tt[0]['0'] = 1;
    Tt[0]['1'] = 0;
    Tt[1]['0'] = 1;
    Tt[1]['1'] = 2;
    Tt[2]['0'] = 1;
    Tt[2]['1'] = 0;

    return 0;
}
```

Código del segundo automata:

```
#include <vector>
#include <unordered_map>

int main() {
    int main() {

    // Fase 1: Generacion del automata determinista
    const int numState = 5;
    int initialState = 0;
```

```
    std::vector<bool> F(numState, false);
    F[3] = true;

    std::vector<std::unordered_map<char,int>> Tt;
    Tt.resize(numState);
    Tt[0]['0'] = 1;
    Tt[0]['1'] = 2;
    Tt[1]['0'] = 1;
    Tt[1]['1'] = 2;
    Tt[2]['0'] = 3;
    Tt[2]['1'] = 2;
    Tt[3]['0'] = 4;
    Tt[3]['1'] = 4;
    Tt[4]['0'] = 1;
    Tt[4]['1'] = 2;

    return 0;
}
```

2.2. Avance 1

Como primer avance, se decidió implementar un algoritmo secuencial que pueda recorrer los automatas implementadas. De está manera, tendremos un algoritmo que pueda resolver el problema planteado. De esta manera, podremos dar inicio a la paralelización del algoritmo sin preocuparse por la funcionalidad. La complejidad del algoritmo planteado es de $O(n)$. El resultado tiene sentido debido a que solo recorre el automatas en función del tamaño de la entrada binaria.

```
int main() {
    // Fase 1: Generacion del automata determinista
    // Se necesita modificar la clase AFD que utilize la lista de variables
    int initialState = 0;
    std::set<int> finalState;
    std::vector<std::map<char,int>> transitionList;
    // Metodo de inserci n de estados y transicciones
    transitionList.resize(3);
    finalState.insert(2);
    transitionList[0]['0'] = 1;
    transitionList[0]['1'] = 0;
    transitionList[1]['0'] = 1;
    transitionList[1]['1'] = 2;
    transitionList[2]['0'] = 1;
    transitionList[2]['1'] = 0;

    // Fase 2: Generacion del input
    const int length = 200;
    std::string binaryString;
    binaryString.reserve(length);
    std::srand(std::time(0));
    for (int i = 0; i < length; ++i) {
        char bit = (rand() % 2) ? '1' : '0';
        binaryString += bit;
    }

    std::cout << "Entrada a evaluar: " << binaryString << std::endl;

    // Fase 3: Recorrido del input dentro del automata
    int found = 0;
    int currentState = initialState;
    std::string way = std::to_string(currentState);
    if(finalState.find(currentState) != finalState.end())
        found++;
    for (int i = 0; i < binaryString.size(); ++i) {
        char bit = binaryString[i];
        if(transitionList[currentState].find(bit) != transitionList[
    currentState].end()) {
            currentState = transitionList[currentState][bit];
```

```
                way = way + " - " + std::to_string(currentState);
                if(finalState.find(currentState) != finalState.end())
                    found++;
            } else {
                way = way + " - X";
                break;
            }
        }
        std::cout << "Numero de matches: " << found << std::endl;
        std::cout << "Recorrido en el automata:\n" << way << std::endl;
        return 0;
    };
```

2.3. Avance 2

Para paralelizar el algoritmo, se aplicó la librería OMP para dividir el input en entradas de menor tamaño y, de este modo, cada thread se encargue de recorrer una parte del input. Sin embargo, un nuevo problema que aparece al paralelizar es que no se conoce el estado inicial en los hilos distintos al maestro. Para ello, fue necesario Determinar el conjunto de posibles estados iniciales para cada hilo. De está manera, cada hilo recorre el automata a partir de cada estado inicial obtenido.

La complejidad de tiempo de ejecución para este algoritmo sería $O(s * Lg(s) * (n/p + p) + s * Lg(s))$, donde n es el número de caracteres del input, p es el número de hilos y s, el número de estados del automata. La aparación de s se debe a la necesidad de iterar sobre el conjunto de estados del automata para determinar los posibles estados inicialmente de cada hilo. Posteriormente, se itera sobre cada estado inicial para recorrer el automata con la parte del input asignado. Sin embargo, la aparición de $Lg(s)$ se debe por el sets para almacenar los estados finales y posibles estados iniciales. Los sets utilizan árboles binarias como estructura de datos, lo que trae como consecuencia que la complejidad de inserción y búsqueda de valores sea logarítmica.

```
int main() {
    // Fase 1: Generaci n del automata determinista
    // Se necesita modificar la clase AFD que utilize la lista de variables
    int initialState = 0;
    std::set<int> finalState;
    std::vector<std::map<char,int>> transitionList;
    // M todo de inserci n de estados y transicciones
    transitionList.resize(3);
    finalState.insert(2);
    transitionList[0]['0'] = 1;
    transitionList[0]['1'] = 0;
    transitionList[1]['0'] = 1;
    transitionList[1]['1'] = 2;
    transitionList[2]['0'] = 1;
    transitionList[2]['1'] = 0;

    // Fase 2: Generaci n del input
    const int length = 200;
    std::string binaryString;
    binaryString.reserve(length);
    std::srand(std::time(0));
    for (int i = 0; i < length; ++i) {
        char bit = (rand() % 2) ? '1' : '0';
        binaryString += bit;
    }
    binaryString = "10101010101111000000111111111111000000";
    // std::cout << "Entrada a evaluar: " << binaryString << std::endl;

    // Fase 3: Recorrido del input dentro del automata
    // Aqu  se debe paralelizar el proceso con OMP
    int total_found = 0;
```

```
        const int num_threads = 4;
        int chunk_size = binaryString.size() / num_threads;

        std::vector<std::string> ways(num_threads);

        #pragma omp parallel num_threads(num_threads) reduction(+:total_found)
        {
            int thread_id = omp_get_thread_num();
            int start = thread_id * chunk_size;
            int end = (thread_id == num_threads - 1) ? binaryString.size() :
    start + chunk_size;

            int currentState = initialState;
            int found = 0;
            std::string way;

            for (int i = start; i < end; ++i) {
                int currentState = initialState;
                way = std::to_string(currentState);

                for (int j = i; j < binaryString.size(); ++j) {
                    char bit = binaryString[j];
                    if (transitionList[currentState].find(bit) != transitionList[
    currentState].end()) {
                        currentState = transitionList[currentState][bit];
                        way += " - " + std::to_string(currentState);
                        if (finalState.find(currentState) != finalState.end()) {
                            found++;
                        }
                    } else {
                        way += " - X";
                        break;
                    }
                }
        }

            total_found += found;
            ways[thread_id] = way;

        }

        std::cout << "N mero total de matches: " << total_found << std::endl;

        for (int i = 0; i < num_threads; ++i)
        {
            std::cout << "Hilo " << i << " recorrido en el automata:\n"
                      << ways[i] << std::endl;
        }

        return 0;

        // std::cout << "N mero de matches: " << found << std::endl;
        // std::cout << "Recorrido en el automata:\n" << way << std::endl;
        // return 0;
    };

    // - Output: Set of reachable states
```

2.4. Avance 3

Se tomo la decisión de reemplazar los sets y maps por unordered_map, de manera que se se utilizen tablas de hash en lugar de árboles binarios. Las tablas de hash tienen una complejidad constante para insertar y buscar datos de manera general. Por tanto, la complejidad logarítmica que estaba en el algoritmo anterior no está presente ahora. La complejidad sería ahora $O(s * (n/p + p) + s$, dando a entender que s todavía influye en el tiempo de ejecución.

```
    int main() {
        // Fase 1: Generacion del automata determinista
```

```
        // Se necesita modificar la clase AFD que utilize la lista de variables
        const int numState = 5;
        int initialState = 0;
        std::vector<bool> F(numState, false);
        F[3] = true;
        std::vector<std::unordered_map<char,int>> Tt;
        Tt.resize(numState);
        Tt[0]['0'] = 1;
        Tt[0]['1'] = 2;
        Tt[1]['0'] = 1;
        Tt[1]['1'] = 2;
        Tt[2]['0'] = 3;
        Tt[2]['1'] = 2;
        Tt[3]['0'] = 4;
        Tt[3]['1'] = 4;
        Tt[4]['0'] = 1;
        Tt[4]['1'] = 2;

        // Fase 2: Generacion del input
        const int length = 1000000;
        std::string binaryString;
        binaryString.reserve(length);
        std::srand(std::time(0));
        for (int i = 0; i < length; ++i) {
            char bit = (rand() % 2) ? '1' : '0';
            binaryString += bit;
        }
         std::cout << "Numero de caracteres: " << binaryString.size() << std::
       endl;

        // Fase 3: Recorrido del input dentro del automata
        // Aqu  se debe paralelizar el proceso con OMP
        int total_found = 0;
        const int num_threads = 8;
        int start, end, found = 0, currentState;
        std::unordered_map<int,std::vector<int>> route[num_threads];
        route[0][initialState] = std::vector<int>();
        #pragma omp parallel num_threads(num_threads) private(start, end, found,
       currentState)
        {
            int id = omp_get_thread_num();
            int chunk_size = binaryString.size() / num_threads;
            int start = chunk_size * id;
            int end = chunk_size*(id+1)-1;
            if(id == num_threads-1)
                end = binaryString.size()-1;

            if(id != num_threads-1) {
                for(int i=0; i < Tt.size(); i++)
                    route[id+1][Tt[i][binaryString[end]]] = std::vector<int>();
            } // O(s); s: N mero de estados del automata

            #pragma omp barrier

            for(const auto& pair: route[id]) {
                currentState = pair.first;
                route[id][currentState] = {currentState};
                if(F[currentState])
                    found++;
                for (int i = start; i <= end; i++) {
                    currentState = Tt[currentState][binaryString[i]];
                    route[id][pair.first].push_back(currentState);
                    if(F[currentState])
                        found++;
                }
            } // O(s*((n/p)+p))

            #pragma omp critical
            {
                std::cout << '\n' << "Proceso " << id << ":" << '\n';
                for (const auto& pair : route[id]) {
                    std::cout << " - Clave: " << pair.first << " -> Valores: ";
                    for (int val : pair.second)
```

```
                    std::cout << val << " ";
                // std::cout << std::endl;
            }
        }
    }
    return 0;
};
```

3. Análisis de tiempos

Datos obtenidos de la ejecución secuencial del algoritmo.

Tamaño de Input	Automata 1	Automata 2
10000	0.0116089	0,0172885
100000	2.82116	3,32962
500000	153.187	170,44
1000000	739.811	868,726

Cuadro 3.0.0.1: Tiempo secuencial por tamaño de input

Datos obtenidos de la ejecución paralelizada con la primera versión del algoritmo PaREM.

Tamaño de Input	Automata 1	Automata 2
10000	0.00205046	0.00189931
100000	0.0139479	0.0209181
500000	0.0647052	0.101981
1000000	0.115523	0.143738

Cuadro 3.0.0.2: Tiempo secuencial por tamaño de input

Datos obtenidos de la ejecución paralelizada con la segunda versión del algoritmo PaREM.

Tamaño de Input	Automata 1	Automata 2
10000	0,00220293	0,00357586
100000	0,0159567	0,0286324
500000	0,0855477	0,0866076
1000000	0,137176	0,183018

Cuadro 3.0.0.3: Tiempo secuencial por tamaño de input

Gráficas generadas del performance del algoritmo.

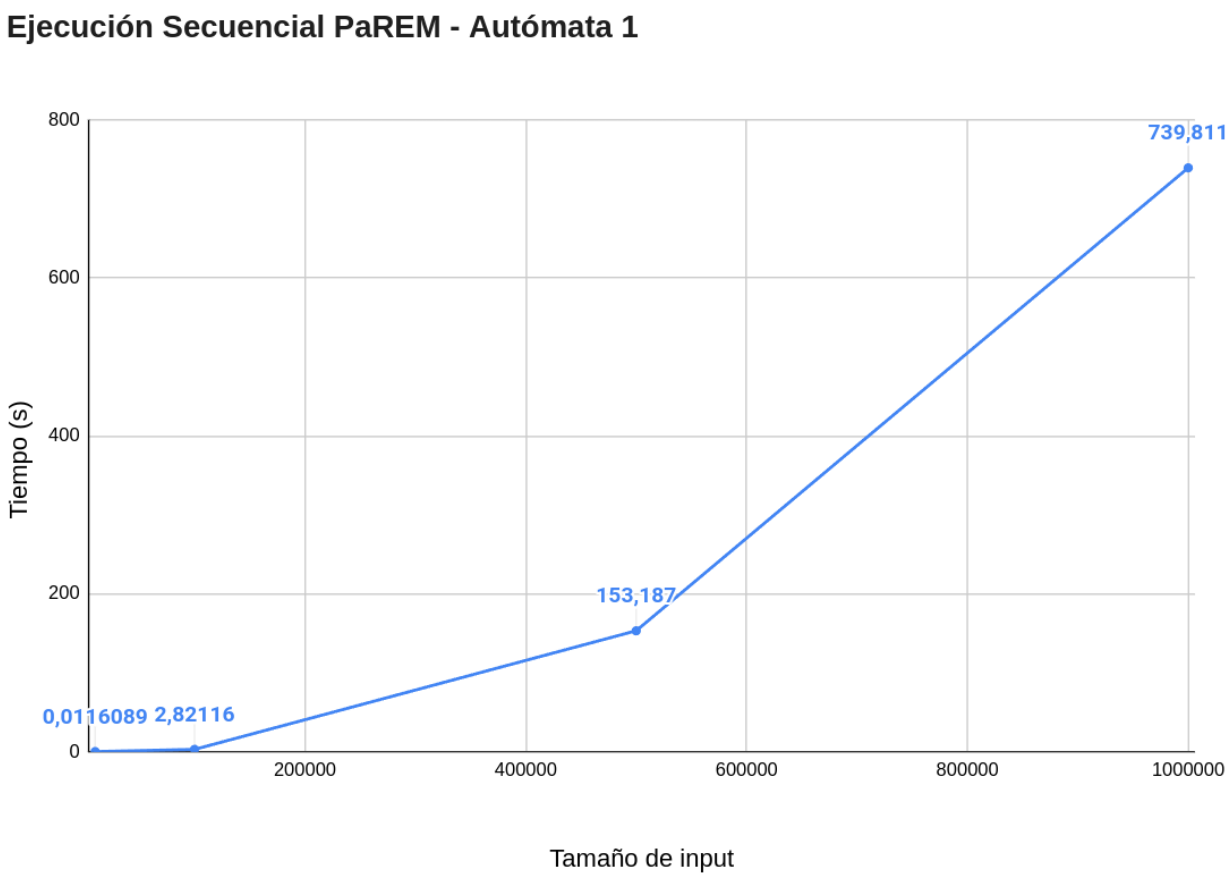

Figura 3.1: Ejecución Secuencial PaREM, Automata 1

4. Conclusiones

- La paralelización compartida es una buena opción para implementar algoritmos PAREM en paralelo, debido a diferentes hilos pueden acceder a diferentes partes de la entrada sin necesidad de interferir en el trabajo de los demás hilos.
- En la algoritmos paralelos, la cantidad de estados del automata influyen en el rendimiento de los algoritmos paralelos, problema que no aparece en versión secuencial. A pesar de este incoveniente, los PAREM paralelos rinden mejor que las versiones secuencias por una notable diferencia.

5. Anexos

5.1. Benchmark

Carpeta de Drive con Anexos y gráficas: Drive

5.2. Código Github

Enlace al repositorio: PaREM-implem

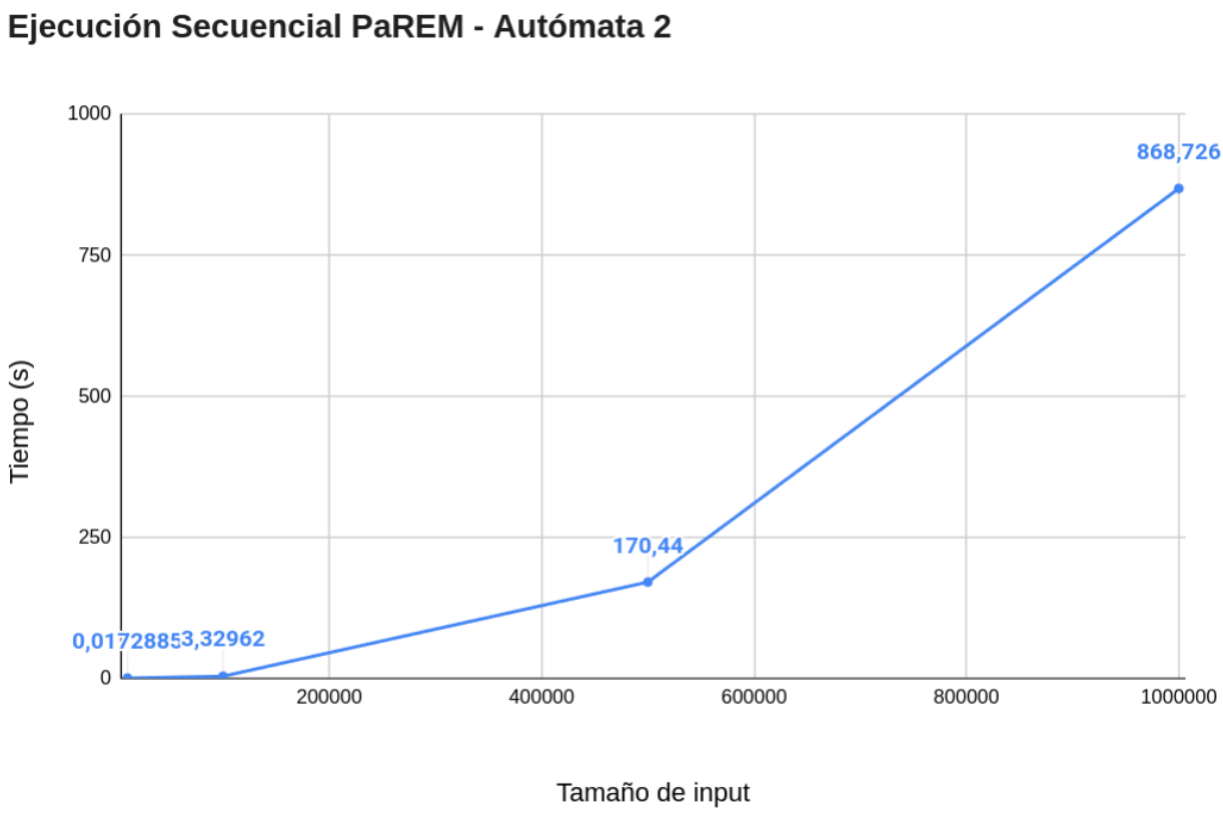

Figura 3.2: Ejecución Secuencial PaREM, Automata 2

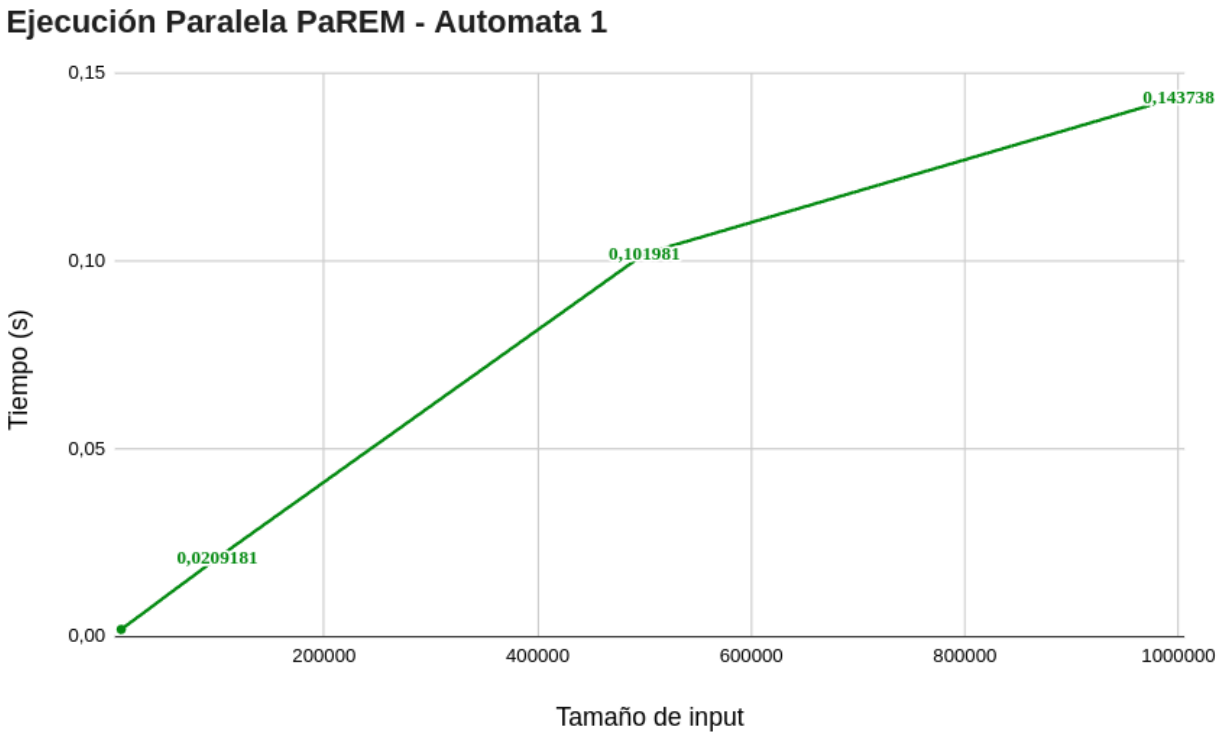

Figura 3.3: Ejecución Paralela PaREM, Automata 1

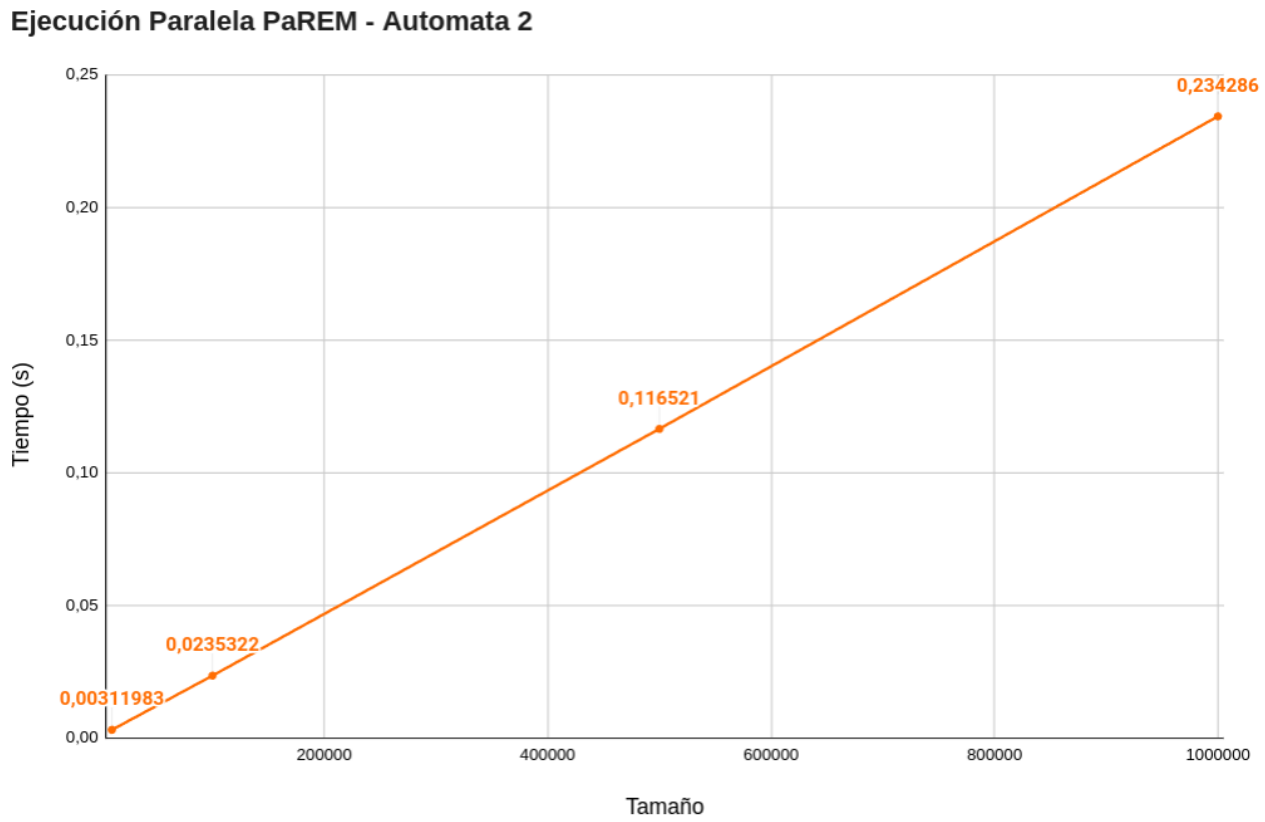

Figura 3.4: Ejecución Paralela PaREM, Automata 2

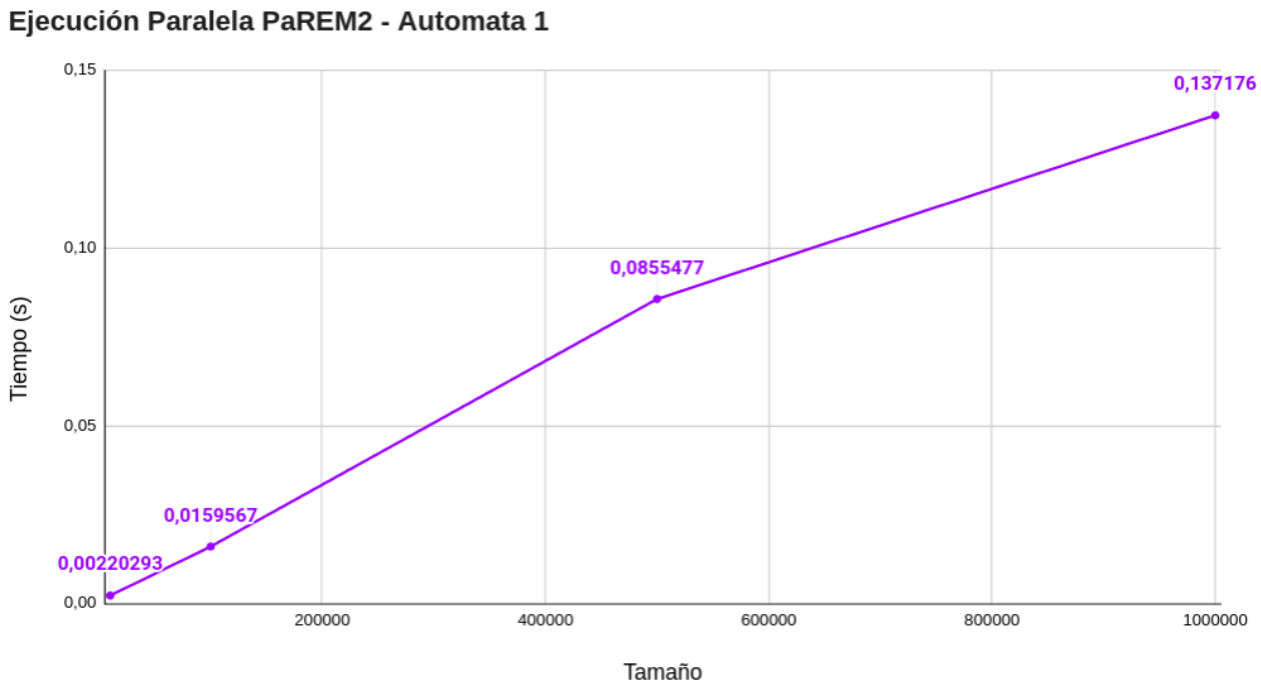

Figura 3.5: Ejecución Paralela PaREM2, Automata 1

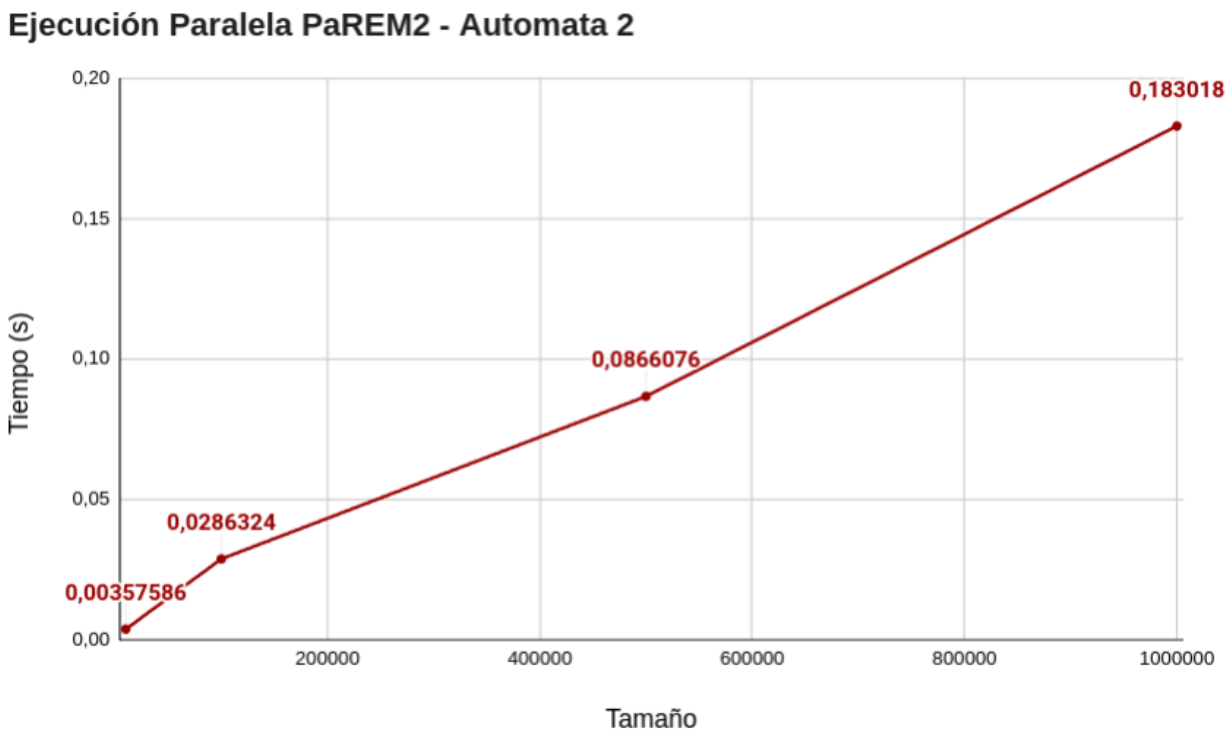

Figura 3.6: Ejecución Paralela PaREM2, Automata 2

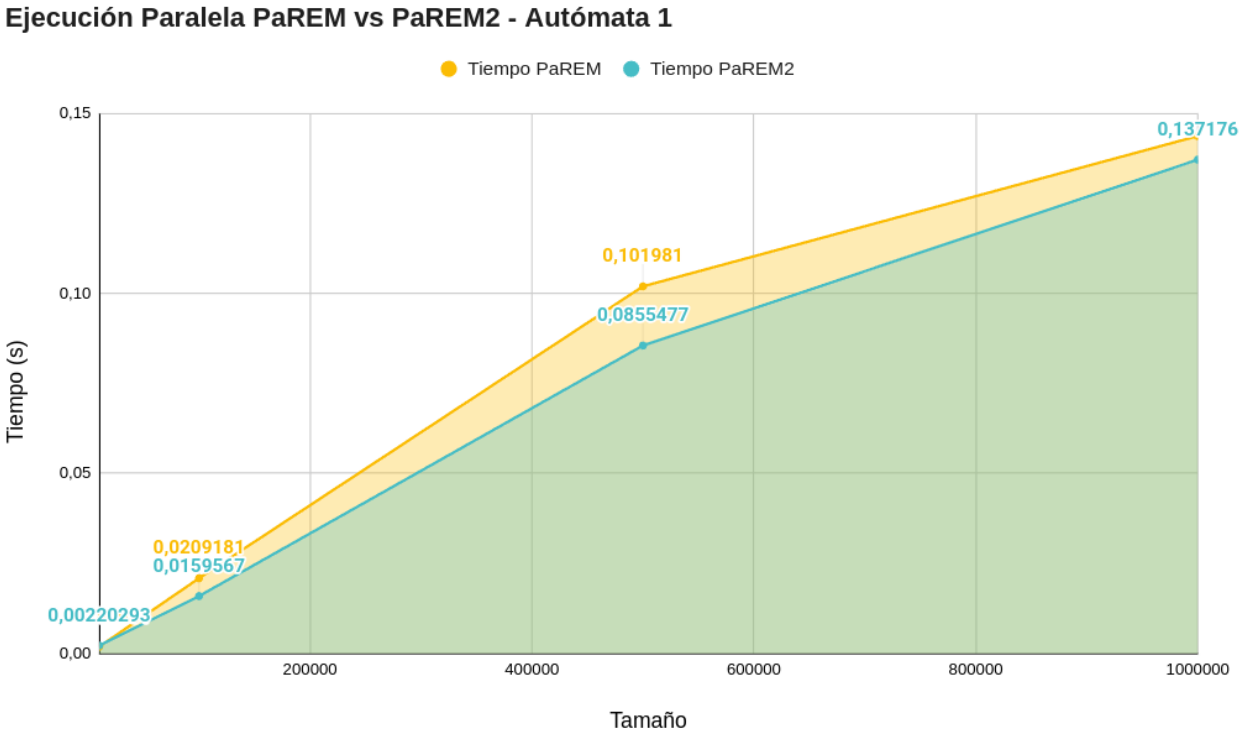

Figura 3.7: Ejecución Paralela PaREM vs PaREM2, Automata 1

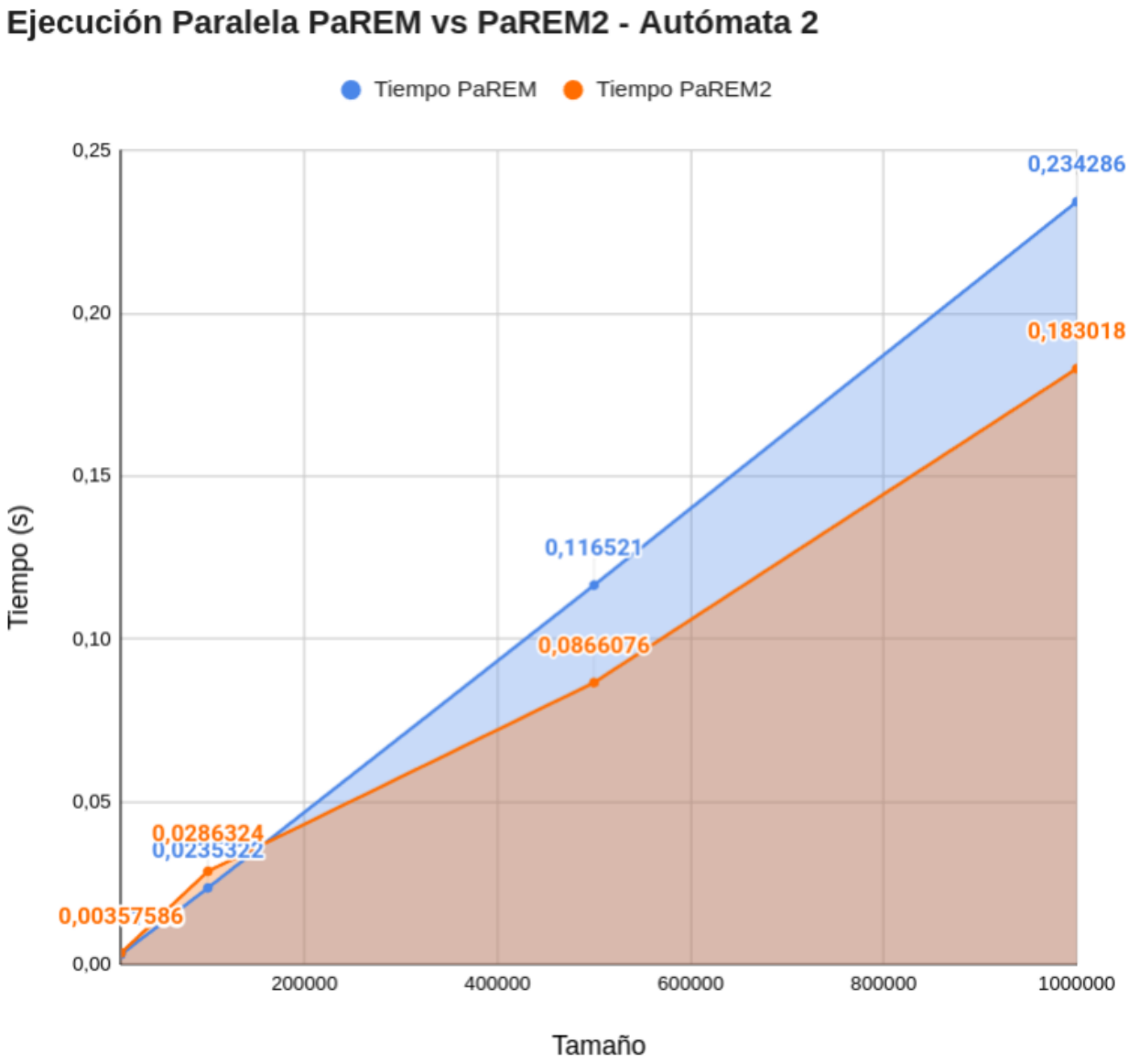

Figura 3.8: Ejecución Paralela PaREM vs PaREM2, Automata 2

Referencias

[1] S. Memeti and S. Pllana, "Parem: A novel approach for parallel regular expression matching," in *2014 IEEE 17th International Conference on Computational Science and Engineering*. IEEE, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1109/CSE.2014.146