# Skip List Paralela

Javier Oswaldo Alvarez Reyes

12 de diciembre de 2022

**Resumen**

La estructura de datos que presentaré como trabajo final es skip list,la razón para elegir esta estructura fue que al momento de paralelizar esta a diferencia de otras estructuras solo necesita un subproceso que nos da la facilidad de paralelizar. En cuanto a la complejidad de programarlo al ser parecido a una lista enlazada simple no supo una dificultad programarlo.

## 1. Introducción

Una lista de saltos es una estructura de datos probabilística con funciones como inserción, eliminación y búsqueda logarítmica en el caso promedio con una secuencia ordenada de elementos mientras se mantiene una estructura similar a una lista enlazada.

## 2. Implementación

### 2.1. Concurrent Skip list

Una lista de omisión simultánea es una implementación multiproceso de la estructura de datos de la lista de omisión en la que el Las operaciones de inserción, eliminación, obtención y rango se pueden realizar juntas en múltiples hilos tomando cada acción como un subproceso.

### 2.2. Estrategias de implementación.

la estructura KeyValuePair almacena una clave y un valor para cada nodo. En mi implementación, la clave es un entero y el valor es una cadena. La variable miembro next apunta al siguiente nodo en cada nivel de la lista de omisión. Cada nodo usa un nodeLock para bloquear el nodo cuando se está modificado. Se utiliza una variable atómica marcada para indicar si un nodo se está eliminando y se utiliza otra variable atómica Full Linked para indicar si el nodo está completamente vinculado a su sucesores y antecesores. La variable miembro Top Level tiene el nivel máximo hasta el cual el nodo particular está disponible.

```
#define INT_MINI numeric_limits<int>::min()
#define INT_MAXI numeric_limits<int>::max()
static int max_level;
using namespace std;
class KeyValuePair {
private:
  int key;
  string value;
public:
  KeyValuePair();
  KeyValuePair(int key, string value);
  ~KeyValuePair();
  int get_key();
  string get_value();
};
```

Listing 1: KeyValuePair

```
class Node {
public:
  // Almacena la clave y el valor del nodo.
  KeyValuePair key_value_pair;
  // Almacena la referencia del siguiente nodo hasta el nivel superior del nodo
  vector<Node*> next;
  // Bloquear para bloquear el nodo al modificarlo
  mutex node_lock;
  // Variable at mica donde al marcar si se elimina este Nodo
  atomic<bool> marked = { false };
  // Variable at mica para indicar que el Nodo est  completamente vinculado a
    predecesores y sucesores.
  atomic<bool> fully_linked = { false };
  //El nivel m ximo hasta el cual el nodo est  disponible
  int top_level;
  Node();
  Node(int key, int level);
  Node(int key, string value, int level);
  ~Node();
  int get_key();
  string get_value();
  void lock();
  void unlock();
};
```

Listing 2: Node

## 2.3. Insertar

Antes de insertar un elemento en la lista de omisión, verificamos si el elemento ya está presente en la lista de omisión y si el nodo está marcado. Si el elemento ya está presente y el nodo es sin marcar, no insertamos el elemento ya que ya está presente en la lista de omisión. Si el elemento está presente y el nodo no está completamente vinculado, luego esperamos hasta que esté completamente vinculado antes de insertar.

```
bool SkipList::add(int key, string value) {

    //Obtener el nivel hasta el cual debe estar disponible el nuevo nodo
    int top_level = get_random_level();

    //Inicializacion de referencias de los predecesores y sucesores
    vector<Node*> preds(max_level + 1);
    vector<Node*> succs(max_level + 1);

    for (size_t i = 0; i < preds.size(); i++) {
        preds[i] = NULL;
        succs[i] = NULL;
    }

    // Sige intentando insertar el elemento en la lista. En caso de que se cambien los
     predecesores y los sucesores,
    // este bucle ayuda a intentar la insercion de nuevo
    while (true) {


// Encuentra los predecesores y sucesores de donde se debe insertar la clave
        int found = find(key, preds, succs);


// Si se encuentra y se marca, esperar y continuar insertar
        // Si se encuentra y no esta marcado, espere hasta que este completamente
    vinculado y regrese. No se necesita insercion
        // Si no lo encuentra, continue con la insercion
        if (found != -1) {
            Node* node_found = succs[found];

            if (!node_found->marked) {
                while (!node_found->fully_linked) {
                }
```

```
33                  return false;
34              }
35              continue;
36          }
37
38
39          // Almacenar todos los Nodos que bloqueamos adquirimos en un mapa
40          // Mapa utilizado para que no intentemos adquirir el bloqueo de un Nodo que ya
     hemos adquirido
41          // Esto puede pasar cuando tenemos el mismo predecesor en diferentes niveles
42          map<Node*, int> locked_nodes;
43
44          // Atraviese la lista de saltos e intente adquirir el bloqueo del predecesor
    en todos los niveles
45          try {
46              Node* pred;
47              Node* succ;
48
49
50          // Se usa para verificar si el predecesor y los sucesores son los mismos que
    cuando intentamos leerlos antes
51              bool valid = true;
52
53              for (int level = 0; valid && (level <= top_level); level++) {
54                  pred = preds[level];
55                  succ = succs[level];
56
57
58              // Si aun no ha adquirido el bloqueo, adquiera el bloqueo
59                  if (!(locked_nodes.count(pred))) {
60                      pred->lock();
61                      locked_nodes.insert(make_pair(pred, 1));
62                  }
63
64                  // Si se marco el predecesor o si el predecesor y los sucesores
    cambian, aborte y vuelva a intentarlo
65                  valid = !(pred->marked.load(std::memory_order_seq_cst)) && !(succ->
    marked.load(std::memory_order_seq_cst)) && pred->next[level] == succ;
66              }
67
68              // No se cumplen las condiciones, libere los bloqueos, cancele y vuelva a
    intentarlo.
69              if (!valid) {
70                  for (auto const& x : locked_nodes) {
71                      x.first->unlock();
72                  }
73                  continue;
74              }
75
76              // Todas las condiciones satisfechas, crea el Nodo e insrrtalo ya que
    tenemos todos los bloqueos requeridos
77              Node* new_node = new Node(key, value, top_level);
78
79              // Actualizar al predecesor y sucesores
80              for (int level = 0; level <= top_level; level++) {
81                  new_node->next[level] = succs[level];
82              }
83
84              for (int level = 0; level <= top_level; level++) {
85                  preds[level]->next[level] = new_node;
86              }
87
88              // Marca el nodo como completamente vinculado.
89              new_node->fully_linked = true;
90
91
92              // Liberar el bloqueo de todos los nodos retenidos una vez que se completa
     la insercion
93              for (auto const& x : locked_nodes) {
94                  x.first->unlock();
95              }
```

```
96
97              return true;
98          }
99          catch (const std::exception& e) {
100 // Si se produce alguna excepcion durante la insercion anterior, libere los bloqueos
    de los nodos retenidos y vuelva a intentarlo.
101             std::cerr << e.what() << '\n';
102             for (auto const& x : locked_nodes) {
103                 x.first->unlock();
104             }
105         }
106     }
107 }
```

Listing 3: Insertar

## 2.4. Busqueda

La búsqueda de un elemento en la lista de saltos se realiza recorriendo toda la lista de saltos a mayor altura. nivel y bajando a niveles más bajos a medida que la búsqueda se acerca a la clave de búsqueda.

```
1  bool SkipList::add(int key, string value) {
2
3      // Obtener el nivel hasta el cual debe estar disponible el nuevo nodo
4      int top_level = get_random_level();
5
6      // Inicializacion de referencias de los predecesores y sucesores
7      vector<Node*> preds(max_level + 1);
8      vector<Node*> succs(max_level + 1);
9
10     for (size_t i = 0; i < preds.size(); i++) {
11         preds[i] = NULL;
12         succs[i] = NULL;
13     }
14
15     // Sigue intentando insertar el elemento en la lista. En caso de que se cambien
       los predecesores y los sucesores.
16
17     // este bucle ayuda a intentar la insercion de nuevo
18     while (true) {
19
20
21     // Encuentra los predecesores y sucesores de donde se debe insertar la clave
22         int found = find(key, preds, succs);
23
24         // Si se encuentra y se marca, esperar y continuar insertar.
25         // Si se encuentra y no esta marcado, espere hasta que este completamente
       vinculado y regrese. No se necesita insercion.
26         // Si no lo encuentra, continue con la insercion.
27         if (found != -1) {
28             Node* node_found = succs[found];
29
30             if (!node_found->marked) {
31                 while (!node_found->fully_linked) {
32                 }
33                 return false;
34             }
35             continue;
36         }
37
38         // Almacenar todos los Nodos que bloqueamos adquirimos en un mapa.
39         // Mapa utilizado para que no intentemos adquirir el bloqueo de un Nodo que ya
        hemos adquirido.
40         // Esto puede pasar cuando tenemos el mismo predecesor en diferentes niveles.
41         map<Node*, int> locked_nodes;
42
43     // Atraviese la lista de saltos e intente adquirir el bloqueo del predecesor en
       todos los niveles.
44         try {
```

```
45          Node* pred;
46          Node* succ;
47
48
49  // Se usa para verificar si el predecesor y los sucesores son los mismos que
    cuando intentamos leerlos antes.
50          bool valid = true;
51
52          for (int level = 0; valid && (level <= top_level); level++) {
53              pred = preds[level];
54              succ = succs[level];
55
56              // Si aun no ha adquirido el bloqueo, adquiera el bloqueo.
57              if (!(locked_nodes.count(pred))) {
58                  pred->lock();
59                  locked_nodes.insert(make_pair(pred, 1));
60              }
61
62              // Si el predecesor esta marcado o si el predecesor y los sucesores
    cambian, cancela y vuelve a intentarlo.
63
64              valid = !(pred->marked.load(std::memory_order_seq_cst)) && !(succ->
    marked.load(std::memory_order_seq_cst)) && pred->next[level] == succ;
65          }
66
67
68  // No se cumplen las condiciones, libere los bloqueos, cancele y vuelva a
    intentarlo.
69          if (!valid) {
70              for (auto const& x : locked_nodes) {
71                  x.first->unlock();
72              }
73              continue;
74          }
75
76          // Todas las condiciones satisfechas, crea el Nodo e insertalo ya que
    tenemos todos los bloqueos requeridos
77          Node* new_node = new Node(key, value, top_level);
78
79          // Actualizar al predecesor y sucesores
80          for (int level = 0; level <= top_level; level++) {
81              new_node->next[level] = succs[level];
82          }
83
84          for (int level = 0; level <= top_level; level++) {
85              preds[level]->next[level] = new_node;
86          }
87
88          // Marca el nodo como completamente vinculado.
89          new_node->fully_linked = true;
90
91
92      // Liberar el bloqueo de todos los nodos retenidos una vez que se completa la
    insercion
93          for (auto const& x : locked_nodes) {
94              x.first->unlock();
95          }
96
97          return true;
98      }
99      catch (const std::exception& e) {
100         // Si se produce alguna excepcion durante la insercion anterior, libere
    los bloqueos de los nodos retenidos y vuelva a intentarlo.
101         std::cerr << e.what() << '\n';
102         for (auto const& x : locked_nodes) {
103             x.first->unlock();
104         }
105     }
106 }
```

```
}
```

Listing 4: Busqueda

## 2.5. Eliminar

Antes de eliminar un elemento de la lista de saltos, verificamos si el elemento está presente en el salto lista y si el nodo no está presente, regresamos. Si el elemento esta presente, comprobamos si está completamente vinculado y desmarcado si no, intentamos eliminar

```
bool SkipList::remove(int key) {

    Node* victim = NULL;
    bool is_marked = false;
    int top_level = -1;


    // Inicializacion de referencias de los predecesores y sucesores
    vector<Node*> preds(max_level + 1);
    vector<Node*> succs(max_level + 1);

    for (size_t i = 0; i < preds.size(); i++) {
        preds[i] = NULL;
        succs[i] = NULL;
    }


    // Sigue intentando eliminar el elemento de la lista. En caso de que se cambien los predecesores y los sucesores,
    // este bucle ayuda a intentar borrar de nuevo
    while (true) {

    // Encuentra los predecesores y sucesores de donde se eliminar  la clave
        int found = find(key, preds, succs);


    // Si lo encuentra, seleccione el nodo a eliminar. de lo contrario volver
        if (found != -1) {
            victim = succs[found];
        }


    // Si no se encuentra el nodo y el nodo que se eliminar  est  completamente vinculado y no est  marcado, devuelva
        if (is_marked |
            (found != -1 &&
                (victim->fully_linked && victim->top_level == found && !(victim->marked))
                )
            ) {
    // Si no est  marcado, bloqueamos el nodo y marcamos el nodo para eliminar
            if (!is_marked) {
                top_level = victim->top_level;
                victim->lock();
                if (victim->marked) {
                    victim->unlock();
                    return false;
                }
                victim->marked = true;
                is_marked = true;
            }


            map<Node*, int> locked_nodes;


            try {
                Node* pred;
                //Node* succ;

```

```
                bool valid = true;

                for (int level = 0; valid && (level <= top_level); level++) {
                    pred = preds[level];


                    if (!(locked_nodes.count(pred))) {
                        pred->lock();
                        locked_nodes.insert(make_pair(pred, 1));
                    }


                    valid = !(pred->marked) && pred->next[level] == victim;
                }


                if (!valid) {
                    for (auto const& x : locked_nodes) {
                        x.first->unlock();
                    }
                    continue;
                }


                for (int level = top_level; level >= 0; level--) {
                    preds[level]->next[level] = victim->next[level];
                }

                victim->unlock();
                for (auto const& x : locked_nodes) {
                    x.first->unlock();
                }

                return true;
            }
            catch (const std::exception& e) {

                for (auto const& x : locked_nodes) {
                    x.first->unlock();
                }
            }

        }
        else {
            return false;
        }
    }
}
```

Listing 5: Eliminar

## 2.6. Rango

La operación de rango funciona de manera similar a la búsqueda en la que recorremos la lista de saltos a mayor nivel y baje a un nivel más bajo a medida que nos acercamos al inicio del rango.

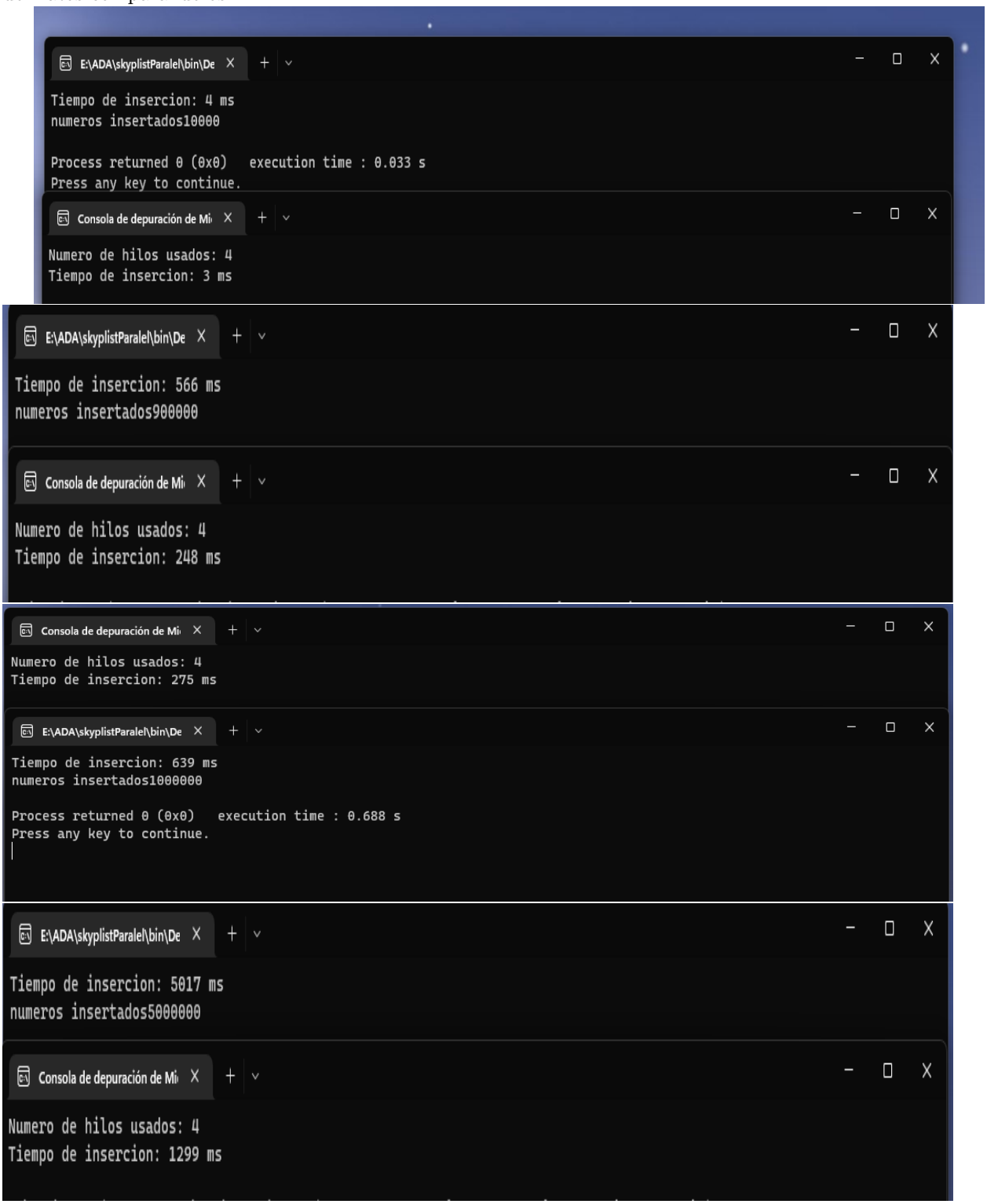
```
void skiplist_range(int start, int end) {
    map<int, string> range_output = skiplist.range(start, end);

    string s = "";
    for (auto const& x : range_output) {
        s += x.second + " ";
    }
    cout << "Range (" << start << ", " << end << ") = " << s << endl;
}
```

Listing 6: Eliminar

## 2.7. Diferencias entre Paralelo

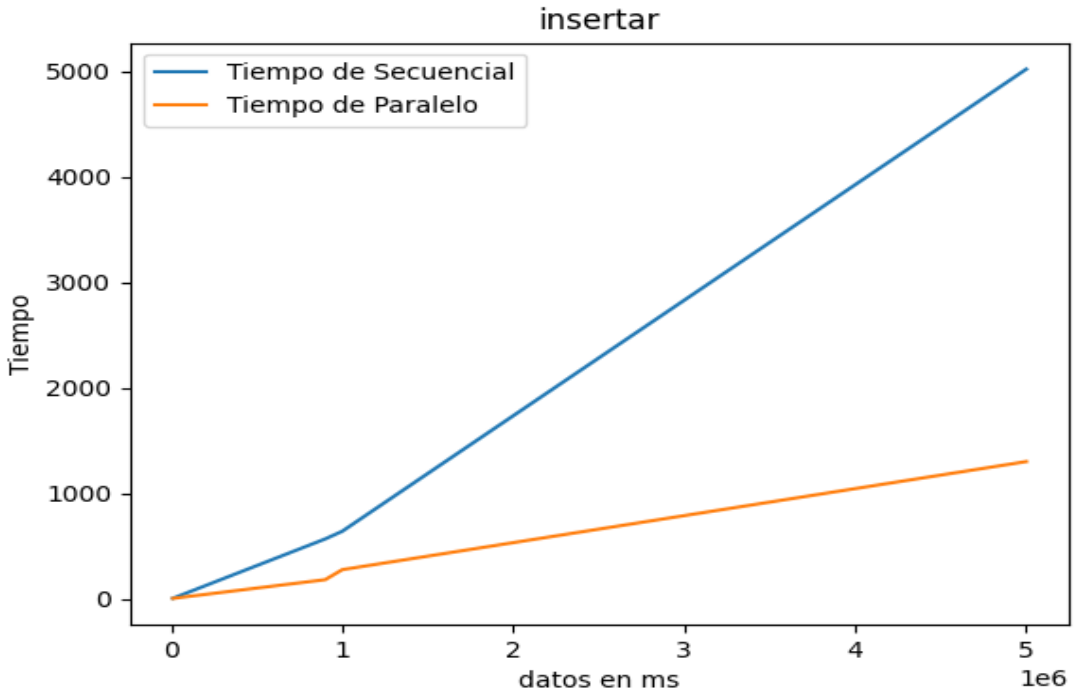
En esta parte Compararemos el tiempo de compilación en la insercion con diferentes Cantidades de Datos comparandolos.

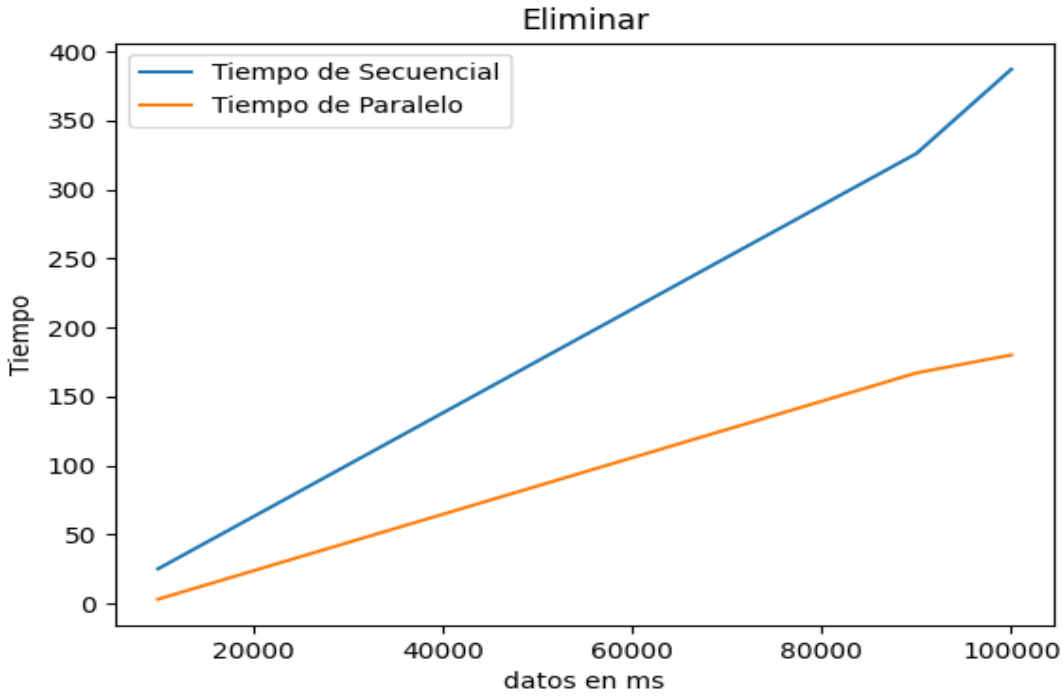



en cuanto a los datos que sacamos con la eliminacion esta nos da con los mismos datos comparandolos.


## 2.8. Conclusion

Como podemos observar comparando el tiempo de compilacion entre un algoritmo paralelo y uno secuencial ,el paralelo siempre va a estar mejor, ahora en cuanto a la cantidad de mejora que este pueda tener no se puede dar exactamente dado que el metodo de insercion para ambos algoritmos es uno Aleatorio e incluso por el mismo sistema en que funciona la skyplist no se podria realizar una comparacion exacta pero aunque la estructura funciona aleatoriamente se puede ver una clara mejora en su comparacion.




## Referencias

[1] Shreyas Gopalakrishna CSCI 7000-017 Concurrent Programming.

[2] https://www.cs.upc.edu/~conrado/research/papers/tcs-gmm96.pdf A design of a parallel dictionary using skip lists.

[3] CONCURRENT MAINTENANCE OF SKIP LISTS https://15721.courses.cs.cmu.edu/spring2019/papers/07-oltpindexes1/pugh-concurrent-tr1990.pdf