Reporte de Proyecto Final para el Curso de Estructuras de Datos Avanzadas (2022-II): Comparación de la Implementación Secuencial y Paralela de SkipList

Joey Patrick Flores Dávila

December 2022

1. Introducción

La programación paralela es el uso de múltiples recursos para resolver un problema, se distingue de la programación secuencial en que varias operaciones pueden ocurrir simultáneamente, gracias a esto es que se puede ejecutar una mayor cantidad de operaciones en menos tiempo, sin embargo su implementación suele ser más difícil. En el presente documento se va a presentar y explicar la implementación de la estructura de datos SkipList tanto en su forma Secuencial como Concurrente y al final se realizarán diversos experimentos para ver que implementación es mas eficiente.

2. SkipList

SkipList es una estructura de datos probabilística con una 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. La SkipList se basa en varias capas. La capa inferior tiene todos los elementos conectados y las capas superiores se comportan como un carril rápido para omitir algunos elementos para atravesar esta estructura de manera más rápida.

2.1. Implementación Computacional

En esta sección se mostrará las funciones más importantes de ambas implementaciones y se explicará a detalle en que consiste cada una de ellas.

2.1.1. Estrategia Secuencial

La estrategia de esta implementación se basa en utilizar una función auxiliar llamada *FindNode*, que va a ser la encargada de guardar los predecesores en cada nivel de un nodo. Gracias a esta función también vamos a poder averiguar si es que existe o no un elemento dentro de la estructura, esto nos va a ayudar al momento de insertar, buscar o eliminar un elemento.

- **SkipListNode**

```
template <typename ElemType>
struct SkipListNode{

    ElemType value;
    int top_level;
    vector<SkipListNode<ElemType>*> next;

};
```

Fig 1. Clase Nodo

Empezando por la clase *SkipListNode*, se tiene una variable *value* que almacena el valor para cada nodo, una variable top_level que tiene el nivel máximo que el nodo puede tener y por último cuenta con un vector de nodos que va a servir para guardar un puntero al siguiente nivel de la estructura.

- **SkipList - random_level**

```
template <typename ElemType>
int SkipList<ElemType>::random_level() {

    int level = 0;
    while (level < max_level) {
        int aux = numero_aleatorio();   //Genera un numero aleatorio entre 0 y 1
        if (aux == 1) {
            level++;
        }
        else {
            break;
        }
    }
    return level;
}
```

Fig 2. Función auxiliar random_level

Esta función auxiliar es la encargada de generar el nivel aleatorio que tendrá cada nodo al momento de ser creado en la inserción, para esto tendremos dos variables, la primera variable *level* guardará el nivel máximo que tendrá el nodo y la segunda variable *aux* va a generar un numero aleatorio entre 0 y 1, si el número generado aleatoriamente es 1, entonces la variable *level* aumentará en 1, caso contrario se detendrá el bucle y retornará el nivel máximo conseguido por esta variable. Este procedimiento se va a repetir mientras que la variable *level* sea menor al nivel máximo permitido por la SkipList o hasta que la variable *aux* sea igual a 0.

- **SkipList - FindNode**

```
template <typename ElemType>
SkipListNode<ElemType>* SkipList<ElemType>::FindNode(ElemType value, vector<
    SkipListNode<ElemType>*>& predecessors) {
    predecessors.resize(max_level + 1);

    for (int i = 0; i < predecessors.size(); i++) {
        predecessors[i] = nullptr;
    }

    SkipListNode<ElemType>* p = head;
    for (int i = current_level; i >= 0; i--) {
        SkipListNode<ElemType>* current_node = p->next[i];
        while (current_node && current_node->value < value) {
            p = current_node;
            current_node = p->next[i];
        }
        predecessors[i] = p;
    }
    p = p->next[0];
    return p;
}
```

Fig 3. Función auxiliar Find Node

Esta es una función auxiliar que se usa para realizar la inserción, eliminación y búsqueda, aquí se tiene un vector de nodos llamado *predecessors* que se encarga de guardar los predecesores en cada nivel del nodo que se desea insertar o eliminar, esta función auxiliar retorna un puntero de la clase *SkipListNode* con el fin de verificar si es que existe o no ese elemento a la hora de realizar las operaciones anteriormente mencionadas.

- **SkipList - insert**

```
1  template <typename ElemType>
2  bool SkipList<ElemType>::insert(ElemType value) {
3
4
5      vector<SkipListNode<ElemType>*> predecessors;
6      SkipListNode<ElemType>* p = FindNode(value, predecessors);
7
8      if (p && p->value == value) {
9          return 0;
10     }
11
12     int new_level = random_level();
13     if (new_level > current_level) {
14         for (int i = current_level + 1; i < new_level + 1; i++) {
15             predecessors[i] = head;
16         }
17         current_level = new_level;
18     }
19
20     SkipListNode<ElemType>* new_node = new SkipListNode<ElemType>(new_level, value
       );
21     for (int i = 0; i <= new_level; i++) {
22         new_node->next[i] = predecessors[i]->next[i];
23         predecessors[i]->next[i] = new_node;
24     }
25
26     return 1;
27 }
```

Fig 4. Función Insertar

Para realizar la inserción se llama a la función auxiliar *FidNode*, gracias a esta función podremos ver si es que existe ese elemento en la SkipList, en caso de que ya se haya insertado con anterioridad este elemento simplemente se retorna *Falso* ya que no se permiten elementos repetidos, en caso no se encuentre el elemento en la estructura se crea un nuevo nodo con el elemento deseado y con su respecto nivel generado aleatoriamente (línea 12). Lo siguiente es vincular el nuevo nodo con sus sucesores y vincular los predecesores con el nuevo nodo, esto se tiene que realizar en todos los niveles de la estructura (líneas 20 a 24). Una vez se haya terminado el proceso de vinculación se retornará *Verdadero* indicando que se insertó correctamente ese elemento.

- **SkipList - remove**

```
1  template <typename ElemType>
2  bool SkipList<ElemType>::remove(ElemType value) {
3
4      vector<SkipListNode<ElemType>*> predecessors;
5      SkipListNode<ElemType>* p = FindNode(value, predecessors);
6
7      if (!(p && p->value == value)) return 0;
8
9      for (int i = 0; i <= current_level; i++) {
10         if (predecessors[i]->next[i] != p) {
11             break;
12         }
13         predecessors[i]->next[i] = p->next[i];
14     }
15
16     delete p;
17     while (current_level > 0 && head->next[current_level] == nullptr) {
18         current_level--;
19     }
20
21     return 1;
22 }
```

Fig 5. Función Eliminar

Para realizar la eliminación se llama a la función auxiliar *FindNode*, gracias a esta función podremos ver si es que existe ese elemento en la SkipList, en caso no exista el elemento simplemente se retorna *Falso* indicando que no se realizó la eliminación, caso contrario se vinculan los predecesores con sus sucesores (líneas 9 a 14). Una vez terminada esta vinculación se elimina ese nodo con el elemento deseado y es necesario hacer una verificación, ya que después de la eliminación del elemento, podría haber niveles sin elementos, por lo que también eliminaremos estos niveles disminuyendo el nivel actual de la SkipList (líneas 17 a 18). Y por ultimo se retorna *Verdadero* indicando que se eliminó correctamente ese elemento.

- **SkipList - search**

```
1  template <typename ElemType>
2  bool SkipList<ElemType>::search(ElemType value) {
3      vector<SkipListNode<ElemType>*> predecessors;
4      SkipListNode<ElemType>* p = FindNode(value, predecessors);
5      if (p && p->value == value) {
6          return true;
7      }
8      return false;
9  }
```

Fig 6. Función Buscar

Para la búsqueda también se usa la función auxiliar *FindNode*, ya que esta nos permite ver si es que existe o no el elemento, Si es que existe el elemento se retorna *Verdadero* y caso contrario retorna *Falso*. La desventaja de esta búsqueda es que siempre se va a tener que bajar al primer nivel de la SkipList así se haya encontrado el elemento en el nivel mas alto, entonces esto aumenta el tiempo al realizar dicha búsqueda.

- **SkipList - optimized_search**

```
1  template <typename ElemType>
2  bool SkipList<ElemType>::optimized_search(ElemType value) {
3
4      SkipListNode<ElemType>* p = head;
5      for (int i = current_level; i >= 0; i--) {
6          SkipListNode<ElemType>* current_node = p->next[i];
7          while (current_node && current_node->value < value) {
8              p = current_node;
9              current_node = p->next[i];
10         }
11         if (current_node && current_node->value == value) {
12             return true;
13         }
14     }
15     return false;
16 }
```

Fig 7. Función Buscar Optimizado

Como su nombre lo dice esta función es una búsqueda optimizada ya que se empieza a buscar el elemento deseado desde el nivel mas alto de la estructura y apenas se encuentre el elemento simplemente retorna *Verdadero*, caso contrario retorna *Falso*.

2.1.2. Estrategia Paralela

La estrategia de la implementación paralela al igual que la secuencial se basa en usar la función auxiliar *FindNode* para guardar los predecesores y sucesores de un nodo y para ver si es que existe o no el elemento en la estructura, sin embargo, es necesario agregar y modificar algunas cosas para que esta implementación pueda permitir el paralelismo.

En esta estrategia se utiliza el bloqueo manual para acceder a los nodos para escribir y durante la lectura no es necesario adquirir bloqueos, lo que requiere es solo una breve validación basada en bloqueos antes de agregar o quitar nodos.

- **SkipListNode**

```
template <typename ElemType>
struct SkipListNode {

    ElemType value;
    vector<SkipListNode<ElemType>*> next;
    mutex node_lock;
    bool marked = false;
    bool fully_linked = false;
    int top_level;

};
```

Fig 8. Clase Nodo

Empezando por la clase *SkipListNode*, se tiene una variable *value* que almacena el valor para cada nodo, este valor siempre va a ser mayor que el valor de su predecesor. Cada nodo utiliza un *node_lock* para bloquear el nodo cuando se está modificando. Una variable *marked* se utiliza para indicar si un nodo se está eliminando y otra variable *fully_linked* se utiliza para indicar si el nodo está completamente vinculado a sus sucesores y predecesores. La variable top_level tiene el nivel máximo que el nodo puede tener.

- **SkipList - FindNode**

```
template <typename ElemType>
bool SkipList<ElemType>::FindNode(ElemType value, vector<SkipListNode<ElemType>*>&
     predecessors, vector<SkipListNode<ElemType>*>& successors, int& current_level
    ) {

    predecessors.resize(max_level + 1);
    successors.resize(max_level + 1);

    for (size_t i = 0; i < predecessors.size(); i++) {
        predecessors[i] = nullptr;
        successors[i] = nullptr;
    }

    bool found = false;
    SkipListNode<ElemType>* prev = head;

    for (int level = max_level; level >= 0; level--) {
        SkipListNode<ElemType>* curr = prev->next[level];

        while (curr->value < value) {
            prev = curr;
            curr = prev->next[level];
        }

        if (!found && curr->value == value) {
            current_level = level;
            found = true;
        }

        predecessors[level] = prev;
        successors[level] = curr;
    }
    return found;
}
```

Fig 9. Función auxiliar Find Node

La inserción, eliminación y búsqueda en la SkipList se realiza mediante la función auxiliar *FindNode*, que toma el *valor* y dos vectores de nivel máximo *preds* y *succs* de punteros de nodo, y busca exactamente como en una SkipList , comenzando en el nivel más alto y procediendo al siguiente nivel inferior cada vez que encuentra un nodo cuya clave es mayor o igual al valor buscado. El subproceso guarda en el vector preds el último nodo con una clave menor que el valor

buscado que encontró en cada nivel, y el sucesor de ese nodo (que debe tener una clave mayor o igual que el valor buscado) en el vector succs. Además, esta función retorna *Verdadero* si es que encuentra un nodo con el valor buscado y guarda el primer nivel donde se encontró dicho nodo para usarlo posteriormente en las demás funciones, y si es que no se encuentra un nodo con el valor buscado simplemente retorna *Falso*. Esta función no adquiere ningún bloqueo, ni lo vuelve a intentar en caso de conflicto de acceso con algún otro subproceso.

- **SkipList - insert**

```
template <typename ElemType>
bool SkipList<ElemType>::insert(ElemType value) {

    int top_level = random_level();
    vector<SkipListNode<ElemType>*> preds;
    vector<SkipListNode<ElemType>*> succs;
    int current_level = 0;

    while (true) {
        bool found = FindNode(value, preds, succs, current_level);
        if (found) {
            SkipListNode<ElemType>* SkipListNode_found = succs[current_level];
            if (!SkipListNode_found->marked) {
                while (!SkipListNode_found->fully_linked) {}
                return false;
            }
            continue;
        }
        int highestLocked = -1;
        SkipListNode<ElemType>* pred;
        SkipListNode<ElemType>* succ;
        SkipListNode<ElemType>* aux = nullptr;
        bool valid = true;

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

            if (pred != aux) {
                pred->lock();
                highestLocked = level;
                aux = pred;
            }

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

            SkipListNode<ElemType>* tmp = nullptr;
            for (int level = 0; level <= highestLocked; level++)
            {
                if (tmp == nullptr || tmp != preds[level])
                    preds[level]->unlock();
                tmp = preds[level];
            }
            continue;
        }
        SkipListNode<ElemType>* new_SkipListNode = new SkipListNode<ElemType>(
    top_level, value);

        for (int level = 0; level <= top_level; level++) {
            new_SkipListNode->next[level] = succs[level];
            preds[level]->next[level] = new_SkipListNode;
        }

        new_SkipListNode->fully_linked = true;
        SkipListNode<ElemType>* tmp = nullptr;

        for (int level = 0; level <= highestLocked; level++)
```

```
59          {
60              if (tmp == nullptr || tmp != preds[level])
61                  preds[level]->unlock();
62              tmp = preds[level];
63          }
64          return true;
65      }
66
67  }
```

Fig 10. Función Insertar

Antes de insertar un elemento se tiene que verificar si el elemento ya se encuentra en la SkipList y si el nodo está marcado, para verificar si el elemento esta presente o no en la SkipList se usa la función auxiliar *FindNode*, si el elemento ya está presente y el nodo no está marcado, entonces no insertamos el elemento ya que esta presente en la SkipList (no acepta repeticiones). Si el elemento está presente y el nodo no está completamente vinculado, entonces esperamos hasta que lo esté (no se necesita inserción). Si el elemento está presente y el nodo está marcado, significa que algún otro subproceso está en proceso de eliminar ese nodo, por lo que el subproceso que realiza la operación de insertar lo intenta después. Y si el elemento no está presente en la SkipList simplemente se continua con el método de inserción.

Para agregar el elemento después de la verificación anterior, encontramos referencias a predecesores y sucesores de la posición en la que este elemento debe insertarse en cada nivel. Estas referencias pueden estar dañadas en el momento en que realmente realizamos la inserción. Dado que cada nodo solo tiene un puntero al siguiente nodo, solo necesitaremos mantener el bloqueo del predecesor (línea 30) y no del sucesor. Pero debemos asegurarnos de que tanto el predecesor como el sucesor no estén marcados y que el siguiente del predecesor sea el sucesor en cada nivel (línea 35). En caso de que no se cumplan estas condiciones, se libera el bloqueo de los predecesores y se intenta realizar la operación de insertar después (líneas 37 a 47).

Si se cumplen las condiciones anteriores, se garantiza que la operación de inserción tendrá éxito porque el subproceso mantiene todos los bloqueos hasta que vincula completamente su nuevo nodo. En este caso, el subproceso asigna un nuevo nodo con el valor y el nivel generado aleatoriamente. Lo siguiente es vincular el nuevo nodo con sus sucesores y vincular los predecesores con el nuevo nodo, esto se tiene que realizar en todos los niveles de la estructura (líneas 50 a 53). Una vez que se haya terminado el proceso de vinculación, el nodo se marca como totalmente vinculado (fully_linked), luego liberamos todos los bloqueos de los predecesores en cada nivel (líneas 58 a 63) y por último retornará *Verdadero* si es que se insertó el elemento correctamente y caso contrario retornará *Falso*.

- **SkipList - remove**

```
1   template <typename ElemType>
2   bool SkipList<ElemType>::remove(ElemType value) {
3
4       SkipListNode<ElemType>* nodeToDelete = nullptr;
5       bool is_marked = false;
6       int top_level = -1;
7       vector<SkipListNode<ElemType>*> preds;
8       vector<SkipListNode<ElemType>*> succs;
9       int current_level = 0;
10
11      while (true) {
12          bool found = FindNode(value, preds, succs, current_level);
13          if (found) {
14              nodeToDelete = succs[current_level];
15          }
16
17          if (is_marked || (found && readyToDelete(succs[current_level],
    current_level))) {
18
19              if (!is_marked) {
20                  top_level = nodeToDelete->top_level;
21                  nodeToDelete->lock();
```

```
22                  if (nodeToDelete->marked) {
23                      nodeToDelete->unlock();
24                      return false;
25                  }
26                  nodeToDelete->marked = true;
27                  is_marked = true;
28              }
29
30              int highestLocked = -1;
31              SkipListNode<ElemType>* pred;
32              bool valid = true;
33              SkipListNode<ElemType>* tmp = nullptr;
34
35              for (int level = 0; valid && (level <= top_level); level++) {
36                  pred = preds[level];
37                  if (tmp != preds[level])
38                      pred->lock();
39                  tmp = preds[level];
40                  highestLocked = level;
41                  valid = !pred->marked && pred->next[level] == nodeToDelete;
42
43              }
44
45              if (!valid) {
46                  SkipListNode<ElemType>* tmp = nullptr;
47                  for (int level = 0; level <= highestLocked; level++)
48                  {
49                      if (tmp == nullptr || tmp != preds[level])
50                          preds[level]->unlock();
51                      tmp = preds[level];
52                  }
53                  continue;
54              }
55
56              for (int level = top_level; level >= 0; level--) {
57                  preds[level]->next[level] = nodeToDelete->next[level];
58              }
59
60              nodeToDelete->unlock();
61              tmp = nullptr;
62              delete nodeToDelete;
63
64              for (int level = 0; level <= highestLocked; level++) {
65                  if (tmp == nullptr || tmp != preds[level])
66                      preds[level]->unlock();
67                  tmp = preds[level];
68              }
69              return true;
70          }
71          else {
72              return false;
73          }
74      }
75  }
```

Fig 11. Función Eliminar

La operación de eliminar también llama a *FindNode* para determinar si ese elemento que deseamos eliminar está en la lista. Si es así, el subproceso verifica si el nodo está *listo para eliminar* (readyToDelete, ver **Figura 12**), lo que significa que está completamente vinculado, no marcado y se encontró en su nivel superior (línea 17). Si el nodo cumple con estos requisitos, se bloquea el nodo (línea 21) y se verifica que todavía no esté marcado. Si es así, se marca el nodo.

El resto del procedimiento logra la eliminación física, eliminando el nodo de la lista bloqueando primero sus predecesores en todos los niveles hasta el nivel del nodo eliminado (líneas 31 a 43), también es necesario verificar si el predecesor no está marcado y también si el siguiente elemento del predecesor es el elemento actual que estamos tratando de eliminar (línea 41). Si no se cumplen las condiciones, liberamos el bloqueo de los predecesores, también liberamos el bloqueo del elemento que se está eliminando e intentamos eliminar después (líneas 45 a 54).

Si se cumplen las condiciones anteriores, entonces se pasa a vincular los predecesores a los sucesores del nodo a eliminar (líneas 56 a 58). Una vez que se realiza la vinculación, se desbloquea el nodo, luego se elimina de la SkipList y por último se libera la memoria para ese nodo en particular (líneas 60 y 62). Después de eliminar el nodo, se liberan los bloqueos de todos los nodos predecesores retenidos (líneas 64 a 68) y por ultimo se retorna *Verdadero* indicando que se eliminó correctamente ese elemento.

Si no se encontró ningún nodo, o el nodo encontrado no estaba *listo para eliminar* (es decir, estaba marcado, no estaba completamente vinculado o no se encontró en su nivel superior), entonces la operación simplemente devuelve *Falso* (línea 72).

- **SkipList - readyToDelete**

```
1  template <typename ElemType>
2  bool SkipList<ElemType>::readyToDelete(SkipListNode<ElemType>* candidate, int
       lFound) {
3
4      return(candidate->fully_linked && candidate->top_level == lFound && !candidate
       ->marked);
5  }
```

Fig 12. Función auxiliar Ready To Delete

- **SkipList - search**

```
1  template <typename ElemType>
2  bool ConcurrentSkipList<ElemType>::search(ElemType value) {
3
4      vector<SkipListNode<ElemType>*> preds;
5      vector<SkipListNode<ElemType>*> succs;
6      int current_level;
7      bool found = FindNode(value, preds, succs, current_level);
8      return(found && succs[current_level]->fully_linked && !succs[current_level]->
       marked);
9  }
```

Fig 13. Función Buscar

Luego tenemos la función buscar que simplemente llama a la función auxiliar *FindNode* para ver si existe el elemento que deseamos buscar, también es necesario verificar que el nodo no esté marcado y que esté completamente vinculado. Si cumple estas 3 condiciones entonces retornará *Verdadero* y en caso que falle alguna de ellas retorna *Falso*. La desventaja de esta búsqueda es que siempre se va a tener que bajar al primer nivel de la SkipList así se haya encontrado el elemento en el nivel más alto, entonces esto aumenta el tiempo al realizar dicha búsqueda.

- **SkipList - optimized_search**

```
1  template <typename ElemType>
2  bool ConcurrentSkipList<ElemType>::optimized_search(ElemType value) {
3
4      SkipListNode<ElemType>* p = head;
5      SkipListNode<ElemType>* current_node = nullptr;
6      bool found = false;
7      for (int i = max_level; i >= 0; i--) {
8          current_node = p->next[i];
9          while (current_node && current_node->value < value) {
10             p = current_node;
11             current_node = p->next[i];
12         }
13         if (current_node && current_node->value == value) {
14             found = true;
15             break;
16         }
17     }
18     return(found && current_node->fully_linked && !current_node->marked);
19 }
```

Fig 14. Función Buscar Optimizado

Por ultimo tenemos la función de Búsqueda Optimizada, que busca desde el máximo nivel de la SkipList si el elemento se encuentra, si no se encuentra en el máximo nivel baja un nivel y así sucesivamente hasta llegar al primer nivel, apenas encuentre el elemento a buscar entonces ya no va a ser necesario seguir bajando en todos los niveles (líneas 13 a 16), luego tenemos que verificar dos cosas, que el nodo no esté marcado y que el nodo este completamente vinculado, si cumple estas condiciones retornará *Verdadero*, y en caso de que el nodo no se haya encontrado o que el nodo este marcado o no esté completamente vinculado retornará *Falso*.

3. Experimentos

En esta sección se realizarán diferentes experimentos para ver el comportamiento de ambas implementaciones de la SkipList. Para estos experimentos se usó una laptop Asus Rog Strix G15 con un procesador AMD Ryzen 5 5600H con 16 gigas de memoria ram, y se usó 12 threads que es lo máximo que soporta este equipo. Además, se usó Visual Studio Community como IDE y como compilador se usó Microsoft Visual c++ (MSVC).

3.1. Rendimiento de la SkipList Concurrente

Antes de pasar a la *Medición de Tiempos* es necesario saber cuál es el comportamiento de la Skiplist Concurrente al usar cierta cantidad de threads y saber si es que a medida que aumenta la cantidad de threads el tiempo al realizar las operaciones disminuye. Para esto se realizará una comparación de tiempo usando cierta cantidad de threads y usando un millón de elementos a la hora de realizar la inserción, búsqueda y eliminación. Los resultados obtenidos fueron los siguientes:

Cantidad de elementos	**Número de threads**	**Tiempo insert**	**Tiempo search**	**Tiempo remove**
1000000	4	2.56554	0.519264	2.09633
1000000	8	3.35813	0.295870	3.61879
1000000	12	6.57585	0.276841	5.72240

Cuadro 1: Comparación de tiempos usando un millón de elementos

Como se puede observar en la tabla anterior, a medida que aumentan los threads también aumenta el tiempo al momento de *insertar* y *eliminar* a pesar de tener la misma cantidad de elementos. Esto se debe a que hay mas threads que compiten por los bloqueos de varios nodos en diferentes niveles de la SkipList. También se puede observar que la *búsqueda* lleva menos tiempo ya que varios threads pueden hacer esta operación en paralelo debido a que no hay bloqueos.

3.2. Medición de Tiempos

En esta subsección se realizará una comparación en base al tiempo que se demora en insertar, eliminar y buscar elementos en ambas implementaciones de la SkipList. Para que la comparación sea justa, se llenó de manera aleatoria 3 vectores para la inserción, eliminación y búsqueda. Y se tomó el tiempo que se demoran ambas implementaciones. Los datos para realizar los experimentos fueron 10000, 100000, 1000000 y 10000000 de elementos. Los resultados obtenidos al realizar dichas comparaciones fueron la siguientes:

3.2.1. Insertar

Cantidad de elementos	Insertar Paralelo	Insertar Secuencial
10000	0.06625	0.15395
100000	0.69861	1.33674
1000000	6.76409	15.4817
10000000	61.2112	168.419

Cuadro 2: Comparación de tiempos al usar la función Insertar

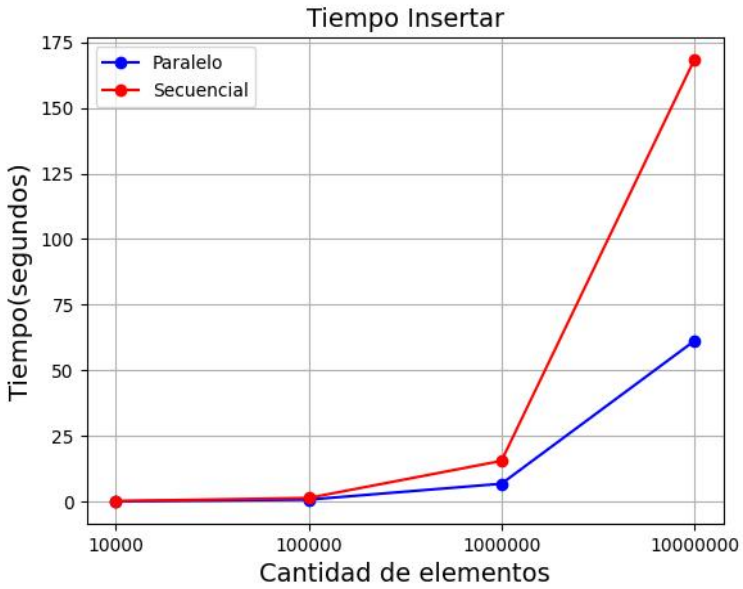

Figura 1: Grafica de tiempo al insertar.

3.2.2. Buscar

Cantidad de elementos	Buscar Paralelo	Buscar Secuencial
10000	0.005893	0.008162
100000	0.015540	0.10906
1000000	0.288534	2.635630
10000000	4.648660	52.45890

Cuadro 3: Comparación de tiempos al usar la función Buscar

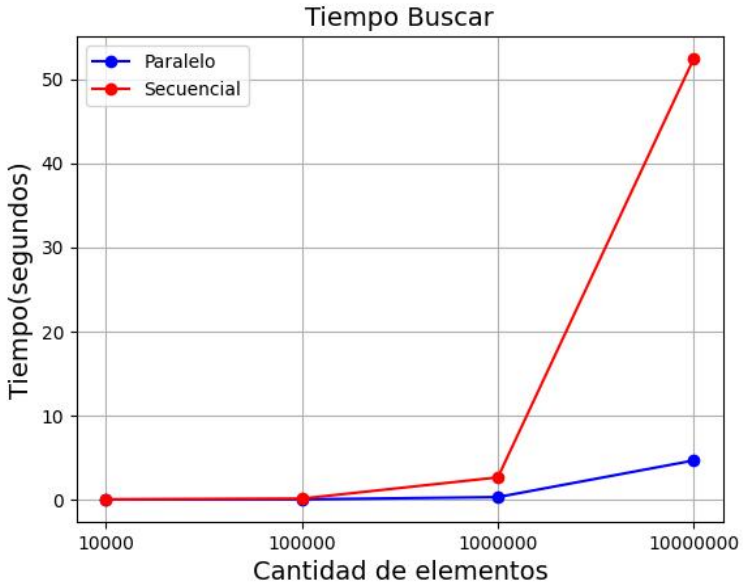

Figura 2: Grafica de tiempo al buscar.

3.2.3. Eliminar

Cantidad de elementos	Eliminar Paralelo	Eliminar Secuencial
10000	0.056533	0.086639
100000	0.568464	0.780972
1000000	5.261920	9.42756
10000000	55.70230	125.136

Cuadro 4: Comparación de tiempos al usar la función Eliminar

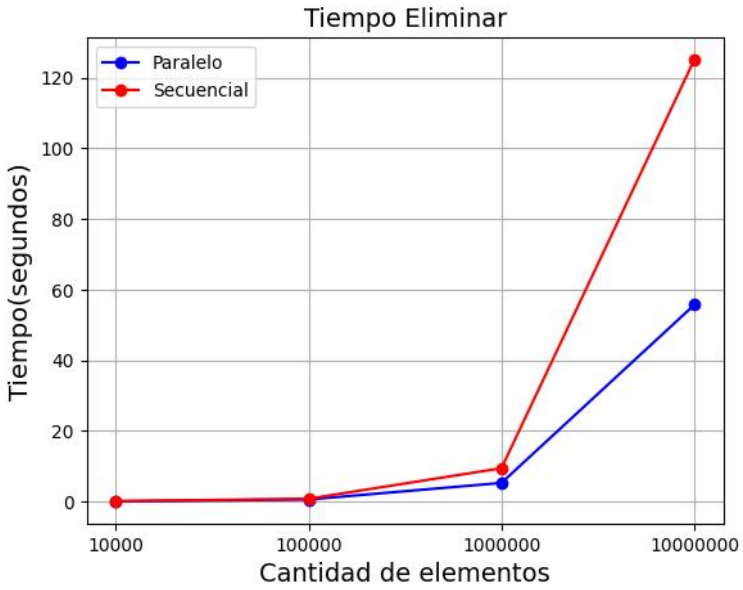

Figura 3: Grafica de tiempo al eliminar.

3.3. Pruebas de Memoria

En esta subsección se realizará una comparación de uso de memoria en ambas implementaciones de la SkipList. Por motivos del hardware utilizado para las ejecuciones de estas pruebas se utilizaron las librerías *"windows.h"* y *"psapi.h"*, que solo se encuentran disponibles para el sistema operativo de Windows. Estas nos permiten medir la memoria utilizada al ejecutar el programa. Para que la diferencia sea notoria utilizamos 10 millones de elementos, de este modo podemos hacer una comparación porcentual de cuánta memoria adicional es requerida para la implementación concurrente.

El siguiente bloque de código fue el utilizado para obtener la memoria utilizada por el programa.

```
void GetCurrentRam() {

    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);
    DWORDLONG totalVirtualMem = memInfo.ullTotalPageFile;
    DWORDLONG virtualMemUsed = memInfo.ullTotalPageFile - memInfo.ullAvailPageFile;
    DWORDLONG totalPhysMem = memInfo.ullTotalPhys;
    DWORDLONG physMemUsed = memInfo.ullTotalPhys - memInfo.ullAvailPhys;
    PROCESS_MEMORY_COUNTERS_EX pmc;
    GetProcessMemoryInfo(GetCurrentProcess(), (PROCESS_MEMORY_COUNTERS*)&pmc, sizeof(
    pmc));
    SIZE_T virtualMemUsedByMe = pmc.PrivateUsage;
    SIZE_T physMemUsedByMe = pmc.WorkingSetSize;

    cout << "Memoria Ram Maxima: " << totalPhysMem / (1024 * 1024) << endl;
    cout << "Memoria Ram Consumida: " << physMemUsed / (1024 * 1024) << endl;
    cout << "RAM utilizada actualmente por mi proceso: " << physMemUsedByMe / (1024 *
    1024) << endl;
}
```

Fig 15. Uso de memoria Ram

Tras haber realizado las pruebas en ambas implementaciones se obtiene que con 10 millones de elementos la implementación secuencial utiliza 2523 MB y la implementación concurrente 3224 MB. Esto implica que la SkipList concurrente requiere un aproximado de 27.78 % adicional de memoria.

4. Conclusiones

Como se pudo observar en la sección de *Implementación Computacional*, se tienen que hacer varias modificaciones para poder adaptar la SkipList Secuencial a su forma Concurrente, siendo más complejo realizar estas modificaciones. También se puede observar que en la sección de *Experimentos* en *Rendimiento de la SkipList Concurrente* a medida que aumentan los threads en el caso de las funciones de insertar y eliminar el tiempo también aumentará, esto se debe a la contienda que se produce cuando varios subprocesos intentan competir en los recursos, ya que muchos intentan realizar modificaciones en la SkipList y todos intentan adquirir el bloqueo de los nodos en todos los niveles que intentan insertar o eliminar. En cambio, cuando llamamos a la función de buscar vemos que el tiempo no es tan elevado, debido a que esta función no tiene bloqueos y lo único que tiene que hacer es buscar el elemento y hacer una pequeña validación.

En la subsección de *Medición de Tiempos* al momento de insertar, buscar y eliminar con pocos elementos no se nota mucho la diferencia respecto al tiempo, pero cuando la cantidad de elementos es elevada ya vemos la eficiencia de la implementación concurrente, es por eso que a partir de un millón de elementos en la gráfica se ve como crece la implementación secuencial respecto al tiempo, mientras que la implementación concurrente se mantiene baja gracias a que se realizan las operaciones en paralelo. Aquí es donde podemos observar la principal ventaja de las implementaciones en paralelo.

Por último, en las *Pruebas de Memoria* se obtuvo que se requiere un aproximado de 28 % de memoria adicional, lo cual tras analizar las variables adicionales utilizadas para cada elemento del SkipList es el resultado esperado. Esto viene a ser la principal desventaja de las implementaciones concurrentes de estructuras de datos, ya que requerimos de variables y condiciones adicionales para su correcto funcionamiento.

Referencias

[1] W. Pugh, "Concurrent maintenance of skip lists," 1998.

[2] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A Simple Optimistic Skiplist Algorithm," in Structural Information and Communication Complexity, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 124–138.

[3] Fraser, K., Harris, T.: Concurrent programming without locks. Unpublished manuscript (2004)

[4] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit, "A lazy concurrent list-based set algorithm," in Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 3–16.

[5] T. Crain, V. Gramoli, and M. Raynal, "No Hot Spot non-blocking skip list," in 2013 IEEE 33rd International Conference on Distributed Computing Systems, 2013, pp. 196–205.

[6] [1] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in Lecture Notes in Computer Science, Cham: Springer International Publishing, 2013, pp. 206–220.

[7] K. Platz, N. Mittal, and S. Venkatesan, "Concurrent Unrolled Skiplist," in 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1579–1589.

[8] I. Dick, A. Fekete, and V. Gramoli, "A skip list for multicore: A SKIP LIST FOR MULTICORE," Concurr. Comput., vol. 29, no. 4, p. e3876, 2017.