Performance Oriented Computing - Sheet 10

Tobias Beiser Andreas Kirchmair Johannes Karrer

June 5, 2024

1 Benchmark-Setup

The method runBenchmarks() runs the benchmark with for a certain collectionSize, readPercentage and insertPercentage. The TieredArray is benchmarked with chunkSizes 2, 4 ,8 and 16, while the UnrolledLinkedList is benchmarked with chunkSizes 8, 16 and 32.

```
template <typename T>
void runBenchmarks(int collectionSize, int readPercentage, int insertPercentage, int benchmarkTime,
bool printInfo)
{
        if (printInfo)
        {

                std::cout << "Running benchmarks for element size: " << sizeof(T) << " bytes" <<
                std::endl;
                std::cout << "Collection size: " << collectionSize << std::endl;
                std::cout << "Read percentage: " << readPercentage << "%" << std::endl;
                std::cout << "Insert percentage: " << insertPercentage << "%" << std::endl;
                std::cout << "--------------------------------" << std::endl;
        }

        std::vector<unique_ptr<Benchmark>> benchmarks;
        benchmarks.push_back(std::make_unique<ArrayBenchmark<T>>(benchmarkTime, collectionSize,
        readPercentage, insertPercentage, "std::vector"));
        benchmarks.push_back(std::make_unique<ArrayRandomAccessBenchmark<T>>(benchmarkTime,
        collectionSize, readPercentage, insertPercentage, "std::vector_random"));
        benchmarks.push_back(std::make_unique<ListBenchmark<T>>(benchmarkTime, collectionSize,
        readPercentage, insertPercentage, "std::forward_list"));
        benchmarks.push_back(std::make_unique<ListRandomAccessBenchmark<T>>(benchmarkTime,
        collectionSize, readPercentage, insertPercentage, "std::forward_list_random"));
        for (int i = 2; i <= 16; i *= 2) {
                benchmarks.push_back(std::make_unique<TieredArrayBenchmark<T>>(benchmarkTime,
                collectionSize, i, readPercentage, insertPercentage, "TieredArrayChunkSize" +
                std::to_string(i)));
                benchmarks.push_back(std::make_unique<TieredArrayRandomBenchmark<T>>(benchmarkTime,
                collectionSize, i, readPercentage, insertPercentage, "TieredArrayRandomChunkSize" +
                std::to_string(i)));
        }
        for (int i = 8; i <= 32; i *= 2) {
                benchmarks.push_back(std::make_unique<UnrolledLinkedListBenchmark<T>>(benchmarkTime,
                collectionSize, i, readPercentage, insertPercentage, "UnrolledLinkedListChunkSize" +
                std::to_string(i)));
                benchmarks.push_back(std::make_unique<UnrolledLinkedListRandomBenchmark<T>>
                benchmarkTime, collectionSize, i, readPercentage, insertPercentage,
                "UnrolledLinkedListRandomChunkSize" + std::to_string(i)));
```

```
        }

        // run all the benchmarks
        for (auto& benchmark : benchmarks) {
                benchmark->runBenchmark();
        }
}
```

The following elements are covered for the benchmark:

```
#pragma once

struct Element8Bytes
{
        char data[8];
};

struct Element512Bytes
{
        char data[512];
};

struct Element8MB
{
        char data[8 * 1024 * 1024];
};
```

For a standardized benchmark setup, we created a template, that was adjusted and used for every collection:

```
#include "YourDataStructureBenchmark.hpp"

template void ArrayBenchmark<Element8Bytes>::runBenchmark();
template void ArrayBenchmark<Element512Bytes>::runBenchmark();
template void ArrayBenchmark<Element8MB>::runBenchmark();

template <typename T>
void YourDataStructureBenchmark<T>::runBenchmark()
{
        //TODO: Initialize your collection here
        std::vector<T> collection(this->collectionSize);

        // reset counters before every benchmark to ensure correct stats
        this->resetCounters();

        const auto end = std::chrono::high_resolution_clock::now() +
        std::chrono::seconds(this->runtime);

        bool run = true;

        while (run)
        {
                //TODO: Depending on your collection type change the iterator
                for (int i = 0; i < this->collectionSize;)
                {
                        if (std::chrono::high_resolution_clock::now() > end)
```

```
                        {
                                run = false;
                                break;
                        }

                        //TODO: Add your implementation here
                        //Read/Writes for range this->getPadding1()
                        //1xInsert
                        //Read/Writes for range this->getPadding2()
                        //1xDelete
                        //make sure to check the outer loop variable before executing any of these
                        //actions
                        //make sure to increment the outer variable and the corresponding counter
                        //after each action
                        //pay attention to create the objects for insertions directly on the stack
                }
        }
        // always print results in the end to be read by the python script
        this->printResults();
}
```

2 Task 1: Unrolled Linked Lists

Implementation of Unrolled Linked List:

```
template <typename T>
struct UnrolledLinkedListNode {
        std::vector<std::shared_ptr<T>> elements;
        std::shared_ptr<UnrolledLinkedListNode<T>> next;

        explicit UnrolledLinkedListNode(size_t chunkSize) : elements(), next(nullptr) {
                elements.reserve(chunkSize);
        }
};

template <typename T>
class UnrolledLinkedList {
        public:
        UnrolledLinkedList(size_t initialSize, size_t chunkSize)
        : chunkSize(chunkSize) {
                size_t numChunks = (initialSize + chunkSize - 1) / chunkSize;
                head = std::make_shared<UnrolledLinkedListNode<T>>(chunkSize);
                auto current = head;
                for (size_t i = 0; i < numChunks - 1; ++i) {
                        for (int j = 0; j < chunkSize; ++j) {
                                current->elements.emplace_back(std::make_shared<T>());
                        }
                        current->next = std::make_shared<UnrolledLinkedListNode<T>>(chunkSize);
                        current = current->next;
                }
                for (int i = 0; i < initialSize % chunkSize; ++i) {
                        current->elements.emplace_back(std::make_shared<T>());
                }
                current->next = nullptr;
        }
```

```
        void read(std::shared_ptr<UnrolledLinkedListNode<T>>& node, size_t& chunkIndex) {
                if (chunkIndex < node->elements.size()) {
                        char data = node->elements[chunkIndex]->data[0];
                        data++;
                }
        }

        void write(std::shared_ptr<UnrolledLinkedListNode<T>>& node, size_t& chunkIndex) {
                if (chunkIndex < node->elements.size()) {
                        node->elements[chunkIndex]->data[0] = 0;
                }
        }

        std::shared_ptr<UnrolledLinkedListNode<T>> insert
        (std::shared_ptr<UnrolledLinkedListNode<T>>& node, size_t& chunkIndex) {
                if (node->elements.size() < chunkSize) {
                        node->elements.insert(node->elements.begin() + chunkIndex,
                        std::make_shared<T>());
                        chunkIndex++;
                        return node;
                }
                else {
                        auto newNode = std::make_shared<UnrolledLinkedListNode<T>>(chunkSize);
                        size_t halfSize = node->elements.size() / 2;
                        newNode->elements.insert(newNode->elements.end(),
                        std::make_move_iterator(node->elements.begin() + halfSize),
                        std::make_move_iterator(node->elements.end()));
                        node->elements.erase(node->elements.begin() + halfSize,
                        node->elements.end());
                        newNode->elements.insert(newNode->elements.end(), std::make_shared<T>());
                        newNode->next = node->next;
                        node->next = newNode;
                        chunkIndex = newNode->elements.size() - 1;
                        return newNode;
                }
        }

        void remove(std::shared_ptr<UnrolledLinkedListNode<T>>& node, size_t& chunkIndex) {
                if (!node->elements.empty()) {
                        node->elements.pop_back();
                }
                if (chunkIndex >= node->elements.size()) {
                        node = node->next;
                        chunkIndex = 0;
                }
        }
        std::shared_ptr<UnrolledLinkedListNode<T>> head;
        private:
        size_t chunkSize;
};
```

While the read and write methods are quite straight-forward, the insert method always created a new node, if the one it tries to insert into is full and moves half of the elements of this node to the newly created node.

3 Task 2: Tiered Arrays

Implementation of TieredArray:

```
template<typename T>
class TieredArray {
        public:
        TieredArray(size_t n, size_t chunkSize)
        : totalElements(n), chunkSize(chunkSize) {
                //        Initialize Layers
                size_t size = n + 1;
                size_t numChunks = ceil(size / (double)chunkSize);
                tiers = std::vector<std::vector<T>>(numChunks);
                for (int i = 0; i < numChunks; i++) {
                        tiers[i] = std::vector<T>(chunkSize);
                }
        }
        virtual ~TieredArray() {}

        void insert(size_t i) {
                T* value = new T();
                size_t chunkIndex = i / chunkSize;
                size_t position = i % chunkSize;
                T* previous = value;
                T* current = &(tiers[chunkIndex][position]);
                //        chunk with element
                for (size_t k = position; k < tiers[chunkIndex].size(); k++) {
                        current = &(tiers[chunkIndex][k]);
                        tiers[chunkIndex][k] = *previous;
                        previous = current;
                }
                for (size_t j = chunkIndex + 1; j < tiers.size(); j++) {
                        for (int k = 0; k < tiers[j].size(); k++) {
                                current = &(tiers[j][k]);
                                tiers[j][k] = *previous;
                                previous = current;
                        }
                }
                delete value;
                totalElements++;
        }

        void del(size_t i) {
                size_t chunkIndex = i / chunkSize;
                size_t position = i % chunkSize;
                T* current = &(tiers.back().back());
                //        take some trash value to set for the last value (which is officially no par
                T* next = &(tiers.back().back());
                for (size_t j = tiers.size() - 1; j > chunkIndex; j--) {
                        for (int k = tiers[j].size() - 1; k >= 0; k--) {
                                current = &(tiers[j][k]);
                                tiers[j][k] = *next;
                                next = current;
                        }
                }
                //        chunk with element
```

```
                for (int k = tiers[chunkIndex].size() - 1; k >= position && k>=0; k--) {
                        current = &(tiers[chunkIndex][k]);
                        tiers[chunkIndex][k] = *next;
                        next = current;
                }
                totalElements--;
        }

        T& at(size_t i) {
                return tiers[i / chunkSize][i % chunkSize];
        }

        size_t size() const {
                return totalElements;
        }

        private:
        size_t chunkSize;
        size_t totalElements = 0;
        std::vector<std::vector<T>> tiers;
};
```

4 Task 3: Extended Benchmarking

To implement random access, we also adjusted the template shown in section 1: Benchmark Setup.
Random access for arrays:

```
template <typename T>
void ArrayRandomAccessBenchmark<T>::runBenchmark()
{
        this->resetCounters();
        auto collection = std::make_unique<std::vector<std::unique_ptr<T>>>();
        collection->resize(this->collectionSize + 1);
        for (int i = 0; i < this->collectionSize + 1; i++)
        {
                collection->at(i) = std::make_unique<T>();
        }
        std::vector<int> randomIndices(collectionSize + 1);
        for (int i = 0; i < collectionSize + 1; i++)
        {
                randomIndices.at(i) = i;
        }
        auto rng = std::default_random_engine{};
        std::shuffle(randomIndices.begin(), randomIndices.end(), rng);


        const auto end = std::chrono::high_resolution_clock::now() +
        std::chrono::seconds(this->runtime);
        bool run = true;
        while (run)
        {
                for (auto it = randomIndices.begin(); it != randomIndices.end();)
                {
                        if (std::chrono::high_resolution_clock::now() > end)
                        {
```

```
                run = false;
                break;
        }

        for (int k = 0; k < this->getPadding1() && it != randomIndices.end(); k += 2)
        {
                // read
                char data = collection->at(*it)->data[0];
                data++;
                ++it;
                this->readWriteOperations++;
                if (it == randomIndices.end())
                {
                        break;
                }

                // write
                collection->at(*it)->data[0] = 0;
                ++it;
                this->readWriteOperations++;
        }



        if (this->insertPercentage > 0 && it != randomIndices.end())
        {
                // insert
                collection->emplace(collection->begin()+ *it, std::make_unique<T>());
                this->insertDeleteOperations++;
                ++it;
        }

        for (int k = 0; k < this->getPadding2() && it != randomIndices.end(); k += 2)
        {
                // read
                char data = collection->at(*it)->data[0];
                data++;
                ++it;
                this->readWriteOperations++;
                if (it == randomIndices.end())
                {
                        break;
                }
                // write
                collection->at(*it)->data[0] = 0;
                ++it;
                this->readWriteOperations++;
        }



        if (this->insertPercentage > 0 && it != randomIndices.end())
        {
                // delete
                collection->erase(collection->begin() + *it);
```

```
                                ++it;
                                this->insertDeleteOperations++;
                        }

                }
        }
        this->printResults();
}
```

Random access for lists:

```
template <typename T>
void ListRandomAccessBenchmark<T>::runBenchmark()
{
        this->resetCounters();
        // to simulate random access, we store the nodes in a vector and shuffle them
        // we then iterate over the vector and get a different starting node for each iteration
        // this way the overhead for generating the random access pattern is not included in the
        //benchmark
        std::forward_list<T> collection(collectionSize);
        std::vector<typename std::forward_list<T>::iterator> nodes;
        nodes.reserve(this->collectionSize);
        for (typename std::forward_list<T>::iterator it = collection.begin();
                it != collection.end(); ++it)
        {
                nodes.push_back(it);
        }
        auto rng = std::default_random_engine{};
        std::shuffle(nodes.begin(), nodes.end(), rng);


        const auto end = std::chrono::high_resolution_clock::now() +
        std::chrono::seconds(this->runtime);

        bool run = true;
        while (run)
        {
                for (int i = 0; i < nodes.size();)
                {
                        auto it = nodes[i];
                        if (std::chrono::high_resolution_clock::now() > end)
                        {
                                run = false;
                                break;
                        }

                        for (int k = 0; k < this->getPadding1() && it != collection.end(); k += 2)
                        {
                                // read
                                char data = (*it).data[0];
                                this->readWriteOperations++;
                                data++;
                                it++;
                                if (it == collection.end())
                                {
                                        break;
```

```
                        }

                        // write
                        (*it).data[0] = 0;
                        this->readWriteOperations++;
                        it++;
                }


                if (this->insertPercentage > 0)
                {
                        // insert
                        collection.emplace_front();
                        this->insertDeleteOperations++;
                        if (it != collection.end())
                        {
                                it++;
                        }
                }

                for (int k = 0; k < this->getPadding2() && it != collection.end(); k += 2)
                {
                        // read
                        char data = (*it).data[0];
                        this->readWriteOperations++;
                        data++;
                        it++;

                        if (it == collection.end())
                        {
                                break;
                        }

                        // write
                        (*it).data[0] = 0;
                        this->readWriteOperations++;
                        it++;
                }

                if (this->insertPercentage > 0 && it != collection.end())
                {
                        // delete
                        collection.pop_front();
                        this->insertDeleteOperations++;
                        if (it != collection.end())
                        {
                                it++;
                        }
                }
        }
    }
    this->printResults();
}
```

5 Results

Since the result plots with this amount of different configurations were quite a lot, we will only show some of them is this result section:

5.1 keeping element size constant to 512 and nElements constant to 1000

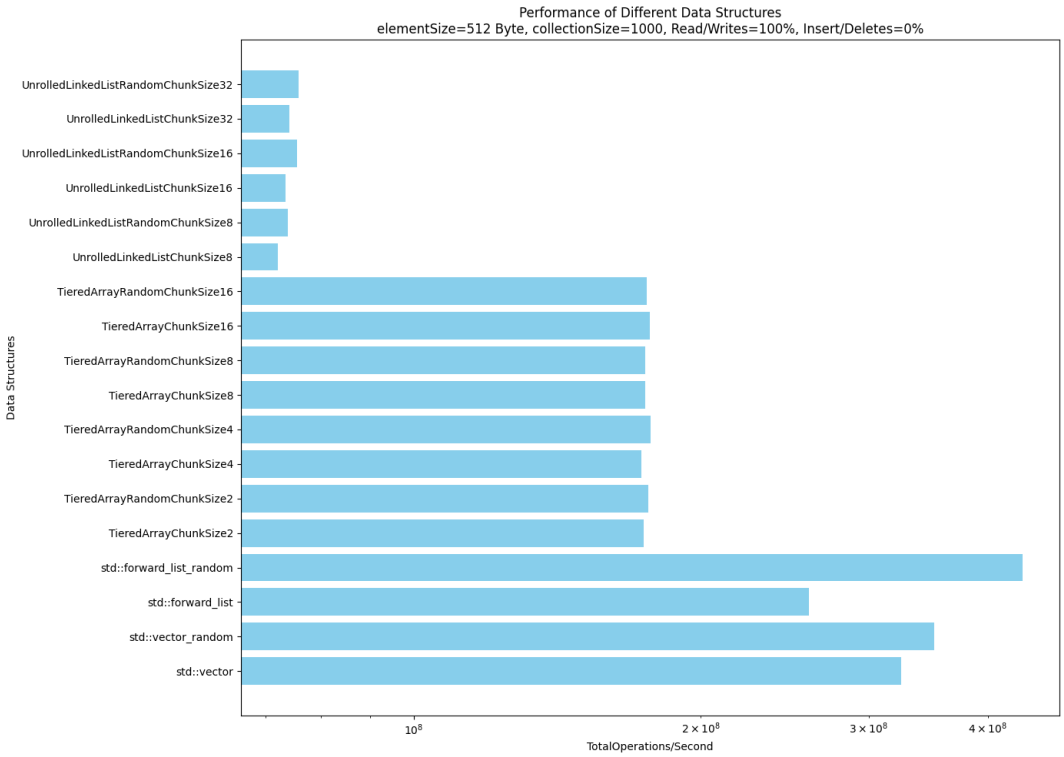

Figure 1: size=512, nElements=1000, RW=100, ID=0

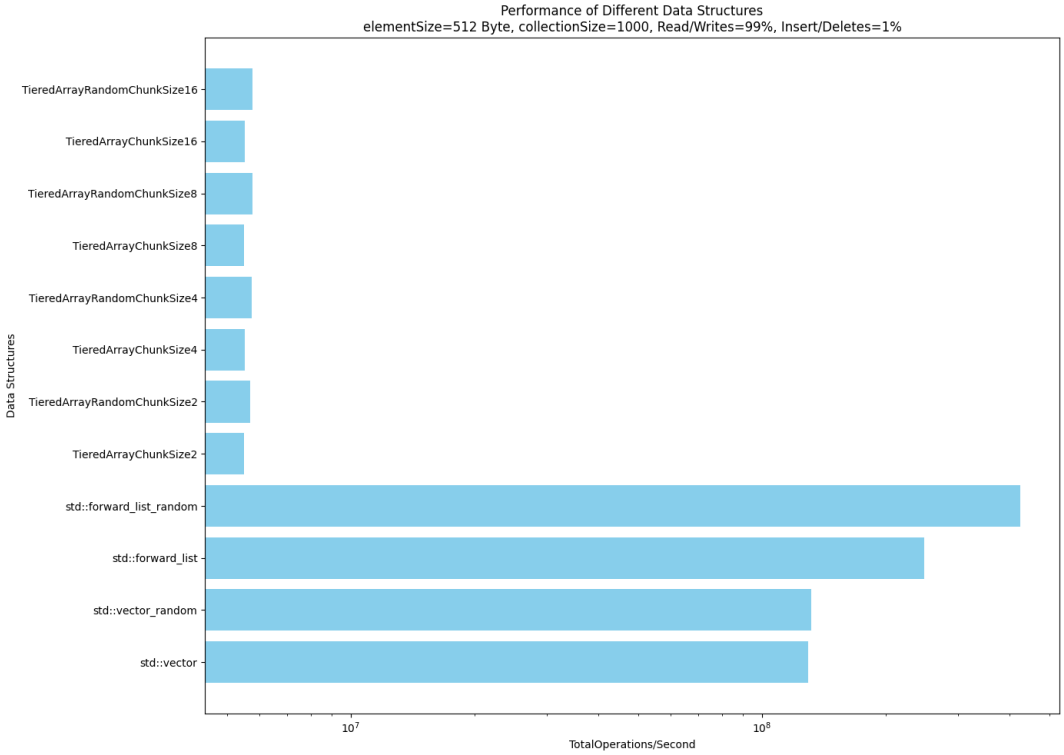

Figure 2: size=512, nElements=1000, RW=99, ID=1

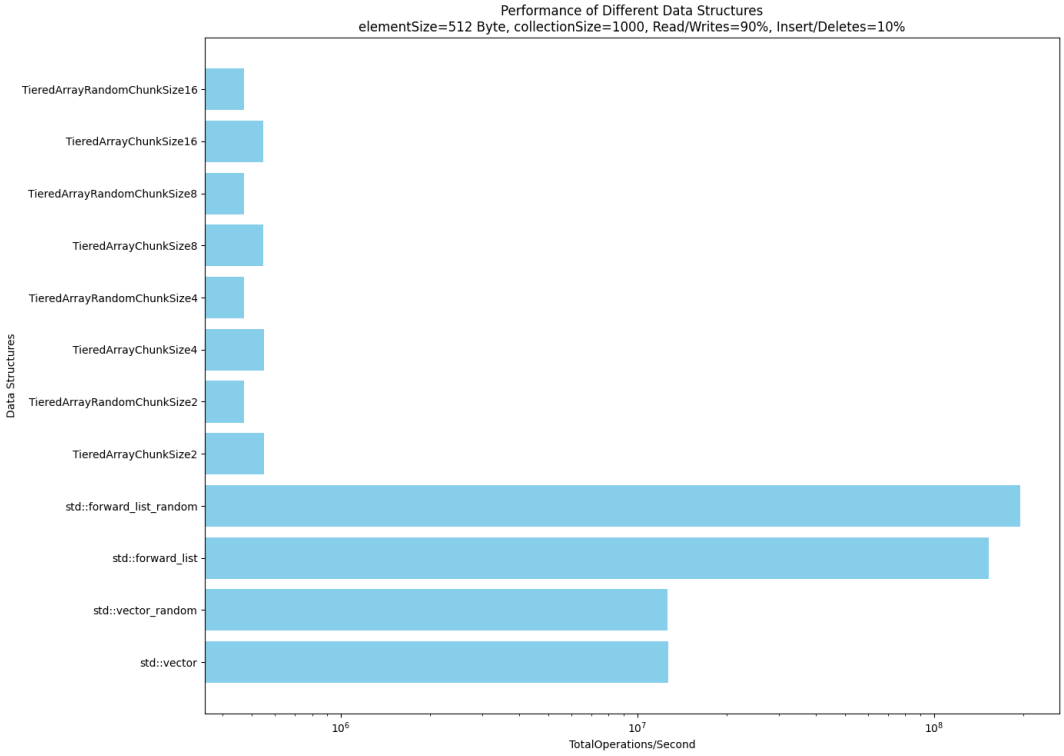

Figure 3: size=512, nElements=1000, RW=90, ID=10

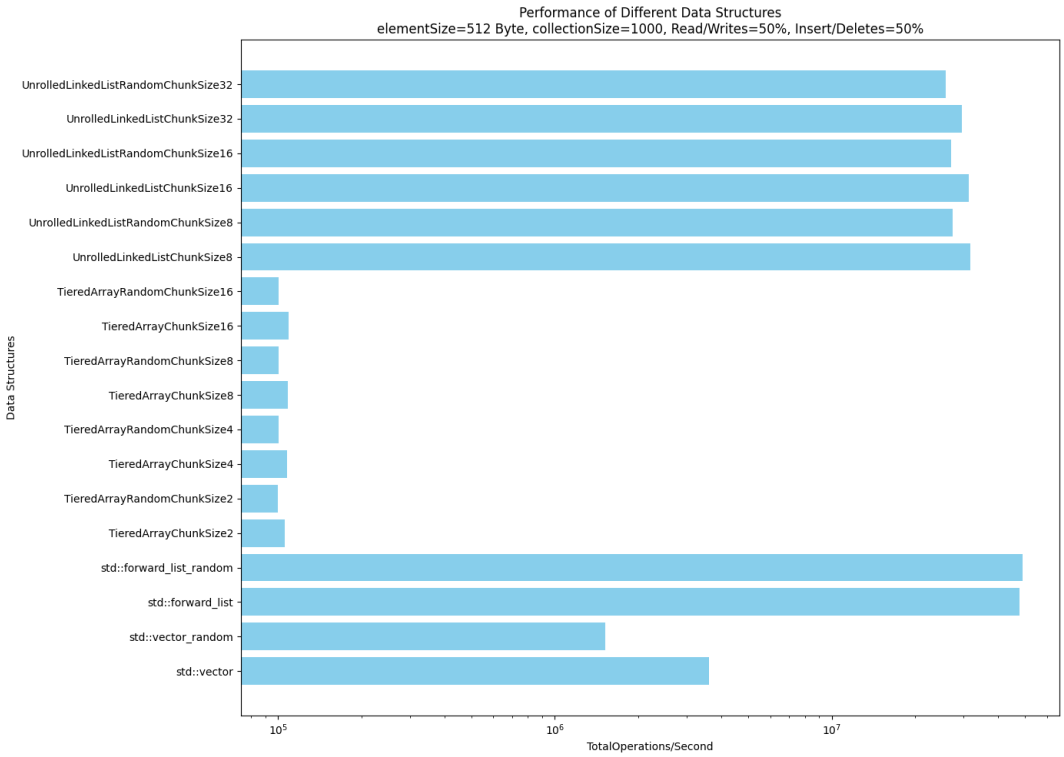

Figure 4: size=512, nElements=1000, RW=50, ID=50

5.2 keeping element size constant to 512 and nElements constant to 100000

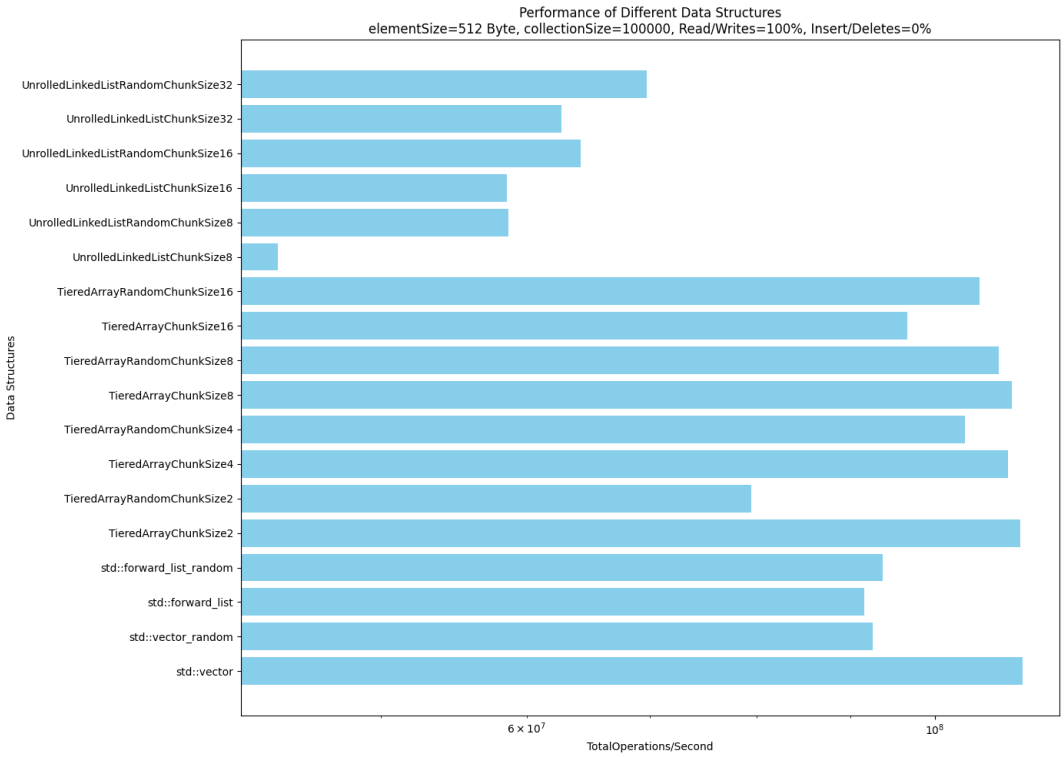

Figure 5: size=512, nElements=100000, RW=100, ID=0

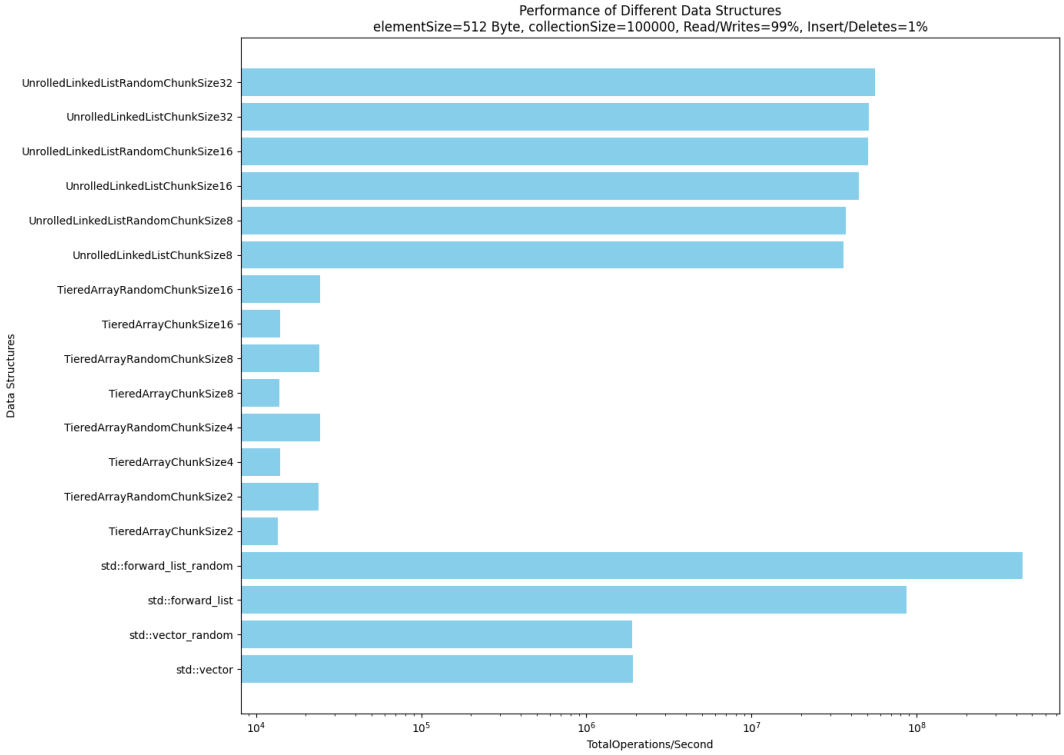

Figure 6: size=512, nElements=100000, RW=99, ID=1

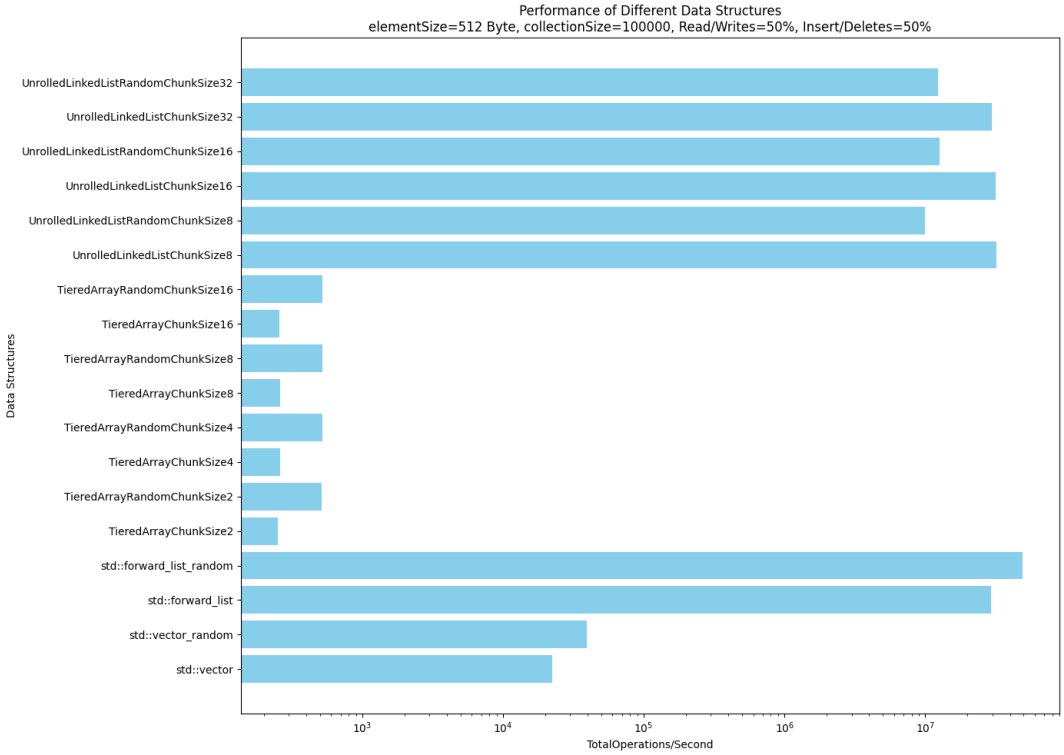

Figure 7: size=512, nElements=100000, RW=50, ID=50

5.3 keeping element size constant to 8MB and nElements constant to 1000

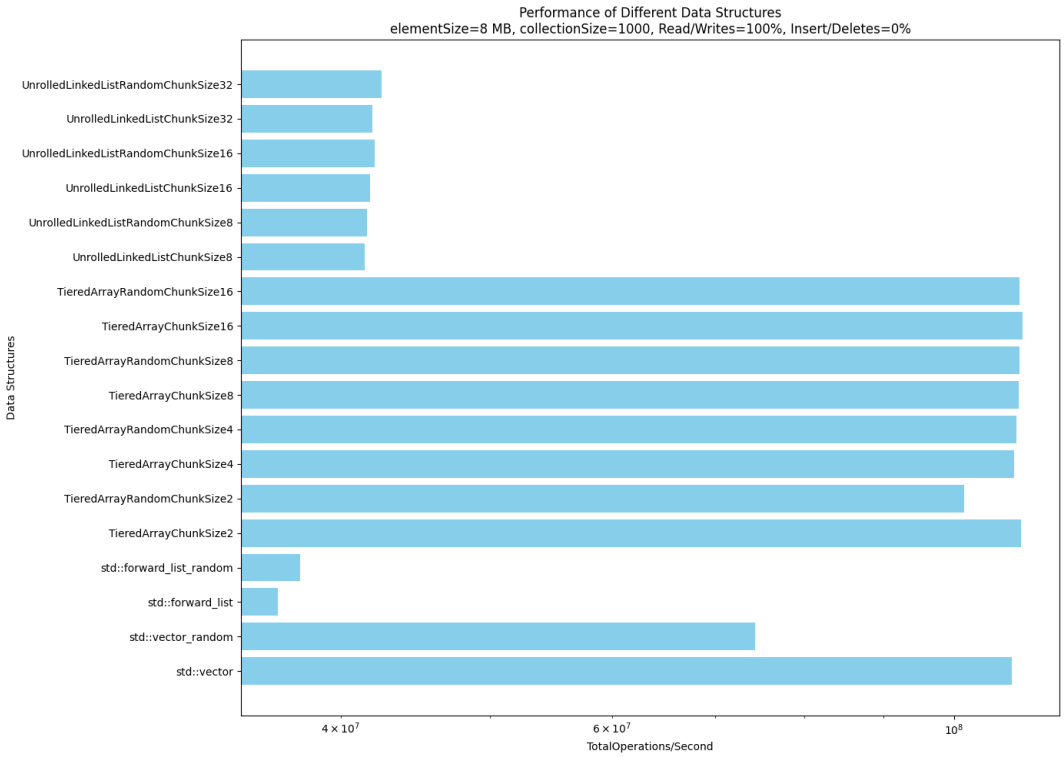

Figure 8: size=8MB, nElements=1000, RW=100, ID=0

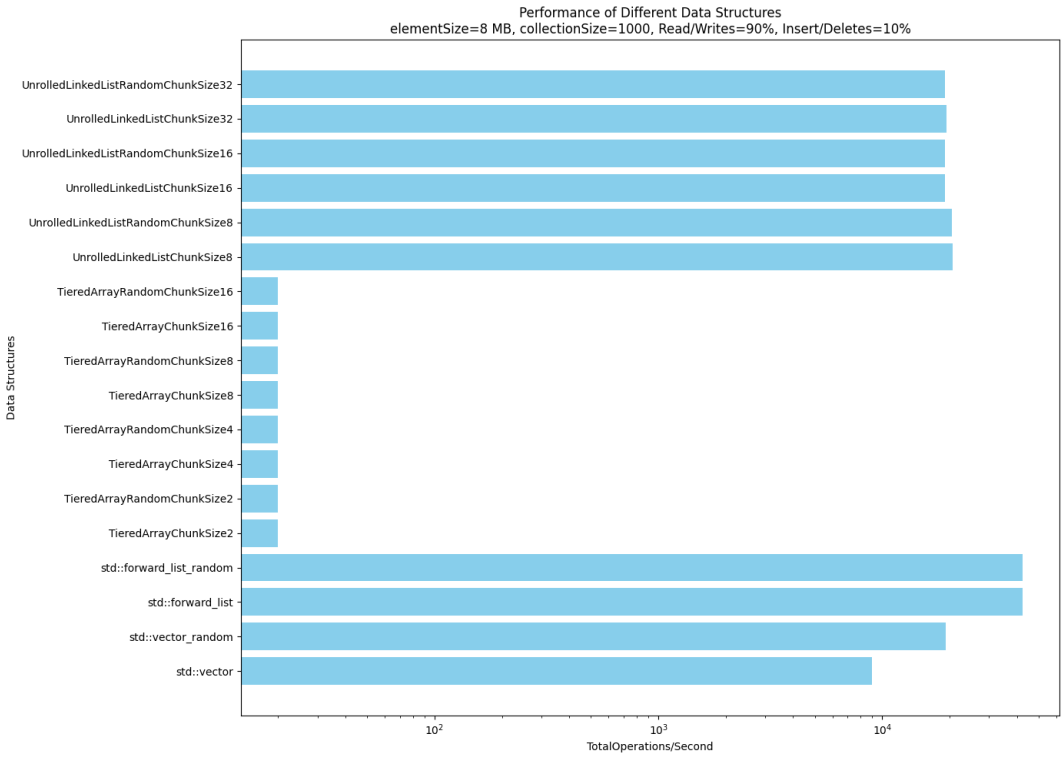

Figure 9: size=8MB, nElements=1000, RW=90, ID=10

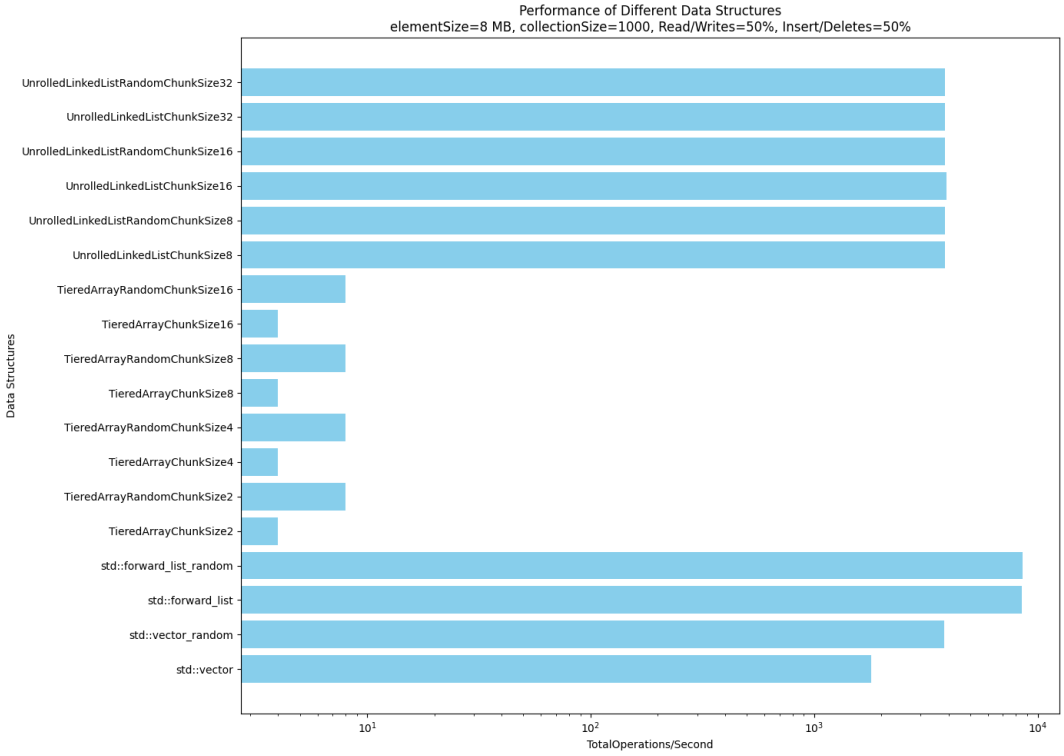

Figure 10: size=8MB, nElements=1000, RW=50, ID=50

6 Interpretation

While UnrolledLinkedList is comparably fast to LinkedList and Array in scenarios with a high ration of INSERT/DELETE operations, TieredArray is only fast in scenarios with a 100% READ/WRITE. It also seems like the chunk size doesn't really matter neither for UnrolledLinkedList nor TieredArray.

The random access patterns implemented in task3 tend to slow down execution time for Arrays, UnrolledLinkedList and TieredArrays. Looking at the results, it is interesting to see that random access patterns slightly increase the operations per second for LinkedList though.