CONTENTS

INTRODUCTION

Concurrent algorithms and data structures are nowadays inevitable in modern state-of-the-art software. The use of parallelism makes it possible to significantly increase the performance of programs on multicore machines, as it allows the simultaneous execution of several tasks.

The design of lock-based concurrent data structure is fraught with several problems, one of which is the use of locks in an implementation, namely their number and position in the code. Having many independent locks increases parallelism and makes it possible to reduce the total wait for obtaining a lock, on the other hand, the cases of excessive lock contention create big queues of threads wanting to acquire the same lock which, in turn, leads to a significant increase in the wait time. Also, having more locks comes at the cost of maintainability.

A large number of different implementations of concurrent algorithms and data structures have been developed that solve identical problems such as sorting arrays [1] and hash tables[2][3]. However, it appears to be hard to analytically (i.e., without running any performance tests) decide which concurrent data structure implementation should be chosen for a particular task. The well-known notions, such as the complexity and the memory usage within the Big O notation, do not take into account very important things: 1) the behavior, stability, and correctness of the data structure in various execution scenarios, and 2) the resource costs associated with various synchronization techniques.

The work "A Concurrency-Optimal List-Based Set" [4] introduces the concept of "concurrency optimality", which tries to present a way how to compare implementations in terms of correctness and stability. Concurrency optimality means the ability of a data structure to "accept" all correct concurrent schedules, that is, to reproduce all correct scenarios of concurrent execution.

As in [5][6], we consider only data structures with lock-based synchronization and just briefly touch on the lock-free approach as it is hard to come up with an exhaustive notion that will be valid for both lock-based and lock-free implementations.

In this work, we present an attempt to experimentally establish the relationship between the number and the granularity of locks in lock-based implementations of concurrent data structures and their performance by proposing a set of criteria that can be utilized to choose between various implementations.

A series of experiments with various implementations of concurrent dynamic arrays, sets, and multisets have led us to the base concept of a contention-aware concurrent data structure. We present such an implementation for concurrent hash tables, sets, and multisets and argue about the perspectives of an application of the proposed technique to any other concurrent data structure.

ROADMAP

In the first chapter, we define the data types considered in this work as well as the experimental environment and the methods of research.

The second chapter is devoted to conventional techniques: 1) we start with versions with no locks at all, we briefly talk about the sequential version (while incorrect, it sets a benchmark for us in terms of performance) and the lock-free one, then, 2) we move to single exclusive lock and read-write lock, lastly, 3) we consider the possibility of using multiple locks by splitting stored elements into blocks and applying different locks to them, arguing about the balance between having too many or too few locks.

In the third chapter, we try to improve locking strategies, introducing the concept of conditional locks, in which a read-write lock may not be taken in some situations. Also, there, we present an adaptive segmentation and explain how we can implement it in contention-aware concurrent sets.

In the fourth chapter, we present the results of the experiments we conducted as well as the results and to which conclusions they may lead.

1. OBJECTS AND METHODS OF RESEARCH

1.1. DATA TYPES

In this work, we study various implementations of the following three data types: *dynamic arrays*, *sets*, and *multisets*. In the description of these data types, we just present the list of operations, that are understandable by their names. Also, to simplify the presentation we split the operations into two groups the ones that modify the data structure, i.e., write operations, and the ones that don't, i.e., read operations.

Sets and *multisets* data types provide the following operations:

- *Read operations*: retrieve an element by value, retrieve the total number of elements stored.
- *Write operations*: add an element, remove an element by value, delete a set of elements, and clear the entire structure.

The array data type is the extension of a set data type. This means that it has the same operations as the set plus the following ones:

- *Read operations*: retrieve an element by index.
- *Write operations*: add an element (to the front/end/arbitrary position), remove an element by position.

As a performance measure, we choose the throughput, i.e., the number of operations per second. For each implementation, we measure the throughput on several different settings, i.e., the varying number of threads and the range of values.

1.2. EXPERIMENTAL ENVIRONMENT AND CONFIGURATION

For each implementation, its throughput was measured depending on various: numbers of threads (from 1 to 16), and the initial number of elements (from 10^2 to 10^6).

As for the data type, we chose a 32-bit integer for two reasons: 1) it is simpler to work with, it does not create a big overhead, and 2) because 32-bit integers can emulate the hash of objects with different distributions.

The results which are shown in the next chapters are obtained when the values are chosen from the full range of integer values.

We chose Synchrobench [7][8] as a benchmarking tool for modeling a high-contention concurrent execution environment since it provides opportunities for configuring various execution parameters.

Each measurement point consists of five independent launches of the benchmarking program with a 5s warmup, after which the resulting throughput is calculated as an average between the results obtained in launches.

Each launch starts with initializing the data structure being tested and filling it with a predetermined number of elements, i.e., the initial size. Values for the elements are chosen with uniform probability. Then the benchmarking program via a predetermined number of threads is performing read and write operations on the data structure for a 5s while counting the number of operations finished. We chose a 40% read-write ratio while the operations for each type (e.g., remove/add for write, size/contains for read) are chosen with uniform probability. The throughput is calculated as the ratio between the number of operations finished and the time of that launch (in our case, 5s).

Another thing worth mentioning is the range of values, which could be chosen as an argument for an operation. As these arguments are chosen with uniform probability, we have to set the range at $2 \cdot initial\ size$, so, for example, a calls of insert and remove operation have a 50% chance of success, i.e., getting successfully insert or remove a value.

2. OVERVIEW. CONVENTIONAL LOCKING TECHNIQUES

To start with, we consider several different implementations of the array data type. We study implementations in the order of the increasing number of locks and their configuration complexity. Starting with a sequential implementation, as in the previously mentioned work "A Concurrency-Optimal List-Based Set" [5], the second one we study is lock-free. Next, we move on to implementations with locks and, in order of increasing complexity, we consider: 1) the usual global exclusive lock, 2) global read-write lock, as well as 3) fine-grained locks applied to blocks of elements, rather than the entire data structure.

2.1. SEQUENTIAL IMPLEMENTATION

This implementation, as it does not satisfy linearizability [9], is not prepared to correctly work in a concurrent environment. However, we use it as our baseline since it should have the "best possible" throughput due to the absence of synchronization. In the experiments, it shows a relatively good performance that increases with the number of threads. Also, this implementation is predictably prone to a large number of unsuccessful operations (meaning the operations that give the result that is not expected, e.g., failing to find an element, which exists in the data structure or failing to add a new element when there is enough space for it and there is nothing to stop the operation) – about 80% of failures. That confirms the incorrectness of the usage of this implementation in an arbitrary concurrent environment.

2.2. LOCK-FREE IMPLEMENTATION

The second examined design is a lock-free array based on the Michael-Scott queue [10]. To maintain its correct operation and satisfy linearizability, while working without the use of locks, it has a large number of additional resource-intensive complications, namely, it implements the state mechanism of stored elements used to transfer them when the core size of the structure itself changes due to applying various write operations to it.

When the array has a static capacity, it is relatively fast, but the main problem comes, when you try to make it dynamically sized, as we considered dynamic data

types for this study. These complications lead to a strong decrease in performance which was confirmed by our experiments. It shows the worst results with a very high proportion of unsuccessful operations among other implementations as expected.

2.3. GLOBAL EXCLUSIVE LOCK IMPLEMENTATION

Now, we start to describe the implementations that use locks as a synchronization mechanism. At first, we consider a version with a single lock that provides exclusive access to the entire data structure. The performance of this version is lower than sequential but significantly better than lock-free and with an expected decrease as the number of threads grows. It should also be noted that this version showed much better results in terms of the success of operations. Such results are due to the very essence of exclusive locking – each operation is performed separately, excluding the problems of parallel writing, namely, conflicting additions and deletions of elements.

2.4. GLOBAL READ-WRITE LOCK IMPLEMENTATION

Moving from exclusive to read-write locks, by definition, we achieve the ability to perform read operations simultaneously. This implementation provides better performance, especially with a bigger number of elements than the previous version, since it allows simultaneous independent read operations, thereby reducing the lock acquisition time for such operations.

2.5. STATIC SEGMENTATION

At this point, we start to consider the set data type, meaning that we are excluding index-related operations.

The main problems with the previously provided implementations of the array data type are 1) the need for a long wait for concurrent write operations even if these operations are working on different parts of the structure, 2) the relatively long time on search operations (and, as a result, insertions, deletions, etc.) due to their linear complexity, since we have to traverse the whole array. To solve the first problem, we start to use locks, not on the entire data structure – we split a data structure into blocks, each one with its lock.

It is worth noting that we rejected the idea of just using a simple "array of arrays" with double linear indexing from the start since operations that involve finding elements still have linear complexity that depends on the size of the entire data structure. Thus, to solve this second problem, we developed our implementation with inspiration from the principle of consistent hashing [11], which is used for sharding in distributed systems. It is shown schematically in Fig. 1. Based on this principle, we create a data structure that is an array of units, each unit, in turn, is an earlier described concurrent array data structure guarded with a read-write lock. This idea is shown in Fig. 2 with elements, units, and their relations from Fig. 1.

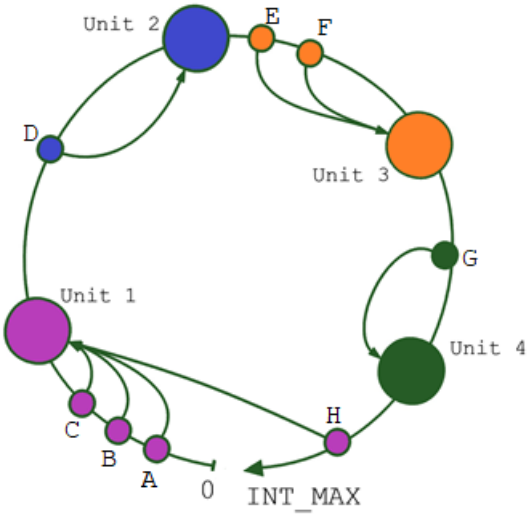

Fig. 1. Schematic illustration of consistent hashing

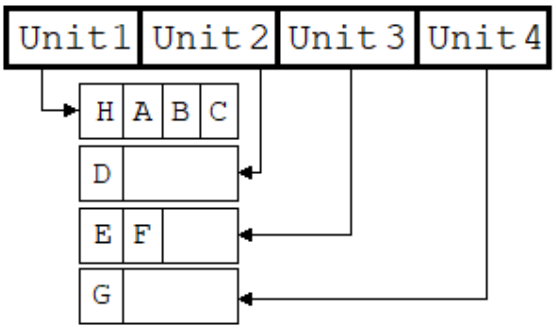

Fig. 2. Schematic illustration of the created data structure based on Fig. 1

Each unit stores elements and is defined by its *right border* – the maximum hash value of an object that can be put in it. Thus, each unit stores elements whose hash function value falls within the segment: less than or equal to its right border, but greater than the previous unit's right border.

Every operation starts with the calculation of the hash of the argument and with determining the corresponding unit of the key by using a binary search. Thus, the complexity of each request is $O(log\ (number\ of\ units)) + O(unit\ size)$ (instead of just $O(number\ of\ all\ elements\ stored)$, as was with array data types before). Such an organization of the data structure makes it possible to reduce the wait time for exclusive locks to be taken: 1) a global write lock on the entire array of units is taken for the rebuilding of the whole data structure which happens rarely; 2) a concurrent insertion of two elements happens in two different units with high probability, thus, taking the two corresponding locks in units independently.

The data structure is determined by two parameters: the width and the number of units. The width of a unit is the range of the hash segment allocated to this unit: for example, if the previous unit has a right border of 200, and the current one has 300, then the width of the current unit is 100.

As the mechanism of matching elements to units is cyclical in terms of hash values (see H and Unit 1 in Fig. 1), any units configuration will cover all possible values of the hash function, the main task is to build such a configuration, in which the units are distributed optimally for a particular use case, so operations will be executed in the shortest possible time.

To maintain this cyclicality, we need to clamp indexes of units we are working with:

```
clampIndex(index) {
    IF index > unitsCount – 1
        Return index – unitsCount
    ELSE IF index < 0
        Return index + unitsCount
    ELSE
        Return i
}
```

Despite that our best results for this data structure showed up to 4.5 times higher throughput than previous versions, the performance of this implementation is highly dependent on the setup of the parameters mentioned above. We should take into

account the potential number of stored elements, the number of threads, the write ratio, the range of element values, and their hashes.

In Chapter 4 we discuss the methodology we used to obtain possibly optimal setups as well as the values of these parameters that we have achieved.

CONCLUSIONS ON CHAPTER 2

The results obtained during the consideration of conventional approaches to the implementation of concurrent arrays and sets allowed us to draw the following conclusions:

1. Sequential implementations are not suitable for use in an arbitrary concurrent environment.
2. Despite being able to work with multiple threads, the lock-free version is not the best choice if performance is one of the primary factors when choosing an implementation.
3. The use of read-write locks is generally preferable to the usual ones, but it is necessary to take into account the peculiarities of their implementation in the libraries used and conduct preliminary tests before choosing them.
4. The division of the data structure into blocks can significantly improve performance, however, static segmentation is very sensitive to the parameters which should be configured for a specific task.

Speaking of the segmentation, it should also be noted that there is a possible workload when all the elements lie in the same hash range, thus, making the whole data structure work as a dynamic array or set data structure, described above and, consequently, have a very bad performance. Thus, we may benefit from allowing for the unit structure to be rebuilt based on the execution environment, thus, adapting for the best performance.

3. SEGMENTATION AND ADAPTIVITY

Before we show how to improve the segmentation approach, we explore the possibility of improving the strategy of using a single global lock, namely, to evaluate in which cases it is possible, without losing linearization, to refuse to take it, since 1) even an instantaneous successful take of a lock is associated with unnecessary time-consuming operations, and 2) taking a read lock with a concurrent write operation causes the need to wait for its release.

As conditional locks showed better results than standard read-write locks, we used them in the development of adaptive implementations of the segmentation approach, namely, segmented data structures that change the size and number of units depending on the change in the execution scenario – an increased or, conversely, an insufficient load of some units.

As a measure of the unit load, we considered two options – the size (number of elements) of the unit and the number of write operations applied to it relative to the number of write operations applied to the entire data structure.

3.1. CONDITIONAL LOCKS

Only for this subsection, do we return to the array data type and compare the resulting data structure with the one that uses standard read-write locks.

To improve the results, we decided to revise the strategy of locks acquisition to explore possible cases where taking a lock is not necessary while maintaining the correctness, i.e., the linearizability. We list the following scenarios:

1. By introducing an additional field to a data structure that indicates the number of elements stored in it (and updated with each record), the operation of obtaining this value can just read this field and take O (1), which allows not to take a read lock, if there are no concurrent operations that change the size of the data structure:
2. When performing operations related to adding new elements, the read lock may not be taken when searching for an element.

Inspired by the state machine used in the lock-free array, an atomic variable was added to the data structure, indicating the state of the array: overwriting an existing element (`WRITING_IN_THE_MIDDLE`), expanding at the front or the end (`EXTENDING`), or deleting an element (`REMOVING`; since these operations are associated with changing the size of the structure), as well as the absence of the above operations (`OK`).

Thus, all write operations still: 1) take an exclusive lock, 2) set the appropriate state of the structure, 3) perform their actions, and, finally, 4) return the state to what it was before they were called. For example, the operation of adding an element at given position:

```
add(element, position) {
        writeLock()
        previousStructureState = structureState.get()
        structureState.set(WRITING_IN_THE_MIDDLE)
        result = data.add(element, position)
        unlockWrite()
        structureState.CAS(WRITING_IN_THE_MIDDLE, previousStructureState)
        Return result
    }
```

Read operations, in the cases described above, do not take the lock and are still correct in terms of linearizability:

```
size() {
    IF structureState is WRITING_IN_THE_MIDDLE or OK
        Return data.elementsCount
    ELSE
        readLock()
        result = data.elementsCount
        unlockRead()
        Return result
}
contains(o) {
    IF structureState is EXTENDING or OK
        Return data.contains(o)
    ELSE
        readLock()
        result = data.contains(o)
        unlockRead()
        Return result
}
```

This approach shows an increase of 15-20% in performance compared to the version with standard read-write locks.

3.2. ADAPTIVITY

From now on we again continue to work with the set data type.

The main drawback of the static segmentation approach, described in 2.5, can be experienced in the following two situations:

1) With the small number of units, the process of choosing a unit takes a short time, however, the waiting time for writing to it, as well as the time of working with it (since the operations of searching, deleting, etc., have linear time complexity to the size of the unit) is relatively large.
2) If the number of units is relatively big, achieving a better distribution of values, and, as a result, a better distribution of operations between units, another problem arises: operations begin to affect only a small fraction of units, leaving most of them empty – extra time is spent on the search of a unit and the waiting for threads increases the work.

To solve these problems, it is necessary to introduce a mechanism that allows us to split overloaded units and combine ones, that are not loaded enough.

3.2.1. SPLITTING AND MERGING UNITS

As described in Chapter 2.5, in our implementation each unit consists of an array with a read-write lock and the biggest hash in it. Additionally, now we use conditional read-write locks rather than standard ones, because, as we have discovered earlier, they have better performance.

Thus, we need to introduce two new operations:

1. The union of two units makes a unit of the union containing the two arrays and a hash maximum which is the biggest one of the two:

```
IF unit1.hashRightBorder > unit2.hashRightBorder
    unit1.elements += unit2.elements
    unitsArray.remove(unit2)
ELSE
    unit2.elements += unit1.elements
    units_array.remove(unit1)
```

2. The split transfers the first half of all elements (in terms of their hashes, from smallest to largest) stored by the unit to a new unit with a new maximum equal to the maximum value of the hash function among the elements transferred.

   This is how we obtain the first half of all elements stored in a unit:

```
unit.elements.sortByHash()

border = elements count / 2
WHILE border < elements count
    border += 1

newUnit.elements = elements up to border
newUnit.hashRightBorder = biggest hash among its elements
```

   This allows us to add a new unit to the array before the current unit and remove copied elements from it:

```
unitsArray.add(curUnitPosition, newUnit)
curUnit.elements -= newUnit.elements
```

If the data type implies the storage of identical elements, it is necessary to take them into account when splitting, by modifying the clause for extracting elements:

```
border = elements count / 2
WHILE border < elements count
  AND current_element.hash == next_element.hash
    border += 1
```

Also, it is worth noting that in the process of splitting a unit, it is necessary to compare the elements by their hash values, not their values.

We decided to perform the merge and split depending on the load on the units to keep their load in heuristically defined borders. Thus, when we merge a unit, we do it with one of its two neighbors, which is loaded the least.

This way, the function of optimizing a unit is as follows:

```
optimizeUnit(currentIndex) {
    writeLock()
    prevIndex = clampIndex(currentIndex - 1)
    nextIndex = clampIndex(currentIndex + 1)

    currentUnit = unitsArray[currentIndex]
    prevUnit    = unitsArray[prevIndex]
    nextUnit    = unitsArray[nextIndex]

    currentUnitLoad = calculateLoad(currentUnit)
    prevUnit        = calculateLoad(prevUnit)
    nextUnit        = calculateLoad(nextUnit)

    IF currentUnitLoad > MAX_LOAD_VALUE
        result = splitUnit(currentUnit)
    ELSE IF currentUnitLoad < MIN_LOAD_VALUE {
        IF prevIndex == currentIndex and nextIndex == currentIndex
            result = False
        ELSE IF prevIndex == currentIndex
            result = mergeUnits(currentUnit, nextUnit)
        ELSE IF nextIndex == currentIndex
            result = mergeUnits(prevUnit, currentUnit)

        ELSE IF prevUnitLoad < nextUnitLoad
            result = mergeUnits(prevUnit, currentUnit)
        ELSE
            result = mergeUnits(currentUnit, nextUnit)
    }
    ELSE
        result = False
    unlockWrite()
    Return result
}
```

As a measure of the load quantity, we considered two approaches: 1) the size of a unit and 2) the number of write operations applied to it relative to the number of write operations applied to a data structure in total.

3.2.2. OPTIMIZATION FREQUENCY AND FORMAT

Another issue that needs addressing is the frequency of merge/split optimizations. Here we are faced with two problems and several solutions.

First, we have to consider the regularity, with which the checks a performed:

1. By introducing an additional parameter, we can perform optimizations in a certain number of write operations. It deprives the system of the ability to respond rapidly to a concentrated load on a single unit, which creates excessive contention around it. The example for remove operation is shown below:

```
remove(e) {
    readLock()
    unitIndex = clampIndex(findUnitByBinarySearch(e.hashCode()))
    result = units[unitIndex].remove(e)
    writesCount = writesGlobalCounter.incrementAndGet()
    unlockRead()
    if writesCount >= MAX_WRITES_TO_OPTIMIZE
        optimize() // zeroes the counter under the write lock
}
```

2. Without this parameter we can perform optimizations instantly:

```
remove(e) {
    readLock()
    unitIndex = clampIndex(findUnitByBinarySearch(e.hashCode()))
    result = units[unitIndex].remove(e)
    unlockRead()
    optimize()
}
```

Second, it is the way the optimizations are performed:

1. We can try optimizing just the unit used in the operation:

```
remove(e) {
    readLock()
    unitIndex = clampIndex(findUnitByBinarySearch(e.hashCode()))
    result = units[unitIndex].remove(e)
    writesCount = writesGlobalCounter.incrementAndGet()
    unlockRead()
    optimizeUnit(unitIndex)
}
```

2. Or we can cycle across all the units trying to optimize each one. The main problem with this approach is that the rebuilding of the system must be done with the write lock acquired for the entire array of units. This means that no other operations can be performed at the same time and are forced to wait

for the rebuild to finish. This rebuild can take a lot of time and doesn't provide the expected improvement of having a completely optimized array of units, but conversely, negatively affects the performance of the whole structure.

The method for this rebuild is present below:

```
optimizeAllUnits() {
    writeLock()
        while (true) {
            wasOptimized = false
            for i from 0 to units count
                if optimizeUnit(i) // it does not take the lock now
                    wasOptimized = true
                    break
            if  not wasOptimized
                break
        }
        unlockWrite()
}
```

3.2.3. ADAPTIVITY BASED ON UNIT SIZE

The idea of interpreting a load of a unit through its size was described in the bachelor's thesis "Development of memory-friendly concurrent data structures" by Smirnov R.A. [12], where blocks that become too big are split into halves.

We update that approach and present the following mechanism: to rebuild the structure of units, units that are too large in terms of the number of elements must be divided in half, and units that are too small must be combined with the smallest of the neighbors.

The system is defined by several parameters. The first ones are the minimum and maximum allowable sizes of units which preserve units being in optimal size.

The next parameter is the number of units with which the structure should be initialized. Here, as before, it is necessary to take into account the range of possible values of the hash function, however, the initializing units must be larger than the optimal size (defined by the first two parameters) – otherwise, when the data structure is only starting to fill with elements, it will waste extra time working with too many

units, and then all these units will start to be merged en masse as soon as the write operations reach them.

Having studied various configurations, we decided to optimize only the unit that corresponds to the write operation – optimization begins with checking if the unit's load is within the allowed limits, which (within the load measures that we consider) practically costs zero time to do.

Based on the stated above, write operations are as follows:

```
add(e) {
    readLock()
    unitIndex = clampIndex(findUnitByBinarySearch(e.hashCode()))
    result = units[unitIndex].add(e)
    unlockRead()
    optimizeUnit(unitIndex)
}
```

And the method of calculating a unit's load is just obtaining the number of elements stored in it:

```
calculateLoad(unit) {
    Return unit.elementsCount
}
```

With this approach, we were able to achieve better results, than with the static version. Even though it leaves a small number of empty units, it allows for an immediate response to an increased load on a unit, thus, preventing it from overloading.

Additionally, size adaptation partially solves the problem of a bad hash function – if its values are poorly distributed, overloaded units will be split into several.

3.2.4. ADAPTIVITY BASED ON THE NUMBER OF OPERATIONS

To count the number of write operations per unit, we add atomic counters to each, as well as an atomic counter to the data structure itself. Next, it was necessary to develop a model to calculate the load of a unit and compare it with the rest to decide whether to split the unit which is being analyzed or to merge it with some of its neighbors. Considering several different ideas, we choose the most reasonable three of them:

1. The decision of whether to split or to merge units is made depending on whether the number of operations with the unit is greater or less than some heuristically defined constants – the same idea used for the size approach described previously, but about the operations instead of the size:

```
calculateLoad(unit) {
    Return unit.operationsCount.get()
}
```

2. A load of a unit is calculated as the ratio of the number of operations applied to it to the number of operations to the whole structure. That is, as the share of the unit's participation in the work of the structure; similarly, the resulting value is compared to some predefined constants:

```
calculateLoad(unit) {
    Return unit.operationsCount.get() / totalOperationsCount.get()
}
```

3. We calculate the ratio of the number of operations per unit to the average number of operations among all units – a characteristic of how much the load per unit stands out from the mass:

```
calculateLoad(unit) {
    averageLoad = totalOperationsCount.get() / unitsArray.size()
    Return unit.operationsCount.get() / averageLoad
}
```

It is worth noting that for this implementation we use the idea to check the data structure on the rebuilding with some frequency parameter – the number of operations, once every time this check is performed and optimize just the unit the operation was performed on.

According to the results of our experiments, the third approach turned out to be the best: both in terms of results and ease of setup. Unlike the size-based adaptivity introduced previously, this approach distributes the load between units more optimally, since the participation of deletion operations is taken into account here directly (by counting them), and not indirectly through the size.

As for the parameters, we introduce a set of heuristically defined constants. The main parameter is the "*adaptivity*" factor, on which all other parameters depend. The

frequency (in operations) of optimization attempts is $offset \cdot adaptivity$, the *limits for the load ratio* obtained as described above are minimum – $m \cdot adaptivity$ and maximum – $M \cdot adaptivity$. The meaning of the last two parameters is the amount by which the load on the unit is allowed to stand out in comparison with the rest in terms of the load. In 4.3 we discuss the methodology we used for setting up these parameters as well as the values for them that we settled on.

The main downside of this approach turned out to be that it is more dependent on conditions such as the number of threads, the size of the structure, and the range of possible values than the size-based adaptive version.

Depending on the settings, it is possible to shift the dynamics of performance changes with a varying number of threads.

CONCLUSIONS ON CHAPTER 3

We obtained experimental confirmation of the success of the concept of conditional locks. In every situation we tested it in, regardless of the choice of metrics, both approaches showed a significant performance gain compared to static segmentation.

The main disadvantage, however, of these adaptive implementations is the dependence on the parameters that determine the structure operation strategy on the program execution conditions, such as the number of threads, the number of elements stored, the contention intensity, the mechanism of the operating system scheduler, and others. We managed to select the optimal parameters for various load tests, however, there are no universal settings.

This problem prompted us to start developing concepts in which these parameters are automatically adjusted in the face of changing thread contention.

Also, we are working on finding, perhaps, a more optimal model to calculate the unit load, for example, through the queue size of threads waiting to acquire a lock.

4. EXPERIMENTS. EVALUATION AND ANALYSIS

For a comprehensive and in-depth study of the behavior of each of the implementations presented, we decided to measure their performance while taking into account different execution conditions.

All the implementations are written in Java using only the standard library. The choice of language was determined, firstly, by the wide capabilities of the standard library and the presence of many various concurrent primitives, and, secondly, by the presence of a garbage collector that is very useful for us, since the development of memory management mechanisms is not the point of this paper.

All the plots in this Chapter has the throughput on the OY axis. Typically, on OX axis we have the number of threads, i.e., when there is a range from one to sixteen. We run all the experiments on one processor Xeon Gold 6230 with 20 cores. We have only up to 16 cores used, since we run in the VK Cloud Solutions where it is the largest possible instance.

The exact description of the data types used in our experiments as well as experimental configuration can be found in Chapter 1.

4.1. ANALYSIS OF THE CONVENTIONAL LOCKING TECHNIQUES

In Fig. 3 we present a comparison of sequential, lock-free, global exclusive/read-write lock approaches, as well as static segmentation. They are described in Chapter 2. Analyzing these charts, we can draw some conclusions.

The sequential version, although being our benchmark in terms of throughput figures, cannot be fully compared with the others, since, as mentioned earlier, this version does not work correctly in a concurrent environment.

The lock-free approach proved to be the worst in terms of performance and number of successful operations due to the inherent complexity of its implementation.

Another important point is that, although on smaller arrays the read-write lock shows close to similar performance as the exclusive lock, it vastly outperforms it as the number of elements increases.

Also, the segmentation approach (i.e., acquiring locks for blocks of data instead of the whole structure) performs much better than any of the other versions, that maintain linearizability.

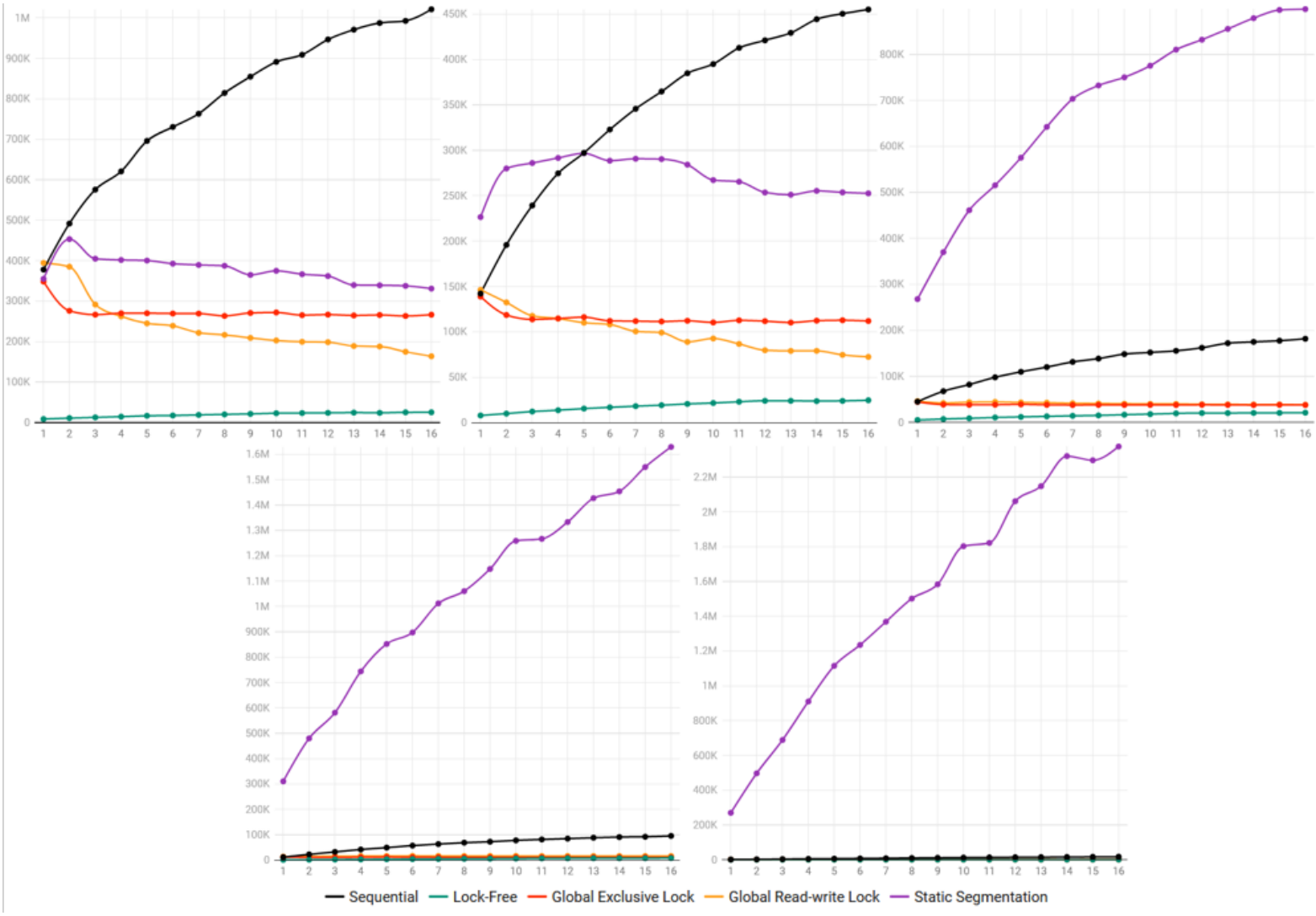

Fig. 3. Conventional techniques' throughput (ops/s) comparison. Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

4.2. SETTING UP STATIC SEGMENTATION

To achieve the best setups for various cases, we used binary search to find the optimal values for the number of units and width as well as their location on the "hash circle" (Fig. 1) to find the optimal units configuration.

The closest to what we can call universally optimal (in terms of maximizing throughput) setting that we discovered in our experiments is a setup in which there are units with a width of 1000, which are evenly spread across all the possible input values' hashes.

Bigger unit sizes lead to a performance decrease, whereas smaller unit sizes, which lead to a larger number of units inevitably face the limits of memory allocation.

4.3. LOCKS COMPARISON

As stated in 4.1, although on smaller arrays the read-write lock shows close to similar performance as the exclusive lock, it vastly outperforms it as the number of elements increases.

The conditional read-write lock approach showed an improvement over the conventional versions. Predictably, with one thread it showed results similar to them, because its mechanism loses its use with no competing threads, however, with several threads it builds its lead on read-write and exclusive locks, and practically retains its performance regardless of the number of threads.

The comparison between the three lock types is presented in Fig. 4 below.

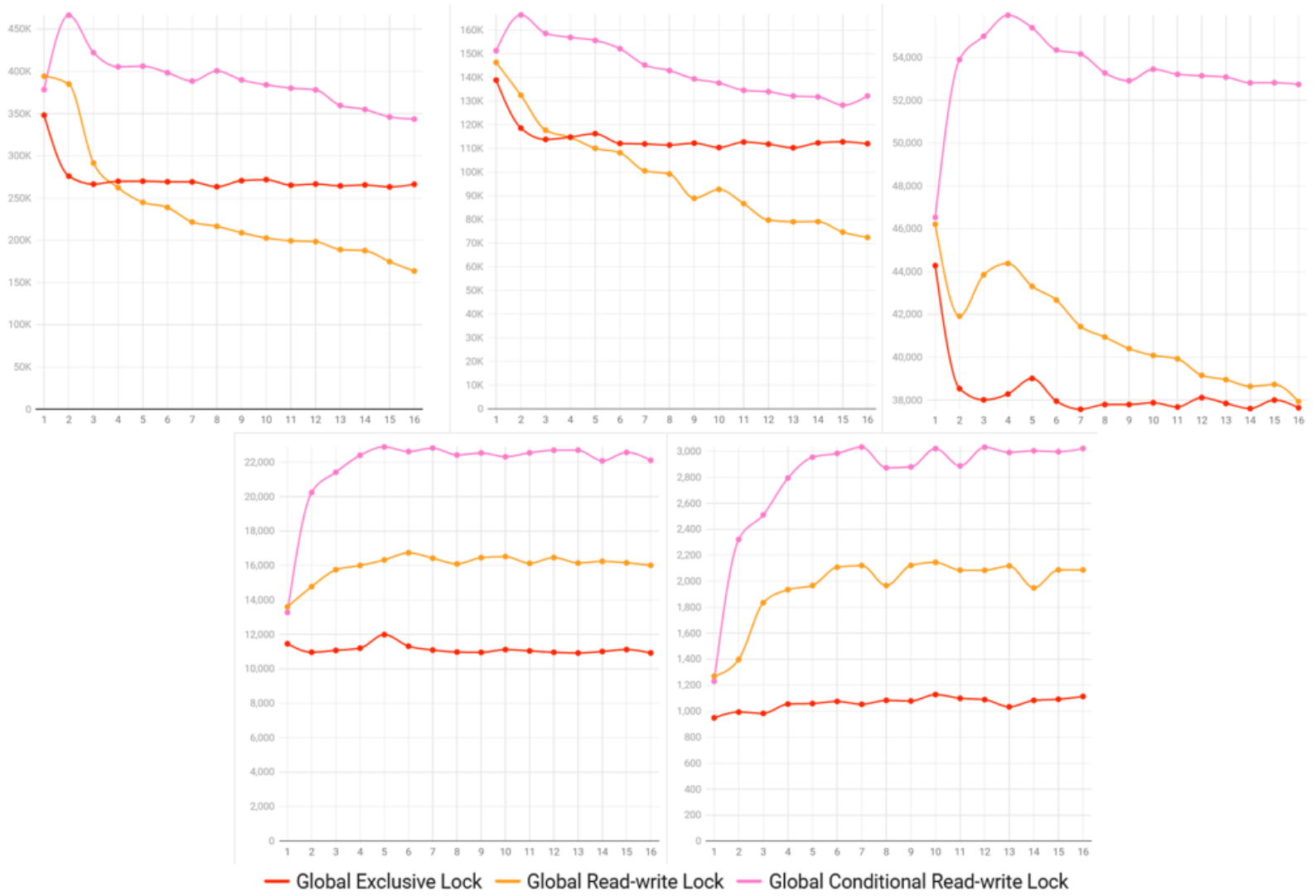

Fig. 4. Different types of locks' throughput (ops/s) comparison. Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

4.4. SETTING UP SIZE-BASED ADAPTIVITY

Our studies showed, that the closest to the optimal unit width lies between 50 and 200. As with static segmentation, borders, allowing for bigger unit sizes lead to a performance decrease, whereas those, which allow for smaller unit sizes, and, in turn, lead to a larger number of units inevitably face the limits of memory allocation.

As an example, Fig. 5 shows a comparison between several settings for the initial size of 10^4 elements in the format "*minimum border-maximum border*".

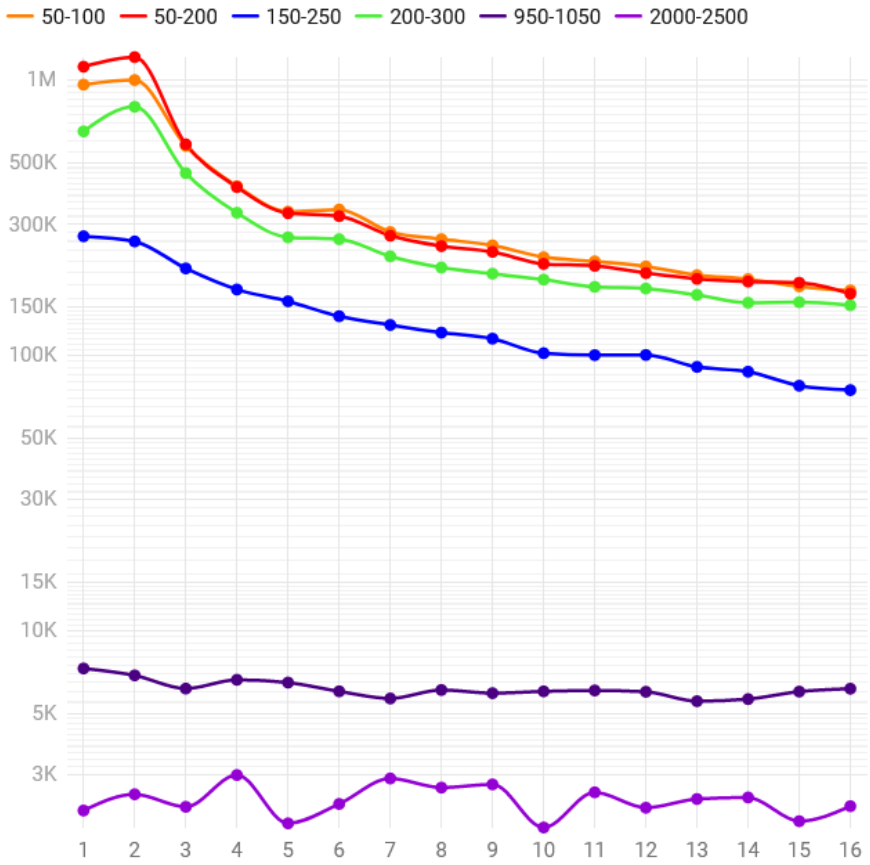

Fig. 5. Different size border settings' throughput (ops/s) comparison. Format: "minimum border-maximum border".

We obtained these results by fixing the execution setup and one of the borders, so we can use binary search to find the best value for the other. After that we would move this "window", repeat the same procedure, and decide, whether this move gives more throughput.

The results presented in this paper were obtained for a structure initialized with one unit, where it then decides how much to expand the number of units while being initially filled, so the following experiment starts with an optimal units setup.

Also, in 3.2.4 we stated, that the size-based adaptivity takes the deletion operations indirectly through the size. The chart, presented in Fig. 6, confirms that: the green line represents a test with a range of $2 \cdot initial\ size$, whereas the blue line – a test with a range of maximum integer value, 2,147,483,647. For both tests, we used the same parameters. As we explain in Chapter 1.2, a bigger range means less chance for deletion to get an argument, that is stored in a data structure, thus, in this case, less chance of units decreasing in size.

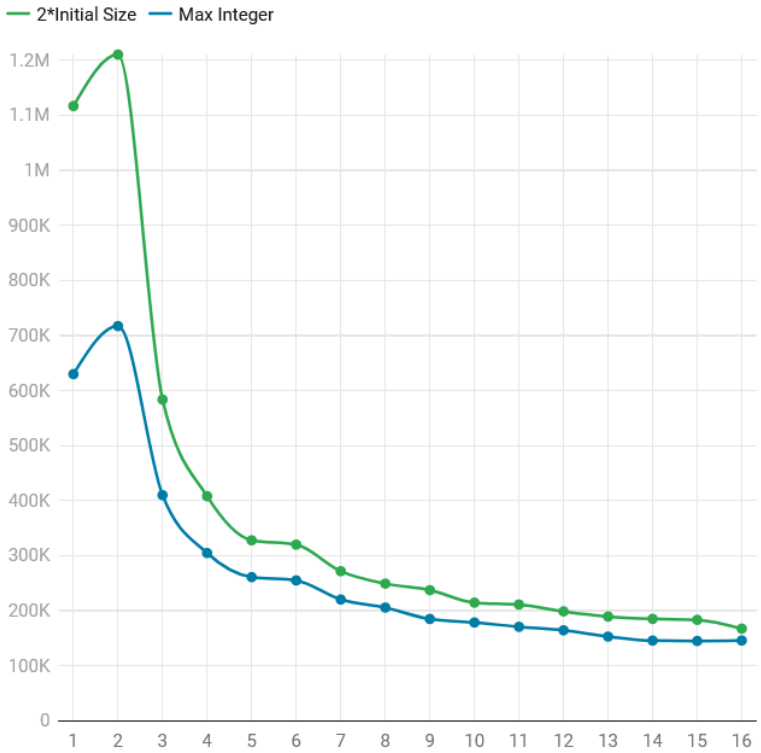

Fig. 6. Different ranges for size-based adaptivity. Initial size – 10^4 elements.

4.5. SETTING UP OPERATIONS-BASED ADAPTIVITY

For this adaptive version, we used the same technique for the limits for the load ratio – finding one border having fixed another, then moving the "window".

The main parameter of this system, as explained in Section 3.2.4, is $adaptivity$. It depends on the number of threads and, possibly, the intensity of contention, but we are yet to fully understand and explain the nature of those dependencies. Based on our experiments, we set it to 2.

The frequency (in operations) of optimization attempts is $offset \cdot adaptivity$ and depends on the contention intensity, as well as the number of write operations. Making it bigger means rarer rebuilds, lower – more frequent rebuilds. Fig. 7 shows a comparison between different values of $offset$ with the same other parameters. This example also shows, how changing $offset$ can differ the "dynamics" of performance: for 5 threads (the upper-right chart) the $offset$ of 10^3 is the best for smaller initial sizes, but for bigger 500 is preferred.

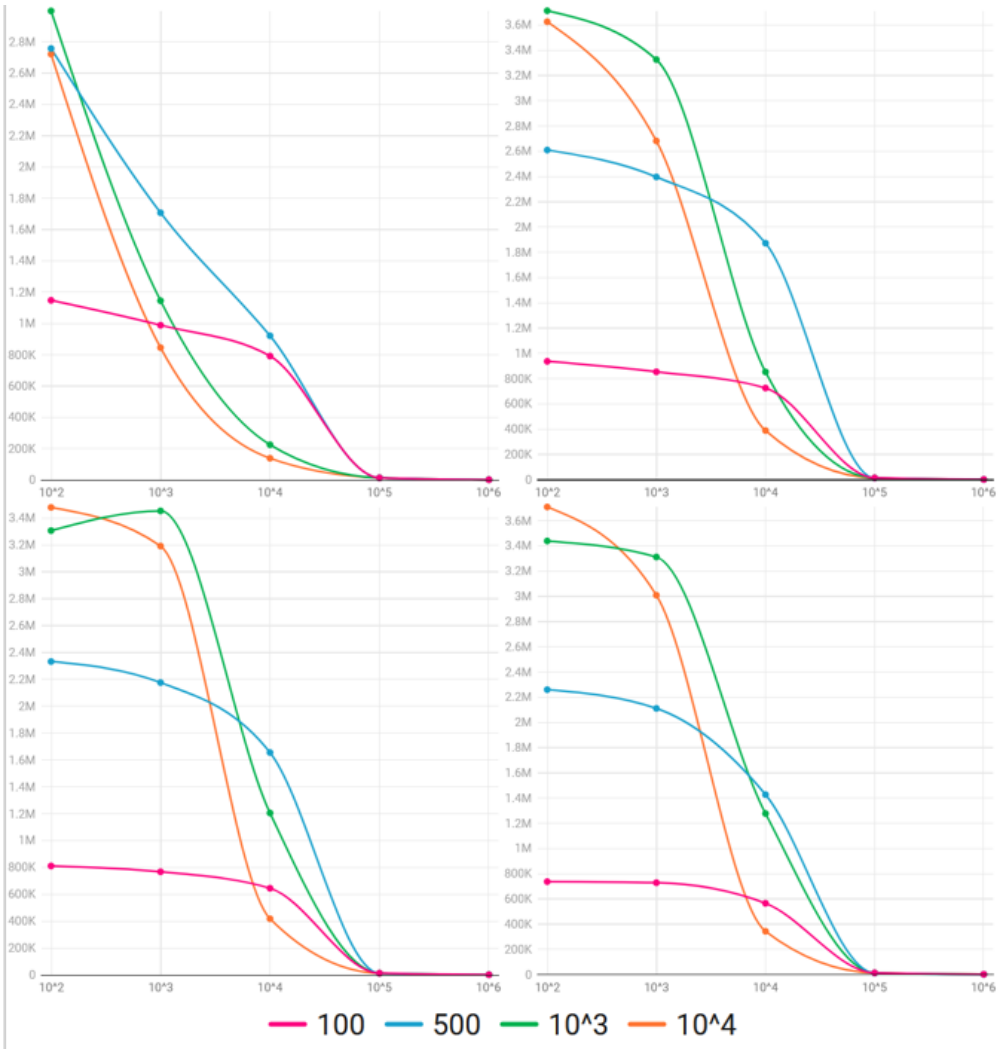

Fig. 7. Different $offset$ values for 1, 5, 10, and 15 threads.

We also discovered that in our experiments, if we remove the split operation for units, the load ratio asymptotically tends to 2.7, meaning, if we set the maximum border higher, no units will be split, as the load ratio will not reach the border.

Different load ratio limits suit better different $offsets$. Fig. 8-9 demonstrate that for $the\ offset$ of 500 for smaller initial sizes those limits give better results when set to $0.1 \cdot adaptivity$ for minimum and $0.7 \cdot adaptivity$ for maximum, whereas for bigger initial sizes a version with $0.1 – 0.4$ betters it. For offset of 1000 the situation is similar – $0.1 – 1$ for smaller and $0.1 – 0.7$ for bigger initial size.

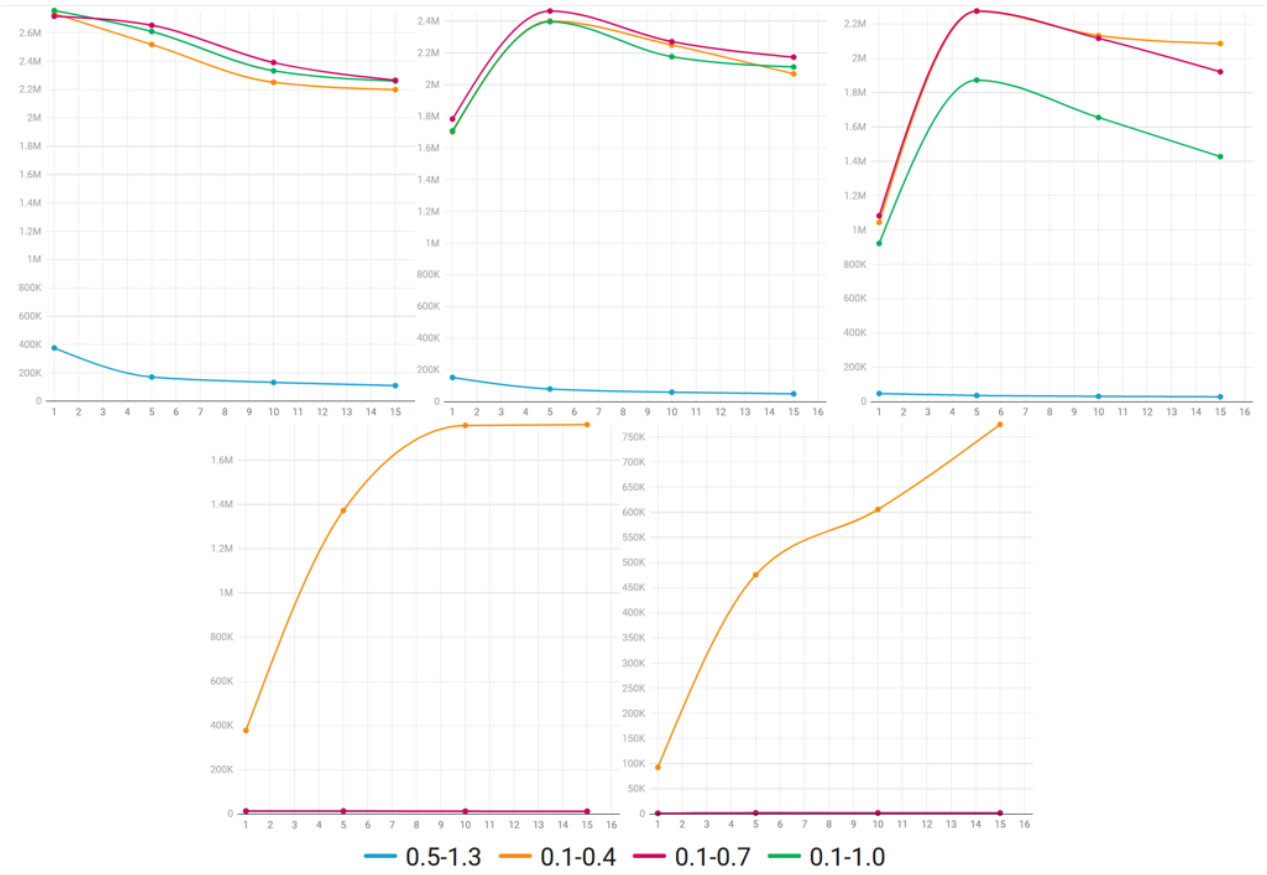

Fig. 8. Different load ratio limits for $the\ offset$ of 500. Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

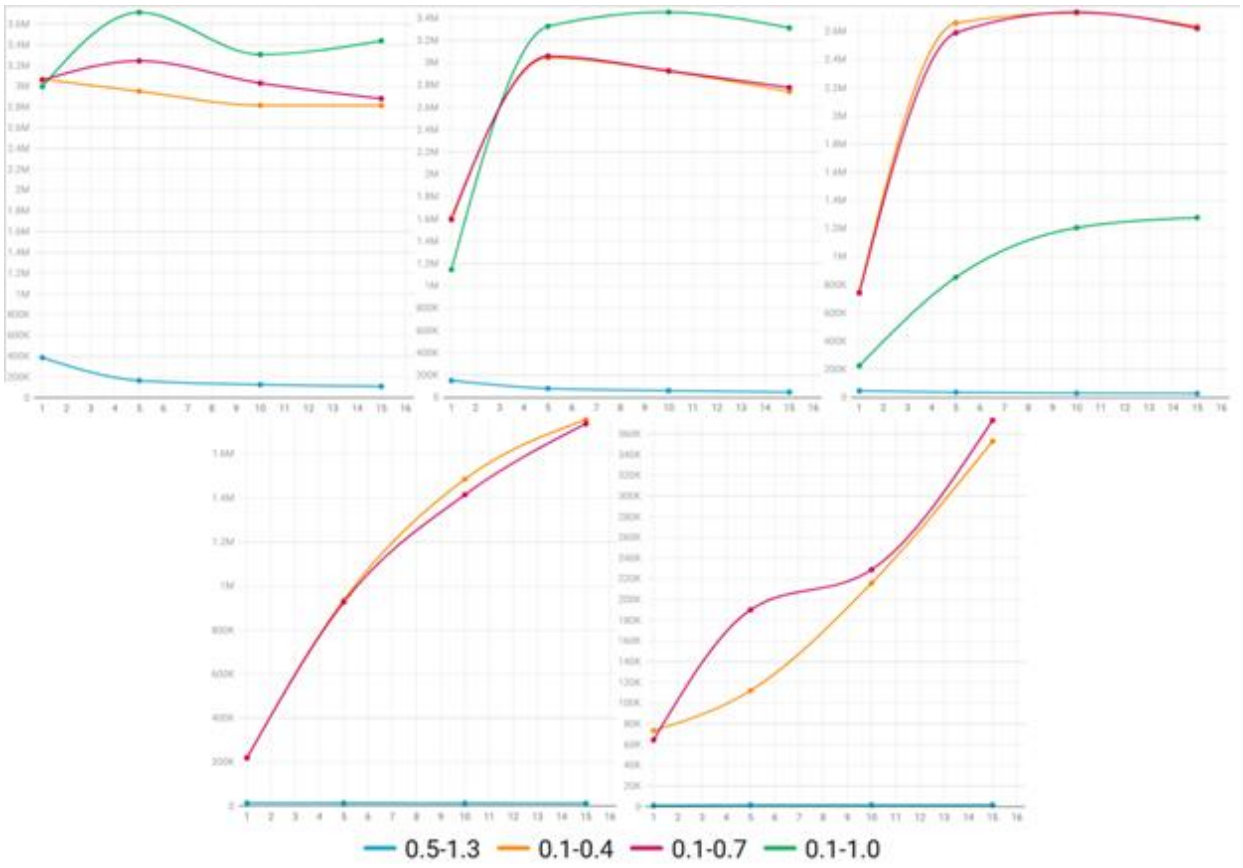

Fig. 9. Different load ratio limits for $the\ offset$ of 1000. Label format: "$min\ load\ ratio – max\ load\ ratio$". Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

Overall (see Fig. 10), for small initial sizes, it is better to start with relatively large pauses between rebuilds, then, for bigger, shrink down the load limits, and then increase the frequency of rebuilds.

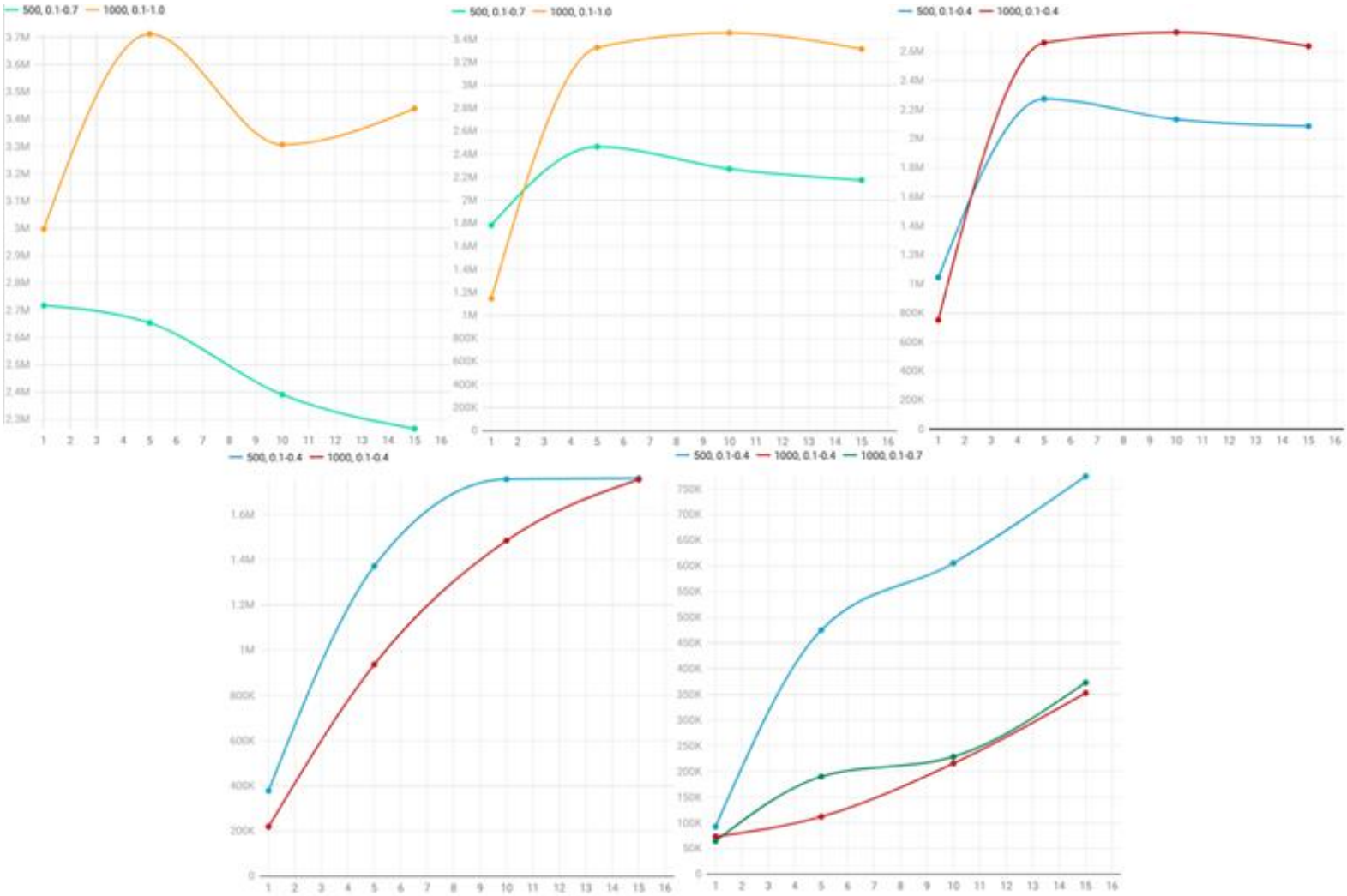

Fig. 10. Best configurations for operations-based adaptivity comparison. Label format: "$offset, min\ load\ ratio – max\ load\ ratio$". Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

4.6. ANALYSIS OF THE ADAPTIVE IMPLEMENTATIONS

The comparison between static segmentation and both adaptive versions is presented in Fig. 11. The first obvious conclusion from this comparison – the operations-based approach is a much better interpretation of a load of a unit. Not only does it have a better throughput than size-based adaptivity in almost every scenario, but its performance also grows with the increasing number of threads. The reason both versions have almost the same throughput for one thread is that they start being tested with already optimal units structure, compared to the static version.

Another point for the adaptivity, based on the number of operations is its tunability. It has more parameters, which react more noticeably for different settings – for a small number of elements one setup is better, for bigger – another, with big

differences in throughput between different setups (see 4.5 for an in-depth analysis). Conversely, in the size-based implementation, there is just one setup that is the best for pretty much any situation, but the resulting performance is worse, as Fig. 11 shows.

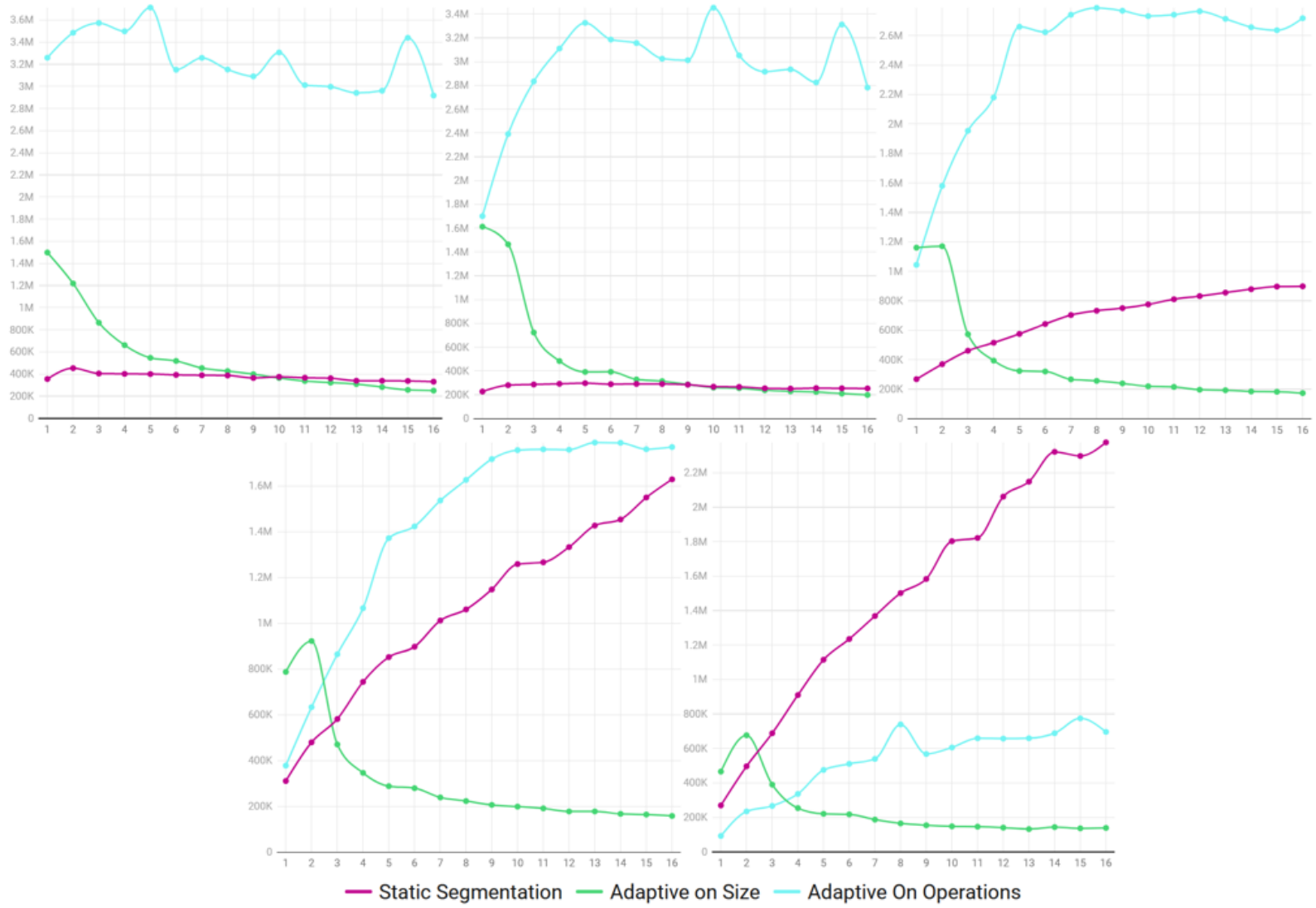

Fig. 11. Adaptive approaches' throughput (ops/s) comparison. Initial sizes of $10^2, 10^3, 10^4, 10^5, 10^6$ elements.

CONCLUSION ON CHAPTER 4

The experimental results confirm the theoretical anticipations, proposed in Chapters 2 and 3:

- Granular locks give more performance than a single global lock.
- Conditional read-write locks are superior to conventional read-write and exclusive locks.
- The adaptive approach to segmentation is better than the static.

- A load of a unit is better measured by counting the number of operations on this unit, rather than by measuring its size.

We presented general ideas of how to tune the parameters in proposed adaptive implementations for maximum performance and how they behave depending on different settings. However, so far, we are not in a position to give an exhaustive methodology, since this requires a much larger volume of experiments, which we hope to do in subsequent work.

CONCLUSION

The concept of an adaptive data structure described in Chapter 3 can be used as a basis for building adaptive concurrent implementations of sets, multisets, and hash tables.

A similar approach allows us to build an adaptive version of a concurrent list, where the blocks into which it is divided adapt their number and size with the same strategy. Based on the results of the experiments, we developed the following criteria for comparing concurrent implementations:

- Using locks is more efficient than the lock-free approach.
- The use of read-write locks is preferable (taking into account the specifics of their implementation in the library used).
- Locks should, if possible, be applied to blocks of elements, and not to the entire structure at once.
- Too small block size, and, as a result, too many locks, leads to performance drops.
- It is necessary to take into account the specifics of the task (the set of operations used, the potential load, etc.) and modify the universal model following them.
- The adaptive approach to segmentation leads to a significant increase in performance.
- In adaptive segmentation, it is better to measure a load of a unit by the number of write operations, applied to it.

PERSPECTIVES

The main disadvantage of the adaptive implementations is the dependence of the parameters that determine the structure operation strategy on the program execution conditions: the number of threads, the mechanism of the operating system scheduler, and others.

We managed to select the closest to optimal parameters for our chosen test configurations and formulate some hypotheses regarding what and in what form these parameters may depend.

In the future, we would need to develop an exhaustive methodology for setting up parameters for both static and adaptive segmentation, as well as a better presentation of the dependencies those parameters have.

We are currently investigating models in which these parameters are automatically adjusted in the face of changing thread competition, as well as other interpretations of unit load, such as the queue size of threads waiting to acquire a lock.

Also, we are studying other approaches to interpreting a unit load, such as the number of threads waiting to acquire the same lock, as well as other adaptation strategies: single thread adaptation, local unit rebuilds, and independent load estimates.

REFERENCES

1. Božidar D, Dobravec T. – Comparison of parallel sorting algorithms – 2015 – 16 P.
2. Feldman S., Laborde P., Dechev D. Concurrent multi-level arrays: Wait-free extensible hash maps // 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) – 2013 – P. 155-163.
3. Zhi Wen C., Xin H., Jianhua S., Hao C., Ligang H. Concurrent hash tables on multicore machines: Comparison, evaluation and implications – 2018 – 19 P.
4. Gramoli V., Kuznetsov P., Ravi S. In the Search for Optimal Concurrency – 2016 – 13 P.
5. Aksenov V., Gramoli V., Kuznetsov P., Ravi S., Shang D. A Concurrency-Optimal List-Based Set – 2015 – 15 P.
6. Aksenov V., Gramoli V., Kuznetsov P., Malova A., Ravi S. A Concurrency-Optimal Binary Search Tree // Euro-Par – 2017 –P. 580-593.
7. Gramoli V. More Than You Ever Wanted to Know about Synchronization Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms // Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP – 2015 – P. 1-10.
8. Synchrobench. URL: https://github.com/gramoli/synchrobench.
9. Herlihy M., Wing, J. Linearizability: a correctness condition for concurrent objects – ACM Trans. Program. Lang. Syst. 12, 3 – 1990 – P. 463–492.
10. Michael M., Scott M. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms // Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96) – Association for Computing Machinery, New York, NY, USA – 1996 – P. 267–275
11. Roughgarden T., Valiant G. CS168: The Modern Algorithmic Toolbox Lecture #1: Introduction and Consistent Hashing – 2015 – 12 P.
12. Смирнов Р. А. Разработка конкурентных структур данных, дружелюбных к памяти: выпускная квалификационная работа. – СПб., 2021. – 44 с.