Semester Project at the Departement of
Information Technology and Electrical Engineering
at the ETH Zurich

Autumn Term 2009

Fabian Gut

Faster Parallel Datastructures

Advisor: Johannes Schneider, ETZ G61.3, `schneider@tik.ee.ethz.ch`

Abstract

This work proposes an implementation of a linked list for faster parallel access by multiple processes or threads. Instead of only being able to lock the whole list or element by element, the list is partitioned into smaller parts which each can be locked individually if access is needed. This enables processes to work simultaneously in different parts of the list without producing a large overhead of locking elements. The list is implemented in Java. The results of the benchmarks show a considerable gain in speed with this new approach.

Contents

1 Introduction

A few years ago, processor manufacturers tried to outperform each other in terms of clockspeed of their processors, until the first multicore processor hit the market. Nowadays terms like ”dualcore” or ”quad core” are used to sell chips instead of 3+GHz figures. Today even most laptops contain a chip with at least two cores. This development leads to new challenges for hardware and software engineers: How to best use all this computation power?

On the hardware side things like shared memories and cache coherency bring new challenges where as on the software side the need for faster datastructures with parallel access for multiple processes or threads arises. Popular datastructures like arrays, linked lists and trees need to be reimplemented to give access to more than one thread at the time.

This work proposes an implementation of a sorted linked list for faster parallel access in Java. Locking nodes are distributed over the list such that parts of the list can be locked individually. The subsequent benchmarks have shown an increase in speed for increasing numbers of threads and locks.

The following pages present an overview of related work (section 2), the basic idea behind the implementation (section 3), the implementation itself (section 4) and the benchmarks (section 5) which the implementation has been subjected to.

2 Related Work

In [1] Moir and Shavit talk about the difficulties that arise while implementing and verifying concurrent data structures. On multicore processors the steps of threads can be interleaved arbitrarily. This means that the computation has to be viewed as completely asynchronous which leads to a lot of challenges in terms of performance and scalability. While lock-based blocking implementations are relatively easy to design and verify, they introduce new problems, while non-blocking approaches are usually a lot harder to design and verify.

2.1 Performance

The ratio of the execution time of an application on a single core to the execution time on P cores is called *speedup*. It measures how efficiently an application uses the available machine. A *linear speedup* of P would be desirable but is very hard and in some situations even impossible to achieve. If speedup grows with P, an application is called *scalable*. Introducing locks into a datastructure can severely limit scalability due to *sequential bottlenecks*. A sequential bottleneck is code which can only be executed by one thread at the time ie. that is executed while holding a lock. Say b is the fraction of the program that is subject to a sequential bottleneck and the program takes one time unit on a singlecore processor. Then the sequential part takes b time units on a P-core processor and the rest of the programm takes $(1 - b)/P$ time units. This means that the speedup S in the best case is limited to $1/(b + (1 - b)/P)$. Hence, if 10% of the application has to be executed sequentially S is limited to 5.3 on a 10-core processor. So the application is running at half the machines capacity. On our 16-core benchmark server the numbers are even more dramatic: The speedup would be at most 6.4 which is a mere 40% of the capacity of the server! Reducing the number and length of sequentially executed sections is therefore crucial to performance.

2.2 Blocking Techniques

Blocking techniques are usually quite simple to implement compared to non-blocking techniques but they have the undesirable effect, that a delayed thread holding a lock also delays all other threads trying to acquire that lock. Parts of applications that use locks lack parallelism and are therefore not scalable. One way to overcome this problem is a fine-grained locking scheme, which means to reduce the number of instructions executed while holding a lock and/or using multiple locks for different parts of a data structure such that individual operations can take place at the same time when not accessing the same parts of the data structure. Another way is to access the data structure in different time intervals. One popular technique to do so is *backoff*.

2.3 Nonblocking Techniques

Nonblocking techniques try to overcome the limitations introduced by the use of locks in a data structure. Thus nonblocking progress conditions have been considered in the literature:

- wait-freedom guarantees that an operations completes after a finite number of its own steps.
- lock-freedom guarantees that after a finit number of an operations steps, *some* operations completes.
- obstruction-freedom guarantees the completion of an operation within a finite number of steps after it stops encountering interference from other operations.

Of course wait-freedom is a stronger condition than lock-freedom, and lock-freedom is a stronger condition than obstruction-freedom. Nevertheless, all three conditions forbid the use of locks. Although stronger conditions seem useful, weaker conditions are usually easier to implement and to prove correct.

Nonblocking data structures are implemented using a number of special atomic instructions such as *compare-and-swap* (CAS) and *load-linked/store-conditional* (LL/SC). Such instructions are *universal*: there exists a wait-free implementation for any datastructure on a system that supports such instructions.

2.4 Linked Lists

Other than globally locking a linked list, the most popular technique is *hand-over-hand locking* where each node has an associated lock. A thread traversing a list always locks the next node before unlocking the previous one, thus preventing overtaking which may cause unnoticed deletion or insertion of a node. However, this limits concurrency as one thread can't overtake another in the list even if its operations need nodes in a completely different part of the list.

3 Idea

To keep the data in a linked list consistent, the accessed data has to be locked on access. This means that if two threads want to access the list simultaneously one of them has to wait for the other to finish its operation. Thus, there is no gain in speed if two threads work on the list. On the contrary: In most situations the operations are slower, as the locking process takes time and resources. This problem has no solution if the operation needs the whole list to be locked. However, what if two operations only need to change a few elements? In this situation a second operation in another part of the list could take place simultaneously! This is especially true for large lists with a few thousand or so elements.

So why not lock each element separately? This would cause a lot of overhead compared with only one lock for the whole list. Furthermore operations like insert and delete would become rather complicated as more than one element has to be locked and one would always have to check whether those locked elements are still part of the list after locking them. This is the motivation behind this work. What this work proposes are several locks distributed over the list which are responsible for locking a certain part of the list.

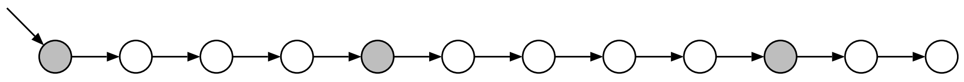

Figure 1: List with gray locking nodes

Fig. 1 shows the basic principle of the distributed lock elements. The first node (header, see Sec. 4.5.1) is always a locking element. Each gray lock node is responsible for locking the following normal nodes up to the next lock node. The nodes between two locks are called a *group*.

As inserting and deleting a node changes the structure of the list, the distribution of locks may become unballanced if for example a lot of nodes are deleted in a certain part of the list. After a certain number of deletes and inserts the list is therefore reballanced. Reballancing is done under these conditions:

- The number of needed locks differs by two from the number of locks currently in the list, or

- There is a group with less than two elements, and

- There is no other reballancing operation currently running.

4 Implementation

The list is implemented in Java. Following is a detailed description of the individual classes.

4.1 AbstractList

The class AbstractList provides a base class for the implementation of a linked list. This class contains definitions for the following abstract methods and variables:

- **listSize** is an AtomicInteger and contains the size of the list. It is incremented or decremented by insert and delete, respectively. The AtomicInteger type ensures that increments and decrements are atomic operations.
- **first ( )** returns the first element of the list. It provides an entrypoint to the list.
- **insert (GenericNode newNode)** inserts a new node newNode into the list. It throws LinkedListExceptions.
- **delete (int key)** deletes the element from the list, that corresponds to the provided key. It too throws LinkedListExceptions.
- **isEmpty( )** returns true if the list is empty, that is if listSize ==0, and false otherwhise.
- **print ( )** prints the list to stdout.

AbstractList also provides these non-abstract functions to handle listSize :

- **size ( )** returns the value of listSize , that is the size of the list.
- **setSize (int n)** sets the size of the list listSize to the value of the parameter n.
- **incrementSize( )** increments the value of listSize by 1.
- **decrementSize( )** decrements the value of listSize by 1.

4.2 GenericNode

GenericNode class provides a basic node for the list. A node consists of a key, an element and a pointer to the next node in the list. The class has two constructors: The first has an object and a key as parameters whereas the second only has the key. However, the second just calls the first with "null" as object.

4.3 LockNode

The LockNode class extends GenericNode. It has the same two constructors as GenericNode and both just call the corresponding superconstructor. The class also provides some additional variables and functions. All variables are protected, so they can't be accessed outside the lists package. The new protected variables are the following:

- **locked** is an AtomicBoolean and provides the means to lock a node. It can only be accessed through functions in the LockList classes.
- **lockedBy** denotes which thread currently holds the lock. It is set to the threads id whenever a lock is acquired.
- **nextLock** is a pointer to the next lock in the list or to the sentinel if it is the last lock in the list.
- **active** is a boolean that indicates whether or not the lock is still an active lock in the list. During reballancing locks might be deleted from the list and active is set to false.

The two public functions of LockNode are nextLock( ) and isActive( ). The first returns a pointer to nextLock and the later returns the value of active.

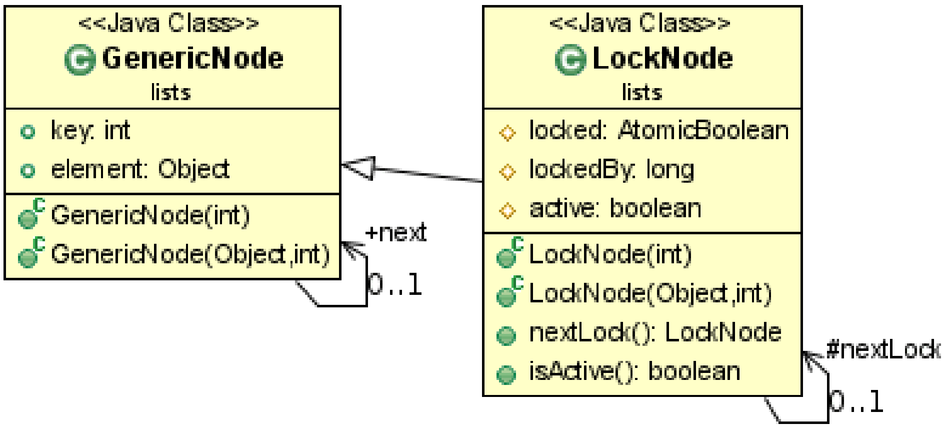

Figure 2: Class diagram of nodes

4.4 LockList

This class is the first attempt. It had several unresolved issues which eventually resulted in LockListV2, a complete redesign of the list class. With this implementation every lock also held data and could be inserted and/or deleted as every other node. During reballancing it was also possible, that a normal GenericNode had to be changed into a LockNode to ensure an even distribution of locks over the list. The problem with this implementation was basically, that the lock responsible for locking a node could change. If a lock was deleted from the list and set to inactive one would have to start from scratch with searching the lock responsible for the node. This made working with the list and especially the insert and delete functions extremely complicated as both functions had a number of special cases:

- Deleting or inserting a node at the front of the list
- Deleting or inserting a node in front of a lock
- Deleting or inserting a LockNode

The handling of these cases led to extremely complicated and error-prone sourcecode.

4.5 LockListV2

In this second version of the LockList no data was held on the locks anymore. So in normal use of the list locknodes don't get deleted or inserted. Only during reballancing locks can be removed or added to the list. This leavs one easy to handle special case:

- during acquisition of a lock an additional lock is added between the lock and the place where a node should be inserted or deleted

LockListV2 implements all abstract classes from AbstractList and has some also some functions and variables of its own.

4.5.1 Variables

Header and sentinel are two LockNodes and are the first and last nodes of the list, respectively. The header is needed as an entrypoint to the list and it is the first lock of the list. The sentinel is there for convenience reasons. The header has key 0 and the sentinel has key Integer.MAX_VALUE, the later being the only invalid key for a node in the list.

Delins, reballance, locks and threads are four auxilliar variables for the reballancing function. Delins and threads are AtomicIntegers. Delins indicates how often delete and insert have been called since the last time the list has been reballanced. It is incremented by both delete and insert and set to 0 by reballance( ). The threads variable is used to store the number of threads working on the list. It is assumed that this number is known.
Reballance is an AtomicBoolean and is used by reballance( ) to determine

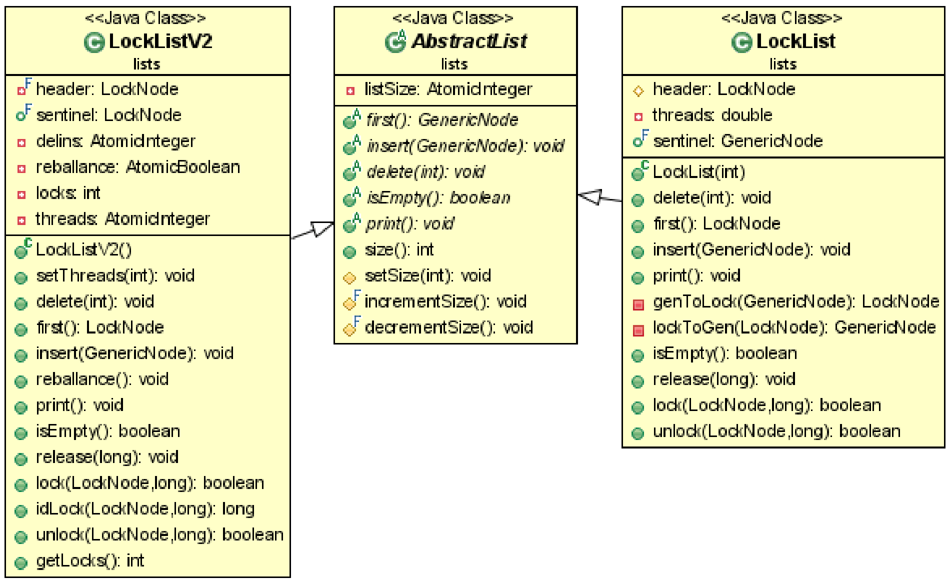

Figure 3: Class diagram of lists

whether another reballancing is currently taking place. If so, all other calls to reballance( ) are aborted.
The integer value locks stores the number of locks currently included in the list. Its value depends on the length of the list and the number of threads currently working with it.

4.5.2 Functions

Constructor LockListV2( ) is the only constructor for the LockListV2 class. LockListV2( ) takes no arguments. It initializes header and sentinel and connects them.

setThreads(int n) is used to set the number of threads working on the list which is provided as an integer argument. Although Java knows how many cores are available in the system, there's no way to determine how many threads use a certain instance of a class. Keeping track of the different thread IDs would cause a huge overhead. The number of threads is a piece of vital information for the reballance function, so for this implementation it is assumed that the number of threads can be determined in future multicore systems or that this number is provided by someone. Hence this function.

Delete(int key) and insert (GenericNode newNode) delete or insert a node into the list, respectively. Both functions are implemented recursively, i.e. they start over if something goes wrong with acquiring a lock (inactive or additional lock). This only happens if reballancing takes place at the same time one of these two functions is executed and a lock is removed or inserted into the list.

First ( ) returns a reference to header.

Reballance( ) adjusts the number and positions of the locks in the list. Reasons that make this necessary are the following: After a number of delete and insert operations unballanced groups of nodes may have been formed, or the length of the list and/or the number of threads working on the list may have changed. Reballancing is only done if a group of nodes is smaller than 2 or the new number of locks differs by more then 2 from the momentary number. The later is to prevent oscillating lock numbers. If the difference would be only 1, it could happen that the length of the list or the number of threads change such, that with every reballancing the number of locks changes up or down by one lock. This would cause unnecessary reballancing overhead. Reballancing is only done if no other thread reballances the list at the same time.

Lock(LockNode lockNode, long id) is the first possibility to lock a node. It tries to lock the node until it succeeds, then it returns **true** if the lock was still active or it unlocks the node again and returns **false** if the lock has been set to inactive.

idLock(LockNode lockNode, long id) is the second locking function. The difference to lock is that it only tries to lock the node once and then returns the id of the thread that currently holds the lock. It can be used to prevent getting blocked while trying to lock a node that has already been locked by the same thread.

Unlock(LockNode lockNode, long id) releases a lock if the thread calling the function holds the lock. It returns true or false on success or failure, respectively.

Release(long id) can be used to release all locks in the list held by a thread.

getLocks( ) returns the number of locks currently active in the list.

5 Benchmark

5.1 Single Lock

The first benchmark was run with only insert and delete operations with an equal chance for both operations. In total 160000 operations were run, evenly distributed over the threads. The benchmark was run for 1 through 16 threads and with 1, 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 locks in a list with a starting length of 10000 nodes. The results of this benchmark are shown in Fig. 4:

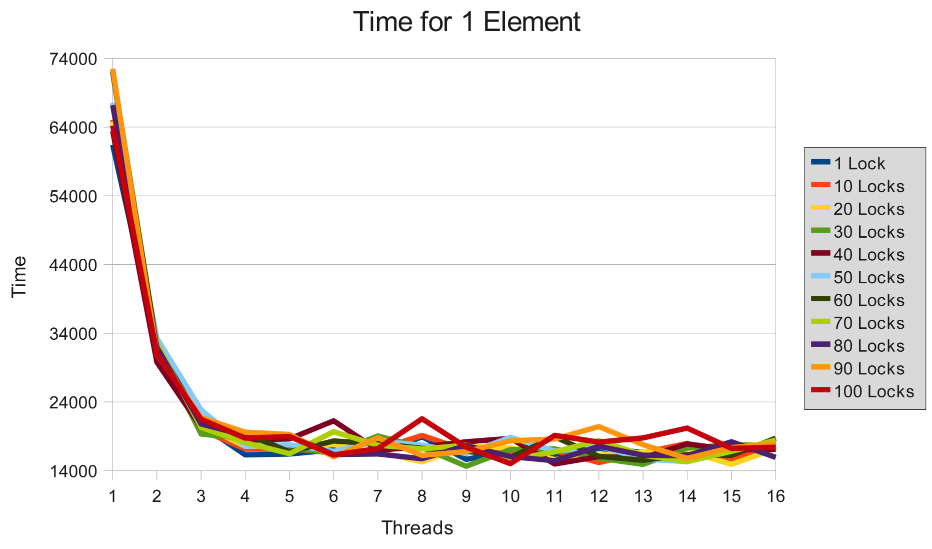

Figure 4: Result of first benchmark run

It shows, that there is an increase in speed with more than one thread. However, the number of locks in the list seams to have no influence on the speed of the operations what so ever. This was not expected, so in search for the reasons measurements of the time for certain operations were taken (Tab. 1).

These value show, that alot more time ($\sim 98.5\%$) is spent traversing the list and searching for the right node than is used to lock, access and unlock the node ($\sim 1.5\%$).

5.2 Multiple Locks

With this second benchmark, locks weren't released immediatly after access to a node. This time all required locks were acquired and held until all nodes were accessed and only then they were released, resulting in a longer lock time than in the first benchmark. This simulates the making of a "snapshot"

Operation	Time $[ns]$
early delete	952
middle delete	66938
late delete	125611
early insert	521
middle insert	52592
late insert	93823
lock node	142
unlock node	183
traversing list	105157
node object access	558

Table 1: Timemeasurements

where valid data of a number of nodes is needed at the same time.
Three parameters were varied this time: Number of locks ($[1, 1000]$),number of accessed nodes per operation ($[1, 1000]$) and of course the number of threads (1, 2, 4, 6, 9, 12 and 15).
The first run was with a list of 10000 nodes. In Fig. 5 you can see the results for 1000 accessed nodes. The time measured is for 77760 operations ($77760 = 2 \cdot 4 \cdot 6 \cdot 9 \cdot 12 \cdot 15$) equally distributed over the threads.

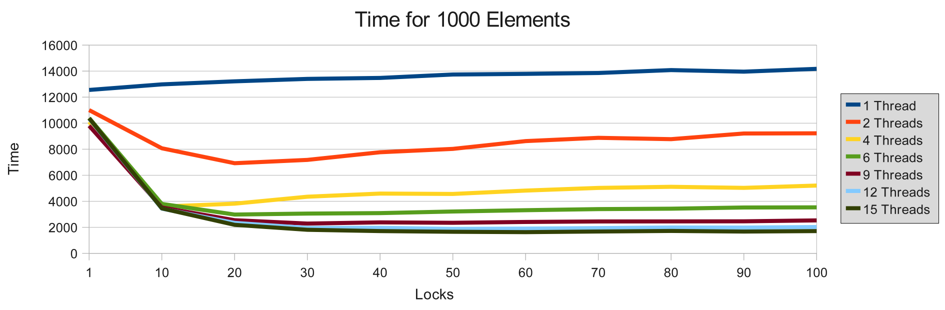

Figure 5: Benchmark with 1000 accessed nodes in a 10000 node list

This looks more like the expected results. One can see a clear increase in speed for increasing numbers of both threads and locks. The achieved speedup S is larger than 7. This means that the part of the code that is subject to a sequential bottleneck p is smaller than 8%

5.3 Optimal number of locks

A dependency between the number of accessed nodes and the optimal number of locks could not be detected. Therefore, and because it would be very costly to keep track of how many elements are in average accessed per operation, the search for a function that determines the optimal number of locks was focused on functions of the form $f(threads, length.of.list) = \#locks$.

For this purpose a benchmark with operations that access 10% of the nodes was run on lists with $x * 1000$ nodes with $x \in [1, 20]$.

List length \ Threads	1	2	4	6	9	12	15
1000	1	4	10	12	21	19	21
2000	1	3	12	24	22	24	35
3000	1	5	11	22	31	37	33
4000	1	7	10	23	35	39	49
5000	1	6	11	17	33	48	61
6000	1	6	14	18	32	44	51
7000	1	7	11	21	51	54	59
8000	1	16	15	17	41	41	53
9000	1	13	13	20	25	59	59
10000	1	4	13	17	29	53	49
11000	1	6	31	29	31	59	55
12000	1	10	19	19	42	60	69
13000	1	6	22	22	37	47	66
14000	1	9	30	24	35	49	64
15000	1	14	27	29	38	55	68
16000	1	11	15	28	30	50	62
17000	1	7	27	37	37	57	65
18000	1	6	32	32	32	50	70
19000	1	19	14	23	36	48	64
20000	1	5	11	35	51	51	66

Table 2: Optimal Locks

The values from Tab. 2 are plotted in Fig. 6.

With the help of the Genetic Algorithm Toolbox for Matlab [2] a function of the form $f(x, y) = a \cdot x^2 + b \cdot y^2 + c \cdot x \cdot y + d \cdot x + e \cdot y + f$ that best approximates the values in Tab. 2 was searched. The result was:

$$f(x, y) = -0.0562 \cdot x^2 - 9.3981 \cdot 10^{-9} \cdot y^2 + 1.2716 \cdot 10^{-4} \cdot x \cdot y + 3.3865 \cdot x - 0.8267$$

The function is plotted in Fig. 7. It has not been tested any further though.

On first sight this might look as if the dependency on the length of the list is rather small (10^{-4} and 10^{-9}). Keep in mind though, that lengths used in this benchmark have a similar power ($y \sim 10^4$ and $y^2 \sim 10^8$).

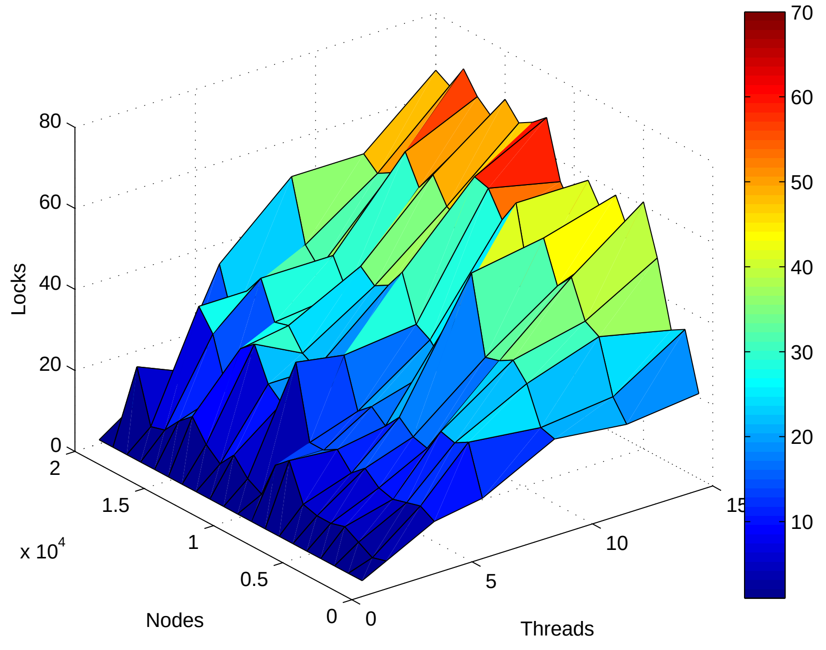

Figure 6: Measured locks plotted over threads and list length

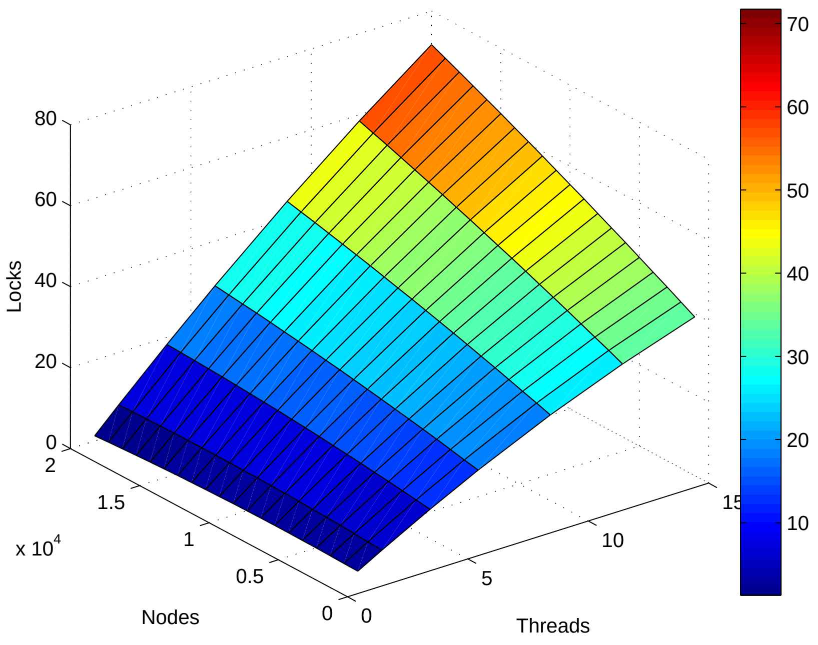

Figure 7: Optimal locks plotted over threads and list length

6 Future Work

6.1 Number of locks

The function that determines the number of locks was derived from a benchmark that locked 10% of the elements in the list with every operation. Maybe a better function can be found that gives better results on speed with mixed benchmarks where some operations acquire a lot of locks and others only lock a single node. Also the number of locks needed outside the tested range is unknown. Hence, longer or shorter lists can be investigated.

6.2 Reballancing

For reballancing a few questions can be answered. How often is reballancing needed? How bad are very small and large groups for the performance? Would it be a good idea to keep track of changing thread numbers and keep the number of locks according to the average thread number?

6.3 Other data structures

The same principle that is used for a linked list in this work could be applied to other data structures like trees.

7 Conclusion

The goal of this work was a faster datasctructure for parallel access using a new approach of a few distributed locks in the list. With a considerable speedup of $S > 7$ this goal has been achieved.

Personally the work was very instructive for me. I have never worked with multithreaded datastructures before, so I needed to adapt a new way of thinking about the code. Debugging therefore posed a real challenge for me and hence needed a lot of time in the beginning. Especially as the very first benchmark version was completely randomized in terms of what happened with the accessed elements (eg. insert, delete, change object, etc.) which sometimes made it almost impossible to understand how the operations where interleaved. However, even without random, reproducing a certain error was sometimes very complicated; especially if there was more than one reason why eg. a NullPointerException occured.

A Javadoc

A.1 AbstractList

Full name: `public abstract class AbstractList`

Package lists

Inherits Object

AbstractList is the abstract base class for all linked list implementations.

Author Fabian Gut

Version 2

Inheritancetable

Element	Inherited from
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Fields

private AtomicInteger listSize The size of the list. It is incremented with every call of insert and decremented with every call of delete

---

Construktors

public AbstractList( )

Methods

public abstract GenericNode first( ) Returns the first element of the list.

See also GenericNode

Return the first element of the list as a GenericNode

---

public abstract void insert(GenericNode newNode) throws LinkedListException Inserts a new GenericNode into the list.

Parameter	GenericNode newNode	the node to insert
Exceptions	LinkedListException	can have multiple causes. What exceptions are thrown depends on the implementation.

---

public abstract void delete(int key) throws LinkedListException Deletes the element from the list that corresponds to the provided key.

Parameter	int key	specifies which element to delete
Exceptions	LinkedListException	can have multiple causes. What exceptions are thrown depends on the implementation.

---

public abstract boolean isEmpty( ) Returns whether the list is empty or not.

Return returns true if the list is empty and false otherwise.

---

public abstract void print( ) Prints the list to stdout.

---

public int size( ) Returns the size of the list

Return returns the size of the list by returning the value of listSize

---

protected void setSize(int n) Sets the size of the list.

Parameter	int n	the new listsize to set

---

protected final void incrementSize( ) Increments the size of the list by 1.

---

protected final void decrementSize( ) Decrements the size of the list by 1.

A.2 GenericNode

Full name: `public class GenericNode`

Package lists

Inherits Object

GenericNode provides a basic list element. It contains a key, an arbitrary element and a pointer to the following node. The constructor has to be provided with a key at least.

Author fabian

Version 1

Inheritancetable

Element	Inherited from
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Fields

public int key The key of the node.

---

public Object element The content of the node. Can be any class that extends the class Object.

---

public GenericNode next The pointer to the next node in the list.

Construktors

public GenericNode(int key) Class constructor. It calls GenericNode(Object, int) with null as the Object.

Parameter	int key	the key of the new node

---

public GenericNode(Object newElement, int key) Class constructor with a specified element. The element and key are set to the parameters newElement and key, respectively and the pointer to the next node is set to null. The pointer is only set once the element is inserted into a list.

Parameter	Object newElement	the element contained in the node
	int key	the key of the node

A.3 LockNode

Full name: `public class LockNode`

Package lists

Inherits Object←GenericNode

LockNode implements a node that can be locked. The class extends GenericNode.

Author Fabian Gut

Version 1

Inheritancetable

Element	Inherited from
Object element	GenericNode
int key	GenericNode
GenericNode next	GenericNode
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Fields

protected AtomicBoolean locked The lock of the node initialised to false.

---

protected long lockedBy The owner of the lock initialised to 0 (no owner).

---

protected LockNode nextLock The pointer to the next lock in the list initialised to null.

---

protected boolean active Shows whether the lock is still an active node of the list. If a lock is removed from the list during reballancing it is set to inactive, that is active is set to false. It is initialised to true.

Construktors

public LockNode(int key) Class constructor. It calls the corresponding constructor of the superclass GenericNode.

Parameter int key

---

public LockNode(Object newElement, int key) Class constructor with a specified element. It calls the corresponding constructor of the superclass GenericNode.

Parameter Object newElement
int key

Methods

public LockNode nextLock( ) Returns the pointer to the next lock.

Return returns nextLock which is the pointer to the next lock in the list

---

public boolean isActive( ) Indicates whether a node is still active or not.

See also #active

Return returns the value of active

A.4 LockListV2

Full name: `public class LockListV2`

Package lists

Inherits Object←AbstractList

LockListV2 is the final implementation of the linked list for parallel access. The class extends the abstract class AbstractList and implements all of the abstract methods in AbstractList. The list is ordered by the keys of the elements.

Author fabian

Version 2

Inheritancetable

Element	Inherited from
void decrementSize( )	AbstractList
void delete(int)	AbstractList
GenericNode first( )	AbstractList
void incrementSize( )	AbstractList
void insert(GenericNode)	AbstractList
boolean isEmpty( )	AbstractList
void print( )	AbstractList
void setSize(int)	AbstractList
int size( )	AbstractList
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Fields

private final LockNode header The first element in the list. It also is the first lock of the list. It has key 0.

---

public final LockNode sentinel The last element of the list. It has key Integer.MAX_VALUE.

---

private AtomicInteger delins The number of times delete and insert have been called on the list. It's reset to 0 after a call to reballance.

See also #reballance

---

private AtomicBoolean reballance Determines whether reballancing is taking place. reballance may only be called once at a time.

See also #reballance

---

private int locks The number of locks in the list. It is set by reballance before exiting.

See also #reballance

---

private AtomicInteger threads The number of threads working on the list. It is used to determine the number of locks that are required by reballance.

See also #reballance

Construktors

public LockListV2( ) Class constructor. It initializes header and sentinel.

Methods

public void setThreads(int n) Sets the number of threads to n and calls reballance.

See also #reballance

Parameter	int n	the new number of threads

---

public void delete(int key) throws EmptyListException, InvalidKeyException Deletes the element from the list that corresponds to the provided key.

See also exception.EmptyListException
exception.InvalidKeyException

Parameter	int key	specifies which element to delete

Exceptions	EmptyListException	if the list is empty there's nothing to delete
	InvalidKeyException	if there's no element with the specified key in the list

---

public LockNode first( )

---

public void insert(GenericNode newNode) throws NonUniqueKeyException, InvalidKeyException Inserts a new GenericNode into the list.

Parameter	GenericNode newNode	the node to insert

Exceptions	NonUniqueKeyException	if there's already an element with the same key in the list. There can't be more than one element with the same key.
	InvalidKeyException	if the key of newNode is equal to the key of the sentinel which is an illegal key.

---

public void reballance( ) Reballances the locks in the list. After a certain amount of deletes and inserts there might be very large or very small groups of nodes between two locks. If this is the case, the list has to be locked completely and a reballancing has to be done. This means that the correct number of locks are distributed evenly over the list. This method resets the value of delins to 0.

See also #delins

---

public void print( )

---

public boolean isEmpty( )

---

public void release(long id) Releases all the locks in the list which are locked by the specified id.

Parameter	long id	the thread id for which all locks have to be released.

---

public boolean lock(LockNode lockNode, long id) Locks the lockNode and specifies the locker as id.

Return returns true if locking was successfull and false if the lock was locked already

Parameter	LockNode lockNode	the node to lock
	long id	the id if the locking thread

---

public long idLock(LockNode lockNode, long id) Locks the lockNode and specifies the locker as id. This method is similar to lock. The difference lies in the return value.

Return returns the id of the thread to which the lock currently belongs

Parameter	LockNode lockNode	the node to lock
	long id	the id of the locking thread

---

public boolean unlock(LockNode lockNode, long id) Unlocks the lockNode if it was locked by id. The operation only succeeds if the lock belongs to the unlocking thread.

Return returns true if the lock belongs to the unlocking thread and false otherwise

Parameter	LockNode lockNode	the node to unlock
	long id	the id of the unlocking thread

---

public int getLocks( ) Returns the number of locks currently in the list.

Return the number of locks in the list

A.5 LinkedListException

Full name: `public class LinkedListException`

Package exception

Inherits Object←Throwable←Exception

Implements a generic Exception that can occur while working with the classes AbstractList and LockListV2.

Author fabian

Version 1

Inheritancetable

Element	Inherited from
Throwable fillInStackTrace( )	Throwable
Throwable getCause( )	Throwable
String getLocalizedMessage( )	Throwable
String getMessage( )	Throwable
StackTraceElement[] getStackTrace( )	Throwable
Throwable initCause(Throwable)	Throwable
void printStackTrace( )	Throwable
void printStackTrace(PrintStream)	Throwable
void printStackTrace(PrintWriter)	Throwable
void setStackTrace(StackTraceElement[])	Throwable
String toString( )	Throwable
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Construktors

public LinkedListException(String message) Class constructor.

Parameter	String message	the message contains information why the exception was thrown

---

A.6 EmptyListException

Full name: `public class EmptyListException`

Package exception

Inherits Object←Throwable←Exception←LinkedListException

EmptyListExceptions are thrown when an illegal operation is performed on an empty list. This class extends LinkedListException.

Author fabian

Version 1

Inheritancetable

Element	Inherited from
Throwable fillInStackTrace( )	Throwable
Throwable getCause( )	Throwable
String getLocalizedMessage( )	Throwable
String getMessage( )	Throwable
StackTraceElement[] getStackTrace( )	Throwable
Throwable initCause(Throwable)	Throwable
void printStackTrace( )	Throwable
void printStackTrace(PrintStream)	Throwable
void printStackTrace(PrintWriter)	Throwable
void setStackTrace(StackTraceElement[])	Throwable
String toString( )	Throwable
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object

void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Construktors

public EmptyListException( ) Class constructor. Calls the constructor of the superclass with the message "List is empty.".

See also LinkedListException

A.7 InvalidKeyException

Full name: `public class InvalidKeyException`

Package exception

Inherits Object←Throwable←Exception←LinkedListException

InvalidKeyExceptions are thrown whenever the key of an element is that of the sentinel which is illegal. This class extends LinkedListException.

Author fabian

Version 1

Inheritancetable

Element	Inherited from
Throwable fillInStackTrace( )	Throwable
Throwable getCause( )	Throwable
String getLocalizedMessage( )	Throwable
String getMessage( )	Throwable
StackTraceElement[] getStackTrace( )	Throwable
Throwable initCause(Throwable)	Throwable

void printStackTrace( )	Throwable
void printStackTrace(PrintStream)	Throwable
void printStackTrace(PrintWriter)	Throwable
void setStackTrace(StackTraceElement[])	Throwable
String toString( )	Throwable
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Construktors

public InvalidKeyException( ) Class constructor. Calls the constructor of the superclass with the message "Invalid Key.".

See also LinkedListException

A.8 NonUniqueKeyException

Full name: `public class NonUniqueKeyException`

Package exception

Inherits Object←Throwable←Exception←LinkedListException

NonUniqueKeyException are thrown when a node with the same key as one already in the list is tried to be inserted into the list. This class extends LinkedListException.

Author fabian

Version 1

Inheritancetable

Element	Inherited from
Throwable fillInStackTrace( )	Throwable
Throwable getCause( )	Throwable
String getLocalizedMessage( )	Throwable
String getMessage( )	Throwable
StackTraceElement[] getStackTrace( )	Throwable
Throwable initCause(Throwable)	Throwable
void printStackTrace( )	Throwable
void printStackTrace(PrintStream)	Throwable
void printStackTrace(PrintWriter)	Throwable
void setStackTrace(StackTraceElement[])	Throwable
String toString( )	Throwable
Object clone( )	Object
boolean equals(Object)	Object
void finalize( )	Object
Class getClass( )	Object
int hashCode( )	Object
void notify( )	Object
void notifyAll( )	Object
String toString( )	Object
void wait(long)	Object
void wait(long, int)	Object
void wait( )	Object

Construktors

public NonUniqueKeyException( ) Class constructor. Calls the constructor of the superclass with the message "Key already in list.".

See also LinkedListException

References

[1] Mark Moir and Nir Shavit, *Concurrent Data Structures*, 2001, `http://www.cs.tau.ac.il/ shanir/concurrent-data-structures.pdf`.

[2] Evolutionary Computation Research Team: Genetic Algorithm Toolbox, 1994, `http://www.shef.ac.uk/acse/research/ecrg/gat.html`.