Concurrency paradigms,
A comparison in Scala

Mohsen Lesani, Martin Odersky, Rachid Guerraoui
EPFL, IC

1 Abstract

There is a rapid rise of multi-cores in recent hardware architectures. Multi core architectures pose challenges how to exploit their computational power. Software should shift to be as concurrent as possible; and therefore should have control mechanisms to ensure correct results for concurrent operations. There are different concurrency control mechanisms such as locking, wait-free algorithms, actors and transactional memory. There is a need to compare these approaches in terms of performance and ease of use. This paper presents a software transactional memory in Scala. Also, two fundamental cases of credit transfer and producer-consumer are implemented in Scala with the four approaches and the quantitative and qualitative results of the experiments are presented.

2 Introduction

Locking, Wait free algorithms, actors and transactions are existing paradigms for concurrent programming. It is still not known what the best programming model for concurrency is. Ease of use and performance are the criteria that the existing models should be judged by. The strength of each should be identified and possibility of their integration should be studied.

This study implements two fundamental cases by each of the paradigms and compares the paradigms according to each of development and performance results.

The paper starts by giving an introduction to each of the paradigms. Then the cases and their implementation with each of the paradigms are explained. Results are presented and finally conclusions and future works will come.

3 Paradigms

Fundamentally, the two mechanisms that are expected from a concurrency control model are isolation and signaling. Isolation requires that operations in concurrent processes cannot access the data in an intermediate (and probably inconsistent) state while an operation is being done. Signaling is the mechanism that a process uses to inform a waiting process of an event. Signaling can be implemented using the isolation abstraction but in a polling (i.e. busy waiting) and not interrupting form. To have an efficient implementation of signaling, scheduler should be engaged and not schedule waiting processes until they are signaled. That is why signaling is supported in Java Object class besides locks. As signaling cannot be efficiently implemented by the isolation abstraction, it should be a first class feature and be implemented on the basic signaling mechanisms of the underlying (virtual) machine.

3.1 Locks and Conditions

The pessimistic approach to preserve isolation is to prevent executions that may violate it. To isolate some operations, a pessimistic approach can be to allow them to be executed only one at a time. The approach is called mutual exclusion. Lock is an abstraction to provide mutual exclusion. When a process acquires a lock, any later process that ties to acquire the lock is suspended until the first process releases it. Therefore, operations can be isolated by acquiring and releasing a lock respectively before and after the operations. Using one lock for all operations is too restrictive and may sacrifice concurrency i.e. the main goal. Using few locks (and one in the extreme) is called coarse-grained locking. Actually, each operation needs to prevent others only from the data that it is going to access. This observation, leads to the idea of defining separate locks for separate parts of the shared data which is called fine-grained locking. To be more precise, a lock can be defined for each largest part of data that has the same set of accessing operations. Each operation is to acquire and release the set of locks that their respective data it is going to access. The problem with acquiring multiple locks is that if they are not acquired in the same order in different operations deadlock may happen. There is a tradeoff between deadlock safety of coarse-grained locking and performance of fine-grained locking.

An inherent shortcoming of locks is their lack of compositionality. When objects from two or more classes should be composed to define a new class and there is an operation of the new class that should perform some operations from composed objects in isolation, fine-grained locking is possible only when composed classes expose their internal locks. Exposing locks out of class has an intense effect on modularity.

A process trying to acquire a lock that is previously acquired by another process is blocked. Cache misses, page faults and preemption by the operating system delay processes. If the current lock owner process is delayed, the blocked process is also delayed. Such long blockings are undesirable in real-time and event driven systems. When a low priority task holds a lock that is required by a high priority task, the latter has to block until the former releases the lock. The relative priorities of the two tasks are effectively inverted which is called priority inversion. Besides, an interrupt handler that needs to acquire a lock that is previously acquired by a preempted process can not proceed. This is while separate event handling processes maintaining GUI, real-time audio rendering, and disk and network I/O need to proceed timely. This is especially true for interactive and rich multimedia applications like electronic games.

Conditions are objects that a process that waits on is suspended until another process signals on the same condition.

3.2 Non-blocking Algorithms

To circumvent blocking problems of locks that are mentioned before, non blocking algorithms are devised for some data structures. Non-blocking algorithms are lockless and ensure (a level of) progress even if some processes are blocked. The basic idea in non-blocking algorithms is redundant data and rechecking. An operation is wait-free if every concurrent execution of it finishes in a finite number of steps. Hence, wait-free operations do not produce priority inversion and not hinder event handlers. Wait-free algorithms are the strongest category of non-blocking algorithms. Stronger algorithms are more attractive but usually harder to devise and sometimes inefficient hence weaker algorithms are sometimes settled for. An operation is lock-free if at least one concurrent execution of it finishes in a finite number of steps. An operation is obstruction-free if when executed alone, it finishes in a finite number of steps. Every wait-free algorithm is lock-free and every lock-free algorithm is obstruction-free. So they are ordered in strength as wait-free, lock-free and obstruction-free.

Condition objects can be used by non-blocking algorithms as means of signaling. Conditions only block, the processes that call wait and those processes should block by the definition of the wait operation.

3.3 Actors

Processes with a unique memory space can communicate on the shared memory. In contrast, processes on different memory spaces communicate by message passing. Message passing communication can be applied to shared memory processes too. Actors are abstraction on threads with message passing features. Although actors can share data, it is a design recommendation not to have shared objects in actors and to do communication only by message passing.

Actors provide isolation by the fact that for each actor instance, one instance of the actor code runs. This means that operations executed in the actor code are serialized by their execution time and hence are done in isolation. Accesses to the actor message boxes by send and receives are also previously synchronized by the supporting library or language. This mechanism for isolation is coarse-grained. For example, handling messages of different types may need to access different data inside the actor and hence can be done concurrently but are done in sequence in the actor code. On the other hand actors provide a rich mechanism for signaling. While primitive condition objects should be used with shared memory that is protected with locks to pass effective messages, actor message passing mechanisms are self contained and easy to use.

3.3.1 STM

The optimistic approach to maintain isolation is to let operations execute without any prevention at the beginning and to rollback and retry an operation if it is invalidated i.e. found not to have or will be executed in isolation. As only one of transactions that write on a transactional object can finally commit, it is better to only allow one transaction at a time to write on a transactional object. Only one writing transaction at a time leads to a single backup of the fields in transactional objects. When the writing transaction of an object is still active and a transaction wants to write on the object, it should select to abort the last writing transaction or itself. As a transaction status in its descriptor may be set to aborted by other transactions, any transaction checks not to have an aborted status before any read, write and also commit. Committing of a transaction i.e. updating written objects to new values should be done at once; otherwise some transactions may experience inconsistencies by accessing some new and old objects. This is usually done by atomically setting the transaction status as committed in the transaction descriptor and the fact that if the writing transaction of an object is committed, any first read or write on the object updates the current fields to the new values of the writing transaction. Multiple transactions are allowed to read from an object. If the writing transaction of an object is active and a transaction wants to read from the object, the transaction can get the current value of the object or get the new value from the writing transaction. In the second case, the transaction will be dependant on the writing transaction and following this dependency is not easy. Hence, usually the current stable state of the object is taken. In this case, the reading transaction will not be able to commit if the writing transaction commits sooner.

Assume the case where a transaction reads some objects and then after another transaction commits, it reads some committed objects. The transaction experiences reading inconsistent data if the committed transaction has updated any of the previously read objects. There are two approaches to this read problem: visible and invisible reads. Invisible reads, the DSTM2 approach, adds any read object to the read list in the transaction descriptor. On any subsequent read or write, it is checked at the beginning whether all the read objects are still current; if not, the transaction is aborted. Visible reads, Scala STM, approach, adds transaction descriptor of any transaction that read an object to the read list of the object. When the object is written by a transaction, it sets the status of all the transaction descriptors in the read list to aborted. With visible reads, also on the read-write conflict, the writing transaction should be set to aborted.

There is no signaling mechanism in STM. Hence, waiting for a condition is implemented as throwing an abort exception and retrying the atomic block. In other words, signaling is implemented by isolation and hence it is busy waiting that may decrease performance. Signaling is one of the operations that can not be rolled back. STM atomic blocks can not contain any operation that can not be rolled back such as I/O.

4 Cases

The two cases that are taken for comparison are chosen according to expected fundamental mechanisms that are previously mentioned i.e. isolation and signaling. Transfer of credit in bank accounts should be done in isolation or the integrity of the bank account balances is violated. In addition to isolation, producer-consumer case needs signaling to inform waiting consumers of a new production.

4.1 Bank Account Credit Transfer

Transferring of credits between bank accounts is the classical example of concurrent access to shared data. An amount of credit should be debited from an account and credited to another account. More important than the other properties that should be preserved is the isolation property of a transfer. Otherwise some credit may be debited form or credited to some accounts that should be.

4.1.1 Locks

4.1.1.1 Coarse-grained

In the coarse-grained locking all the transfers even if they are not conflicting are linearized by the bank lock.

```
this.synchronized {
   account1.withdraw(amount)
   account2.deposit(amount)
}
```

4.1.1.2 Fine-grained

In the fine-grained locking rather than having a lock for the whole bank, each account has a lock. It is notable that locks are always acquired in the same order.

```
   if (accNo1 <= accNo2) {
      account1.lock.lock
      account2.lock.lock
   } else {
      account2.lock.lock
      account1.lock.lock
   }

   account1.withdraw(amount)
   account2.deposit(amount)

   account1.lock.unlock
   account2.lock.unlock
```

4.1.2 Actors

For each bank account, a transferer actor is created. The transferer of an account is responsible for credit transfers of the account to and from other accounts with larger account numbers. Client transfer requests are forwarded by the bank actor to the right transferer.

```
if (accNo1 < accNo2)
   transferers(accNo1) ! ITransferRequest(/*...*/)
else
   transferers(accNo2) ! ITransferRequest(/*...*/)
```

To transfer, the transferer sends a message to the transferer of the other account to ask him to wait. When a transferer receives a wait request, it waits until the requesting transferer sends a go on message. When the wait request is acknowledged by the other transferer, the transfer is done and then a message is sent to the waiting transferer to go on. When a transfer is being done, only one transferer actor is accessing the two accounts; and hence isolation is preserved.

The transferer of an account only waits for transferers of accounts with larger account numbers. Hence there can be no cycle in the waiting chains and there is no deadlock in waiting transferers.

```
def act() {
      react {
            case itr @ ITransferRequest(/*...*/ accountNo2, amount, forward) =>
                  //...
                  transferers(accountNo2) ! WaitRequest
                  react {
                        case WaitOK =>
                              if  (forward)
                                    transfer(accountNo1, accountNo2, amount)
                              else
                                    transfer(accountNo2, accountNo1, amount)
                              sender ! GoOnRequest
                              // ...
                              act
                  }
            case WaitRequest =>
                  sender ! WaitOK
                  react {
                        case GoOnRequest =>
                              act
                  }
            case TerminateRequest =>
      }
}
```

4.1.3 STM

Credit transfer is simply an atomic block in STM.

```
atomic {
   accounts(accNo1).withdraw(amount)
   accounts(accNo1).deposit(amount)
}
```

4.2 Producer-Consumer

Producer-consumer is a pattern that reoccurs in designs of various software systems. Several producers concurrently produce productions that are concurrently consumed by consumers. Addition and elimination from the entity that holds the productions should be done in isolation to preserve consistency and prevent production loss. When there is no production available consumers should wait until one is produced.

4.2.1 Locks

4.2.1.1 Coarse-grained

In the coarse-grained locking, the implicit lock of the Queue synchronizes the whole bodies of enqueue and dequeue operations.

4.2.1.2 Fine-grained

To locks are defined for the rear and front cursors of the queue. To maximize parallelism of enqueue and dequeue, instead of acquiring both locks for the worst case, we first acquire the most often needed lock which is rearLock for enqueue and frontLock for dequeue. If the only lock that is acquired is not enough to complete the operation, the other lock should also be acquired later.

If the current state of the object necessitates acquisition of the second lock to perform the operation, the previously executed part of the operation may be not isolated. Hence, all or some lines of the executed part may be needed to be repeated after the second lock acquisition. For example, when there is only one node in the queue, what about concurrent access to its next field? In the dequeue operation, it is after the "else" that it is known that rearLock should also be acquired. If the rearLock had been acquired after the "else" and then the rearCursor had been made null, that could generate a race. An enqueue could be done just after the "else" and then the rearCursor would be made null. That is lost of the just enqueued value! The fact that the next field is null cannot be relied on just after the "else". Null inequality should be checked again after the second lock acquisition. Such a non-isolation cannot happen in the enqueue operation. This is because in the “else” where it is known that the second lock should be acquired, rear is null and that means the queue is empty. When the queue is empty, dequeue operation cannot and does not change this state. The current thread already in enqueue operation and have acquired the first lock; hence no other thread can enter enqueue and change the state by this operation. Hence no operation can change the current state and enqueue operation can safely rely on its current information about the object state and go on with its execution.

Besides, to prevent dead-lock, the order of acquiring the locks should be the same in all operations. To have the same order of lock acquisition, as the two operations have acquired different locks at the beginning, one of the operations should release its current lock and restart the operation by acquiring the other lock first and then its current lock again. Concerning performance, it seems better not to restart the operation that does not need to repeat some previously done parts of the operation and leave it as is; but to restart the operation that needs some repetitions anyway.

The lock related to the condition should have been acquired before waiting on the condition. As non-emptiness condition is waited on at the beginning of dequeue and frontLock and rearLock are respectively acquired at the beginning of dequeue and conservative dequeue, a condition is defined on each of the locks. Each is also signaled when an enqueue is done on an empty queue.

```
def enqueue(v: Int) = {
      val newNode = new Node(0, null)
```

```
        newNode.value = v
        rearLock.lock
        val rear = rearCursor.node
        if (rear != null) {
              rear.next = newNode
              rearCursor.node = newNode
        }
        else {
              frontLock.lock
              frontCursor.node = newNode
              rearCursor.node = newNode
              //To awaken the threads that are waiting to dequeue.
              notEmptyForFront.signalAll
              notEmptyForRear.signalAll
              frontLock.unlock
        }
        rearLock.unlock
        enqueueCount += 1
  }


  def dequeue(): Int = {
        frontLock.lock
        while (frontCursor.node == null) {
              notEmptyForFront.await()
        }
        var front = frontCursor.node
        var value = front.value
        front = front.next

        if (front != null) {
              frontCursor.node = front
              frontLock.unlock
        } else {
              frontLock.unlock
              value = conservativeDequeue
        }
        dequeueCount += 1

        value
  }

  def conservativeDequeue(): Int = {
        rearLock.lock
        while (rearCursor.node == null) {
              notEmptyForRear.await()
        }
        frontLock.lock

        var front = frontCursor.node
        var value = front.value
        front = front.next

        frontCursor.node = front
        if (front == null)
              rearCursor.node = null

        frontLock.unlock
        rearLock.unlock


        value
  }
```

4.2.2 Non-blocking algorithms

This implementation employs the "wait-free" algorithm described in “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” by Maged M. Michael and Michael L. Scott.

4.2.3 Actors

The mediator actor is the single reference point for producers and consumers.

When it receives a request from the producers (or consumers) and there is no previously stored request from consumers (or producers), the mediator stores the request in an internal queue for producers (or consumers). If there is a stored request from the other party, it simply services the current and the stored requests.

```
def act() {
    if (count != TOTAL_PRODUCTION_COUNT)
            react {
                    case p: Production =>
                            if (! consumerQueue.isEmpty) {
                                    consumerQueue.dequeue ! p
                                    count = count + 1
                            }
                            else
                                    prodcutionQueue.enqueue(p)
                            act()
                    case ConsumeRequest =>
                            if (! prodcutionQueue.isEmpty) {
                                    sender ! prodcutionQueue.dequeue
                                    count = count + 1
                            }
                            else
                                    consumerQueue.enqueue(sender)
                            act()
            }
}
```

4.2.4 STM

The STM implementation of the queue is straightforward from the sequential implementation. The enqueue and dequeue operations are put inside atomic blocks. When the queue is empty instead of QueueEmptyException, AbortException should be thrown to retry the transaction. Cursor and Node classes with the same definition as the sequential ones are annotated as atomic.

5 Results

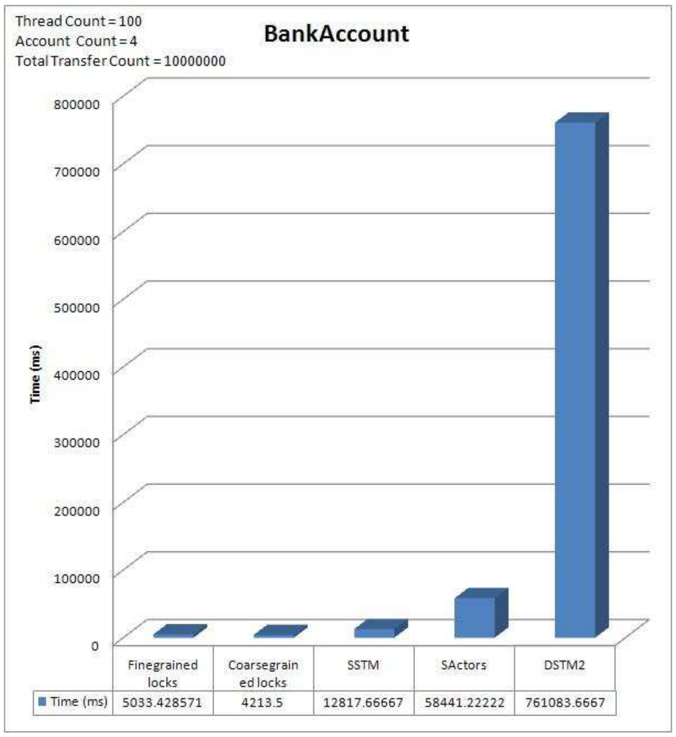
The paradigms are compared in ease of programming and performance for the implemented cases.

5.1 Ease of use

According to the presented implementations, STM was the simplest to implement Credit Transfer case with; on the other hand, Actor implementation was the most straightforward implementation of Producer-Consumer case. A subjective order of simplicity of paradigms for Credit Transfer case is STM, Coarse-grained locking, Fine-grained locking, Actors. For the Producer-Consumer case the order would be Actors, STM, Coarse-grained locking, Fine-grained locking and the wait-free algorithm.

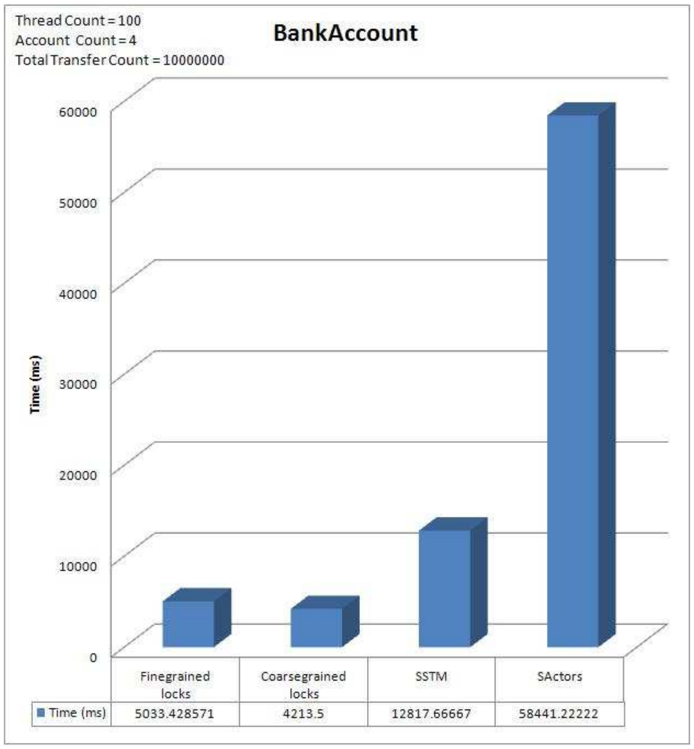
5.2 Performance

Experiments for Credit Transfer case are done with 100 threads that each performs 100,000 transfers on 4 accounts. Experiments for Producer-Consumer case are done with 50 producers and 50 consumers that each produce or consume 40,000 productions. The execution times of different implementations of Credit Transfer and Producer-Consumer cases are presented respectively in Figure 1 and 3. As the execution time of DSTM2 makes it difficult to compare others, the other execution times from Figures 1 and 3 are respectively repeated in Figures 2 and 4. The results presented in Figures 1 to 4 are from executions on a 2-core machine (Exact machine spec to be added).



	Finegrained locks	Coarsegrained locks	SSTM	SActors	DSTM2
■ Time (ms)	5033.428571	4213.5	12817.66667	58441.22222	761083.6667

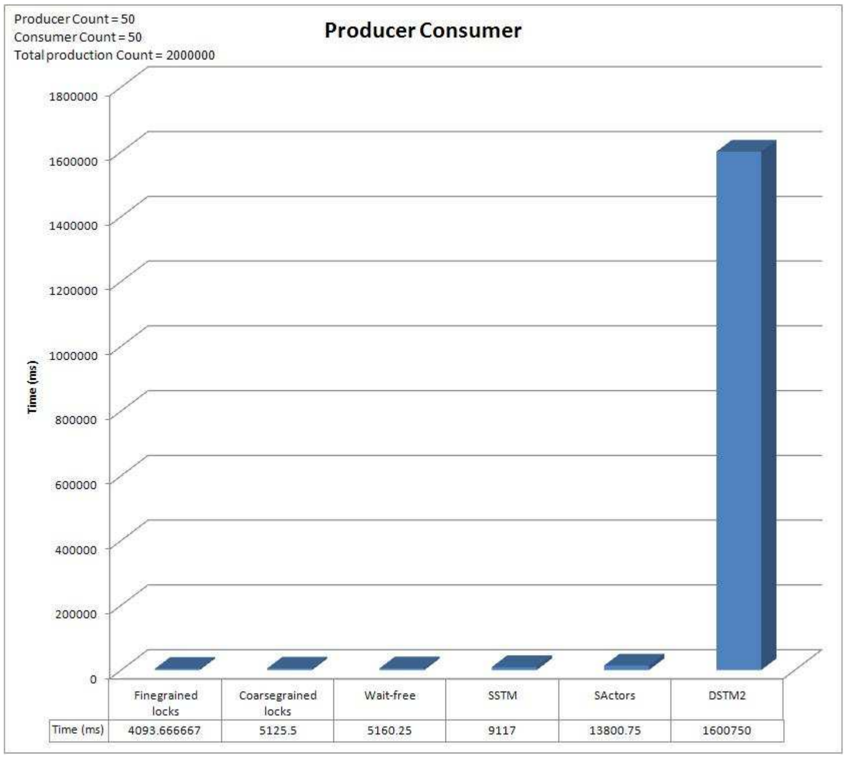
Figure 1. Execution times for Bank Account Credit Transfer (with DSTM2)



	Finegrained locks	Coarsegrained locks	SSTM	SActors
■ Time (ms)	5033.428571	4213.5	12817.66667	58441.22222

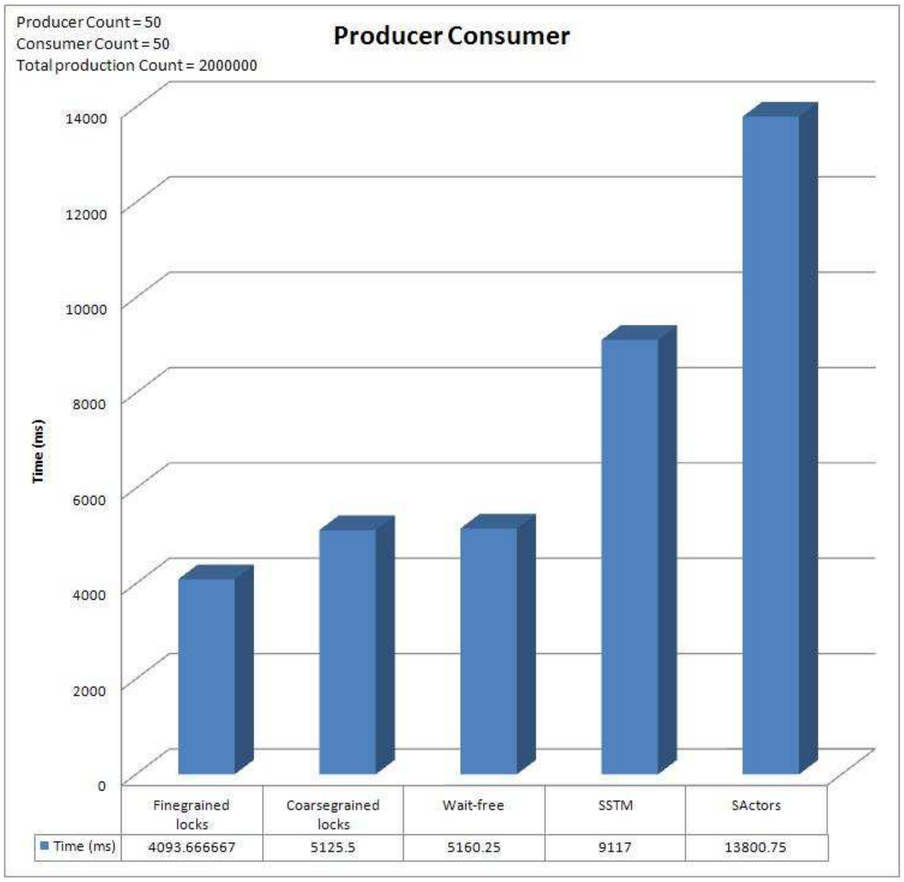
Figure 2. Execution times for Bank Account Credit Transfer (without DSTM2)

It is interesting that, in contrast to expectations, the performance of coarse-grained locking is better than that of fine-grained locking in this case.



	Finegrained locks	Coarsegrained locks	Wait-free	SSTM	SActors	DSTM2
Time (ms)	4093.666667	5125.5	5160.25	9117	13800.75	1600750

Figure 3. Execution times for Producer-Consumer (with DSTM2)



	Finegrained locks	Coarsegrained locks	Wait-free	SSTM	SActors
■ Time (ms)	4093.666667	5125.5	5160.25	9117	13800.75

Figure 4. Execution times for Producer-Consumer (without DSTM2)

6 Conclusions and Future Work

Although locking has good performance, it is hard to program it fine-grained and more importantly it has some inherent shortcomings such as lack of compositionality, possibility of priority inversion and blocking event handlers. Wait-free algorithms are also well at performance but developing such algorithms are hard enough to expect them only from experts. Wait-free algorithms seem to be the best paradigm for thread safe libraries not only because they don't block but more importantly because they compose well with classes of any paradigm. Hence STM or Actors are the better choices for the average programmers.

Based on the performed experiments, from the programmer point of view, some applications are suited to be programmed with STM while others are more easily programmed with Actors. This is while the simulations show better performance for STM than for Actors. The results also show performance superiority of Scala STM to DSTM2, although Scala STM is not yet full-featured.

Supporting nested transactions and throwing exception out of atomic blocks and also implementing and experimenting other strategies of contention management in Scala are future works.

Actor and STM approaches to concurrency have strength in different aspects. Actors support high level message passing while transactions support isolation well. As argued in the STM section, Signaling is not supported by STM and hence waiting is done in a busy-waiting style that may influence performance. An issue is how to integrate the two approaches in a semantically well-defined and efficient way.