Concurrent Programming Paradigms, A Comparison in Scala

Mohsen Lesani
EPFL, IC

Martin Odersky
EPFL, IC

Rachid Guerraoui
EPFL, IC

ABSTRACT

There is a rapid rise of multi-cores in recent hardware architectures. To exploit computational power of multi-core architectures, software should shift to be as concurrent as possible; and therefore should have concurrency control mechanisms. There are different concurrency programming paradigms such as locking and conditions, non-blocking algorithms, actors and software transactional memory (STM). There is a need to compare these approaches in terms of ease of use and performance. This work implements three fundamental cases of credit transfer, producer-consumer and token ring with different paradigms in Scala and the quantitative and qualitative results of the experiments are presented. Besides an STM implementation in Scala is presented.

Keywords

Concurrent Programming, Actors, STM.

1. INTRODUCTION

As modern hardware architectures require concurrent software, the need for a concurrent programming model is sensed more than ever before. There are four known concurrency programming models: locking and conditions, non-blocking algorithms, actors and software transactional memory. Locks and conditions is the primitive concurrency programming model that is provided by most languages. Non-blocking algorithms are ad-hoc algorithms that are lockless and hence provide progress. Actor [6] is an abstraction over thread with high level message passing features. A transaction [2][3][4][7][8] is a code block that its reads and writes to the shared memory occur logically at a time instant. This study is to compare these paradigms and identify the strengths of each.

To compare these paradigms three fundamental cases are selected: bank account credit transfer, producer-consumer and token ring. Each case is implemented with each of the paradigms. The comparison is based on ease of use and performance results.

The report starts by giving an introduction for 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 follow.

2. Paradigms

Fundamentally, the two mechanisms that are expected from a concurrency programming model are isolation and signaling. Isolation mechanism prevents concurrent operations from accessing shared data in an intermediate (and probably inconsistent) state. Signaling is the mechanism that a process employs to inform another process about an event.

Signaling can be implemented using the isolation mechanism and a shared memory variable. The signaling thread writes on the variable in isolation and the waiting thread continuously reads from the variable in isolation. The problem with this implementation of signaling is that it is polling (i.e. busy waiting). To have an efficient implementation of signaling, the scheduler should also be engaged. It should not schedule the waiting process until the variable is written by the signaling thread. In fact, this is why intrinsic condition is supported in Java Object class besides the intrinsic lock.

2.1 Locks and Conditions

2.1.1 Isolation

The pessimistic approach to preserve isolation is to prevent executions that may violate it. To isolate some operations, a pessimistic approach is to allow them to be executed only one at a time. This 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 the lock. Therefore, a block can be mutually exclusive by acquiring and releasing a lock respectively before and after the block. Hence, blocks with the same lock can execute operations in isolation.

Using one lock for all operations is too restrictive. It can serialize processes that could potentially execute in parallel; and hence may sacrifice concurrency. Using few locks (and one in the extreme) is called coarse-grained locking. In fact, each process needs to prevent others from accessing only the data that it is going to access. This observation, leads to the idea of defining separate locks for separate parts of shared data which is called fine-grained locking. To be more precise, a lock can be defined for each largest part of shared data that has the same set of concurrently accessing operations. Each operation is to acquire and release the set of locks that correspond to parts of data that it is going to access. The problem with acquiring multiple locks is that if they are not acquired in the same order in different processes deadlock may happen. There is a tradeoff between deadlock safety of coarse-grained locking and performance of fine-grained locking.

Locks are known to have some inherent shortcomings. The first shortcoming of locks is their lack of compositionality. For instance, suppose that objects of two or more classes should be composed to define a new class. If there is an operation of the new class that should perform some operations of the composed objects in isolation, fine-grained locking is possible only when

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

composed classes expose their internal locks. Exposing locks out of classes has an intense effect on modularity.

The second problem of locks is that they prevent progress. A process trying to acquire a lock that is previously acquired by another process is blocked. If the current lock owner is delayed, every process that is blocked for the lock is also delayed. Cache misses, page faults and preemption by the operating system delay processes. Such long delays 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. In such situations, the relative priorities of the two tasks are inverted. This is called priority inversion. Besides, if an interrupt handler needs to acquire a lock that is previously acquired by a preempted process, the handler cannot 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.

2.1.2 Signaling

Condition is the signaling mechanism that is used with locks. A process that calls wait on a condition object is suspended until another process calls signal on the same condition.

2.2 Non-blocking Algorithms

To circumvent blocking problems of locks that are mentioned before, ad hoc non-blocking algorithms are devised for some data structures. Non-blocking algorithms are lockless and their basic idea 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-freedom is the strongest progress property. Although stronger progress properties are more attractive, they are usually harder to devise and sometimes inefficient. Hence, weaker properties 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 it is 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 the properties are ordered in strength as wait-freedom, lock-freedom and obstruction-freedom.

2.3 Actors

Processes with a unique memory space can communicate via the shared memory. In contrast, processes on different memory spaces communicate by message passing. Communication via message passing can be applied to shared memory processes too. Actor is an 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.

2.3.1 Isolation

Actors provide isolation by the fact that for each actor instance, there is one (virtual) thread that executes the actor code. This means that operations executed in the actor code are serialized and hence are done in isolation. Accesses to the actor message boxes by send and receive operations are also already 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 by the actor.

2.3.2 Signaling

On the other hand actors provide a rich mechanism for signaling. Primitive condition objects can only send plain signals. To pass informative messages, they should be used together with shared objects. This is while actor message passing mechanisms are self contained and easy to use.

2.4 Software Transactional Memory

2.4.1 Isolation

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. An operation is invalidated when it is found that its execution has not been or will not be in isolation. The following paragraphs present how this work has implemented transactional memory.

The atomic method creates a transaction descriptor object for every transaction. The transaction descriptor is saved as a thread local variable to be accessed later by read and write methods of transactional objects.

In order to rollback, transactional objects should be backed up before being written. As only one of concurrent transactions that write on a transactional object can finally commit, only one transaction at a time is allowed to write on an object. As there is only one writing transaction at a time, transactional objects have only a single backup of their fields.

Every transactional object stores a reference to the descriptor of its last writing transaction. When the last writing transaction of an object is still active and the current transaction wants to write on the object i.e. on a write-write conflict, the current transaction should select to abort the last writing transaction or itself. It can abort itself by throwing an abort exception. The atomic method catches the exception and retries executing the transaction code. It can abort the last writing transaction by setting the status in its descriptor to aborted. As a transaction status may be set to aborted by other transactions, any transaction checks not to have an aborted status before any read, write and also commit.

Multiple transactions are allowed to read from an object.

The read-write conflict is when a transaction T_r reads an object O and then another transaction T_w writes on O. The problem is that then T_w can commit and update some objects S_1 including O and T_r can then read some committed objects S_2 from S_1. T_r can experience reading inconsistency between the old value of O and S_2. There are two approaches to the read-write problem: visible and invisible reads.

With invisible reads, the DSTM2 approach, transaction descriptor has a read list. Every transaction that reads an object adds the object to the read list in its descriptor. On any subsequent read or write, transactions check at the beginning whether all the read objects are still current; if not, the transaction can abort itself.

With visible reads, this implementation's approach, transactional objects have read lists. Every transaction that reads an object adds its own descriptor to the read list of the object. On read-write conflict, T_w should select to abort itself or to abort all the previously reading transactions T_rs of O. To abort T_rs, T_w changes the status of all the transaction descriptors in the read list of O to

aborted. This is because not only these transactions can observe inconsistency but also they can propagate it to writes and finally commit.

The write-read conflict is when the last writing transaction T_w of an object O is active and another transaction T_r wants to read from O. T_r can get the current stable value of O or its tentative value from T_w. For the latter, the current transaction will be dependent on the writing transaction and following this dependency is hard. Hence, T_r usually reads the current stable state of the object. The problem is that then T_w can commit and update a set of objects S_1 in addition to O. Then T_r can read some objects S_2 from S_1 and experience inconsistency between the old value of O and new values of S_2 objects. There are two approaches to this problem based on the strategy chosen for read-write conflict.

With invisible reads, validation is done on the read list of the transaction on any read. Hence the fact that O is updated is detected in the validation that is done when T_r wants to read any object from S_1. Hence the transaction can abort itself not to observe any inconsistent data.

It is notable that in visible reads, on the write-read conflict, one of the transactions should be aborted. Otherwise, T_r is unable to detect inconsistencies later and reading inconsistency can lead to written inconsistency by T_r that can finally 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 together. This is usually done by atomically setting the transaction status as committed in its descriptor and also the fact that if the last writing transaction of an object is committed, any first read or write on the object updates the current fields with the new values from the last writing transaction.

2.4.2 Signaling

There is no signaling mechanism in the fundamental STM model. Waiting for a condition can be throwing an abort exception and retrying the atomic block. In other words, signaling is implemented by isolation. This is busy waiting and hence affects performance. An optimization to STM conditions is to retry the transaction only when at least one of the previously read transactional objects is written by another transaction [5]. Although many useless retries are eliminated by this optimization, some are still present. This is because updates to any of the previously read objects that are irrelevant to the waited condition cause the transaction to be retried.

The following paragraphs explain how STM conditions are implemented in Scala. When a transaction calls conditionWait, a wait exception is thrown that is caught in the transaction retry loop. On catching a wait exception from a try, the current thread sets the status of the transaction to waiting and waits on the intrinsic condition of the transaction descriptor.

In the visible reads strategy, a read list is maintained by each transactional object. A new list called waiting list is added to each transactional object. A transaction that writes on the object traverses the read list of the object. Active transactions of the list are aborted and waiting transactions are enqueued to the waiting list.

A new list named write list is added to each transaction descriptor. When a transaction writes on a transactional object, it adds the object to the write list of its descriptor. When a transaction succeeds in committing, it traverses the objects in the write list of its descriptor. From each of these objects, it gets the waiting list and notifies the transactions in the waiting lists.

3. Cases

The three cases that are selected 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. The token ring case employs signaling to pass the token to the next stations.

3.1 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.

3.1.1 Locks and Conditions

3.1.1.1 Coarse-grained

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

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

3.1.1.2 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
```

3.1.2 Actors

The bank account case is implemented twice with two different designs.

3.1.2.1 First implementation

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 class to the right transferer.

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

The code of transferer actor is presented in the following code snippet. Assume two bank accounts B_1 and B_2 respectively with transferer actors TB_1 and TB_2 where without loss of generality, account number of B_1 is less than that of B_2. A transfer between accounts B_1 and B_2 is forwarded to TB_1. It is the owner of B_1 and can readily access it. But for accessing B_2, it should communicate with TB_2 to preserve isolation of transfer operations. To be more precise, before performing a transfer, TB_1 sends a message to TB_2 to ask him to wait. When a transferer (i.e. TB_2) receives a wait request, it sends an acknowledge message to the requester (i.e. TB_1) and waits until the requester (i.e. TB_1) sends a "go on" message. When the wait request is acknowledged by TB_2, TB_1 performs the transfer operation and then a message is sent to the TB_2 to go on. When a transfer is being done, only one transferer actor is accessing the two accounts; and hence isolation is preserved.

```
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 BalanceRequest =>
      sender ! accounts(accountNo1).balance
    case TerminateRequest =>
  }
}
```

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

3.1.2.2 Second implementation

Each account is modeled as an actor that handles withdraw and deposit requests.

```
def act() {
  react {
    case Withdraw(amount) =>
      b -= amount
      sender ! WithdrawDone
      act
    case Deposit(amount) =>
      b += amount
      sender ! DepositDone
      act
    case BalanceRequest =>
      sender ! Balance(b)
      act
    case TerminateRequest =>
  }
}
```

As messages are handled one at a time by actors, withdraw, deposit and balance requests are done is isolation.

The transfer operation of the bank sends withdraw and deposit requests to account actors and wait for their acknowledgments before returning.

```
accounts(accNo1) ! Withdraw(amount)
accounts(accNo2) ! Deposit(amount)
receive {
  case WithdrawDone =>
    receive {
      case DepositDone =>
    }
}
```

The first implementation provides isolation but this implementation turns out to be very inefficient. The second implementation provides an eventual guarantee. Finally the sum of all the account balances is the same as before the transfers. The second implementation is more efficient and hence it is used in performance comparisons.

3.1.3 STM

Credit transfer is simply an atomic block in STM.

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

3.2 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 to 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.

3.2.1 *Locks and Conditions*

3.2.1.1 *Coarse-grained*

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

3.2.1.2 *Fine-grained*

The code is presented in the following code snippet. Two locks are defined for the rear and front cursors of the queue. For each of the enqueue and dequeue operations, acquiring both locks is required only in the worst case. To maximize parallelism of enqueue and dequeue, the most often needed lock which is rearLock for enqueue and frontLock for dequeue is acquired first. The other lock can be acquired later, if needed.

```
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
}
```

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 have not been run in isolation. Hence, all or some lines of the executed part may be needed to be repeated after the second lock acquisition. 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 change this state. The current thread is already in enqueue operation and has acquired the first lock; hence no other thread can enter enqueue and change the state. Therefore no operation can change the current state and enqueue operation can safely rely on its current information about the object state and go on its execution.

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 enqueue and dequeue 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. This is why dequeue is repeated in the conservative dequeue method.

Before waiting on a condition, the lock related to it should have been acquired. Non-emptiness condition should be waited on at the beginning of both dequeue and conservative dequeue methods. As the frontLock and rearLock are respectively acquired at the beginning of these methods, a non-empty condition is defined on each of these locks. Both of these conditions are also signaled when an enqueue is done on an empty queue.

3.2.2 *Non-blocking algorithms*

The performance evaluations employ the wait-free algorithm proposed in [1].

3.2.3 *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
    }
}
```

3.2.4 STM

The STM implementation of a queue is straightforward from the sequential implementation. The enqueue and dequeue operations are put inside atomic blocks. When the queue is empty an abort or wait exception should be thrown to retry the transaction. Cursor and Node classes with the same definition as the sequential ones are annotated as atomic.

3.3 Token Ring

Token ring is basically a local area network (LAN) protocol at the data link layer (DLL). Stations on a token ring LAN are logically organized in a ring topology with data being transmitted sequentially from one ring station to the next. A control token circulates around the ring controlling access.

As passing of the token to the next station is essentially signaling, the simulation of token ring protocol can compare different paradigms according to their signaling mechanisms.

3.3.1 Locks and Conditions

The station waits on the intrinsic condition of the incoming port while the token is not inside the port yet. When the station finds the token inside the incoming port (maybe after being notified by the neighbor station), it takes the token from the incoming port and puts it inside the outgoing port. The next station may have been suspended after a wait on the outgoing port. To awake the next station, the station notifies on the outgoing port after putting the token in it.

```
inPort.synchronized {
  while (inPort.value == null)
    inPort.wait
  outPort.synchronized {
    outPort.value = inPort.value
    outPort.notify
    inPort.value = null
  }
}
```

3.3.2 Actors

The token ring is very straightforward with Actors. The actor reacts to receiving of the token by sending it to the next station.

```
def act {
  if (currentRound != roundCount)
    react {
      case Token =>
        nextStation ! Token
        currentRound += 1
        act
    }
}
```

3.3.3 STM

Port is defined as a transactional object. Inside an atomic block, the station reads the value of the incoming port. If there is no value inside it, conditionWait is called. As explained in section 2.4.2, by calling conditionWait, the transaction is essentially aborted and not retried until only after the incoming port object is updated. When a (retrying) transaction succeeds in reading the token from the incoming port, it updates values of both incoming and outgoing ports to null and the token respectively.

```
atomic {
  if (inPort.value == null) {
    //throw new AbortException
    conditionWait
  }
  else {
    outPort.value = inPort.value
    inPort.value = null
  }
}
```

4. Results

The paradigms are compared in ease of use and performance in the following subsections.

4.1 Ease of use

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

4.2 Performance

For each case, separate experiments are done and in each case, variation of definite parameters is studied. A chart shows total time spent by each paradigm against variation of the definite parameter.

The experiments are done on two different machines M_1 and M_2. Machine M_1 is a Dell Latitude E6400 Intel® Core™2 Duo CPU P8600 @2.40GHz and machine M_2 is <lpdquad spec> with eight cores.

4.2.1 Bank Account Credit Transfer

Two experiments that are done on this case are based on variation of two parameters: total transfer count and number of accounts.

Clients request equal number of transfers in both experiments.

4.2.1.1 Total Time vs. Total Transfer Count

Figure 1: Total time vs. total transfer count in M_1Figure 1 depicts the time spent by each paradigm for various number of transfers in machine M_1 where constant parameters are client count that is equal to 20 and account count that is equal to 100. Client count is the number of concurrently requesting threads or actors. To have a closer view of less time consuming paradigms, Figure 2 depicts only these paradigms from Figure 1.

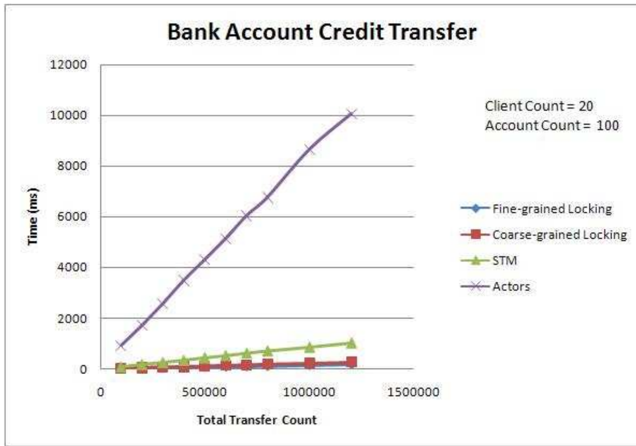

Figure 1: Total time vs. total transfer count in M_1

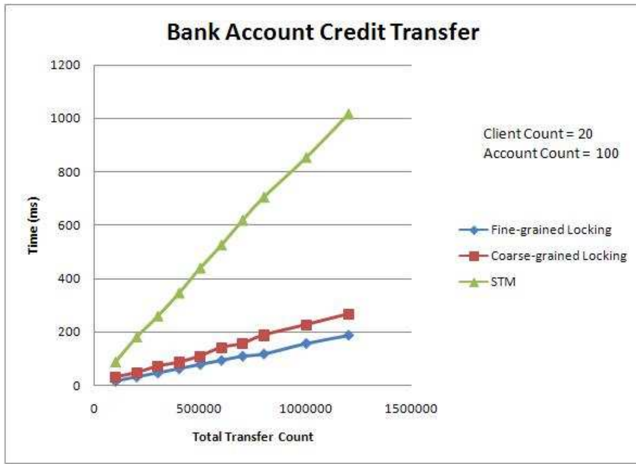

Figure 2: Total time vs. total transfer count in M_1 (without actors)

As expected, there is a monotonic increase of total time against transfer count in all the paradigms.

4.2.1.2 Total Time vs. Number of accounts

Figure 3 depicts the time spent by each paradigm for various number of accounts in machine M_1 where constant parameters are client count that is equal to 100 and total transfer count that is equal to 10000000. A closer view of actors is depicted in Figure 5. A closer view for the other paradigms is depicted in Figure 5.

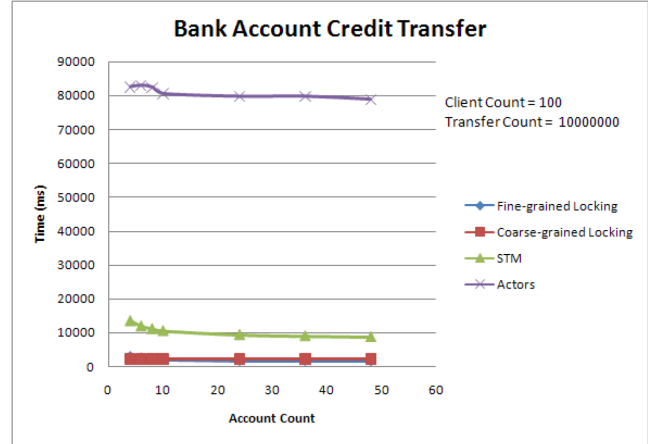

Figure 3: Total time vs. account count in M_1

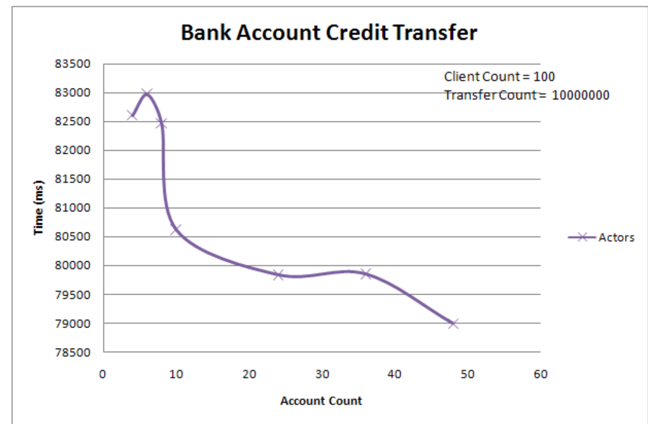

Figure 4: Total time vs. account count in M_1 (actors)

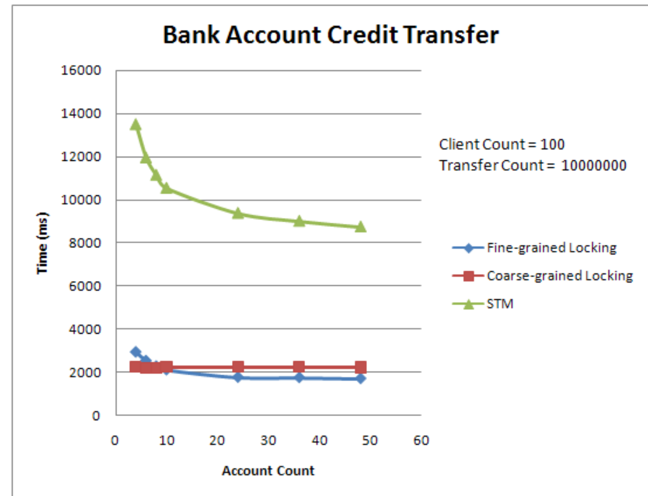

Figure 5: Total time vs. account count in M_1 (without actors)

In coarse-grained locking, it is obvious that increasing account count has no effect on performance. This is because there is only one lock for all the accounts. In the other paradigms, increasing the number of accounts decreases the probability of contention on each account. In fine-grained locking, less contention on an account means less blocking for clients that try to acquire its lock and hence faster lock acquisition and faster transfers. It is

interesting that with small number of accounts, coarse-grained locking is more efficient than fine-grained locking. This is because contention is high and very few transfers can be performed concurrently anyway. This is while one lock should be acquired for coarse-grained locking while fine-grained locking needs two. In STM, less contention means less abortion and invalidation and hence less retries. Less retries lead to faster execution. In Actors, less contention means shorter message queues for the account actors and hence less waiting time for service. That leads to faster withdraw and deposit and therefore, faster transfers.

4.2.2 Producer-Consumer

The parameters that their variation for producer-consumer case is studied are production count and producer/consumer count.

In the experiments, the number of producers is equal to the number of consumers and all the producers/consumers perform equal number of productions/consumptions.

4.2.2.1 Total Time vs. Total Production Count

Figure 6 and Figure 7 depict the total time for various production counts respectively in machines M_1 and M_2 where the number of producers/consumers is equal to 20.

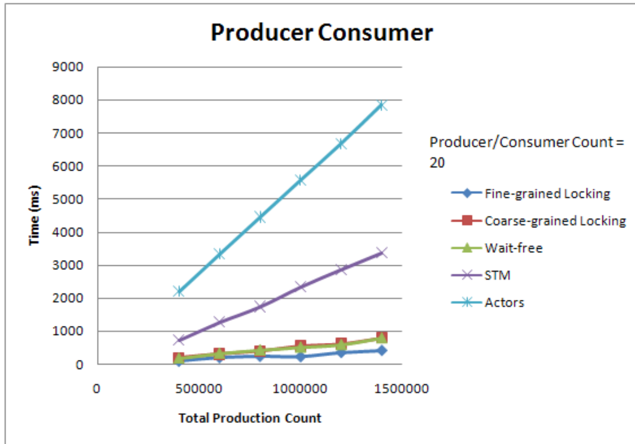

Figure 6. Total time vs. production count in M_1

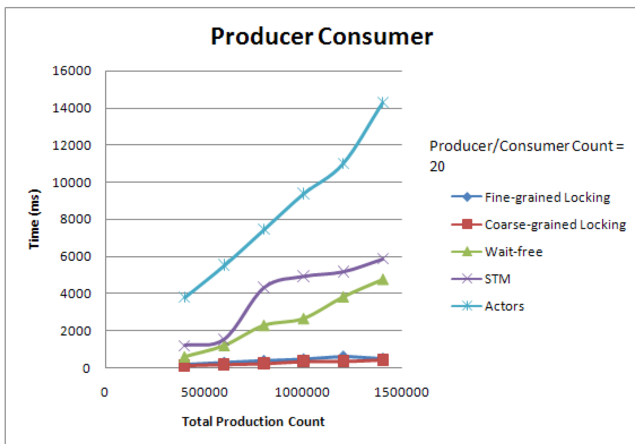

Figure 7: Total time vs. production count in M_2

4.2.2.2 Total Time vs. Producer/Consumer Count

In this experiment, the total number of productions/consumptions is constant (equal to 9,000,000). The varying parameter is the number of producers/consumers (and hence the number of productions per producer/consumer). Figure 8 shows the chart for machine M_1. This experiment shows how paradigms behave in different number of context switches.

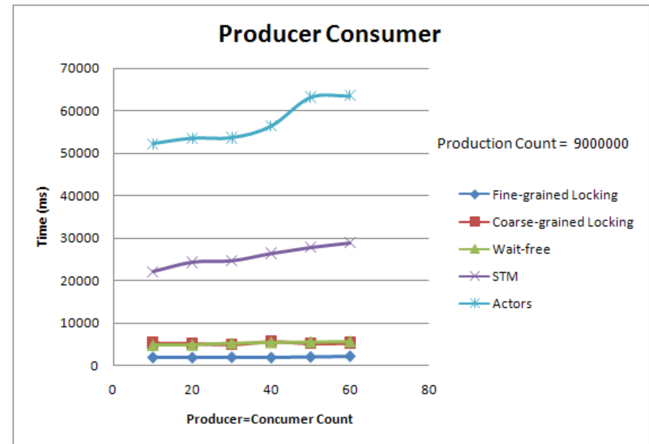

Figure 8: Total time vs. producer/consumer count in M_1

There is not much performance change in the coarse and fine-grained locking. In coarse-grained locking, all the operations are serialized by the intrinsic lock of the queue. The operations are always executed in sequence regardless of the number of concurrent operations on the queue. This argument also applied to fine-grained locking. The only difference is the fact that all enqueue and dequeue operations are respectively serialized on the rear and front locks. There is a noticeable performance decrease in STM. When there are more threads, there is more context switch. In STM, the more context switch, the more the probability of contention and abortion. All the messages sent to an actor are handled in sequence regardless of the number of concurrent requests; hence little change in performance is expected by increase in the producer/consumer count. The step in the actor's performance curve is because of thread creations by the actor scheduler. This happens when there are many blocked consumers.

Locks show the least sensitivity to context switch in this case.

4.2.3 Token Ring

4.2.3.1 Total time vs. total token passings

This experiment shows the total time spent for various number of total token passings in a ring with forty stations. Figure 9 and Figure 10 are respectively from machines M_1 and M_2.

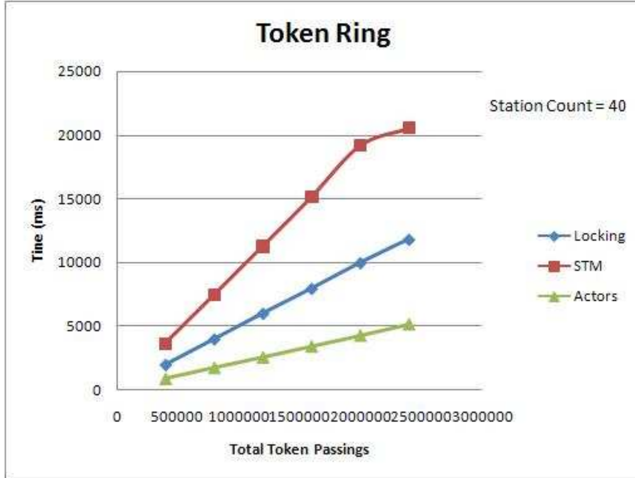

Figure 9: Total time vs. total token passing in M_1

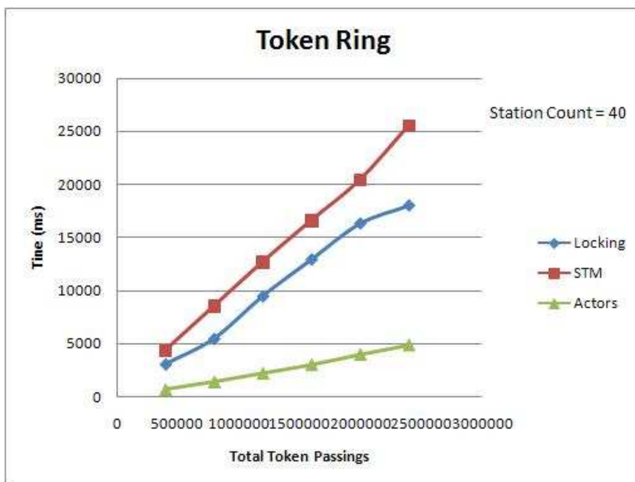

Figure 10: Total time vs. total token passing in M_2

Interestingly actors are even more efficient than locks and conditions in this case. This is because of efficient scheduling of actors. When an actor sends a message, the code of the receiving actor can be executed by the current thread. Hence much of the context switches are eliminated this way.

In this case, the transactions are never retried because of update to irrelevant objects. Transactional signaling is expected to have even less efficiency for transactions that read several objects before waiting on a condition.

4.2.3.2 Total Time vs. Station Count

In this experiment, the total number of token passings is constant (equal to 450000) and the number of station counts vary. Varying the number of active entities shows how each paradigm behaves against context switch. Figure 11 shows results from execution on M_1.

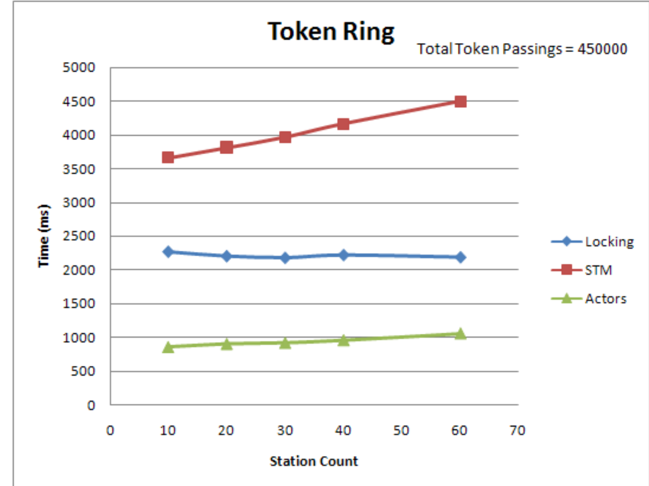

Figure 11: Total time vs. station count in M_1

Increase of context switch increases the abortion probability in STM. Hence, STM has the most performance sensitivity to context switch.

5. Conclusions and Future Work

Although locking is very efficient, it is hard to program fine-grained locking and more importantly it has some inherent shortcomings such as lack of compositionality, possibility of priority inversion and blocking event handlers. Non-blocking algorithms are also well at performance but developing such algorithms are hard enough to expect them only from experts. Non-blocking algorithms seem to be the best paradigm for thread safe libraries as they don't block and are also efficient. Hence STM or Actors are the choices for a general concurrent programming paradigm.

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. The choice of STM vs. Actors is a question of application and design.

The simulations show STM to be more efficient in providing isolation. This is while the operations that are experimented are short or medium-sized transactions and longer transactions should also be experimented. In addition, experiments show sensitivity of STM performance to context switch. The simulations show that Actors are very efficient in providing signaling. Interestingly they are even more efficient than primitive conditions.

Actor and STM have strength in different aspects. Actors support high level message passing while transactions support isolation well. If the problem is a data consistency problem then it is better to take advantage of efficiency of STM isolation. On the other hand, if it is a coordination problem then it can be implemented efficiently with Actors.

A future work is how to integrate the two approaches in a semantically well-defined and efficient way. It should be investigated how the isolation ideas from transactions and the message passing ideas from actors can be integrated in a semantically well defined and efficient way.

6. REFERENCES

[1] Michael, M. M. and Scott M. L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, Proc. 15th ACM Symp. on Principles of Distributed Computing

[2] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. 2003. Software transactional memory for dynamic-sized data structures. In Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (Boston, Massachusetts, July 13 - 16, 2003). PODC '03. ACM, New York, NY, 92-101. DOI= http://doi.acm.org/10.1145/872035.872048

[3] Herlihy, M., Luchangco, V., and Moir, M. 2006. A flexible framework for implementing software transactional memory. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 253-262. DOI= http://doi.acm.org/10.1145/1167473.1167495

[4] Dice, D., Shalev, O., and Shavit, N. Transactional Locking II, Lecture Notes in Computer Science

[5] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. 2005. Composable memory transactions. In Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Chicago, IL, USA, June 15 - 17, 2005). PPoPP '05. ACM, New York, NY, 48-60. DOI= http://doi.acm.org/10.1145/1065944.1065952

[6] Scala Actors: Unifying thread-based and event-based programming, Philipp Haller and Martin Odersky, Theoretical Computer Science, Volume 410, Issues 2-3, February 2009, Pages 202-220. (PDF, abstract, doi:10.1016/j.tcs.2008.09.019)

[7] Dragojevic, A., Guerraoui, R., Kapalka, M. Stretching Transactional Memory

[8] H. T. Kung, John T. Robinson. On optimistic methods for concurrency control, ACM Transactions on Database System