Shared-State Concurrency

Annalise Tarhan

July 17, 2020

1 Number of Interleavings

1.1 Generalize the argument used to calculate the number of possible interleavings of n threads, each doing k operations. Calculate a closed form approximation to this function.

For a function with n threads doing k operations each, the total number of operations will be $n * k$. We can't simply calculate $(nk)!$ because the operations in each thread must happen in order. If we consider all the operations in a single thread to be equivalent, we can impose the correct order at the end.

For a visual example, imagine ten red balls, ten blue balls, and ten yellow balls. If we calculate how many permuations there are of the thirty balls, it is important that two orders where the only difference is one red ball switched with another red ball are considered equivalent. For any given order, you can go down the line, marking the first of each colored ball 1, the second 2, until each ball is labeled and each color set is in order.

The formula, then, will be the number of permutations of a set of nk where there are n sets of k equivalent objects. By the formula for permutations with repetitions, this is $(nk)!/(n * k!)$. By Sterling's formula, this expands to $n^{nk-n-1} * k^{nk+1/2} * e^{n-nk}$

2 Concurrent Counter

2.1 Explain why the following counter, which should be callable from any number of threads, does not work. Propose a simple fix.

```
local X in {Exchange C X X+1}
```

There is a similar example in the chapter that does work, since the Exchange operation itself is atomic. The difference is that in the given example, the incremented variable is bound after Exchange completes. In this version, where X+1 occurs inside the Exchange operation, Exchange is no longer atomic. If a different thread interrupted at exactly the wrong time, it would result in two threads incrementing the same value of X, which would effectively cause one of them not to be counted. A simple fix would be using the same strategy as the other example: binding the variable to X+1 after Exchange completes.

```
local X1 X2 in {Exchange C X1 X2} X2=X1+1
```

2.2 Would this fix still be possible in a language without dataflow variables?

No. This solution depends on the fact that an unbound variable is put in the cell atomically and the calculation to increment the value takes place afterwards.

2.3 Give a solution that works in a language without dataflow variables.

As long as the language had a mechanism for locking a section of code, it would be simple to place a lock befor the exchange operation and unlock it afterwards. Otherwise, a lock could be implemented manually, with a boolean variable that is set to true during an increment operation and set back to false at the end. If a thread tries to increment while the lock is true, the operation would either fail or keep trying.

3 Maximal Concurrency and Efficiency

3.1 Investigate the job-based concurrent model, which is partway between the shared-state concurrent model and the maximally concurrent model. Is it a good choice for a concurrent programming language?

The problem with the job-based concurrent model is that it violates the principle of choosing the least concurrent model that suffices for the program. Except in rare cases, the message-passing and shared-state models are sufficient. As opposed to the maximally concurrent model, where every operation executes in its own thread, the job-based model can be implemented efficiently, but it is still difficult to reason about and program in. Therefore, the job-based model is not a good choice for most concurrent programming languages.

4 Simulating Slow Networks

4.1 Define a function based on SlowNet2 that creates slow objects that impose order among object calls within the same thread, but not object calls from different threads.

```
fun {SlowNet3 Obj D}
     D={NewDictionary}
in
     proc {$ M}
          Cell Old New
          Thread={Thread.this}
     in
          if {Dictionary.member D Thread $} then
               Cell={Dictionary.get Thread}
          else
               Cell={NewCell unit}
               {Dictionary.put D Thread Cell}
          end
          {Exchange Cell Old New}
          thread
               {Delay D} {Wait Old} {Obj M} New=unit
          end
     end
end
```

5 The MVar Abstraction

5.1 Implement the MVar abstraction, a box that can be full or empty, with two procedures: Put and Get.

```
class MVar
    attr box m
    meth init()
        box:=nil
        m:={NewMonitor}
    end
    meth put(X)
        {@m.'lock'
            proc {$}
                if @box\=nil
                then {@m.wait} {self put(X)}
                else box:=X {@m.notifyAll}
                end
            end
        }
    end
    meth get(X)
        {@m.'lock'
            proc {$}
                if @box==nil
                then {@m.wait} {self get(X)}
                else X=@box box:=nil {@m.notifyAll}
                end
            end
        }
    end
end
```

6 Communicating Sequential Processes

6.1 Implement a CSP style Channel with send and receive functions.

```
local SendLock ReceiveLock Unbound1 Unbound2
     {NewLock SendLock}
     {NewLock ReceiveLock}
     Send={NewCell Unbound1}
     Receive={NewCell Unbound2}

     fun {NewChannel}
          Send#Receive#SendLock#ReceiveLock
     end

     proc {Send C M}
          case C of S#R#Lock#_
          then
               lock Lock then
                    S:=M
                    {Wait @R}
               end
          end
     end

     proc {Receive ChannelList Flag}
          case ChannelList
          of nil then skip
          [] Channel#Procedure|Tail then
               thread
                    {Receive Tail Flag}
               end
               case Channel of S#R#_#Lock then
                    lock Lock then
                         Unbound1 Unbound2
                    in
                         {WaitTwo @S Flag}
                         if {IsDet @S} then
                              Flag=unit
                              {Procedure @S}
                              @R=unit
                              S:=Unbound1
                              R:=Unbound2
                         else skip end
                    end
```

```
                end
            end
        end

        proc {MReceive ChannelList}
            Flag in {Receive ChannelList Flag}
        end
in
        Channel=channel(new:NewChannel send:Send
                        mreceive:MReceive)
end
```

6.2 Extend the Channel.mreceive operation with guards.

```
proc {Receive ChannelList Flag}
        case ChannelList
        of nil then skip
        [] Channel#Guard#Procedure|Tail then
            thread
                {Receive Tail Flag}
            end
            case Channel of S#R#_#Lock then
                lock Lock then
                    Unbound1 Unbound2
                in
                    {WaitTwo @S Flag}
                    if {IsDet @S} then
                        if {Guard @S} then
                            Flag=unit
                            {Procedure @S}
                            @R=unit
                            S:=Unbound1
                            R:=Unbound2
                        else
                            S:=Unbound1
                            {Receive
                            Channel#Guard#Procedure|nil
                            Flag}
                        end
                    else skip end
                end
            end
        end
end
```

7 Comparing Linda with Erlang

7.1 Compare and contrast Linda's read operation and Erlang's receive operation and the abstractions that they are part of. For what kinds of application is each best suited?

Both operations sift through data in the order the tuples were written or messages were received. They both have blocking and non-blocking options, Linda's explicit and Erlang's based on the timout. An infinite timeout is blocking and a timeout of zero is non-blocking. Unlike Linda, Erlang allows for intermediate values, timing out after a certain point. Linda groups tuples based on their labels, Erlang keeps a flat list of messages.

The most significant difference between Linda's read and Erlang's receive operations is that Linda's read can occur anywhere in a program and will have access to the same tuple space. Erlang's receive is tied to a particular mailbox. In that sense, Linda's tuples are global and Erlang's messages are local.

Linda fits the shared-state model, where a database (tuple-space) is accessed and updated concurrently, and Erlang fits the message-passing model, which consists of autonomous entities that communicate with each other. Linda's read is best suited for shared-state applications that use a database, and Erlang's receive is better for applications based on message passing.

8 Termination Detection with Monitors

8.1 Write an algorithm that works for detecting when a group of threads has terminated using a monitor.

```
proc {NewThread P ?SubThread}
     Tracker={NewCell 0}
     Monitor={NewMonitor}
     Finished
     proc {Increment}
          A B in
          {Exchange Tracker A B}
          B=A+1
     end
     proc {Decrement}
          A B in
          {Exchange Tracker A B}
          B=A-1
          if B==0 then Finished=unit end
     end
in
     proc {SubThread P}
          {Monitor.'lock'
               proc {$}
                    {Increment}
                    thread
                         {P}
                         {Decrement}
                    end
               end
          }
     end
     {Wait Finished}
end
```

9 Monitors and Conditions

9.1 Reimplement the bounded buffer example using monitors with conditions.

```
declare
class Buffer
     attr m ...
     meth init(N)
          m:={NewMonitor [nonempty nonfull]}
          ...
     end
     meth put(X)
          {@m.'lock'
          proc {$}
               if @i>=@n then {@m.wait nonfull}
               else
               ...
               {@m.notify nonempty}
               end
          end}
     end
     meth get(X)
          {@m.'lock'
          proc {$}
               if @i==0 then {@m.wait nonempty}
               else
               ...
               {@m.notify nonfull}
               end
          end}
     end
end
end
```

9.2 Modify the monitor implementation to implement monitors with conditions.

```
fun {NewMonitor Conditions}
     Queues={NewDictionary}
     DefaultQueue={NewQueue}
     L={NewGRLock}

     for Condition in Conditions do
          Queues.Condition:={NewQueue}
     end

     proc {WaitM Condition}
          X in
          if {Dictionary.member Queues Condition $}
          then {@(Queues.Condition).insert X}
          else {DefaultQueue.insert X}
          end
          {L.release} {Wait X} {L.get}
     end

     proc {NotifyM Condition}
          U in
          if {Dictionary.member Queues Condition $}
          then U={@(Queues.Condition).deleteNonBlock}
          else U={DefaultQueue.deleteNonBlock}
          end
          case U of [X] then X=unit else skip end
     end
     ...
end
```

10 Breaking Up Big Transactions

10.1 Rewrite the Sum example as a series of small transactions that only lock a few cells. Define a representation for a partial sum, so that a small transaction can see what has already been done and determine how to continue. Verify that you can perform transactions while a sum calculation is in progress.

```
Sum={NewCellT 0}
Next={NewCellT 1}

proc {SetUp}
     {Trans
          proc {$ T _}
               {T.assign Sum 0}
          end _ _}
end
proc {DoChunk}
     {Trans
          proc {$ T _}
          N={T.access Next}
     in
          {T.assign Next N+5}
          for I in N..N+4 do
               {T.assign Sum
                    {T.access Sum}+{T.access D.I}} end
     end _ _}
end
fun {Finish}
     {Trans
          fun {$ T}
               {T.access Sum}
     end _}
end
fun {SmallSum}
     {SetUp}
     {Trans
          proc {$ T _}
               for _ in 1..20 do {DoChunk} end
     end _ _}
     {Finish}
end
```

```
fun {Checker I}
     {Trans
          fun {$ T}
               {T.access D.I}
     end _}
end
thread
     {Browse {SmallSum}}
end
thread
     for I in 1..50 do
          {Browse {Checker I}}
     end
end
```

11 Lock Caching

11.1 Optimize the getlock and savestate protocols of the transaction manager so they use the least possible number of messages.

```
meth Trans(P ?R TS)
     Cells={NewCell nil}
     ...
     fun {HasCell C List}
          case List
          of nil then false
          [] H|T andthen H==C then true
          [] H|T then {HasCell C T}
          end
     end
     proc {ExcT C X Y}
          if {Not {HasCell C.name @Cells}}
          then S1 S2 in
               {@tm getlock(T C S1)}
               if S1==halt then raise Halt end end
               {@tm savestate(T C S2)} {Wait S2}
               Cells:=Cell.name|@Cells
          end
          {Exchange C.state X Y}
     end
     ...
end
```

12 Read and Write Locks

12.1 Extend the transaction manager to use read and write locks.

```
class TMClass
      ...
      meth Unlockall(T RestoreFlag)
            for save(cell:C state:S) in
            {Dictionary.items T.save} do
                  if @(C.writer)==T
                  then
                        (C.writer):=unit
                        if {Not {C.writequeue.isEmpty}} then
                              Sync2#T2={C.writequeue.dequeue}
                        in
                              (T2.state):=running
                              (C.writer):=T2
                              Sync2=ok
                        elseif {Not {C.readqueue.isEmpty}}
                              then
                              {While
                              {Not {C.readqueue.isEmpty}}
                              proc {$}
                                    SyncX#TX=
                                    {C.readqueue.dequeue}
                              in
                                    (TX.state):=running
                                    (C.readers):=TX|@(C.readers)
                                    SyncX=ok
                              end}
                        end
                  else
                        (C.readers):=
                              {RemoveFromList @(C.readers) T}
                        if {@(C.readers)==nil} then
                              if {Not {C.writequeue.isEmpty}}
                                    then
                                    Sync2#T1=
                                    {C.writequeue.dequeue}
                              in
                                    (T2.state):=running
                                    (C.writer:=T2
                                    Sync2=ok
                              end
```

```
                end
            end
        end
end
meth Trans(P ?R TS)
        Halt={NewName}
        T=trans(stamp:TS save:{NewDictionary} body:P
        state:{NewCell running} result:R)
        proc {ExcT C X Y} S1 S2 in
            {@tm getwritelock(T C S1)}
            if S1==halt then raise Halt end end
            {@tm savestate(T C S2)} {Wait S2}
            {Exchange C.state X Y}
        end
        proc {AccT C ?X} S in
            {@tm getreadlock(T C S)}
            if S==halt then raise Halt end end
        end
        proc {AssT C X} S1 S2 Y in
            {Exct C _ X}
        end
        proc {AboT}
            {@tm abort(T)} R=abort raise Halt end
        end
in
        ...
end
meth getwritelock(T C ?Sync)
        if @(T.state)==probation then
            {self Unlockall(T true)}
            {self Trans(T.body T.result T.stamp)}
            Sync=halt
        elseif @(C.writer)==unit
            andthen @(C.readers)==nil then
            (C.writer):=T
            Sync=ok
        elseif @(C.writer)==unit
            andthen {Size @(C.readers)}==1
            andthen @(C.readers).1==T then
            (C.readers):=nil
            (C.writer):=T
            Sync=ok
        elseif T.stamp==@(C.writer).stamp then
            Sync=ok
        elseif {Not @(C.writer)==unit}
            T2=@(C.writer) in
            {C.writequeue.enqueue Sync#T T.stamp}
            (T.state):=waiting_on_write(C)
            if T.stamp<T2.stamp then
                case @(T2.state)
                of waiting_on_write(C2) then
                    Sync2#_=
                    {C2.writequeue.delete T2.stamp}
                in
                    {self Unlockall(T2 true)}
                    {self Trans(
                        T2.body T2.result T2.stamp)}
                    Sync2=halt
                [] waiting_on_read(C2) then
                    Sync2#_=
                    {C2.readqueue.delete T2.stamp}
                in
                    {self Unlockall(T2 true)}
                    {self Trans(
                        T2.body T2.result T2.stamp)}
                    Sync2=halt
                [] running then
                    (T2.state):=probation
                [] probation then skip
                end
            end
        end
end
meth getreadlock(T C ?Sync)
    if @(T.state)==probation then
        {self Unlockall(T true)}
        {self Trans(T.body T.result T.stamp)}
        Sync=halt
    elseif @(C.writer)==unit then
        (C.readers):=T|@(C.readers)
        Sync=ok
    elseif {Contains @(C.readers) T}
        Sync=ok
        else T2=@(C.writer) in
        {C.readqueue.enqueue Sync#T T.stamp}
        (T.state):=waiting_on_read(C)
        if T.stamp<T2.stamp then
            case @(T2.state)
            of waiting_on_write(C2) then
                {self Unlockall(T2 true)}
                {self Trans(
                    T2.body T2.result T2.stamp)}
```

```
                              Sync2=halt
                        [] waiting_on_read(C2) then
                              {self Unlockall(T2 true)}
                              {self Trans(
                                    T2.body T2.result T2.stamp)}
                              Sync2=halt
                        [] running then
                              (T2.state):=probation
                        [] probation then skip
                        end
                  end
            end
      end
      meth newtrans(P ?R)
            timestamp:=@timestamp+1
            {self Trans(P R @timestamp)}
      end
      meth savestate(T C ?Sync)
            if {Not {Dictionary.member T.save C.name}} then
            (T.save).(C.name):=save(cell:C state:@(C.state))
            end Sync=ok
      end
      meth commit(T) {self Unlockall(T false)} end
      meth abort(T) {self Unlockall(T true)} end
      proc {NewTrans ?Trans ?NewCellT}
            TM={NewActive TMClass init(TM)} in
            fun {Trans P ?B} R in
                  {TM newtrans(P R)}
                  case R of abort then B=abort unit
                  [] abort(Exc) then B=abort raise Exc end
                  [] commit(Res) then B=commit Res end
                  end
            end
            fun {NewCellT X}
                  cell(name:{NewName} writer:{NewCell unit}
                        readers:{NewCell nil}
                        writequeue:{NewPrioQueue}
                        readqueue:{NewPrioQueue}
                        state:{NewCell X})
            end
      end
end
```

13 Concurrent Transactions

13.1 Extend the transaction manager so that the individual transactions can be concurrent.

The thread abstraction from an earlier chapter that uses a port to receive information about subthread launches and terminations can be used to allow transactions to be concurrent. Subthreads within the transactions need to be launched using the unbound SubThread variable and passed to {Trans P ?B ?SubThread} in order for their launch and termination to be detected.

```
meth Trans(P ?R TS ?SubThread)
     ...
in
     {NewThread
          proc {$}
               try
                    Res={T.body t(access:AccT assign:AssT
                                   exchange:ExcT abort:AboT)}
               in
                    {@tm commit(T)}
                    R=commit(Res)
               catch E then
                    {@tm abort(T)}
                    R=abort(E)
               end
          end
          SubThread
     }
end
```

14 Combining Monitors and Transactions

14.1 Consider a concurrency abstraction that combines the abilities of monitors and transactions with the ability to wait and notify and also the ability to abort without changing any state. Is this a useful abstraction?

It is not. The monitor part of the abstraction would simply be a dumber version of the transaction manager's priority queue. Without timestamps, it would risk deadlock and thread starvation. With timestamps, it would be the same as the transaction manager.