**Background**

1. **Model of a distributed computation**

Our model of a distributed computation is identical to that in [1]. It is based on message passing, without any assumptions on the upper bound of message delays. A distributed program consists of processes denoted by , and a set of unidirectional channels, each of which connects two processes together. Note that a bidirectional channel can be modeled as two unidirectional channels. The channels are assumed to have reliable delivery, and an arbitrary but finite delay. Each process is defined as a sequence of events that transition the process from one state to another. Examples of events include the beginning of the execution of a function, the end of its execution, the sending of a message, and the reception of a message [1]. We also assume that events on the same process execute sequentially, that is, each process is single-threaded.

As argued in [2], a true distributed system can only have a partial order, dictated by the messages exchanged in the system. This order is called the happened before relation.

In the simplest case, if we consider two events occurring on the same process, then the one that occurs earlier is said to happen before the other. Note that this is well-defined since we assume processes are single-threaded. If we now consider two events occurring on different processes, where one event is the send event of a message, and the other event is the receive event of the same message, then the first naturally happen before the second. The definition of happens before is transitively extended based on these two base cases. More formally, let denote that is the event directly preceding on the same process, and let denote that is the send event of a message and is the receive event of a message, then the happened before relation, denoted by , can be defined as:

As discussed in [1], happens before can only ensure a partial order on the set of events in a computation. For two distinct events and , it may neither be true that nor that . In this case, the events are said to be concurrent, and denoted by .

[1] vijay

[2] lamport

1. **Dependency model**

**Algorithm**

Our algorithm takes a distributed computation as input, annotated with the variables that are being accessed at each event. Each event is annotated with a list of the variables that have been read right before this event, and a list of variables that have been written right before this event. By right before, we mean that the event has been read (resp. written) after the immediately preceding event and before the event in question (for the initial event, we assume that the variable has been read or written after the start of the execution and before the event).

The output of our algorithm is a list of computations, each of which identical to the input computation, but with some added synchronization messages to satisfy a certain ordering between the events that access the same variables.

To illustrate our algorithm, let us first consider the very basic example below, where we have only two processes and one event on each process. Suppose both of these events write on the same shared variable, say . Note that both of these events are concurrent, meaning that in some executions might execute earlier than , while in others, might execute earlier. In either of these two cases, there will be an output dependence, since the latter event will be writing to the same variable than the earlier event wrote to.

If the developer is interested in exploring output dependencies, then our output should be a list of computations that explore all the possible ordering. For this example, since there are only two possible orderings, we will output two computations, one of which has an added synchronization message from to , and the other has an added synchronization message from to .

The above describes a simplified version of the base case of our algorithm. Below we describe the full algorithm.

generateSynchronizedComputations(computation):

concurrentEvents = getConcurrentEvents(computation)

result = { computation }

for each event1, eventsConcurrentTo1:

generated = synchronize(event1, eventsConcurrentTo1, index)

result = mergeLists(result, generated)

synchronize(computation, event1, eventsConcurrentTo1, index):

if index = size(eventsConcurrentTo1):

return { computation }

event2 = eventsConcurrentTo1[index]

generated = {}

message1Added = false

message2Added = false

if checkDependenciesSatisfied(event1 -> event2):

newComputation, messageAdded = addMessage(computation, event1 -> event2)

if messageAdded:

generated = generated U synchronize(newComputation, event1, eventsConcurrentTo1, index + 1)

if checkDependenciesSatisfied(event2 -> event1):

newComputation, messageAdded = addMessage(computation, event2 -> event1)

if messageAdded:

generated = generated U synchronize(newComputation, event1, eventsConcurrentTo1, index + 1)

if not(message1Added) and not(message2Added):

generated = generated U synchronize(computation, event1, eventsConcurrentTo1, index + 1)

* *generateConcurrentEvents(computation)*

The generateConcurrentEvents function returns a mapping between each event and the list of all events it is concurrent with. For performance reasons, if was included in the set of concurrent events for , we don’t include in the set of concurrent events for , so as to not process the same pair of concurrent events twice.

To compute the list of events concurrent to an event , we first find all the events that are reachable from , subtract it from the set of all events in the computation, and then filter out all the events from which is reachable. We obviously also need to remove itself, as is not considered concurrent with itself.­­ Following this procedure, we will be left with all the events such that ,, and , which is the definition of all events concurrent with . In this part of the algorithm, we use dynamic programming, in order to find the set of events reachable from a given event.

* *addMessage(computation, e1 -> e2)*

The addMessage function creates a copy of the computation, possibly with the added message. As will be explained in the following paragraph, it is not always possible to add a message. This function returns that modified copy and a Boolean indicating whether the message was added successfully or not. Note that in terms of the pseudo-code above, addMessage does not modify the computation passed to it.

A message will not be added if it generates a cycle in the computation, as this would violate the asymmetric nature of the happens before relation. Note that even though we are only looking at pairs of concurrent events in terms of the original computation, and an added message between two concurrent events should not result in a cycle, the function addMessage might not be acting on the original computation. For example, if and are concurrent, and if we have already added a message from to into a copy of the computation, then in the recursive call, a message from to could not be added to this copy, as this would create a cycle in the copy of the computation (even though a message form to could be added to the original computation).

* *checkDependenciesSatisfied(e1 -> e2)*

This function checks whether a synchronization message from to is actually needed, based on what variables are being read an written in each of these events. For example, if reads , and write on , and if the developer is only interested in flow dependencies, then we do not have to explore the order of events where happens before , since this order does not result in a flow dependence. However, if the developer is also interested in anti-dependencies, then a message from to should be added, as the resulting order would cause to read from before writes to it.

* *mergeLists(list1, list2)*

This function takes two lists of computations, and generates a list where each computation in *list1* is merged with every computation in *list2.* In other words, the size of the resulting list will be up to . Note that the size might be actually smaller because some computations could be impossible to merge (if the merging would result in a cycle). The pseudocode for this function is

result = {}

for each comp1 in list1

for each comp2 in list2

mergedComp = merge(comp1, comp2)

if mergedComp is not NULL

result = result U { mergeComp }

We finally describe the function that merges two computations mentioned above. In this function, the two computations are exactly the same in terms of processes, events, and ordinary messages, but they might differ in the synchronization messages that they have. One might initially consider simply merging the synchronization messages of each of these two computations, and ignoring the ones that could not be added due to a cycle. While this is correct, the resulting computation created will often have redundant synchronization messages that are undesirable. Consider the two computation below, where the red arrows denote the synchronization messages, and the black arrows denote ordinary messages.

If we were to follow the simple approach of merging the message, we would obtain the following computation.

While this is valid, the synchronization message from ….. to …. is not needed because the relation is already satisfied from the other two messages.

The following merge algorithm avoids this problem:

merge(computation1, computation2)

result = copy all events and ordinary messages from either computation1 or computaiton2

for each synchronized message from to in computation1:

if not (isReachable(computation2, , )):

add synch message from to to result

for each synchronized message from to in computation2:

if not (isReachable(computation2, , )) or computation2.containsMessage(,:

add synch message from to to result

This approach will add all the synchronization messages from computation1 into result as long as the corresponding happens before relation is not already satisfied in computaiton2, and then do the same thing for the synchronization messages in comptation2. Note the additional call to computation2.containsMessage(,, which is needed in order to not skip adding a message if it is present in both computaiton1 and computation2.