Distributed ORC

Adrian Quark
quark@mail.utexas.edu

May 6, 2008

Abstract

This report describes a distributed implementation of an interpreter for the computation orchestration language ORC. While ORC includes primitives for distributed computation in the form of "sites", these have limitations and it is difficult to convert a non-distributed ORC program into a distributed one. I therefore introduce a new syntax for annotating distributable expressions in an existing ORC program, and modify the existing ORC interpreter to transparently distribute such expressions. The result is a language which makes it very easy to write correct distributed programs for some problems.

1 Introduction

Distributed computing is the cooperation of multiple physically-separated computers solving a single problem. There are many reasons this might be desirable:

Processing Scalability Solving bigger problems faster using more computing resources.

Data Scalability Combining information from multiple large data sources without requiring the infrastructure for storing and processing such data to be duplicated.

Fault Tolerance A distributed system may be designed to tolerate failure of some of the participating computers.

Security Physically separating processes allows communication between them to be restricted to protect against unwanted access to information or computing resources.

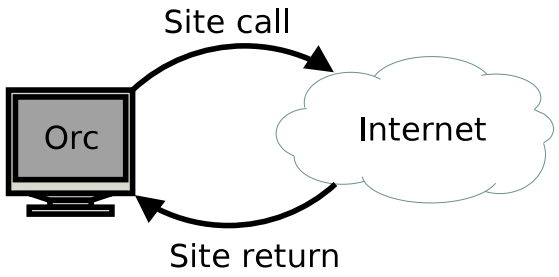

Figure 1: ORC calling a site

Geographical Localization Some problems require information acquired from specific geographic locations.

With these advantages come some unique problems.

Latency The speed of light and limitations of our infrastructure place hard limits on the speed of communication between physically separated computers.

Unreliability Computers and communication links may go offline for a variety of reasons. Increasing the number of computers and communication links in a system increases the likelihood of such an occurrence.

Correct Concurrency Distributed systems are naturally concurrent, and writing concurrent programs which are free of deadlock, starvation, etc. remains challenging.

ORC is a language specifically designed to implement large-scale distributed programs while coping with these

problems [3]. It is essentially a scripting language for distributed systems. Like a traditional scripting language, it does very little computation itself; rather its role is to orchestrate the flow of data between several processes (programs, computers, systems, networks). In ORC, such processes are called "sites", and exchanging information with a site is similar to a function call in any other language. ORC makes no assumptions about how sites are implemented or where they are located. This neutrality is important because it allows ORC to coexist and interact with a variety of existing systems, and to orchestrate distributed computations just as easily as non-distributed ones. Figure 1 illustrates this concept.

However, a key limitation of ORC is that it there is no mechanism to define sites within ORC. It can only invoke externally-defined sites, and all distributed communication must occur via site calls. This causes problems in two situations:

- In order to implement a trivial distribution task (for example, "open a yes/no dialog box on another computer and get the user's response"), the programmer must implement a site and a communication protocol to connect to it, which may be a lot of work for such a simple task.

- A large program written in ORC cannot be easily broken into parts and run in a distributed manner. Again, it would be necessary for the programmer to implement sites to run on each distributed system and explicitly handle all the communication between ORC processes.

The goal of this project is to solve these problems by introducing a new syntax for distributed expressions. The programmer simply annotates an existing ORC program to indicate where (on which computer in the distributed system) each sub-expressions should be run, and the ORC interpreter transparently handles all distributed communication.

The remainder of this paper is organized as follows. Section 2 summarizes the concepts, syntax, and semantics of the ORC language. Section 3 describes my extensions to the language to enable easier distributed computing. Section 4 reviews the current implementation of the non-distributed ORC interpreter. Section 5 explores the distributed implementation. Section 6 summarizes the results and discusses future work.

2 The ORC Language

This section provides a high-level overview of the ORC language. Readers familiar with the language may wish to skip to the next section.

ORC is based on a simple computational model. ORC programs are represented by expressions, which "publish" (evaluate to) values. Unlike a traditional sequential programming language, each expression may publish more than one value, if it represents a concurrent computation with more than one thread. Expressions are built from two core primitives: site calls and combinators. Site calls represent communication with some service, while combinators are used to sequence and direct this communication.

Site calls are analogous to function calls in any other language, with one key difference: a site call may take an indefinite amount of time to return a value, or may never return. For this reason site calls are perfect to represent distributed communication, which may have arbitrary latency and be unreliable.

There are three combinators which are used to combine site calls into a larger expressions. The "parallel" combinator runs two expressions in parallel. The "sequential" combinator takes each value published from the left sub-expression and uses it to evaluate the right sub-expression. The "asymmetric" combinator allows one-way communication between two parallel expressions: the left sub-expression may wait on a single value to be produced by the right sub-expression.

In addition to expressions built from combinators and site calls, ORC allows the programmer to define and call functions, which are similar to those in most other languages.

Table 1 summarizes a simplified core syntax of ORC expressions, where `f` and `g` stand for arbitrary expressions, `x` and `y` represents arbitrary variable names, and `M` and `E` represent site and function names respectively. Operators are listed in order of increasing precedence. Sites and expressions are shown here taking two arguments but in fact may take any number. As a shorthand, it is often useful to use nested expressions as arguments to expressions and functions in place of variable names. The

`def E(x,y) = f`	function definition
`M(x,y)`	site call
`E(x,y)`	function call
`f >x> g`	sequential combination
`f | g`	parallel combination
`f <x< g`	asymmetric combination

Table 1: ORC syntax

notation `M(f)`, where `f` is an expression, should be understood as shorthand for `M(x) <x< f`.

For a full treatment of the ORC syntax, see [2].

The meaning of each of the syntactic elements, informally, is:

`def E(x) = f` The function `E` is defined with the body `f` and formal parameter `x`. Functions are lexically scoped, so that any free variables in `f` refer to those in scope where the function is defined.

`M(x,y)` When the values of `x` and `y` become available, send those values to the site `M` and wait for a response. When and if `M` responds, publish its response.

`E(x,y)` The body of `E` is substituted for the expression, with `x` and `y` substituted for the formal parameters of `E` within the body. Then the body is evaluated. Unlike a site call, a function call does not have to wait for the value of its argument to become available before it can be evaluated and begin publishing values.

`f | g` The expressions `f` and `g` are evaluated in parallel, and any expression published by either is published by the overall expression. There is no direct communication between `f` and `g`.

`f >x> g` The expression `f` is evaluated. Whenever it publishes a value, the expression `g` is evaluated with the published value substituted for `x`. In essence, each value published by `f` triggers a new thread evaluating `g`.

`f <x< g` Expressions `f` and `g` are evaluated in parallel, with `x` bound in `f`. As soon as `g` publishes a value, it is made available as the value of `x` in `f`, and `g` is immediately terminated.

`let(x)`	the identity site: return the value of `x`
`if(x)`	return a value if and only if `x` is true
`Rtimer(x)`	return a value after `x` time units elapse

Table 2: Fundamental ORC sites

It is impractical to give a full formal treatment of ORC's semantics in this context. The interested reader should refer to [2].

Because `Orc` relies entirely on sites for computation, it requires a few fundamental sites in order to do anything useful. These are defined in Table 2. In addition, it is convenient to assume that sites have been defined corresponding to arithmetic operators and values, and these may be used with standard infix syntax, so that `1 + 2` means that the site `+` is called with values `1` and `2` and returns their sum, `3`.

3 DORC: Distributed ORC

As mentioned in the introduction, the primary goal of the distributed implementation of ORC (DORC) is to provide programmers with a middle ground between non-distributed ORC expressions and distributed ORC sites. Programmers should be able to gain some of the benefits of distributed computation without using sites to implement it.

Why is this functionality desirable? ORC already provides sites as a primitive for distributed communication, and any conceivable form of communication can be handled with sites. For example, even though sites cannot directly publish multiple values, they can be called repeatedly to request a sequence of values. It is even possible to implement distributed program logic in ORC by using special-purpose "shared" sites (like channels) which allow values to be passed between multiple ORC interpreters. However any solution involving sites places some burden on the programmer to structure their program so that all distributed communication is mediated by site calls. In some cases, especially when dealing with a large existing ORC program, this requirement may be impractical.

The alternative offered by DORC is to allow users to annotate ORC expressions to indicate that they should be

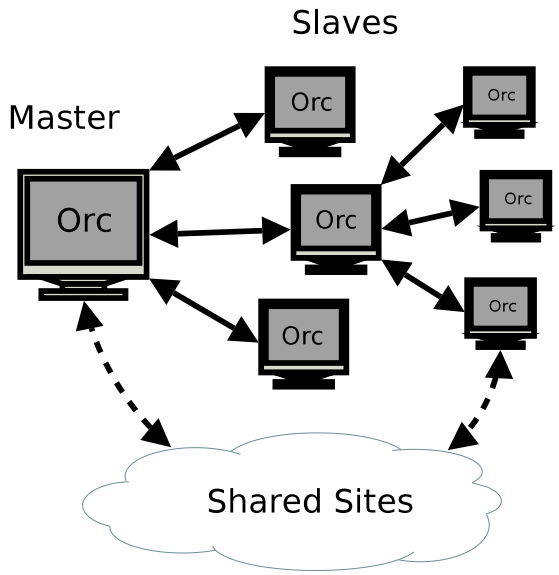

Figure 2: DORC architecture

run remotely. Such expressions use remote computing resources, as does a site call, but otherwise have all the properties of regular ORC expressions, including:

- The same ORC syntax
- The ability to call functions and refer to variables defined in the surrounding ORC program
- The ability to publish multiple values
- The ability to participate in asymmetric composition

DORC introduces one new type of value and one new syntactic construct to the language. The new value type is a "server", which corresponds to a computer in a distributed computation[1]. The new syntactic construct is the "remote expression", which specifies that an expression should be evaluated on a specific server.

Every server participates in one and only one distributed computation, and all servers in the computation share the same environment (variables and function definitions). In order to enforce this restriction, it is impossible to create a new reference to an existing server. Servers can only be created in the context of a specific distributed computation, passed around, and used in the context of that computation. In the context of evaluating a specific expression across two servers, one server (which initiated the computation) is the master and the other server (which is performing the evaluation) is the slave. However in the context of the program as a whole, each server may be participating in the evaluation of several expressions, and therefore there may be no clear master-slave structure to the overall computation. Figure 2 shows the overall architecture; compare this with Figure 1.

To create a new server, a DORC program calls a meta-server, which is simply an ordinary ORC site which returns servers. Each meta-server has a known global name, so that any DORC process may request a new server from it. Typically a meta-server corresponds to a specific physical computer and produces servers which evaluate expressions on that computer, but it is entirely possible to implement a meta-server which acts on behalf of a pool of physical computers, returning servers which may evaluate expressions on any member of the pool.

The added DORC syntax is very simple: `f @ x` remote expression where `f` is an arbitrary expression and `x` is some expression publishing a server value. The `@` operator has higher precedence than any other operator.

The meaning of this expression is: when the value of `x` is available, evaluate the expression `f` on the server specified by the value of `x`. The precise semantics of this expression are discussed in the next section, but for now this intuitive description should suffice.

3.1 Examples

Let us consider some examples of distributed programs. These are not very interesting because they are all equivalent to similar non-distributed programs, but they serve to illustrate the variety of communication which may occur between distributed servers. In the following examples, I will assume the existence of sites `c.put` and `c.get`, which put and get to an asynchronous buffer. If the buffer is empty, `c.get` waits to return until a new item is placed in the buffer by `c.put`. I will also assume that a remote

[1] In the literature this is usually called a "node", but I have chosen "server" to avoid confusion with the term "node" used in the context of the ORC implementation.

server has been created and is available in the variable `r`.

`1 @ r` is the simplest remote expression. It evaluates the constant `1` at the remote server `r` and finally publishes the value `1` back to the local server.

`(1 + 2)@r` is a slightly more complex expression. It actually carries out some computation on `r`, evaluating `1+2` and publishing the result.

`(Rtimer(1) | Rtimer(2))@r` will start two `Rtimers` on `r` and publish values after 1 and 2 time units. This example illustrates that, unlike a site call, a remote expression may publish multiple values.

`Rtimer(1)@r | Rtimer(2)@r` gives the exact same result as `(Rtimer(1) | Rtimer(2))@r`, modulo timing concerns discussed in the next section. This example illustrates that it is entirely possible to initiate multiple computations on a server at the same time.

`c.get() | c.put(1)@r` places a value on the buffer at the remote server, and retrieves it locally. Distributed expressions can communicate via sites just like local expressions.

`(c.put(1) >> let(x))@r <x< Rtimer(1)` evaluates the `Rtimer` locally in parallel with the `c.put(1)` remotely. When the remote server reaches the `let(x)`, it must wait for the local server to publish a value to `x` before it can proceed.

`let(x) <x< Rtimer(1) | Rtimer(2)@r` evaluates one `Rtimer` locally and one remotely. The local `Rtimer` publishes a value first, and when it does, all further computation of the parallel remote expression is terminated and no value will be published from it.

3.2 Semantics

The semantics of a remote expression are closely related to the timing semantics of ORC. Traditionally, ORC sites are classified into immediate sites, whose values are published at precisely-defined times or not at all, and non-immediate sites, whose values may be published after arbitrary delay. `let` and `Rtimer` are examples of immediate sites. The fact that these sites are immediate means that the expression `Rtimer(1) >> 1 | Rtimer(2) >> 2` is guaranteed to publish the values 1 and 2 in that order.

This requirement is problematic for a distributed implementation, because starting a distributed expression may involve arbitrary delay.

A simple, but heavy-handed, solution would be to discard the concept of immediate sites. If all sites are allowed to wait an arbitrary amount of time before publishing a value, any delay introduced by remote communication may be attributed to the delay in some site returning a value. Unfortunately, this means that it is no longer possible for a program to rely on the order that values are published by any expression. Whether this is a problem for typical ORC programs remains a subject for future study.

Fortunately a slightly more refined solution may be possible. The delay introduced by distributed communication affects the semantics only if it is observable. Therefore, it suffices to ensure that the chain of causal relationships connecting any local event to any locally observable result of evaluation on a remote server (the remote server publishing a value or calling a stateful site) includes some non-immediate site to which the delay in distributed communication can be attributed.

I believe, but have not proven, that the expression `f@r` is exactly equivalent to `LET() >> f >x> LET(x)`, where `LET` is a non-immediate form of `let`, provided that all stateful sites (such as buffers) are also considered non-immediate. The reasoning behind this is as follows: the local node can only communicate with the remote node via the initiation of the expression, through a stateful site, or through a future. The delay in the initiation of the expression is accounted for by the non-immediate site call `LET()`. Communication through stateful sites is also subject to the non-immediacy of these sites. The publication of a where value cannot by itself convey any timing information, and so can only be given a precise time relative to some event observed via another means.

A good example to illustrate the problems with a distributed semantics is:

```
( let(a) <a< let(x) | let(y) )@r
    <x< Rtimer(1) >> 1
    <y< Rtimer(2) >> 2
```

This example shows that even with the `LET`-based semantics described in the previous paragraph, the distributed communication mechanism must guarantee in-order delivery to ensure that the correct value is published by `let(a)`. This problem is subtle enough that a proof of the correctness of the distributed implementation must be provided before the programmer relies on any semantics involving immediate sites.

4 Non-Distributed Implementation

I will summarize relevant details of the implementation of non-distributed ORC in order to better explain the changes introduced by the distributed implementation. In some cases I have introduced simplifications relative to the real implementation in order to aid understanding, but the core ideas are accurate. For a more thorough explanation of the implementation, see [1].

The basic approach of the interpreter is to first compile each ORC expression into a directed acyclic graph (DAG) with nodes in the graph representing primitive steps in the computation. Expressions are built by composing their sub-expressions, so that the final result is a single DAG for the main program and a separate DAG for each function body.

During evaluation, "tokens" are used to keep track of the state of an ongoing thread of computation. of the expression's DAG. The token corresponds to a continuation or a stack pointer in a sequential programming language. It tracks a variety of book-keeping information for the thread, including:

- the current node
- a value to publish
- an environment mapping variable names to futures
- a return pointer for function calls
- a group for asymmetric operations

Evaluation consists of placing a token at the root node of the expression DAG and then updating it according to the operation represented by the node. Some operations (like sequential composition) will update the token and move it on to the next node, while others (like parallel composition) may copy the token to create a new conceptual thread of execution.

The ORC interpreter maintains a list of active tokens and loops through these tokens in a single thread, removing each from the active list and processing it. This allows the ORC interpreter to support millions of ORC threads while only using one or two threads of the host operating system.

The following subsections address the relevant details of specific operations.

4.1 Site Calls

ORC evaluates a site call by looking up the values of the arguments from the environment, sending them to the site, and storing the current token while it waits for a response from the site.

When a response is received from the site, the stored token has its value set to the value returned by the site, then it is moved to the next node and activated.

4.2 Function Calls

ORC evaluates a function call by first creating a copy of the current token and moving it to the next node. The return pointer of the original token is set to this copy, so that it will know where to return when the function is complete. Finally, the token is moved to the root node of the body of the function, its environment set to the function's environment, and it is activated.

When a token reaches the end of the function body, it creates a copy of the token indicated by the return pointer, sets the copy's value to the value published by the function, and activates it. The returning token is no longer needed and can be discarded.

Unlike a site call, a function call can return multiple times, so the return pointer token must be kept until all the tokens in the function body have either reached the end of the function or died.

4.3 Asymmetric Combinator

The asymmetric combinator is special for two reasons: it can introduce a future into its left-hand side (a variable which does not yet have a value), and it can kill computations on its right-hand side when the right-hand side publishes a value. Both of these tasks are handled by a single object called a "group cell".

When a token reaches the node representing an asymmetric expression, a new group cell is created.

On the left-hand side of an asymmetric expression, the group cell is part of the environment and represents a pending value. If a token needs the value of the group cell (for example, to evaluate a site call), and the group cell does not yet have one, the token is put on a waiting list to be notified once the group cell's value is available.

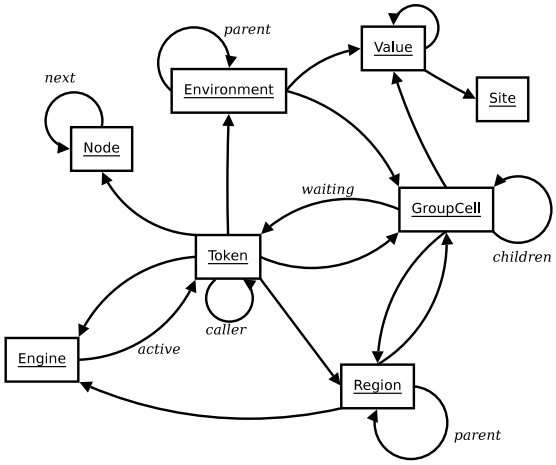

Figure 3: ORC communication

On the right-hand side of an asymmetric expression, the group cell is used to tag all of the tokens involved in generating its value. Before doing anything else, every token checks to see whether its group cell is still "alive" (does not yet have a value). If so, it can proceed. If not, it is part of the right-hand side of an asymmetric expression which has already produced a value and should be terminated.

When a token reaches the end of the right-hand side of an asymmetric expression, its value is copied to the group cell and any tokens from the left-hand side waiting on that value are notified.

5 Distributed Implementation

I had four goals for the distributed implementation of ORC:

- Dont hurt non-distributed computation
- Optimize to avoid communication
- Be conservative with optimization
- Keep as much of existing architecture and code as possible

In order to achieve these goals I preferred an evolutionary, rather than revolutionary, approach to the implementation. I began by studying the communication structure of the existing implementation (see Figure 3). I identified objects which could be safely copied between distributed servers, and those which must be shared. I estimated the frequency of communication between various objects. I used this information to identify objects which would always be local versus those which may reside on a remote server. Finally, I introduced proxies and adjusted communication patterns to optimize communication.

I suspect this approach to designing a distributed system could be formalized, somewhat like this:

1. Create a directed graph where nodes are classes and edges are messages
2. For any classes which may be safely copied between servers, introduce a separate copy of the corresponding node for every message it is involved in
3. Weight the edges according to the frequency of communication
4. Choose one node which must be local and one which must be remote
5. Apply a graph min-cut algorithm to find the set of edges isolating the local node from the remote node with minimum cost
6. This set of edges constitute the remote links

The result of this process was to establish the following list of messages which may need to go from local to remote servers and visa versa:

- Sending arguments to a site
- Returning a value from a site
- Activating a token at the start of a remote expression
- Publishing a value from a remote expression
- Assigning a value to a group cell
- Notifying a token waiting on a group cell that a value is available
- Killing tokens associated with a group cell which has received a value

The main task in implementing the distributed interpreter was therefore simply to add proxy objects so that these messages could be sent to a distributed server if necessary. For the actual implementation of distributed communication, I used Java RMI, which has several benefits:

- Objects on remote servers can be called using the same syntax as local objects.
- Arguments to remote methods are automatically serialized. Objects which are registered with RMI as remote objects are serialized to a remote reference which allows remote method calls, while all other objects are copied and not shared between servers.
- The RMI system provides garbage collection for remote references via reference counting[2].

Automatic serialization was by far the biggest benefit: the ORC environment may contain arbitrarily-complicated values, and implementing my own serialization routine for such values would be time consuming and also complicate future work on the ORC implementation.

The following sections will describe some of the details of the distributed implementation of specific operations.

5.1 Remote Expressions

DORC remote expressions are handled exactly like ORC function calls, with only two significant differences. First, since remote expressions are always executed in the environment of the caller, there is no need to change the environment of the token when it is moved to the body of the remote expression. Second, instead of simply moving the token to the body of the expression, it is serialized together with the expression DAG and sent to the remote server to be evaluated.

5.2 Remote Site Calls

The ORC environment itself is immutable and can safely be copied between servers. The same is true for the majority of primitive values used in ORC programs, including numbers, lists, tuples, and strings. However there are a few mutable sites, such as buffers, which must be shared between servers.

This sharing is implemented by translating references in the environment to such mutable sites into remote references when the environment is copied to a remote server. If the remote server tries to call such a site, it will be a remote site call. When making a remote site call, the arguments to the site and a remote reference to the return token must be serialized and passed to the site via distributed communication. When the site is ready to return a value, it uses the remote reference to send the value back to the token on the local site.

5.3 Remote Futures

The asymmetric combinator requires careful implementation in the distributed case. The naive approach is to simply allow a group cell to always be handled as a remote reference. This is bad for two reasons.

First, a token must check its group cell every time it is processed in order to ensure the group is still alive. If the group cell is a remote reference this introduces a significant overhead to every token processing step. This can be easily rectified by introducing a local proxy for the remote group cell. The token only has to check the local proxy, which will be automatically notified by the remote group cell when the group is killed.

Second, a token may communicate with a group cell to check if it has received a value, to wait on that value, and finally to be notified when the value arrives. If multiple tokens on the same server all need the value, they will make redundant requests: they will all request the value separately from the group cell, even though some other token on the same server may already know the value. My solution is to introduce a local cache on each server which acts as an intermediary between tokens and group cells. Instead of every token asking the group cell directly for a value, they ask the local cache. If the local cache does not have the value, it asks the group cell on their behalf. If necessary, the local cache waits for the value and is notified by the remote group cell when a value is ready, so that it in turn may notify all of the tokens waiting on its server.

One important question is whether these optimizations are correct. Specifically, when a group cell is killed, there may be a race condition due to the delay in notifying the

[2]RMI garbage collection does not collect cycles. It remains to be seen whether this is a problem for typical DORC programs.

tokens working on that group cell. How do we know this delay will not cause problems? Correctness requires that we must guarantee two things:

1. The group publishes exactly one value.
2. It should not be possible to observe progress of tokens in the group after the value is published.

The second requirement deserves further explanation. What does it mean to "observe progress ... after the value is published"? In a semantics with no immediate sites (such as that proposed for DOrc), this simply means that there cannot exist a causal relationship between the publishing of a value and some event which occurs on the right-hand side after the value is published. In other words, a token on the right-hand side should not observe any event from the left-hand-side that depends on the value being published, and a token on the left-hand side should not observe any event from the right-hand side that occurs after the value is published.

The following example illustrates both points:

```
let(x) >> d.put(1)
    <x< c.get()
      | (c.put(1) >> d.get())@r
    <c< Buffer()
    <d< Buffer()
```

If requirement (1) is not guaranteed, `x` may receive a value twice. If requirement (2) is not guaranteed, then the remote node may observe that the left-hand side of the asymmetric combinator has made progress by receiving a value via `d`.

Note that with no immediate sites, the following program may legally produce a value:

```
let(x) >> c.get()
    <x< c.get()
      | (c.put(1) >> c.put(2))@r
    <c< Buffer()
    <d< Buffer()
```

The explanation is that even though `c.put(1)` ultimately causes `x` to receive a value, there may be an arbitrary delay between sending the value to the channel and that value being received by the first `c.get()`, during which `c.put(2)` may still legally run. This would only be a problem if the program could observe definitively that `x` had received a value before `c.put(2)` started running, which is covered by requirement (2) above.

In my implementation the two requirements are guaranteed easily:

1. is guaranteed by a mutex on the group cell, which ensures only one value may be published at a time and ignores any attempts to publish values after the first has been published
2. is guaranteed by waiting until all tokens in the right-hand side have been notified of group death to proceed with the left-hand side

5.4 Deadlock Freedom

A valuable property of Orc is that any program which does not use mutable sites is free from deadlock. A proof sketch: in the absence of mutable sites, communication between any two concurrent computations is one-way. There can be no communication between the left and right sides of `|`, `>x>` does not enable communication between concurrent instances of the right sub-expression, and `<x<` only allows communication from the left side to the right. Therefore if one side of such a combinator depends on the other for progress, the converse cannot be true, so deadlock is impossible.

Because `DOrc` does not change the core combinators, this statement remains true provided the DOrc implementation itself is free from deadlock. I have not proven this latter result, but it should be fairly straightforward to do so. Because the DOrc implementation uses essentially the same communication structure as the original Orc implementation, assuming the original implementation is free from deadlock, the DOrc implementation only has to worry about distributed deadlock. Distributed deadlock can be avoided by ensuring that all distributed messages are non-blocking if they may need to acquire a lock. I accomplish this by running key distributed remote procedure calls in a separate thread, which makes them non-blocking with respect to the main thread.

With arbitrary stateful sites, deadlock freedom is clearly not guaranteed, in either the distributed or non-distributed case. Further research is needed to determine whether there is some useful subset of sites or structures

for expressions which preserve deadlock freedom while still enabling more interesting computation.

6 Conclusion and Future Work

Significant future work remains in improving the efficiency of the current implementation. There are three key areas for improvement:

- Remote references are not unpacked when sent back to their originating node. In other words, if a server creates a channel and passes it to another server, which then passes it back, the original server will end up with a remote reference to a local object. Anytime the server uses this object, it will incur needless overhead serializing arguments and transmitting them via the RMI protocol. The solution to this problem involves creating a global identifier for every remote object which is unchanged as the object is passed between servers. Each local server can keep a cache of identifiers for remote objects it originated, and when it receives such a remote object it can replace it with its local implementation.

- Whenever a remote expression is evaluated, the entire expression is copied to the remote server. Since the expression is immutable, this copying may be unnecessary if the remote expression was evaluated before. The solution is to copy the entire DAG to the remote server when it is used for the first time, and then for subsequent uses it can refer to its local copy of the DAG rather than being sent a new one. One complication is that this requires a global identifier for the node at the start of a remote expression, so that the remote server can be told where to evaluate a token.

- Finally, the entire environment is copied to the remote server whenever a remote expression is evaluated. If the expression only needs a small part of the environment this is very wasteful. Since infrastructure already exists to track free variables, it should be straightforward to identify the free variables in an expression and only copy the portion of the environment they refer to, transitively.

Another important area for future work is in proving the correctness of the distributed implementation, and providing stronger guarantees about timing. My intuition is that this requires a type system which can be used to prove assertions about sites, so that it is possible to automatically verify whether important semantic properties might be violated by distributing an expression.

Finally, the current DOrc implementation does not handle logical timers (`LTimers`) which are useful for implementing simulations in Orc. Adding this feature should be straightforward, although I considered it beyond the scope of the current work.

In conclusion, I have described a distributed extension to the Orc language which enables distributed programs to be written in a straightforward manner, and discussed some important properties of its correctness.

References

[1] William R. Cook and Jayadev Misra. Implementation outline of orc. December 2005. 6

[2] William R. Cook David Kitchin and Jayadev Misra. A language for task orchestration and its semantic properties. August 2006. 3

[3] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006. 2