Local Operations Should Appear to Be Remote
Consistent Semantics Enable Transparent Distribution

John A. Thywissen
Arthur Michener Peters
jthywiss@cs.utexas.edu
amp@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA

Christopher J. Rossbach
rossbach@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin
Austin, Texas, USA
VMware Research
Palo Alto, California, USA

1 INTRODUCTION

Programming languages have progressively taken on managing the tedious aspects of program operation. For example, most modern languages manage memory automatically, which imposes a performance cost. Yet, these costs are worthwhile for many applications because they are outweighed by the benefits gained by offloading responsibility to the language system.

Modern applications are increasingly becoming distributed applications. In current general-purpose distributed programming models, such as Java RMI, Microsoft .NET Remoting, Akka Actors, and gRPC, distribution is manual and tedious: Programmers explicitly partition programs into a collection of modules, each executing at a single location, and then specify each interaction. These modules are then placed in the locations where they will execute. This explicit separation and placement impedes programming by interfering with the application’s modularization and obscuring the valuable application-specific code.

Many alternative distributed programming models, such as MapReduce [4] or the Lambda Architecture [11], improve this situation by abstracting a specific communication pattern. However, the program’s modularization must conform to the given pattern, such as a map and reduce function with sets as parameters. We consider distribution of general purpose programs.

For many workloads, distribution is merely a technique to take advantage of resources available in the execution environment, which includes the hardware resources available to the program across a dynamically changing set of distributed nodes. In these cases, the interfaces between distributed parts of a program are immaterial to the developers and users. Some distributed programming languages abstracted away where each part of the program executes [2, 3, 6, 9, 19]. This language facility is called location transparency. This abstraction has been sharply criticized [20], and location transparency is not in widespread use. The essence of the criticism is that remote operations should not appear to be local operations. We agree, and propose the reverse: Make local operations appear to be remote operations, and support concurrency everywhere, rather than just between explicit threads. This approach is discarded in Waldo et al. [20] as too complex. We argue in this paper that, with the appropriate language semantics, this is not the case.

This “local seems like distributed” programming model seems inevitable, even in non-distributed applications. In modern application architectures, an increasing number of nominally local operations include inter-process communication, which raises the possibility of failure and increases latency. For example, application calls into crypto libraries were formerly completely local calls, but are now often performed in a separate address space (e.g., Windows CSPs). As shown in Table 1, even memory operations now have some characteristics of network operations in current processor architectures, for example in NUMA systems. These trends in both software and hardware architecture mean that, in the limit, all program operations would have what we now think of as “remote” semantics.

This paper presents the programming language design implications of location transparency, describes an instance of such a language, provides evidence for the language’s ease of use, discusses costs of location transparency, and illustrates the performance of a simple map-reduce workload.

2 LOCATION TRANSPARENCY NEEDS PERVERSIVE CONCURRENCY

Remote semantics add several concerns beyond local semantics, one of which is communication latency. To handle latency, programs use concurrency to avoid waiting on any result that the program does not yet need, and allow threads to proceed concurrently when possible. In a conventional language, expressing concurrency can be unwieldy. However, in a language that provides for structured, pervasive concurrency, it can be handled deftly. The orchestration language Orc [10, 13] is such a language. In pervasive concurrency, a program’s operations do not depend on sequential semantics, so the language system is free to adapt execution to the conditions actually present at runtime. Like dataflow languages, execution proceeds based on the availability of values, not control flow.

With location transparency, the program is not aware which operations cross node boundaries, so network latency is a potential throughout the program. Pervasive concurrency compensates for location transparency’s pervasive potential latency.

3 D-ORC, A LANGUAGE WITH LOCATION TRANSPARENCY

We have developed a location transparent distributed programming language, Distributed Orc (d-Orc) [17, 18], based on the pervasively concurrent language, Orc. The placement of execution across the
available infrastructure is managed by the language system, guided potentially by program static and dynamic analyses, by external data, and (in some cases) by hints in the program. This placement is not static: d-Orc transparently migrates executing operations among locations.

D-Orc implements the same programmer-visible language and semantics as Orc, but allows programs to execute across multiple locations. To allow this, d-Orc tracks the locations at which data reside and uses this information to determine locations at which to execute operations. Some data may reside on sets of locations, instead of just a single location.

As a simple example, consider the d-Orc expression max(getCurrentTemperature(), getForecastHigh()), where the operation getCurrentTemperature() is available at nodes that have thermometers, which we will denote as set $T$, operation getForecastHigh() is available at nodes that have a copy of the weather forecast, set $W$, and operation max is available at every location. Any location in the intersection $T \cap W$ can execute the whole expression without any need for communication. If $T \cap W = \emptyset$ or if the current execution location is not in $T \cap W$, then communication is required to execute the expression. For example, getCurrentTemperature() could be run on a location in $T$, and execution could migrate to a location in $W$, where the getForecastHigh() and max operations can be performed. Alternatively, the forecast could be copied to a location in $T$ (thereby expanding $W$), and the entire expression executed there. D-Orc’s semantics allows many different distributions, thereby enabling smart compilers and runtimes to optimize the distribution for various desired properties, including load-balancing.

D-Orc’s additions to Orc can be grouped into three areas, further described below:

- **Migration.** Migrating threads of execution in the program among distributed locations.
- **Value–Location Relation.** Identifying the dynamic locations of values in the distributed environment. D-Orc tracks both the current locations of a value and the permitted locations of a value.
- **Revised Semantics.** Extending existing Orc language features to handle remote values.

The Orc language is designed for concurrent orchestration, meaning that the Orc program manages various other modules that may or may not be written in Orc. These modules are called sites and they include everything from arithmetic operations to user interface devices. Sites are called with arguments and return a stream of zero or more values to their caller. Migration of execution in d-Orc is driven by the locations of the site and the call arguments. If a site call is to be executed, but the current location does not have local copies of these values, d-Orc can copy values between locations, migrate execution to another location, or some combination thereof. Copying values between locations does not require a coherence protocol or any other communication after the copy since all Orc-level values are immutable. Mutable data structures available through sites can either be restricted to a single location or reimplemented as a distributed data structure, using Orc or some other language.

The locations of existing copies, costs of communication, and any policy constraints on allowed locations of copies may be taken into account by the d-Orc implementation when choosing which migration and copy actions to take. These policies and costs will be represented using a configuration language. The specifics of the configuration language is not in the scope of this paper, because d-Orc is agnostic to how the configuration is expressed or how the runtime uses it.

Orc provides five **combinators** to process streams of returned values from site calls or expressions, namely parallel, branch, graft, trim, and otherwise. The parallel and branch combinators are unaffected by distribution, since their execution does not require coordination among subexpressions. For brevity, we omit further discussion of them here. All combinators are described in Peters et al. [13].

The **graft combinator** $\text{val} x = e_1 \# e_2$ creates a future. Graft executes both expressions $e_1$ and $e_2$ concurrently, with the variable $x$ bound to a future in expression $e_2$. The future will be resolved to the first returned value of expression $e_1$. If expression $e_2$ refers to the variable $x$ before it has been resolved, the reference to $x$ will block, awaiting the value. In the distributed setting, execution of a graft combinator must collect any returned value from the various locations where the expression $e_1$ is executing, and must provide the resolved value to any locations that read the variable $x$.

The **trim combinator** $[\{ e \}]$ provides the ability terminate other execution in the expression when any part returns a result. Trim is useful, for example, in concurrent divide-and-conquer activity, to stop the other concurrent activity when any subproblem is successful. In the distributed setting, execution of a trim combinator needs to coordinate the nomination of returned value from its subexpression $e$ among all locations executing parts of $e$, and notify all these locations to terminate further execution of the expression.

### Table 1: Attributes of operations in the classic single-core systems, in distributed systems, and in multiprocessor systems.

<table>
<thead>
<tr>
<th></th>
<th>Single-core</th>
<th>Distributed</th>
<th>Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Small; consistent</td>
<td>Large; variable</td>
<td>Moderate; variable</td>
</tr>
<tr>
<td>Processor resource availability</td>
<td>Guaranteed</td>
<td>Not guaranteed</td>
<td>Not guaranteed</td>
</tr>
<tr>
<td>Failure</td>
<td>External ops; rare</td>
<td>Potentially any op; unpredictable</td>
<td>Potentially any op; unpredictable</td>
</tr>
<tr>
<td>Ordering</td>
<td>Consistent</td>
<td>Unpredictable</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Locality of effects</td>
<td>Yes</td>
<td>No</td>
<td>Maybe</td>
</tr>
</tbody>
</table>

Multiprocessor systems are increasingly similar to distributed systems.
The otherwise combinator \( e_1 \); \( e_2 \) provides failure handling. Otherwise executes expression \( e_1 \), and if execution halts without a result, then executes the expression \( e_2 \). In the distributed setting, execution of an otherwise combinator must monitor execution, across any involved locations, of its left-hand expression \( e_1 \) for returned values and termination.

A core concern of distributed programs is failure recovery. To handle failure, Orc programs use well-known failure detection and recovery techniques, but Orc simplifies their implementation. In Orc, a failure results in an expression completing execution without producing any values. The otherwise combinator is used to detect this and execute a compensating action, such as a retry or reporting the failure. In the results reported below, we do not simulate any failures.

4 PROGRAMMABILITY EVALUATION

We compared six pairs of small programs to evaluate the programmability advantages of d-Orc. Each pair included an d-Orc program written by the authors, and a Java program written based on a publicly-available implementation and modified by the authors where necessary to ensure functional comparability. Each pair of programs uses the same algorithms and, where applicable, the same kind of synchronization and approach to partitioning application logic across distributed execution resources; all use communication frameworks that enable distribution across multiple machines.

In Table 2, we report the number of lines of code for both Java and d-Orc implementations, along with the percentage of those lines of code devoted to functionality other than core application logic, for example, launching worker processes or configuration of RMI registries. The remaining columns include a check mark where distribution and concurrency force the use of programming idioms in the Java implementations that are absent for d-Orc. The **sched/data-xfer** column indicates explicit scheduling and data transfer code is present. The **RMI** column indicates Java RMI is used to invoke remote work. The **synchronized** column indicates the implementation requires programmer-managed mutual exclusion. The **cond-sync** column indicates wait/notify+ is used. The **futures** column indicates implementations that use futures. The **sockets/msg** column indicates explicit communication using sockets or message passing.

The results show that d-Orc has huge potential to simplify distributed programming with respect to Java. The d-Orc versions are consistently shorter, with a geometric mean of 82% fewer lines of code. The d-Orc versions also devote a much larger fraction of their code to the application logic than the Java versions, with 97% of the code devoted to the application logic for d-Orc vs. 44% for Java. Any code that does not implement some element of the application logic is additional conceptual load for the programmer, and is code that must be maintained. Moreover, the vast majority of the additional code is devoted to functionality such as synchronization, which is challenging and bug-prone.

5 CHALLENGES OF LOCATION TRANSPARENCY

Transparent management of location introduces a significant challenge: the runtime scheduler is responsible for deciding when and where sub-computations should run. For some workloads, heuristics to guide placement decisions are straightforward. For example, in a Map-Reduce or Spark workload, locality is a guiding principle: computations should run on resources where their data is already present, otherwise on resources that are under-utilized. For “think-like-a-vertex” graph processing workloads [12] computations should transparently move to the node with the proper vertex. However, in the general case, transparent placement is likely a very hard problem. When done poorly, location transparency has the potential to introduce unacceptable overhead due to communication that could/should have been avoided.

5.1 Favorable Placement

The most drastic potential for poor performance comes from adverse placement of operations’ execution. Tightly-coupled operations should be collocated. Poor placement decisions can lead to behaviors analogous to cache line ping-ponging. We have focused on three areas to improve placement decisions:

**Runtime Adaptation.** When the runtime detects patterns of communication, it can place execution to eliminate or facilitate the communication. For example, values that are used by a sequence of operations can be collocated. The d-Orc site call migration approach enables such adaptation by placing few constraints on placement. Placement strategies can use information from program static analyses such as program flow graphs, dynamic analyses, and application-specific knowledge. Placement strategies may be speculative and can “undo” unproductive decisions.

**Static Analysis.** Static analysis could improve placement by determining where parts of the program should run based on where values will be and where the results will be needed. Such an analysis would use the techniques of bi-directional program slicing and distributed dependence graphs [5] along with a model which describes the communication costs of value copying and migration. For example, an optimizing d-Orc implementation could perform a program slicing analysis to take into account destination of results of operations, and choose execution nodes that reduce the need for communications later in the execution.

**Placement Hints or Specifications.** Alternatively, distribution specifications could partially specify where parts of the program should execute, and how values should be copied. These specifications would be external to the d-Orc program, and therefore permit adaptation to a particular execution environment without program modifications. This specifications could take the form of a concrete cost function which describes the communication costs of migration and the cost of running a given site in a given location.

The current experimental d-Orc implementation uses placement specifications. Static analysis and runtime adaptation are the next phases of the d-Orc work program.

5.2 Overhead Costs

Part of transparent distribution’s overhead costs are avoidable communication, which are reduced in two main ways:

**Language Design.** Extensive use of mutable state, as is typical in conventional languages, could impose a high cost of coordination
Table 2: Comparison of d-Orc versus Java implementations for a group of benchmarks. LoC were measured with SLOC-Count [21], and are reported along with the percentage of those lines devoted to things other than core application logic.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>d-Orc LoC</th>
<th>Java LoC</th>
<th>sched/data-xfer</th>
<th>RMI</th>
<th>synchronized cond-sync</th>
<th>futures</th>
<th>sockets/msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map–Reduce</td>
<td>31 (3%)</td>
<td>403 (74%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Randomized Byzantine Agreement</td>
<td>43 (0%)</td>
<td>718 (90%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td>65 (0%)</td>
<td>332 (56%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>28 (0%)</td>
<td>260 (60%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Depth-First Search</td>
<td>23 (0%)</td>
<td>138 (40%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Sudoku</td>
<td>60 (13%)</td>
<td>152 (35%)</td>
<td>✅</td>
<td>✔</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

of this state among nodes. Whenever a mutable variable is written, possibly from multiple locations, nodes need to come to a consensus on the resulting value. On the other hand, a concurrent language, such as Orc, minimizes the use of mutable state by encouraging functional-style programming. Orc’s language constructs also are designed to avoid coordination; of Orc’s five combinator types, two (parallel and branch) can execute without any distributed coordination. The other combinators may trigger communication, but there is no globally-shared state, ordering guarantees are flexible, and there are no global synchronization points necessitated by d-Orc’s semantics.

Program Optimization. Work on optimizing pervasively parallel programs [14] applies to distributed languages as well. For example, eliminating futures and otherwise combinators is important in d-Orc due to the cost of distributed synchronization.

6 AN EXAMPLE MAP–REDUCE APPLICATION

As preliminary evidence that location transparency can be used to implement real systems, we investigate a map–reduce workload, with the goal of having the d-Orc runtime transparently perform the work that requires an explicitly coded job manager and scheduler in a conventional implementation. We consider three variants of a map–reduce distributed word count application, presenting source code size and performance results for four variants:

(1) A conventional, pure Java, single-threaded implementation;
(2) A concurrent pure d-Orc variant, where each file is counted in parallel;
(3) A mixed d-Orc–Java variant, where each file is counted in parallel, but the d-Orc program orchestrates concurrent calls to a Java word count routine; and
(4) A Hadoop Java map–reduce variant, where the HDFS replication factor drives the level of concurrency.

See Table 3 for source code sizes. The pure d-Orc variant and mixed d-Orc–Java variant were written as single-node Orc programs. Adapting them to a distributed setting requires no changes to the single-node Orc programs’ source code. Instead of changes to the program source, an execution placement policy is given, which in this case specifies a file-to-node mapping. With this placement policy, the word count file operations migrate to the location appropriate for the file they are processing. When the counts are summed, these reduce operations communicate in a tree pattern, reflecting Orc’s afold (associative fold) library function’s tree recursion.

Method. We executed these variants on a cluster of 28 Dell Precision T1650 nodes, each with an Intel Xeon E3-1270 v2 CPU (4 cores each) at 3.50 GHz, with 16 GiB of physical RAM. The OS was Ubuntu 16.04.3 LTS with GraalVM 0.30.2, a modified Java SE Runtime Environment 1.8.0_151-b12. The JVM maximum heap size was set to 12 GiB.

The Hadoop variant was run on Hadoop 2.8.1, using 6 HDFS datanodes/YARN resource nodes. The HDFS replication factor and number of mappers was set to 6.

Each experimental condition (combination of program, input size, and cluster size) is repeated 20 times. The first 9 repetitions are
Table 3: Source code size of the map–reduce word count implementations. SLOC by language is the number of physical non-blank, non-comment source lines of code of the application logic, excluding test harness overhead code.

<table>
<thead>
<tr>
<th>Implementation variant</th>
<th>SLOC by language</th>
<th>Local-to-distributed adaptation effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded conventional Java</td>
<td>99 Java</td>
<td>Complete rewrite</td>
</tr>
<tr>
<td>Mixed d-Orc–Java</td>
<td>21 Orc, 42 Java</td>
<td>No source changes</td>
</tr>
<tr>
<td>Concurrent pure d-Orc</td>
<td>36 Orc</td>
<td>No source changes</td>
</tr>
<tr>
<td>Hadoop Java</td>
<td>76 Java</td>
<td>n/a (Written as distributed ab initio)</td>
</tr>
</tbody>
</table>

Results. Figure 1 shows the elapsed times for three of the variants to process 18.2 GB of input (split into 120 files of 152 MB each) in a single-node (cluster of one) configuration. Figure 2 shows the relative speed-up of the mixed d-Orc–Java variant for various cluster sizes. Figure 3 shows the elapsed time of the Hadoop variant versus the mixed d-Orc–Java variant for a cluster of 6 nodes. Our current Orc runtime has an undesirably high overhead for each Orc-to-Java call. In the pure Orc variant, this overhead dominates its performance. See Peters et al. [14] for an effort to improve Orc’s local performance.

Limitations. We need to perform this experiment for workloads beyond map–reduce. The Hadoop variant performance was run using Hadoop’s default configuration. Its performance may be improved by tuning, but it’s unclear how much tuning effort would be a good basis for comparison to the other variants.

7 RELATED WORK

There are a number of distributed languages with features similar to d-Orc. The most notable of such languages are the following: The Emerald language [2, 9] was an impressive early leader in this area. Emerald argued for a cohesive programming style, with no explicit communications (RPC, RMI) code. Obliq [3] was an influential language that provided location transparency, but without migration of objects among locations. Distributed Oz [6, 19] has strong similarities to the d-Orc approach; however, Orc has distinctly different combinators, and does not have the consistency needs implied by Oz’s constraint store. Dryad [8] is a distributed dataflow language with a set of combinators that have some similarities to Orc. Unlike Orc, Dryad’s execution engine contains a job manager that maintains a global state of the distributed computation graph to schedule work on cluster nodes. Several languages (Hop [16], STIP.JS [15]) address distribution in the Web development setting. Cω [1], now implemented as Language Integrated Query (LINQ), unifies distributed computation for the case of database queries. Relatedly, Remote Batch Invocation [7] handles code blocks in a language which computes on a mix of local and remote objects. Both Cω and Remote Batch Invocation implement powerful extended remote call semantics, but do not support full migration of execution.

8 CONCLUSION

We transformed a word counting program written as a single-node program into a distributed map–reduce program with no changes other than supplying the file-to-node mapping. The resulting distributed workload shows strong scaling at roughly 60% scaling efficiency. The map–reduce distributed computation pattern emerged
REFERENCES


