From Clarity to Efficiency for Distributed Algorithms *

Yanhong A. Liu Scott D. Stoller Bo Lin Michael Gorbovitski

Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, USA

{liu,stoller,bolin,mickg}@cs.stonybrook.edu

Abstract

This paper describes a very high-level language for clear description of distributed algorithms and optimizations necessary for generating efficient implementations. The language supports high-level control flows where complex synchronization conditions can be expressed using high-level queries, especially logic quantifications, over message history sequences. Unfortunately, the programs would be extremely inefficient, including consuming unbounded memory, if executed straightforwardly.

We present new optimizations that automatically transform complex synchronization conditions into incremental updates of necessary auxiliary values as messages are sent and received. The core of the optimizations is the first general method for efficient implementation of logic quantifications. We have developed an operational semantics of the language, implemented a prototype of the compiler and the optimizations, and successfully used the language and implementation on a variety of important distributed algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.2 [*Programming Languages*]: Language Classifications—Very high-level languages; D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; I.2.4 [*Computing Methodologies*]: Knowledge Representation Formalisms and Methods—Predicate logic

General Terms Algorithms, Design, Languages, Performance

Keywords distributed algorithms, incrementalization, logic quantifications, program optimization, very high-level languages

1. Introduction

Distributed algorithms are at the core of distributed systems. Yet, developing practical implementations of distributed algorithms with correctness and efficiency assurances remains a challenging, recurring task.

- Study of distributed algorithms has relied on either pseudocode with English, which is high-level but imprecise, or formal specification languages, which are precise but harder to understand, lacking mechanisms for building real distributed systems, or not executable at all.
- At the same time, programming of distributed systems has mainly been concerned with program efficiency and has relied mostly on the use of low-level or complex libraries and to a lesser extent on built-in mechanisms in restricted programming models.

What's lacking is (1) a simple and powerful language that can express distributed algorithms at a high level and yet has a clear semantics for precise execution as well as for verification, and is fully integrated into widely used programming languages for building real distributed systems, together with (2) powerful optimizations that can transform high-level algorithm descriptions into efficient implementations.

We have developed a very high-level language, DistAlgo, for clear description of distributed algorithms, combining advantages of pseudocode, formal specification languages, and programming languages.

- The main control flow of a process, including sending messages and waiting on conditions about received messages, can be stated directly as in sequential programs; yield points where message handlers execute can be specified explicitly and declaratively.
- Complex synchronization conditions can be expressed using high-level queries, especially quantifications, over message history sequences, without manually writing

* This work was supported in part by ONR under grant N000140910651 and N000140710928; and NSF under grant CCF-0964196 and CNS-0831298.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

message handlers that perform low-level incremental updates and obscure control flows.

DistAlgo supports these features by building on an object-oriented programming language. We also developed an operational semantics for the language. The result is that distributed algorithms can be expressed in DistAlgo clearly at a high level, like in pseudocode, but also precisely, like in formal specification languages, and be executed as part of real applications, as in programming languages.

Unfortunately, programs containing control flows with synchronization conditions expressed at such a high level are extremely inefficient if executed straightforwardly: each quantifier will cause a linear factor in running time, and any use of the history of messages sent and received will cause space usage to be unbounded.

We describe new optimizations that allow efficient implementations to be generated automatically, extending previous optimizations to distributed programs and to the most challenging quantifications.

- Our method transforms sending and receiving of messages into updates to message history sequences, incrementally maintains the truth values of synchronization conditions and necessary auxiliary values as those sequences are updated, and finally removes those sequences as dead code as appropriate.
- To incrementally maintain the truth values of general quantifications, our method first transforms them into set queries. In general, however, translating nested quantifications simply into nested queries can incur asymptotically more space and time overhead than necessary. Our transformations minimize the nesting of the resulting queries.
- Quantified order comparisons are used extensively in non-trivial distributed algorithms. They can be easily incrementalized when not mixed with other conditions or with each other. We systematically extract single quantified order comparisons and transform them into efficient incremental operations.

Overall, our method significantly improves time complexities and reduces the unbounded space used for message history sequences to the auxiliary space needed for incremental computation. Systematic incrementalization also allows the time and space complexity of the generated programs to be analyzed easily.

There has been a significant amount of related research, as discussed in Section 7. Our work contains three main contributions:

- A very high-level language that combines the best of pseudocode, specification, and programming languages.
- A systematic method for incrementalizing complex synchronization conditions with respect to all sending and receiving of messages in distributed programs.
- A general and systematic method for generating efficient implementations of arbitrary logic quantifications together with general high-level queries.

We have implemented a prototype of the compiler and the optimizations and experimented with a variety of important distributed algorithms, including Paxos, Byzantine Paxos, and multi-Paxos. Our experiments strongly confirm the benefits of a very high-level language and the effectiveness of our optimizations.

2. Expressing distributed algorithms

Even when a distributed algorithm appears simple at a high level, it can be subtle when necessary details are considered, making it difficult to understand how the algorithm works precisely. The difficulty comes from the fact that multiple processes must coordinate and synchronize to achieve global goals, but at the same time, delays, failures, and attacks can occur. Even determining the ordering of events is nontrivial, which is why Lamport's logical clock [33] is so fundamental for distributed systems.

Running example. We use Lamport's distributed mutual exclusion algorithm [33] as a running example. Lamport developed it to illustrate the logical clock he invented. The problem is that n processes access a shared resource, and need to access it mutually exclusively, in what is called a critical section (CS), i.e., there can be at most one process in a critical section at a time. The processes have no shared memory, so they must communicate by sending and receiving messages. Lamport's algorithm assumes that communication channels are reliable and first-in-first-out (FIFO).

Figure 1 contains Lamport's original description of the algorithm, except with the notation $<$ instead of $\Longrightarrow$ in rule 5 (for comparing pairs of timestamps and process ids) and with the word "acknowledgment" added in rule 5 (for simplicity when omitting a commonly omitted [19, 43] small optimization mentioned in a footnote). This description is the most authoritative, is at a high level, and uses the most precise English we found.

The algorithm satisfies safety, liveness, and fairness, and has a message complexity of $3(n-1)$. It is safe in that at most one process can be in a critical section at a time. It is live in that some process will be in a critical section if there are requests. It is fair in that requests are served in the order of the logical timestamps of the request messages. Its message complexity is $3(n-1)$ in that $3(n-1)$ messages are required to serve each request.

Challenges. To understand how this algorithm is carried out precisely, one must understand how each of the n processes acts as both P_i and P_j in interactions with all other processes. Each process must have an order of handling all the events according to the five rules, trying to reach its own goal of entering and exiting a critical section while also responding to messages from other processes. It must also

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message T_m*:*P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m*:*P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.

3. To release the resource, process P_i removes any T_m*:*P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

4. When process P_j receives a P_i *releases resource* message, it removes any T_m*:*P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m*:*P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $<$. (To define the relation $<$ for messages, we identify a message with the event of sending it.) (ii) P_i has received an acknowledgment message from every other process timestamped later than T_m.

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

Figure 1. Original description in English.

keep testing the complex condition in rule 5 as events happen.

State machine based formal specifications have been used to fill in such details precisely, but at the same time, they are lower-level and harder to understand. For example, a formal specification of Lamport's algorithm in I/O automata [43, pages 647-648] occupies about one and a fifth pages, most of which is double-column.

To actually implement distributed algorithms, details for many additional aspects must be added, for example, creating processes, letting them establish communication channels with each other, incorporating appropriate logical clocks (e.g., Lamport clock or vector clock [44]) if needed, guaranteeing the specified channel properties (e.g., reliable, FIFO), and integrating the algorithm with the application (e.g., specifying critical section tasks and invoking the code for the algorithm as part of the overall application). Furthermore, how to do all of these in an easy and modular fashion?

Our approach. We address these challenges with the DistAlgo language, compilation to executable programs, and especially optimization by incrementalization of expensive synchronizations, described in Sections 3, 4, and 5, respectively. An unexpected result is that incrementalization let us discover simplifications of Lamport's original algorithm in Figure 1; the simplified algorithm can be expressed using basically two send-statements, a receive-definition, and an await-statement.

3. DistAlgo Language

To support distributed programming at a high level, four main concepts can be added to commonly used object-oriented programming languages, such as Java and Python: (1) processes as objects, and sending of messages, (2) yield points and waits for control flows, and handling of received messages, (3) synchronization conditions using high-level queries and message history sequences, and (4) configuration of processes and communication mechanisms. DistAlgo supports these concepts, with options and generalizations for ease of programming. We have developed an operational semantics for DistAlgo.

Processes and sending of messages. Distributed processes are like threads except that each process has its private memory, not shared with other processes, and processes communicate by message passing. Three constructs are used, for defining processes, creating processes, and sending messages.

Process definition can use any class, say `P`, that extends a special class `Process`.

```
class P extends Process:
    class_body
```

So a process is an object of class `Process`. This is analogous to thread definition in Java and Python, except that `Process` is used in place of `Thread`, and that the fields of an object of class `Process` are local to the process. So, like for an object of `Thread`, one can define a `run` method and call `start` to start the process and execute the `run` method.

Process creation can use a statement of the following form, where `P` is a class that extends class `Process`, and `s` is an optional additional parameter that specifies a site, i.e., a machine, by its host name or IP address.

```
new P(...,s)
```

This creates a new process of class `P` on site `s`, or on the machine running this statement if `s` is omitted, and returns a reference to the process. This is the same as thread creation except for the additional parameter `s`. For high-level programming, `newprocesses(n,P,s)` creates and returns a set of `n` processes of class `P` on site `s`. Process references are ordered.

Sending messages to other processes uses a send-statement:

```
send m to p
```

This sends message `m` to process `p`. A message can be a value of any type and is usually a tuple where the first component is a string specifying the kind of the message. We allow a set `ps` of processes in place of `p`, to send the message to each process in `ps`.

Control flows and handling of received messages. The key idea is to use labels to specify program points where control flow can yield to handling of messages and resume

afterwards. Three constructs are used, for specifying yield points, handling of received messages, and synchronization.

A yield point is a statement of the following form, where `l` is a label that names this point in the program:

```
-- l
```

This specifies a program point, `l`, that can be referred to in specifying handling of messages, described next, to specify where the messages can be handled.

Handling of received messages uses receive-definitions, which are members of class definitions for processes and have the form:

```
receive m1 from p1,...,mi from pi at l1,...,lj:
    stmt
```

where each m_k is a variable or tuple pattern. This allows messages that match any one of m_1 `from` p_1, ..., m_i `from` p_i to be handled at yield points labeled any one of $l_1,\ldots,l_j$, by executing the statement `stmt` at those points. A tuple pattern is a tuple in which each component is a constant, a variable possibly prefixed with "=", or a wildcard. A constant or a variable prefixed with "=" means that the corresponding component of the tuple being matched must equal the constant or the value of the variable, respectively, for pattern matching to succeed. A variable not prefixed with "=" matches any value and gets bound to the corresponding part of the tuple being matched. A wildcard, written as "_", matches any value. The at-clause is optional, and the default is all yield points. The from-clause is also optional, and if used, the language provides the identity of the sender. Support for receive-definition mimics common usage in pseudocode, allowing a message handler to be associated with multiple yield points without using method definition and invocations. As syntactic sugar, a `receive` that is handled at only one yield point can be written at that point.

Synchronization can use await-statements of the form:

```
await bexp timeout time
```

This waits for the value of Boolean expression `bexp` to become true or until `time` seconds have passed. The timeout-clause is optional, and the default is to wait only for `bexp` to become true. If an await-statement exits due to a timeout, it sets `self.timeout` to `true`. If it exits due to the awaited condition being true, it sets `self.timeout` to `false`. We require that an await-statement be preceded by a yield point; if a yield point is not specified explicitly, the default is that all message handlers can be executed at this point. Otherwise, the program would deadlock here if `bexp` is false.

These few constructs make it easy to specify any process that has its own flow of control while also responding to messages. It is also easy to specify any process that only responds to messages, for example, by writing just receive-definitions and a `run` method containing only `await false`, or by writing just a `run` method containing only a `while true` loop whose body is a receive-definition.

Synchronization conditions using high-level queries. Synchronization conditions and other conditions can be expressed using high-level queries—quantifications, comprehensions, and aggregates—over sets of processes and sequences of messages. High-level queries are used commonly in distributed algorithms because (1) they make complex synchronization conditions clearer and easier to write, and (2) the theoretical efficiency of distributed algorithms is measured by message complexity, not time complexity of local processing.

Quantifications are especially common because they directly capture the truth values of synchronization conditions. We discovered a number of errors in our initial programs that used aggregates in place of quantifications before we developed the method to systematically optimize quantifications. For example, we regularly expressed "`v` is larger than all elements of `s`" as `v > max(s)` and either forgot to handle the case that `s` is empty or handled it in ad hoc fashions. Naive use of aggregates like `max` may also hinder generation of more efficient implementations.

We define operations on sets; operations on sequences are the same except that elements are processed in order, and square brackets are used in place of curly braces.

- Quantifications are of the following two forms. Each variable v_i enumerates elements of the set value of expression exp_i; the return value is whether, for each or some, respectively, combination of values of $v_1,\ldots,v_k$, the value of Boolean expression `bexp` is `true`. When an existential quantification returns true, variables $v_1,\ldots,v_k$ are bound to a witness.

  ```
  each v1 in exp1, ..., vk in expk | bexp
  some v1 in exp1, ..., vk in expk | bexp
  ```

- Comprehensions are of the following form. Each variable v_i enumerates elements of the set value of expression exp_i; for each combination of values of $v_1,\ldots v_k$, if the value of Boolean expression `bexp` is `true`, the value of expression `exp` forms an element of the resulting set.

  ```
  {exp: v1 in exp1, ..., vk in expk | bexp}
  ```

  We abbreviate `{v: v in exp | bexp}` as `{v in exp | bexp}`.

- Aggregates are of the form `agg(exp)`, where `agg` is an operation, such as `size`, `sum`, or `max`, specifying the kind of aggregation over the set value of `exp`.

- In the query forms above, each v_i can also be a tuple pattern, in which case each enumerated element of the set value of exp_i is first matched against the pattern before expression `bexp` is evaluated. We omit `|bexp` when `bexp` is `true`.

We use `{}` for empty set; use `s.add(x)` and `s.del(x)` for element addition and deletion, respectively; and use `x in s` and `x not in s` for membership test and its negation, respectively. We allow tuple patterns to be used in any access

of set elements. We assume that hashing is used in implementing sets, and the expected time of set membership tests and updates involving one element is $O(1)$.

DistAlgo has two built-in sequences, `received` and `sent`, containing all messages received and sent, respectively, by a process.

- Sequence `received` is updated only at yield points. An arrived message `m` for which the program contains a matching receive-definition is added to `received` when the program reaches a yield point where `m` is handled, and all matching message handlers associated with that yield point are executed for `m`. An arrived message for which the program contains no matching receive-definitions is added to `received` at the next yield point. The sequence `sent` is updated at each send-statement.
- We use `received(m from p)` as a shorthand for `m from p in received`; `from p` is optional, but when specified, each message in `received` is automatically associated with the corresponding sender. We use `sent(m to p)` as a shorthand for `m to p in sent`; `to p` is optional, but when specified, `p` is the process or set of processes in the corresponding send-statement.

If implemented straightforwardly, `received` and `sent` can create a huge memory leak, because they can grow unbounded, preventing their use in practical programming.

Configuration. One can specify channel types, handling of messages, setup for starting processes, and other configuration items. Such specifications are declarative, so that algorithms can be expressed without unnecessary implementation details. We describe a few basic kinds of configuration items.

Channel can be specified to be `fifo`, for FIFO, in which case messages between two processes are guaranteed to be received in the order that they were sent. This is specified using:

```
use fifo_channel
```

Similarly, channels can be specified to be reliable using `use reliable_channel`. By default, channels are not required to be FIFO or reliable. One can also specify different channel types for different channels.

One can specify how much effort is spent processing messages at yield points. For example,

```
use handling_all
```

means that all matching received messages that are not yet handled must be handled before execution of the main flow of control continues past any yield point; this is the default. For another example, one can specify a time limit. One can also specify different handling effort for different yield points.

Logical clocks [17, 33, 44] are used in many distributed algorithms. One can specify that Lamport logical clock is used:

```
use Lamport_clock
```

which configures sending and receiving of messages to update the clock appropriately; one can call `Lamport_clock()` to get the value of the clock. This can be implemented with a module that provides the function `Lamport_clock()` as well as the functions called at sending and receiving of messages.

Other language constructs. For other constructs, we use those in high-level object-oriented languages. We mostly use Python syntax (indentation for scoping, '`:`' for separation, '`#`' for comments, etc.), for succinctness, except with a few conventions from Java (keyword `extends` for subclass, keyword `new` for object creation, and omission of `self`, equivalent of `this` in Java, when there is no ambiguity), for ease of reading.

Example. Figure 2 shows Lamport's algorithm expressed in DistAlgo. The algorithm in Figure 1 corresponds to the body of `cs` and the two receive-definitions, 15 lines total; the rest of the program, 15 lines total, shows how the algorithm is used in an application. The execution of the application starts with method `main`, which configures the system to run (lines 24-30). Method `cs` and the two `receive`-definitions are executed when needed and follow the five rules in Figure 1 (lines 5-20).

Note that Figure 2 is not meant to replace Figure 1, but to realize Figure 1 in a precisely executable manner. Figure 2 is meant to contrast with lower-level specifications and programs.

4. Compiling to executable programs

Compilation generates code to create processes on the specified machine, take care of sending and receiving messages, and realize the specified configuration. In particular, it inserts appropriate message handlers at each yield point.

Processes and sending of messages. Process creation is compiled to creating a process on the specified or default machine and that has a private memory space for its fields. Each process is implemented using two threads: a main thread that executes the main flow of control of the process, and a helper thread that receives and enqueues messages sent to this process. High-level programming constructs, such as `newprocesses(n,P,s)`, can easily be compiled into loops.

Sending a message `m` to a process or set of processes, `p`, is compiled into calls to a standard message passing API. If the sequence `sent` is used in the program, we also insert `sent.add(m to p)` to be executed. Calling a method on a remote process object is compiled into a remote method call.

Control flows and handling of received messages. Each yield point `l` is compiled into a call to a message handler method `l()` that updates the sequence `received`, if it is used in the program, and executes the bodies of the receive-definitions whose at-clause includes `l`. Precisely:

```
 1 class P extends Process:
 2   def setup(s):
 3     self.s = s              # set of all other processes
 4     self.q = {}             # set of pending requests

 5   def cs(task):             # for calling task() in CS
 6     -- request
 7     self.c = Lamport_clock()                    # 1 in Fig 1
 8     send ('request', c, self) to s              #
 9     q.add(('request', c, self))                 #
                               # wait for own req < others in q
                               #  and for acks from all in s
10     await each ('request',c2,p2) in q |         # 5 in Fig 1
             (c2,p2) != (c,self) implies (c,self) < (c2,p2)
11           and each p2 in s |                    #
               some received('ack',c2,=p2) | c2 > c
12     task()                  # critical section
13     -- release
14     q.del(('request', c, self))                 # 3 in Fig 1
15     send ('release', Lamport_clock(), self) to s #

16   receive ('request', c2, p2):                  # 2 in Fig 1
17     q.add(('request', c2, p2))                  #
18     send ('ack', Lamport_clock(), self) to p2   #

19   receive ('release', _, p2):                   # 4 in Fig 1
20     q.del(('request', _, =p2))                  #

21   def run():                # main method for the process
       ...                     # may do non-CS tasks of the proc
22     def task(): ...         # define critical section task
23     cs(task)                # call cs to do task in CS
       ...                     # may do non-CS tasks of the proc

24 def main():                 # main method for the application
     ...                       # other tasks of the application
25   use reliable_channel      # configure channel to be reliable
26   use fifo_channel          # configure channel to be FIFO
27   use Lamport_clock         # configure to use Lamport clock
28   ps = newprocesses(50,P)   # create 50 processes of P class
29   for p in ps: p.setup(ps-{p}) # pass to each proc other procs
30   for p in ps: p.start()    # start each proc, call method run
     ...                       # other tasks of the application
```

Figure 2. Original algorithm (lines 3-4 and 6-20) in a complete program in DistAlgo.

1. Each receive-definition is compiled into a method that takes a message `m` as argument, matches `m` against the message patterns in the receive-clause, and if the matching succeeds, binds the variables in the pattern appropriately, and executes the statement in the body of this receive-definition.
2. Method `l()` compiled for yield point `l` does the following: for each message `m` from `p` in the queue of messages not yet handled, (1) if `m` matches a message pattern in a receive-definition whose at-clause includes `l`, then execute `received.add(m from p)` if `received` is used in the program and call the methods generated from the receive-definitions whose at-clause includes `l`; (2) if `m` does not match any message pattern in any receive-definition, then execute `received.add(m from p)` if `received` is used in the program. In both these cases, remove `m` from the message queue afterward.

An await-statement can be compiled into a synchronization using busy-waiting or blocking. For example, for busy-waiting, a statement `await bexp` that immediately follows a label `l` is compiled into a call `l()` followed by `while not bexp: l()`.

Configuration. Configuration options are taken into account during compilation in a straightforward way. Libraries and modules are used as much as possible. For example, when `fifo_channel` and `reliable_channel` are specified, the compiler can generate code that uses TCP sockets.

5. Incrementalizing expensive synchronizations

Incrementalization transforms expensive computations into efficient incremental computations with respect to updates to the values on which the computations depend. It (1) identifies all expensive queries, (2) determines all updates to the parameters of these queries, and (3) transforms the queries and updates into efficient incremental computations. Much of this has been studied previously.

The new method here is for (1) systematic handling of quantifications for synchronization as expensive queries, especially nested alternating universal and existential quantifications and quantifications containing complex order comparisons and (2) systematic handling of updates caused by all sending, receiving, and handling of messages in the same way as other updates in the program. The result is drastic reduction of both time and space complexities.

Expensive computations using quantifications. Expensive computations in general involve repetition, including loops, recursive functions, comprehensions, aggregates, and quantifications over collections. Loops were studied most; less for recursive functions and comprehensions, and least for quantifications, basically corresponding to how frequently each construct has traditionally been used in programming. However, high-level queries are increasingly used in programming, and quantifications are dominantly used in writing synchronization conditions and assertions in specifications and very high-level programs. Unfortunately, if implemented straightforwardly, each quantification incurs a cost factor that is linear in the size of the collection quantified over.

Optimizing expensive quantifications in general is difficult, which is a main reason that they are not used in practical programs, not even logic programs, and programmers manually write more complex and error-prone code. The difficulty comes from expensive enumerations over collections and complex combinations of join conditions. We address this challenge by converting quantifications into aggregate queries that can be optimized systematically using previously studied methods. However, a quantification can be converted into multiple forms of aggregate queries. Which one to use depends on what kinds of updates must be handled, and on how the query can be incrementalized under those updates. Direct conversion of nested quantifications into nested queries can lead to much more complex incremental computation code and asymptotically worse time and

space complexities for maintaining the intermediate query results.

Note that, for an existential quantification, we convert it to a more efficient aggregate query if a witness is not needed; if a witness is needed, we incrementally compute the set of witnesses.

Converting quantifications to aggregate queries. We present all converted forms here and describe which forms to use after we discuss the updates that must be handled. The process to develop them was nontrivial, even though the end results look simple. The correctness of all rules presented are proved using first-order logic and set theory. These rules ensure that the value of a resulting query expression equals the value of the original quantified expression.

Table 1 shows general rules for converting single quantifications into equivalent queries that use `size` aggregates. These rules are general because `bexp` can be any Boolean expression, but they are for converting single quantifications. Nested quantifications could be converted one at a time from inside out, but the results can be much more complicated than necessary. For example,

```
each x in s | some y in t | bexp
```

would be converted using rule 1 to

```
each x in s | size({y in t | bexp}) != 0
```

and then using rule 2 to

```
size({x in s | size({y in t | bexp}) != 0})
== size(s)
```

	Quantification	Using Aggregate
1	`some x in s | bexp`	`size({x in s | bexp}) != 0`
2	`each x in s | bexp`	`size({x in s | bexp}) == size(s)`
3		`size({x in s | not bexp}) == 0`

Table 1. Rules for converting single quantifications.

Table 2 shows general rules for converting nested quantifications into equivalent, but non-nested, queries that use `size` aggregates. These rules yield much simpler results than repeated use of the rules in Table 1. For example, rule 2 in this table yields a much simpler result than using two rules in Table 1 in the previous example. More significantly, rules 1, 4, and 5 generalize to any number of the same quantifier, and rules 2 and 3 generalize to any number of quantifiers with one alternation. We have not encountered more complicated quantifications than these. It is well-known that more than one alternation is rarely used, so commonly used quantifications can all be converted to non-nested aggregate queries.

Table 3 shows general rules for converting single quantifications with a single order comparison into equivalent queries that use `max` and `min` aggregates. These rules are useful because single quantified order comparison, when there are no element deletions, can be computed more efficiently, with a constant instead of linear space overhead. Boolean combinations of order comparisons and other conditions can be transformed first into quantifications each involving at most one order comparison at a time.

	Existential	Using Aggregate
1	`some x in s | y <= x`	`s != {} and y <= max(s)`
2	`some x in s | x >= y`	
3	`some x in s | y >= x`	`s != {} and y >= min(s)`
4	`some x in s | x <= y`	
5	`some x in s | y < x`	`s != {} and y < max(s)`
6	`some x in s | x > y`	
7	`some x in s | y > x`	`s != {} and y > min(s)`
8	`some x in s | x < y`	

	Universal	Using Aggregate
9	`each x in s | y <= x`	`s == {} or y <= min(s)`
10	`each x in s | x >= y`	
11	`each x in s | y >= x`	`s == {} or y >= max(s)`
12	`each x in s | x <= y`	
13	`each x in s | y < x`	`s == {} or y < min(s)`
14	`each x in s | x > y`	
15	`each x in s | y > x`	`s == {} or y > max(s)`
16	`each x in s | x < y`	

Table 3. Rules for single quantified order comparison.

Table 4 shows general rules for decomposing combinations of conditions in general quantifications, to extract quantifications each involving a single order comparison. For example,

```
each x in s | bexp implies y < x
```

can be converted using rule 6 to

```
each x in {x in s | bexp} | y < x
```

which can be converted using rule 13 of Table 3 to

```
{x in s | bexp} == {} or y < min({x in s | bexp})
```

	Quantification	Decomposed Quantifications
1	`some x in s` `| e1 and e2`	`some x in {x in s | e1}` `| e2`
2	`some x in s` `| e1 or e2`	`some x in s | e1 or` `some x in s | e2`
3	`some x in s` `| e1 implies e2`	`some x in s | not e1 or` `some x in s | e2`
4	`each x in s` `| e1 and e2`	`each x in s | e1 and` `each x in s | e2`
5	`each x in s` `| e1 or e2`	`each x in {x in s | not e1}` `| e2`
6	`each x in s` `| e1 implies e2`	`each x in {x in s | e1}` `| e2`

Table 4. Rules for decomposing conditions to extract quantified comparisons.

Updates caused by message passing. Parameters of a query are variables in the query whose values may affect the query result. Updates to a parameter are operations that

	Nested Quantifications	Using Aggregate
1	`some x in s | some y in t | bexp`	`size({true: x in s, y in t | bexp}) != 0`
2	`each x in s | some y in t | bexp`	`size({x: x in s, y in t | bexp}) == size(s)`
3	`some x in s | each y in t | bexp`	`size({x: x in s, y in t | not bexp}) != size(s)`
4	`each x in s | each y in t | bexp`	`size({(x,y): x in s, y in t | bexp}) == size({(x,y): x in s, y in t})`
5		`size({(x,y): x in s, y in t | not bexp}) == 0`

Table 2. Rules for converting nested quantifications.

may change the value of the parameter. The most common updates are assignments, `v = exp`, which is an update to `v`. Other updates can all be expressed as assignments. For objects, all updates can be expressed as field assignments, `o.f = exp`. For collections, all updates can be expressed as initialization to empty and element additions and removals.

For distributed algorithms, a distinct class of important updates are caused by message passing. Updates are caused in two ways:

1. Sending and receiving messages updates the sequences `sent` and `received`, respectively. Before incrementalization, code is generated, as described in Section 4, to explicitly perform these updates.
2. Handling of messages by code in receive-definitions updates variables that are parameters of the queries for computing synchronization conditions, or that are used to compute the values of these parameters.

Once these are established, updates can be determined using previously studied analysis methods, e.g., [21, 39].

Incremental computation. Given expensive queries and updates to the query parameters, efficient incremental computations can be derived for large classes of queries and updates based on the language constructs used in them or by using a library of rules built on existing data structures [39–41, 50].

For aggregate queries converted from quantifications, algebraic properties of the aggregate operations are exploited to efficiently handle possible updates. In particular, each resulting aggregate query result can be obtained in $O(1)$ time and incrementally maintained in $O(1)$ time per update to the sets maintained and affected plus the time for evaluating the conditions in the aggregate query once per update. Additionally, if `max` and `min` aggregates are used and there are no element deletions from the sets queried, the space overhead is constant. Note that if `max` and `min` are used naively and there are element deletions, there would be an overhead of $O(n)$ space and $O(\log n)$ update time from using more sophisticated data structures to maintain the `max` or `min` under element deletion [13, 22, 67, 68].

To allow the most efficient incremental computation under all given updates, our method transforms each top-level quantification as follows:

- For nested quantifications, the rules in Table 2 are used. For non-nested quantifications, if the conditions contain no order comparisons or there are deletions from the sets or sequences whose elements are compared, the rules in Table 1 are used. The space overhead is linear in the sizes of the sets maintained and being aggregated over.
- For non-nested quantifications, if the conditions contain order comparisons and there are no deletions from the sets or sequences whose elements are compared, the rules in Table 4 are first used to extract single quantified order comparisons, and then the rules in Table 3 are used to convert the extracted quantifications. In this case, the space overhead is constant.
- Multiple ways of conversion may be possible: for universal quantifications using rules 2 and 3 in Table 1 and rules 4 and 5 in Table 2, for nested quantifications with two or more alternations using rules 2 and 3 in Table 2 (each way of conversion corresponds to a choice of which two alternating quantifiers to eliminate using one of the rules), and for quantifications with symmetric ways of decomposing combinations of conditions using rules 1, 5, and 6 in Table 4. Our method transforms in all these ways, obtains the time and space complexities for each result, and chooses one with the best complexities.

Table 5 summarizes well-known incremental computation methods for these aggregate queries. The methods are expressed as incrementalization rules: if a query in the program matches the query form in the table, and each update to a parameter of the query in the program matches an update form in the table, then transform the query into the corresponding replacement given in the table and insert at each update the corresponding maintenance; a fresh variable is introduced for each different query.

The overall incrementalization algorithm [39, 40, 50] (1) introduces new variables to store the results of expensive queries and subqueries, as well as appropriate additional values, (2) transforms the queries and subqueries to use the stored query results and additional values, and (3) transforms updates to query parameters to also do incremental maintenance of the stored query results and additional values.

If queries are nested, inner queries are transformed before outer queries. Note that a comprehension such as `{x in s \| bexp}` is incrementalized with respect to changes to parameters of Boolean expression `bexp` as well as addition and removal of elements of `s`; if `bexp` contains nested subqueries, then after the subqueries are transformed, incremental main-

Query	Replacement	Cost
`size(s)`	`count`	$O(1)$
Updates	**Inserted Maintenance**	**Cost**
`s = {}`	`count = 0`	$O(1)$
`s.add(x)`	`if x not in s: count += 1`	$O(1)$
`s.del(x)`	`if x in s: count -= 1`	$O(1)$

Query	Replacement	Cost
`max(s)`	`maximum`	$O(1)$
Updates	**Inserted Maintenance**	**Cost**
`s = {x}`	`maximum = x`	$O(1)$
`s.add(x)`	`if x > maximum: maximum = x`	$O(1)$

Table 5. Incrementalization rules for `size` and for `max`. The rule for `min` is similar to the rule for `max`.

tenance of their query results become additional updates to the enclosing query. This is one of the reasons that incrementalization is challenging.

At the end, variables and computations that are dead in the transformed program are eliminated. In particular, sequences `received` and `sent` will be eliminated as appropriate, because queries using them have been compiled into message handlers that only store and maintain values needed for incremental evaluation of the synchronization conditions.

Example. In the program in Figure 2, three quantifications are used in the synchronization condition in the await-statement, and two of them are nested. The condition is copied below, except that `('ack',c2,p2) in received` is now used.

```
each ('request',c2,p2) in q |
  (c2,p2) != (c,self) implies (c,self) < (c2,p2)
and each p2 in s |
  some ('ack',c2,=p2) in received | c2 > c
```

Converting quantifications into aggregates as described using Tables 1 through 4 proceeds as follows. In the first conjunct, the universal quantification is converted using rule 2 or 3 in Table 1, because it contains an order comparison with elements of `q` and there are element deletions from `q`; rule 3 is used here because it is slightly simpler after the negated condition is simplified. In the second conjunct, the nested quantification is converted using rule 2 in Table 2. The resulting expression is:

```
size({('request',c2,p2) in q |
        (c,self) > (c2,p2)})  ==  0
and
size({p2: p2 in s, ('ack',c2,=p2) in received |
        c2 > c})  ==  size(s)
```

Updates to parameters of the first conjunct are additions and removals of requests to and from `q`, and also assignment to `c`. Updates to parameters of the second conjunct are additions of ack-messages to `received`, and assignment to `c`, after the initial assignment to `s`.

Incremental computation [39–41, 50] introduces variables to store the values of all three aggregates in the converted query, transforms the aggregates to use the introduced variables, and incrementally maintains the stored values at each of the updates, yielding the following:

- For the first conjunct, store the set value and the `size` value in two variables, say `earlier` and `count`, respectively; when `c` is assigned a new value, let `earlier` be `q` and let `count1` be its size, taking $O(|\texttt{earlier}|)$ time, amortized to $O(1)$ time when each request in `earlier` is served; when a request is added to `q`, if `c` is defined and `(c,self) > (c2,p2)` holds, add the request to `earlier` and increment `count1` by 1, taking $O(1)$ time; similarly for deletion from `q`.

  Note that at the addition and removal of `('request',c,self)` in particular, `earlier` and `count1` are not updated, because `(c,self) > (c,self)` is trivially false.

- For the second conjunct, store the set value and the two `size` values in three variables, say `responded`, `count2`, and `total`, respectively; when `s` is initialized in `setup`, assign `total` the size of `s`, taking $O(|\texttt{s}|)$ time, done only once for each process; when `c` is assigned a new value, let `responded` be `{}`, and let `count2` be 0, taking $O(1)$ time; when an ack-message is added to `received`, if the associated conditions hold, increment `count2` by 1, taking $O(1)$ time.

Note that incrementalization uses basic properties about primitives and libraries. These properties are incorporated in incrementalization rules. For the running example, the property used is that a call to `Lamport_clock()` returns a timestamp larger all timestamps of messages previously received, and thus at the assignment to `c`, we have that `earlier` is `q` and `responded` is `{}`.

Figure 3 shows the optimized program after incrementalization of the synchronization condition on lines 10-11 in Figure 2. All commented lines are new except that the synchronization condition in the await-statement is simplified. The synchronization condition now takes $O(1)$ time, compared with $O(|\texttt{s}|^2)$ if implemented straightforwardly. The trade-off is the much smaller amortized $O(1)$ time overhead at updates to `c` and `q` and on receiving of ack-messages.

Note that the sequence `received` used in the synchronization condition in Figure 2 is no longer used after incrementalization. All values needed for evaluating the synchronization condition are stored in new variables introduced: `earlier`, `count1`, `responded`, `count2`, and `total`, a drastic space improvement from unbounded for `received` to linear in the number of processes.

Simplifications to the original algorithm. Consider the original algorithm in Figure 2. Note that incrementalization determined that there is no need for a process to update auxiliary values for its own request. Based on this, we discovered that updates to `q` for a process's own request do not

```
1 class P extends Process:
2   def setup(s):
3     self.s = s
4     self.q = {}
5     self.total = size(s)        # total num of other procs

6   def cs(task):
7     -- request
8     self.c = Lamport_clock()
9     self.earlier = q            # set of pending earlier reqs
10    self.count1 = size(earlier) # num of pending earlier reqs
11    self.responded = {}         # set of responded procs
12    self.count2 = 0             # num of responded procs
13    send ('request', c, self) to s
14    q.add(('request', c, self))
15    await count1 == 0
            and count2 == total   # use maintained results
16    task()
17    -- release
18    q.del(('request', c, self))
19    send ('release', Lamport_clock(), self) to s

20  receive ('request', c2, p2):
21    if c != undefined:          # if c is defined
22      if (c,self) > (c2,p2):    # comparison in conjunct 1
23        if ('request',c2,p2) not in earlier: # if not in
24          earlier.add(('request', c2, p2)) # add to earlier
25          count1 += 1                      # increment count1
26    q.add(('request', c2, p2))
27    send ('ack', Lamport_clock(), self) to p2

28  receive ('ack', c2, p2):      # new message handler
29    if c2 > c:                  # comparison in conjunct 2
30      if p2 in s:               # membership in conjunct 2
31        if p2 not in responded: # if not responded already
32          responded.add(p2)     # add to responded
33          count2 += 1           # increment count2

34  receive ('release', _, p2):
35    if c != undefined:          # if c is defined
36      if (c,self) > (c2,p2):    # comparison in conjunct 1
37        if ('request',c2,p2) in earlier:   # if in earlier
38          earlier.del(('request', c2, p2)) # delete it
39          count1 -= 1                      # decrement count1
40    q.del(('request', _, =p2))
```

Figure 3. Optimized program after incrementalization. Definitions of `run` and `main` are as in Figure 2.

affect the only use of `q`, on line 10, so we can remove the updates to `q` on lines 9 and 14 as well as the test `(c2,p2) != (c,self)`, which becomes always true, in the synchronization condition. Furthermore, note that the remaining updates to `q` are merely maintaining pending requests by others, so we can remove lines 4, 17, 20 and the entire receive-definition for release, by using, for the first conjunct in the await-statement,

```
each received('request',c2,p2) |
  not (some received('release',c3,=p2) | c3 > c2)
  implies (c,self) < (c2,p2)
```

Figure 4 shows the simplified algorithm. Incrementalizing this program yields essentially the same optimized program as in Figure 3.

6. Implementation and experiments

We have implemented a prototype compiler and optimizer for DistAlgo. The system can parse a DistAlgo program, apply analyses and optimizations to it, and generate executable Python code.

```
1 class P extends Process:
2   def setup(s):
3     self.s = s

4   def cs(task):
5     -- request
6     self.c = Lamport_clock()
7     send ('request', c, self) to s
8     await each received('request',c2,p2) |
              not (some received('release',c3,=p2) | c3 > c2)
              implies (c,self) < (c2,p2)
9           and each p2 in s |
              some received('ack',c2,=p2) | c2 > c
10    task()
11    -- release
12    send ('release', Lamport_clock(), self) to s

13  receive ('request', c2, p2):
14    send ('ack', Lamport_clock(), self) to p2
```

Figure 4. Simplified original algorithm. Definitions of `run` and `main` are as in Figure 2.

We implemented DistAlgo as slightly extended Python because Python has rich support of very high-level constructs for ease of programming, and simple and consistent syntax for ease of reading. We mostly exploit Python's support for comprehensions, aggregates, and quantifications over sets and sequences, albeit with a slightly different syntax than in this paper. Processes are implemented using Python's multiprocessing package. Message passing is implemented using the socket library, with support for UDP or TCP as the underlying protocol. Await-statements are compiled into blocking synchronization, not busy-waiting.

Our compiler and optimizer are implemented in Python, building on Python's parser module and AST package. Our compiler runs on either a modified or unmodified Python parser. The modified parser is a 390-line patch to the C code of the Python source distribution and extends the Python grammar to support labels and message handlers with the syntax used in this paper. When using unmodified Python, our compiler supports labels and message handlers specified using specially named Python statements and methods; this alternate syntax avoids the need for the Python parser patch. The latter approach is used also for the rest of the extensions: process creation, sending messages, synchronization, and configurations. The rest of the compiler and optimizer consists of about 1850 lines of Python, including code for transforming quantifications into set and aggregate queries, and excluding code for applying incrementalization to those queries. Applying incrementalization uses the methods and implementation from previous work [21, 39–41]. The best program generated from incrementalizing differently converted aggregate queries is currently selected manually.

We have programmed a variety of well-known distributed algorithms using DistAlgo, applied our analyses and optimizations, and generated executable Python code for all of them. We discuss our experiments and experiences with twelve of them, listed in Table 6. DistAlgo has also been used by undergraduate and graduate students to easily implement a variety of distributed algorithms used in dis-

Algorithm	Description
La mutex	Lamport's distributed mutual exclusion [33]
La mutex2	La mutex with optimization in footnote in [33]
RA mutex	Ricart-Agrawala's distributed mutual exclusion [57]
RA token	Ricart-Agrawala's token-based mutual exclusion [58]
SK token	Suzuki-Kasami's token-based mutual exclusion [61]
CR leader	Chang-Robert's leader election [11]
HS leader	Hirschberg-Sinclair's leader election [27]
2P commit	Two-phase commit [24]
DS crash	Dolev-Strong's consensus under crash failures [15]
La Paxos	Lamport's Paxos for distributed consensus [34, 35]
CL Paxos	Castro-Liskov's Paxos under Byzantine failures [10]
vR Paxos	van Renesse's pseudocode for multi-Paxos [66]

Table 6. Well-known distributed algorithms.

Algorithm	DistAlgo	PlusCal	IOA	Overlog	Bloom
La mutex	32	90 [46]	64 [43]		
La mutex2	33				
RA mutex	35				
RA token	43				
SK token	42				
CR leader	30		41 [28]		
HS leader	56				
2P commit	44	68 [65]			85 [64]
DS crash	22				
La Paxos	43	83 [45]	145 [29]	230 [49]	157 [51]
CL Paxos	63	166 [45]			
vR Paxos	156				

Table 7. Sizes of programs in different languages, with citations.

tributed file sharing and other services, including Kademlia, Tapestry, Pastry, and Chord, and parts of HDFS and Upright.

All reported running times are obtained with all processes running on one Xen virtual machine with 6GB of main memory on an Intel Core-i7 2600K CPU with 16GB of main memory, running a Linux 3.2.0 kernel. Unless stated otherwise, experiments use Python 3.2.2, reported results are averages over 10 runs, and program sizes are numbers of lines excluding comments and empty lines.

Programming distributed algorithms. We compared distributed algorithms expressed in DistAlgo with distributed algorithms expressed in very different programming and specification languages. We found that DistAlgo programs are generally much easier to read and write. It took very little time to actually write them, some just a few minutes before generated code ran as intended, but significant effort was spent trying to understand the algorithms from papers and textbooks, some taking days and weeks. Being able to express synchronization conditions using high-level quantifications and apply incrementalization also allowed us to uncover errors in our initial DistAlgo programs that subconsciously used extensive message handlers to do ad hoc incremental updates. It also helped us discover improvements to some of the algorithms, for correctness and for efficiency [42], such as the simplifications that led to the algorithm in Figure 4.

Directly quantifying the ease of programming and clarity of programs is hard, so we use code size as an indirect measure, as is common in programming practice. Table 7 lists the sizes of DistAlgo programs that express these algorithms, and sizes of programs written by other people in other languages, PlusCal [36], IOA [30, 43], Overlog [3], and Bloom [8], that also express these algorithms. DistAlgo programs are consistently small, expressing the algorithms almost exactly like the pseudocode descriptions except with a precise meaning for execution.

We also compared Lamport's distributed mutual exclusion algorithm written in C, Java, Python, Erlang, PlusCal, and DistAlgo, as summarized in Table 8. These programs or specifications vary in the mechanisms used for processes and communications, and their sizes. The first four were developed by ourselves before we started implementing DistAlgo and are our best efforts to use each language in the best way for implementing this algorithm. The PlusCal version is from [46]. The C and Java programs required much more effort than the Python and Erlang programs, which required much more effort than the DistAlgo program. For comparison of program sizes, we formatted our programs according to the suggested styles of the languages; for C, the K&R style is used. Our experience confirmed that, as higher-level language features are used, programming effort and program size decrease.

Language	Distributed programming features used	Total	Clean
C	TCP socket library	358	272
Java	TCP socket library	281	216
Python	multiprocessing package	165	122
Erlang	built-in message passing	177	99
PlusCal	single process simulation using array	134	90
DistAlgo	built-in high-level synchronization	48	32

Table 8. Main distributed programming features used and program sizes (total number of lines, and number of lines without comments and empty lines) for Lamport's distributed mutual exclusion algorithm.

Compilation and optimization. We describe compilation and optimization times, generated program sizes, and performance of generated implementations. We do not discuss implementation of automatic incrementalization [21, 39–41], because it is prior work. We implemented an interface between our DistAlgo compiler and the incrementalizer InvTS [21, 41], so all the examples can be automatically compiled and optimized except for some remaining trivial manual steps and selection of the best program as mentioned above. It is not easy to integrate our DistAlgo compiler and the incrementalizer InvTS, because the incrementalizer uses

Algorithm	Compilation time (ms)	DistAlgo size	Compiled size	Incrementalized size
La mutex	13.3	32	1395	1424
La mutex2	15.3	33	1402	1433
RA mutex	12.3	35	1395	1395
RA token	12.9	43	1402	1402
SK token	16.5	42	1405	1407
CR leader	10.7	30	1395	1395
HS leader	18.7	56	1415	1415
2P commit	21.4	44	1432	1437
DS crash	10.5	22	1399	1414
La Paxos	20.7	43	1428	1498
CL Paxos	32.3	63	1480	1530
vR Paxos	43.4	160	1555	1606

Table 9. Compilation time and sizes of generated programs after compilation and after incrementalization.

Python 2.5, which is incompatible with Python 3, used by our compiler.

Table 9 shows the compilation time and the sizes of the DistAlgo programs, generated Python programs after compilation, and generated Python programs after incrementalization. Compilation time does not include incrementalization time, which was well under 30 seconds for all of our programs. The generated programs include 1300 lines of fixed library code. For some of the algorithms, the given original description and pseudocode already contain all incremental updates, so applying incrementalization to the generated program from compilation does not change the program. These algorithms and some of the other algorithms could be written at a higher level, so the synchronization conditions can be expressed more directly and thus easier to understand and verify. Incremental updates for efficient implementations can then be generated by incrementalization; higher-level programs can also allow more efficient incremental programs to be generated.

Figures 5–7 compare the time and space performance of generated implementations, for original programs and incrementalized programs. A process's memory usage is measured after the process has completed all of its work, so the sequences `sent` and `received` no longer grow. The reported memory usage is the sum of the raw sizes of all data structures created by the generated Python code, measured using Pympler.[1] Figure 5 shows the running time and memory usage of Lamport's distributed mutual exclusion algorithm; the running time is the CPU time for each process to complete a call to `cs(task)`, including time spent handling messages from other processes, averaged over processes and over runs of 30 calls each. Figure 6 shows the running time and memory usage of Castro-Liskov's Byzantine Paxos algorithm; the running time is the CPU time for a proposer to succeed, averaged over proposers, from when it first proposes

[1] `http://packages.python.org/Pympler/`

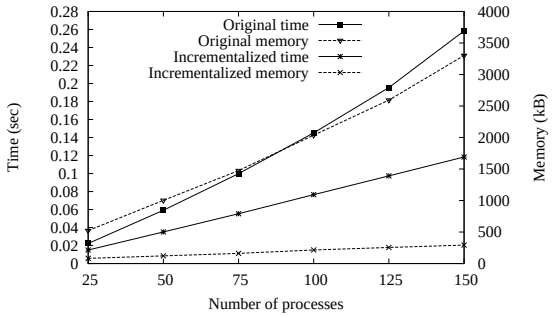

Figure 5. Running time and memory usage of Lamport's distributed mutual exclusion.

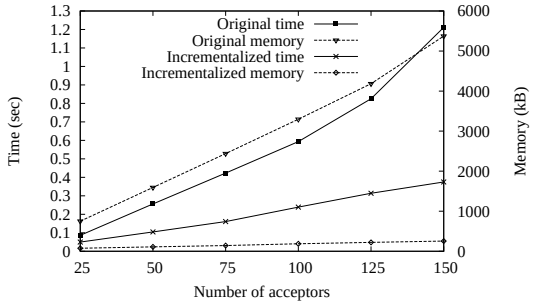

Figure 6. Running time and memory usage of Castro-Liskov's Byzantine Paxos.

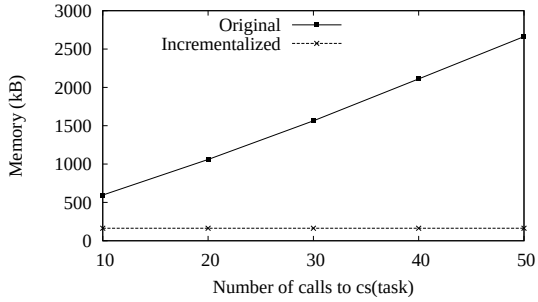

Figure 7. Memory usage of Lamport's distributed mutual exclusion.

until it learns that one of its proposals has succeeded, with the indicated number of acceptor processes and 10 proposer processes. When an acceptor accepts a proposal, it notifies only the proposer. The fault tolerance parameter f is set to $\lfloor (N-1)/3 \rfloor$, where N is the number of acceptors. Figure 7 shows the average per-process memory usage of Lamport's distributed mutual exclusion algorithm with 75 processes, as a function of the number of calls to `cs(task)`. Use of the sequences `received` and `sent` causes memory usage of the un-incrementalized program to grow linearly with the number of calls, while the memory usage of the incrementalized program remains constant. We can see that incrementalization improves the time and space performance of generated implementations asymptotically.

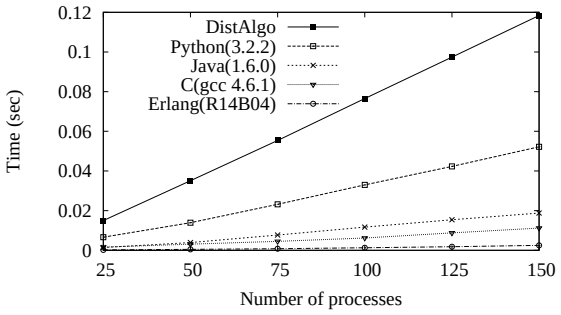

Figure 8. Running time of programs for Lamport's distributed mutual exclusion written in different languages.

Figure 8 compares the efficiency of our programs for Lamport's distributed mutual exclusion algorithm written in different languages. For C, Java, and Erlang, we used GCC 4.6.1, JDK 1.6.0, and R14B04, respectively. The figure shows the CPU time for each process to do a round of requesting and releasing the CS, averaged over processes and over runs of 30 rounds each. We can see that Erlang is the fastest; we think it is partly because Erlang processes are lighter weight. The Python code generated from DistAlgo takes 3 times as long as the manually written Python; it is because of the overhead in the generated code for pattern matching against messages received and for method invocation for message handling at yield points.

This overhead is not intrinsic and could be optimized away so that the generated implementations can be as fast as manually written Python programs. Python is generally slower than Erlang, C, and Java, but is nevertheless widely used for distributed applications.

7. Related work

A wide spectrum of languages and notations have been used to describe distributed algorithms, e.g., [4, 19, 32, 36, 43, 55, 56, 63]. At one end, pseudocode with English is used, e.g., [32], which well gives a high-level flow of the algorithms, but lacks the details and precision needed for a complete understanding. At the other end, state machine based specification languages are used, e.g., I/O automata [30, 43], which is completely precise, but uses low-level control flows that make it harder to write and understand the algorithms. There are also many notations in between these extremes, some being much more precise or completely precise while also giving a high-level control flow, e.g., Raynal's pseudocode [55, 56] and Lamport's PlusCal [36]. However, all of these languages and notations still lack concepts and mechanisms for building real distributed applications, and most of the languages are not executable at all.

Many programming languages support programming of distributed algorithms and applications. Most support distributed programming through messaging libraries, ranging from relatively simple socket libraries to complex libraries such as MPI [47]. Many support Remote Procedure Call (RPC) or Remote Method Invocation (RMI), which allows a process to call a subroutine in another process without the programmer coding the details for this. Some programming languages, such as Erlang [16], based on the actor model [2], have support for message passing and process management built into the language. They all lack constructs for expressing control flows and complex synchronization conditions at a much higher level, because these high-level constructs are extremely difficult to implement efficiently. DistAlgo's construct for declaratively and precisely specifying yield points for message handlers is a new feature that we have not seen in other languages.

There has been much work on generating executable implementations from formal specifications, e.g., from process algebras [26], I/O automata [20], Unity [23], and Seuss [31], as well as from more recently proposed high-level languages for distributed algorithms, e.g., Datalog-based languages Overlog [3] and Bloom [8], and a logic-based language EventML [9, 14]. Compilation of DistAlgo to executable implementations is easy because it is designed to be so and is given an operational semantics. High-level queries and quantifications used for synchronization conditions can be compiled into loops straightforwardly, but they may be extremely inefficient. None of these prior works study powerful optimizations of quantifications. Efficiency concern is a main reason that similar high-level language constructs, whether for queries or assertions, are rarely used, if supported at all, in widely used languages.

Incrementalization has been studied extensively, e.g., [54], both done systematically based on languages, and routinely applied in ad hoc fashions to specific problems. However, all systematic incrementalization methods based on languages have been for centralized sequential programs, e.g., for set languages [25, 40, 50], recursive functions [1, 37, 53], logic rules [38, 60], and object-oriented languages [39, 48, 59]. This work is the first to extend incrementalization to distributed programs, where all sending and receiving of messages are systematically transformed into updates to message history sequences. This allows the large body of previous work on incrementalization, especially on sets and sequences, to be used for optimizing distributed programs.

Quantifications are the centerpiece of first-order logic, and are dominantly used in writing synchronization conditions and assertions in specifications, but there are few results on generating efficient implementations of them. In databases, despite extensive work on efficient implementation of high-level queries, efficient implementation of universal quantification has only been studied in limited scope or for extremely restricted query forms, e.g., [5–7, 12]. In logic programming, implementations of universal quantification are all based on variants of brute-force Lloyd-Topor transformations, e.g., [18, 52]; even state-of-the-art logic programming systems [62, 69] do not support universal quantification. Our method is the first general and system-

atic method for incrementalizing arbitrary quantifications. Although they are much more challenging to optimize than set queries, our method combines a set of general transformations to transform them into aggregate queries that can be most efficiently incrementalized using the best previous methods.

To conclude, this paper presents a powerful language and method for programming and optimizing distributed algorithms. There are many directions for future work, from formal verification on the theoretical side, to generating code in lower-level languages on the practical side, with many additional analyses and optimizations in between.

Acknowledgments

We are grateful to the following people for their helpful comments and discussions: Ken Birman, Andrew Black, Jon Brandvein, Wei Chen, Ernie Cohen, John Field, Georges Gonthier, Leslie Lamport, Nancy Lynch, Lambert Meertens, Stephan Merz, Don Porter, Michel Raynal, John Reppy, Gun Sirer, Doug Smith, Robbert van Renesse, and anonymous reviewers.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems*, 28(6):990–1034, 2006.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] P. Alvaro, T. Condie, N. Conway, J. Hellerstein, and R. Sears. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 43(4):25–30, 2010.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2nd edition, 2004.

[5] A. Badia. Question answering and database querying: Bridging the gap with generalized quantification. *Journal of Applied Logic*, 5(1):3–19, 2007.

[6] A. Badia, M. Gyssens, and D. Van Gucht. Query languages with generalized quantifiers. In R. Ramakrishnan, editor, *Applications of Logic in Databases*. Kluwer Academic, 1994.

[7] A. Badia, B. Debes, and B. Cao. An implementation of a query language with generalized quantifiers. In *Proceedings of the 27th International Conference on Conceptual Modeling*. Springer, 2008.

[8] Berkeley Orders of Magnitude. Bloom Programming Language. `http://www.bloom-lang.net/`.

[9] M. Bickford. Component specification using event classes. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, pages 140–155. Springer, 2009.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20:398–461, 2002.

[11] E. J. H. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[12] J. Claußen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 286–295. Morgan Kaufman, 1997.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[14] CRASH Project. EventML. `http://www.nuprl.org/software/#WhatisEventML`, Last dated March 2012.

[15] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[16] Erlang Programming Language. Erlang Programming Language. `http://www.erlang.org/`.

[17] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[18] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program transformation for development, verification, and synthesis of programs. *Intelligenza Artificiale*, 5(1):119–125, 2011.

[19] V. K. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[20] C. Georgiou, N. A. Lynch, P. Mavrommatis, and J. A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. *International Journal on Software Tools for Technology Transfer*, 11(2):153–171, 2009.

[21] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, pages 27–42. ACM, 2010.

[22] D. Goyal and R. Paige. The formal reconstruction and improvement of the linear time fragment of Willard's relational calculus subset. In *Algorithmic Languages and Calculi*, pages 382–414. Chapman & Hall, 1997.

[23] A. Granicz, D. M. Zimmerman, and J. Hickey. Rewriting UNITY. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, pages 138–147, 2003.

[24] J. Gray. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481, 1978.

[25] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, 1993.

[26] D. Hansel, R. Cleaveland, and S. A. Smolka. Distributed prototyping from validated specifications. *Journal of Systems and Software*, 70(3):275–298, 2004.

[27] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[28] I/O Automata Description of Leader Election Algorithm. `http://groups.csail.mit.edu/tds/ioa/leader.html`.

[29] IOA toolkit extended version. `http://groups.csail.mit.edu/tds/ioa/distributions/IOA_Toolkit-tools.tar.gz`. The Paxos code is under Examples/Paxos.

[30] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan Claypool Publishers, 2nd edition, 2010.

[31] I. H. Krüger. An experiment in compiler design for a concurrent object-based programming language. Master's thesis, The University of Texas at Austin, 1996.

[32] A. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[34] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[35] L. Lamport. Paxos made simple. *SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[36] L. Lamport. The PlusCal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, pages 36–60, 2009.

[37] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1-2):37–62, 2003.

[38] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):1–38, 2009.

[39] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

[40] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–120, 2006.

[41] Y. A. Liu, M. Gorbovitski, and S. D. Stoller. A language and framework for invariant-driven transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 55–64, 2009.

[42] Y. A. Liu, S. D. Stoller, and B. Lin. High-level executable specifications of distributed algorithms. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2012. To appear.

[43] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[44] F. Mattern. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms*, pages 120–131, 1989.

[45] Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm. Mechanically checked safety proof of a Byzantine Paxos algorithm. `http://research.microsoft.com/en-us/um/people/lamport/tla/byzpaxos.html`. Last modified 1 September 2011.

[46] S. Merz. Lamport's algorithm, 2010. Email.

[47] Message Passing Interface (MPI) Forum. Message Passing Interface (MPI) Forum. `http://www.mpi-forum.org/`.

[48] H. Nakamura. Incremental computation of complex object queries. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 156–165, 2001.

[49] P2. `https://svn.declarativity.net/overlog-paxos/src/olg/core/`.

[50] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[51] Paxos in Bud Sandbox. `https://github.com/bloom-lang/bud-sandbox/tree/master/paxos`.

[52] V. Petukhin. Programs with universally quantified embedded implications. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 310–324, 1997.

[53] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[54] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[55] M. Raynal. *Distributed Algorithms and Protocols*. Wiley, 1988.

[56] M. Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool, 2010.

[57] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[58] G. Ricart and A. K. Agrawala. Author's response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol. *Communications of the ACM*, 26(2):147–148, 1983.

[59] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 55–66, 2008.

[60] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *Proceedings of the 19th International Conference on Logic Programming*, pages 392–406, 2003.

[61] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4): 344–349, 1985.

[62] T. Swift, D. S. Warren, et al. *The XSB System Version 3.3*. Sourceforge.Net, 2011. `http://xsb.sourceforge.net/`.

[63] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.

[64] Two-phase commit in Bud Sandbox. `https://github.com/bloom-lang/bud-sandbox/blob/master/2pc/2pc.rb`.

[65] Two-Phase Commit in PlusCal. `http://research.microsoft.com/en-us/um/people/lamport/tla/two-phase.html`.

[66] R. van Renesse. Paxos made moderately complex, October 11, 2011. An online version is at `www.cs.cornell.edu/courses/CS7412/2011sp/paxos.pdf`.

[67] D. E. Willard. Efficient processing of relational calculus expressions using range query theory. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 164–175, 1984.

[68] D. E. Willard. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65:295–331, 2002.

[69] G. Yang, M. Kifer, H. Wan, and C. Zhao. *Flora-2: User's Manual Version 0.95*. Sourceforge.Net and Stony Brook University, 2008. `http://flora.sourceforge.net/`.