Parallel Prefix Polymorphism Permits Parallelization, Presentation & Proof

Jiahao Chen
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence
Laboratory
77 Massachusetts Avenue
Cambridge, Massachusetts 02139, USA
jiahao@mit.edu

Alan Edelman
Massachusetts Institute of Technology
Department of Mathematics, and Computer
Science and Artificial Intelligence Laboratory
77 Massachusetts Avenue
Cambridge, Massachusetts 02139, USA
edelman@mit.edu

ABSTRACT

Polymorphism in programming languages enables code reuse. Here, we show that polymorphism has broad applicability far beyond computations for technical computing: **parallelism** in distributed computing, **presentation** of visualizations of runtime data flow, and **proofs** for formal verification of correctness. The ability to reuse a single codebase for all these purposes provides new ways to understand and verify parallel programs.

Categories and Subject Descriptors

D.1.3 [**Concurrent programming**]: Distributed programming; D.3.2 [**Programming languages**]: Very high-level languages; G.1.0 [**General numerical analysis**]: Parallel algorithms

General Terms

Algorithms

Keywords

Julia, prefix sum, scan, parallel prefix, polymorphism, instrumentation, visualization

1. INTRODUCTION

Abstractions are the lifeblood of computer science [2]. Separating higher level concepts from lower level implementation details allows code to be more composable and reusable, and code using appropriate abstractions is simpler to understand and maintain. However, the desire for high level abstractions often seems inimical to writing high-performance code; oftentimes performance can only result from code that takes advantage of low-level implementation details and hence break abstractions. Thus areas like technical computing and

parallel programming, where performance is a key consideration, exhibit a general dearth of abstraction and suffer the consequent complexities in large, inscrutable and nearly unmaintainable codebases. Such challenges are particularly acute for parallel programs, which are written primarily for performance reasons. However, parallel programs are notoriously prone to subtle, nondeterministic bugs arising from race conditions, and they are difficult to verify for correctness.

Programs for technical computing tend to sacrifice abstraction for performance, but pay the cost in expressiveness. In contrast, the mathematical structure of technical computations is inherently amenable to abstraction, particularly polymorphism. Just think of the myriad ways different mathematical objects can be multiplied together: one can multiply two numbers together, or two matrices together, or a number and a matrix. The notion of multiplication itself can also be extended to dot products, cross products, tensor products, wedge products, outer products, and smash products, to just name a few. All these operators share a common metaphor and it is natural to implement these products with polymorphic code that can express the mathematical commonality.

In this paper, we argue that high level languages like Julia provide a suitable framework for expressing high level abstractions that allow for extremely powerful code reuse, while affording a reasonable level of performance. In particular, we explore how Julia's generic function system and type system are suitable for expressing polymorphism at the operator level, and show how polymorphism can be used not just to encompass different kinds of computations, but can be applied to parallelize code, enable visualizations of code implementations, and even verify correctness of code.

1.1 Multimethods in Julia

In this paper, we use the Julia language[1], a very high level dynamic language designed specifically for technical computing [5]. Julia offers language constructs that support different mechanisms of polymorphism, which programming language theorists call universal polymorphism and *ad hoc* polymorphism, or overloading [50]. In this paper, we focus only on Julia's overloading mechanism provided by multi-

---

[1]Julia is MIT-licensed open source software and can be downloaded freely from `julialang.org`. We use v0.3.1 in this paper.

methods.

Ad hoc polymorphism, or overloading, is a language construct that naturally expresses the polymorphism inherent in the mathematical structure of technical computing [4]. Consider the `*` operator which represents multiplication: the product can be taken between two integers, two floating-point numbers, a scalar and a vector, or a matrix and a matrix, just to list a few typical possibilities. All these different semantics can be represented by with the same syntax, namely an expression of the form `a*b`. All programming languages resolve the ambiguity by considering the types of the arguments `a` and `b`, which is formally equivalent to specifying the domains of the operands [44].

In practice, programming languages vary greatly in how they allow users to reason about types. Some languages, like Julia, offer the ability to define multimethods, where a single generic function like `*` can be defined with at more than one method, each with a different type signature: in Julia notation, `*(a::Number, b::Number)` defines a method for scalar multiplication, whereas `*(a::Matrix, b::Vector)` defines a method for matrix-vector products, and so on. Closely related is the support for multiple dispatch, where the method chosen to match an expression like `a*b` can depend on the type of more than one argument. In this respect, multimethods differs greatly from more conventional languages that provide class-based objects; dispatch only occurs on the first argument, which is the type of the class.

In this paper, we demonstrate how multimethod polymorphism is far more general than just dispatching on computational kernels. We study one specific algorithm, namely scan, and show how the same exact code written in Julia for serial computation can be composed with appropriately overloaded operators to generate parallel code, visualizations, and also proofs of correctness.

1.2 The scan algorithm

The basic problem of interest is to compute from some initial data `y` the partial sums `z` such that:

```
z[1] = y[1]
z[2] = y[1] + y[2]
z[3] = y[1] + y[2] + y[3]
...
```

One way to compute this sum efficiently is to note the prefix property, i.e. that the kth partial sum depends only on the $(k-1)$th partial sum and the kth element:

```
z[1] = y[1]
z[2] = z[1] + y[2]
z[3] = z[2] + y[3]
...
```

which leads to the simple algorithm:

```
function prefix_serial!(y, +)
    for i=2:length(y)
        y[i] = y[i-1] + y[i]
    end
    y
end
```

The cumulative sum problem generalizes to any associative operator; in this Julia function, the argument `+` specifies the operator of interest, allowing the same code to be

Application	Operator
	Addition
Poisson random variates [35]	sequence lengths
Minimal coverings [40]	joining 2D regions
Stream reduction [26]	counting records
	Maximization
Line of sight [7]	height
String alignment [24, 14]	substring length
	Multiplication
Binary addition [47]	Boolean matrices
Polynomial interpolation [19]	scalars
Sorting [24, 6]	permutations
Tridiagonal equations [36]	matrices
	Function composition
Finite state automata [34, 24]	transition functions

Table 1: **Representative applications of the scan algorithm, employing four basic types of operations: addition, maximization, multiplication, and function composition.**

reused for other operators like multiplication (`*`), maximization (`max`) [46], or even string concatenation[2]. The `!` suffix is a Julia convention denoting that the function mutates at least one of its arguments; in this case, the cumulative sums are computed in-place on `y`.

The general problem is called the prefix sum [6, 8] or scan [28, 29]. Nominally, it appears that the data has to be scanned in one sweep from first to last and is a naturally serial process. However, the insight behind parallel prefix algorithms [6, 9, 32, 33, 34, 47] is that associativity allows the operations to regrouped in different ways which can expose potential for concurrent execution, which can be interpreted as generic divide-and-conquer strategies for recursive computation [48].

In its general form, scan algorithms can be computed as a higher-order function which takes as input some associative operator. Table 1 shows a representative list of applications of parallel prefix, showing the diversity of applications and associative operators associated with those applications [7, 8]. Scan is therefore a prime example of an algorithm that can exploit polymorphism for genericity.

1.3 The Brent–Kung form of parallel prefix

In this paper, we focus on the Brent–Kung form [9] of parallel prefix, where the computation is organized into two trees. For simplicity, we present first the special case of parallel prefix for $n = 8$ data points.

```
function prefix8!(y, +)
    length(y)==8 || error("length 8 only")
    for i in [2,4,6,8] y[i] = y[i-1] + y[i] end
    for i in [  4,  8] y[i] = y[i-2] + y[i] end
```

[2]As written, the `prefix_serial!` function assumes, but does not check, that the function passed to it is associative. If necessary, checks of the form

```
@assert (y[1]+y[2])+y[3] == y[1]+(y[2]+y[3])
```

can be included, but for simplicity of presentation, we omit such checks from the code presented in this paper. We also neglect concerns relating to *approximate* associativity, such as roundoff errors in floating-point addition or multiplication [36].

```
    for i in [      8] y[i] = y[i-4] + y[i] end
    for i in [   6  ] y[i] = y[i-2] + y[i] end
    for i in [ 3,5,7 ] y[i] = y[i-1] + y[i] end
    y
end
```

Figure 1 illustrates the difference between the number and order of operations in `prefix_serial!` and `prefix8!`. Each vertical line represents a processor `i` operating on the data `y[i]`. Each operation of the form `y[i] = y[j] + y[i]` is represented by a gate with inputs on lines `i` and `j` and a single output on line `i`. The main idea is that even though it takes more operations to organize the computation in the double tree form of `prefix8!`, it is possible to execute each stage of the computation tree concurrently, and parallel speedup can be achieved if the depth of the resulting tree is shorter than the depth of the tree for the serial algorithm. Nevertheless, at this point we have not actually computed anything in parallel, merely organized the computation in a way that would *allow* for concurrent execution. Running the code as is on an `Array` object would run the operations sequentially, from left to right, then top to bottom of the computation tree.

To conclude our exposition of the scan problem, we present the `prefix!` function that solves the general case of n data points. While the indices are somewhat less clear than when explicitly written out in `prefix8!`, the `prefix!` function nonetheless preserves the double tree structure.

```
function prefix!(y, +)
    l=length(y)
    k=iceil(log2(l))
    #The "reduce" tree
    for j=1:k, i=2^j:2^j:min(l, 2^k)
        y[i] = y[i-2^(j-1)] + y[i]
    end
    #The "broadcast" tree
    for j=(k-1):-1:1, i=3*2^(j-1):2^j:min(l, 2^k)
        y[i] = y[i-2^(j-1)] + y[i]
    end
    y
end
```

Again, at this point we have only written serial code that introduces more computations than the naive algorithm `prefix _serial!`. However, we will argue in Section 2.1 that the exact same code in `prefix!` can be reused for parallel execution which can achieve speedup over `prefix_serial!`.

2. OPERATORS FOR DISTRIBUTED COMPUTATIONS

In this section we show how the prefix algorithm we wrote above can be run in a distributed setting without modification. The key is to make use of overloading using the multimethod dispatch feature of Julia.

Julia provides native support for multiprocess distributed computing based on one-sided message passing. The basic functionality is provided by the `remotecall` function, which initiates a nonblocking remote function call and returns an explicit future [21] (a remote pointer of type `RemoteRef`) whose value is retrieved by the `fetch` function, which is a blocking operation. Julia also provides more convenient syntax for `remotecall` with the `@spawn` and `@spawnat` macros,

```
render(prefix_serial!(AccessArray(8),+))
```
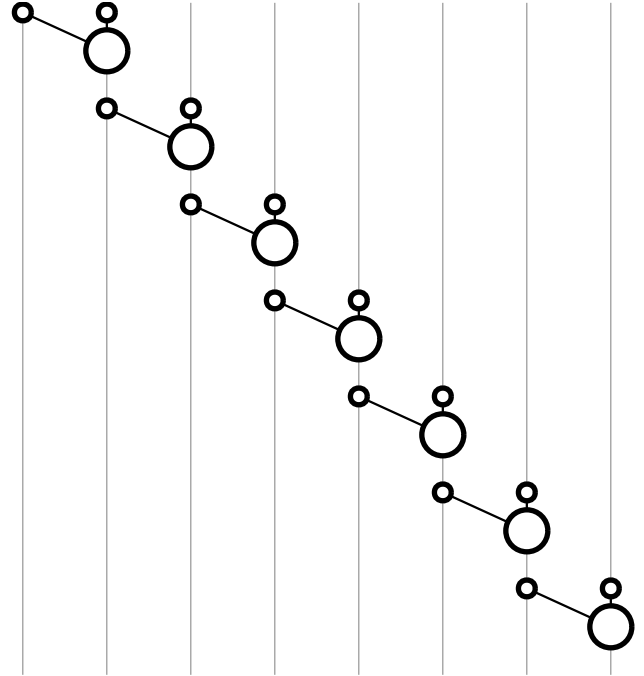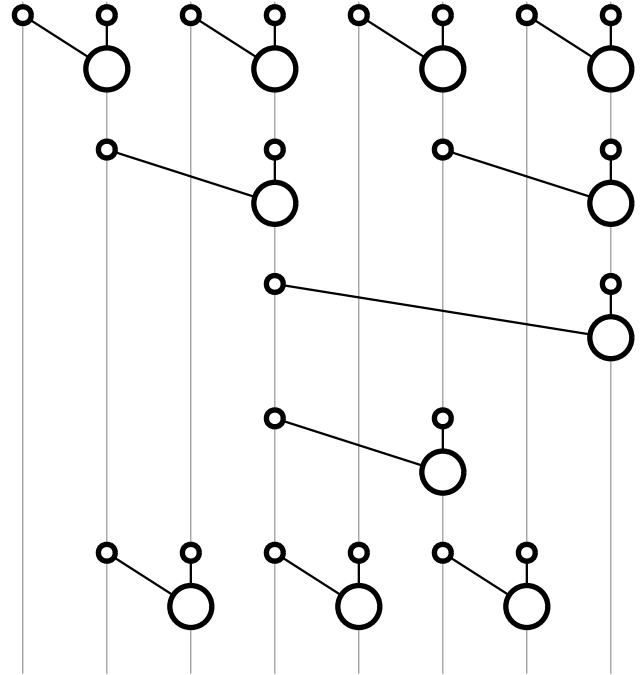


```
render(prefix8!(AccessArray(8),+))
```


Figure 1: Above: operation order generated from the left-associative algorithm `prefix_serial!`. Below: operation order generated from the tree algorithm `prefix8!`. The figures were generated directly from the same kernels used for computation in Section 2.1. The code listing for the `render` function is given in Section A. This figure was rendered in Compose, a Julia package for declarative vector graphics [31].

which automatically rewrite Julia expressions into `remotecall` function calls.

We can use Julia's multiple dispatch feature to define associative operators which act on remote data rather than local data. Julia's generic function system allows new methods which act on remote data to be defined for functions like `+` and `*`, which are simply functions for which the parser supports infix notation. In effect, we can overload addition and multiplication (or in general any binary associative function) transparently to work on remote data.

For example, we can run the following code:

```
#Start a Julia process on every available core
#addprocs(n) adds n processors
#Sys.CPU_CORES is the total number of available
#CPU cores
#nprocs() returns the total number of Julia
#processes attached to the current master
#(including itself)
addprocs(max(0, Sys.CPU_CORES-nprocs()))

import Base.* #Extend existing generic function

#Define elementary operations on remote data
*(r1::RemoteRef,r2::RemoteRef)=
    @spawnat r2.where fetch(r1)*fetch(r2)
```

This one method defines multiplication on remote data by `fetch`ing the remote data from the process containing the data of `r1`, copying the data of `fetch(r1)` to the memory space of the process with id `r2.where`, which already stores the data of `r2`. The process `r2.where` now contains local copies of both operands. Assuming that the local data are of type `T`, the Julia code then invokes another round of method dispatch based on the method signature `*(::T, ::T)`. In this way, any data type `T` that supports multiplication will now also support remote multiplication, regardless of whether the data are scalar numbers, $N \times N$ matrices, or something else entirely.

The main point of this paper is that the very same function `prefix!` which was executed in serial in previous sections will now run in parallel, simply by passing to it an associative operator over remote data rather than local data. Julia's multimethods and multiple dispatch semantics allow operations on remote data to share the same syntax as their corresponding operations on local data, thus removing any syntactic difference between remote and local operations. The new method for `*` defines new behavior specific to `RemoteRef`s, which are Julia's explicit futures. With this new method defined in the current scope, running `prefix!(y, *)` will automatically compute cumulative products on remote data if `y` is an array of `RemoteRef`s. Julia will automatically dispatch on the `*(r1::RemoteRef, r2::RemoteRef)` method within the inner loops of `prefix!` by comparing the types of the data elements of `y` with method signatures defined for `*`.

2.1 Parallel prefix

We now run the `prefix!` function in parallel. The remote operations `*(r1::RemoteRef, r2::RemoteRef)` contain blocking operations implied by `fetch(r1)`, and Julia dynamically schedules all remote operations simultaneously so long as they are not waiting on the result of a `fetch` operation. The scheduling and dependency structure of `prefix!`

thus results in all operations in each stage of the tree being executed simultaneously. Neglecting overhead from communication latency and bandwidth, the total execution time of `prefix!` depends only on the depth of the trees defined by the inner loops of `prefix!` and visualized in Figure 1.

From the indices of each loop in `prefix!` for l data points, the first tree has at least one operation at depth k for $l \geq 2^k$, and therefore the depth of the entire tree is $k = \lfloor \log_2 l \rfloor$. Similarly, the second tree has at least one operation at depth k for $l \geq 3 \cdot 2^{k-1}$, and hence has depth $k = 1 + \lfloor log_2 \frac{l}{3} \rfloor$. Adding these depths and assuming that we distribute one datum per processor, we therefore obtain the theoretical speedup ratio for p processors running `prefix!` over `prefix_serial!` as:

$$r(p) = \frac{p-1}{\lfloor \log_2 p \rfloor + 1 + \lfloor \log_2 \frac{p}{3} \rfloor}. \tag{1}$$

Figure 2 summarizes benchmark timings for a sample problem where we generated p square random matrices with Gaussian entries of size $n = 4096$ and timed how long it took to multiply these matrices together on an 80-core Xeon E7-8850 machine with 1TB of shared memory. We specifically left out the time needed to broadcast the data to the remote processes, so as to focus only on the execution times of the kernels of interest. We also took care to disable the garbage collector. Julia, like many high-level dynamic languages, provides a garbage collector to aid in memory management. Julia v0.3.1 uses a simple stop-the-world, non-moving, precise mark and sweep garbage collector, where deallocation and finalization of garbage objects may not happen immediately after objects become unused[3] [37]. Therefore, it becomes important to factor out the possible effects of stop-the-world garbage collection. We explicitly disabled garbage collection with `gc_disable()` before running each kernel, then re-enabled garbage collection with `gc_enable()` after running each kernel. As an additional precaution, we timed the kernels multiple times and took the minimum time for each kernel so as to reduce fluctuations due to general nondeterministic delays.
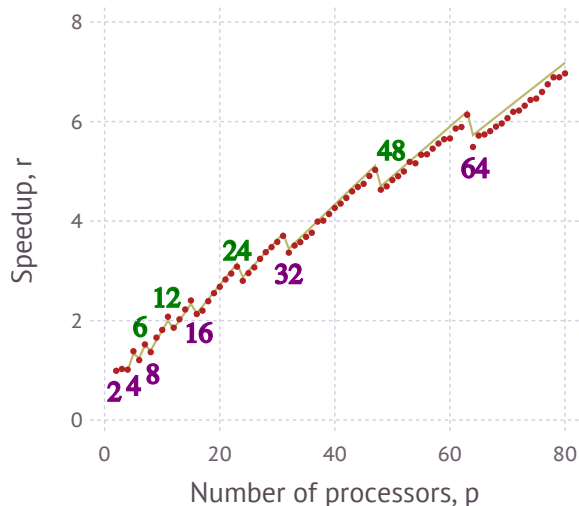
The empirical timings shown in Figure 2 show excellent agreement with the theoretical prediction of Equation 1, with slight deterioration for $p > 40$ cores reflecting the increased communication overhead. The steps in the graph are as predicted by theory, arising from the depth of the computation tree growing by one to accommodate the extra data.

2.2 Other variants of parallel prefix

A wide variety of parallel prefix algorithms exist beyond the Brent–Kung form [6, 20, 32, 33, 34, 41, 47, 43, 45, 53]. All of these variants can also be expressed as generic higher-order functions analogously to `prefix!`; however, not all of them can be written as in-place computations [38]. Nevertheless, the general principle still holds that generic kernels can be written for an arbitrary parallel prefix computation tree, and that these generic kernels can be composed effortlessly in Julia to support exactly the same range of operations as the Brent–Kung form in `prefix!`.

The genericity of `prefix!` is also very useful for imple-

---

[3]The code for Julia's garbage collector may be found at `https://github.com/JuliaLang/julia/blob/275afc8b74b9c6ea5d34aefb8085525ff5dfc239/src/gc.c`

Figure 2: Weak scaling of the prefix sum kernels. Speedup ratios are the timings for `prefix!` over `prefix_serial!`. Plotted as a solid line is the theoretical speedup ratio $r(p)$ of Equation 1. This figure was rendered in Gadfly, a Julia package for native plotting and visualization [30].

menting so-called meta-strategies for parallel prefix such as scan-then-fan [38, 54]. The scan-then-fan variant of parallel prefix is a divide-and-conquer algorithm, where parts of the scan are chunked and computed serially on each process, after which each chunk is offset by the correct result by the value of the last element in its preceding chunk. The offsets can be propagated simply by running `prefix!` on the distributed array of chunks, with the operator

```
+(a, b) = a[end] .+ b
```

This variant of parallel prefix avoids the extra work incurred by generating a computation tree that is wider than the total number of available processes, and thus is more efficient to compute.

3. OPERATOR-LEVEL INSTRUMENTATION

Earlier, we showed in Figure 1 visualizations demonstrating the double tree structure of the Brent–Kung parallel prefix algorithm and also the cascading or rippling structure of the serial scan. These figures were generated programmatically from the exact same kernels `prefix!` and `prefix_serial!` used to perform the computations.

Many visualizations of algorithms are bespoke; the representations are completely decoupled from executable implementations. Alternatively, one may envision generating visualizations of algorithms directly from code implementations. Visualizations of algorithms can be generated by static analysis: feed the compute kernel into another program as data to compute the internal data flow. The static approach, however, is tantamount to reimplementing the compiler to generate the correct execution trace, from which the data flow can be inferred. Instead, one can employ dynamic analysis, instrumenting the program much like a debugger. Conventional debuggers either work on modified code with explicit instrumentation hooks embedded into the

original kernel, or run the program in a special virtual machine with instrumentation hooks built into the low-level machine architecture. In these dynamic analyses, the execution trace is reconstructed from the global machine state, and again the data flow is inferred from the execution flow.

In this section, we describe a simple way to generate visualizations programmatically by instrumenting the interface of specific data objects, namely arrays. Instrumentation at this level retains the advantage of composing with unmodified compute kernels, but does not require the sophisticated infrastructure of an instrumented virtual machine, and reuses the static analysis of the original compiler. Furthermore, the instrumentation occurs at the level of individual variables, enabling highly selective traces which are cheaper than conventional approaches which instrument the entire program state. Additionally, the instrumentation measures the data flow directly, rather than inferring it from global execution flow. The resulting visualization provides an individual variable's point of view of what happens over the course of a computation.

Our implementation in Julia takes advantage of genericity in the object model. Unlike most traditional object-oriented paradigms, which focus on data encapsulation [10], the object model in Julia focuses on the interface to objects provided by method calls [39]. Making the behavior primary over the data contents lends more naturally to data abstraction [39, 1], and furthermore admits less conventional object models involving multimethods and multiple dispatch [11].

`Array`s in Julia are containers of a given size (possibly with multiple dimensions) and element type. The basic array interface for Julia provides size and indexing semantics [4]. The basic interface is provided by three functions:

`length(A)` returns the number of elements in the array `A`,

`getindex(A, idx...)` retrieves the element of the array `A` with index `idx`,

`setindex!(A, val, idx...)` puts the value `val` in the array `A` at the index `idx`.

The Julia parser also provides syntax sugar for the latter two operations: code like

```
A[i] = A[j] + A[k]
```

is desugared into code of the form

```
x = getindex(A, j)
y = getindex(A, k)
z = x + y
setindex!(A, z, i)
```

All the operations in the prefix sum kernels presented have array access operations of this form: two `getindex` calls followed by one `setindex!`. Based on this observation, we can write a very simple data type that nominally provides exactly the same interface as an `Array`, but rather than actually storing data elements, merely records the indices accessed by indexing operations.

Here is the entire Julia code that implements `AccessArray`, an abstract array type which instruments its indexing operations:

```julia
import Base: getindex, setindex!, length

type AccessArray
    length :: Int
    read :: Vector
    history :: Vector
    AccessArray(length)=new(length, Any[], Any[])
end

length(A::AccessArray)=A.length

function getindex(A::AccessArray, i)
    push!(A.read, i)
    nothing
end

function setindex!(A::AccessArray, x, i)
    push!(A.history, (A.read, Any[i]))
    A.read = Any[]
end

#Dummy associative operator
+(a::Void, b::Void) = nothing
```

The `AccessArray` type contains three fields:

length the effective length of the array,

read the history of indices accessed by `getindex` that have yet to be followed by a `setindex!` call, and

history the history of indices accessed by (`getindex`, `setindex!`) calls.

The `Any[]` construct defines an empty array which is explicitly typed to allow elements of any type. `getindex` is defined to always return the value `nothing`[4], while recording the index `i` into `A.read`. `setindex!` records the index `i`, pairs it with the current value of `A.read`, and stores the pair into `A.history`.

As implemented, the `AccessArray` type can only accurately trace code where a `setindex!` call uses all the data from previous `getindex` calls. Furthermore, it does not handle cases where execution flow depends on the values of the array elements. Nevertheless, the `AccessArray` type is sufficiently powerful to record transactions relevant for prefix sums, and can be extended to more general execution flow patterns if necessary by wrapping actual data elements. The Appendix further defines the `render` function used to construct vector graphical objects from the instrumentation data stored in an `AccessArray`, and Figure 1 shows graphical renderings of the access patterns produced by `prefix_serial!` and `prefix!`.

4. OPERATORS FOR FORMAL VERIFICATION

In Section 1.2 we introduced several different kernels to compute scans. But how do we know that these kernels compute the prefix sum correctly? Each of these kernels have exactly the same function signature `(y, +)` representing the data `y` and associative binary operator `+`. It turns out that

---

[4] `nothing` is a value of the special singleton type `Void`, akin to Python's `none` or Haskell's `Nothing`.

the inputs (`y`, `+`) to the scan algorithm turn out to have exactly the algebraic structure of a monoid, if the domain of array elements `y[i]` contains an identity under the operation `+`. The monoidal structure has been used in at least two ways to prove correctness. First, [25] constructed a formal algebra that allows correctness of circuits to be proved by derivation: all circuits which are equivalent to a known correct circuit, up to certain algebraic transformations, will all be correct. However, the algebraic proof of correctness is not constructive and does not lend itself easily to programmatic verification. Second and more recently, [15] proved that the correctness of a kernel can be demonstrated by proving correctness for the interval monoid (Definition 1), which formalizes the notion of indexing the subarrays being accessed over the course of the prefix sum computation. The latter method of proof is easy to verify programmatically.

In this section, we show how polymorphism allows the same Julia code written in previous sections for practical computations to also be used in the formal setting of verifying correctness. For convenience, we quote the definition of the interval monoid:

DEFINITION 1. *[15, Definition 4.3]*
The interval monoid I has the elements

$$\mathbb{S}_I = \{(i_1, i_2) \in \text{Int} \times \text{Int} \mid i_1 \leq i_2\} \cup \{\mathbf{1}_I, \top\} \quad (2)$$

and a binary operator $\oplus_I$ defined by:

$$\mathbf{1}_I \oplus_I x = x \oplus_I \mathbf{1}_I = x \text{ for all } x \in \mathbb{S}_I \quad (3a)$$
$$\top \oplus_I x = x \oplus_I \top = \top \text{ for all } x \in \mathbb{S}_I \quad (3b)$$
$$(i_1, i_2) \oplus_I (i_3, i_4) = \begin{cases} (i_1, i_4) & \text{if } i_2 + 1 = i_3 \\ \top & \text{otherwise.} \end{cases} \quad (3c)$$

The elements $(i, j) \in \mathbb{S}_I$ are abstractions of array indexing operations which produce array slices; they are produced by Julia code like `y[i:j]` where `i:j` is of type `UnitRange` and is a range of unit stride representing the set $\{i, i+1, \ldots, j\}$. The definition of $\oplus_I$ in (3c) formalizes the notion of combining the results from the subarrays `y[i:j]` and `y[j+1:k]` to get the result for the subarray `y[i:k]`. The identity element $\mathbf{1}_I$ formalizes an empty interval, while the annihilator $\top$ encodes noncontiguous ranges, which correspond to partial sums which cannot be represented by slicing with a `UnitRange`.

The key insight of [15] is that correct computations of prefix sums cannot generate noncontiguous elements $\top$, otherwise they would by definition violate the prefixing property `prefix!(y[1:j+1], +) = prefix!(y[1:j], +) + y[j+1]`. From this insight, the authors of [15] derive two correctness results:

1. A function that computes the prefix sum in serial is correct for n data points if and only if that function computes the correct answer for the input $(((1,1), (2,2), \ldots, (n,n)), \oplus_I)$[5] [15, Theorem 4.5]. Furthermore, the correct answer is $((1,1), (1,2), \ldots, (1,n))$, as the kth partial sum involves summing the subarray `y[1:k]`.

---

[5] Our presentation differs from the original only in that Julia arrays are 1-based, in contrast to C/OpenCL arrays studied in the original [15], which are 0-based.

2. A function that computes the prefix sum in parallel is correct if it is free of data races and its equivalent serialization is correct [15, Theorem 5.3].

We can use these results directly to verify the correctness of the Julia code we have written in earlier sections. By construction, the `fetch`es on `RemoteRef`s insert implicit synchronization barriers and thus the parallel code is free of data races. Thus only the serial correctness result needs to be verified explicitly.

Julia allows us to encode the interval monoid directly from the definition, by making use of the rich type system which is exposed to the user. The type system is conventionally used in Julia for type inference and data abstraction; here, we exploit the Curry–Howard correspondence to use the type system as a computational resource that can be used to prove correctness [16, 51, 27]. A convenient feature of Julia's type system is the ability to use abstract data types as singleton values: Julia types are values, and types can be used as singleton values using the `Type{T}` construct. Thus, the domain $\mathbb{S}_I$ can be written as a Julia type `S`, which is the `Union` (type union) of:

- `UnitRange`,

- `Type{Id}`, the identity singleton $\mathbf{1}_I$, and

- `Type{Tee}`, the annihilator singleton $\top$.

With this mapping of the abstract interval monoid domain $\mathbb{S}_I$ onto Julia types, Definition 1 translates directly into the following code:

```
#S is the domain of the interval monoid, 𝕊_I
abstract Tee  #⊤
abstract Id   #1_I
typealias S Union(UnitRange, Type{Tee}, Type{Id})

#+ is the operator of the interval monoid, ⊕_I
+(I::UnitRange, J::UnitRange) = #+_1
    I.stop+1==J.start ? (I.start:J.stop) : Tee
+(::Type{Id}, ::Type{Id}) = Id   #+_2
+(I::S, ::Type{Id}) = I          #+_3
+(::Type{Id}, J::S) = J          #+_4
+(I::S, J::S) = Tee              #+_5
```

Figure 3 summarizes the method dispatch table for the interval monoid, which demonstrates the use of some interesting features of Julia's method dispatcher [5]. First, the Julia method dispatcher chooses the most specific method that matches the type signature of a given set of arguments. Thus even though `+` may appear ambiguous for inputs of type `(::UnitRange, ::UnitRange)`, which matches both $+_1$ and $+_5$ methods, Julia resolves the ambiguity in favor of $+_1$ which has the more specific type signature, since by definition `UnitRange` is a subtype of `S`. Second, Julia uses symmetric multiple dispatch: the positions of the arguments are not used to resolve ambiguities. Hence we need the special-case method $+_2$ with type signature `(::Type{Id}, ::Type{Id})`, which lies in the intersection of the type signatures of $+_3$ and $+_4$. Bearing these rules in mind, it is straightforward to verify that the definition of `+` in the code block above is equivalent to that of $\oplus_I$ in Definition 1. Julia's method dispatch rules allow `+` to be defined in a way

+	J::UnitRange	1	⊤
I::UnitRange	⊤ or UnitRange $+_1$	I $+_3$	⊤
1	J	1 $+_2$	⊤
	$+_4$		
⊤	⊤	⊤	⊤
			$+_5$

Figure 3: Operation table for the interval monoid $(\mathbb{S}_I, \oplus_I)$**, showing the overlapping domains of the various methods** $+_{i \in \{1,\ldots,5\}}$**. The dispatch rules in Julia choose the most specific method defined over the given cell [5].**

that reveals the catch-all nature of ⊤: method $+_5$, which returns `Tee`, is dispatched only when none of the other methods matches the type signature of the given arguments.

Verifying some function `kernel` for the problem size `n` simply reduces to writing the assertion:

```
#Test that kernel is correct for problem size n
@assert kernel([k:k for k=1:n],+)==[1:k for k=1:n]
```

Attempting to verify an incorrect kernel results in at least one ⊤ being produced during the computation, thus poisoning the program state and precipitating type conversion errors of the form

```
`convert` has no method matching
convert(::Type{UnitRange}, ::Type{Tee})
```

which arise from the inability of noncontiguous ranges to be expressed as `UnitRange`s.

The Curry–Howard correspondence allows us to verify correct programs as programs with no type errors; programs with type errors must necessarily be incorrect. Julia thus allows for the same kernels used for computation to be verified directly without any rewriting or translation, simply by exploiting the polymorphism arising from the generic nature of prefix sum kernels, and composing such generic functions with appropriate choices of input data types and associative operators over those types.

5. RELATED WORK

Julia does not provide parallel prefix in the base library; however, several languages like APL [28, 29], Chapel [17], C** [52] and ZPL [13, 18] do provide scan primitives. Other languages can use commonly-used library routines for scans: MPI provides the `MPI_scan` primitive [49, 22], and in MPI-2, also the `MPI_Exscan` primitive for exclusive scan [23]. Intel's Threading Building Blocks (TBB) library provides similar functionality [42]. GPU-specific implementations also exist, such in Haskell's Accelerate library [12] and the Thrust C++ library [3]. Most of these implementations, however,

either lack genericity or express genericity through cumbersome language semantics. APL does not provide generic parallel prefix, and parallelism is provided only by nonstandard implementations. C** only supports user-definable parallel prefix for commutative operations [52]. ZPL allows only for a limited form of overloading in terms of existing operators [18, 17]. MPI allows parallel prefix on any `MPI_Datatype` and `MPI_op`; user-defined operations and data types can be used, but must be explicitly wrapped to do so, and furthermore requires users to reason explicitly about low-level parallelism. The Haskell Accelerate library provides genericity by generating code from user-specified expressions into code that implements parallel prefix, but such code does not benefit from Haskell's static type system since they are not statically analyzable. Thrust and TBB, being written in C++, requires user-specified functions to be specified as C++ functors, which are cumbersome to write as they must be expressed using C++ expression templates. Chapel [17] comes closest to our work in providing user-definable scan operators in convenient syntax; however, operators have to be explicitly defined as classes, which do not support the full polymorphic expressiveness of multimethods.

Our implementation of parallel prefix as a higher-order function in Julia is generic in that it makes use of duck typing: so long as the input operation is associative and the collection of data is indexable, then the prefix sum computation will simply work. The simplicity of this generic approach, however, is by design rather naive and does not account for the complexities in real world implementations, for example possible synchronicity issues produced by higher levels of the broadcast and reduce trees that could result in bus saturation. Nevertheless, we can handle resource constraints by using a more sophisticated dynamic scheduler.

6. CONCLUSIONS AND OUTLOOK

We have demonstrated how polymorphism using multimethods allows us to write generic computation kernels for parallel prefix, which can then be composed seamlessly with appropriate types and associative operators not only for computation for a wide variety of problems, but also for various code introspection tasks such as visualization and formal verification using the interval monoid. Julia's language features lends to very natural and minimalist code that takes full advantage of *ad hoc* polymorphism to provide a single set of computational kernels that can also be visualized directly and verified without the need for retranslation and reimplementation in a more specialized language.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] M. Abadi and L. Cardelli. *A theory of objects.* Number 1 in Monographs in Computer Science. Springer-Verlag, New York, New York, USA, 1996.

[2] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 2 edition, 1996.

[3] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-M. W. Hwu, A. Schuh, N. Mohammad, T. Bradley, F. Jargstorff, P. Micikevicius, R. Tonge, P. Wang, and C. Wooley, editors, *GPU Computing Gems Jade Edition*, Applications of GPU Computing, chapter 26, pages 359–371. Morgan-Kaufmann, Waltham, MA, 2012.

[4] J. Bezanson, J. Chen, S. Karpinski, V. Shah, and A. Edelman. Array operators using multiple dispatch: A design methodology for array implementations in dynamic languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 56–61, New York, NY, USA, 2014. ACM.

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv:1209.5145 [cs.PL]*.

[6] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, 1989.

[7] G. E. Blelloch. *Vector models for data-parallel computing*. Artificial Intelligence. MIT Press, Cambridge, Massachusetts, 1990.

[8] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 35–60. Morgan Kaufmann, San Mateo, California, 1993.

[9] R. P. Brent. A regular layout for parallel adders. *IEEE Trans. Comput.*, C-31(3):260–264, Mar. 1982.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surveys*, 17(4):471–523, 1985.

[11] G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997.

[12] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming - DAMP '11*, pages 3–14, New York, New York, USA, 2011. ACM Press.

[13] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL: a machine independent programming language for parallel computers. *IEEE Trans. Software Eng.*, 26(3):197–211, Mar. 2000.

[14] L. Chi and K. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, chapter 19, pages 230–243. Springer, Berlin, Heidelberg, 1992.

[15] N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*, pages 397–409,

New York, New York, USA, 2014. ACM Press.

[16] H. B. Curry. *Combinatory Logic, Volume I*, volume 22 of *Studies in logic and the foundations of mathematics*. North-Holland Publishing, Amsterdam, 2 edition, 1958.

[17] S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '06*, pages 40–47, New York, New York, USA, 2006. ACM Press.

[18] S. J. Deitz, B. L. Chamberlain, and L. Snyder. High-level language support for user-defined reductions. *The Journal of Supercomputing*, 23(1):23–37, 2002.

[19] Ö. Eğecioğlu, E. Gallopoulos, and Ç. K. Koç. A parallel method for fast and practical high-order Newton interpolation. *BIT Numer. Math.*, 30(2):268–288, June 1990.

[20] Ö. Eğecioğlu and Ç. K. Koç. Parallel prefix computation with few processors. *Comput. Math. App.*, 24(4):77–84, 1992.

[21] D. Friedman and D. Wise. The impact of applicative programming on multiprocessing. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 263–272, Long Beach, CA, 1976. IEEE.

[22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 2 edition, 1999.

[23] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 1999.

[24] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[25] R. Hinze. An algebra of scans. In D. Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, chapter 11, pages 186–210. Springer, Berlin, Heidelberg, 2004.

[26] D. Horn. Stream reduction operations for gpgpu applications. In M. Pharr, editor, *GPU Gems 2*, chapter 36, pages 573–589. Addison-Wesley, 2005.

[27] W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, New York, USA, 1980.

[28] K. E. Iverson. *A programming language*. John Wiley & Sons, New York, NY, USA, 1962.

[29] K. E. Iverson. Operators. *ACM Trans. Program. Lang. Sys.*, 1(2):161–176, Oct. 1979.

[30] D. C. Jones, D. Chudzicki, A. Sengupta, et al. Gadfly.jl v0.3.9 (gadflyjl.org) - native plotting and visualization for Julia.

[31] D. C. Jones, D. Darakananda, K. Fischer, et al. Compose.jl v0.3.9 (composejl.org) - declarative vector graphics for Julia.

[32] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, C-22(8):786–793,

Aug. 1973.

[33] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Trans. Comput.*, C-34(10):965–968, 1985.

[34] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[35] T.-C. Lu, Y.-S. Hou, and R.-J. Chen. A parallel Poisson generator using parallel prefix. *Comput. Math. App.*, 31(3):33–42, Feb. 1996.

[36] R. Mathias. The instability of parallel prefix matrix multiplication. *SIAM J. Sci. Comput.*, 16(4):956–973, July 1995.

[37] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[38] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. Technical Report December 2009, Department of Computer Science, University of Virginia, Charlottesville, Virginia, 2009.

[39] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Sys.*, 10(3):470–502, 1988.

[40] D. Moitra. Finding a minimal cover for binary images: An optimal parallel algorithm. *Algorithmica*, 6(1-6):624–657, June 1991.

[41] Y. Ofman. On the algorithmic complexity of discrete functions. *Sov. Phys. Dokl.*, 7(7):589–591, 1963.

[42] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, volume 2007. O'Reilly, Sebastopol, CA, 2007.

[43] P. Sanders and J. L. Träff. Parallel prefix (scan) algorithms for MPI. In B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, chapter 15, pages 49–57. Springer, Berlin, Heidelberg, 2006.

[44] D. Scott. Data types as lattices. *SIAM J. Comput*, 5(3):522–87, 1976.

[45] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07 Proceedings of the 22nd ACM SIGGRAPH / EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[46] V. B. Shah, A. Edelman, S. Karpinski, and J. Bezanson. Novel algebras for advanced analytics in julia. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–4, Waltham, MA, 2013. IEEE.

[47] J. Sklansky. Conditional-sum addition logic. *IEEE Trans. Electronic Comput.*, EC-9(2):226–231, 1960.

[48] D. R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Sci. Comput. Program.*, 8(3):213–229, 1987.

[49] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1995.

[50] C. Strachey. Fundamental concepts in programming languages. *Higher-Order Symbol. Comput.*, 13(1-2):11–49, 2000.

[51] W. W. Tait. Infinitely long terms of transfinite type. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, chapter 10, pages 176–185. North-Holland Publishing, Amsterdam, 1965.

[52] G. Viswanathan and J. R. Larus. User-defined Reductions for Efficient Communication in Data-Parallel Languages. Technical report, University of Wisconsin, Madison, 1996.

[53] H. Wang, A. Nicolau, and K.-Y. S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Trans. Comput.*, 45(11):1257–1271, 1996.

[54] N. Wilt. Scan. In *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, chapter 13. Pearson Education, 2013.

APPENDIX

A.   THE RENDER FUNCTION

This appendix shows the `render` function used to generate the figures in Figure 1. We use here the Compose package for declarative vector graphics in Julia [31].

`render` is defined with two methods. The first describes how to render each elementary operation is represented as a logic `gate` with inputs `ins` and outputs `outs`. The `render( G::gate, ...)` method draws the inputs as small circles at coordinates `ipoints` and links them to the outputs, which are drawn as large circles at coordinates `opoints`. The second method for `render` describes how to render the instrumentation data in an `AccessArray`: a first pass through the data computes the depth of the tree to draw, and the second pass actually places gates appropriately for each operation, and finally vertical guidelines for every processor are added.

The code as written uses a heuristic taking advantage of the sequential left-to-right access order of the serialized prefix sum kernels: if an operation accesses an index lower than the most recently accessed index, then the current operation defines a new layer of the computation tree. This simplifying assumption does not fundamentally change the main idea of rendering instrumentation data being acquired at the level of individual variables, and more sophisticated reasoning about tree depths can be used as necessary.

```julia
using Compose

type gate
    ins :: Vector
    outs:: Vector
end

function render(G::gate, x, y, y0; ri=0.1, ro=0.25)
    ipoints = [(i, y0+ri) for i in G.ins]
    opoints = [(i, y0+0.5) for i in G.outs]
    igates  = [circle(i..., ri) for i in ipoints]
    ogates  = [circle(i..., ro) for i in opoints]
    lines = [line([i, j]) for i in ipoints,
                              j in opoints]

    compose(context(units=UnitBox(0.5, 0, x, y+1)),
        compose(context(), stroke("black"),
            fill("white"), igates..., ogates...),
```

```julia
        compose(context(), linewidth(0.3mm),
            stroke("black"), lines...))
end

function render(A::AccessArray)
    #Scan to find maximum depth
    olast = depth = 0
    for y in A.history
        (any(y[1] .<= olast)) && (depth += 1)
        olast = maximum(y[2])
    end
    maxdepth = depth

    #Map each operation onto a gate
    olast = depth = 0
    C = Any[]
    for y in A.history
        (any(y[1] .<= olast)) && (depth += 1)
        push!(C, render(gate(y...), A.length,
            maxdepth, depth))
        olast = maximum(y[2])
    end

    #Compose everything together with processor
    #guidelines
    push!(C, compose(context(
      units=UnitBox(0.5, 0, A.length, 1)),
      [line([(i,0), (i,1)]) for i=1:A.length]...,
      linewidth(0.1mm), stroke("grey")))
    compose(context(), C...)
end
```