Parallel Programming with Big Operators

Changhee Park
KAIST
changhee.park@kaist.ac.kr

Guy L. Steele Jr.
Oracle Labs
guy.steele@oracle.com

Jean-Baptiste Tristan
Oracle Labs
jean.baptiste.tristan@oracle.com

Abstract

In the sciences, it is common to use the so-called "big operator" notation to express the iteration of a binary operator (the reducer) over a collection of values. Such a notation typically assumes that the reducer is associative and abstracts the iteration process. Consequently, from a programming point-of-view, we can organize the reducer operations to minimize the depth of the overall reduction, allowing a potentially parallel evaluation of a big operator expression. We believe that the big operator notation is indeed an effective construct to express parallel computations in the Generate/Map/Reduce programming model, and our goal is to introduce it in programming languages to support parallel programming. The effective definition of such a big operator expression requires a simple way to generate elements, and a simple way to declare algebraic properties of the reducer (such as its identity, or its commutativity). In this poster, we want to present an extension of Scala with support for big operator expressions. We show how big operator expressions are defined and how the API is organized to support the simple definition of reducers with their algebraic properties.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Parallelism, Mathematical notation, Scala

1. A useful parallel programming tool

In mathematics, in the sciences, it is sometimes convenient to express the iteration of an operator to a collection of values using a "big operator". As an example, $\sum$ is a big operator that denotes the iterated application of the binary operator $+$ over a collection of numbers. This operator can be used as part of what we call in this a paper a "big expression" such as:

$$\sum_{x \in [1..n]} e^{ix}$$

This expression is a compact (and declarative) way to denote the sum of all the complex numbers e^{ix} for all values of x comprised between 1 and n. We can think of such a big operator as a lifting of an operator to a collection of values. Typically, the lifted operator is binary and associative. This last property explains why we can abstract away the order in which we apply the lifted operator to our collection of values, resulting in a declarative expression. In our work, we also require the lifted operator to possess an identity: it is often convenient to have an identity element at hand and it allows for big expression to be defined even when the iterated collection is empty.

In a typical programming language, a big expression could be implemented with a loop, a recursive function definition, or simply as a call to a function such as `fold` if the collection has one. The drawback of these definitions is that they typically enforce a particular evaluation order. As an example, summing the list of numbers $[x_1, x_2, x_3, x_4]$ may correspond to the evaluation of the expression $((x_1 + x_2) + x_3) + x_4$. While this is semantically correct, we are missing an opportunity to exploit the associativity of $+$, which could allow for a balanced parallel evaluation of the expression, as in $(x_1 + x_2) + (x_3 + x_4)$. This is what makes big operator notation interesting for parallel programming. As an example it was in some form a feature of `Fortress` [1], a programming language oriented towards parallel programming. The more a programmer expresses application of an operator over a collection using this notation, the less he arbitrarily enforces an evaluation order where it is not semantically necessary.

Before adding big expressions to a language, it is useful to identify and name what constitutes such an expression. As an example, the big expression mentioned at the beginning of this section has four important constituents:

- The collection of integers $[1..n]$: It is the collection from which we generate values, we call it the *generator*. In general, the generator can be composed from other generators by taking their cross product, nesting them, or filtering them. These are fairly standard operations in any language that provides support for collection-comprehension or for-comprehension.
- The function from integers to complex numbers e^{ix}: we call it the *mapping*. It is also standard in any language that supports collection-comprehension or for-comprehension.
- The type of the values we are reducing, which does not appear explicitly in our example. In this case, it corresponds to the type of complex numbers. We refer to this type as the type of the big expression.
- Finally, $\sum$, the big operator itself. We refer to it as the *reducer*.

A big expression therefore needs to specify a big operator, a generator –perhaps defined by composing simpler generators–, and a mapping. Aside from the big operator, this is very similar to comprehension notations such as $\{\ f(x) \mid x \in [1..n]\ \}$. In fact, the big expression is a generalization of such comprehension notations. As an example, the previous set-comprehension can be rewritten as the big expression: $\bigcup_{x \in [1..n]} \{\ f(x)\ \}$.

2. An implementation in Scala

To experiment with big expressions, we introduce them into the Scala programming language [2]. There are at least two reasons that make Scala an appropriate language to support big expressions.

PPoPP'13, February 23–27 2013, Shenzhen, China.
ACM 978-1-4503-1922-5/13/02.

First, Scala supports for-comprehensions, and provides a means to compose generators. Second, Scala has a parallel collection library that we can build on to evaluate big expressions in parallel.

Let us assume for now that we have `sum`, a big operator that sums integers. We program the summation of the square of all even integers between 1 and n as

```
BIG sum (i <- 0 to n if i % 2 == 0) i * i
```

Our big expression is introduced by the keyword "BIG", and followed by the big operator, the generator, and the mapping. The type of this big expression is `Int` and the type of the big operator is `Reducer[Int]`.

2.1 The definition of big operators

To define a big operator, the end user must extend the class `Reducer` whose definition is similar to (the actual definition is slightly different, but this is an implementation detail):

```
trait Associative

trait Identity[T] { val identity: T }

trait Reducer[T] extends Function2[T,T,T]
                    with Associative
                    with Identity[T]
```

A reducer should have a binary operator (the `apply` method inherited from `Function2`), be associative, and possess an identity, as inherited from trait `Identity`. For instance, we can define `sum` as follows:

```
val sum = new Reducer[Int] {
     def apply(x: Int, y: Int): Int = x + y;
     val identity = 0
}
```

2.2 Compilation of big expressions

Big expressions are compiled by desugaring. In the simplest case, a big expression of the form

```
BIG big_op (i <- g) mapping
```

can simply be desugared into

```
g.generate(big_op, i => mapping)
```

We require every collection that is to be used as a generator to mix in the trait `Generator`. We define this trait as follows:

```
trait Generator[A] {
  def generate[B](bin_op: (B,B) => B,
                  mapping: A => B)
}
```

It contains a single function `generate` that expects a binary operator and the mapping. It is up to the library writer to implement this function. As an example, he could implement it by mapping `mapping` on the collection and then calling method `reduce` with `bin_op` as an argument.

We explain in the next section why we do not require `bin_op` to be a `Reducer`. To evaluate a big operator in parallel, the library writer may define function `generate` as follows:

```
def generate[B](bin_op: (B,B) => B, mapping: A => B) =
  bin_op match {
    case b: Reducer[B] => self.par.aggregate ...
  }
```

where method `par` is the method provided by Scala collections that converts serial collections to their parallel counterpart, allowing parallel evaluation of the big expression.

2.3 Inline definition of big operators

To improve the use of big expressions, we would like to allow the end user to be able to simply define a big operator within the big expression. For example, the end user can avoid pre-defining `sum` and write the previous big expression as

```
BIG (fn with { val identity = 0 } (x,y) => x + y)
    (i <- 0 to n if i % 2 == 0) i * i
```

or even

```
BIG + with { val identity = 0 }
    (i <- 0 to n if i % 2 == 0) i * i
```

Also, an end user who doesn't want to provide the identity element can define the big expression as

```
BIG + (i <- 0 to n if i % 2 == 0) i * i
```

This is for that reason that method `generate` is declared with such generality. In such a case, our desugarer will compile the method name into an object of type `Function2` which is not a subtype of `Reducer`. If the collection is empty, the implementation of `generate` may handle such a binary operator by throwing an exception.

Finally, not all type and binary operator have a simple identity. Think for instance of the binary operator set-intersection. In this case, we provide the end-user with a simple way of lifting the type `Set[...]` to `Option[Set[...]]` where None represents the universe.

3. Configuration of the big operator

As presented above, the class `Reducer` mixes in the trait `Associative` and the trait `Identity`. Trait `Associative` is only informative, it does not provide or require any definition. Yet, we "attach" this information to the binary operator, since it is assumed that the binary operator is associative. This extra piece of information is a means to communicate to the compiler and the runtime a property that could used to increase performance.

Following this idea, we add to our API other algebraic traits, allowing the programmer to attach as much information as possible to the binary operator underlying the big operator. For instance, on a cluster, the runtime may make good use of the fact that a binary operator is commutative to distribute appropriate workloads to different nodes. Such algebraic could include, among other things:

```
trait Commutative

trait Absorber[T] { val absorber: T }
```

4. Conclusion

To conclude, we think that big operators and their associated big expressions are a useful tool to write better parallel programs. Also, we think that it is useful to tell the compiler and the runtime of properties we know about the big operator. The design questions remain largely open. We are experimenting with some designs in Scala, but deciding on the syntax of big expressions, especially inlined one, and on an API to support simple description of algebraic properties of big operators requires more practice and feedback.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, March 2008.

[2] M. Odersky. *The Scala Language Specification*. EPFL, May 2011.