PGX Algorithm

Introduction

PGX Algorithm consists of two components: a Java API that you write your code against and a compiler that can transform said code. Using PGX Algorithm you can write your graph algorithm in Java, but have it compiled to optimized code that runs on PGX.SM or PGX.DIST.

A PGX Algorithm program is a syntactically valid Java program. Therefore, this document does not specify a syntax definition and only explains the static and dynamic semantics of PGX Algorithm. Unless specified otherwise, the semantics of a PGX Algorithm program are the same as the semantics of the equivalent Java program.

Restrictions

PGX Algorithm is a language that looks a lot like Java. Indeed, every syntactically valid PGX Algorithm program is a syntactically valid Java program. However, PGX Algorithm is different from Java in several ways. First, PGX Algorithm only supports a small subset of Java. For example, it is not possible to instantiate arbitrary objects. Second, PGX Algorithm assigns a different semantics to the language constructs. For example, there is no such thing as a *reference* type and all values are passed by-value. Moreover, you cannot do any of the following in PGX Algorithm:

- Call any Java method other than those mentioned in this specification.
- Have the graph algorithm class extend some other class.
- Instantiate objects using `new`.

These restrictions ensure that users cannot write arbitrary Java code. For example, it is not possible to write code that reads directly from memory or performs IO such as reading from the filesystem, opening a connection to an external server, etc.

Anatomy of a PGX Algorithm program

A PGX Algorithm program consists of a single class that is annotated with `@GraphAlgorithm`. The class contains exactly one public method which is the entry-point of the program. The class may contain any number of non-public methods which act as local procedures.

The methods may take any number of parameters. A distinction is made between input and output parameters. A parameter is an output parameter if it

is annotated with `@Out` and an input parameter otherwise. Input parameters can only be read (read-only), output parameters can be read and written (read-write). If the input parameter is a collection-type (i.e. property, set, sequence, map, or vector), then neither the collection itself nor any of its contents can be altered.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public void algorithm(PgxGraph g) {
    System.out.println("Hello, world!");
  }
}
```

In the labeled-property graph model, a vertex (resp. edge) can hold any number of attributes (key-value pairs) called *properties*. In PGX Algorithm, a vertex property (resp. edge property) is represented by the `VertexProperty` (resp. `EdgeProperty`) type. This is a generic type that can be parameterized by the type of the attribute value, e.g. `VertexProperty<Integer>`. Many algorithms assume the existence of some property, in which case the algorithm will expect the property to be provided as a parameter:

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public void algorithm(PgxGraph g, VertexProperty<Integer> age) {
    g.getVertices().forEach(v -> {
      System.out.println(age.get(v));
    });
  }
}
```

Types

The PGX Algorithm compiler translates method calls to certain types to optimized code. We can categorize these types as *Basic types*, *Statement types*, and *Collection types*.

Basic types	Statement types	Collection types
PgxGraph	ControlFlow	PgxMap
PgxVertex	Traversal	PgxVect
PgxEdge	Reduction	VertexProperty
-	-	EdgeProperty
-	-	VertexSequence
-	-	EdgeSequence
-	-	VertexSet
-	-	EdgeSet
-	-	Scalar

Unlike Java, in PGX Algorithm you cannot use the `new` construct to instantiate an object of an arbitrary class. Instead, some PGX Algorithm types provide a static `create()` method that you can use to create an instance of the class. The overloaded version `create(T t)` instantiates the type and uses the given value for initialization. The types that do not provide a `create()` method cannot be instantiated. This is the case for `PgxGraph`, `PgxVertex`, and `PgxEdge`.

Another difference between Java and PGX Algorithm is the distinction of primitive types and reference types. Whereas Java makes a distinction between primitive types and reference types, PGX Algorithm makes no such distinction. Consequently, the programmer does not need not worry about boxing and unboxing. The programmer must use reference types when necessary (e.g. as type parameters), but can use either reference types (e.g. `Integer`) or primitive types (e.g. `int`) when both are possible.

The rest of this section discusses for each of the above types the methods that are available and the semantics of calling said methods.

PgxGraph

A `PgxGraph` cannot be created and instead needs to be passed to the main procedure as parameter. A `PgxGraph` has the following methods:

- `getNumEdges()` returns the numer of edges in the graph.
- `getNumVertices()` returns the number of vertices in the graph.
- `getRandomVertex()` returns a random vertex in the graph.
- `getEdges()`
- `getVertices()`

The methods `getEdges()` and `getVertices()` return an `EdgeSet` and `VertexSet`, respectively. These methods cannot be called in isolation. Instead, the method call must be chained with a call to one of the methods on `VertexSet` or `EdgeSet`. For example, one may iterate over all vertices in parallel by calling `g.getVertices().forEach(Callable<T>)`.

VertexProperty and EdgeProperty

A `VertexProperty<T>` (resp. `EdgeProperty<T>`) is a generic type that can be parameterized with the type of elements of the property. A `VertexProperty<T>` (resp. `EdgeProperty<T>`) supports the following methods to read and write property values:

- `get(PgxVertex)` to get the value of the property for the given vertex.
- `set(PgxVertex, T)` to set the value of the property for the given vertex.
- `setDeferred(PgxVertex, T)` to set the value of the property for the given vertex, but the write won't be visible until the end of the parallel region (more information on *deferred assignments* below).
- `setAll(T)` to set the value of the property for all vertices.
- `setAll(Function<PgxVertex, T>)` to set the property value for each vertex to the result of evaluating the given function for said vertex.

In addition, the following methods can be used to perform reductions (more information on *reductions* below):

- `decrement(PgxVertex)`
- `increment(PgxVertex)`
- `reduceAdd(PgxVertex, T)`
- `reduceAnd(PgxVertex, T)`
- `reduceMax(PgxVertex, T)`
- `reduceMin(PgxVertex, T)`
- `reduceMul(PgxVertex, T)`
- `reduceOr(PgxVertex, T)`

Collections (set, sequence, map)

PGX Algorithm supports *sets*, *sequences*, and *maps*. Sets and sequences can hold either *vertices* or *edges*, which gives rise to the four types `VertexSet`, `VertexSequence`, `EdgeSet`, and `EdgeSequence`. Maps can hold primitive types (*) as keys and primitive types + x (**) as values.

Sets and sequences provide a `forEach(Consumer<PgxVertex>)` and `forSequential(Consumer<PgxVertex>)` methods to iterate over the elements

in parallel and in sequence, respectively. Maps provide the `keys()` and `values()` methods which return a `PgxCollection<K>`. This type, in turn, provides a `forEach(Consumer<PgxVertex>)` and `forSequential(Consumer<PgxVertex>)` method to iterate over the elements.

The collection types support methods that one would generally expect on sets and sequences. Specifically, `VertexSet` and `EdgeSet` support the following methods.

(Vertex/Edge)Set	(Vertex/Edge)Sequence	PgxMap
add(PgxVertex)	back()	clear()
allMatch(Predicate)	clear()	containsKey(K)
anyMatch(Predicate)	contains(PgxVertex)	decrement(K)
avg(Function)	filter(Predicate)	get(K)
filter(Predicate)	front()	getKeyForMaxValue()
max(Function)	pop()	getKeyForMinValue()
orderBy(Function, Order)	popBack()	getMaxValue()
remove(PgxVertex)	popFront()	getMinValue()
size()	push(PgxVertex)	hasMaxValue(K)
sum(Function)	pushBack(PgxVertex)	hasMinValue(K)
-	pushFront(PgxVertex)	increment(K)
-	size()	keys()
-	-	reduceAdd(K, V)
-	-	remove(K)
-	-	set(K, V)
-	-	size()
-	-	values()

Note that the `filter`, `orderBy`, `forEach`, and `forSequential` methods are always used in a chain of method calls that ends with `forEach` or `forSequential`.

PgxVect

In PGX Algorithm, the length of a vector is part of its type. This allows the compiler to check at compile time whether certain operations (e.g. adding two vectors) are valid. However, this also means that every `PgxVect<T>` type needs to be annotated with a `@Length` annotation. This annotation takes a single string parameter that should be the name of an input parameter. For example:

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.PgxVect;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;
import oracle.pgx.api.beta.annotations.Length;

@GraphAlgorithm
public class Algorithm {
  public int algorithm(PgxGraph g, int k, @Length("k") PgxVect<Integer> vector) {
    return vector.get(0);
  }
}
```

Iteration

Arguably the most common operation in PGX Algorithm is to iterate over elements such as looping over all vertices, looping over all edges, or looping over a vertex's neighbors. All collection types (`VertexSet`, `EdgeSet`, `VertexSequence`, `EdgeSequence`) support a `forEach` and `forSequential` method by which you can iterate over the collection *in parallel* and *in sequence*, respectively. For example:

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public int algorithm(PgxGraph g) {
    G.getVertices().forEach(v -> {
      // This body is executed for every vertex, in parallel
    });

    G.getVertices().forSequential(v -> {
      // This body is executed for every vertex, in sequence
    });
  }
}
```

Traversal

PGX Algorithm supports traversing a graph in breath-first and depth-first order. A traversal always starts with a call to the static method `Traversal.inBFS` or `Traversal.inDFS`. These methods take a `PgxGraph` (the graph being traversed) and a `PgxVertex` (the vertex at which the traversal starts) as argument. The call to `inBFS` (resp. `inDFS`) should be chained with a call to `forward` or `backward`

(or both) to indicate what computation needs to be performed in the forward- and backward traversal. The traversal can be further configured by adding a call to `filter`, `backwardFilter`, `navigator`, or `direction` to the chain of method calls.

Within the `forward` and `backward` function there are several methods that you can call for extra information:

- The static method `Traversal.currentLevel` to retreive the current level of the BFS traversal
- The static method `Traversal.stopTraversal` to stop the traversal at the earliest opportunity.
- The method `PgxVertex.getUpNeighbors` and `PgxVertex.getDownNeighbors` to get the up- and down-neighbors of the current vertex in this traversal.

Reductions

When execution takes place in a parallel threads, it is possible that multiple threads read or write the same memory segment. Reductions allow you to reduce multiple values to a single result without introducing read-write conflicts. PGX Algorithm supports both reductions in assignment form and reductions in expression form. Reductions in expression form are syntactic sugar for reductions in assignment form. That is, reductions in expression form do not add any features to the language but make it more convenient to express certain reductions.

Reduction in assignment form The following reduction methods are available on the types `EdgeProperty<T>`, `VertexProperty<T>`, `PgxMap<T, U>`, `PgxVect<T>`, and `Scalar<T>`:

- `reduceAdd`
- `reduceAnd`
- `reduceMax`
- `reduceMin`
- `reduceMul`
- `reduceOr`

Some methods take a key as parameter, e.g. `EdgeProperty<T>.reduceAdd(PgxEdge, T)` takes the edge whose property should be reduced to.

Reduction in expression form Some common reductions can be expressed more concisely in expression form. PGX Algorithm supports the following methods, whose return value is the result of the reduction:

- `VertexSet<T>.sum(Function<PgxVertex, T>)`
- `EdgeSet<T>.sum(Function<PgxVertex, T>)`

PGX Algorithm supports the following methods to sum the values of a vertex or edge property:

- `VertexSet<T>.sum(VertexProperty<T>)`
- `EdgeSet<T>.sum(VertexProperty<T>)`

For example, to print the sum of the degree of each vertex:

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public void totalDegree(PgxGraph g) {
    System.out.println(g.getVertices().sum(v -> v.getDegree()));
  }
}
```

Multiple atomic reductions Sometimes you want to reduce more than one variable atomically. For example, you may be interested in the smallest degree as well as the vertex that has this degree. If you implement this as two independent reductions then there is a potential of race conditions under parallel memory consistency.

PGX Algorithm provides the `Reduction` class to express multiple atomic reductions. The `Reduction` class has two static methods by which you can update a variable to a minimum (maximum) value: `updateMinValue` and `updateMaxValue`. The return type of these methods is `Reduction` such that you can chain any number of calls to `andUpdate`. Only if a new minimum (maximum) is found will the chain of updates take effect, and all updates will occur atomically.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.PgxVertex;
import oracle.pgx.api.beta.Reduction;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public void smallestDegree(PgxGraph g) {
    double minDegree;
```

```
    PgxVertex minDegreeVertex = PgxVertex.NONE;

    g.getVertices().forEach(v -> {
      Reduction
        .updateMinValue(minDegree, v.getDegree())
        .andUpdate(minDegreeVertex, v);
    });

    System.out.println(minDegree);
    System.out.println(minDegreeVertex);
  }
}
```

Deferred Assignments Deferred assignments are assignments that are not visible until the end of the closest parallel region they occur in. Deferred assignments are expressed using one of the following two methods:

- `VertexProperty<T>.setDeferred(PgxVertex, T)`
- `EdgeProperty<T>.setDeferred(PgxVertex, T)`

The following example shows when deferred assignments are useful. The `forEach` loop iterates in parallel over all vertices `v`. The `sum` loop iterates in parallel over the neighbors `w` of `v` and reads `w.score`. Since `w` may be the neighbor of many vs, it is possible that `w.score` is read by multiple threads at the same time. By using `setDeferred` the write is not visible until at the end of the `forEach` and all threads will read the same value.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;
import oracle.pgx.api.beta.annotations.Out;

@GraphAlgorithm
public class Algorithm {
  public void algorithm(PgxGraph g, @Out VertexProperty<Double> score) {
    g.getVertices().forEach(v -> {
      double totalScore = v.getNeighbors().sum(w -> score.get(w));
      score.setDeferred(v, totalScore);
    });
  }
}
```

Scalar Imagine you want to reduce many values to a single value. For example, you may want to compute the sum of the degree of each vertices. In Java, such

a reduction can be expressed by declaring some variable outside the loop and incrementing the variable inside the loop. The problem is that the body of an iteration in PGX Algorithm is represented by a lambda expression and Java requires that variables used in lambda expression are final or effectively final. To work around this limitation, PGX Algorithm provides the `Scalar<T>` type. The `Scalar<T>` type provides the following methods:

- A `Scalar<T>` can be created using the `create()` or `create(T)` methods.
- A scalar can be reduced to using one of the reduction methods (e.g. `reduceAdd`, `reduceMul`, `reduceAnd`).
- A scalar can be unwrapped by calling the `get()` method.

The following example uses a `Scalar<T>` to reduce to `sumOfDegrees` from inside the lambda expression.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.Scalar;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
public class Algorithm {
  public int algorithm(PgxGraph g) {
    Scalar<Integer> sumOfDegrees = Scalar.create(0);

    g.getVertices().forEach(v -> {
      sumOfAges.reduceAdd(v.getDegree());
    });

    return sumOfAges.get();
  }
}
```

Supported Java Values and Methods

PGX Algorithm supports the following Java values and methods. Names that live in the `java.lang` package are imported by default similar to Java. Names that are not imported by default can be improted using an `import` or `import static` statement.

- `java.lang.Integer.MAX_VALUE`
- `java.lang.Integer.MIN_VALUE`
- `java.lang.Long.MAX_VALUE`
- `java.lang.Long.MIN_VALUE`

- `java.lang.Float.POSITIVE_INFINITY`
- `java.lang.Float.NEGATIVE_INFINITY`
- `java.lang.Float.MAX_VALUE`
- `java.lang.Float.MIN_VALUE`
- `java.lang.Double.POSITIVE_INFINITY`
- `java.lang.Double.NEGATIVE_INFINITY`
- `java.lang.Double.MAX_VALUE`
- `java.lang.Double.MIN_VALUE`
- `java.lang.Math.abs`
- `java.lang.Math.sqrt`
- `java.lang.Math.pow`
- `java.lang.Math.log`
- `java.lang.Math.min`
- `java.lang.Math.max`
- `System.out.println`

The value of these identifiers is analogus to their values in Java. For example, `Integer.MAX_VALUE` represents the largest possible value in a PGX Algorithm program and `Math.abs` computes the absolute value of the given number. The method `System.out.println` prints to the logger that is registered with the runtime.

Pass by Value vs. Pass by Reference

Java distinguishes between primitive values and reference values. All values are passed by value, i.e. a copy if the primitive value or reference value is created.

In PGX Algorithm, non-wrapper objects act as if they are primitive values. When assigning an object to another object, the user *must* call `.clone()`. The only exception is when calling a local procedure, in which case a reference is passed just like in Java.

Control-Flow

PGX Algorithm supports many of the conventional control-flow constructs such as `if`, `if-else`, `while`, `do-while`, and `return`. These constructs have the same semantics as in Java. The following Java control-flow constructs are not supported:

- Try, throw, and catching exceptions
- Regular for loops (`for (init; condition; increment)`)
- Enhanced for loops (`for (parameter : expression)`)
- Switch statements

- Synchronized statements
- Labeled statements
- Assert statements

In addition, PGX Algorithm allows you to return from a function by calling the static method `ControlFlow.exit(Object)` or `ControlFlow.exit()`, depending on whether you want to return a value or not. The semantics of this method depend on the execution context:

- If the `exit` method is called from within the lambda that is passed to `forSequential`, then execution terminates immediately and control is passed back to the calling method.
- If the `exit` method is called from within the lambda that is passed to `forEach`, then there is no guarantee when parallel execution threads are terminated. That is, it is possible that all other parallel execution threads are terminated immediately or that they run until completion.

User-Defined Functions

A User-Defined Function (UDF) is a function that is defined outside of PGX Algorithm (possibly even in a different programming language). A UDF has a namespace, name, list of argument types, and return type. PGX Algorithm only defines how to call a UDF and leaves it up to the runtime to decide how to register a UDF.

To make the UDF available in a PGX Algorithm program, create an abstract method with the correct name, argument types, and return type. This abstract method should be annotated with `@Udf`. This annotation takes a parameter `namespace` that can be assigned the namespace of the UDF (as registered with the runtime). The following example calls a UDF `acme.getRating` that takes an integer and returns an integer.

```
import oracle.pgx.api.beta.annotations.GraphAlgorithm;
import oracle.pgx.api.beta.annotations.Udf;

@GraphAlgorithm
public abstract class Example {
  public void example() {
    System.out.println(getRating(2));
  }

  @Udf(namespace = "acme")
  abstract int getRating(int number);
}
```

Parallel Execution and Consistency

PGX Algorithm is designed to make it easy to exploit the data parallelism that is present in many graph algorithm. The basic idea is that the user describes parallel regions with calls such as `VertexSet.forEach`, while the compiler and the runtime takes care of the details of parallel execution such as thread creation or job scheduling.

Parallelism in PGX Algorithm follows the fork–join model. The execution branches off in parallel at designated points in the program, to "join" at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
class SumAges {
  public Integer sumAges(PgxGraph g, VertexProperty<Integer> age) {
    Scalar<Integer> sumAge = Scalar.create(0);

    g.getVertices().forEach(v -> {
      sumAge.reduceAdd(age.get(v));
    });

    return sumAge.get();
  }
}
```

Conceptually, the execution of a parallel region is maximally parallelized. For instance, in the above code example, the parallel region is executed concurrently for every node `n` in the graph `g`. No ordering is guaranteed between instances of this parallel execution region. An implementation is free to choose how many threads are actually utilized.

Nested Parallelism

Concurrent execution instances are independent. Therefore, in the following example, even when all the t-instances from a single n-instance have been merged, there can be other t-instances concurrently – those t-instances are originated from other n-instances.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
```

```
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
class Algorithm {
  public Integer algorithm(PgxGraph g, VertexProperty<Integer> age) {
    Scalar<Integer> sum = Scalar.create(0);

    g.getVertices().forEach(n -> {
      n.getNeighbors().forEach(t -> {
        sum.reduceAdd(age.get(t));
      })
    });

    return sum.get();
  }
}
```

The compiler is free to decide the degree of parallelism. For the preceding example this means that the number of logical execution threads can be anywhere between 1 and the number of vertices.

Memory Consistency

The basic memory model in PGX Algorithm is sequential memory consistency. ... Parallel regions, however, adopt a different memory consistency model. This model assumes that the parallel region is being executed concurrently and that there are data races between concurrent execution instances. This model does NOT guarantee anything about the visibility of writes which are made by concurrent executions. For example, a write made by one instance of a parallel region may or may not be visible to other instances of the parallel region. The model guarantees the following things:

- Self-visibility: A write by an execution instance is always visible to the current instance later in program order, unless the write is overwritten by another concurrent instance.
- Eventual visibility: At the end of the parallel region, when all the concurrent executions are merged, every write made by concurrent execution instance of the region becomes visible.

A write-write data race is the case when multiple concurrent writes is being written to the same variable (or property location). In such a case, at the end of the parallel region, one (and only one) of those writes becomes effective; however, it is non-deterministic which of those writes will be the effective one. A read-write data race is the case when a variable (or a property access) is concurrently read

and write at the same time. In this case, a concurrent read may or may not see the result of a write from another concurrent instance. Moreover, under parallel memory consistency, there is no guarantee that multiple writes would be visible in the same order to every concurrent execution instance.

Fundamentally, PGX Algorithm's parallel memory consistency assumes that concurrent reads and writes inevitably incur data-races and thus becomes non-deterministic. The designers of PGX Algorithm believe that such non-determinism is a fundamental nature of parallel graph processing. Nevertheless, PGX Algorithm provides methods which can enforce certain determinism out of such non-deterministic concurrent execution.

Determinism Under Parallel Memory Consistency

Parallel memory consistency allows data races among concurrent writes and concurrent reads. PGX Algorithm provides certain mechanisms to enforce deterministic results. This section discusses these mechanisms.

Reductions Reductions are the most important determinism-enforcing mechanism in PGX Algorithm. Reductions are a mathematical mechanism to compute a single value out of a set of values in a deterministic way. For example, summation is a reduction which sums up all values in a collection.

Reductions are represented as the methods `reduceAdd`, `reduceSub`, `reduceMul`, `reduceOr`, `reduceAnd`, `reduceOr`, `reduceMax`, and `reduceMin` on the classes `Scalar`, `VertexProperty`, `EdgeProperty`, `VertexSet`, `EdgeSet`, `PgxMap`, `PgxVect`, and `Scalar`. For example, `VertexProperty<Integer>#reduceAdd(PgxVertex, Integer)` adds the given integer to the property value of the given vertex. The code below shows one reduction and two non-reductions.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;

@GraphAlgorithm
class SumAges {
  public Integer sumAges(PgxGraph g, VertexProperty<Integer> age) {
    Scalar<Integer> sumAge = Scalar.create(0);
    int sumAgeWrong = 0;

    g.getVertices().forEach(v -> {
      // Reduction by addition.
      sumAge.reduceAdd(age.get(v));

      // Not a reduction! The result is non-deterministic.
```

```
      sumAge.set(sumAge.get() + age.get(v));

      // Not a reduction! The result is non-deterministic.
      sumAgeWrong += age.get(v);
    });

    return sumAge.get();
  }
}
```

Reductions on a `VertexSet` or `EdgeSet` can be expressed more succinctly using the `sum`, `max`, `avg`, `anyMatch`, or `allMatch` methods. For example, to compute the average degree of your neighbors:

```
int averageDegree = v.getNeighbors().avg(w -> w.getDegree());
```

which can be expressed even more succinctly using a method reference:

```
int averageDegree = v.getNeighbors().avg(PgxVertex::getDegree);
```

Argument Attaining Minimum/Maximum Value The PGX Algorithm class `Reduction` can be used compute the minimum/maximum value as well as the vertex at which

Deferred Assignments In the bulk-synchronous memory consistency model, a write to a memory location is not visible to any concurrent execution instance, even to the instance that has made the write, until the (specified) synchronization point. At the synchronization point, however, all updates made inside the loop become visible at once.

PGX Algorithm supports bulk-synchronous memory consistency through *deferred assignments*. The method `VertexProperty<T>#setDeferred(PgxVertex, T)` sets the property of the given vertex to the given value. The method `EdgeProperty<T>#setDeferred(PgxEdge, T)` sets the property of the given edge to the given value. In both cases the synchronization point is the end of the closest loop.

The example below shows how deferred assignments may be used. Every vertex computes a rank based on its neighbors. The update for one vertex should not be visible by other vertices until after the parallel region.

```
import oracle.pgx.api.beta.PgxGraph;
import oracle.pgx.api.beta.VertexProperty;
import oracle.pgx.api.beta.annotations.GraphAlgorithm;
```

```
import oracle.pgx.api.beta.annotations.Out;

@GraphAlgorithm
class Pagerank {
  public void pagerank(PgxGraph g, @Out VertexProperty<Integer> rank) {
    g.getVertices().forEach(v -> {
      Integer receive = v.getInNeighbours().sum(w ->
        rank.get(w) / w.getOutDegree()
      );

      rank.setDeferred(v, receive);
    });
  }
}
```