Ugrid Reference

Contents

1 Overview

Ugrid is a fast and general purpose distributed data processing system. It provides a high-level API in Javascript and an optimized parallel execution engine.

A Ugrid application consists of a *master* program that runs the user code and executes various *parallel operations* on a cluster of *workers*.

The main abstraction Ugrid provides is a *distributed array* (DA) which is similar to a Javascript *array*, but partitioned accross the workers that can be operated in parallel.

There are several ways to create a DA: *parallelizing* an existing array in the master program, or referencing a dataset in a distributed storage system (such as HDFS), or *streaming* the content of any source that can be processed through Node.js *Streams*. We call *source* a function which initializes a DA.

DAs support two kinds of operations: *transformations*, which create a new distributed array from an existing one, and *actions*, which return a value to the *master* program after running a computation on the DA.

For example, `map` is a transformation that applies a function to each element of a DA, returning a new DA. On the other hand, `reduce` is an action that aggregates all elements of a DA using some function, and returns the final result to the master.

Sources and *transformations* in Ugrid are *lazy*. They do not start right away, but are triggered by *actions*, thus allowing efficient pipelined execution and optimized data transfers.

A first example:

```
var uc = require('ugrid').context();        // create a new context
uc.parallelize([1, 2, 3, 4]).               // source
   map(function (x) {return x+1}).          // transform
   reduce(function (a, b) {return a+b}, 0). // action
   then(console.log);                       // process result: 14
```

2 Working with Distributed Arrays

After having initialized a cluster context using ugrid.context(), one can create a distributed array using the following sources:

Source Name	Description
lineStream(stream)	Create a DA from a text stream
objectStream(stream)	Create a DA from an object stream
parallelize(array)	Create a DA from an array
textFile(path)	Create a DA from a regular text file

Transformations operate on a DA and return a new DA. Note that some transformation operate only on DA where each element is in the form of 2 elements array of key and value (`[k,v]` DA):

```
[[Ki,Vi], ..., [Kj, Vj]]
```

A special transformation `persist()` enables one to *persist* a DA in memory, allowing efficient reuse accross parallel operations.

Transformation Name	Description	in	out
cartesian(other)	Perform a cartesian product with the other DA	v w	[v,w]
coGroup(other)	Group data from both DAs sharing the same key	[k,v] [k,w]	[k,[[v],[w]]]
distinct()	Return a DA where duplicates are removed	v	w
filter(func)	Return a DA of elements on which function returns true	v	w
flatMap(func)	Pass the DA elements to a function which returns a sequence	v	w
groupByKey()	Group values with the same key	[k,v]	[k,[v]]
intersection(other)	Return a DA containing only elements found in both DAs	v w	v
join(other)	Perform an inner join between 2 DAs	[k,v]	[k,[v,w]]
leftOuterJoin(other)	Join 2 DAs where the key must be present in the other	[k,v]	[k,[v,w]]
rightOuterJoin(other)	Join 2 DAs where the key must be present in the first	[k,v]	[k,[v,w]]
keys()	Return a DA of just the keys	[k,v]	k
map(func)	Return a DA where elements are passed through a function	v	w
mapValues(func)	Map a function to the value field of key-value DA	[k,v]	[k,w]
reduceByKey(func, init)	Combine values with the same key	[k,v]	[k,w]
persist()	Idempotent. Keep content of DA in cache for further reuse.	v	v
sample(rep, frac, seed)	Sample a DA, with or without replacement	v	w
subtract(other)	Remove the content of one DA	v w	v
union(other)	Return a DA containing elements from both DAs	v	v w
values()	Return a DA of just the values	[k,v]	v

Actions operate on a DA and return a result to the *master*. Results are always returned asynchronously. In a case of a single result, it is returned through a either a callback, or an ES6 promise. In the case of multiple value results, the action returns a readable stream.

Action Name	Description	out
aggregate(func, func, init)	Similar to reduce() but may return a different type	value
collect()	Return the content of DA	stream of elements
count()	Return the number of elements from DA	number
countByKey()	Return the number of occurrences for each key in a `[k,v]` DA	stream of [k,number]
countByValue()	Return the number of occurrences of elements from DA	stream of [v,number]
foreach(func)	Apply the provided function to each element of the DA	**not implemented**
lookup(k)	Return the list of values `v` for key `k` in a `[k,v]` DA	stream of v
reduce(func, init)	Aggregates DA elements using a function, return a single value	value

3 Ugrid module

The Ugrid module is the main entry point for Ugrid functionality. To use it, one must `require('ugrid')`.

3.1 ugrid.context([config])

Creates and returns a new context which represents the connection to the Ugrid cluster, and which can be used to create DAs on that cluster. Config is an *Object* which defines the cluster server, with the following defaults:

```
{
  host: 'localhost',    // Cluster server host, settable also by UGRID_HOST env
  port: '12346'         // Cluster server port, settable also by UGRID_PORT env
}
```

Example:

```
var ugrid = require('ugrid');
var uc = ugrid.context();
```

3.1.1 uc.end()

Closes the connection to the cluster.

3.1.2 uc.parallelize(array)

Returns a new DA containing elements from the *Array* array.

Example:

```
var a = uc.parallelize(['Hello', 'World']);
```

3.1.3 uc.textFile(path)

Returns a DA of lines composing the file specified by path *String*.

Note: If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.

Example, the following program prints the length of a text file:

```
var lines = uc.textFile('data.txt');
lines.map(s => s.length).reduce((a, b) => a + b, 0).then(console.log);
```

3.1.4 uc.lineStream(input_stream)

Returns a DA of lines of text read from input_stream *Object*, which is a readable stream where DA content is read from.

The following example computes the size of a file using streams:

```
var stream = fs.createReadStream('data.txt', 'utf8');
uc.lineStream(stream).
   map(s => s.length).
   reduce((a, b) => a + b, 0).
   then(console.log);
```

3.1.5 uc.objectStream(input_stream)

Returns a DA of Javascript *Objects* read from input_stream *Object*, which is a readable stream where DA content is read from.

The following example counts the number of objects returned in an object stream using the mongodb native Javascript driver:

```
var cursor = db.collection('clients').find();
uc.objectStream(cursor).count().then(console.log);
```

3.2 Distributed Arrays methods

DA objects, as created initially by above ugrid context source functions, have the following methods, allowing either to instantiate a new DA through a transformation, or to return results to the master program.

3.2.1 da.aggregate(reducer, combiner, init[,obj][,done])

Returns the aggregated value of the elements of the DA using two functions *reducer()* and *combiner()*, allowing to use an arbitrary accumulator type, different from element type (as opposed to `reduce()` which imposes the same type for accumulator and element). The result is passed to the *done()* callback if provided, otherwise an ES6 promise is returned.

- *reducer*: a function of the form `function(acc,val[,obj[,wc]])`, which returns the next value of the accumulator (which must be of the same type as *acc*) and with:
  - *acc*: the value of the accumulator, initially set to *init*
  - *val*: the value of the next element of the DA on which `aggregate()` operates
  - *obj*: the same parameter *obj* passed to `aggregate()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *combiner*: a function of the form `function(acc1,acc2[,obj])`, which returns the merged value of accumulators and with:
  - *acc1*: the value of an accumulator, computed locally on a worker
  - *acc2*: the value of an other accumulator, issued by another worker
  - *obj*: the same parameter *obj* passed to `aggregate()`
- *init*: the initial value of the accumulators that are used by *reducer()* and *combiner()*. It should be the identity element of the operation (i.e. applying it through the function should not change result).
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA
- *done*: a callback of the form `function (error, result)` which is called at completion. If *undefined*, `aggregate()` returns an ES6 promise.

The following example computes the average of a DA, avoiding a `map()`:

```
uc.parallelize([3, 5, 2, 7, 4, 8]).
   aggregate((a, v) => [a[0] + v, a[1] + 1],
        (a1, a2) => [a1[0] + a2[0], a1[1] + a2[1]],
        [0, 0],
        function (err, res) {
            console.log(res[0] / res[1]);
        });
// 4.8333
```

3.2.2 da.cartesian(other)

Returns a DA wich contains all possible pairs `[a, b]` where `a` is in the source DA and `b` is in the *other* DA.

Example:

```
var da1 = uc.parallelize([1, 2]);
var da2 = uc.parallelize(['a', 'b', 'c']);
da1.cartesian(da2).collect().toArray().then(console.log);
// [ [ 1, 'a' ], [ 1, 'b' ], [ 1, 'c' ],
//   [ 2, 'a' ], [ 2, 'b' ], [ 2, 'c' ] ]
```

3.2.3 da.coGroup(other)

When called on DA of type `[k,v]` and `[k,w]`, returns a DA of type `[k, [[v], [w]]]`, where data of both DAs share the same key.

Example:

```
var da1 = uc.parallelize([[10, 1], [20, 2]]);
var da2 = uc.parallelize([[10, 'world'], [30, 3]]);
da1.coGroup(da2).collect().on('data', console.log);
// [ 10, [ [ 1 ], [ 'world' ] ] ]
// [ 20, [ [ 2 ], [] ] ]
// [ 30, [ [], [ 3 ] ] ]
```

3.2.4 da.collect([opt])

Returns a readable stream of all elements of the DA. Optional *opt* parameter is an object with the default content `{text: false}`. if `text` option is `true`, each element is passed through `JSON.stringify()` and a 'newline' is appended, making it possible to pipe to standard output or any text stream.

Example:

```
uc.parallelize([1, 2, 3, 4]).
   collect({text: true}).pipe(process.stdout);
// 1
// 2
// 3
// 4
```

3.2.5 da.count([callback])

Returns the number of elements in the DA.

The *callback* is the form of `function(error, result)`, and is called asynchronously at completion. In *undefined*, an ES6 promise is returned.

Example:

```
uc.parallelize([10, 20, 30, 40]).count().then(console.log);
// 4
```

3.2.6 da.countByKey()

When called on a DA of type `[k,v]`, computes the number of occurrences of elements for each key in a DA of type `[k,v]`. Returns a readable stream of elements of type `[k,w]` where `w` is the result count.

Example:

```
uc.parallelize([[10, 1], [20, 2], [10, 4]]).
   countByKey().on('data', console.log);
// [ 10, 2 ]
// [ 20, 1 ]
```

3.2.7 da.countByValue()

Computes the number of occurences of each element in DA and returns a readable stream of elements of type `[v,n]` where `v` is the element and `n` its number of occurrences.

Example:

```
uc.parallelize([ 1, 2, 3, 1, 3, 2, 5 ]).
   countByValue().
   toArray().then(console.log);
// [ [ 1, 2 ], [ 2, 2 ], [ 3, 2 ], [ 5, 1 ] ]
```

3.2.8 da.distinct()

Returns a DA where duplicates are removed.

Example:

```
uc.parallelize([ 1, 2, 3, 1, 4, 3, 5 ]).
   distinct().
   collect().toArray().then(console.log);
// [ 1, 2, 3, 4, 5 ]
```

3.2.9 da.filter(filter[,obj])

- *filter*: a function of the form `callback(element[,obj[,wc]])`, returning a *Boolean* and where:
  - *element*: the next element of the DA on which `filter()` operates
  - *obj*: the same parameter *obj* passed to `filter()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

Applies the provided filter function to each element of the source DA and returns a new DA containing the elements that passed the test.

Example:

```
function filter(data, obj) { return data % obj.modulo; }

uc.parallelize([1, 2, 3, 4]).
   filter(filter, {modulo: 2}).
   collect().on('data', console.log);
// 1 3
```

3.2.10 da.flatMap(flatMapper[,obj])

Applies the provided mapper function to each element of the source DA and returns a new DA.

- *flatMapper*: a function of the form `callback(element[,obj[,wc]])`, returning an *Array* and where:
  - *element*: the next element of the DA on which `flatMap()` operates
  - *obj*: the same parameter *obj* passed to `flatMap()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

Example:

```
function flatMapper(data, obj) {
    var tmp = [];
    for (var i = 0; i < obj.N; i++) tmp.push(data);
    return tmp;
}

uc.parallelize([1, 2, 3, 4]).
   flatMap(flatMapper, {N: 2}).
   collect().on('data', console.log);
// [ 'hello', 2 ]
// [ 'hello', 2 ]
// [ 'world', 4 ]
// [ 'world', 4 ]
```

3.2.11 da.flatMapValues(flatMapper[,obj])

Applies the provided flatMapper function to the value of each [key, value] element of the source DA and return a new DA containing elements defined as [key, mapper(value)], keeping the key unchanged for each source element.

- *flatMapper*: a function of the form `callback(element[,obj[,wc]])`, returning an *Array* and where:
  - *element*: the value v of the next [k,v] element of the DA on which `flatMapValues()` operates
  - *obj*: the same parameter *obj* passed to `flatMapValues()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

Example:

```
function valueFlatMapper(data, obj) {
    var tmp = [];
    for (var i = 0; i < obj.N; i++) tmp.push(data * obj.fact);
    return tmp;
}

uc.parallelize([['hello', 1], ['world', 2]]).
   flatMapValues(valueFlatMapper, {N: 2, fact: 2}).
   collect().on('data', console.log);
```

3.2.12 da.foreach(callback[, obj][, done])

not implemented

This action applies a *callback* function on each element of the DA.

- *callback*: a function of the form `function(val[,obj[,wc]])`, which returns *null* and with:
  - *val*: the value of the next element of the DA on which `foreach()` operates
  - *obj*: the same parameter *obj* passed to `foreach()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA
- *done*: a callback of the form `function (error, result)` which is called at completion. If *undefined*, `foreach()` returns an ES6 promise.

In the following example, the `console.log()` callback provided to `foreach()` is executed on workers and may be not visible:

```
uc.parallelize([1, 2, 3, 4]).
   foreach(console.log).then(console.log('finished'));
```

3.2.13 da.groupByKey()

When called on a DA of type `[k,v]`, returns a DA of type `[k, [v]]` where values with the same key are grouped.

Example:

```
uc.parallelize([[10, 1], [20, 2], [10, 4]]).
   groupByKey().collect().on('data', console.log);
// [ 10, [ 1, 4 ] ]
// [ 20, [ 2 ] ]
```

3.2.14 da.intersection(other)

Returns a DA containing only elements found in source DA and *other* DA.

Example:

```
var da1 = uc.parallelize([1, 2, 3, 4, 5]);
var da2 = uc.parallelize([3, 4, 5, 6, 7]);
da1.intersection(da2).collect().toArray().then(console.log);
// [ 3, 4, 5 ]
```

3.2.15 da.join(other)

When called on source DA of type `[k,v]` and *other* DA of type `[k,w]`, returns a DA of type `[k, [v, w]]` pairs with all pairs of elements for each key.

Example:

```
var da1 = uc.parallelize([[10, 1], [20, 2]]);
var da2 = uc.parallelize([[10, 'world'], [30, 3]]);
da1.join(da2).collect().on('data', console.log);
// [ 10, [ 1, 'world' ] ]
```

3.2.16 da.keys()

When called on source DA of type `[k,v]`, returns a DA with just the elements `k`.

Example:

```
uc.parallelize([[10, 'world'], [30, 3]]).
   keys.collect().on('data', console.log);
// 10
// 30
```

3.2.17 da.leftOuterJoin(other)

When called on source DA of type `[k,v]` and *other* DA of type `[k,w]`, returns a DA of type `[k, [v, w]]` pairs where the key must be present in the *other* DA.

Example:

```
var da1 = uc.parallelize([[10, 1], [20, 2]]);
var da2 = uc.parallelize([[10, 'world'], [30, 3]]);
da1.leftOuterJoin(da2).collect().on('data', console.log);
// [ 10, [ 1, 'world' ] ]
// [ 20, [ 2, null ] ]
```

3.2.18 da.lookup(k)

When called on source DA of type `[k,v]`, returns a readable stream of values `v` for key `k`.

Example:

```
uc.parallelize([[10, 'world'], [20, 2], [10, 1], [30, 3]]).
   lookup(10).on('data', console.log);
// world
// 1
```

3.2.19 da.map(mapper[,obj])

Applies the provided mapper function to each element of the source DA and returns a new DA.

- *mapper*: a function of the form `callback(element[,obj[,wc]])`, returning an element and where:

  - *element*: the next element of the DA on which `map()` operates
  - *obj*: the same parameter *obj* passed to `map()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

The following example program

```
uc.parallelize([1, 2, 3, 4]).
   map((data, obj) => data * obj.scaling, {scaling: 1.2}).
   collect().toArray().then(console.log);
// [ 1.2, 2.4, 3.6, 4.8 ]
```

3.2.20 da.mapValues(mapper[,obj])

- *mapper*: a function of the form `callback(element[,obj[,wc]])`, returning an element and where:
  - *element*: the value v of the next [k,v] element of the DA on which `mapValues()` operates
  - *obj*: the same parameter *obj* passed to `mapValues()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

Applies the provided mapper function to the value of each `[k,v]` element of the source DA and return a new DA containing elements defined as `[k, mapper(v)]`, keeping the key unchanged for each source element.

Example:

```
uc.parallelize([['hello', 1], ['world', 2]]).
   mapValues((a, obj) => a*obj.fact, {fact: 2}).
   collect().on('data', console.log);
// ['hello', 2]
// ['world', 4]
```

3.2.21 da.reduce(reducer, init[,obj][,done])

Returns the aggregated value of the elements of the DA using a *reducer()* function. The result is passed to the *done()* callback if provided, otherwise an ES6 promise is returned.

- *reducer*: a function of the form `function(acc,val[,obj[,wc]])`, which returns the next value of the accumulator (which must be of the same type as *acc* and *val*) and with:
  - *acc*: the value of the accumulator, initially set to *init*
  - *val*: the value of the next element of the DA on which `reduce()` operates
  - *obj*: the same parameter *obj* passed to `reduce()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *init*: the initial value of the accumulators that are used by *reducer()*. It should be the identity element of the operation (i.e. applying it through the function should not change result).

- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA
- *done*: a callback of the form `function (error, result)` which is called at completion. If *undefined*, `reduce()` returns an ES6 promise.

Example:

```
uc.parallelize([1, 2, 4, 8]).
   reduce((a, b) => a + b, 0).
   then(console.log);
// 15
```

3.2.22 da.reduceByKey(reducer, init[, obj])

- *reducer*: a function of the form `callback(acc,val[,obj[,wc]])`, returning the next value of the accumulator (which must be of the same type as *acc* and *val*) and where:
  - *acc*: the value of the accumulator, initially set to *init*
  - *val*: the value `v` of the next `[k,v]` element of the DA on which `reduceByKey()` operates
  - *obj*: the same parameter *obj* passed to `reduceByKey()`
  - *wc*: the worker context, a persistent object local to each worker, where user can store and access worker local dependencies.
- *init*: the initial value of accumulator for each key. Will be passed to *reducer*.
- *obj*: user provided data. Data will be passed to carrying serializable data from master to workers, obj is shared amongst mapper executions over each element of the DA

When called on a DA of type `[k,v]`, returns a DA of type `[k,v]` where the values of each key are aggregated using the *reducer* function and the *init* initial value.

Example:

```
uc.parallelize([[10, 1], [10, 2], [10, 4]]).
   reduceByKey((a,b) => a+b, 0).
   collect().on('data', console.log);
// [10, 7]
```

3.2.23 da.rightOuterJoin(other)

When called on source DA of type `[k,v]` and *other* DA of type `[k,w]`, returns a DA of type `[k, [v, w]]` pairs where the key must be present in the *source* DA.

Example:

```
var da1 = uc.parallelize([[10, 1], [20, 2]]);
var da2 = uc.parallelize([[10, 'world'], [30, 3]]);
da1.rightOuterJoin(da2).collect().on('data', console.log);
// [ 10, [ 1, 'world' ] ]
// [ 30, [ null, 2 ] ]
```

3.2.24 da.sample(withReplacement, frac, seed)

- *withReplacement*: *Boolean* value, *true* if data must be sampled with replacement
- *frac*: *Number* value of the fraction of source DA to return
- *seed*: *Number* value of pseudo-random seed

Returns a DA by sampling a fraction *frac* of source DA, with or without replacement, using a given random generator *seed*.

Example:

```
uc.parallelize([1, 2, 3, 4, 5, 6, 7, 8]).
   sample(true, 0.5, 0).
   collect().toArray().then(console.log);
// [ 1, 1, 3, 4, 4, 5, 7 ]
```

3.2.25 da.subtract(other)

Returns a DA containing only elements of source DA which are not in *other* DA.

Example:

```
var da1 = uc.parallelize([1, 2, 3, 4, 5]);
var da2 = uc.parallelize([3, 4, 5, 6, 7]);
da1.subtract(da2).collect().on('data', console.log);
// 1 2
```

3.2.26 da.union(other)

Returns a DA that contains the union of the elements in the source DA and the *other* DA.

Example:

```
var da1 = uc.parallelize([1, 2, 3, 4, 5]);
var da2 = uc.parallelize([3, 4, 5, 6, 7]);
da1.union(da2).collect().toArray().then(console.log);
// [ 1, 2, 3, 4, 5, 3, 4, 5, 6, 7 ]
```

3.2.27 da.values()

When called on source DA of type `[k,v]`, returns a DA with just the elements `v`.

Example:

```
uc.parallelize([[10, 'world'], [30, 3]]).
   keys.collect().on('data', console.log);
// 'world'
// 3
```

4 References

ES6 promise https://promisesaplus.com

readable stream https://nodejs.org/api/stream.html#stream_class_stream_readable