Contents

1 Ugrid Reference **1**

1.1 Overview 1

1.2 Working with Distributed Arrays 2

1.3 Ugrid module 3

1.3.1 ugrid.context([config]) 3

1.3.2 Distributed Arrays methods 4

1 Ugrid Reference

- Overview
- Working with Distributed Arrays
- Ugrid module
  - ugrid.context([config])
    - uc.end()
    - uc.parallelize(array)
    - uc.textFile(path)
    - uc.lineStream(input_stream)
    - uc.objectStream(input_stream)
  - Distributed Arrays methods
    - da.cartesian(other)
    - da.coGroup(other)
    - da.distinct()
    - da.filter(filter[,obj])
    - da.flatMap(flatMapper[,obj])
    - da.flatMapValues(flatMapper[,obj])
    - da.groupByKey()
    - da.intersection(other)
    - da.join(other)
    - da.keys()
    - da.values()
    - da.leftOuterJoin(other)
    - da.rightOuterJoin(other)
    - da.map(mapper[,obj])
    - da.mapValues(mapper[,obj])
    - da.reduceByKey(reducer[,obj])

1.1 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 consist 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
```

1.2 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]]`

Transformation Name	Description	in	out
cartesian(other)	Cartesian product with the other DA	v w	[v,w][coGroup(o
filter(func)	Return a DA of elements on which function returns true	v	w
flatMap(func)	Similar to map(), but where function returns a sequence of elements	v	w
groupByKey()	Group values with the same key	[k,v]	[k,[v]][intersecti
join(other)	Perform an inner join between 2 DAs	[k,v]	[k,[v,w]][leftOut
map(func)	Return a DA where elements are passed through a function	v	w
mapValues(func)	Similar to map(), but where function is applied to value of key-value DA	[k,v]	[k,w][reduceByK
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:

Action Name	Description	out
aggregate(func, func, init)	Similar to reduce() but may return a different type	value
collect()	Return the content of DA	strea
count()	Return the number of elements from DA	numb
countByKey()	Return the number of occurrences of elements for each key in a `[k,v]` DA	strea
lookup(k)	Return the list of values `v` for key `k` in a `[k,v]` DA	strea
reduce(func, init)	Apply a function against an accumulator and each element of DA, return a single value	value

1.3 Ugrid module

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

1.3.1 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();
```

1.3.1.1 uc.end()

Closes the connection to the cluster.

1.3.1.2 uc.parallelize(array)

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

Example:

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

1.3.1.3 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);
```

1.3.1.4 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.

Example:

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

1.3.1.5 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);
```

Users may also *persist* a DA in memory, allowing efficient reuse accross parallel operations.

1.3.2 Distributed Arrays methods

Transformations are methods of the DA class. They all operate on a DA and return a new DA, so they can be chained. A transformation can take the following parameters:

- A DA callback function, called for each element. The helper function must be self-contained, or rely on dependencies accessible through the worker context (see below).
- An additional data object, which will be passed to the helper function. Those data must be serializable (it must be possible to apply `JSON.stringify()` on it)

DA callback function has a form of `function helper(element, [[data] [, wc]])`, where:

- *element* is the next element of the DA on which the transformation operates.
- *data* is the user additional data as passed to the transformation. It must be serializable.
- *wc* is the worker context, a global object defined in each worker and persistent accross transformations. It can be used to extend the worker capabilities through `wc.require()`.

Example:

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

function mapper(element, data, wc) {
    if (!wc.maxmind) wc.maxmind = wc.reqire('maxmind');
    return wc.maxmind.getCountry(element);
}
var res = uc.parallelize(vect).map(mapper).collect();
```

Following is the detailed description of each transformation.

1.3.2.1 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, 3, 4]);
var da2 = uc.parallelize(['a', 'b', 'c']);
da1.cartesian(da2).count().then(console.log);
```

1.3.2.2 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);
```

1.3.2.3 da.distinct()

Returns a DA where duplicates are removed.

Example:

```
uc.parallelize([ 1, 2, 3, 1, 4, 3, 5 ]).
   distinct().
   collect().on('data', console.log);
```

1.3.2.4 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);
```

1.3.2.5 da.flatMap(flatMapper[,obj])

- *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

Applies the provided mapper function to each element of the source DA and returns a new 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);
```

1.3.2.6 da.flatMapValues(flatMapper[,obj])

- *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

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.

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);
```

1.3.2.7 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 ] ]
```

1.3.2.8 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();
// 3 4 5
```

1.3.2.9 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' ] ]
```

1.3.2.10 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
```

1.3.2.11 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
```

1.3.2.12 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 ] ]
```

1.3.2.13 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 ] ]
```

1.3.2.14 da.map(mapper[,obj])

- *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

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

The following example program

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

function mapper(data, obj) { return data * obj.scaling }

var res = uc.parallelize([1, 2, 3, 4]).
     map(mapper, {scaling: 1.2}).
     collect();

res.on('data', console.log);
res.on('end', uc.end);
```

will display

```
1.2
2.4
3.6
4.8
```

1.3.2.15 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 [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.

Example:

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

1.3.2.16 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]
```

1.3.2.17 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 ]
```

1.3.2.18 da.subtract(other)

1.3.2.19 da.union(other)

Supported transformations, not yet documented:

- sample(withReplacement, frac, seed)
- groupByKey()
- reduceByKey()
- union()
- join(other)
- leftOuterJoin(other)
- rightOuterJoin(other)
- coGroup(other)
- crossProduct(other)
- intersection(other)
- subtract(other)
- keys()
- values()

Actions:

- aggregate()
- reduce()
- collect()
- count()
- forEach()
- lookup(key)
- countByValue()
- countByKey()