River Trail API (draft specification v2)

Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram

Intel Corporation

March 7, 2012

Introduction

This proposal describes a gentle extension to EcmaScript that will enable access to the parallelism found in all modern processors.

Goals

The goal of this proposal is to enable data-parallelism in web applications. Browser applications and in particular EcmaScript often need to leverage all available computing resources to provide the best possible user experience. Today web applications do not take full advantage of parallel client hardware due to the lack of appropriate programming models. This proposal puts the parallel computing power of client's hardware into the hands of the web developer while staying within the safe and secure boundaries of the familiar EcmaScript programming paradigm. It gently extends EcmaScript with simple deterministic data-parallel constructs that enable runtime translation to a low-level hardware abstraction layer. By leveraging multiple CPU cores and vector instructions, River Trail achieves significant speedup over sequential EcmaScript.

Proposal

Design approach

Three Pillar Approach

The design of RiverTrail is based on three pillars:

- A type called ParallelArray that holds data values
- Several prototypical methods of ParallelArray that implement parallel constructs like map
- The concept of an elemental function which is passed to the parallel constructs and typically returns a single data element.

We have chosen a set of parallel constructs that we feel is minimal and upon which other data parallel constructs can be built. For example sum would be implemented using the reduce construct while prefix sum would be implemented using the scan construct. We anticipate useful libraries and infrastructure being built upon these constructs. This approach enables a "do few things well" implementation strategy while ensuring the composability needed to build other abstractions.

ParallelArray Data Structure

We add ParallelArray, a new data type, to EcmaScript. ParallelArray is a read only array-like data structure that is created by a call to a constructor or is returned from a call to one of the ParallelArray prototype methods. By making ParallelArray immutable we can guard against race conditions. In addition to normal constructors we also support comprehensions where a ParallelArray object can be created by specifying an iteration space and providing a function that maps indices to values.

Elemental Functions

Similar in spirit to the use of kernel functions and callback functions our approach makes use of elemental functions written in EcmaScript. Any EcmaScript function can be used as long as it has an appropriate signature and is side effect free, i.e., the function does not mutate global state. Violations of the latter property will result in an exception.

The system will determine if the elemental function can be optimized to take advantage of data parallel hardware. If optimization is not possible the function is simply executed sequentially. Providing an elemental function that can be optimized is the responsibility of the developer though there are a few hints that will serve the programmer well. Recursion is an example of a typical EcmaScript construct that is likely to defeat optimization. Sequences of arithmetic operations on ParallelArrays that hold homogeneous data, e.g. ParallelArray objects that contain only floating point numbers, are good candidates for optimization.

EcmaScript provides the object oriented concept of a `this` variable. As a general rule, we define `this` within elemental functions to be the ParallelArray object associated with the called method.

Prototype Methods

ParallelArray comes with the following five data parallel methods that map ParallelArrays to ParallelArrays: `map`, `combine`, `scan`, `filter`, and `scatter`. When combined with elemental functions each of these methods creates a freshly minted ParallelArray. ParallelArray also includes a sixth data parallel method, `reduce,` which maps a ParallelArray to a single value. We believe that with these six methods one can create a very robust and complete data parallel library that covers a large number of applications. Since ParallelArrays are immutable we do not provide destructive array methods such as `push`, `pop`, `shift`, or `unshift`.

Example

This simple example creates a three element ParallelArray `myPA` using `new`. It then uses the prototype method `map` and the elemental function `function(element){return element+1;}` to create a freshly minted ParallelArray `myPlusPA` with each element in `myPA` incremented by 1.

```
myPA = new ParallelArray([1, 2, 3]);                            // <1, 2, 3>
myPlusPA = myPA.map(function(element){return element+1;});      // <2, 3, 4>
```

API

ParallelArray

The ParallelArray prototype is the central data structure around which River Trail programs are built. A ParallelArray data structure can be constructed using the following three constructor forms:

Synopsis

```
ParallelArray();
```

No arguments: return an empty ParallelArray

```
ParallelArray(anArray);
```

Argument 0 is array-like and the only argument: Use the values in the array-like argument to populate the new ParallelArray. Array-like is defined as having a length attribute and having enumerable properties from 0 to length -1. If elements are missing, the undefined value is used. Typed arrays are considered array-like.

```
ParallelArray(size, elementalFunction);
```

Argument 1 is an instance of a function (the elemental function), argument 0 is the size of the resulting array: Return a ParallelArray of size `size` where each value is the result of calling the elemental function with the index where its result is placed. If `size` is a number, then the index passed to the elemental function will be a number, as well. To support the construction of multi-dimensional arrays, `size` furthermore can be a one dimensional array. In such case, the elemental function will take multiple index arguments up to the length of the array `size`.

Returns

A freshly minted ParallelArray.

Throws

When the constructor is invoked with two or more arguments but `elementalFunction` is not a function.

map

Synopsis

```
myArray.map(elementalFunction, arg1, arg2, ...)
```

Arguments

`elementalFunction` described below
`arg1`, `arg2`, ... optional array-like objects that are indexed similar to myArray, which is available through the *`this`* variable

Elemental Function

```
function (val, val1, val2, ...)
```

`this` The entire ParallelArray
`val` An element from the ParallelArray
`val1, val2, ...` – values from arg1, arg2 and so forth from the same relative locations as val in *`this`*.
The result of the function will be used as an element to be placed in the result at the same offset we found `val` in the source array.

Returns

A freshly minted ParallelArray

Elements are the results of applying the elemental function to the elements in the original ParallelArray plus values from any optional arguments.

Throws

When `elementalFunction` is not a function.

Discussion

If an additional argument's length is greater than the original ParallelArray's length then the extra values are ignored. Any missing values will use the `undefined` value.

Unlike `combine`, `map` does not provide a depth argument to steer the number of dimensions the map operation iterates over. Instead, the dimensionality of the source array can be modified. Using a helper function `flatten`, which collapses the outer two dimensions of an array into a single dimension, `map` can be applied to elements across multiple dimensions. The result can then be transformed to the original shape using `partition`, a helper function that splits dimensions by dividing elements into groups of a given size. For example, instead of

```
result = pa.map(2, f); // NOT LEGAL
```

where 2 would be the depth argument, the programmer can instead use

```
tmp = pa.flatten();
tmp = tmp.map(f);
result = tmp.partition(pa.shape[0]);
```

Note that this transformation does not work for `combine`. As combine exposes the iteration index to the elemental function, collapsing the iteration space to a single index would be observable from the elemental function.

Example: an identity function

`result = pa.map(function(val){return val;});`

Example: Scale Alpha X plus Y

Perform the DAXPY operation found in the vector library BLAS (Basic Linear Algebra Subprograms)

```
alpha = 2;  // some scalar scale factor (a number)
xPA = new ParallelArray(...);
yPA = new ParallelArray(...);
resultPA = xPA.map(function(x,y){return alpha*x + y;}, yPA);
               // x is an element of xPA, y is an element of yPA
```

combine

Synopsis

```
myArray.combine(elementalFunction)
myArray.combine(depth, elementalFunction)
```

Arguments

`depth` (optional) the number of dimensions traversed to access an element in *this*; the default is 1
`elementalFunction` described below

Elemental Function

```
function (index1, index2, ...)
```

`this` The ParallelArray
`index1, index2,...` Each index is a scalar value indicating where the result of the elemental function is placed. The number of indices is determined by the dimensionality of the iteration space, i.e., by the value of the `depth` argument. If `depth` is omitted or has the value 1, only a single index is passed. To write dimensionality independent code, the EcmaScript variable `arguments` can be used as the first argument to `get` to retrieve source values. Future version of EcmaScript may provide rest parameters which are likely to be more convenient.

The result is used as an element to be placed in `combine`'s result at the location indicated by `index1, index2, . . .`

Returns

A freshly minted ParallelArray containing elements are the results of applying the elemental function.

Throws

When `elementalFunction` is not a function and if `depth` indicates an iteration space that is not available in the ParallelArray.

Discussion

`Combine` is similar to `map`, except an index is provided. This allows elemental functions to access elements from other source arrays relative to the one at the current index position. While any element in the source arrays can be accessed the result returned by the elemental function will be placed at the location indicated by the index.

Example: an identity function

`result = pa.combine(function(i){return this[i];})`

reduce

Synopsis

```
myArray.reduce(elementalFunction)
```

Arguments

`elementalFunction` described below

Elemental Function

```
function (a, b)
```

`this` The entire ParallelArray
`a, b` Arguments to be reduced and returned

Returns

The final value, if the ParallelArray has only 1 element then that element is returned.

Throws

When `elementalFunction` is not a function and if the source ParallelArray object is empty.

Discussion

Reduce is free to group calls to the elemental function and reorder the calls. For an elemental function that is associative and commutative the final result will be the same as reducing from left to right. Modular addition of integers is an example of an associative and commutative function and in this case the sum of a ParallelArray will always be the same regardless of the order that `reduce` calls the addition operator. On the other hand, averaging is an example of a non-associative function. The expression Average(Average(2, 3), 9) produces the value 5 2/3 while the expression Average(2, Average(3, 9)) produces the value 4. `reduce` is permitted to chose whichever call ordering it finds convenient.

`reduce` is only required to return a result consistent with some call ordering and is not required to chose the same call ordering on subsequent calls. Furthermore, `reduce` does not magically resolve problems related to overflow and the well documented fact that some floating point numbers are not represented exactly in EcmaScript and the underlying hardware so floating point addition and multiplication are not truly associative.

`reduce` also requires the elemental function be commutative since `reduce` can induce reordering of the arguments passed to the elemental functions.

Typically the programmer will only call `reduce` with associative and commutative functions but there is nothing preventing them doing otherwise. Calling `reduce` with a non-associative and/or non-commutative function will lead to a result that is guaranteed only to be consistent with some ordering of applying the elemental function on some ordering of the arguments.

scan

Synopsis

```
myArray.scan(elementalFunction)
```

Arguments

`elementalFunction` described below

Elemental Function

```
function (a, b)
```

`this` - the entire ParallelArray

`a, b` - arguments to be reduced and result returned

Returns

A freshly minted ParallelArray whose i-th element is the result of using the elemental function to reduce the elements between 0 and i, inclusively, in the original ParallelArray.

Throws

When elementalFunction is not a function.

Example: an identity function

pa.scan(function(a, b){return b;})

Discussion:

The construct implements what is known as an inclusive scan which means that the value of the i-th result is the same as what would be produced by `[0..i].reduce(elementalFunction).` Notice that the first element of the result is the same as the first element in the original ParallelArray. An exclusive scan can be implemented by shifting right by one the results of an inclusive scan, dropping the rightmost value, and inserting the identity at location 0. Similar to reduce, scan can reorder the calls to the elemental functions. Ignoring overflow and floating point anomalies, this cannot be detected if the elemental function is associative and commutative, in which case using an elemental function such as addition to create a partial sum will produce the same result regardless of the order in which the elemental function is called. However using a non-associative or non-commutative function can produce different results due to the ordering that `scan` calls the elemental function. While `scan` will produce a result consistent with a legal ordering the ordering and the result may differ for each call to `scan`.

Typically the programmer will only call `scan` with associative and commutative functions but there is nothing preventing them doing otherwise. Calling `scan` with a non-associative and/or non-commutative function will lead to a result that is guaranteed only to be consistent with some ordering of applying the elemental function.

scatter

Synopsis

```
myArray.scatter(indices, defaultValue, conflictFunction, length)
```

Arguments

`indices` array of indices in the resulting array
`defaultValue` optional argument indicating the value of elements not set by scatter. When not present, the default value is `undefined`
`conflictFunction` optional function to resolve conflicts, details below.
`length` optional argument indicating the length of the resulting array. If absent, the length is the same as the length of the original ParallelArray.

Returns

A freshly minted ParallelArray `A` where each element `A[i]` is defined as

- `A[indices[i]] = this[i]`, when `indices[i]` is unique
- `A[indices[i]] = conflictFunction(valA, valB)` when multiple elements are scattered to the same location. see below
- `defaultValue`, when `i` is not present in `indices` array

Example: an identity function

`result = pa.scatter(indices);` where `indices` is a ParallelArray where element === index

Handling conflicts with the Conflict Function

A conflict occurs when multiple elements are scattered to the same location. It results in a call to `conflictFunction`, which is an optional third argument to scatter. If `conflictFunction` is undefined, `scatter` throws an expection when a conflict occurs.

Synopsis

```
 conflictFunction(valA, valB)
```

Arguments

`this` the entire ParallelArray
`valA, valB` the two values that conflict

To ensure determinism it is the programmer's responsibility to provide a conflictFunction that is associative and commutative since there is no guarantee in what order the conflicts will be resolved.

Returns

Value to place in result[indices[index]]

Throws

The length of `indices` does not match the length of the ParallelArray.
When `conflictFunction` is neither `undefined` nor a function.
If a conflict occurs but no conflict function has been supplied by the programmer.
If `indices` contains an index greater than the result array's length.

Example: Resolve conflict with the larger value

```
function chooseMax(valA, valB){
      return (valA>valB)?valA:valB;
 }
pa = new ParallelArray([0,1,2,3,4,5]);
result = pa.scatter([0,3,1,4,2,5]);                  // <0,2,4,1,3,5>
result2 = pa.scatter([0,0,1,1,2,2], chooseMax);      // <1,3,5,undefined,undefined, undefined>
result3 = pa.scatter([0,0,1,1,2,2], chooseMax, 3);  // <1,3,4>
```

filter

Synopsis

```
myArray.filter(elementalFunction)
```

Arguments

`elementalFunction` described below

Elemental Function

```
function (index1, index2, . . .)
```

`this` The ParallelArray
`index1, index2, . . .` The indices specifying the location in *this* where the source element is found.

The result of the elemental function is interpreted as a truthy value. If the result equals `true`, the corresponding element will be included in `filter`'s result. Otherwise it will be omitted.

Returns

A freshly minted ParallelArray holding the source elements for which the results of applying the elemental function equals to `true`. The order of the elements in the returned ParallelArray is the same as the order of the elements in the source ParallelArray.

Throws

When elementalFunction is not a function.

Example: an identity function

`result = pa.filter(function(){return true;})`

flatten

Synopsis

```
myArray.flatten()
```

Arguments

none

Returns

A freshly minted ParallelArray whose outermost two dimensions have been collapsed into one.

Example

```
pa = new ParallelArray([[1,2][3,4]])   // <<1,2>,<3,4>>
pa.flatten()                           // <1,2,3,4>
pa3D = new ParallelArray(([[[1,2][3,4]],[[11,12][13,14]],[[11,22][23,24]]]);
                     // <<<1,2>,<3,4>>, <<11,12>,<13,14>>, <<11,22>,<23,24>>>
pa2D = pa3D.flatten(); // <<1,2>,<3,4>,<11,12>,<13,14>,<11,22>,<23,24>>
pa1D = pa2D.flatten(); // <1,2,3,4,11,12,13,14,11,22,23,24>
```

partition

Synopsis

```
myArray.partition(size)
```

Arguments

`size` the size of each element of the newly created dimension; the outermost dimension of `myArray` needs to be divisible by `size`

Returns

A freshly minted ParallelArray where the outermost dimension has been partitioned into elements of size `size`.

Example

```
pa = new ParallelArray([1,2,3,4])    // <1,2,3,4>
pa.partition(2)                     // <<1,2>,<3,4>>
```

Discussion

While one could implement both `flatten` and `partition` using the other constructs we call them out here to make it easy for the compiler to recognize `flatten` or `partition` and make optimizations easier.

Throws

When outermost dimension is not divisible by `size`.

[ ]

Synopsis

```
myArray[index];
```

Arguments

`index` a number value representing a valid index in the outermost dimension of `myArray`.

Returns

The value found at `index` or `undefined` if no such value exists. If `myArray` is a multi-dimensional ParallelArray object, [] returns a ParallelArray object that represents a slice of the source ParallelArray object at index `index`. All ParallelArray methods can be applied to such slice. In particular, `[]` can be used to further slice the ParallelArray or ultimately select single elements.

Example

```
pa = new ParallelArray([[0,1,2,3,4], [10,11,12,13,14], [20,21,22,23,24]])

pa.get[1][1];                // 11
pa[1];                       // <10,11,12,13,14>
```

Discussion

Since ParallelArrays are immutable using `[]` as part of the left hand side of an assignment is not allowed and results in a throw.

get

Synopsis

```
myArray.get(indices);
```

Arguments

`indices:` an array of number values that represent valid indices into `myArray`. The first index references the outer most dimension, the second index references the next dimension and so forth.

Returns

The value found at the indices or `undefined` if no such value exists.

Throws

If `indices` is not an array like object or if the length of `indices` is larger than the number of dimensions in the source array.

Example

```
pa = new ParallelArray([0,1,2,3,4], [10,11,12,13,14], [20,21,22,23,24])

pa.get([1,1]);                    // 11 same as pa[1][1].
pa.get([1]);                      // <10,11,12,13,14>, same as pa[1].
```

length

Synopsis

```
myArray.length
```

Returns

The toplevel (first dimension) length of the ParallelArray.

Example

```
pa = new ParallelArray([1,2,3,4])      // <1,2,3,4>
pa.length                              // 4
```

shape

Synopsis

```
myArray.shape
```

Returns

An Array containing the length of each dimension of the ParallelArray starting with the outermost dimension.

Discussion

pa.shape.length gives the dimensionality of the parallel array.

Example

```
pa = ParallelArray([[1,2,3],[4,5,6]])            // <<1,2,3>,<4,5,6>>
pa.shape                                          // [2, 3]
```

Implementations as non-performant Library

This has been implemented as a library available on GitHub.com/rivertrail/rivertrail/. There is a prototype implementation of a compiler that compiles this proposal into OpenCL and demonstrates an up to 10x speedup on current hardware.

The River Trail code on GitHub can be used to experiment with the API and over the past year we have learnt a lot about what the API should look like and a lot about how to simplify the API. What you see here is the result of that ongoing work.

Discussions and back stories

Immutability discussion

ParallelArray is a separate type and is not a subtype of Array. This section distills the arguments that lead to this decision.

1. A dedicated type simplifies the detection of arrays that will take part in parallel execution and thus eases optimizations, e.g., (speculative) use of different storage layout and lazy evaluation of the constructs.
2. Extending Array and the various typed arrays would create an API that has to be kept in sync across different prototypes. Using a dedicated object would allow us to use inheritance between ParallelArray and TypedParallelArray, if we choose to go that route
3. ParallelArray is multidimensional and rectangular whereas Arrays traditionally are not.
4. ParallelArray allows for a clear separation of APIs: Array is always sequential whereas ParallelArray is parallel, the former allows mutation in lambdas while the latter does not. Otherwise we would end up with map and pmap with different signatures and semantics, which is awkward.

Extra arguments vs. free variables discussion

The current implementation on GitHub passes any extra (… rest) arguments given to the ParallelArray method on to the elemental function. In this document we describe an API that instead relies on EcmaScript's ability to create closures with access to free variables to accomplish the functionality extra arguments typically supply. The new pattern is simple. If one wants to write a program that adds some number, in this case 4, to every element in a ParallelArray it could be accomplished as follows:

```
function addN(increment) {
    return function (val) {return val+increment;};
};
pa = new ParallelArray([1,2,3,4])    // <1,2,3,4>
pa.map(addN(4));                     // <5,6,7,8>
```

The current implementation on GitHub is unable to properly recognize free variables so instead it allows the passing of extra arguments to the parallel constructs which are in turn passed to the elemental functions. The use of extra arguments and passing scalar indices to elemental function creates ambiguities so we decided to eliminate the use of extra arguments. Below is an example of how the GitHub implementation would accomplish the above.

```
function addNBrokenLegacy(val, increment) {
    {return val+increment;};
};
pa = new ParallelArray([1,2,3,4])   // <1,2,3,4>
pa.map(addNBrokenLegacy, 4);                    // <5,6,7,8>
```

Fallback to sequential vs. throwing an exception

In a development environment we probably want to throw or at least log why parallel optimizations were inhibited. In a deployment environment we probable want to fall back to a sequential implementation.

Extending ParallelArray and implicit flattening

This is a difficult problem if we want to optimize for multidimensional arrays. For example if we have a 2D array but the inner dimension holds ParallelArrays that have been extended, say with an attribute color, then if a straight forward implementation flattens the data the attribute will be lost.

GPU, threads, vector instructions discussion

Currently many GPUs are unable to deal gracefully with programs that are long running. This vulnerability is particularly troublesome in web applications since bugs or malicious code can result in denial of service (DOS) attacks. If long running code is dealt with at all it is by using web inappropriate heavy duty clubs, such as resetting the GPU. While running them on the CPU does not in any way bound the algorithms runtime, CPUs do provide context switching which can be used to manage the amount of time the HW is dedicated to the data parallel algorithm. Since vector instruction and multiple cores provide context switching today it is appropriate that current implementations focus on this hardware. This will provide an opportunity to developers to realize the power of current hardware without worrying about DOS attacks while knowing that future version of River Trail will be able to take advantage of GPUs once (if) they eventually provide more appropriate ways to deal with resource allocation and sharing.