Using algorithmic skeletons in EcmaScript to parallelize computation in browsers through Web Workers

Damian Schenkelman and Matias Servetto

School of Engineering. University of Buenos Aires.
Paseo Colon 850, Buenos Aires, Argentina
{dschenkelman,mservetto}@fi.uba.ar
http://www.fi.uba.ar/

Abstract. With the growing tendency of browsers being used for CPU intensive applications (such as 3D games) it is important to take advantage of multiple cores to reduce user perceived latency. This paper presents a JavaScript library based on algorithmic skeletons that allows users to take advantage of parallel processing in browsers while maintaining a familiar coding style.

Keywords: browsers, ecmascript, parallelization, algorithmic skeletons

1 Introduction

In 1965 Gordon Moore proposed the popularly known "Moore's law" [1], which predicts that the number of transitors in integrated circuits will double every two years. As a consequence, programs that use a single processor/core will automatically become faster without the need to modify them with that purpose in mind.

In the year 2010 it was predicted that this tendency would be slowly reaching its end, mostly due to reasons related to heat dissipation. This is the reason why newer computers have a greater number of processors instead of processors with more computing power.

To be able to maximize the performance of these computers, it is paramount that systems professionals can create programs that process data in parallel, through different mechanisms, such as threads or processes executing simultaneously in different processors. In the case of applications executed on a single device (e.g. browsers and web pages, applications for mobile devices, desktop apps, server applications) a lot of languages and platforms provide a simple way for programmers to abstract the complexity and coordination required for this type of processing. For example, the .NET platform has the Task Parallel Library (TPL) and Parallel LINQ (PLINQ) [2] libraries that use algorithmic skeletons modelled with high order functions so developers can perform complex operations through a simple API.

On a different note, one of the tendencies that is beginning to take off is the use of EcmaScript [3] (also known as Javascript) to create applications, such as video games, that were previously only considered viable in a native environment. This has been possible thanks to components like asm.js [4] and WebGL [5] together with the constant evolution of browsers and JavaScript runtime engines. In this context, one of the plans of the commitee that develops the language is to provide an API to simplify the processing of data in parallel for version 7 (ES7) of the language, with the goal of keeping up with native applications. Initiatives such as ParallelJS [7] and River trail [6] or the possibility of taking advantage of SIMD instructions [8] (single instruction multiple data) are some of the options to implement this.

The goal of this paper is to provide an alternative for parallel code execution in EcmaScript, through a library that exposes high order functions to model algorithmic skeletons such as map, filter and reduce. This library will take advantage of Web Workers [9] (the mechanism proposed by browsers for parallelism at the time of writing) to allow the simultaneous execution of code in multiple processors.

This paper is organized as follows: Section 2 provides some context on the current state of thread-base parallelization in JavaScript. Section 3 provides an overview of how the library works. Section 4 explains the challenges the library faces, the alternative solutions and the decisions taken. Section 5 compares the performance of using the library against a sequential approach for an image transformation. Finally, Section 6 presents future research directions.

2 Background

All modern versions of major browsers (Internet Explorer, Mozilla Firefox, Google Chrome, Safari and Opera) allow the execution of JavaScript code in a single threaded event loop runtime [10]. User actions add elements to the event loop queue and these are processesed sequentially. All process input/output is asynchronous to prevent the UI thread from blocking, thus keeping it responsive. While such a runtime is a good fit for most web applications, JavaScript and the browser are now also being used for some CPU intensive tasks, such as games, which commonly require physics calculations, image manipulation (in 2D) and graphics generation (in 3D). Despite the large improvement in JavaScript's performance, obtained through optimizing compilers such as Google Chrome's V8 [11] and Mozilla Firefox's SpiderMonkey [12], it is important to be able to take advantage of multiple cores in modern devices to achieve even greater performance.

In 2010, Web Workers were made part of the web standard. Web Workers allow the creation of "thread like" constructs in a browser environment, but they don't allow shared memory and instead communicate via message passing. The message passing overhead is very big for common objects and Web Workers "have a high start-up performance cost" [9], so they were commonly considered for long running tasks and operating on generally the same set of data during a single

execution, making them unfit for a thread pool model [13].
However, a spec is being developed for version 7 of the EcmaScript standard that changes this, by introducing shared memory through `SharedArrayBuffer`s [14]. In essence, the proposal allows the same memory to be shared across multiple Web Workers, and also aims to provide the necessary atomic/lock constructs to deal with shared memory. In the particular case of embarrassingly parallel computations (such as map, filter and reduce operations on an array), it is simple to take advantage of the speed benefits of shared memory without incurring in any overhead due to synchronization.

3 Library Overview

JavaScript developers are familiar with the concept of map, filter and reduce skeletons as language constructs. In JavaScript, `Array`s have `map`, `filter`, and `reduce` functions and version 6 of the EcmaScript adds those same methods to `TypedArray`s [15]. For example, to multiply all the elements of an `Array` by two:

plain JS map sample

```
var newValues = [1,2,3,4].map(function(e){
  return e * 2;
});

// newValues is a new array [2,4,6,8]
```

(Example creating a new array with the values of the original one multiplied by two)

Our library (p-j-s) only works with `SharedTypedArray`s (and `TypedArray`s) due to performance reasons discussed in Section 4. It aims to provide a similar interface while not changing the native object's prototypes and dealing with the asynchronous nature of the operation. Given a `SharedTypedArray`, the following code would do the same as the previous example but parallelizing the work:

p-j-s map sample

```
var pjs = require('p-j-s');
pjs.init({ maxWorkers: 4 });
var xs = new SharedUint8Array(4);
xs.set([1,2,3,4]);
pjs(xs).map(function(e){
  return e * 2;
}).seq().then(function(newValues){
  // newValues is a new SharedUint8Array [2,4,6,8]
});
```

(Example creating a new array with the values of the original one multiplied by two using p-j-s)

The call to `pjs.init` initializes the library and the amount of workers to be used for processing (a worker pool is created with them). In browsers that have an API to detect the amount of available cores in a machine, the `maxWorkers` value is limited by the amount of cores.

The API provided by p-j-s [16] is very similar to the synchronous one that is provided by the language. The asynchronous nature of the operation is reflected by the fact that the call to `seq` returns a `Promise` (alternatively a callback could be passed as a parameter to it).

It is possible to chain multiple operations before calling `seq` to avoid passing data back and forth between the Web Workers and the UI thread. Before `seq` is invoked, an operation is a chain of one or more steps. From the same main chain, multiple chains could be created:

p-j-s chaining sample

```
var xs = new SharedUint8Array(4);
xs.set([1,2,3,4]);

var chain = pjs(xs).map(function(e){
  return e * 3;
});

var evenChain = chain.filter(function(e){
  return e % 2 === 0;
});

var sumChain = chain.reduce(function(c, v){
  return c + v;
}, 0, 0);

evenChain.seq().then(function(evens){
  // evens is [6, 12]
  sumChain.seq().then(function(sum){
    // sum is [3,6,9,12]
  });
});
```

(Example creating multiple chains from the same base chain)

4 Implementation

Let's consider an array of `N` elements on which a particular transformation is to be performed through the `map` function. If the transformation is executed using a single thread (no parallelism) and the average time to process each element is `t` then the total time (T_{ser}) for the operation can be approximated as:

$$T_{ser} = \sum_{i=0}^{N} t = Nt \ . \tag{1}$$

When trying to parallelize this operation using K threads the ideal goal is to reach a total time (T_{par}) that is:

$$T_{par} = \frac{T_{ser}}{K} \ . \tag{2}$$

Nevertheless, there are additional time consuming tasks other than the main computation that need to be considered when performing the operation in parallel in programs where not all memory is shared. These are:

- Serializing/deserializing the elements to transfer.
- Transferring the elements back and forth between the UI thread and the workers.
- Serializing/deserializing the function to transfer.
- Transferring to each worker the functions for the transformation.

If we drill down into the different parts:

- T_{ft} as the function transfer time
- T_{et} as the elements transfer time
- T_{fs} as the function serialization/deserialization time
- T_{es} as the elements serialization/deserialization time

It is clear that:

$$T_{sync} = T_{ft} + T_{et} + T_{fs} + T_{es} \ . \tag{3}$$

$$T_{par} \approx \frac{T_{ser}}{K} + T_{sync} \ . \tag{4}$$

Based on Eq. 3 and Eq. 4 it can be deduced that the more T_{sync} can be reduced, the closer to the ideal scenario the computation will be.

In our case we are trying to transfer objects between a browser's JavaScript UI thread and Web Workers so we are constrained by the means of that environment. The Worker interface is the following [9]:

Web Worker interface

```
[Constructor(DOMString scriptURL)]
interface Worker : EventTarget {
  void terminate();

  void postMessage(any message, optional sequence<Transferable> transfer);
  [TreatNonCallableAsNull] attribute Function? onmessage;
};
Worker implements AbstractWorker;
```

(Example extracted from the W3C Web Workers specification)

As the aforementioned interface states one can either send just a message or send a message with a sequence of transferable objects. From section **2.7.5** of the HTML Standard [18]: *"Some objects support being copied and closed in one operation. This is called transferring the object, and is used in particular to transfer ownership of unsharable or expensive resources across worker boundaries."*

4.1 Elements

Considering the definition of a `Transferable`, it seems like a good alternative to minimize both T_{et} and T_{es}. Even more so when one considers that otherwise objects are copied using structured cloning (explained in section **2.7.6** of that same standard).

To verify our hypothesis, we put together two benchmarks[1] that aim to verify the difference between invoking `postMessage` with and without structured cloning for a `SharedTypedArray`. Both transfer a `SharedTypedArray` back and forth between the UI thread and a worker; the only difference between the two is that one[2] has 100,000 (a hundred thousand) elements in the `SharedTypedArray` and the other one[3] 1,000,000 (a million). As it can be seen from the results in Tab. 2, increasing the amount of elements by 10x does not change the amount of operations that can be performed when using `Transferable` objects, it scales. On the other hand, the amount of operations that can be performed with structured cloning greatly decreases. For that reason our library only works with `SharedTypedArray`s and `TypedArrays`.

Computer	Mac Book Air
Processor	Intel Core i5
Clock Frequency	1.8 GHz
System Memory	4 GB
Operating System	OS X Yosemite 10.10.3
Browser version	Firefox Nightly 41.0a1

Table 1. Benchmarks environment.

Elements	Cloning [ops/sec]	Transferring [ops/sec]
$1E5$	734	**1596**
$1E6$	91	**1134**

Table 2. Difference between transferring and cloning shared buffers.

[1] All benchmarks in this document use the environment described in Tab. 1.
[2] http://jsperf.com/transferrable-vs-cloning/2
[3] http://jsperf.com/longer-transferrable-vs-cloning/2

4.2 Functions

The library must handle the distribution of the code between the different workers when an operation is executed. This is a challenging problem with many possible alternatives: Cloud Haskell [19] for example, requires a customized version of the Glasgow Haskell Compiler (GHC) to serialize closures and their captured variables as a code pointer and an environment, operating under the assumption that the same code is executed in all nodes. A proposal for using Scheme for distributed programming [20] uses a custom serializable data structure to represent procedures, and macros in order to capture a closure's captured variables and serialize their values. It also assumes that all nodes are executing the same program since serialized functions store pointers to parts of source code files.

In our case, modifying the runtime is not a viable alternative since we want our library to be usable by anyone creating applications for major browser versions.

A naive approach would be to try to transfer a function as part of the `message` parameter of `postMessage` but that is not an option (it throws an error).

Considering the fact that `Function` objects cannot be directly transferred a different serialization approach is to be considered a possible alternative is to:

1. Serialize the `Function` to a `String`.
2. Transfer the `String` to the worker.
3. Create a new `Function` on the worker from that `String`.

Sec. 4.3 proposes a solution to sharing additional data and functions between the UI thread and Web Workers as a replacement for variables captured by closures.

Serialization and deserialization One possible way of passing serialized `Function` to workers would be to encode them as binary and transfer them as an `ArrayBuffer`. To encode the strings there are two possible approaches:

- Using the Encoding API[21]
- Implementing non native encoding/decoding functions

The results for the benchmark[4] are in Tab. 3:

Native encoding API [ops/sec]	Non-native function [ops/sec]
5017	4398

Table 3. Difference encoding/decoding strings natively and in JavaScript.

[4] http://jsperf.com/pjs-serialization-comparison/2

Transference `Function`'s code not only needs to be serialized but also transferred. For that reason it was also worth comparing the time it takes to encode/decode a `String` and tranfer the resulting `ArrayBuffer` to the worker against just passing the `String` to the worker.

Additionally, in most scenarios code would have to be sent along with a `TypedArray` so it would also be interesting to see if the transfer time was affected by this fact.

We created benchmarks[5] using two functions whose string representation has different lengths to see if the function's length affected the serialization and transfer time (results in Tab. 4). We measured three approaches to transfer the string:

1. Encoding the `String` into an `TypedArray`, sending it as a `Tranferable` and decoding it on the Web Worker.
2. Sending the `String` directly.
3. Using a `Blob` to store the stream and retrieving that in the Worker.

Function length	Blob [ops/sec]	Copy [ops/sec]	Tranferable [ops/sec]
Short	1070	**6629**	5927
Long	1075	**3273**	2967

Table 4. Sending function strings from UI thread to Web Workers.

We created a separate benchmark[6] to compare the best option (copying) against encoding the function into a `SharedTypedArray` and transferring it to four workers. We expected the latter to be slower since it involves encoding the code once and decoding it in each worker. The results are shown in Tab. 5.

Copying [ops/sec]	SharedArrayBuffer [ops/sec]
1315	1226

Table 5. Sending code to four workers by copying compared to encoding it and using a SharedArrayBuffer.

We also created a benchmark[7] that tranfers a `SharedTypedArray` and a `String` that are properties of the same `message` object back and forth between the UI thread an a Web Worker to understand if transferring additional objects affected the transfer time when comparing it with just tranferring a `String` (results in Tab. 6).

[5] http://jsperf.com/pjs-serialization-long/3 and http://jsperf.com/pjs-serialization/3
[6] http://jsperf.com/shared-function-transfer
[7] http://jsperf.com/pjs-encoding/2

Copying code [ops/sec]	Tranferring code [ops/sec]
4509	4270

Table 6. Sending code and a TypedArray to a Web Worker.

4.3 Contexts

Functions in JavaScript can capture variables as long as they are within the variable's lexical scope. Functions that make use of this feature are closures. As explained in subsection 4.2, `Function` objects cannot be passed as messages to workers. This means that there is no out of the box mechanism to make variables that are available in the environment when the parallel computation is started accessible when executing the callback functions in each Web Worker.

To solve this problem, p-j-s introduces the *context* concept. There are two kinds of contexts:

Global context Shared across all chains.
Local context Specific to a particular step in a chain.

The following code snippet shows how developers can work with each type of context:

p-j-s context sample

```
var xs = new SharedUint8Array(4);
xs.set([1,2,3,4]);

pjs.updateContext({
  max: 3
}).then(function(){
  pjs(xs).filter(function(e, ctx){
    return e <= ctx.max && e >= ctx.min;
  }, {
    min: 2
  }).seq().then(function(range){
    // range is [2,3]
  });
});
```

(Example using local and global context for chains)

When executing each step in a chain, the local context is merged with the global context (neither is modified) into a separate object which is the `ctx` parameter that the function receives. If the same key is present in both the global and local context, the one in the local context is the one that will be available. Given the following code snippet, the available `ctx` for each function is displayed in Fig. 1.

p-j-s context merge sample

```
var xs = new SharedUint8Array(4);
xs.set([1,2,3,4]);

pjs.updateContext({
  A: 0,
  B: 25,
  C: 44
}).then(function(){
  var chain = pjs(xs)
    .map(function(e, ctx){
      // code here
    }, { A: 5, B: 13 })
    .filter(function(e, ctx){
      // code here
    }, { A: 2, E: 9 })
});
```

(Example for resulting ctx)

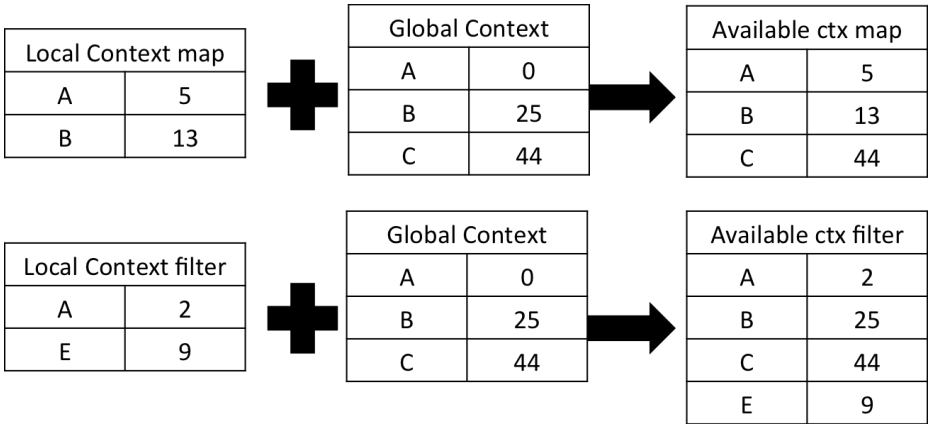

Fig. 1. Example of global and local context being merged for chain steps.

One alternative to hide the *context* concept from application developers could be to introduce a *build* step that detects the variables captured by p-j-s callbacks and transforms the code. By doing this, they wouldn't need to do anything out of the ordinary with regards to closures and their environment when working with p-j-s. However, this approach would require transforming the original code, and would be complex and time consuming to implement.

5 Experiment

It is important to compare the performance of our parallel implementation against a serial one, as some processing is required before the computation begins and after it ends to distribute the data and work and gather the results respectively, as explained in Section 4.

In this regard, the more complex the computation to be performed on each array element the better the parallel implementation will perform. The reason for this is that, in the parallel case, the cost of this processing is dwarfed by the cost of the actual computation.

5.1 Algorithm selection

For the experiment we decided to use an algorithm that applies a sepia tone effect to pictures. This meets the criteria of both being useful and having some calls to `Math.random` which make it more complex from a computational standpoint. The benchmark is publicly available [22].

5.2 Results

The results for the benchmark are the displayed in Tab. 7. Based on those results, we created a chart (Figure 2) that shows the relative performance of the parallel approach compared to the sequential one.

Elements	Serial [ops/sec]	Parallel [ops/sec]
100	**142223**	328
1,000	**16503.8**	359
10,000	**1683.8**	321.8
100,000	169.6	**184.4**
1,000,000	16.8	**34.1**

Table 7. Comparing p-j-s (parallel) and plain JS (serial) performance for an image transformation.

5.3 Conclusion

A couple of interesting findings result from the experiments. An expected one is that the amount of operations per second that can be performed with a serial implementation is almost inversely proportional to the amount of items to transform with a ratio of 1. On the other hand, this is not the case for the parallel approach.

An acceptable speedup of approximately **2x** can be achieved with a relatively large amount of items ($10E6$) and performing a computationally expensive computation (generating pseudo-random numbers). Larger speedups might

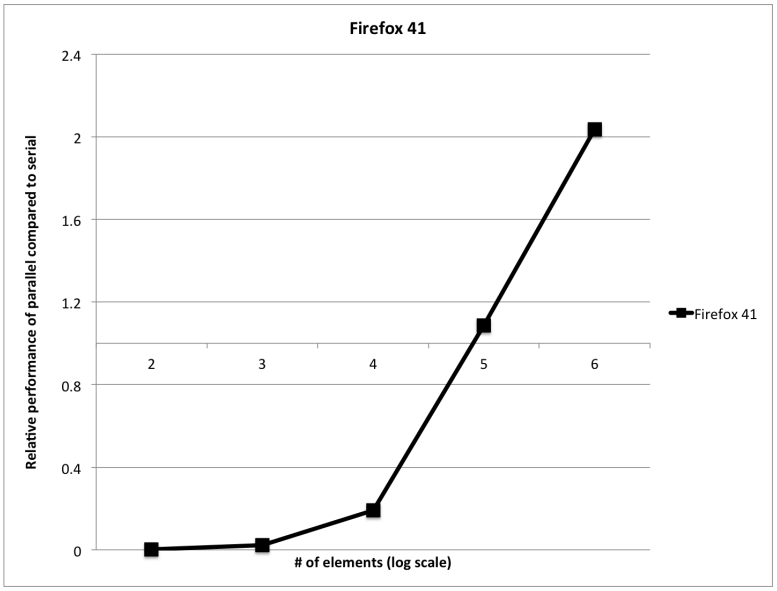

Fig. 2. Relative performance of parallel transformation compared to sequential one.

be possible by applying optimizations to the library and there might also be some potential improvements for operations with `SharedArrayBuffer`s as they are likely not optimized, being that they are only an experimental feature in Firefox Nightly.

6 Future Work

There are still some parts of the library that would benefit from optimizations. Of particular interest to us is to perform further profiling of the code executing in each Web Worker. As of today, using browser profiling tools for Web Workers is not as simple as doing so for the UI thread.

During our research we found that browser memory usage greatly increases when using shared memory after transforming an image multiple times. Memory is relased after a particular time period (which seems to indicate there is not a leak), but while memory usage is high, performance degrades. We plan to spend some time on this issue.

There is a strawman proposal to support typed objects [23] in a future version of EcmaScript, which in turn would allow `TypedArray`s of said typed objects. Once a browser has an experimental version available we intend to make sure that typed arrays of typed objects work with the library.

Once the `SharedArrayBuffer` type is available in other browsers (there is already a proposal for Google Chrome [24]) we plan to perform benchmarks and tests to compare with Firefox Nightly.

Acknowledgments. Thanks to Rosa Wachenhauzer, our tutor, for reviewing this paper and also for all her support and guidance throughout this entire project.

References

1. Moore, G.E.: Cramming More Components onto Integrated Circuits. (1965)
2. Parallel Programming in the .NET Framework, Microsoft Corp., `http://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx`
3. EcmaScript, ECMA, `http://www.ecmascript.org/`
4. asm.js, `http://asmjs.org/`
5. WebGL, Khronos Group, `http://www.khronos.org/webgl/`
6. Herhut, S., Hudson, R. L., Shpeisman, T., Sreeram, J.: River trail: A path to parallelism in javascript. SIGPLAN Not., 48(10):729–744, (2013).
7. Wang, J., Rubin, N., Yalamanchili, S.: River trail: ParallelJS: An Execution Framework for JavaScript on Heterogeneous Systems. SIGPLAN Not. (2014)
8. McCutchan, J., Feng, H., Matsakis, N. D., Anderson, Z., Jensen, P.: A SIMD Programming Model for Dart, JavaScript, and other dynamically typed scripting languages (2014)
9. Web Workers, W3C, `http://www.w3.org/TR/workers/`
10. Concurrency model and Event Loop, Mozilla Developer Network, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`
11. v8, Google, `https://developers.google.com/v8/intro`
12. SpiderMonkey, Mozilla, `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`
13. Schmidt, D. C., Vinoski, S.,: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool (1996)
14. JavaScript Shared Memory, Atomics, and Locks, `https://docs.google.com/document/d/1NDGA_gZJ7M7w1Bh8S0AoDyEqwDdRh4uSoTPSNn77PFk/edit`
15. TypedArray objects, ECMA, `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-typedarray-objects`
16. p-j-s API, `https://github.com/pjsteam/pjs/blob/dev/README.md`
17. Promise, MDN, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`
18. HTML Standard, WHATWG, `https://html.spec.whatwg.org`
19. Epstein, J., Black, A. P., Peyton-Jones, S.: Towards Haskell in the Cloud (2011)
20. Schwendner, A.: Distributed Functional Programming in Scheme (2010)
21. Encoding Standard, WHATWG, `https://encoding.spec.whatwg.org`
22. Sepia tone transformation benchmark comparison between p-j-s and plain JS, `http://jsperf.com/pjs-shared-vs-normal-vs-seq-100/4`
23. Typed Objects, ECMA, `http://wiki.ecmascript.org/doku.php?id=harmony:typed_objects`
24. Shared Array Buffers, Chromium Dashboard, `https://www.chromestatus.com/feature/4570991992766464`