University of Buenos Aires

Professional Assignment

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

Authors:
Damian Schenkelman
Matias Servetto

Tutor:
Prof. Rosa Wachenchauzer

June 1, 2015

1 Introduction

In 1965 Gordon Moore proposed the popularly known "Moore's law" [1], which predicts that the number of transistors 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 committee 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 [6] and River trail [7] 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.

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 processed 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" [13], 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 [14].

However, a spec is being developed for version 7 of the EcmaScript standard that changes this, by introducing shared memory through `SharedArrayBuffer`s [15]. 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.

We developed our library to work with instances of the already standarized `ArrayBuffer` type using the Google Chrome broswer. Additionally, we support the experimental `SharedArrayBuffer`, which at the time of writing is only available in Firefox Nightly[1].

[1]Current version is 41.0a1

3 Library Overview

JavaScript developers are familiar with the concept of map, filter and reduce skeletons as language constructs. In JavaScript, `Arrays` have `map`, `filter`, and `reduce` functions and version 6 of the EcmaScript standard adds those same methods to `TypedArrays` [16]. For example, to multiply all the elements of an `Array` by two:

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

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

Listing 1: Example creating a new array with the values of the original one multiplied by two

Our library (p-j-s) only supports `SharedTypedArrays` and `TypedArrays` due to performance reasons discussed in Section 6. 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:

```
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]
});
```

Listing 2: 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).[2]

The API provided by p-j-s [17] 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` [18] (alternatively a callback function could be passed as a parameter to it).

[2]More details in Section 5.

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:

```
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]
  });
});
```

Listing 3: Example creating multiple chains from the same base chain

4 About Microbenchmmarks

A common practice to measure the performance of a fragment of code, or to compare it to the performance of some other code fragment is to create a micro benchmark. Throughout this document, there are multiple references to benchmarks of this kind, since their results helped us make decisions about how to best implement some specific library features.

A micro benchmark usually involves wrapping the code under evaluation inside a loop, to:

First Be able to evaluate the time it takes to perform the operation multiple times (a single execution could either be too small or an outlier and the measurement could have too much variance/error).

Second Allow JIT optimizations to kick in.

The first point is applicable to all programming languages but the second one is only important in engines/VMs that perform such optimizations. This document discusses the performance of JavaScript and in particular v8, so the second point does affect our benchmarks.

4.1 A small example[3]

Let's assume that we want to profile the following function:

```javascript
function between(a, b, c){
  return a <= b && b <= c;
}
```

Listing 4: Function to benchmark

A naive benchmark for it could be:

```javascript
// begin setup
function between(a, b, c){
  return a <= b && b <= c;
}
var a = Math.round(Math.random()*100);
var b = Math.round(Math.random()*100) + a;
var c = Math.round(Math.random()*100) + b;
//end setup
```

[3]The example is based on [19].

```
//start benchmark
console.time('between');
for (var i = 0; i < 100000; i++){
  between(a, b, c);
}
console.timeEnd('between');
//end benchmark
```

Listing 5: Naive benchmark

The problem with the above benchmark is that it does not consider optimizations. Let's assume that the executing engine is smart enough to:

1. Inline functions

2. Detect loop invariants

Then due to #1 the above could result in:

```
// begin setup
var a = Math.round(Math.random()*100);
var b = Math.round(Math.random()*100) + a;
var c = Math.round(Math.random()*100) + b;
//end setup
//start benchmark
console.time('between');
for (var i = 0; i < 100000; i++){
  var j = a <= b && b <= c;
}
console.timeEnd('between');
//end benchmark
```

Listing 6: Naive benchmark after inlining

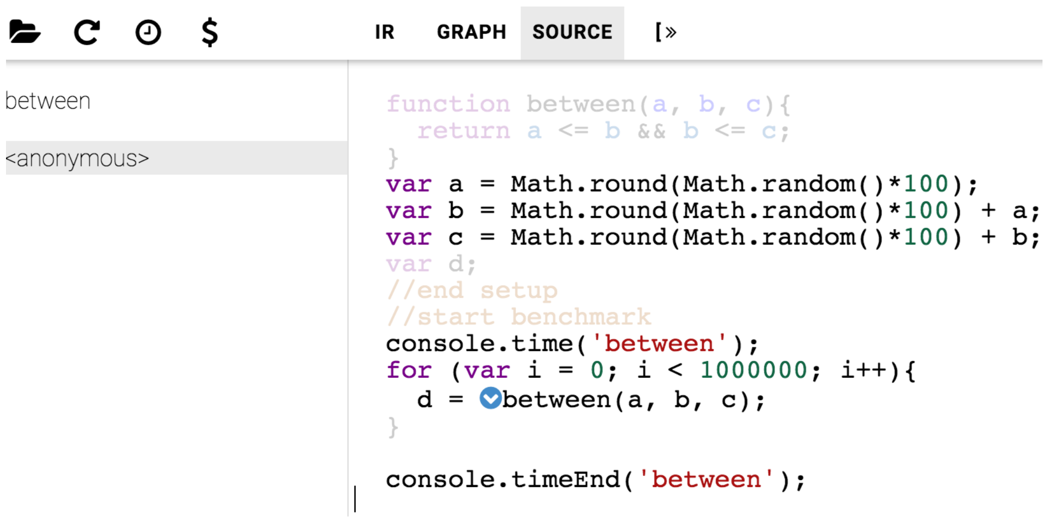
When it applies #2, it realizes it does not need to perform the comparison in every iteration so it could extract it outside the loop, resulting in

```
// begin setup
var a = Math.round(Math.random()*100);
var b = Math.round(Math.random()*100) + a;
var c = Math.round(Math.random()*100) + b;
//end setup
//start benchmark
console.time('between');
var j = a <= b && b <= c;
for (var i = 0; i < 100000; i++){
}
```

```
console.timeEnd('between');
//end benchmark
```

Listing 7: Naive benchmark after evaluating loop invariants

Alternatively it could notice that the variable j is not being used so it could just throw the line away. In any case, we would just be measuring how long it takes for JavaScript to execute the loop, which is not what we originally intended to do. To verify that is not the case the tool IRHydra [20] can be used , which *can display intermediate representations used by V8.* As shown in the following figure, v8 inlined the function (the blue chevron marks that) but the loop invariant was not detected and optimized:



```
function between(a, b, c){
  return a <= b && b <= c;
}
var a = Math.round(Math.random()*100);
var b = Math.round(Math.random()*100) + a;
var c = Math.round(Math.random()*100) + b;
var d;
//end setup
//start benchmark
console.time('between');
for (var i = 0; i < 1000000; i++){
  d = between(a, b, c);
}

console.timeEnd('between');
```

Figure 1: Optimized benchmark code with inlined function.

4.2 Details

The benchmarks are executed using two different machines:

Machine 1

- Mac Book Pro Retina
- Intel Core i7 2.8GHz
- 16 GB RAM DDR3 1600 MHz

- SSD 250 GB
- Chrome 41.0.2272

Machine 2

- Mac Book Air
- Intel Core i5 1.8GHz
- 4 GB RAM DDR3 1600 MHz
- SSD 128 GB
- Chrome 40.0.2214
- Firefox Nightly 41.0a1

Note: Each benchmark is executed 5 (five) times to reduce variance.

Note: No browser plug-ins are enabled when running the benchmarks as those could introduce additional sources of error.

5 Initialization

As explained in the Web Workers standard [13] *"Generally, workers are expected to be long-lived, have a high start-up performance cost, and a high per-instance memory cost."*.

There are two factors to consider from the above sentence regarding the library's implementation:

First The high start-up time.

Second The high per-instance memory cost.

5.1 High start-up time

This factor raises the issue of when workers should be instantiated. On relation to this, we performed a benchmark to get an idea of the difference in time taken to process messages. One scenario considers cold starting a Web Worker, and the other one having a Web Worker already available.

5.1.1 Cold start benchmark

The following is the code for the benchmark:

```
// http://jsperf.com/worker-cold-start/6

// HTML setup
<script>
  var __finish;

  function setupForPreExistentWorker(){
    var wCode = function(event){
      var codeBuffer = event.data;
      var codeArray = new Uint8Array(codeBuffer);
      var decoder = new TextDecoder();
      var code = decoder.decode(codeArray);

      eval('var __f = ' + code);

      __f();
    };

    var blob = new Blob([
        "onmessage = " + wCode.toString()]);

    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  }

  var encoder = new TextEncoder();

  function postCode(worker, f){
    var b = encoder.encode(f.toString());
    worker.postMessage(b.buffer, [b.buffer]);
  }

  var worker = setupForPreExistentWorker();

  worker.onmessage = function(event){
    __finish();
  }

  function setupForNewWorkerEachTime(){
    var wCode = function(event){
      postMessage(null);
    };

    var blob = new Blob([
      "onmessage = " + wCode.toString()]);

    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  }
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Pre existent worker
postCode(worker, function() { postMessage(null); });

// Test case 2 - New workers
var newWorker = setupForNewWorkerEachTime();

newWorker.onmessage = function(event){
  newWorker.terminate();
  deferred.resolve();
}
newWorker.postMessage(null);
```

Listing 8: Benchmark at http://jsperf.com/worker-cold-start/6

The results for it are displayed in Figure 2.

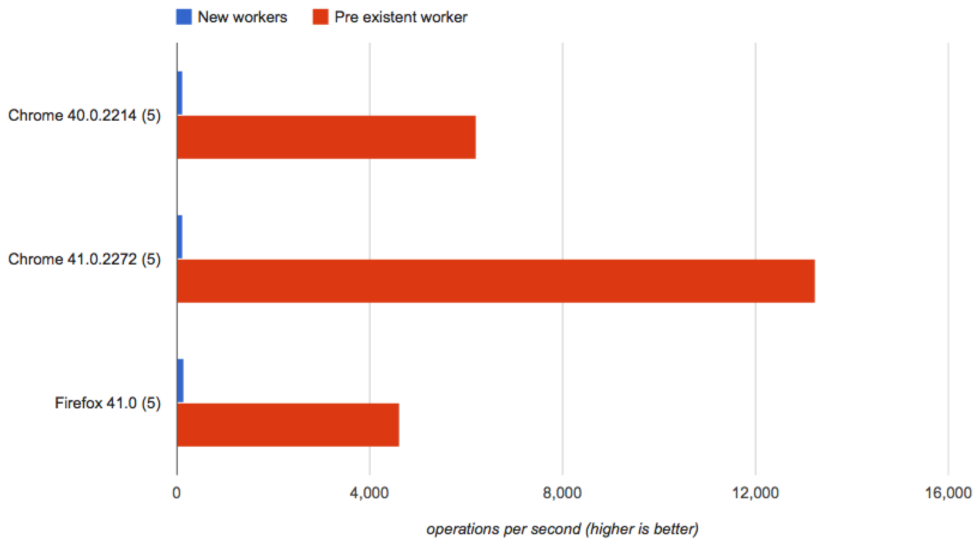


Figure 2: Evaluating the inefficiency of cold starting a Web Worker

5.1.2 Conclusion

The previous benchmark shows a huge difference between the time required to process a message on a pre-existent Web Worker compared to the one necessary to start a Web Worker and process a message. For that reason, we decided to create a pool of Web Workers when the library is initialized and dispatch work to them when it arrives. Since operations are asynchronous, if an operation is executing when another one is added for processing, the latter will be added to a queue and processed when the former completes.

5.2 High per-instance memory cost

The second factor makes it important for users to be able to control/limit the amount of web workers that the library instantiates. For that reason the library allows users to specify a maximum amount of workers to be instantiated, and also provide through its API a way to terminate all workers in existence.

5.3 Maximum amount of workers

Since the work being done is CPU intensive (i.e.: the workers' goal is not to perform I/O operations), the maximum amount of workers to be created should not be greater than the amount of cores in the machine W_{cores}, otherwise interleaving would start to affect performance. Additionally, considering the necessity of users to control how many workers are created, an upper bound can be specified $W_{upperbound}$[4]. If no upper bound is specified, the user upper bound for workers is assumed to be infinite (because the user is not providing a restriction). To summarize, the number of workers to be instantiated is:

$$W = min(W_{cores}, W_{upperbound})$$

[4]If the executing browser does not have an API to detect the amount of cores in a device $W_{cores} = W_{upperbound}$. If no $W_{upperbound}$ is provided, $W_{cores} = 1$.

6 Serialization and Transference

Let's assume that we have an array of `N` elements on which we want to perform a particular transformation through the `map` function. If code is executed in a single thread (no parallelism) and the average time to process an element is t then we could approximate the total time T_{ser} for the operation as:

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

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

$$T_{par} = \frac{T_{ser}}{K}$$

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 memory is not shared by default. These are:

- Partitioning or joining the elements to transfer.
- Serializing/deserializing the elements to transfer.
- Transferring the elements back and forth between the main thread and the workers.

All of the above happens twice, once when starting the computation and once when it ends, since the result from each worker needs to be consolidated before returning the new array with the transformed elements.

If we consider T_t as the transfer time, T_s as the serialization/deserialization time and T_p as the partitioning/joining time, it is clear that:

$$T_{sync} = T_t + T_s + T_p$$

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

Based on the previous equations we deduce that: the more we can reduce T_{sync}, the closer to the best possible scenario we 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 one [9]:

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

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

Listing 9: The Worker interface

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 [21]: *"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*[5] *or expensive resources across worker boundaries."*

Considering the definition of a `Transferable` the latter seems like a good alternative to minimize both T_s and T_t. Even more so when one considers that otherwise objects are copied using structured cloning (explained in section **2.7.6** of that same standard).

In the case of `SharedArrayBuffer` instances, T_p is also minimized. Instead of actually making copies of the buffer's data, all that is required a simple calculation to determine the boundaries on which each Web Worker should operate. Additionally, T_t should also decrease as there is no need to transfer the `SharedArrayBuffer` back to the UI thread.

6.1 Benchmarks

To verify the above we put together two benchmarks that aim to verify the difference between invoking `postMessage` with and without structured cloning for a `TypedArray`. Since `SharedArrayBuffers` cannot be cloned, we created a separate benchmark to compare the performance of transferring `SharedArrayBuffers` and `ArrayBuffers`.

The benchmarks transfer the arrays back and forth between the UI thread and a worker; the only difference between the two is that one has 100,000

[5]Note that the definition does not consider `SharedArrayBuffer` instances as those are sharable and transferable.

(a hundred thousand) elements in the array and the other one 1,000,000 (a million).

6.1.1 TypedArray With 100,000 elements

The following is the code for the benchmark:

```
// http://jsperf.com/transferrable-vs-cloning/3

// HTML setup
<script>
  var __finish;

  function createElements() {
    var total = 100000;
    var elements = new Uint32Array(total);

    for (var i = total; i > 0; i--){
      elements[i - 1] = Math.floor(Math.random() * 10000);
    }
    return elements;
  };

  function setupTransferrableWorker(){
    var wCode = function(event){
      var elements = event.data;
      postMessage(elements, [elements.buffer]);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  function setupCloningWorker(){
    var wCode = function(event){
      var elements = event.data;
      postMessage(elements);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var elementsA = createElements();
  var elementsB = createElements();
  var a = true;
  var elements = elementsA;

  var cloningWW = setupCloningWorker();
  cloningWW.onmessage = function(event) {
    __finish();
  }

  var transferrableWW = setupTransferrableWorker();
  transferrableWW.onmessage = function(event) {
    if (a) { // https://esdiscuss.org/topic/arraybuffer-
        neutering
      elementsA = event.data;
      elements = elementsB;
    } else {
      elementsB = event.data;
      elements = elementsA;
    }
    a = !a;
    __finish();
  }
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
}

// Test case 1 - Cloning
cloningWW.postMessage(elements);

// Test case 2 - Transferrable
transferrableWW.postMessage(elements, [elements.buffer]);
```

Listing 10: Benchmark at http://jsperf.com/transferrable-vs-cloning/3

The results for it are displayed in Figure 3.

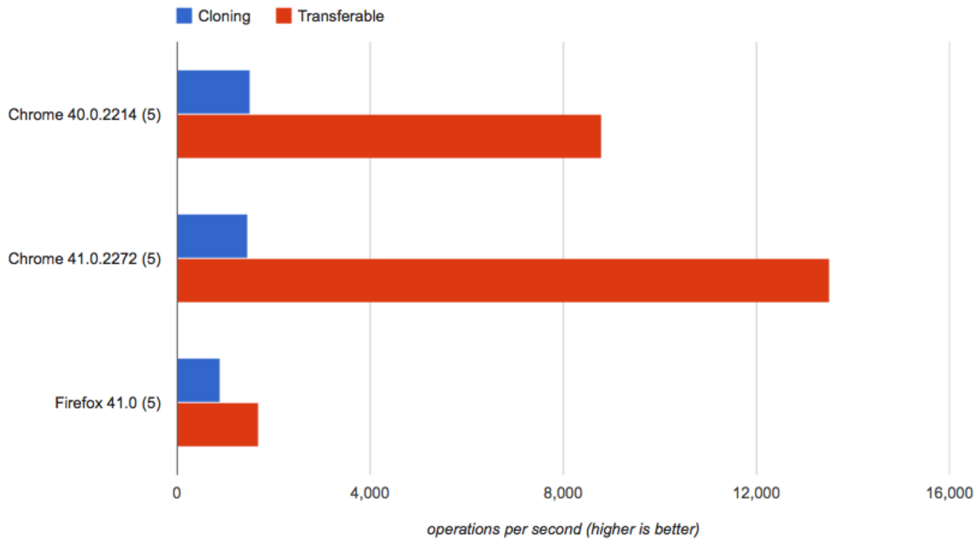


Figure 3: Results with 100,000 elements

6.1.2 TypedArray With 1,000,000 elements

The following is the code for the benchmark:

```
// http://jsperf.com/longer-transferrable-vs-cloning/3

// HTML setup
<script>
  var __finish;

  function createElements() {
    var total = 1000000;
    var elements = new Uint32Array(total);

    for (var i = total; i > 0; i--){
      elements[i - 1] = Math.floor(Math.random() * 10000);
    }
    return elements;
  };

  function setupTransferrableWorker(){
    var wCode = function(event){
      var elements = event.data;
      postMessage(elements, [elements.buffer]);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  function setupCloningWorker(){
    var wCode = function(event){
      var elements = event.data;
      postMessage(elements);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var elementsA = createElements();
  var elementsB = createElements();
  var a = true;
  var elements = elementsA;

  var cloningWW = setupCloningWorker();
  cloningWW.onmessage = function(event) {
    __finish();
  }

  var transferrableWW = setupTransferrableWorker();
  transferrableWW.onmessage = function(event) {
    if (a) {
      elementsA = event.data;
      elements = elementsB;
    } else {
      elementsB = event.data;
      elements = elementsA;
    }
    a = !a;
    __finish();
  }
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
}

// Test case 1 - Long cloning
cloningWW.postMessage(elements);

// Test case 2 - Long trasnferrable
transferrableWW.postMessage(elements, [elements.buffer]);
```

Listing 11: Benchmark at http://jsperf.com/longer-transferrable-vs-cloning/3

The results for it are displayed in Figure 4.

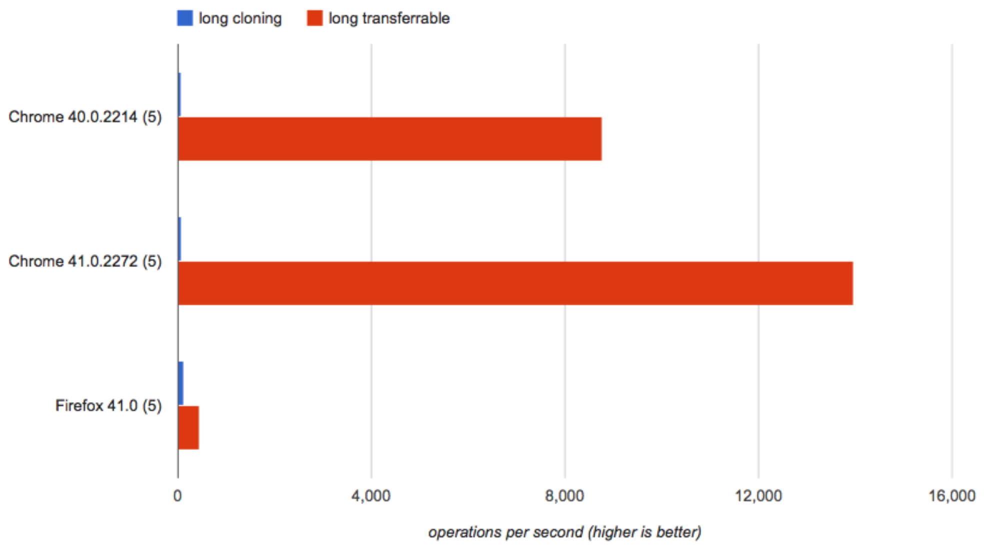


Figure 4: Results with 100,000 elements

6.1.3 Transferring SharedArrayBuffers and ArrayBuffers

The following is the code for the benchmark:

```
// http://jsperf.com/transfer-shared-vs-not-shared

// HTML setup
<script>
  var __finish;

  function createNotSharedElements() {
    return createElements(Uint32Array)
  };

  function createSharedElements() {
    return createElements(SharedUint32Array)
  };

  function createElements(type) {
    var total = 1000000;
    var elements = new type(total);
    for (var i = total; i > 0; i--){
      elements[i - 1] = Math.floor(Math.random() * 10000);
    }
    return elements;
  };

  function setupTransferableWorker(){
    var wCode = function(event){
      var elements = event.data;
      postMessage(elements, [elements.buffer]);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var elementsNotSharedA = createNotSharedElements();
  var elementsNotSaherdB = createNotSharedElements();

  var elementsSharedA = createSharedElements();
  var elementsSaherdB = createSharedElements();

  var a = true;
  var b = true;
  var elementsNotShared = elementsNotSharedA;
  var elementsShared = elementsSharedA;

  var transferableNotSharedWW = setupTransferableWorker();
  transferableNotSharedWW.onmessage = function(event) {
    if (a) {
      elementsNotSharedA = event.data;
      elementsNotShared = elementsNotSaherdB;
    } else {
      elementsNotSaherdB = event.data;
      elementsNotShared = elementsNotSharedA;
    }
    a = !a;
    __finish();
  }

  var transferableSharedWW = setupTransferableWorker();
  transferableSharedWW.onmessage = function(event) {
    if (b) {
      elementsSharedA = event.data;
      elementsShared = elementsSaherdB;
    } else {
      elementsSaherdB = event.data;
      elementsShared = elementsSharedA;
    }
    b = !b;
    __finish();
  }
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
}

// Test case 1 - not shared transferrable
transferableNotSharedWW.postMessage(elementsNotShared, [
    elementsNotShared.buffer]);

// Test case 2 - shared transferable
transferableSharedWW.postMessage(elementsShared, [
    elementsShared.buffer]);
```

Listing 12: Benchmark at http://jsperf.com/transfer-shared-vs-not-shared

The results for it are displayed in Figure 5.

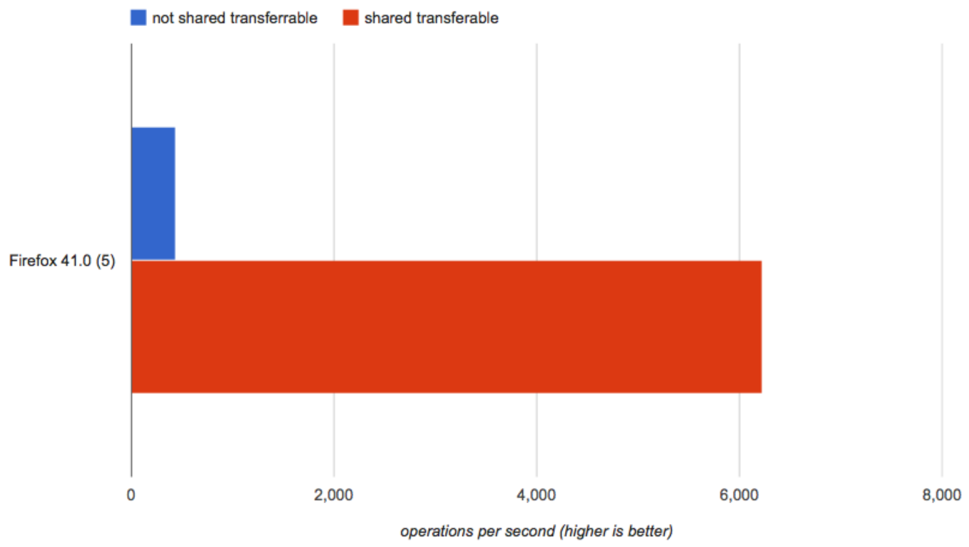


Figure 5: Transferring SharedArrayBuffers and ArrayBuffers

6.1.4 Conclusion

As we can see from the previous results, 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 we will only work with `TypedArray`s and `SharedTypedArray`s and all of them will be passed to/from Web Workers by transferring their ownership.

Although it does not affect our decision, it is also interesting to note the big difference between transferring `SharedArrayBuffer`s and `ArrayBuffer` in Firefox Nightly. This difference is probably going to be smaller over the course of time.

7 Code sharing

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 [22] 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 [23] 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.

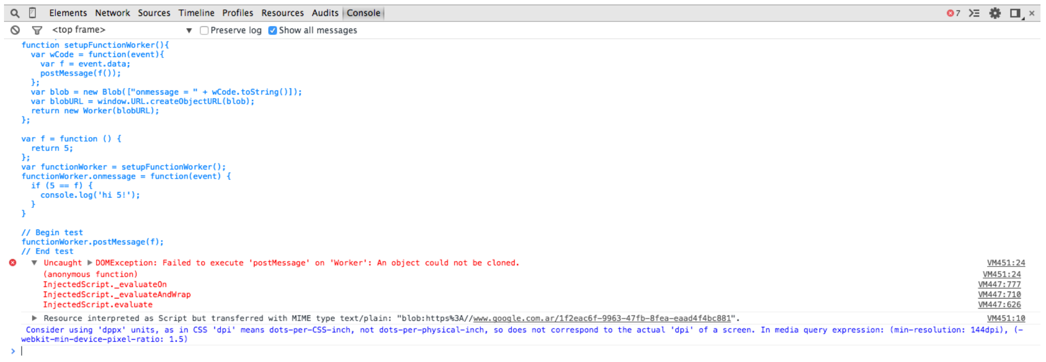
A naive approach would be trying to send the function to be executed directly to the worker. The following code fragment attempts to do so:

```javascript
// Test ran using Chrome javascript console

// JavaScript setup
function setupFunctionWorker(){
  var wCode = function(event){
    var f = event.data;
    postMessage(f());
  };
  var blob = new Blob(["onmessage = " + wCode.toString()]);
  var blobURL = window.URL.createObjectURL(blob);
  return new Worker(blobURL);
};

var f = function () {
  return 5;
};
var functionWorker = setupFunctionWorker();
functionWorker.onmessage = function(event) {
  if (5 == f) {
    console.log('hi 5!');
  }
}

// Test case - Begin test
```

```
functionWorker.postMessage(f);
```

Listing 13: Trying to send a function to a worker

When executed in the Google Chrome console the error shown in Figure 6 is thrown.



```
function setupFunctionWorker(){
  var wCode = function(event){
    var f = event.data;
    postMessage(f());
  };
  var blob = new Blob(["onmessage = " + wCode.toString()]);
  var blobURL = window.URL.createObjectURL(blob);
  return new Worker(blobURL);
};

var f = function () {
  return 5;
};
var functionWorker = setupFunctionWorker();
functionWorker.onmessage = function(event) {
  if (5 == f) {
    console.log('hi 5!');
  }
}

// Begin test
functionWorker.postMessage(f);
// End test
Uncaught ▶ DOMException: Failed to execute 'postMessage' on 'Worker': An object could not be cloned.            VM451:24
    (anonymous function)                                                                                    VM451:24
    InjectedScript._evaluateOn                                                                              VM447:777
    InjectedScript._evaluateAndWrap                                                                         VM447:710
    InjectedScript.evaluate                                                                                 VM447:626
▶ Resource interpreted as Script but transferred with MIME type text/plain: "blob:https%3A//www.google.com.ar/1f2eac6f-9963-47fb-8fea-eaad4f4bc881".   VM451:10
 Consider using 'dppx' units, as in CSS 'dpi' means dots-per-CSS-inch, not dots-per-physical-inch, so does not correspond to the actual 'dpi' of a screen. In media query expression: (min-resolution: 144dpi), (-webkit-min-device-pixel-ratio: 1.5)
> |
```

Figure 6: Error when trying to send a Function object to a Web Worker

That error is expected. As explained in section 9.5.3 of the HTML standard [21], the value being sent must support the structured clone algorithm and a Function does not meet that requirement as explained in section 2.7.5 of that same standard. It could be assumed that this is because of the challenges of serializing functions.

7.1 Serializing functions

Considering the fact that Function objects cannot be directly transferred, a different serialization approach is to be considered. Since function closures will not be transferrable 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.

Note: An approach to provide contextual data to worker functions is explored in Section 8.

7.1.1 `String` encoding benchmarks

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

- Use the Encoding API [24]
- Implement non native encoding/decoding functions

We created a benchmark to understand the difference between each approach:

```
// http://jsperf.com/pjs-serialization-comparison/4

// HTML setup
<script src="http://mrale.ph/irhydra/jsperf-renamer.js"></
    script>
<script>
  var fStringified = (function () {
    var n = 1000;
    var sum = 0;
    for (var i = 0; i < n; i++) {
      sum = i + (n * 2 - (i * 3));
      sum--;
    }
    var counter = ((function () {
      var safeCounter = 0;
      return function () { safeCounter++; return safeCounter
        ;};
    })());
    while (800 > counter()) {
      sum--;
    }
    return sum;
  }).toString();

  var nonNativeEncode = function (str) {
    var strLength = str.length;
    var buf = new ArrayBuffer(strLength * 2);
    var bufView = new Uint16Array(buf);
    for (; strLength--; ) {
      bufView[strLength] = str.charCodeAt(strLength);
    }
    return bufView;
  };
  var nonNativeDecode = function (ab) {
    return String.fromCharCode.apply(null , ab);
  };

  var encoder = new TextEncoder();
  var decoder = new TextDecoder();

  //
  var withNonNativeFunction = function () {
    var encodedF = nonNativeEncode(fStringified);
    var decodedF = nonNativeDecode(encodedF);
    if (decodedF === fStringified) {
      console.log(decodedF);
    } else {
      console.log('error');
    }
  }

  var withEncodingAPI = function () {
    var encodedF = encoder.encode(fStringified);
    var decodedF = decoder.decode(encodedF);
    if (decodedF === fStringified) {
      console.log(decodedF);
    } else {
      console.log('error');
    }
  };
</script>

// Test case 1 - Non-native function
withNonNativeFunction();

// Test case 2 - Native encoding API
withEncodingAPI();
```

Listing 14: Benchmark at http://jsperf.com/pjs-serialization-comparison/4

The results for it are displayed in Figure 7.

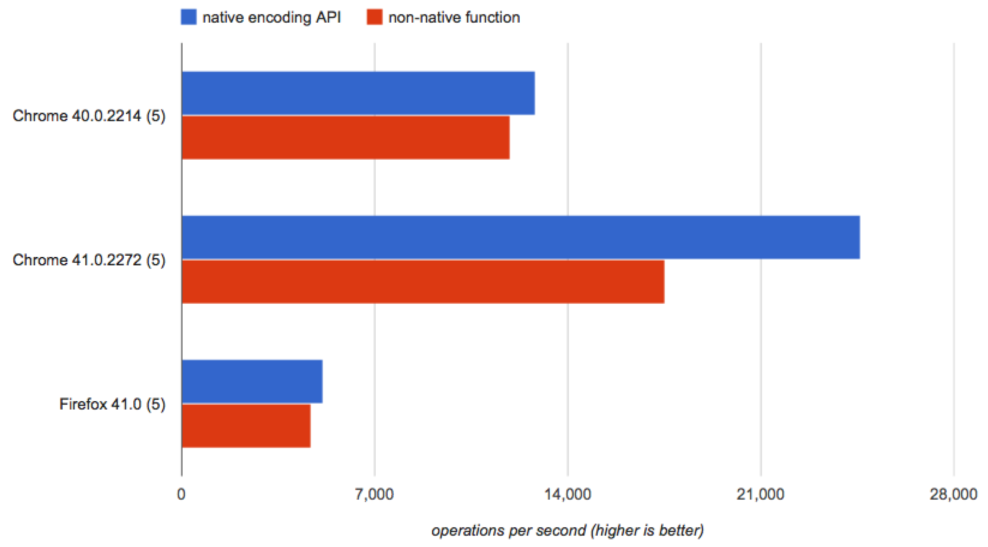


Figure 7: Comparing string encoding/decoding approaches

7.1.2 Conclusion

The native implementation was faster than the non-native one and the difference increased between Chrome versions, which seems to indicate that some extra optimizations could be expected in the future.

7.2 Transferring strings

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 transfer 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.

7.2.1 `String` encoding and transfer benchmarks

We created a benchmark using two functions whose string representation has different lengths to see if the function's length affected the serialization and transfer time. We measured three approaches to transfer the string:

1. Encoding the `String` into an `ArrayBuffer`, sending it as a `Transferable` and decoding it on the Web Worker.

2. Sending the string directly

3. Using a `Blob` to store the stream and retrieving it in the Worker.

The benchmark with the short function was the following one:

```
// http://jsperf.com/pjs-serialization/5

// HTML setup
<script>
  var __finish;

  function setupCopyWorker(){
    var wCode = function(event){
      var code = event.data;
      eval('var __f = ' + code);
      __f();
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var copyWorker = setupCopyWorker();
  copyWorker.onmessage = function(event) {
    __finish();
  }

  // ---

  function setupTransferrableWorker(){
    var wCode = function(event){
      var codeBuffer = event.data;
      var codeArray = new Uint8Array(codeBuffer);
      var code = decoder.decode(codeArray);
      eval('var __f = ' + code);
      __f();
    };

    var blob = new Blob(["var decoder = new TextDecoder();
       onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var transferrableWorker = setupTransferrableWorker();
  transferrableWorker.onmessage = function(event) {
    __finish();
  }

  var encoder = new TextEncoder();

  // ------

  function setupForBlobSerializationWorker(){
    var wCode = function(event){
      var blobURL = event.data;
      importScripts(blobURL);
      __f();
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  }

  var blobWorker = setupForBlobSerializationWorker();
  blobWorker.onmessage = function(event) {
    __finish();
  }

  var fStringified = (function () {
    postMessage(null);
  }).toString();
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Sending code by blob
var blob = new Blob(['__f = ' + fStringified], { type: '
    application/javascript' });
```

```
var blobURL = window.URL.createObjectURL(blob);

blobWorker.postMessage(blobURL);

// Test case 2 - Sending code by copy
copyWorker.postMessage(fStringified);

// Test case 3 - Sending code by transferrable objects
var b = encoder.encode(fStringified);
transferrableWorker.postMessage(b, [b.buffer]);
```

Listing 15: Benchmark at http://jsperf.com/pjs-serialization/5

The results for it are displayed in Figure 8.

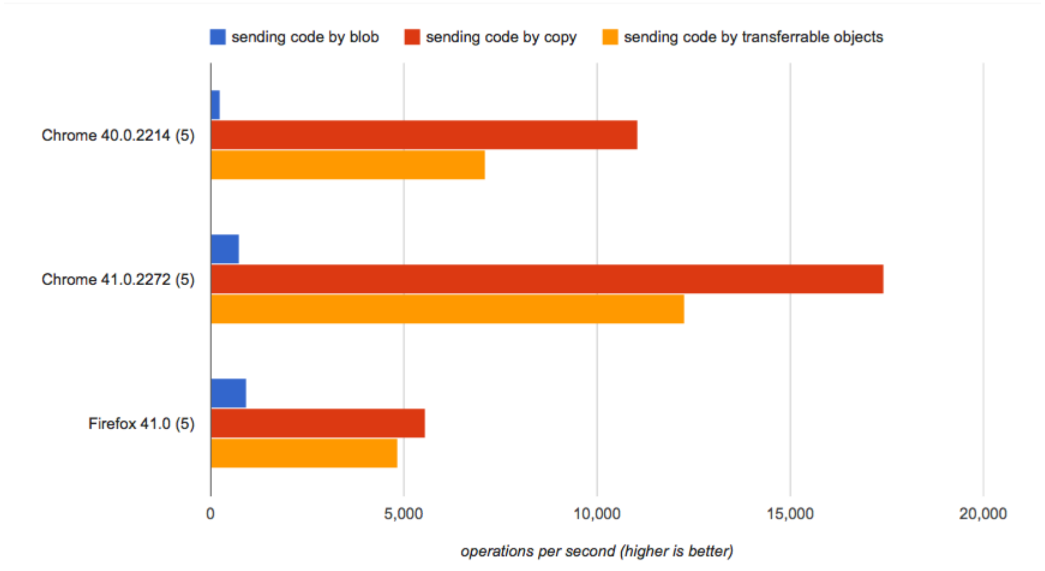


Figure 8: Comparing string encoding/decoding approaches

The benchmark with the long function was the following one:

```
// http://jsperf.com/pjs-serialization-long/3

// HTML setup
<script>
  var __finish;

  function setupCopyWorker(){
    var wCode = function(event){
      var code = event.data;
      eval('var __f = ' + code);
      postMessage(__f.length);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var copyWorker = setupCopyWorker();
  copyWorker.onmessage = function(event) {
    __finish();
  }

  // ---

  function setupTransferrableWorker(){
    var wCode = function(event){
      var codeBuffer = event.data;
      var codeArray = new Uint8Array(codeBuffer);
      var code = decoder.decode(codeArray);
      eval('var __f = ' + code);
      postMessage(__f.length);
    };

    var blob = new Blob(["var decoder = new TextDecoder();
       onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var transferrableWorker = setupTransferrableWorker();
  transferrableWorker.onmessage = function(event) {
    __finish();
  }

  var encoder = new TextEncoder();

  // ------

  function setupForBlobSerializationWorker(){
    var wCode = function(event){
      var blobURL = event.data;
```

```
      importScripts(blobURL);
      postMessage(__f.length);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  }

  var blobWorker = setupForBlobSerializationWorker();
  blobWorker.onmessage = function(event) {
    __finish();
  }

  var fStringified = (function () {
    var n = 1000;
    var sum = 0;
    for (var i = 0; i < n; i++) {
      sum = i + (n * 2 - (i * 3));
      sum--;
    }
    var counter = ((function () {
      var safeCounter = 0;
      return function () { safeCounter++; return safeCounter
          ;};
    })());
    while (800 > counter()) {
      sum--;
    }
    return sum;
  }).toString();
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Sending code by blob
var blob = new Blob(['__f = ' + fStringified], { type: '
    application/javascript' });
var blobURL = window.URL.createObjectURL(blob);

blobWorker.postMessage(blobURL);
```

```

// Test case 2 - Sending code by copy
copyWorker.postMessage(fStringified);

// Test case 3 - Sending code by transferrable objects
var b = encoder.encode(fStringified);
transferrableWorker.postMessage(b, [b.buffer]);
```

Listing 16: Benchmark at http://jsperf.com/pjs-serialization-long/3

The results for it are displayed in Figure 9.

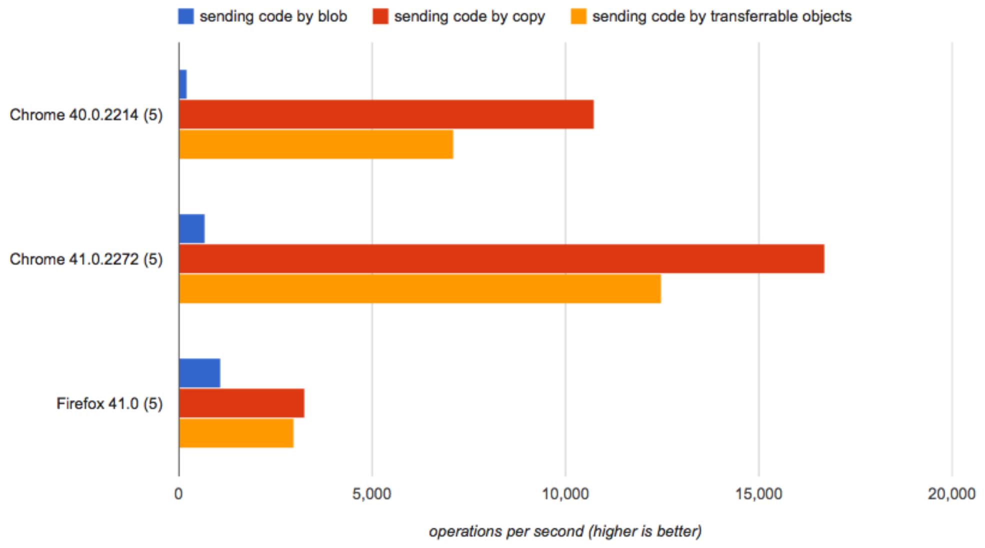


Figure 9: Comparing string encoding/decoding approaches

7.2.2 `String` serialization and shared transfer benchmark

We created a benchmark that compares these two scenarios:

1. Sending the `String` representation of a function to four Web Workers.
2. Encoding a function into a `SharedArrayBuffer` and sending it to four Web Workers.

The code for the benchmark is presented in the following code listing:

```
// http://jsperf.com/shared-function-transfer

// HTML setup
<script>
  var workersCount = 4;
  var finishedCount;
  var __finish;

  function setupCopyWorker(){
    var wCode = function(event){
      var code = event.data;
      eval('var __f = ' + code);
      postMessage(__f.length);
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  // ---

  function setupTransferrableWorker(){
    var wCode = function(event){
      var codeArray = event.data;
      var aux = new Uint8Array(codeArray.length);
      aux.set(codeArray);
      var code = decoder.decode(aux);
      eval('var __f = ' + code);
      postMessage(__f.length);
    };

    var blob = new Blob(["var decoder = new TextDecoder();
       onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  // -------

  var transferrableWorkers = [];
  var copyWorkers = [];

```

```
for (var k = 0; k < workersCount; k += 1) {
  var transferrableWorker = setupTransferrableWorker();
  transferrableWorker.onmessage = function(event) {
    finishWork();
  }
  transferrableWorkers.push(transferrableWorker);

  var copyWorker = setupCopyWorker();
  copyWorker.onmessage = function(event) {
    finishWork();
  }
  copyWorkers.push(copyWorker);
}

var encoder = new TextEncoder();

// ------

var fStringified = (function () {
  var n = 1000;
  var sum = 0;
  for (var i = 0; i < n; i++) {
    sum = i + (n * 2 - (i * 3));
    sum--;
  }
  var counter = ((function () {
    var safeCounter = 0;
    return function () { safeCounter++; return safeCounter
        ;};
  })());
  while (800 > counter()) {
    sum--;
  }
  return sum;
}).toString();

//shared array should be inside on trasnfer function's body
    but we had a problem with that: benchmark can not
   create inifinite shared buffer.
var shared = new SharedUint8Array(encoder.encode(
   fStringified).length);

var transfer = function () {
  finishedCount = workersCount;
  var b = encoder.encode(fStringified);
```

```
    shared.set(b);
    for (var k = 0; k < workersCount; k += 1) {
      transferrableWorkers[k].postMessage(shared, [shared.
          buffer]);
    }
  }

  var copy = function () {
    finishedCount = workersCount;
    for (var k = 0; k < workersCount; k += 1) {
      copyWorkers[k].postMessage(fStringified);
    }
  }

  var finishWork = function () {
    finishedCount--;
    if (finishedCount === 0) {
      __finish();
    }
  }
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Copy
copy();

// Test case 2 - Transferrable
transfer();
```

Listing 17: Benchmark at http://jsperf.com/shared-function-transfer

The results for it are displayed in Figure 10.

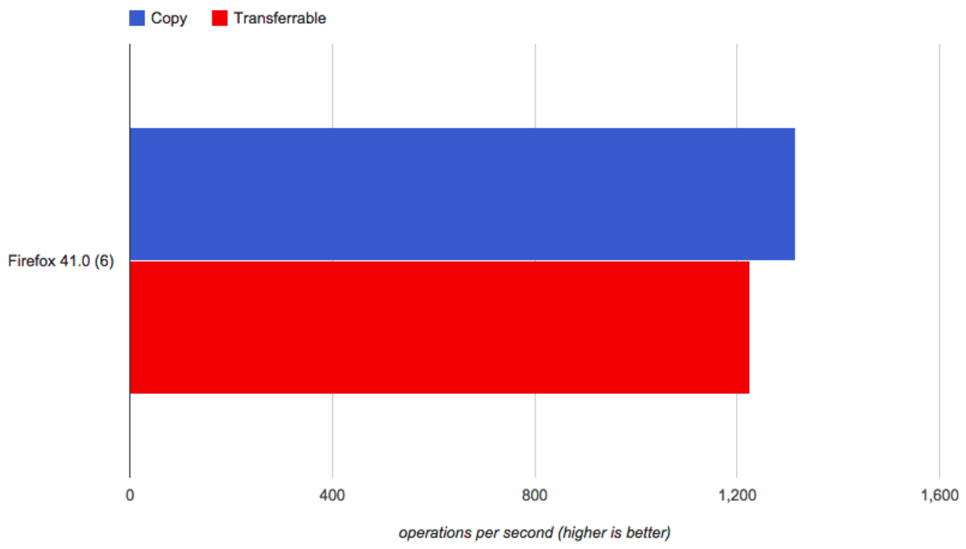


Figure 10: Comparing string serialization vs shared buffer transfer approaches

7.2.3 `String` and `TypedArray` serialization and transfer benchmarks

We created a benchmark that transfers a `TypedArray` and a `String` that are properties of the same object back and forth between the main thread an a worker to understand if transferring additional objects affected the transfer time when comparing it with just transferring a `String`.

The benchmark's code is the following one:

```
// http://jsperf.com/pjs-encoding/3

// HTML setup
<script>
  var __finish;

  function setupCopyWorker(){
    var wCode = function(event){
      var code = event.data.code;
      var elements = event.data.elements;
      if (elements.length === 10000) {
        postMessage(elements, [elements.buffer]);
      }
    };

    var blob = new Blob(["onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var copyWorker = setupCopyWorker();
  copyWorker.onmessage = function(event) {
    __finish();
  }

  // ---

  function setupTransferrableWorker(){
    var wCode = function(event){
      var elements = event.data.elements;
      var codeBuffer = event.data.code;
      var codeArray = new Uint8Array(codeBuffer);
      var code = decoder.decode(codeArray);
      if (elements.length === 10000) {
        postMessage(elements, [elements.buffer]);
      }
    };

    var blob = new Blob(["var decoder = new TextDecoder();
        onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var transferrableWorker = setupTransferrableWorker();
  transferrableWorker.onmessage = function(event) {
    __finish();
  }

  var encoder = new TextEncoder();

  // --

  var generateElements = function () {
    var total = 10000;
    var typed = new Uint8Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = i;
    }
    return typed;
  };
  var generateElements2 = function () {
    var total = 10000;
    var typed = new Uint8Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = i;
    }
    return typed;
  };

  var fStringified = (function () {
    postMessage(null);
  }).toString();
</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Sending code by copy
var data = {
  elements: generateElements(),
  code: fStringified
};
copyWorker.postMessage(data, [data.elements.buffer]);

// Test case 2 - Sending code by transferrable objects
var data = {
  elements: generateElements2(),
  code: encoder.encode(fStringified)
};
transferrableWorker.postMessage(data, [data.code.buffer, data
    .elements.buffer]);
```

Listing 18: Benchmark at http://jsperf.com/pjs-encoding/3

The results for it are displayed in Figure 11.

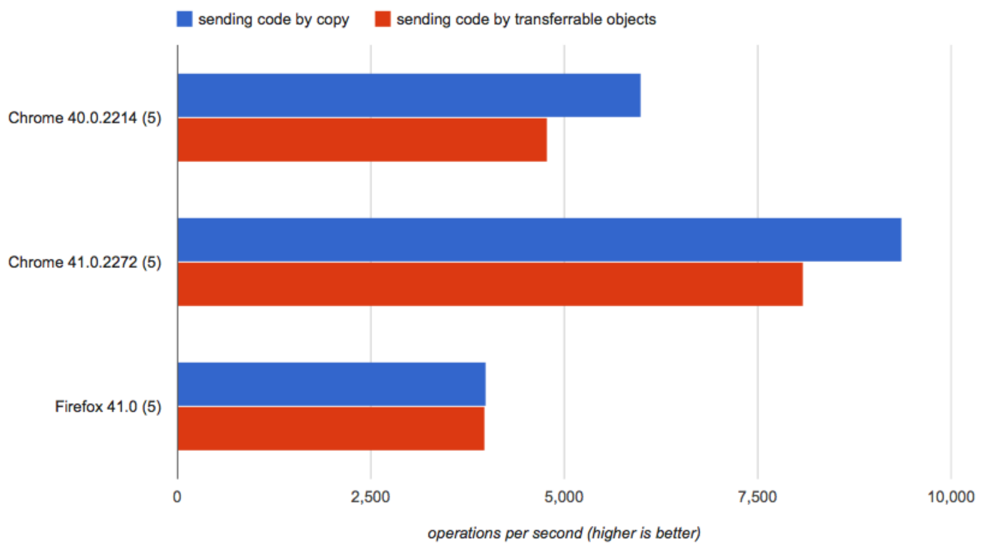


Figure 11: Comparing string encoding/decoding approaches

7.2.4 Conclusion

As the benchmarks show, the fastest way to transfer the `String` between the UI thread and workers is by sending it directly.

8 Context

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 Section 7, `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:

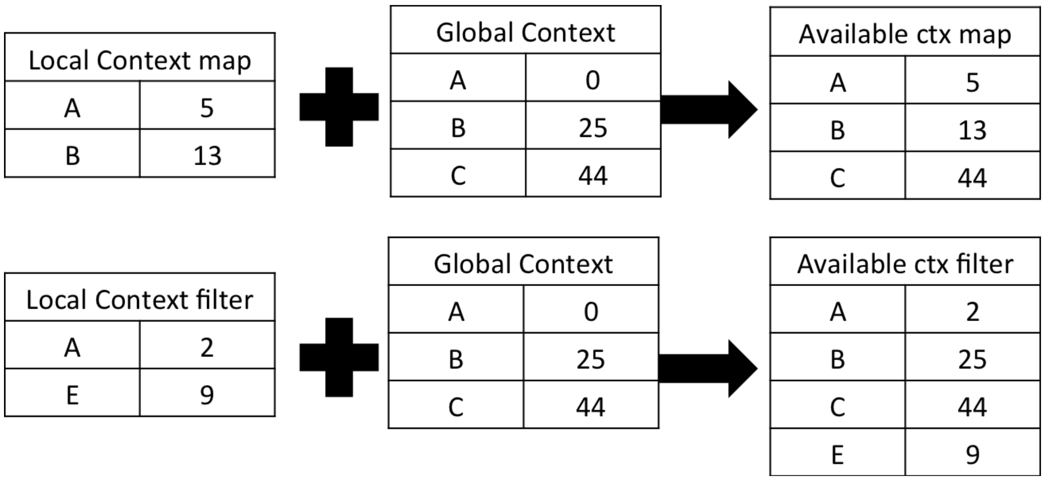
```
var xs = new SharedUint8Array(4);
xs.set([1,2,3,4]);

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]
  });
});
```

Listing 19: 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. 12.

```
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 })
});
```

Listing 20: Example for resulting ctx



Local Context map	
A	5
B	13

Global Context	
A	0
B	25
C	44

Available ctx map	
A	5
B	13
C	44

Local Context filter	
A	2
E	9

Global Context	
A	0
B	25
C	44

Available ctx filter	
A	2
B	25
C	44
E	9

Figure 12: Comparing `TypedArray` merge approaches

The reason why there are two kinds of context is because there will usually be data that does not need to be passed to the Web Workers with each chain, so having a global context allows this. Additionally, the local context provides a way for developers to avoid polluting the global one with keys that are not necessary for all operations.

We created a benchmark to explore the difference between sending the context and chains to execute as part of the same message, and using two different messages. The code for the benchmark is in the following listing:

```
// http://jsperf.com/additional-sendingdata-comparison/3

// HTML setup
<script>
  var __finish;

  function createTypedArray (length) {
    var result = new Uint32Array(length);
    for (var i = length; i > 0; i--){
      result[i - 1] = Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000);
    }
    return result;
  };

  function createPackage(typedArray) {
    return {
      index: 2,
      buffer: typedArray.buffer,
      operations:  [{
        name: 'map',
        args: ['e'],
        code: '{ return e * 4 + Math.rand() % 1000 + 1 / e; }
           '
      }],
      elementsType: 'Uint32Array'
    };
  };

  function setupForPreExistentWorker(){
    var wCode = function(event){
      var data = event.data;
      if (data.pack) {
        postMessage(data.pack.buffer, [data.pack.buffer]);
      }
    };

    var blob = new Blob([
        "onmessage = " + wCode.toString()]);

    var blobURL = window.URL.createObjectURL(blob);

    return new Worker(blobURL);
  };

  var worker = setupForPreExistentWorker();

  worker.onmessage = function(event){
    if (a) {
      elementsA = new Uint32Array(event.data);
      packA = createPackage(elementsA);
      elements = elementsB;
      pack = packB;
    } else {
      elementsB = new Uint32Array(event.data);
      packB = createPackage(elementsB);
      elements = elementsA;
      pack = packA;
    }
    a = !a;
    __finish();
  };

  var strFunc = (function (e) { return (Math.rand() % 1000) +
      e * 3; }).toString();
  var ctx = {
    func: {
      __isFunction: true,
      code: strFunc
    },
    data: 'daaaaaata'
  };

  var a = true;
  var elementsA = createTypedArray(1000000);
  var elementsB = createTypedArray(1000000);
  var packA = createPackage(elementsA);
  var packB = createPackage(elementsB);
  var elements = elementsA;
  var pack = packA;

</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};

```

```

// Test case 1 - sending together
worker.postMessage({ctx: ctx, pack: pack}, [pack.buffer]);

// Test case 2 - sending separated
worker.postMessage({ctx: ctx});
worker.postMessage({pack: pack}, [pack.buffer]);
```

Listing 21: Benchmark at http://jsperf.com/additional-sendingdata-comparison/3

The results for the benchmark are displayed in Figure 13.

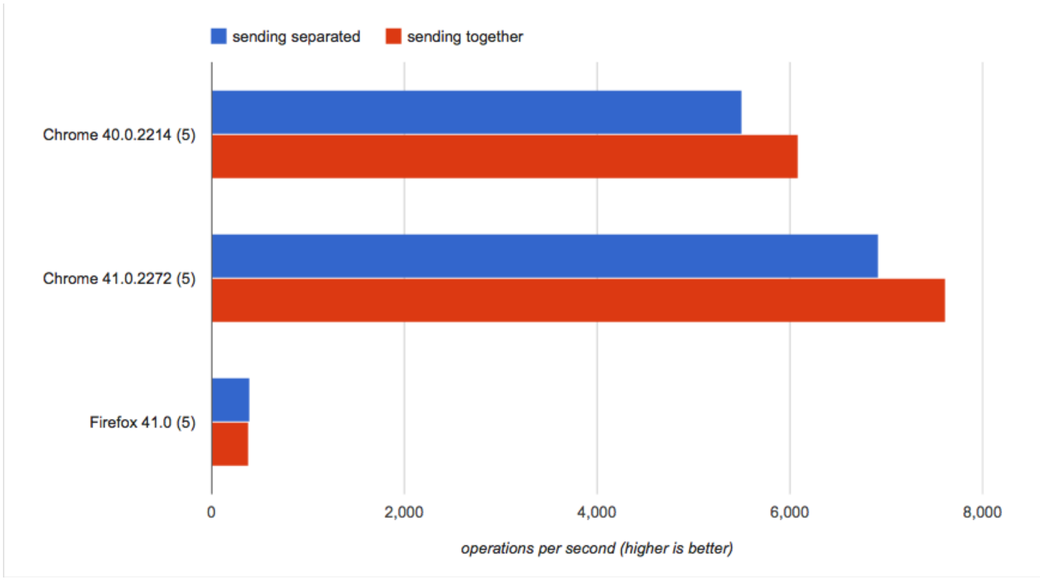


Figure 13: Comparing small context message passing

8.1 Implementation

Internally the context is sent as a message to all Web Workers to make its properties available. Two possible implementatios were possible:

- Sending the object directly and rely on structured cloning
- Serializing the object using `JSON.stringify` and deserialize it in the workers

Note: None of the aforementioned approaches support sending functions as context keys. This is covered in Subsection 8.2.

We created two separate benchmarks, one that sends small objects to a Web Worker and another one that sends large objects to a Web Worker. Both benchmarks compare the approaches against each other.

The code for the benchmark with the small object is in the following listing:

```
// http://jsperf.com/additional-clonningdata-comparison/3

// HTML setup
<script>
  var __finish;

  function createTypedArray (length) {
    var result = new Uint32Array(length);
    for (var i = length; i > 0; i--){
      result[i - 1] = Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000);
    }
    return result;
  };

  function createPackage(typedArray) {
    return {
      index: 2,
      buffer: typedArray.buffer,
      operations:  [{
        name: 'map',
        args: ['e'],
        code: '{ return e * 4 + Math.rand() % 1000 + 1 / e; }
          '
      }],
      elementsType: 'Uint32Array'
    };
  };

  function setupForPreExistentWorker(){
    var wCode = function(event){
      var data = event.data;
      if (data.pack) {
        postMessage(data.pack.buffer, [data.pack.buffer]);
      }
    };
    var blob = new Blob([
        "onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);
    return new Worker(blobURL);
  };
  var worker = setupForPreExistentWorker();

  function setupForPreExistentStrfyWorker(){
    var wCode = function(event){
      var data = event.data;
      var strfyCtx = data.ctx;
      var ctx = JSON.parse(strfyCtx);
      if (data.pack && ctx) {
        postMessage(data.pack.buffer, [data.pack.buffer]);
      }
    };
    var blob = new Blob([
        "onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);
    return new Worker(blobURL);
  };
  var strfyWorker = setupForPreExistentStrfyWorker();

  worker.onmessage = strfyWorker.onmessage = function(event){
    if (a) {
      elementsA = new Uint32Array(event.data);
      packA = createPackage(elementsA);
      elements = elementsB;
      pack = packB;
    } else {
      elementsB = new Uint32Array(event.data);
      packB = createPackage(elementsB);
      elements = elementsA;
      pack = packA;
    }
    a = !a;
    __finish();
  };

  var strFunc = (function (e) { return (Math.rand() % 1000) +
      e * 3; }).toString();
  var ctx = {
    func: {
      __isFunction: true,
      code: strFunc
    },
    data: 'daaaaaata'
  };
  var strfyCtx = JSON.stringify(ctx);

  var a = true;
  var elementsA = createTypedArray(1000000);
  var elementsB = createTypedArray(1000000);
  var packA = createPackage(elementsA);
  var packB = createPackage(elementsB);
  var elements = elementsA;
  var pack = packA;

</script>

// JavaScript setup
__finish = function () {
  deferred.resolve();
};


// Test case 1 - sending clonned context
worker.postMessage({ctx: ctx, pack: pack}, [pack.buffer]);

// Test case 2 - sending stringified context
strfyWorker.postMessage({ctx: strfyCtx, pack: pack}, [pack.
   buffer]);
```

Listing 22: Benchmark at http://jsperf.com/additional-clonningdata-comparison/3

The results for the benchmark are displayed in Figure 14.

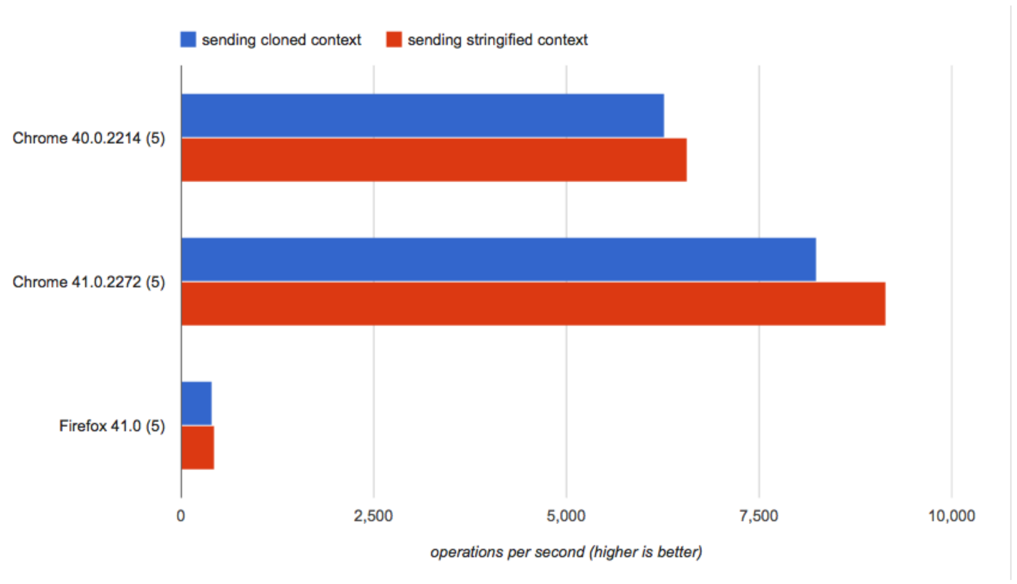


Figure 14: Comparing small context message passing

The code for the benchmark with the large object is in the following listing:

```
// http://jsperf.com/additional-clonningdata-comparison-long
   /3

// HTML setup
<script>
  var __finish;

  function createTypedArray (length) {
    var result = new Uint32Array(length);
    for (var i = length; i > 0; i--){
      result[i - 1] = Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000)
      + Math.floor(Math.random() * 10000);
    }
    return result;
  };

  function createPackage(typedArray) {
    return {
      index: 2,
      buffer: typedArray.buffer,
      operations:  [{
        name: 'map',
        args: ['e'],
        code: '{ return e * 4 + Math.rand() % 1000 + 1 / e; }
            '
      }],
      elementsType: 'Uint32Array'
    };
  };

  function setupForPreExistentWorker(){
    var wCode = function(event){
      var data = event.data;
      if (data.pack) {
        postMessage(data.pack.buffer, [data.pack.buffer]);
      }
    };
    var blob = new Blob([
        "onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);
    return new Worker(blobURL);
  };
  var worker = setupForPreExistentWorker();

  function setupForPreExistentStrfyWorker(){
    var wCode = function(event){
      var data = event.data;
      var strfyCtx = data.ctx;
      var ctx = JSON.parse(strfyCtx);
      if (data.pack && ctx) {
        postMessage(data.pack.buffer, [data.pack.buffer]);
      }
    };
    var blob = new Blob([
        "onmessage = " + wCode.toString()]);
    var blobURL = window.URL.createObjectURL(blob);
    return new Worker(blobURL);
  };
  var strfyWorker = setupForPreExistentStrfyWorker();

  worker.onmessage = strfyWorker.onmessage = function(event){
    if (a) {
      elementsA = new Uint32Array(event.data);
      packA = createPackage(elementsA);
      elements = elementsB;
      pack = packB;
    } else {
      elementsB = new Uint32Array(event.data);
      packB = createPackage(elementsB);
      elements = elementsA;
      pack = packA;
    }
    a = !a;
    __finish();
  };

  var strFunc1 = (function (e) { return (Math.rand() % 1000)
     + e * 3; }).toString();
  var strFunc2 = (function (e) { return (e && 0xff00ff00) +
     Math.rand() % 5000 + 54 > 0xff0a0000; }).toString();
  var strFunc3 = (function (e) { return (Math.rand() % 7364)
     + Math.pow(e, 2) * 3; }).toString();
  var ctx = {
    func1: {
      __isFunction: true,
      code: strFunc1
    },
    func2: {
      __isFunction: true,
      code: strFunc2
    },
    func3: {
      __isFunction: true,
      code: strFunc3
    },
    data: 'daaaaaata',
    data2: 32424,
    data3: 'looooongdataaaastr'
  };
  var strfyCtx = JSON.stringify(ctx);

  var a = true;
  var elementsA = createTypedArray(1000000);
  var elementsB = createTypedArray(1000000);
  var packA = createPackage(elementsA);
  var packB = createPackage(elementsB);
  var elements = elementsA;
  var pack = packA;

</script>
```

```

// JavaScript setup
__finish = function () {
  deferred.resolve();
};


// Test case 1 - sending clonned context
worker.postMessage({ctx: ctx, pack: pack}, [pack.buffer]);

// Test case 2 - sending stringified context
strfyWorker.postMessage({ctx: strfyCtx, pack: pack}, [pack.
    buffer]);
```

Listing 23: Benchmark at http://jsperf.com/additional-clonningdata-comparison-long/3

The results for the benchmark are displayed in Figure 15.

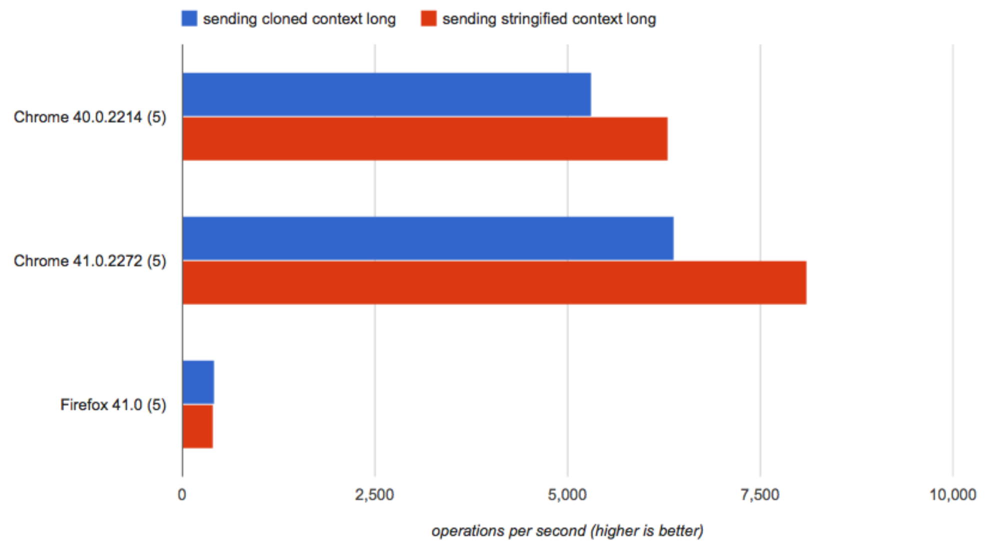


Figure 15: Comparing large context message passing

8.2 Contexts with functions

Properties whose values are `Function` instances are serialized into an object. For example if the following object were passed as a context:

```
{
  a: 2,
  b: function(c) {
    return c + 1;
  }
}
```

Listing 24: Example context with a function property

It would result in the following object before being serialized:

```
{
  a: 2,
  b: {
    args: [ 'c' ],
    code: '\n\treturn c + 1;\n',
    __isFunction: true
  }
}
```

Listing 25: Example processed context

8.3 Conclusion

Based on the benchmark results, we decided to use `JSON.stringify` to serialize `context` objects. This involves a tradeoff as, unlike structured cloning, it does not support cyclic references. For this scenario, we believe that the performance benefit is more valuable than supporting cyclic references.

9 Array partitioning

As explained in Section 6 of this document, transferring `TypedArray` objects to and from Web Workers requires the use of `Transferables` to achieve acceptable performance. Since our library will receive `TypedArray` objects to be partitioned and handed over to multiple workers, it is important that it spends as little time as possible partitioning the arrays.[6]

If the library is configured to use N workers then the original data in the `TypedArray` will copied into N smaller `TypedArrays`. For ideal performance it would be nice to be able to pass a *pointer* into the array and an amount of elements to process, but because `TypedArrays` do not support shared memory, and element ownership is tranferred to the Web Workers, the elements must be copied to the sub-arrays.

9.1 Copying `TypedArrays`

Sections 22.2 and 24.1 of the ECMAScript 6 draft [16] specify the API for `TypedArray` and `ArrayBuffer` objects respectively. Some of the exposed operations provide a way to copy the contents of an `ArrayBuffer` into another `ArrayBuffer`, so they are to be considered as viable alternatives for the copy operation. Other alternatives to consider are:

- Implementing a non-native function that copies the array elements one by one.
- Using a `Blob` to store the contents of a `TypedArray` and reading the `Blob`'s contents into a new `TypedArray`.

9.1.1 `TypedArray` copy benchmark

A benchmark was created based on the aforementioned alternatives:

```
// http://jsperf.com/arraybuffer-copy/4

// HTML setup
<script src="http://mrale.ph/irhydra/jsperf-renamer.js"></
    script>
<script>
```

[6]This does not apply to `SharedTypedArrays` as those can be transferred without being partitioned.

```
function createElements() {
  var total = 1000000;
  var elements = new Uint8Array(total);

  for (var i = total; i > 0; i--){
    elements[i - 1] = Math.floor(Math.random() * 10000);
  }
  return elements;
};

var elements = createElements();
var elementsCount = elements.length;

var fileReader = new FileReader();

function manualCopy (xs) {
  var ys = new Uint8Array(xs.length);
  for (var i = 0; i < xs.length; i++) {
    ys[i] = xs[i];
  }
  return ys;
};

function bufferSliceCopy (xs) {
  var ysBuffer = xs.buffer.slice(0);
  return new Uint8Array(ysBuffer);
};

function constructorFromBufferCopy(xs) {
  return new Uint8Array(xs.buffer, 0, xs.length);
};

function constructorFromArrayLikeCopy(xs) {
  return new Uint8Array(xs);
};

function typedSetCopy (xs) {
  var ys = new Uint8Array(xs.length);
  ys.set(xs);
  return ys;
};

function bufferSubarrayCopy (xs) {
  var ys = xs.subarray();
  return new Uint8Array(ys);
  };

  function blobCopy (xs, callback) {
    var b = new Blob([xs.buffer]);
    fileReader.onload = function() {
        callback(new Uint8Array(this.result));
    };
    fileReader.readAsArrayBuffer(b);
  };

  function dataViewCopy(xs) {
    var buffer = new ArrayBuffer(xs.length);
    var dv = new DataView(buffer);
    for (var i = 0; i < xs.length; i++) {
      dv.setInt8(i, xs[i]);
    }
    return new Uint8Array(buffer);
  };
</script>

// Test case 1 - Manual
var r = manualCopy(elements);
if (elementsCount !== r.length || r.buffer === elements.
    buffer) {
  console.log('error manual');
}

// Test case 2 - Buffer slice
var r = bufferSliceCopy(elements);
if (elementsCount !== r.length || r.buffer === elements.
    buffer) {
  console.log('error slice');
}

// Test case 3 - Constructor
var r = constructorFromArrayLikeCopy(elements);
if (elementsCount !== r.length || r.buffer === elements.
    buffer) {
  console.log('error constructor');
}

// Test case 4 - TypedArray set
var r = typedSetCopy(elements);
if (elementsCount !== r.length || r.buffer === elements.
    buffer) {
```

```
  console.log('error set');
}

// Test case 5 - Buffer subarray
var r = bufferSubarrayCopy(elements);
if (elementsCount !== r.length || r.buffer === elements.
    buffer) {
  console.log('error subarray');
}

// Test case 6 - Blob
blobCopy(elements, function (result) {
  if (elementsCount === result.length) {
    deferred.resolve();
  } else {
    console.log('error blob');
  }
});
```

Listing 26: Benchmark at http://jsperf.com/arraybuffer-copy/4

The results for it are displayed in Figure 16.

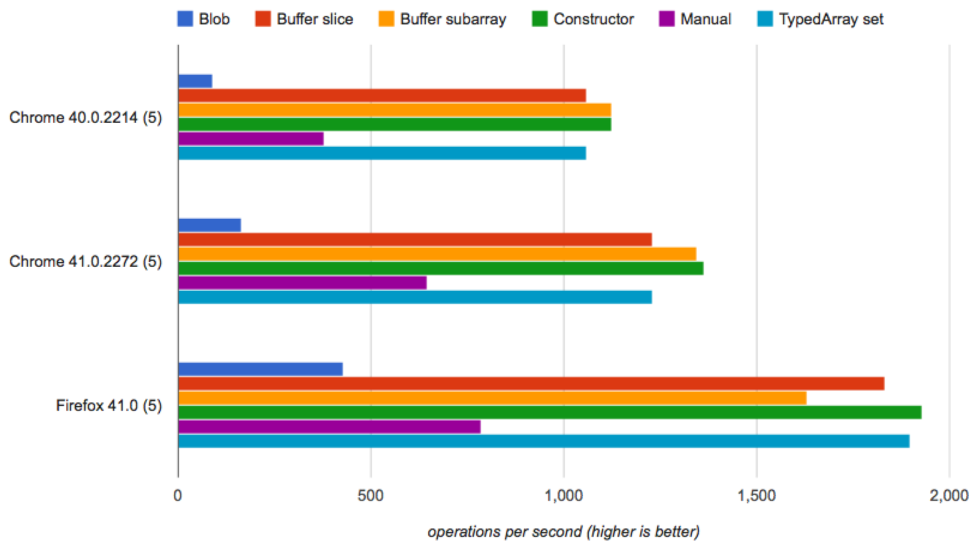


Figure 16: Comparing `TypedArray` copy approaches

Based on those results we can discard the `Blob` approach for partitioning `TypedArrays`.

Note: Alternatives *Buffer subarray* and *Constructor* do the same thing in the case of copying an entire `TypedArray`. However, *Constructor* is not a viable alternative for copying part of an array

9.1.2 `TypedArray` partition benchmark

Based on the result from the previous benchmark we created a new benchmark to understand which alternative is better to partition an array:

```
// http://jsperf.com/arraybuffer-split/3

// HTML setup
<script src="http://mrale.ph/irhydra/jsperf-renamer.js"></
   script>
<script>
  function createElements() {
    var total = 1000000;
    var elements = new Uint8Array(total);

    for (var i = total; i > 0; i--){
      elements[i - 1] = Math.floor(Math.random() * 10000);
    }
    return elements;
  };

  var parts = 4;
  var elements = createElements();
  var elementsCount = elements.length;
  var subElementsCount = elementsCount / parts;

  function manualSplit (xs) {
    var ys = [];
    for (var i = 0; i < parts; i++) {
      ys.push(new Uint8Array(subElementsCount));
    }

    for (var i = 0; i < subElementsCount; i++) {
      for (var p = 0; p < parts; p++) {
        ys[p][i] = xs[i + p * subElementsCount];
      }
    }
    return ys;
  };

  function bufferSliceSplit (xs) {
    var ys = [];
    for (var i = 0; i < parts; i++) {
      var b = xs.buffer.slice(i * subElementsCount * xs.
          BYTES_PER_ELEMENT, (i * subElementsCount +
          subElementsCount) * xs.BYTES_PER_ELEMENT);
      ys.push(new Uint8Array(b));
    }
    return ys;
  };

  function bufferSubarraySplit (xs) {
    var ys = [];
    for (var i = 0; i < parts; i++) {
      var subXs = xs.subarray(i * subElementsCount, i *
          subElementsCount + subElementsCount);
      ys.push(new Uint8Array(subXs));
    }
    return ys;
  };

  function typedSetSplit (xs) {
    var ys = [];
    for (var i = 0; i < parts; i++) {
      var subXs = xs.subarray(i * subElementsCount, i *
          subElementsCount + subElementsCount);
      var subYs = new Uint8Array(subElementsCount);
      subYs.set(subXs);
      ys.push(subYs);
    }
    return ys;
  };
</script>

// Test case 1 - Manual
var r = manualSplit(elements);
if (parts !== r.length) {
  console.log('error manual');
}

// Test case 2 - Buffer slice
var r = bufferSliceSplit(elements);
if (parts !== r.length) {
  console.log('error slice');
}

```

```
// Test case 3 - Buffer subarray
var r = bufferSubarraySplit(elements);
if (parts !== r.length) {
  console.log('error subarray');
}

// Test case 4 - TypedArray set
var r = typedSetSplit(elements);
if (parts !== r.length) {
  console.log('error set');
}
```

Listing 27: Benchmark at http://jsperf.com/arraybuffer-split/3

The results for it are displayed in Figure 17.

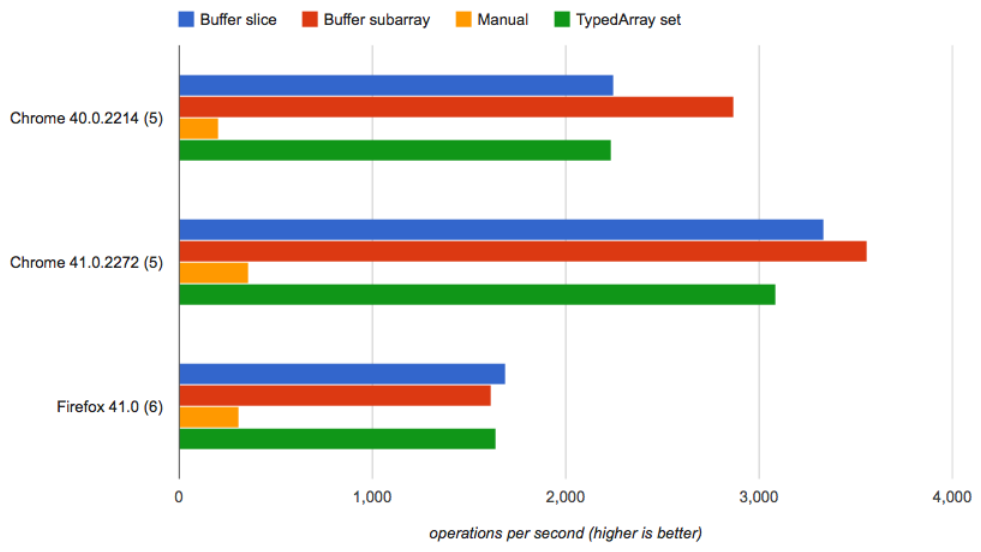


Figure 17: Comparing `TypedArray` partition approaches

9.1.3 Conclusion

The benchmark shows that there are three very similar alternatives to partition a `TypedArray`. We decided to use *Buffer subarray*. The implementation of the slice function could look something like this:

```
function typedArraySlice(array, from, to) {
  return new array.constructor(array.subarray(from, to));
}
```

Listing 28: Possible approach to copy a slice of a `TypedArray`

10 Array merging

Once all the workers have executed the `Function` provided to them on each element of their portion of the original array, they are to notify the UI thread and provide the resulting `ArrayBuffer` or `SharedArrayBuffer`.

At this point the UI thread must consolidate the results from the Web Workers into a new `TypedArray` or `SharedTypedArray` that represents the result for the entire operation.[7]

For the `ArrayBuffer` case, this means copying all the elements of each partial result into a larger `TypedArray` while maintaining the original order.

For the `SharedArrayBuffer` case there are two possible situations:

1. If all steps in the chain perform the `map` transformation, then no changes are required.

2. If there is at least one `filter` step, the elements need to be placed at the beginning of the resulting `SharedTypedArray` in order to avoid *holes*.

The latter is similar to the `ArrayBuffer` case, but instead of copying from one array to another, the source and target are the same.

10.1 `TypedArray` merge benchmark

Unlike the case for `TypedArray` splitting (Section 9.1) there aren't that many native alternatives to create a `TypedArray` from smaller ones.

The benchmark presented only has one alternative that uses a method with a native implementation:

```
// http://jsperf.com/typedarray-merge/2

// JavaScript setup
<script>
  var arraySize = 250000;
  var arraysCount = 4;

  function generateElements() {
    var arrays = [];

```

[7]The case for chains with `reduce` as the last step similar and simpler. Thus, it is not worth analyzing.

```
  for (var i = 0; i < arraysCount; i++) {
    arrays[i] = new Uint8Array(arraySize);
  }

  for (var j = 0; j < arraySize; j++) {
    for (var i = 0; i < arrays.length; i++) {
      arrays[i][j] = j + i * arraySize;
    }
  }
  return arrays;
};

var parts = generateElements();

function caseNonNativeFunction(parts) {
  var totalSize = 0;
  for (var j = 0; j < parts.length; j++) {
    totalSize += parts[j].length;
  }
  var result = new Uint8Array(totalSize);

  var from = 0;
  for (var j = 0; j < parts.length; j++) {
    var part = parts[j];
    var partLength = part.length;
    for (var i = 0; i < partLength; i++) {
      result[from + i] = part[i];
    }
    from += partLength;
  }
  return result;
};

function caseTypedArraySet(parts) {
  var totalSize = 0;
  for (var j = 0; j < parts.length; j++) {
    totalSize += parts[j].length;
  }
  var result = new Uint8Array(totalSize);

  var from = 0;
  for (var j = 0; j < parts.length; j++) {
    var array = parts[j];
    result.set(array, from);
    from += array.length;
  }
  return result;
};

function caseDataViewSet(parts) {
  var totalSize = 0;
  for (var j = 0; j < parts.length; j++) {
    totalSize += parts[j].length;
  }
  var buffer = new ArrayBuffer(totalSize);
  var dv = new DataView(buffer);

  var from = 0;
  for (var j = 0; j < parts.length; j++) {
    var part = parts[j];
    var partLength = part.length;
    for (var i = 0; i < partLength; i++) {
      dv.setUint8(from + i,  part[i]);
    }
    from += partLength;
  }
  return new Uint8Array(buffer);
};

function caseArrayLikeFunction(parts) {
  var totalSize = 0;
  for (var j = 0; j < parts.length; j++) {
    totalSize += parts[j].length;
  }
  var arrayLike = new Array(totalSize);
  var from = 0;
  for (var i = 0; i < parts.length; i++) {
    var part = parts[i];
    var partLength = part.length;
    for (var j = 0; j < partLength; j++) {
      arrayLike[from + j] = part[j];
    }
    from += partLength;
  }
  return new Uint8Array(arrayLike);
}

function withNonNativeFunction(parts) {
  var result = caseNonNativeFunction(parts);
  if (result.length !== (arraySize * arraysCount)) {
    console.log('error');
  }
}

function withTypedArraySet(parts) {
  var result = caseTypedArraySet(parts);
  if (result.length !== (arraySize * arraysCount)) {
    console.log('error');
  }
}

function withDataViewSet(parts) {
  var result = caseDataViewSet(parts);
  if (result.length !== (arraySize * arraysCount)) {
    console.log('error');
  }
}

function withArrayLikeFunction(parts) {
  var result = caseArrayLikeFunction(parts);
  if (result.length !== (arraySize * arraysCount)) {
    console.log('error');
  }
}
</script>

// Test case - Non-native function
withNonNativeFunction(parts);

// Test case - TypedArray set function
withTypedArraySet(parts);

// Test case - DataView set function
withDataViewSet(parts);

// Test case - Array like function
withArrayLikeFunction(parts);
```

Listing 29: Benchmark at http://jsperf.com/typedarray-merge/2

The results for the benchmark are displayed in Figure 18.

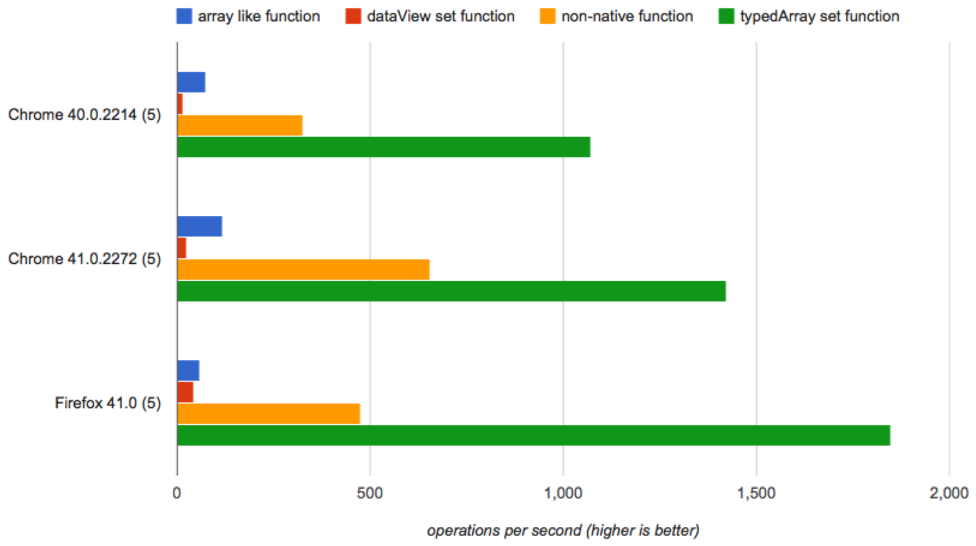


Figure 18: Comparing `TypedArray` merge approaches

10.2 `SharedTypedArray` merge benchmark

The benchmark modifies the `TypedArray` merge benchmark to support instances of the `SharedTypedArray` type. The implementation does not include the `DataView` alternative as it is not supported by `SharedTypedArrays`:

```
// http://jsperf.com/sharedtypedarray-merge

// JavaScript setup
<script>
  var arraySize = 250000;
  var arraysCount = 4;

  function generateElements() {
    var arrays = [];

    for (var i = 0; i < arraysCount; i++) {
      arrays[i] = new SharedUint8Array(arraySize);
    }

    for (var j = 0; j < arraySize; j++) {
      for (var i = 0; i < arrays.length; i++) {
        arrays[i][j] = j + i * arraySize;
      }
    }
    return arrays;
  };

  var parts = generateElements();

  function caseNonNativeFunction(parts) {
    var totalSize = 0;
    for (var j = 0; j < parts.length; j++) {
      totalSize += parts[j].length;
    }
    var result = new SharedUint8Array(totalSize);

    var from = 0;
    for (var j = 0; j < parts.length; j++) {
      var part = parts[j];
      var partLength = part.length;
      for (var i = 0; i < partLength; i++) {
        result[from + i] = part[i];
      }
      from += partLength;
    }
    return result;
  };

  function caseTypedArraySet(parts) {
    var totalSize = 0;
    for (var j = 0; j < parts.length; j++) {
      totalSize += parts[j].length;
    }
    var result = new SharedUint8Array(totalSize);

    var from = 0;
    for (var j = 0; j < parts.length; j++) {
      var array = parts[j];
      result.set(array, from);
      from += array.length;
    }
    return result;
  };

  function caseArrayLikeFunction(parts) {
    var totalSize = 0;
    for (var j = 0; j < parts.length; j++) {
      totalSize += parts[j].length;
    }
    var arrayLike = new Array(totalSize);
    var from = 0;
    for (var i = 0; i < parts.length; i++) {
      var part = parts[i];
      var partLength = part.length;
      for (var j = 0; j < partLength; j++) {
        arrayLike[from + j] = part[j];
      }
      from += partLength;
    }
    var result = new SharedUint8Array(arrayLike.length);
    result.set(arrayLike);
    return result;
  }

  function withNonNativeFunction(parts) {
    var result = caseNonNativeFunction(parts);
    if (result.length !== (arraySize * arraysCount)) {
      console.log('error');
    }
  }

  function withTypedArraySet(parts) {
    var result = caseTypedArraySet(parts);
    if (result.length !== (arraySize * arraysCount)) {
      console.log('error');
    }
  }

  function withArrayLikeFunction(parts) {
    var result = caseArrayLikeFunction(parts);
    if (result.length !== (arraySize * arraysCount)) {
      console.log('error');
    }
  }
</script>

// Test case - shared non-native function
withNonNativeFunction(parts);

// Test case - shared typedArray set function
withTypedArraySet(parts);

// Test case - shared array like function
```

```
withArrayLikeFunction(parts);
```

Listing 30: Benchmark at http://jsperf.com/sharedtypedarray-merge

The results for the benchmark are displayed in Figure 19.

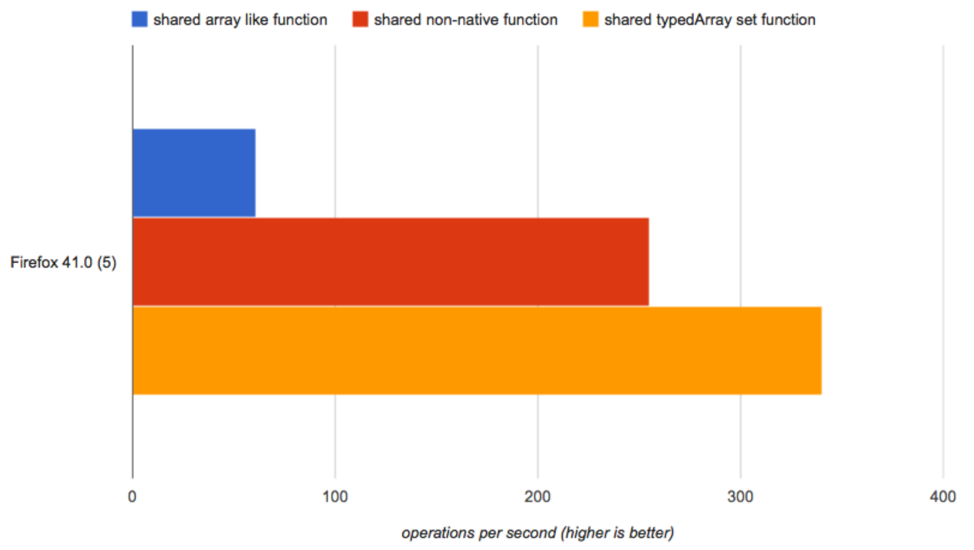


Figure 19: Comparing `SharedTypedArray` merge approaches

10.3 Conclusion

The benchmark shows that the best alternative to merge small arrays into a larger one is using `Set` method of either `TypedArray` or `SharedTypedArray`.

For illustration purposes, a naive implementation (without optimizations or error checking) would be similar to the following one:

```
function merge(arrays){
  var first = arrays[0];
  var total = arrays.reduce(function(c,a) { return c + a.
     length; }, 0);
  var result = new first.constructor(total);
  var start = 0;

  arrays.forEach(function(a){
    result.set(array, start);
    start += array.length;
  });

  return result;
}
```

Listing 31: Simple `TypedArray` merge function

11 Decreasing function transfer time

Each time `seq` is invoked on a chain, all callback functions related to `map`, `filter` and `reduce` steps are serialized and sent to the Web Workers.

Each Web Worker has a cache to avoid using different instances of the same function (as explained in Section 13). Nevertheless, if the same function is going to be used multiple times in a program as the callback for steps, it might make sense to send it to the Web Workers once and then simply make a reference to it.

This can be achieved by taking advantage of the global context object (introduced in Section 8).

The first thing that needs to be done is to call `pjs.updateContext` by providing a `Function` as a property of the context object. An example is shown in the following code snippet:

```
var promise = pjs.updateContext({
  add: function (x) { return x + 2; },
});
```

Listing 32: Sending function in global context

Once the global context has been updated, the `String` representation of the function's key (`'add'`) can be used as a callback parameter instead of using a `Function` instance. An example is shown in the following code snippet:

```
var promise = pjs.updateContext({
  add: function (x) { return x + 2; },
}).then(function(){
  pjs(new Uint8Array([1,2,3,4])).map('add').seq()
    .then(function(result){
      // result is [ 3, 4, 5, 6 ]
    });
});
```

Listing 33: Passing a function name from the global context instead of a `Function` as a callback

11.1 Benchmark

We created a benchmark to analyze the improvement obtained by just sending a `String` instead of a `Function` instance.

The following listing shows the code for the benchmark:

```
// http://jsperf.com/global-ctx-inlined-func/3

// HTML setup
<script src="http://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p
    -j-s.min.js"></script>
<script>
  var pjs = require('p-j-s');
  pjs.init();

  var generateElements = function (total) {
    var typed = new Uint32Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = i;
    }
    return typed;
  };

  var xs = generateElements(100000); // 100.000

  var wrappedXs = pjs(xs);

  function inlinedMapper(pixel) {
    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;
    var noise_r = Math.random() * 0.5 + 0.5;
    var noise_g = Math.random() * 0.5 + 0.5;
    var noise_b = Math.random() * 0.5 + 0.5;

    var new_r = Math.max(Math.min(255, noise_r * ((r * 0.393)
        + (g * 0.769) + (b * 0.189)) + (1 - noise_r) * r), 0)
       ;
    var new_g = Math.max(Math.min(255, noise_g * ((r * 0.349)
        + (g * 0.686) + (b * 0.168)) + (1 - noise_g) * g), 0)
       ;
    var new_b = Math.max(Math.min(255, noise_b * ((r * 0.272)
        + (g * 0.534) + (b * 0.131)) + (1 - noise_b) * b), 0)
       ;

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + new_r;
  };

  pjs.updateContext({
    x: inlinedMapper
  });
</script>

// JavaScript Setup
var __finish = function (result) {
  if (!result) {
    console.log('error');
  }
  deferred.resolve();
};

// Test case 1 - map with function instance
wrappedXs.map(inlinedMapper).seq(function (result) {
  __finish();
});

// Test case 2 - map with function key
wrappedXs.map('x').seq(function (result) {
  __finish();
});
```

Listing 34: Benchmark at http://jsperf.com/global-ctx-inlined-func/3

The results for the benchmark are displayed in Figure 20.

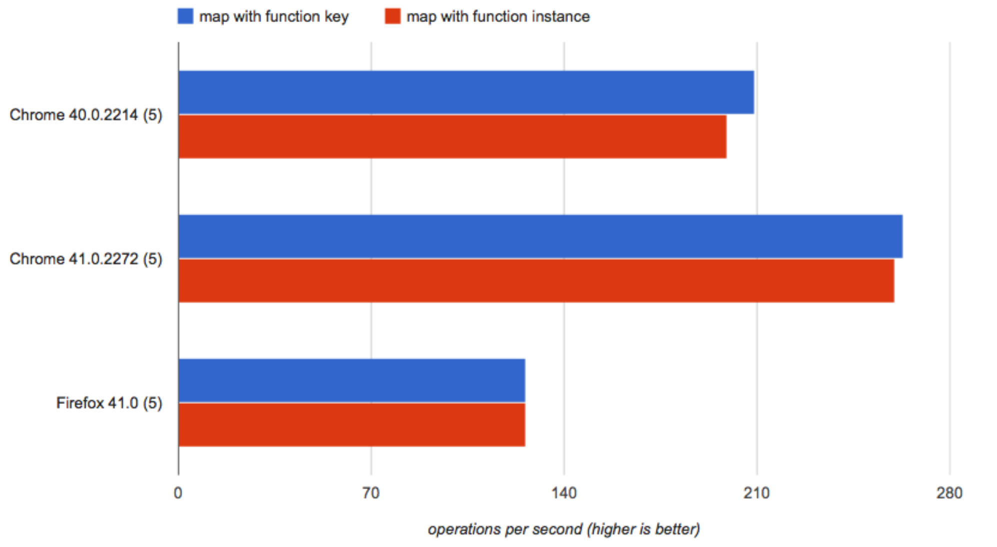


Figure 20: Comparing sending function key in global context against `Function` instance

11.2 Conclusion

While the performance improvement obtained is not remarkable, it is still noticeable and does not require a lot of effort from a developer's standpoint. It is worth pointing out that the improvement shown in Figure 20 will vary if the difference in character length between the key and the function's code does.

12 The importance of function inlining

One of the things found while comparing the performance of the parallel and non-parallel implementations of `map` is that our parallel implementation performed a lot worse than expected when considering the size of the `TypedArray` and the `mapper` function.

12.1 The scenario

An image tranformation to apply a sepia tone effect to pictures was used for the comparison. The core linear implementation was:

```
function noise() {
  return Math.random() * 0.5 + 0.5;
};

function clamp(component) {
  return Math.max(Math.min(255, component), 0);
}

function colorDistance(scale, dest, src) {
  return clamp(scale * dest + (1 - scale) * src);
};

function map(binaryData, len) {
  for (var i = 0; i < len; i += 4) {
      var r = binaryData[i];
      var g = binaryData[i + 1];
      var b = binaryData[i + 2];

      binaryData[i] = colorDistance(noise(), (r * 0.393) + (g
          * 0.769) + (b * 0.189), r);
      binaryData[i + 1] = colorDistance(noise(), (r * 0.349)
         + (g * 0.686) + (b * 0.168), g);
      binaryData[i + 2] = colorDistance(noise(), (r * 0.272)
         + (g * 0.534) + (b * 0.131), b);
  }
};
```

Listing 35: Sepia tone linear implementation

Note: The serial code works with a `Uint8ClampedArray` but accesses three elements at the same time. The parallel implementation uses a `Uint32Array` and works on one pixel (A,R,G,B) at a time.

It was implemented as shown in the following listing for the parallel case:

```
pjs(new Uint32Array(canvasData.data.buffer)).map(function(
    pixel){
  function noise() {
      return Math.random() * 0.5 + 0.5;
  };

  function clamp(component) {
      return Math.max(Math.min(255, component), 0);
  }

  function colorDistance(scale, dest, src) {
      return clamp(scale * dest + (1 - scale) * src);
  };

  var r = pixel & 0xFF;
  var g = (pixel & 0xFF00) >> 8;
  var b = (pixel & 0xFF0000) >> 16;

  var new_r = colorDistance(noise(), (r * 0.393) + (g *
     0.769) + (b * 0.189), r);
  var new_g = colorDistance(noise(), (r * 0.349) + (g *
     0.686) + (b * 0.168), g);
  var new_b = colorDistance(noise(), (r * 0.272) + (g *
     0.534) + (b * 0.131), b);

  return (pixel & 0xFF000000) + (new_b << 16) + (new_g << 8)
     + (new_r & 0xFF);
}).seq(function(result){
  // work on result
});
```

Listing 36: Sepia tone initial parallel implementation

12.2 The problem

Taking the experience from previous experiments into account we considered that the performance of the two implementations with a source `Uint32Array` of about 8,000,000 (eight million) elements should be similar, even favoring the parallel one. The initial measurements showed that the linear implementation fared approximately four times better than the parallel one.

12.3 The hypothesis

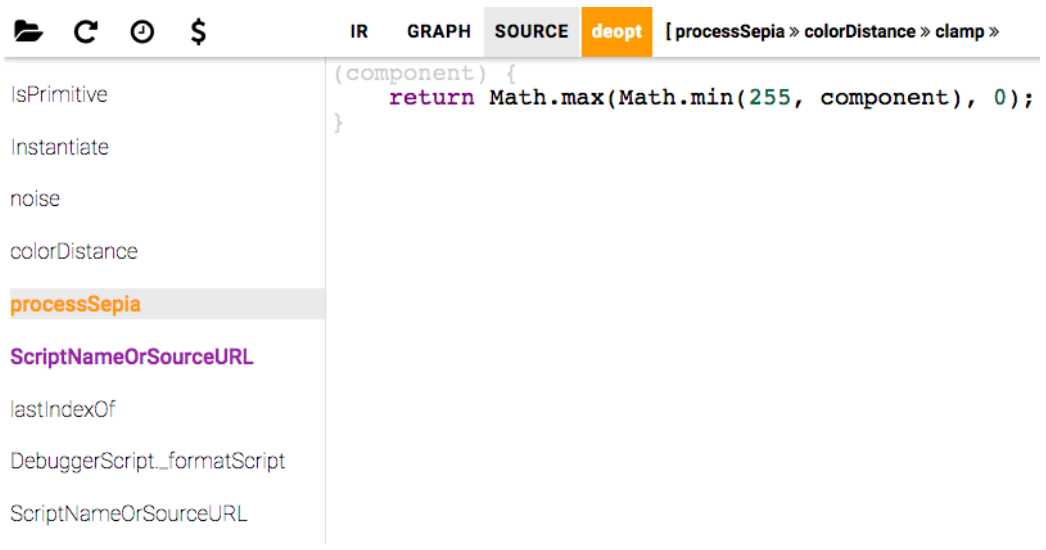
As the two code samples were similar our hypothesis was that for some reason some v8 optimizations were not taking place. It was a definite possibility considering that the parallel implementation depends on the `Function` constructor to create `Function` objects in the Web Workers. Specifically we believed that the internal functions `noise`, `clamp` and `colorDistance` were not being inlined. Inlining could be the cause of both benefits [25] and drawbacks [26] but the belief was that performance was being affected because of lack of it.

Using IR Hydra it was easy to verify the hypothesis, as Figures 21 through 28 show.

Note: The blue chevron indicates that a function has been inlined.

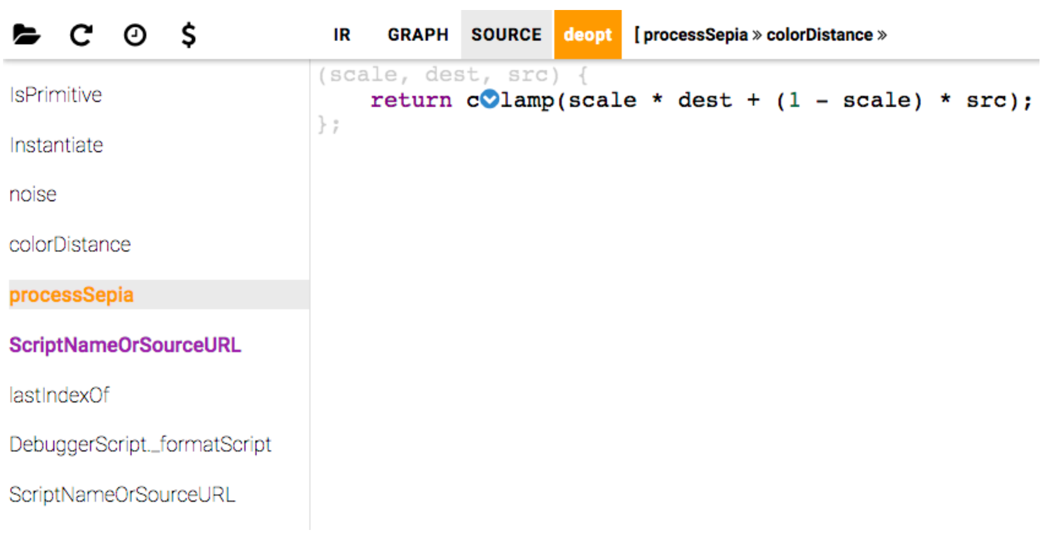
12.3.1 Serial implementation generated code

Figures 21 through 24 are related to the serial implementation. As it can be seen from them, all functions are being inlined.



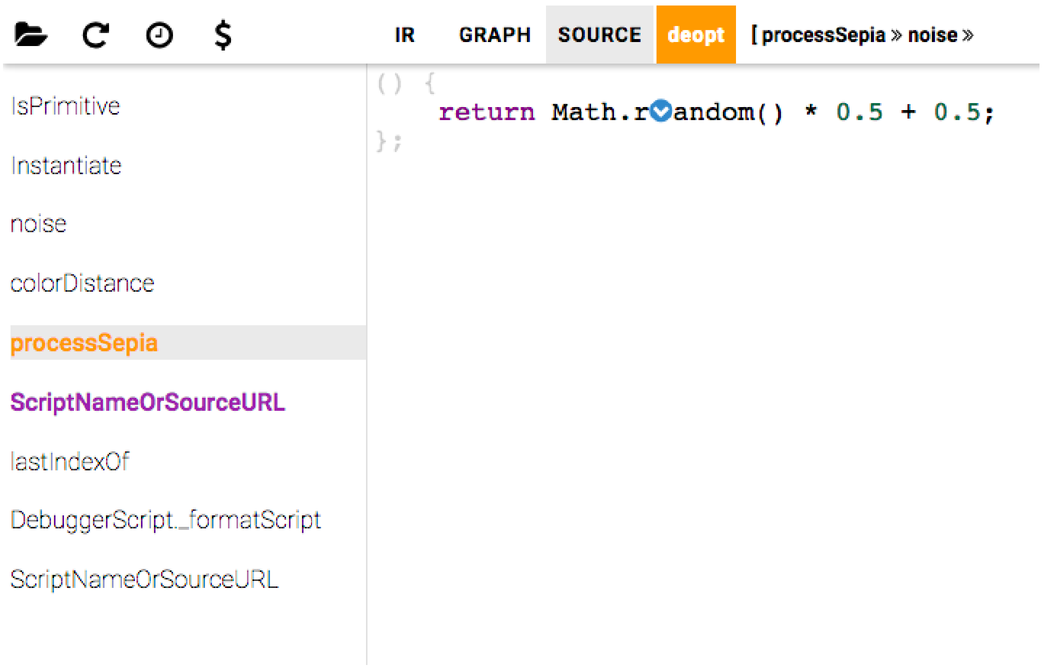
```
(component) {
    return Math.max(Math.min(255, component), 0);
}
```

Figure 21: `clamp` function - serial implementation



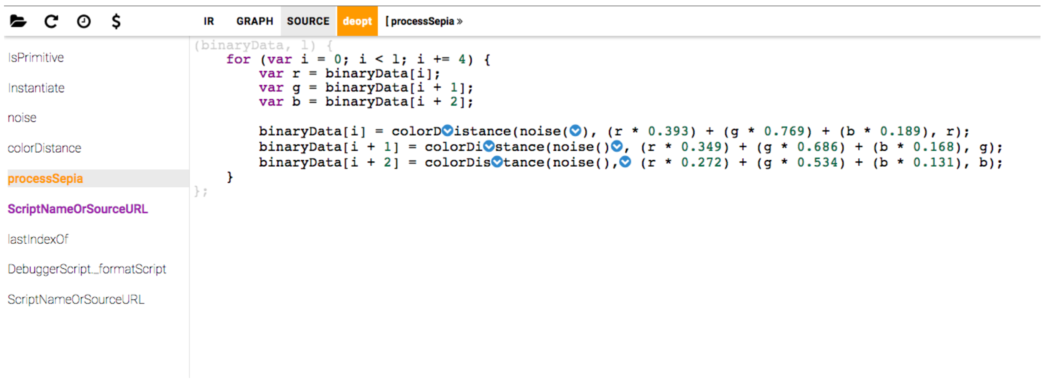
```
(scale, dest, src) {
    return clamp(scale * dest + (1 - scale) * src);
};
```

Figure 22: `colorDistance` function - serial implementation



```
() {
    return Math.random() * 0.5 + 0.5;
};
```

Figure 23: `noise` function - serial implementation

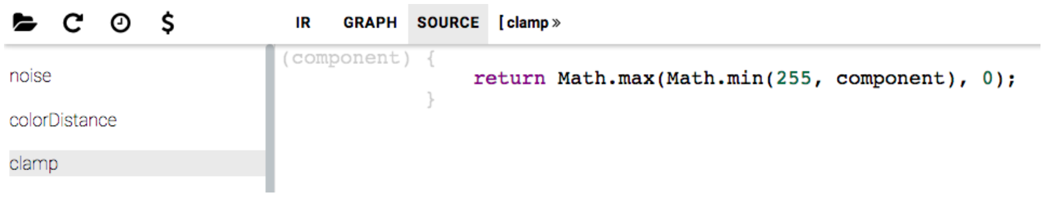


```
(binaryData, l) {
    for (var i = 0; i < l; i += 4) {
        var r = binaryData[i];
        var g = binaryData[i + 1];
        var b = binaryData[i + 2];

        binaryData[i] = colorDistance(noise(), (r * 0.393) + (g * 0.769) + (b * 0.189), r);
        binaryData[i + 1] = colorDistance(noise(), (r * 0.349) + (g * 0.686) + (b * 0.168), g);
        binaryData[i + 2] = colorDistance(noise(), (r * 0.272) + (g * 0.534) + (b * 0.131), b);
    }
};
```

Figure 24: `processSepia` function - serial implementation

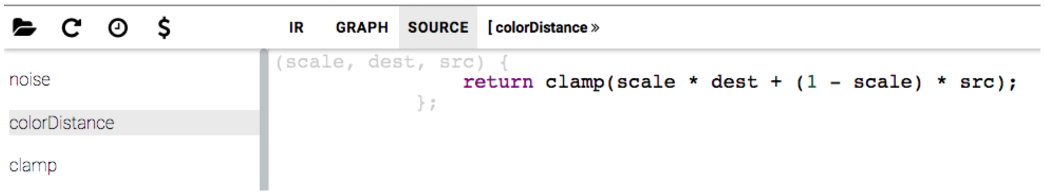
12.3.2 Parallel implementation generated code

Figures 25 through 28 are related to the parallel implementation. As it can be seen from them, only `Math.random` is being inlined.



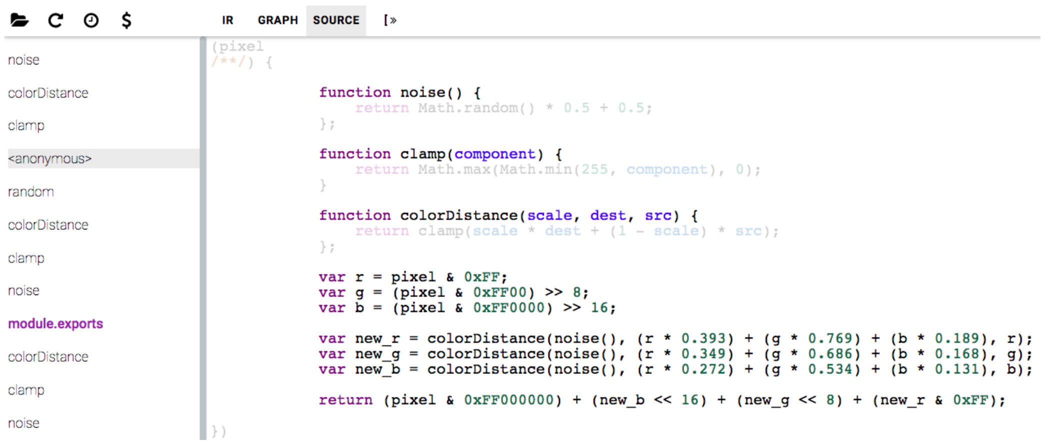
```
(component) {
            return Math.max(Math.min(255, component), 0);
        }
```

Figure 25: `clamp` function - parallel implementation



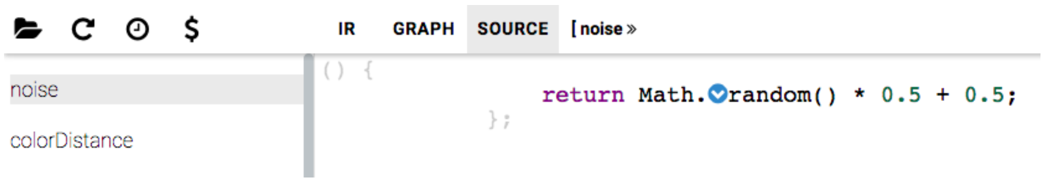
```
(scale, dest, src) {
            return clamp(scale * dest + (1 - scale) * src);
        };
```

Figure 26: `colorDistance` function - parallel implementation



```
(pixel
/**/) {

                function noise() {
                    return Math.random() * 0.5 + 0.5;
                };

                function clamp(component) {
                    return Math.max(Math.min(255, component), 0);
                }

                function colorDistance(scale, dest, src) {
                    return clamp(scale * dest + (1 - scale) * src);
                };

                var r = pixel & 0xFF;
                var g = (pixel & 0xFF00) >> 8;
                var b = (pixel & 0xFF0000) >> 16;

                var new_r = colorDistance(noise(), (r * 0.393) + (g * 0.769) + (b * 0.189), r);
                var new_g = colorDistance(noise(), (r * 0.349) + (g * 0.686) + (b * 0.168), g);
                var new_b = colorDistance(noise(), (r * 0.272) + (g * 0.534) + (b * 0.131), b);

                return (pixel & 0xFF000000) + (new_b << 16) + (new_g << 8) + (new_r & 0xFF);
})
```

Figure 27: `mapper` function - parallel implementation



```
() {
                return Math.random() * 0.5 + 0.5;
            };
```

Figure 28: `noise` function - parallel implementation

12.4 The solution

The way to solve the problem was to manually inline the functions. The resulting parallel implementation was:

```
pjs(buff).map(function(pixel){
  var r = pixel & 0xFF;
  var g = (pixel & 0xFF00) >> 8;
  var b = (pixel & 0xFF0000) >> 16;
  var noiser = Math.random() * 0.5 + 0.5;
  var noiseg = Math.random() * 0.5 + 0.5;
  var noiseb = Math.random() * 0.5 + 0.5;

  var new_r = Math.max(Math.min(255, noiser * ((r * 0.393) +
      (g * 0.769) + (b * 0.189)) + (1 - noiser) * r), 0);
  var new_g = Math.max(Math.min(255, noiseg * ((r * 0.349) +
      (g * 0.686) + (b * 0.168)) + (1 - noiseg) * g), 0);
  var new_b = Math.max(Math.min(255, noiseb * ((r * 0.272) +
      (g * 0.534) + (b * 0.131)) + (1 - noiseb) * b), 0);

  return (pixel & 0xFF000000) + (new_b << 16) + (new_g << 8)
     + (new_r & 0xFF);
}).seq(function(result){
  // work on result
});
```

Listing 37: Sepia tone final parallel implementation

We created a benchmark to understand the difference between the initial version (the one in which functions were not inlined) and the final one (with inlining in charge of the developer):

```
// http://jsperf.com/pjs-map-inlining/7

// HTML setup
<script src="http://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p
   -j-s.min.js"></script>
<script>
  var pjs = require('p-j-s');
  pjs.init({ maxWorkers: 4});

  var generateElements = function (total) {
    var typed = new Uint32Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = i;
    }
    return typed;
  };

  var xs = generateElements(1000000); // 1.000.000

  var wrappedXs = pjs(xs);

  function notInlinedMapper(pixel) {
    function noise() {
        return Math.random() * 0.5 + 0.5;
    };

    function clamp(component) {
        return Math.max(Math.min(255, component), 0);
    }

    function colorDistance(scale, dest, src) {
        return clamp(scale * dest + (1 - scale) * src);
    };

    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;

    var new_r = colorDistance(noise(), (r * 0.393) + (g *
       0.769) + (b * 0.189), r);
    var new_g = colorDistance(noise(), (r * 0.349) + (g *
       0.686) + (b * 0.168), g);
    var new_b = colorDistance(noise(), (r * 0.272) + (g *
       0.534) + (b * 0.131), b);

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + (new_r & 0xFF);
  };

  function inlinedMapper(pixel) {
    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;
    var noise_r = Math.random() * 0.5 + 0.5;
    var noise_g = Math.random() * 0.5 + 0.5;
    var noise_b = Math.random() * 0.5 + 0.5;

    var new_r = Math.max(Math.min(255, noise_r * ((r * 0.393)
        + (g * 0.769) + (b * 0.189)) + (1 - noise_r) * r), 0)
       ;
    var new_g = Math.max(Math.min(255, noise_g * ((r * 0.349)
        + (g * 0.686) + (b * 0.168)) + (1 - noise_g) * g), 0)
       ;
    var new_b = Math.max(Math.min(255, noise_b * ((r * 0.272)
        + (g * 0.534) + (b * 0.131)) + (1 - noise_b) * b), 0)
       ;

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + (new_r & 0xFF);
  };

  function notInlinedTest() {
    wrappedXs.map(notInlinedMapper).seq(function (result) {
      if (!result) {
        console.log('error');
      }
      __finish();
    });
  };

  function inlinedTest() {
    wrappedXs.map(inlinedMapper).seq(function (result) {
      if (!result) {
        console.log('error');
      }
      __finish();
    });
  };
</script>

// Javascript setup
__finish = function () {
  deferred.resolve();
};

// Test case 1 - Not inlined map
notInlinedTest();

// Test case 2 - Inlined map
inlinedTest();
```

Listing 38: Benchmark at http://jsperf.com/pjs-map-inlining/7

The results for it are displayed in Figure 29.

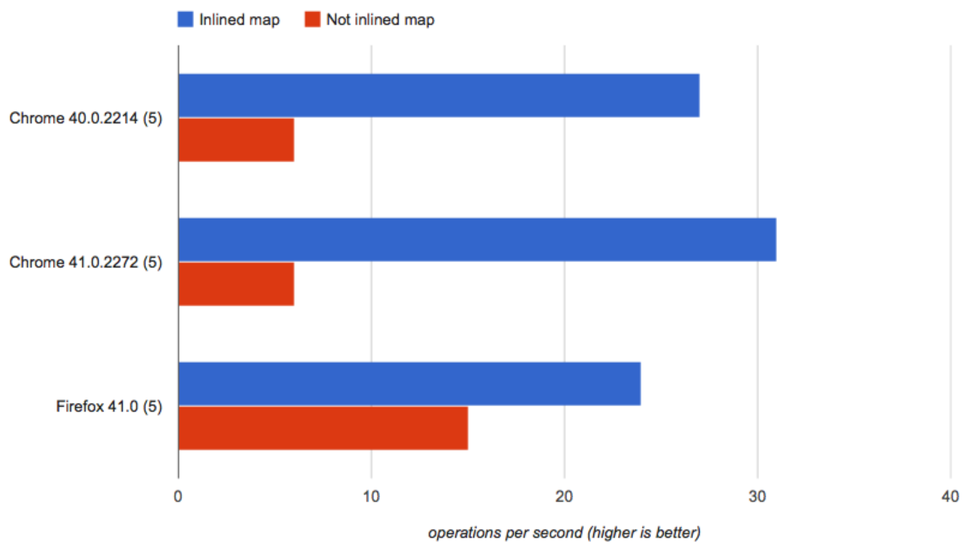


Figure 29: Comparing a version of `map` without inlining against a developer inlined one

12.5 Conclusion

This section shows that in those cases in which the compiler inlines some operations that are executed as part of the skeleton, it is important for developers to explicitly inline those when using the library.

13 Function caching

Before comparing the sequential implementation of `map` against the parallel one, we believed that the cost of dynamically instantiating a `Function` from a `String` in the Web Workers could be avoided by introducing a cache for cases where the Web Worker has already seen the code for a particular function.

We decided to introduce a simple key/value map acting as a cache where the keys are `String`s with the function's code and the values are the `Function` objects.

To verify its performance we created a benchmark that runs both with and without the function cache:

```
// http://jsperf.com/p-j-s-with-vs-without-function-cache/4

// HTML setup
<script src="http://rawgit.com/pjsteam/pjs/1
    bf39be3012cedf13fdf2f85bd4b1197eaabde96/dist/p-j-s.min.js"
    ></script>
<script src="http://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p
    -j-s.min.js"></script>
<script>
  var __finish;
  var pjs = require('p-j-s-no-cache');
  var pjsCacheFunction = require('p-j-s');
  pjs.init({ maxWorkers: 4});
  pjsCacheFunction.init({ maxWorkers: 4});

  // JavaScript setup
  var generateElements = function (total) {
    var typed = new Uint32Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = i;
    }
    return typed;
  };

  var xs = generateElements(1000000); // 1.000.000

  var wrappedNoCacheXs = pjs(xs);
  var wrappedCacheXs = pjsCacheFunction(xs);

  function mapper(pixel) {
    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;
    var noise_r = Math.random() * 0.5 + 0.5;
    var noise_g = Math.random() * 0.5 + 0.5;
    var noise_b = Math.random() * 0.5 + 0.5;

    var new_r = Math.max(Math.min(255, noise_r * ((r * 0.393)
        + (g * 0.769) + (b * 0.189)) + (1 - noise_r) * r), 0)
       ;
    var new_g = Math.max(Math.min(255, noise_g * ((r * 0.349)
        + (g * 0.686) + (b * 0.168)) + (1 - noise_g) * g), 0)
       ;
    var new_b = Math.max(Math.min(255, noise_b * ((r * 0.272)
        + (g * 0.534) + (b * 0.131)) + (1 - noise_b) * b), 0)
       ;

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + (new_r & 0xFF);
  };

  function mapCallback (result) {
    if (!result) {
      console.log('error');
    }
    __finish();
  };
</script>

// javascript setup
__finish = function(){
  deferred.resolve();
}

// Test case 1 - Without function cache
wrappedNoCacheXs.map(mapper).seq(mapCallback);

// Test case 2 - Function cache
wrappedCacheXs.map(mapper).seq(mapCallback);
```

Listing 39: Benchmark at http://jsperf.com/p-j-s-with-vs-without-function-cache/4

The results for the benchmark are displayed in Figure 30.

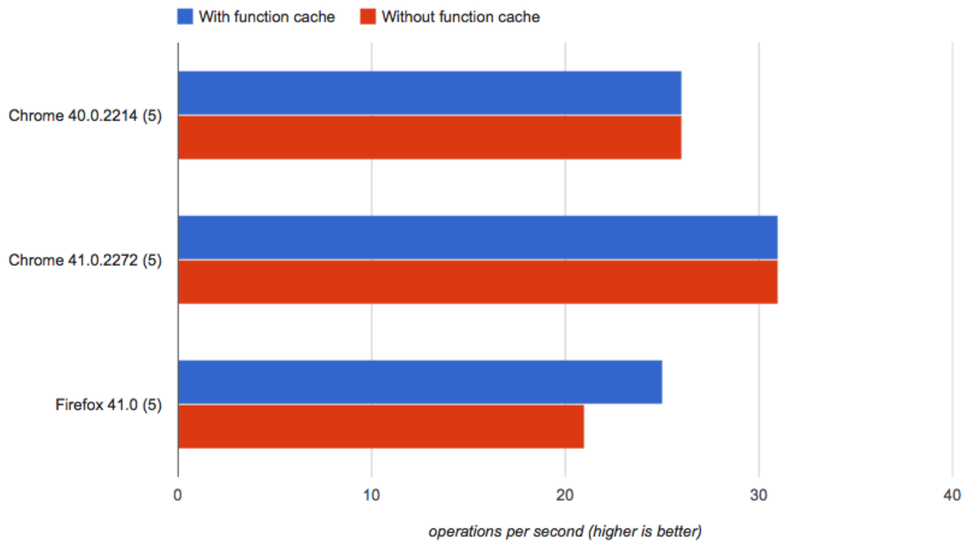


Figure 30: Comparing p-j-s `map` with and without a function cache

13.1 Conclusion

As the benchmark shows, there is almost no difference in versions 40 and 41 of Google Chrome between the implementations with and without the cache. To determine whether introducing the cache was worth it or not we considered the result from Firefox Nightly, which shows a big increase in performance (near 25%). That is why we decided to make the `Function` cache a part of our implementation.

14 `map`: Sequential vs parallel

Since as explained in Section 6 the cost of transferring objects is not zero, it is important to compare the performance of our parallel implementation against a serial one.

In this regard, the more complex the computation to be performed on each array element, the better the parallel implementation will perform. That is because, in the parallel case, the source `TypedArray` must be copied to sub-arrays in order to be transferred. On the contrary, the serial implementation operates on elements during that time. To observe the effects of this difference, we also perform the same benchmark using `SharedTypedArrays`.

14.1 Algorithm selection

For this benchmark 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 computationally complex.

14.2 TypedArray Benchmark

The benchmark code is introduced in the following code listing:

```
// http://jsperf.com/pjs-map-vs-serial/8

// HTML preparation
<script src="http://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p
    -j-s.min.js"></script>
<script>
  var pjs = require('p-j-s');
  pjs.init({ maxWorkers: 4});

  var generateElements = function (total) {
    var typed = new Uint32Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = 0xdddddddd;
    }
    return typed;
  };
  var xs100 = generateElements(100);
  var wrappedXs100 = pjs(xs100);
  var xs1000 = generateElements(1000);
  var wrappedXs1000 = pjs(xs1000);
  var xs10000 = generateElements(10000);
  var wrappedXs10000 = pjs(xs10000);
  var xs100000 = generateElements(100000);
  var wrappedXs100000 = pjs(xs100000);
  var xs1000000 = generateElements(1000000);
  var wrappedXs1000000 = pjs(xs1000000);

  function mapper(pixel) {
    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;

    var noise_r = Math.random() * 0.5 + 0.5;
    var noise_g = Math.random() * 0.5 + 0.5;
    var noise_b = Math.random() * 0.5 + 0.5;

    var new_r = Math.max(Math.min(255, noise_r * ((r * 0.393)
        + (g * 0.769) + (b * 0.189)) + (1 - noise_r) * r), 0)
       ;
    var new_g = Math.max(Math.min(255, noise_g * ((r * 0.349)
        + (g * 0.686) + (b * 0.168)) + (1 - noise_g) * g), 0)
       ;
    var new_b = Math.max(Math.min(255, noise_b * ((r * 0.272)
        + (g * 0.534) + (b * 0.131)) + (1 - noise_b) * b), 0)
       ;

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + (new_r & 0xFF);
  };

  function serialMap(pixels, l) {
    var result = new Uint32Array(l);
    for (var i = 0; i < l; i++) {
      var pixel = pixels[i];
      var r = pixel & 0xFF;
      var g = (pixel & 0xFF00) >> 8;
      var b = (pixel & 0xFF0000) >> 16;

      var noise_r = Math.random() * 0.5 + 0.5;
      var noise_g = Math.random() * 0.5 + 0.5;
      var noise_b = Math.random() * 0.5 + 0.5;

      var new_r = Math.max(Math.min(255, noise_r * ((r *
         0.393) + (g * 0.769) + (b * 0.189)) + (1 - noise_r)
         * r), 0);
      var new_g = Math.max(Math.min(255, noise_g * ((r *
         0.349) + (g * 0.686) + (b * 0.168)) + (1 - noise_g)
         * g), 0);
      var new_b = Math.max(Math.min(255, noise_b * ((r *
         0.272) + (g * 0.534) + (b * 0.131)) + (1 - noise_b)
         * b), 0);

      result[i] = (pixel & 0xFF000000) + (new_b << 16) + (
         new_g << 8) + (new_r & 0xFF);
    }
    return result;
  };

  function runSerial(xs) {
    var r = serialMap(xs, xs.length);
    if (0 === r.length) {
      console.log('ups serial');
    }
  };

  function runPjs(wrappedXs) {
    wrappedXs.map(mapper).seq(function (err, r) {
      __finish();
    });
  };
</script>

// Javascript setup
__finish = function () {
  deferred.resolve();
};

// Test Case - serial map 100
runSerial(xs100);

// Test Case - pjs map 100
runPjs(wrappedXs100);

// Test Case - serial map 1,000
runSerial(xs1000);

// Test Case - pjs map 1,000
runPjs(wrappedXs1000);

// Test Case - serial map 10,000
```

```
runSerial(xs10000);

// Test Case - pjs map 10,000
runPjs(wrappedXs10000);

// Test Case - serial map 100,000
runSerial(xs100000);

// Test Case - pjs map 100,000
runPjs(wrappedXs100000);

// Test Case - serial map 1,000,000
runSerial(xs1000000);

// Test Case - pjs map 1,000,000
runPjs(wrappedXs1000000);
```

Listing 40: Benchmark at http://jsperf.com/pjs-map-vs-serial/8

Note: Unlike other benchmarks in which we were comparing different ways to achieve the same result, this benchmark compares a single way with a varying amount of elements. For that reason, the original charts are not displayed.

The results for the benchmark are displayed in Figure 31.

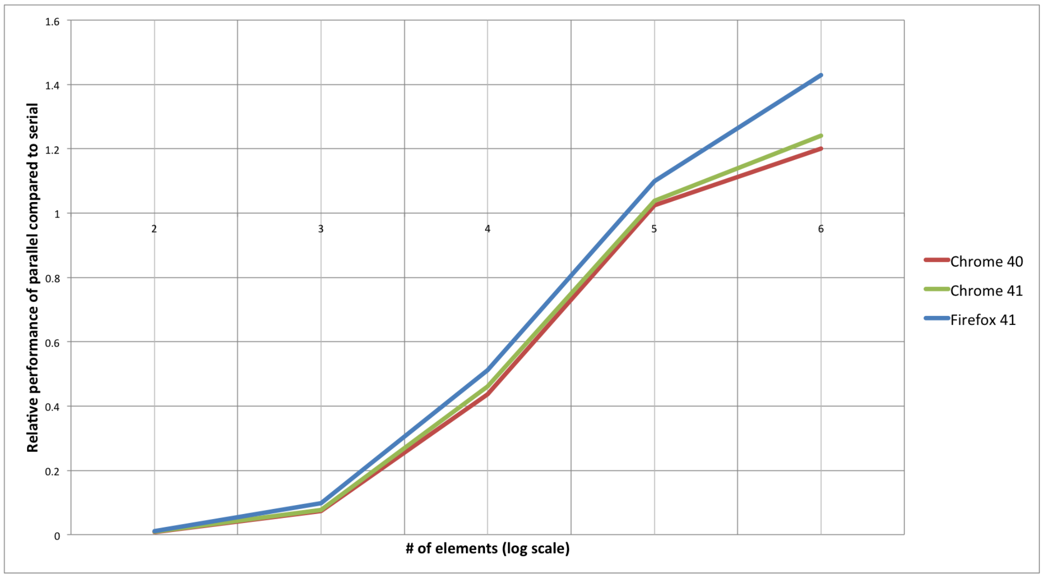


Figure 31: Relative performance of parallel implementation with regards to sequential one using `ArrayBuffers`

14.3 SharedTypedArray Benchmark

The benchmark code is introduced in the following code listing:

```
// http://jsperf.com/pjs-map-vs-serial/9

// HTML preparation
<script src="http://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p
   -j-s.min.js"></script>
<script>
  var pjs = require('p-j-s');
  pjs.init({ maxWorkers: 4});

  var generateElements = function (total) {
    var typed = new SharedUint32Array(total);
    for (var i = total; i > 0; i--){
      typed[i - 1] = 0xdddddddd;
    }
    return typed;
  };
  var xs100 = generateElements(100);
  var wrappedXs100 = pjs(xs100);
  var xs1000 = generateElements(1000);
  var wrappedXs1000 = pjs(xs1000);
  var xs10000 = generateElements(10000);
  var wrappedXs10000 = pjs(xs10000);
  var xs100000 = generateElements(100000);
  var wrappedXs100000 = pjs(xs100000);
  var xs1000000 = generateElements(1000000);
  var wrappedXs1000000 = pjs(xs1000000);

  function mapper(pixel) {
    var r = pixel & 0xFF;
    var g = (pixel & 0xFF00) >> 8;
    var b = (pixel & 0xFF0000) >> 16;

    var noise_r = Math.random() * 0.5 + 0.5;
    var noise_g = Math.random() * 0.5 + 0.5;
    var noise_b = Math.random() * 0.5 + 0.5;

    var new_r = Math.max(Math.min(255, noise_r * ((r * 0.393)
        + (g * 0.769) + (b * 0.189)) + (1 - noise_r) * r), 0)
       ;
    var new_g = Math.max(Math.min(255, noise_g * ((r * 0.349)
        + (g * 0.686) + (b * 0.168)) + (1 - noise_g) * g), 0)
       ;
    var new_b = Math.max(Math.min(255, noise_b * ((r * 0.272)
        + (g * 0.534) + (b * 0.131)) + (1 - noise_b) * b), 0)
       ;

    return (pixel & 0xFF000000) + (new_b << 16) + (new_g <<
       8) + (new_r & 0xFF);
  };

  function serialMap(pixels, l) {
    var result = new Uint32Array(l);
    for (var i = 0; i < l; i++) {
      var pixel = pixels[i];
      var r = pixel & 0xFF;
      var g = (pixel & 0xFF00) >> 8;
      var b = (pixel & 0xFF0000) >> 16;

      var noise_r = Math.random() * 0.5 + 0.5;
      var noise_g = Math.random() * 0.5 + 0.5;
      var noise_b = Math.random() * 0.5 + 0.5;

      var new_r = Math.max(Math.min(255, noise_r * ((r *
         0.393) + (g * 0.769) + (b * 0.189)) + (1 - noise_r)
         * r), 0);
      var new_g = Math.max(Math.min(255, noise_g * ((r *
         0.349) + (g * 0.686) + (b * 0.168)) + (1 - noise_g)
         * g), 0);
      var new_b = Math.max(Math.min(255, noise_b * ((r *
         0.272) + (g * 0.534) + (b * 0.131)) + (1 - noise_b)
         * b), 0);

      result[i] = (pixel & 0xFF000000) + (new_b << 16) + (
         new_g << 8) + (new_r & 0xFF);
    }
    return result;
  };

  function runSerial(xs) {
    var r = serialMap(xs, xs.length);
    if (0 === r.length) {
      console.log('ups serial');
    }
  };

  function runPjs(wrappedXs) {
    wrappedXs.map(mapper).seq(function (err, r) {
      __finish();
    });
  };
</script>

// Javascript setup
__finish = function () {
  deferred.resolve();
};

// Test Case - serial map 100
runSerial(xs100);

// Test Case - pjs map 100
runPjs(wrappedXs100);

// Test Case - serial map 1,000
runSerial(xs1000);

// Test Case - pjs map 1,000
runPjs(wrappedXs1000);

// Test Case - serial map 10,000
runSerial(xs10000);

// Test Case - pjs map 10,000
runPjs(wrappedXs10000);

// Test Case - serial map 100,000
runSerial(xs100000);

// Test Case - pjs map 100,000
runPjs(wrappedXs100000);

// Test Case - serial map 1,000,000
runSerial(xs1000000);

// Test Case - pjs map 1,000,000
runPjs(wrappedXs1000000);
```

Listing 41: Benchmark at http://jsperf.com/pjs-map-vs-serial/9

The results for the benchmark are displayed in Figure 32.

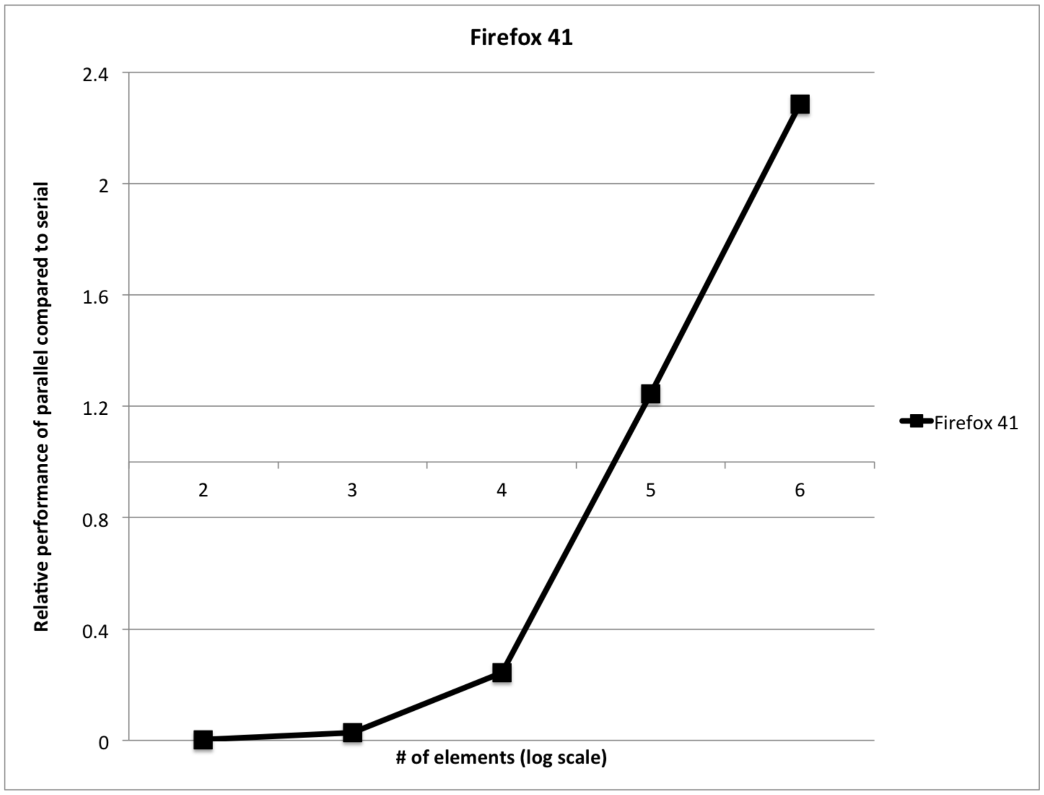


Figure 32: Relative performance of parallel implementation with regards to sequential one using `SharedArrayBuffer`s

14.4 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 **2.4x** can be achieved with a relatively large amount of items ($10E6$) and performing a computationally expensive operation (generating pseudo-random numbers). Larger speedups might 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 fully optimized, being that they are only an experimental feature in Firefox Nightly.

The speedup obtained without shared memory is relative small (approximately **1.2x**) when compared to that obtained with shared memory. That can be explained by the overhead of copying array elements, to work around the lack of shared memory, when communicating with the Web Workers.

15 Modules system

Note: At the time of writing, neither Google Chrome nor Mozilla Firefox support the ES6 module specification so that is not a viable choice.

The two most known module formats/API in JavaScript are:

CommonJS Recognized for being used in node.js [27].

AMD [28] The most known implementation being require.js [29].

CommonJS has many similar aspects with the ES6 module specification, although it does not provide a way to load modules asynchronously. AMD is very different from the ES6 module specification but does support asynchronous module loading which is useful for browser applications.

Instead of making the choice for users, our library will support both systems in a way that makes it easy for users to consume either of them.

15.1 CommonJS support

All that needs to be done to use the library is shown in the following code snippet:

```
<script src="https://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/
   p-j-s.min.js"></script>

<script>
  var pjs = require('p-j-s');
  pjs.init();

  /* use below */
</script>
```

Listing 42: CommonJS usage

15.2 AMD support

This example shows how to use the library with require.js. It assumes a folder structure like the one in the following listing:

```
.
|__ index.html
```

```
|__ scripts
    |__ main.js
    |__ require.js
```

Listing 43: AMD folder structure

The file **main.js** must have the following contents:

```
require(["https://rawgit.com/pjsteam/pjs/v1.0.0-beta/dist/p-j
   -s-standalone.min.js"], function(pjs) {
  pjs.init();

  /* use below */
});
```

Listing 44: main.js AMD support

15.3 Conclusion

As the previous examples show, using our library with either module system is really simple and is done in a way that should feel familiar for users of those systems.

References

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," 1965.

[2] *Parallel Programming in the .NET Framework*, Microsoft Corp. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx

[3] EcmaScript. ECMA. [Online]. Available: http://www.ecmascript.org/

[4] asm.js. [Online]. Available: http://asmjs.org/

[5] WebGL. Khronos Group. [Online]. Available: http://www.khronos.org/webgl/

[6] J. Wang, N. Rubin and S. Yalamanchili, "Paralleljs: An execution framework for javascript on heterogeneous systems," 2014.

[7] S. Herhut, R. L. Hudson, T. Shpeisman, J. Sreeram, "A path to parallelism in javascript," 2013.

[8] A. Rauschmayer. (2013) Paralleljs: An execution framework for javascript on heterogeneous systems. [Online]. Available: http://www.2ality.com/2013/12/simd-js.html

[9] Web Workers. W3C. [Online]. Available: http://www.w3.org/TR/workers/

[10] Concurrency model and event loop. Mozilla Developer Network. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[11] v8. Google. [Online]. Available: https://developers.google.com/v8/intro

[12] SpiderMonkey. Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[13] Web Workers start-up. W3C. [Online]. Available: http://www.w3.org/TR/workers/

[14] D. C. Schmidt and S. Vinoski, "Comparing alternative programming techniques for multi-threaded corba servers," 1996.

[15] Javascript shared memory, atomics, and locks. [Online]. Available: http://bit.ly/1FxLqV0

[16] (2015, April) ECMA-262 6th Edition (Draft). [Online]. Available: http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

[17] p-j-s api. [Online]. Available: https://github.com/pjsteam/pjs/wiki/API

[18] Promise. Mozilla Developer Network. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[19] V. Egorov. The black cat of microbenchmarks. [Online]. Available: http://mrale.ph/blog/2014/02/23/the-black-cat-of-microbenchmarks.html

[20] ——. IRHydra. [Online]. Available: http://mrale.ph/irhydra/2

[21] HTML Standard. WHATWG. [Online]. Available: https://html.spec.whatwg.org

[22] J. Epstein, A. P. Black, S. Peyton-Jones, "Towards Haskell in the Cloud," 2011. [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf

[23] A. Schwendner, "Distributed functional programming in scheme," Master's thesis, Massachusetts Institute of Technology, 2010. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2010/alexrs-meng-thesis.pdf

[24] Encoding Standard. WHATWG. [Online]. Available: https://encoding.spec.whatwg.org

[25] S. J. Randy Allen, "Compiling C for vectorization parallelism and inline expansion," 1988.

[26] T. C. W. H. William Chen, Pahua Chang, "The Effect of Code Expanding Optimizations on Instruction Cache Design," April 1991.

[27] node.js. [Online]. Available: https://nodejs.org

[28] Asynchronous Module Definition API. [Online]. Available: https://github.com/amdjs/amdjs-api/blob/master/AMD.md

[29] require.js, a JavaScript module loader. [Online]. Available: http://requirejs.org