# Parallel Programming for the Web

Stephan Herhut    Richard L. Hudson    Tatiana Shpeisman    Jaswanth Sreeram
Intel Labs
{stephan.a.herhut, rick.hudson, tatiana.shpeisman, jaswanth.sreeram}@intel.com

## Abstract

Parallel hardware is today's reality and language extensions that ease exploiting its promised performance flourish. For most mainstream languages, one or more tailored solutions exist that address the specific needs of the language to access parallel hardware. Yet, one widely used language is still stuck in the sequential past: JavaScript, the *lingua franca* of the web.

Our position is that existing solutions do not transfer well to the world of JavaScript due to differences in programming models, the additional requirements of the web, like safety, and to developer expectations. To address this we propose River Trail, a new parallel programming API designed specifically for JavaScript and we show how it satisfies the needs of the web. To prove that our approach is viable, we have implemented a prototype JIT compiler in Firefox that shows an order of magnitude performance improvement for a realistic web application.

## 1 Introduction

Despite Wired magazine proclaiming the Web dead two years ago [1], it is alive and well. Browser-based applications written in HTML and JavaScript continue to flourish. Moreover, HTML and JavaScript are also emerging as a popular development platform for stand-alone applications, especially for smart phones and tablets [2, 3]. Part of HTML's popularity is due to its growing capabilities. 3D graphics, audio, video, web cam input, geo location, offline storage, video conferencing - all are features traditionally reserved for native applications that are proposed for or will be part of HTML5 [4] or WebGL [9], the upcoming standards for the next iteration of the web. One capability, however, remains exclusive to native application: the ability to take advantage of parallel hardware.

Parallel hardware is an everyday reality, across all vendors and form factors. All major processors support vector instructions and a majority of them come with multiple cores. Parallel software is also slowly moving to mainstream, as various parallel programming features and tools are emerging for most popular programming languages. Yet, JavaScript applications remain predominately sequential.

This lack of parallel web applications is not due to the lack of demand for more computing power for the client side of the web. Many applications, such as photo and video editing, physics-based games, augmented reality, financial software and data visualization could readily benefit from unused compute cycles available in parallel client hardware, as they have abundant latent data parallelism. The problem lies in the lack of appropriate parallel programming models.

The web developer community has refused to adopt traditional multi-threaded shared memory parallel programming models claiming that such models are too dangerous for their domain. While delivering performance, shared memory programming comes together with classical pitfalls of data races, dead locks and live locks, all of which can easily lead to hard to reproduce concurrency bugs - something neither web developers nor users want as part of the web browsing experience. Web workers [13], the only widely adopted support for parallel compute in JavaScript that bring actor style threads to the web, were carefully designed to steer around all these issues. While they achieve their design goal of offloading long running computations to background threads, they are not suitable for the development of parallel scalable compute intense workloads due to high cost of communication and low level of abstraction.

We believe that parallelism should not be reserved to expert programmers writing in C, C++, or Java. Web developers should be able to exploit parallel hardware without fundamentally changing their programming style. In particular, they should not need to learn a new language or adapt to different semantics.

River Trail puts this belief into action. Building on well known data-parallel programming techniques [6, 5], we have designed an API that makes expressing parallelism easy, sacrificing performance for productivity where need be. We have not extended JavaScript's semantics, nor introduced fundamentally new concepts. The API can be fully implemented in JavaScript, albeit without performance improvements. However, River Trail is carefully crafted so that it can easily be compiled to a broad range of parallel hardware, from vector units to multi-core CPUs. We have proven this with an open source prototype, written for the popular Firefox web browser, that exploits vector instructions and multiple cores to achieve an order of magnitude speedup on an off-the-shelf desktop system.

## 2 JavaScript

*JavaScript* is the language behind today's web applications. It was developed specifically for client side scripting on the open web, a purpose that has heavily influenced its design: Safety and portability are two of the key concerns.

The safety requirement stems from the particulars of the open web as an application platform. Whereas traditionally, a user would have to acquire software, either by download or a physical medium, install it on his local machine and run it, web applications are instant on. A single click on a URL suffices to start an application, and thus run JavaScript code. Even more, the origin of an application is often not apparent, and thus trust cannot easily be established.

Portability is a requirement for any open platform. Web applications in particular need to run across devices, spanning different architectures and form factors. As a consequence, JavaScript is completely hardware agnostic.

Other than the name suggests, JavaScript is not related to Java. Both share the above design goals and they can be categorized as object oriented languages. Yet, JavaScript is dynamically typed and has no notion of classes. Instead, it employs a form of prototype based inheritance in the spirit of Self [11]. This design is particularly well suited for rapid and iterative application development, a style that is prevalent in web applications. Software is released early and released often, resolving bugs and issues as they arise.

Until today, JavaScript has remained mostly sequential and program execution is fully deterministic. The latter forces implementations to essentially halt the world while scripts execute. To avoid unresponsive browsers, typical JavaScript programs therefore make heavy use of callbacks and asynchronicity. Larger tasks, in particular those involving long latency operations, are decomposed into independent chunks that are executed atomically and only ensure a consistent state when yielding control. This cooperative multi-tasking style has trained programmers to design concurrent programs while relying on deterministic sequential execution.

So in essence, JavaScript can be described as *safe*, *portable*, *rapid prototyping* friendly and *deterministic*. A parallel programming extension for JavaScript thus should strive to maintain these properties.

## 3 River Trail Design

River Trail builds on well known principles of data-parallel programming drawing inspiration from [6, 5, 8]. The design is based on three pillars, a type called ParallelArray that holds data values, several prototypical methods of ParallelArray that implement parallel constructs like map, and the concept of an elemental function which is passed to the constructs and typically returns a single data element. In the simple example below, `in` and `out` are two parallel arrays - `in` created using new and `out` computed by invoking the prototype method map with the elemental function `inc1` to create a freshly minted ParallelArray with each element in pa incremented by 1.

```
function inc1(val) {return val + 1.0;};
var in =
    new ParallelArray([1.0, 2.0, 3,0]);
var out = in.map(inc1);
```

The next example shows a pairwise add of two ParallelArrays, `pa1` and `pa2`. First, the example creates a function that takes a ParallelArray as an argument and returns a function that adds the values located at index *i* of the *this* array and the *otherPA* array. Then it calls a combine method that produces a new parallel array with each element being a result of invoking the generated elemental function on the corresponding elements of `pa1` and `pa2`. Notice how this example leverages closures, free variables, and the object-oriented JavaScript programming style.

```
function pairwiseAdd(otherPA){
  return function(i){
           return this[i]+otherPA[i];}
};
out = pa1.combine(pairwiseAdd(pa2));
```

ParallelArray comes with multiple constructors, including a comprehension, and the following 5 data parallel methods: *map*, *combine*, *scan*, *filter*, and *scatter*. When combined with elemental functions each of these methods creates a freshly minted ParallelArray. ParallelArray's sixth core data parallel method is *reduce* which typically returns a scalar value. We have chosen a small set of constructs that compose well and can be used to create other data parallel constructs. For example, *gather* can be implemented using a comprehension parallel array constructor, while *sum* could be implemented using *reduce*. This approach enables a *do few things well* implementation strategy which reduces the complexity of the compiler and increases our confidence in its correctness while anticipating the creation of useful libraries and infrastructures.

We now show how our design meets JavaScript's properties of safety, portability, rapid prototyping friendliness, and deterministic execution.

## 3.1 Safety

The burden of ensuring safety and security in C/C++ based parallel programming models is placed upon the programmer. Given the trusted environment and performance requirements historically found in HPC environments this is appropriate. For the web environment, however, safety and security are fundamental requirements. River Trail achieves the safety level of JavaScript by simply expressing all the parallel constructs in JavaScript. Because all the parallel code is written in JavaScript it provides exactly the same safety and security guarantees as sequential JavaScript code. In fact, the whole API can be implemented as a JavaScript library, although without the performance benefits of parallel execution. While seemingly obvious, this approach is in a sharp contrast with alternative proposals, such as, WebCL [12] that suggests incorporating OpenCL with all its security pitfalls directly into HTML code.

## 3.2 Portability

ParallelArray methods can successfully execute on any platform that supports execution of standard JavaScript. If parallel execution is not possible, elemental functions simply execute sequentially using the provided sequential library. This provides functional portability across all browsers. Performance portability is a much more difficult problem and we do not claim to have completely solved it. The strategy and effectiveness of parallel execution will depend on the particular platform, as it is not possible to write parallel code that will have optimal performance on all forms and factors of available client devices. Consequently, we have chosen a high-level data-parallel programming approach [6, 5], featuring well known primitives like *map* and *reduce*. This forces the programmer to express the structure of a parallel algorithm and leaves platform dependent execution strategies to the specific runtime system.

## 3.3 Rapid Prototyping

River Trail preserves JavaScript friendliness to rapid prototyping by staying within the boundaries of the same programming language. ParallelArray is nothing more than a new JavaScript type and it behaves as the JavaScript programmer expects. An elemental function is just a JavaScript function; it can be used both as an argument to a parallel array prototype method and in any other way legal in JavaScript. We have not added unfamiliar syntax, low level intrinsics, nor elements of other programming languages. Our APIs look like JavaScript and are JavaScript, thus, allowing JavaScript developers to continue programming in a familiar style and freely move code between River Trail and non River Trail parts of the application.

## 3.4 Deterministic Execution

JavaScript programmers are used to a sequential programming model. They understand that events can fire asynchronously and that the order in which events are handled may not be deterministic. However, once inside JavaScript code they expect deterministic execution without concern for race conditions, memory models, locks, or any of the other devils that plague traditional parallel programming models.

One source of non-determinism is concurrent modification of shared state. To ensure deterministic execution, River Trail requires elemental functions to be side effect free, e.g., they may not mutate non-local variables. An implementation of River Trail will detect violations of this property, either statically or dynamically, and raise an exception. The current prototype detects violations statically during compilation of the elemental function.

Another source of non-determinism is more difficult to completely avoid: The execution order in reduce and scan operations may influence the result. Using a fixed execution order limits available implementation strategies. We have chosen a middle ground. River Trail only guarantees deterministic results for reduce and scan if the used elemental function is associative and commutative.

## 4 Experimental Evaluation

We have prototyped the River Trail compiler and library as an extension to the Firefox web browser (version 9.0.1). The prototype uses a three stage just-in-time compilation process. First, the JavaScript code is analyzed and translated to OpenCL. We have only implemented the bare essentials required for this step, i.e., type inference, bounds checks elimination, and static heap management. Our prototype compiler is independent from the system's JIT. Further optimisation and integration with the systems JIT remains future work. In a second step, we then use the Intel OpenCL SDK (version 1.5) to generate binary code. In the third step OpenCL runtime executes the code. OpenCL is responsible for the number of threads created, whether vector instructions are used, and other runtime optimizations. This process is completely transparent to the web-developer as well as to the user.

We have ported several programs to the River Trail API including a dense matrix multiply kernel and a full-fledged 3D web application, both of which we discuss below. While we have compiled elemental functions with around one thousand lines, the matrix multiple elemental function is around ten lines and the 3D web application's elemental functions are around one hundred lines. The entire 3D web application is of course much larger. The complete River Trail system as well as these and other programs along with their input sets are freely available at [10].

### 4.1 Dense Matrix-Multiply

Dense matrix-multiplication is an important kernel for many emerging web applications such as 3D animation, in-browser physics and video processing. We have implemented the standard O($n^3$) multiplication algorithm with the River Trail API using ParallelArray objects and, for comparison, in standard sequential JavaScript and C using the languages' respective standard array types.

We also implemented variants of these kernels that represent the matrices using flat arrays instead of nested arrays (1D and 2D). Finally, we have created a JavaScript implementation that relies on typed arrays. Typically, arrays in JavaScript are untyped and thus may contain elements of different types. This leads to a boxed representation of values which adds further overheads. Typed arrays are a recent addition to JavaScript that have semantics similar to C arrays. In particular, the elements of these arrays all have to be of a single statically known type, thereby allowing for an unboxed continuous storage layout. The reduced overhead is reflected in the speed up shown in Figure 1.

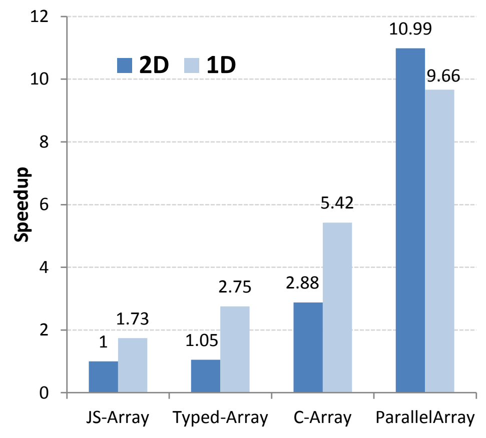

Figure 1: Dense Matrix-Multiply performance for $1k \times 1k$ matrices

Experiments were run for $1k \times 1k$ matrices on a machine with a hyperthreading-enabled 2nd generation Intel Core i5 dual core (2.5GHz) processor and 4GB RAM. The Y-axis in this figure indicates the speed up of the different kernels over the kernel that uses two-dimensional JavaScript arrays. Without further optimisations, the River Trail implementation outperforms the JavaScript base case by a factor of 10.9 and 9.6 for 2D and 1D storage layouts, respectively. Even the best JavaScript implementation is more than 3.5 times slower. The improvements come not only from parallelization but also from the fact the River Trail JIT knows the type and shape of its inputs and uses type inference to deduce the types all variables as well as the iteration space. This information is comparable to what a C compiler has available. Not surprisingly a comparison with the naïve C implementation further shows that our speed up is not only due to JavaScript overheads: River Trail outperforms sequential C by about a factor of two.

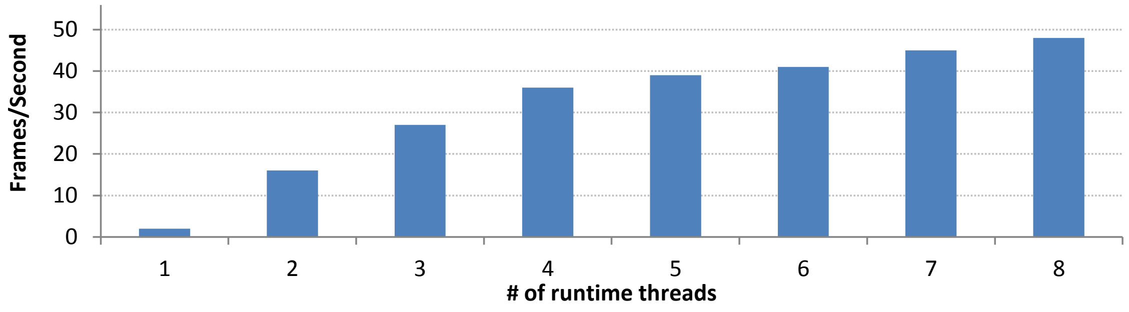

Figure 2: Frame rates for sequential and River Trail execution of a 3D in-browser particle flocking simulation

### 4.2 3D Particle Flocking

Apart from small kernels, we have studied full web applications. We took a large realistic 3D animation application and ported it to River Trail. This program implements $n$-body physics between a set of bodies (where $n$ is user-defined). Each body exerts a repulsive/attractive force on all other bodies and the bulk of the computation in each time step is for calculating the new velocity and position of each body under these accumulated forces. This computation is a good fit for the *combine* primitive. We use an elemental function that computes the cumulative forces acting on a single body, computes and returns the velocity (and position) of that body. The motion of the bodies and the scene itself are continuously animated using WebGL [9]. The application can dynamically switch between sequential and River Trail execution.

Figure 2 shows the frame rates (Y-axis) using a hyperthreading-enabled quad-core 2nd generation Core i7 running at 3.4 GHz with 4GB of usable RAM. To measure scaling behavior, we have limited the number of threads (X-axis) employed by River Trail. Due to a technical limitation in the used OpenCL stack, we cannot measure single thread performance of the River Trail implementation. Instead, the left most bar gives the frames per second rendered by the sequential JavaScript implementation.

Many factors outside the domain of River Trail influence system behavior, including the interplay with WebGL. While the results varied from run to run they clearly show that River Trail achieves significant speed up over the sequential case and scales well with increasing number of cores. We attribute some of the two thread speed up over the sequential version to better code generation by our specialized compiler compared to the code generated by the browser's more general purpose just-in-time compiler. Also, note that we have used a hyperthreaded processor, so scalability beyond four threads is limited. Furthermore, Firefox spawns some additional threads which leads to scheduling contention.

## 5 Related Work

*Web workers* [13], an API that extends the web platform with actor-style concurrency primitives, is currently being standardized. Compared to River Trail, web workers provide lower level concurrency constructs aimed at a diffrent level of granularity. River Trail uses a fine grained data-parallel model and on purpose avoids the concept of threads and related issues like mapping and scheduling. Although generally possible, an implementation of River Trail using web workers would have to address those issues and bridge the granularity gap.

*WebCL*[12], a JavaScript wrapper around OpenCL [8], is closer related to River Trail's programming model but less tailored to the web. WebCL is less structured than River Trail, *i.e.*, it does not provide data-parallel primitives like *reduce* or *scatter*. Instead, it essentially provides a parallel for loop without guarantees on side effects or determinism. Computations are expressed in C and many safety aspects of JavaScript do not apply.

Google's *NaCl* [7] addresses the problem at a even lower level than web workers. It allows slightly instrumented machine code to run in the browser. The use of static analyses and code instrumentation ensures safe execution. As in web workers, message passing is used to communicate between NaCl and JavaScript, thus decoupling the two worlds and ensuring deterministic execution of the JavaScript. However, by using machine code as interchange format, NaCl applications are limited to a specific platform.

## 6 Conclusion

River Trail shows that a data parallel programming API and a data parallel programming model can live comfortable in JavaScript. We have shown how River Trail's approach meets the four requirements of the web programmer: the requirement to provide safety and security, the requirement of deterministic execution, the requirement of maintaining a single programming model to preserve rapid prototyping, and finally the requirement of portability. The River Trail prototype shows that the API is feasible and can be efficiently implemented.

## References

[1] "The Web is Dead. Long Live the Internet." Chris Anderson, in Wired Magazine, `http://www.wired.com/magazine/2010/08/ff_webrip/all/1`

[2] "Boot To Gecko." `https://wiki.mozilla.org/B2G`

[3] "Project Spartan: Facebook's Hush-Hush Plan to Take On Apple On Their Own Turf: iOS." Mig Siegler, `http://techcrunch.com/2011/06/15/facebook-project-spartan/`

[4] "HTML5. A vocabulary and associated APIs for HTML and XHTML". `http://dev.w3.org/html5/spec/Overview.html`

[5] "Data parallel algorithms" W. Daniel Hillis, Guy L. Steele, Jr., in Communications of the ACM Volume 29 Issue 12, December 1986.

[6] "NESL: A Nested Data-Parallel Language", Guy E. Blelloch, Carnegie Mellon, Technical Report CMU-CS-95-170, September 1995.

[7] "Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code", Jason Ansel, Petr Marchenko, lfar Erlingsson, Elijah Taylor, Brad Chen, Derek Schuff, David Sehr, Cliff L. Biffle, Bennet S. Yee, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2011.

[8] "The OpenCL Specification", Khronos OpenCL Working Group, Aaftab Munshi, `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`

[9] "The WebGL Specification", Khronos OpenCL Working Group, Chris Marrin, `http://https://www.khronos.org/registry/webgl/specs/1.0/`

[10] "River Trail prototype implementation" `www.github.com/rivertrail/rivertrail/wiki`

[11] "The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages", Craig Chambers Ph. D. dissertation, Computer Science Department, Stanford University, March 1992.

[12] "WebCL - Heterogeneous parallel computing in HTML5 web browsers", `http://www.khronos.org/webcl/`

[13] "Web Workers" W3C Working Draft 01 September 2011, Ian Hickson, `http://dev.w3.org/html5/workers/`