JavaScript Concurrency Models

Dmitry Namiot, Vladimir Sukhomlin

Abstract— **As soon as most of the modern applications are deployed on the web, JavaScript has become a mainstream programming environment. JavaScript applications nowadays should support heavyweight web applications. We can mention here online games, graphics, media management, and even data mining. In the same time, processors on computing devices (including mobile terminals) are getting more and more elaborated. For example, it is common to see heterogeneous processors, comprised of both CPUs and GPUs almost everywhere. It creates a natural demand to parallelism in the programming tools. In this paper, we provide a survey of parallel programming models in JavaScript.**

Keywords—**JavaScript, concurrency, parallel execution, threads.**

I. Introduction

JavaScript has been around more than 20 years and nowadays is one of the most popular web development languages. It has the ability to deliver rich, dynamic web content as well as being relatively lightweight and easy to use [1]. JavaScript applications nowadays should support heavyweight web applications. We can mention here online games, graphics, media management, and even data mining. In the same time, processors on computing devices (including mobile terminals) are getting more and more elaborated. For example, it is common to see heterogeneous processors, comprised of both CPUs and GPUs almost everywhere. It creates a natural demand to parallelism in the programming tools. And JavaScript is not an excuse.

In this paper, we would like to survey concurrency patterns in JavaScript. We would like to cover the basic elements like Web Workers, as well as various frameworks for parallel execution in JavaScript. By our opinion, frameworks are extremely important for JavaScript. At the first hand, JavaScript itself has got a long story of frameworks development and deployment. On the other side, frameworks can seriously decrease development time. And time to market is one of the most important characteristics for any development tool [2]. JavaScript frameworks could be especially useful for some vertical markets. As the first example, we should mention here mobile development. And adding parallelism to JavaScript via frameworks could be the simplest way for adding concurrency to JavaScript.

The rest of the paper is organized as follows. In Section II we describe Web Workers. The Section III is devoted to WebCL. And is section IV we discuss the various frameworks.

Manuscript received May 20, 2015.

D.Namiot is senior scientist at Lomonosov Moscow State University (e-mail: dnamiot@gmail.com).

V.Sukhomlin is professor at Lomonosov Moscow State University (e-mail: sukhomlin@mail.ru).

II. Web Workers

Originally, AJAX approach [3] introduced asynchronous computation for JavaScript. AJAX introduced asynchronous HTTP request. Of course, any HTTP requests can invoke some JavaScript code. The requested page can simply contain JavaScript code. Web Workers [4] are the next natural step in this direction. A web worker is a JavaScript running in the background, without affecting the performance of the page. At this moment, Web workers are the only widely adopted support for parallel computation in JavaScript. In general, Web workers bring actor style [5] threads to the web.

As per Mozilla manual [6], Web Workers provide a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. In addition, they can perform input/output using *XMLHttpRequest* (AJAX with some limitations). Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa.). This message exchange is borrowed from actors.

A worker is an object created using a constructor that runs a named JavaScript file.

```
var myWorker = new Worker("worker.js");
```

So, the whole JavaScript file is a code offloading unit. This file contains the code that will run in the worker thread. Web workers run in another global context that is different from the current window. Thus, using the *window* shortcut to get the current global scope (instead of *self*) within a Worker will return an error.

There are two types of workers: shared and dedicated. A dedicated worker is only accessible from the script that first spawned it, whereas a shared worker can be accessed from multiple scripts. We can run any code inside the worker thread, with some exceptions. For example, it is not possible to directly manipulate the DOM from inside a worker, but, for example, we can use *WebSockets* and data storage mechanisms.

Data is sent between workers and the main thread via a system of messages — both sides send their messages using the *postMessage()* method, and respond to messages via the *onmessage* event handler (the message is contained within the Message event's data attribute.) The data is copied rather than shared. It is very important remarks. Data is copied and their size in the modern applications could be huge. So, starting a worker is costly just due to data copying process.

Posting message:

```
myWorker.postMessage(data);
```

Receiving message:

```
onmessage = function(e) {
   console.log('Message  received  from
main script');
   var  workerResult  =  'Result:  '  +
(e.data[0] * e.data[1]);
   console.log('Posting message back to
main script');
   postMessage(workerResult);
```

While Web workers 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 [7].

III. WebCL

In this section, we would like to discuss WebCL [8]. WebCL is JavaScript binding to OpenCL [9], which allows web applications to leverage heterogeneous parallel computing resources such as multi-core CPU and GPU [10]. It enables significant acceleration of compute and visual-intensive web applications such as image/video processing, advanced physics, gaming, augmented reality, etc.

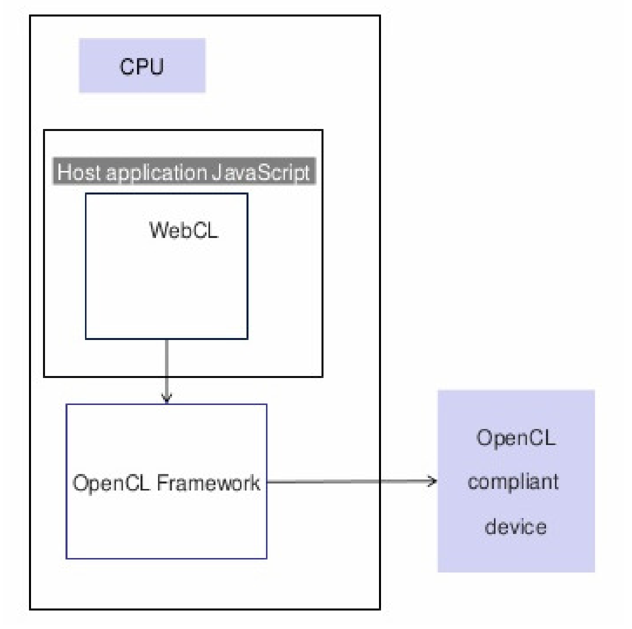

Fig. 1 WebCL model

OpenCL (and so, WebCL) follows to this model:

• A Host contains one or more Compute Devices. A Host has its own memory.

• Each Compute Device (e.g. CPU, GPU, DSP) is composed of one or more compute units (e.g. cores). Each Compute Device has its own memory.

• Each Compute Unit is divided in one or more Processing Elements (e.g. hardware threads). Each processing element has its own memory.

The Host here is a device onto which the WebCL program (JavaScript code with WebCL API calls) is executed. It is illustrated in Figure 1.

Depending on the implementation, WebCL operations may be running concurrently with JavaScript. There is more complex memory management model than in Web workers. In particular, it may be possible for the application to modify any data array while it's being asynchronously copied to/from a WebCL memory. But to avoid corrupting the contents of either buffer, applications should not modify data that has been assigned for asynchronous read/write until the relevant WebCL command queue has finished.

IV. Concurrent frameworks

In this section, we would like to discuss several JavaScript concurrent frameworks.

The goal of Intel Labs' River Trail project, also known as Parallel JavaScript, is to enable data parallelism in web applications [11]. River Trail gently extends JavaScript with simple deterministic data-parallel constructs that are translated at runtime into a low-level hardware abstraction layer. By leveraging multiple CPU cores and vector instructions, River Trail programs can achieve significant speedup over sequential JavaScript. Actually, there is The Parallel JavaScript draft API specification (ECMA) differs from the River Trail implementation in a number of ways and is the API currently being considered for standardization by the ECMA TC39 committee [12].

The central component of River Trail is the *ParallelArray* type. *ParallelArray* objects are essentially ordered collections of scalar values. *ParallelArray* objects can represent multi-dimensional collections of scalars. All *ParallelArray* objects have a shape that succinctly describes the dimensionality and size of the object. *ParallelArrays* are immutable once they are created. *ParallelArrays* are manipulated by invoking methods on them, which produce and return new *ParallelArray* objects.

ParallelArray objects could be created with the constructors and come with several methods to manipulate them. These methods typically produce a new *ParallelArray* object (except the reduce method, which produces a scalar value) [13]. River Trail uses several well-known in the parallel programming world methods. For example, the *map* method expects a function as its first argument that, given a single value, produces a new value as its result. River Trail calls such functions elemental functions, since they are used to produce the elements of a *ParallelArray* object. The *map* method computes a new *ParallelArray* object from an existing *ParallelArray* object by applying the provided elemental function to each element of the source array and storing the result in the corresponding position in the result array. For example:

```
 var source = new
ParallelArray([1,2,3,4,5]);
 var plusOne = source.map(function
inc(v) { return v+1; });
```

Here, we define a new *ParallelArray* object source that contains the numbers 1 to 5. We then call the *map* method of our source array with the function *inc*() that returns its argument, incremented by one. Thus, *plusOne* contains the values 2 to 6. Also, note that *plusOne* has the same shape as the original array source. The *map* method is shape-preserving.

The *reduce* method implements another important parallel pattern: reduction operations. This reduction operation reduces the elements in an array to a single result. A classical example to start with is computing the sum of all elements of an array:

```
  var source = new
ParallelArray([1,2,3,4,5]);
  var sum = source.reduce(function
plus(a,b) {
    return a+b; });
```

As the example shows, the *reduce* method expects as its first argument an elemental function that, given two values as arguments, produces a new value as its result. In our example, we use plus, which adds two values, as the elemental function. A reduction over plus then defines the sum operation. Note here that the reduction may be computed in any order. In particular, this means that the elemental function has to be commutative and associative to ensure deterministic results.

Other basic functions are *combine*, *filter*, *scatter* and *scan*. Of course, the map-reduce paradigm is familiar to developers, involved into big data processing (e.g., *Hadoop*).

In general this specification looks very solid and could be a good foundation to the various extensions.

The above-mentioned Web workers provide thread-like programming construct, but with shared-nothing semantics. In addition, the data communication cost between parallel contexts (workers) is extremely high in JavaScript. Low communication bandwidth with no shared memory support significantly increases the overhead of distributing input data to and merging the kernel output from multiple parallel contexts. High communication latency degrades the effectiveness of the work dispatching loop. To address these challenges, JAWS [14] introduces JavaScript framework for efficient CPU-GPU work sharing for data-parallel workloads. JAWS framework provides an efficient online work partitioning algorithm without requiring offline training runs. To efficiently merge the output chunks from both devices, JAWS supports JavaScript-level shared arrays between Web Workers, hence eliminating extra copy overhead.

JAWS implements a task scheduler, which partitions the kernel input into chunks and distributes them to a multicore CPU (running a JavaScript kernel on multiple Web Workers) and a GPU (running a WebCL kernel) for concurrent execution. A chunk is formed by taking a contiguous subset of the flattened input data [15], specified by a pair of array indices pointing to the first and last elements of the subset, as we focus on array-based data parallel workloads. To effectively communicate input and output data between workers by references, instead of values, JAWS allocates shared arrays accessible by all workers (including one worker managing GPU execution). The scheduler needs to send only pairs of array indices instead of sending the entire data. The scheduler dynamically adapts the chunk size for each device to minimize the overhead of the work dispatching loop. Once execution of a chunk is finished on a computing device, the device writes the produced output chunk into the shared output buffer and signals the task scheduler to fetch a new task. This process continues until the task queue becomes empty

WorkerJS [16] library provides some kind of “middleware” for Web workers. WorkerJS makes it very easy to push functions from the main Javascript thread into the scope of a Worker by converting the web worker message passing interface into an RPC style interface. WorkerJS makes it very easy to pull Worker functionality back out into the main Javascript thread, exposing Worker functionality as RPC functions on an object. WorkerJS allows easy clustering of many Workers - all the Workers in a cluster appear as a single object, invoking a function on the cluster will send the request to every Worker in the cluster and the callback will fire when every Worker in the cluster has finished.

Parallel Closures [17] uses immutable data structures (like RiverTrail above). It proposes two small changes to the common model for lightweight task frameworks. The proposed changes make it possible to statically guarantee a deterministic result with only minimal added complexity. The only requirement is the ability to declare transitive read-only pointers, which is itself a commonly requested and useful feature even in sequential code. The first proposed change is to prevent parent and child tasks from executing concurrently. Typically, in the parallel systems, forked tasks can potentially run in parallel with their creator. In Parallel Closures, however, multiple child tasks are accumulated and then forked-and-joined in one atomic action. This prevents the parent task from racing with its children. The second proposed change is to specify the body of a child task using a parallel closure. As with traditional closures, a parallel closure is a block of code which can access variables from its surrounding environment. In a novel twist on traditional closures, however, parallel closures are only granted read-only access to the data which they inherit from their environment. This change means that two parallel closures can safely execute in parallel with one another, as any data that is shared between them is read-only. JavaScript implementation is available on Github [18].

V. Conclusion

In this short paper, we provide a survey for some JavaScript concurrency models. As the Web programming with JavaScript continues evolving, the lack of parallelism in JavaScript would eventually correspond to a serious technological barrier. By our opinion, JavaScript frameworks could be an easiest (fastest) way for adding parallel programming models to JavaScript.

ACKNOWLEDGMENT

We would like to thank Samsung Research Center in Moscow for the inspiration of this research.

REFERENCES

[1] Is JavaScript The Primary Programming Language For The Enterprise? http://www.codeinstitute.net/javascript-primary-programming-language-enterprise/ Retrieved: May, 2015.

[2] Namiot, Dmitry, and Manfred Sneps-Sneppe. "On software standards for smart cities: API or DPI." ITU Kaleidoscope Academic Conference: Living in a converged world-Impossible without standards?, Proceedings of the 2014. IEEE, 2014.

[3] Garrett, Jesse James. "Ajax: A new approach to web applications." (2005): 1-6.

[4] Herhut, S., Hudson, R. L., Shpeisman, T., & Sreeram, J. (2012, June). Parallel programming for the web. In Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar (Vol. 12, p. 1).

[5] Agha, G., & Callsen, C. J. (1993). ActorSpace: an open distributed programming paradigm (Vol. 28, No. 7, pp. 23-32). ACM.

[6] Using Web Workers https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers Retriewed: May, 2015

[7] Herhut, Stephan, et al. "Parallel programming for the web." Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar. Vol. 12. 2012.

[8] Aarnio, T., and M. Bourges-Sevenier. "WebCL working draft." Khronos WebCL Working Group (2012).

[9] Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." Computing in science & engineering 12.1-3 (2010): 66-73.

[10] Jeon, W., Brutch, T., & Gibbs, S. (2012). WebCL for hardware-accelerated web applications. In TIZEN Developer Conference May (pp. 7-9).

[11] RiverTrail https://github.com/IntelLabs/RiverTrail Retrieved: May, 2015

[12] ECMA TC39 http://www.ecma-international.org/memento/TC39.htm

[13] RiverTrail tutorial http://intellabs.github.io/RiverTrail/tutorial/

[14] Piao, X., Kim, C., Oh, Y., Li, H., Kim, J., Kim, H., & Lee, J. W. (2015, January). JAWS: a JavaScript framework for adaptive CPU-GPU work sharing. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 251-252). ACM.

[15] P. Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In CGO, 2014

[16] WorkerJS http://hungrygeek.holidayextras.co.uk/WorkerJS/ Retrieved: May, 2015

[17] Matsakis, Nicholas D. "Parallel closures: a new twist on an old idea." Proceedings of the 4th USENIX conference on Hot Topics in Parallelism. USENIX Association, 2012.

[18] Parallel Closures in JavaScript https://github.com/mozilla/pjs Retrieved: May, 2015