# High Performance Computing in Web Browsers

Henning Lohse

***Abstract*—Traditionally, High Performance Computing (HPC) applications require parallelized processing to handle input data and runtimes. Heterogenous systems for specialized computations are advantegeous. Today's consumer electronics are heterogenous multiprocessor systems equipped with web browsers. The latter provide portability for applications. This work analyzes the current state of web browser technologies for portable HPC purposes.**

**asm.js allows optimized JavaScript execution. HTML5 Web Workers are threads communicating via message passing allowing intra-node processing. WebRTC DataChannel is a configurable peer-to-peer socket for inter-node processing. WebCL and OpenGL Compute Shaders are GPU computing candidates.**

***Index Terms*—HPC, JavaScript, asm.js, HTML5, Web Workers, WebRTC, DataChannel, WebCL, WebGL, Compute Shaders**

## I. Introduction

TODAY'S computational applications originate from broad interdisciplinary fields. These include meteorology, astro physics and molecular biology in scientific computing, as well as fluid dynamics, crash tests and distributed databases in commercial areas, or online games for consumers.

They share high demands to the computational platform. Results are prefered to be achieved as fast, precise and energy efficient as possible or required. Large data sets have to be handled accordingly. Techniques to meet these demands have been explored and aggregated in the field of High Performance Computing. (HPC)

Runtime and storage requirements of HPC applications traditionally exceed the capabilities of a single processor or machine. Therefore, parallel computing became the major factor for performance optimizations. Today, multiple heterogeneous processors can be used in interconnected machines. When designing an HPC application, the following aspects as illustrated in Figure 1 have to be considered:

- *Processor*. Determine which application parts should be executed on a distinct processor. This can be a core of a multicore CPU.
- *Intra-node processing*. A machine node might contain multiple sockets for CPUs, where each CPU provides multiple cores. The application design should benefit from synergetic effects of caching, memory access patterns and locality while minimizing communication overhead. For programming, pthreads or OpenMP can be used.
- *Inter-node processing*. Multiple nodes are usually connected via network. Inter-node communication is orders of magnitude slower then intra-node communication. While synergetic effects to be considered here are similar to intra-node processing, communication overhead induces a more significant penalty. sockets or MPI can be used for implementations.
- *Coprocessors*. Complementary processors like GPUs can be used to improve performance for certain types of computations. GPUs for example are highly optimized for SIMD workloads with preferably short runtimes per data point on a huge set of them. CUDA or other coprocessor-dependent libraries can be utilized.

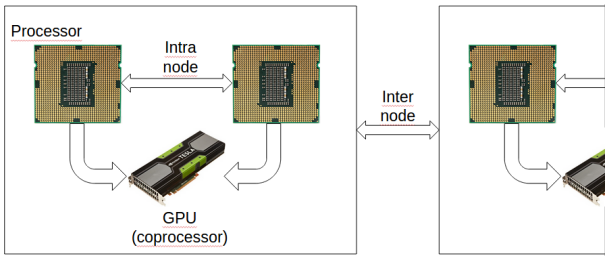


Fig. 1: HPC design aspects illustration

These aspects are tightly coupled. Designs considered for each one are dependent on the others. Furthermore, when implementing the HPC application, optimizations are usually done for a specific platform or range of platforms. Applications come as compiled binaries. While these approaches provide the best performance, portability is not available or time-consuming to achieve.

### A. Motivation

Consumer electronics today are heterogeneous multiprocessor systems. Notebooks, Tablets and Smartphones are equipped with multicore CPUs, GPUs and (wireless) network technologies. While they run different and diverse operating systems hindering portability of native applications, each platform comes with a web browser.

Modern web browsers can be considered application platforms. They provide GUI rendering, network communication, user interaction and dynamic content control. The development of HTML5 introduced several improvements and new features, like threads or client-server sockets. Plenty of frameworks allowing developers to connect the browser frontend with a server backend exist. The JavaScript engines are constantly improved for faster script execution. And all these features work on virtually any platform with a modern web browser, resulting in inherent application portability.

The topic of this work is to present the current state of web browser technologies regarding HPC application design aspects processor, intra-node and inter-node processing. As coprocessors can be diverse and specialized, this work focuses on GPUs.

### B. Section Overview

Section II gives a brief overview of the JavaScript language used in HTML scripts. Important concepts and constructs for further sections are discussed. The following section III introduces asm.js as an annotation-based JavaScript subset for performance optimizations. Intra-node processing capabilities come with HTML5 Web Workers in section IV. Section V presents WebRTC DataChannel and its inter-node processing capabilities. The state of GPUs as coprocessors using WebCL and OpenGL Compute Shaders is shown in section VI. Finally, conclusions are drawn in section VII.

## II. JavaScript

JavaScript is the programming language used for scripting in HTML documents. Code written in JavaScript resides in the script block of the HTML Domain Object Model (DOM), as seen in figure 2. This language's purpose is amongst others to allow dynamic reload of content on events, like clicks, to reduce bandwidth and allow user interactions. Input can also be verified and the document's design can be altered. These operations can be done by manipulating the DOM tree, yet arbitrary computations can also be done. JavaScript is a just-in-time compiled language.

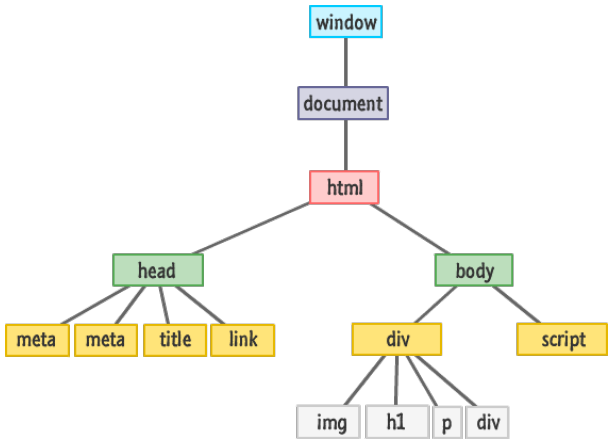


Fig. 2: HTML Domain Object Model [kirupa.com]

### A. Type System

When looking at arbitrary computations, it is important to note the key characteristics of JavaScript's type system:

- *Dynamically typed*. Types of objects are resolved at runtime; static checks are not done. This means operations like late binding or downcasting are performed during execution. This allows developers to write code faster, but operations might fail at runtime due to e.g. missing operators.
- *Object-oriented*. As noted before, JavaScript utilizes objects. It is important to note that even CPU native types, like a 32 bit integer, are wrapped into JavaScript primitive type objects. While this simplifies development by adding utility functions, addtitional overhead occurs during runtime.
- *Classless*. Objects are not instantiations of pre-defined classes. They are defined as key-value pairs inside curly brackets, as seen in listing 1. This notation is refered to as the JavaScript Object Notation. (JSON) It allows arbitrary hierarchical nesting of data structures, like additional objects or arrays.
- *Prototypes*. As JavaScript is classless, prototypes are used. If multiple objects with the same members are desired, a prototype object is created and copied. These copies can again be arbitrarily modified.

Listing 1: JavaScript type system and JSON example

```
var prototype = {
  "Publisher": "Xema",
  "ID": "1234-5678-9012-3456",
  "Owner": {
    "Name": "Max",
    "male": true,
    "Hobbys": ["Riding", "Golf"],
    "Age": 42,
  }
};

var copy = prototype;
copy["Currency"] = "EURO";
copy["Owner"]["Hobbys"].push("Reading");
```

### B. Memory Management

JavaScript does not support explicit memory management. Functions like malloc or free, new or delete are not available. The web browser must provide a Garbage Collection mechanism. Its purpose is to automatically remove unused objects from memory. This makes development easier, as object ownership has not to be considered, and prevents memory leaks. Implementations in current web browsers are based on the mark-and-sweep algorithm. Basically, unreferenced, in the sense of unreachable, objects are to be removed.

Applications though may suffer from badly timed garbage collecting. JavaScript and current web browsers provide no interface to control the time a garbage collection occurs. Today's implementations are highly optimized. Still, real-time and memory intensive applications, like many HPC applications, have to be implemented carefully. Creation of too many objects should be avoided and patterns benefiting garbage collection should be used. [1]

## III. asm.js

As thoroughly explained in [2], memory access patterns and caching effects are crucial for application performance. Yet internally, JavaScript does not necessarily use CPU native data types, but wrapper objects. While at some point execution comes down to instructions on native data types, wrapping functions induce overhead. Furthermore, when using JSON, memory layout of data is implicit. Without special functions, an e.g. fixed size array of 32 bit integers cannot be instantiated, hindering cache optimizations.

Mozilla identified these problems and defined asm.js to improve JavaScript performance. asm.js utilizes annotations on data that do not alter JavaScript semantics, but can be detected by the browser to use CPU native data types. The

JavaScript implementation in listing 2 assigns the number 2 to the variable p, whereas the asm.js implementation in listing 3 assigns (2|0) to p. While semantically identical, an asm.js-compatible web browser will interpret (2|0) as a 32 bit integer of value 2. This allows the browser to deploy ahead-of-time optimization strategies for the following just-in-time compilation by not using wrapper objects. Several annotations for different data types exist. [3]

For additional optimizations, data structures like arrays can be created from functions like Int32Array. This allows the web browser to allocate an efficiently accessible array.

Listing 2: JavaScript find prime numbers implementation

```
var primes = [];

for (var p = 2; p <= max; p++) {
  var is_prime = true;

  for (var i = 2; i <= max_sqrt; i++)
    if (p % i == 0 && p != i) {
      is_prime = false;
      break;
    }

  primes[p] = is_prime;
}
```

Listing 3: asm.js find prime numbers implementation

```
var primes = new Int32Array(max);

for (var p = (2|0); p <= max; p++) {
  var is_prime = (1|0);

  for (var i = (2|0); i <= (max_sqrt|0); i++)
    if (p % i == (0|0) && p != i) {
      is_prime = (0|0);
      break;
    }

  primes[p] = is_prime;
}
```

### A. Performance Comparison

The vision of asm.js is to provide near-native application execution speed. The implementations in figures 2 and 3 have been compared to a C implementation compiled with GCC and -O3. The average runtimes on the machine used for this work in ms for finding prime numbers in the range of 2 to 10,000,000 are:

- *C:* 12,198 (1x)
- *JavaScript:* 15,080 (1.236x)
- *asm.js:* 12,244 (1.004x)

Performance is highly dependent on the application's underlying algorithms. For this example, annotating integer values and using an Int32Array reduced to slowdown from 1.236x to 1.004x. In general, Mozilla states that Firefox is able to execute arbitrary asm.js code with a slowdown of 1.5x to 2x compared to a native binary compiled with clang. [3] [4]

### B. Emscripten

Utilizing the asm.js annotations correctly must be done carefully. Additionally, controlling garbage collection and using (cache) efficient memory accesses is difficult in a language like JavaScript not intended for such actions. Mozilla started the Emscripten project to compile C/C++ code to asm.js. This allows development of highly optimized JavaScript-compatible asm.js applications for experienced C/C++ developers.

For this to work, Emscripten uses the clang compiler toolchain to generate an LLVM intermediate representation (IR) of C/C++ code. LLVM, Low Level Virtual Machine, is a virtual machine with an instruction set optimized for cross compilation purposes. Given this IR, Emscripten generates asm.js code from it.

This code profits amongst others from ahead-of-time optimizable instructions on native CPU data types and cache efficient data memory layouts. Even thread usage is supported. Furthermore, no garbage collection occurs. This is done by allocating an array used as a virtual heap where all objects are created and removed from. Memory management is done manually on this array by Emscripten. [3]

In March 2014, Mozilla showed a demo of the Unreal Engine 4 video game engine compiled to asm.js using Emscripten. It was running in Firefox at 67% native speed. [5]

### C. Compatibility

Current versions of Firefox, Chrome and Internet Explorer support optimized execution of asm.js-compatible JavaScript. There are significant performance differences, with Firefox and Chrome around twice as fast as Internet Explorer. Safari does currently not detect asm.js code. [4]

## IV. HTML5 Web Workers

For efficient machine utilization and speedups, HPC applications must make use of all available CPU cores by designing the application with respect to intra-node processing capabilities. Traditionally, this is realized by using multiple threads in a process. While each thread may allocate its own memory, currently common shared memory systems allow direct access of shared memory between all threads.

Until the release of HTML5, an HTML document's JavaScript script was always executed in a single thread. To allow threaded execution, HTML5 specified Web Workers. The main thread can spawn Web Workers and assign a script file to them, as seen in listing 4. This script file can contain arbitrary code to be executed in its own thread, asynchronously from the main thread. See listing 5. One constraint for Web Workers is that they cannot access and manipulate the HTML document's DOM; only the main thread is able to do that.

Listing 4: Main thread spawns Web Worker

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

worker.addEventListener("message", function(e) {
  console.log(e.data);
}, false);

worker.postMessage("Hello!");
```

Listing 5: Web Worker script

```
self.addEventListener("message", function(e) {
```

```
  // Async computations go here
  self.postMessage(e.data);
}, false);
```

### A. Message Passing

Another constraint for Web Workers is that shared memory is not possible. Memory allocated in the main thread or in a Web Worker is never accessible by others. Communication is based on message passing. This is done by using the postMessage function. One has to differentiate between two use cases of this function.

By simply handing an object to postMessage, a structured cloning on this object is performed to create a copy. This copy is made available to the Web Worker; direct access to the original is not possible. Structured cloning is the way to systematically copy JSON objects, which can be hierarchically nested. Copying such data structeres is detrimental for achieved bandwidth. On the machine used for this work, postMessage structured cloning of a simple array caps at around 1 GB/s, as seen in figure 3. As arrays are continously stored in memory, this actually represents the upper limit. Compared to traditional shared memory accesses, this is hardly acceptable for HPC applications.

Additional issues may arise from sending messages of several MBs in size too fast. The garbage collector might potentially not be fast enough in freeing unused copied messages. Results might be huge or spiked memory consumption, or even an application crash if memory is exhausted.

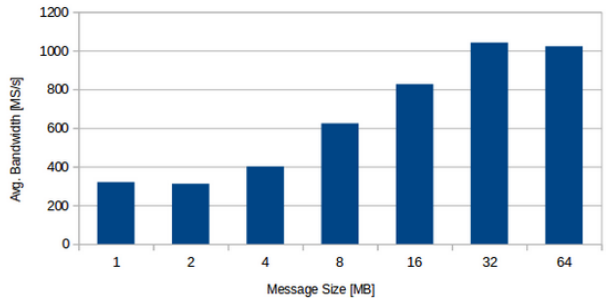


Fig. 3: postMessage structured cloning bandwidth of an array

Alternatively, transferable objects can be send via postMessage. Transferable objects are simple data structures like arrays, which need no structured cloning. The syntax used is shown in listing 6. It is important to note that sent data switch contexts. After the call to postMessage, the sender has no longer access to the data; the variable evaluates to null. Only the receiver has access now. This is realized internally by simple pointer exchanges, making bandwidth measurements obsolete. The latency for this operation on the machine used for this work is 53us. HPC applications should structure their Web Workers communication using this approach.

Listing 6: Sending a transferable object array

```
var array = new ArrayBuffer(1024); // 1kB
worker.postMessage(array.buffer, [array.buffer]);
```

### B. Compatibility

Web Workers are basically supported in all modern web browser. Unfortunately, sending transferable objects via postMessage is not part of the HTML5 specification. Currently, Internet Explorer does not support transferable objects via postMessage.

## V. WebRTC DataChannel

Text. Figure 4 Figure 5

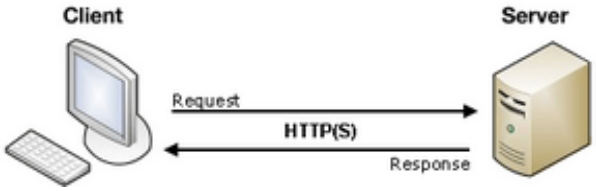


Fig. 4: HTTP communication [developer.mozilla.org]

Text. Figure 6

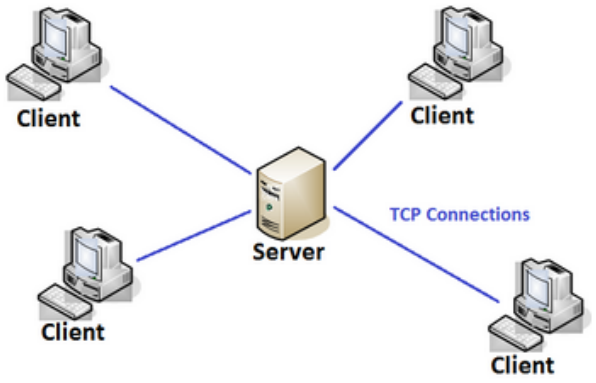


Fig. 5: Client-server architecture [cs.montana.edu]

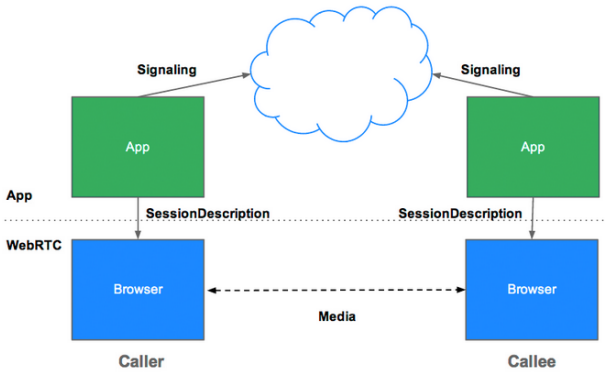


Fig. 6: WebRTC overview [html5rocks.org]

Text.

### A. Compatibility

Text.

## VI. GPUs as Coprocessors

### A. WebCL

### B. OpenGL Compute Shaders

## VII. Conclusion

Text.

## References

[1] Mozilla Foundation, *Memory Management*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management (05.01.2015)
[2] U. Drepper, *What Every Programmer Should Know About Memory*. Red Hat, Inc., 2007.
[3] Mozilla Foundation, *asm.js Working Draft*, http://asmjs.org/spec/latest/ (05.01.2015)
[4] Mozilla Foundation, *asm.js performance improvements in the latest version of Firefox make games fly!*, https://hacks.mozilla.org/2014/05/asm-js-performance-improvements-in-the-latest-version-of-firefox-make-games-fly/ (05.01.2015)
[5] Mozilla Foundation, *Mozilla and Epic Preview Unreal Engine 4 Running in Firefox*, https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/ (05.01.2015)
[6] &yet, *Is WebRTC ready yet?*, http://iswebrtcreadyyet.com/ (05.01.2015)
[7] Mozilla Foundation, *[WebCL] add openCL in gecko*, https://bugzilla.mozilla.org/show_bug.cgi?id=664147#c30 (05.01.2015)