Parallelize JavaScript-Computations with Ease

Micha Reiser

University of Applied Sciences Rapperswil
Supervised by Prof. Dr. Luc Bläser
AT 2016

Abstract

TODO

1 Introduction

The role of JavaScript drastically changed in recent years. From an unpopular language used to add dynamic effects to web pages to one widely used with a strong and growing community. It emerged from a browser-only language to a general-purpose language used to write web-, desktop-, mobile-, and server-applications. The new use cases come along with new requirements demanding for more computational resources that can no longer be provided by a single threaded environment without negative affecting the user experience. This becomes, even more, severe with the spreading use of mobile devices in the world-wide-web. The CPUs of nowadays mobile devices have — compared to desktop computers — a lower clock rate that is recompensed by the higher number of CPU-cores. However, this requires that applications make use of the available computation resources.

The W3C responded to these new requirements with the web worker draft in 2009 providing the infrastructure for creating multi-threaded applications [1] in the browser. The draft defines the web worker API that allows running scripts in background threads. Each thread has its own memory area since the JavaScript memory model is not defined for concurrent access. This requires that cloned values be passed between threads using messaging. However, the messaging based programming model does not fit well into existing applications and the gap between the models needs to be bridged by the programmer — adding non-inherent complexity to the program. The web worker standard further requires that code running in background threads be located in designated files. This separates coherent logic — making the code less understandable. It also increases the complexity of the build process since two artifacts have to be created, one for the logic residing in the main thread and another for the code running in web workers.

Unfortunately, implementing multi-threaded applications targeting different environments in JavaScript is non-trivial either since no uniform standard for creating background threads exists. The web worker standard, defined by the W3C consortium, is only implemented in browsers. NodeJS allows spawning subprocess using the child-process API [2]. JavaScript applications running on the JVM can use RingoJS [3] that enables multi-threaded JavaScript — including shared memory and all problems shared state brings with it. Therefore, an application needs to be explicitly adopted to the standards provided by its supported environments.

I believe that the platform dependent standards together with the inherent complexity caused by the programming model gap are the main reasons for the low spread of multi-threaded applications in JavaScript. This paper presents *Parallel.es*, a platform independent, type safe API and runtime system for creating multi-threaded applications in JavaScript. The API abstracts the messaging based programming model used by some of the underlining standards, allowing a seamless integration into existing code. The runtime system consists of two parts: A low-level API allowing to run single functions on a background thread and a reactive API inspired by the commonly used underscore [4] and lodash [5] libraries. The reactive API is mainly designed with simplicity in mind covering the aspects of work partitioning and result joining while providing a well-known and familiar API that allows an easy transformation of existing code. The low-level API is designated for the cases where more flexibility is needed or when the reactive programming model does not fit well with the problem to solve.

This work further presents a transpiler that hides some of the limitations caused by the programming model gaps applying for task functions since it is technically impossible to cover up all limitations at runtime. The transpiler rewrites the code to reduce the programming model gap to a minimum. It adds debugging support, allows a function to import functionalities from libraries, calling other functions or accessing read-only variables from the outer closure. However, the use of the transpiler remains optional.

The first section of this paper describes the related work and compares it to the presented solution. The following describes the programming model of parallel tasks and is fol-

lowed by the section 4 that explains the functionality of the runtime system. The section 5 explains which programming model differences can be hidden by using the transpiler and how this is accomplished. The section 6 compares the presented runtime system with the existing work and is followed by the conclusion.

2 Related Work

There exist various open source projects addressing similar or equal goals. This section describes the differences in the presented work to the already existing ones. One main difference of the presented system to the related work is that it offers the same debugging functionalities as developers are used to when working with synchronous code.

2.1 Hamsters.js

Hamsters.js [6] is the library with the highest attention measured by the number of GitHub stars. It provides a low-level API for running functions in a background thread. The background threads are managed in a global thread pool. Also, various helper functionalities like array sorting, aggregating or caching are provided. The library has support for transferable objects and claims to support scheduling threads from a background thread.

The main difference to Hamsters.js is that the proposed work offers a higher-level API that simplifies parallelization.

2.2 Parallel.js

Parallel.js [7] has been initiated in 2013 and is the oldest of the evaluated libraries. Its main goal is to provide a simple API for multiprocessing in JavaScript. It provides a uniform API for the browser and NodeJS — web workers are used in the browser, child processes [2] on NodeJS. Parallel.js provides a low-level API for running a function in a background thread and a reactive API providing an automatic task scheduling for the provided data.

This work differs from Parallel.js in the fact that a thread pool is used that reuses already created background threads and performs queuing if needed. Parallel.js has kind of a thread pool approach but only on a per-operation (e.g. map or filter) basis. The second important difference is that Parallel.js awaits the sub-results of the proceeding operation before continuing with the next operation if multiple operations are chained together, e.g. the reduce step summing up the values of a filtered array waits until all background threads have completed filtering before starting with summing up the values. Furthermore, the sub-results are always transmitted back to the main thread before starting the next operation on new background threads. This results in unneeded — and potentially very expensive — copying of intermediate results from and to background threads.

The latest published version on npm[1] spawns a separate worker for every element in the input array exhausting the thread limit of the browser. The latest version on GitHub has adjusted this behavior to only spawn as many workers as logical processors are offered by the hardware. Therefore, when Parallel.js is referenced, the latest version[2] from GitHub is meant.

2.3 Threads.js

Thread.js [8] aims to be a powerful, but simple to use, multi-threading library for NodeJS and the browser. Like Parallel.js, child processes are used in NodeJS and web workers in the browser.

The main difference of Threads.js is that it is messaging based. It, therefore, fits better to the programming model used by web workers and child processes. However, this requires that the programmer bridge the gap between the programming model used inside the application and the messaging based programming model of Threads.js.

3 Programming Model

The programming model of Parallel.es motivates the programmer to perform time-intensive computations asynchronously and potentially concurrently in background tasks.

3.1 Background Task

A *background task* — further referred to as task — represents a single asynchronous operation executed on a background thread and is implemented by a JavaScript function. Listing 3.1 shows an example that computes the Fibonacci number for the value 100 in a task and logs the result to the console. The task is started using the `run` method by passing the function to execute together with the arguments for the function call (line 9). The returned object implements the promise interface [9, Section 18.3.18] allowing to register a `then` callback that is invoked with the result if the computation was successful and an error handler (**catch**) that is triggered otherwise. The callbacks are executed on the main thread and allow retrieval of the result or error.

```
1   function fib(num) {
2     if (num <= 2) {
3         return 1;
4     }
5
6     return fib(num - 1) + fib(num - 2);
7   }
8
9   parallel.run(fib, 100)
10    .catch(error => console.error(error))
11    .then(result => console.log(result));
```

Listing 3.1. Fibonacci Implementation

The valid task functions are only a subset of all JavaScript functions since not all JavaScript features can be used. This

---

is rooted in the fact that neither web workers nor child processes have a shared memory[3] that can be used to store data accessed by different threads. It is therefore disallowed that a task function references symbols — variables, functions, or imports from external files — from the outer scope[4]. Data needed by a task function needs to be explicitly passed when the task is started and is serialized using the structured clone algorithm [11, Section 2.9.4]. This enforces a pass-by-value semantic and requires that the passed data be serializable according to the structured clone standard, e.g. Functions, DOM-Elements and errors are not cloneable. The same applies to the result returned by the task.

The global context of a task differs from the global context the function is defined in. Changes made to the global context in the main thread or inside of a single task are not reflected across the threads. Furthermore, the API exposed by the environment may differ, e.g. the DOM API is not exposed inside of web workers. This does not add strong limitations to task functions since they are mainly used to perform compute-intensive, but side effect free, operations only depending on local data.

Parallel tasks are isolated from one another since each task operates on its own copy of the variables and every thread executes one task at a time. However, the global state of a thread is shared among all tasks executed using this specific instance. It is, therefore, possible that a task affects another task if they access and modify the same global state. This can not be prevented by the runtime system but is strongly discouraged as changes are only thread-local — and therefore, not replicated between threads — and may introduce memory leaks.

3.2 Reactive API

The runtime system further offers a reactive API. It is inspired by the commonly used underscore and lodash libraries and motivates the programmer to define the computations as operations on data streams. The runtime system takes care of splitting the work into several sub-tasks and aggregating the sub-results into the end-result. The created tasks perform all operations on a subset of the input values and are potentially executed concurrently if enough resources are available. The goal of this API is to provide a well-known and understood API that uses the available computation resources without any further actions needed by the programmer. This simplifies parallelizing existing synchronous code. The reactive API uses the infrastructure provided by the low-level API. Therefore, the same programming model applies.

An implementation of the Mandelbrot computation using the reactive API of Parallel.es is shown in listing 3.2. The implementation differs only slightly from the synchronous implementation shown in listing 3.3. The `range` method (line 8) defines the data stream to process. It creates a data

stream containing the values from 0 up to the image height. The input stream is transformed by mapping (`map` on line 11) each element from the input stream to an output element that is computed by the `computeMandelbrotLine` function (line 1). The `computeMandelbrotLine` function — that is executed in a background thread — has access to the current array element and as well to the `environment` object that is specified using the `inEnvironment` method (line 10). The computation is started using the `then` method (line 12) that registers a callback that is invoked if the computation succeeds — an optional error callback can be defined as well. The `then` callback is executed in the main thread.

```
1  function computeMandelbrotLine(y, environment) {
2    const arrazSize = environment.imageWidth * 4;
3    const line = new Uint8ClampedArray(arraySize);
4    // compute...
5    return line;
6  }
7
8  parallel
9    .range(options.imageHeight)
10   .inEnvironment(options)
11   .map(computeMandelbrotLine)
12   .then(result => console.log(result);
```

Listing 3.2. Mandelbrot Implementation in Parallel.es

```
1  function computeMandelbrotLine(y, options) {
2    const arraySize = options.imageWidth * 4;
3    const line = new Uint8ClampedArray(arraySize);
4    // compute...
5    return line;
6  }
7
8  const result = _.chain()
9    .range(options.imageHeight)
10   .map(y => computeMandelbrotLine(y, options))
11   .value();
12
13 console.log(result);
```

Listing 3.3. Synchronous Mandelbrot Implementation

4 Runtime System

The runtime system of Parallel.es consists of two parts: The slaves running in background threads executing the tasks and the public API in the main thread that forms the facade and acts as the master for the slaves. Applications are using the facade provided by the master to run functions in a background thread. The master is responsible for spawning the slaves and distributing the work onto these. It, therefore, uses a thread pool that manages the created slave instances and queues tasks if no idle slave is available. The default thread pool uses a FIFO queue and the number of slaves is limited to the number of logical processors provided by the hardware[5].

---

[3]This might change with the SharedMemory [10] standard that is currently a draft in stage 2.

[4]The special identifiers **this** and **super** are treated equally to other identifiers referring to variables from the outer scope and their usage inside of a task function is therefore prohibited. This also implies that an arrow function used as task function is semantically equal to a function expression.

[5]The number of logical processors can be determined using `navigator.hardwareConcurrency`. The implementation assumes the hardware has four logical processors if the API is not supported by the used browser.

The next section describes how the runtime system processes a single task.

4.1 Task Roundtrip

The steps needed to process a single task are shown in fig. 1. The application passes the task function together with the arguments to the facade that acts as the master (1). The created task is queued in the thread pool and executed on the first slave that gets available. The master transmits the serialized representation of the function call — consisting of a unique id identifying the function to call and the arguments to pass to the function — to the slave (2). The slave performs a lookup in the function cache to obtain the function with the given id (3). If the function is executed the first time on this slave, then the function is not present in the function cache and its definition is therefore requested from the master (4, 5). The master transmits the function definition to the slave that deserializes the definition and registers the function in the function cache (6). The slave calls the deserialized function with the provided arguments (7) and returns the (structured cloned) result back to the master (8). The master invokes the success handler in the main thread to pass the result to the application (9).

The caching of the function definitions on the slave have the advantage that performed JIT-optimizations are not thrown away if a task has completed. It is believed that the gain of reusing the JIT-Optimized function outweighs the additional costs caused by the function lookup and additional roundtrip for function retrieval. This especially for frequent but short running tasks for which the serialization overhead weights heavier.

4.2 Limitations

The current implementation supports the essential features. However, it misses support for asynchronous task operations and the NodeJS environment. There are no technical reasons for not supporting either of these features. Adding support for NodeJS requires a structured clone polyfill to have the same behavior in NodeJS as in the browser.

A further limitation is that a task can not start other tasks. An efficient implementation to support recursive tasks requires a communication channel between all web workers that allows starting a task from a background thread on another, idle background thread without an additional roundtrip over the main thread. However, web workers only allow having a single communication channel between the thread that has started the Web Worker and the spawned web worker. Multiple channels are supported by Shared Workers that lack support in older browsers [1, section 4.6.4].

5 Transpiler

The absence of a shared and by all threads accessible memory results in various limitations that apply to task functions. These restrictions are described in the previous section and have in common, that no symbols from the outer scope are allowed to be referenced from inside of a task function.

This impedes code reuse and hinders a seamless integration into existing code. The presented transpiler hides the limitations of the runtime system from the programmer by rewriting the task start and task functions.

The transpiler is based on top of webpack[6] [12] and Babel[7] [13]. It extracts the task functions from the code, adds it to the file loaded by the slaves, adds required import statements and pre-registers the task function in the slave's function cache.

The Mandelbrot implementation referencing the `imageWidth` variable (line 12, 14) and the function `computePixel` (line 4) from the outer scope is shown in listing 5.1. The transpiler rewrites the code and makes the `imageWidth` variable and the `computePixel` function available in the task function. The transpiled code is shown in listing 5.2. The transpiler creates a new function that returns an object containing the values of all referenced variables and inserts it above the definition of the task function (line 9). This function is used to extract the current values of the referenced variables in the master thread. The object returned by the function is made available inside the task by setting it as environment using the `inEnvironment` method (line 16). The transpiler further replaces the task definition by a unique function-id (lines 17-20) that the slave uses to resolve the function from the function cache[8].

```
1   const imageWidth = 10000;
2   const imageHeight = 10000;
3
4   function computePixel(x, y) {
5     //...
6     return n;
7   }
8
9   parallel
10    .range(imageHeight)
11    .map(function (y) {
12      const line = new Uint8ClampedArray(imageWidth * 4);
13
14      for (let x = 0; x < imageWidth; ++x) {
15        line[x * 4] = computePixel(x, y);
16      }
17      return line;
18  });
```

Listing 5.1. Mandelbrot Implementation Referencing Symbols from the Outer Scope

The code inserted by the transpiler into the script run by the slave is shown in listing 5.3. The transpiler injects the code of the task function (lines 7-13) and the referenced `computePixel` function (lines 2-5)[9]. Further, an *entry*-function (lines 15-23) is generated that initializes the `imageWidth` variable (line 1) with the value stored in the

---

[6]A JavaScript module bundler

[7]Framework for transforming JavaScript code

[8]The task function is not replaced if it is defined using a function declaration since it might be used by other code. Task functions declared using function declarations that are only referenced from a parallel call are removed by webpack using tree shaking.

[9]The transpiler also creates an immediately invoked function expression that isolates the task functions of different modules and avoids naming clashes
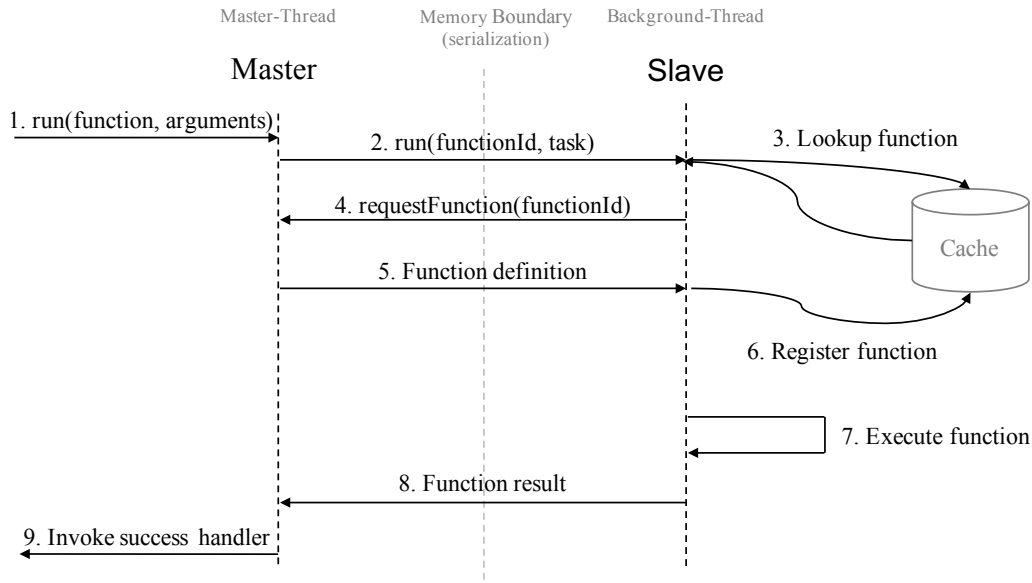
Figure 1.   Parallel.es Runtime System

```
1    const imageWidth = 10000;
2    const imageHeight = 10000;
3
4    function computePixel(x, y) {
5      //...
6      return n;
7    }
8
9    function _environmentExtractor() {
10     return {
11       imageWidth: imageWidth
12     };
13   }
14
15   parallel.range(imageHeight)
16     .inEnvironment(_environmentExtractor())
17     .map({
18       identifier: "static:file.js/_entry_anonymous",
19       _____isFunctionId: true
20     });
```

Listing 5.2.   Transpiled Mandelbrot Implementation

environment — that contains the values of the variables from the main thread — and calls the actual task function. The entry function is registered in the function cache (lines 25-28) using the same function id as used in the master thread. This allows the slave to retrieve the function immediately from the function cache without the need to request the function definition from the master — that requires (de-) serialization of the function.

The transpiler further generates source-maps that point back to the original location of the extracted task function — and transitive functions. This enables a true debugging experience that allows setting breakpoints inside of the browser developer tools[10]. Without these source-maps, breaking inside of a task function is only possible by using the inflexible **debugger** statement. The source-maps further allow the browser to translate error messages back to the original code.

———

[10]This is currently only supported by the developer tools of Google Chrome and Microsoft Edge.

```
1    var imageWidth;
2    function computePixel(x, y) {
3      //...
4      return n;
5    }
6
7    function _anonymous(y) {
8      var line = new Uint8ClampedArray(imageWidth * 4);
9      for (var x = 0; x < imageWidth; ++x) {
10       line[x * 4] = computePixel(x, y);
11     }
12     return line;
13   }
14
15   function _entry_anonymous() {
16     try {
17       var _environment = arguments[arguments.length - 1];
18       imageWidth = _environment.imageWidth;
19       return _anonymous.apply(this, arguments);
20     } finally {
21       imageWidth = undefined;
22     }
23   }
24
25   slaveFunctionLookupTable.registerStaticFunction({
26     identifier: 'static:file.js/_entry_anonymous',
27     _____isFunctionId: true
28   }, _entry_anonymous);
```

Listing 5.3.   Generated Slave-Code for Transpiled Mandelbrot Implementation

The source-map support is a distinct feature not provided by any of the related work.

5.1 Limitations

The current implementation of the transpiler only supports the reactive API even though no technical reason exists for this limitation. However, the transpiler is limited only to allow references to constant variables — either the variable is declared as **const** or the value does not change after the initialization. This is intentional to avoid concurrent read-write access from different threads.

The current implementation only uses static scoping to

Runtime System	Version
Parallel.es	0.1.14
Hamsters.js	3.9.0[11]
Parallel.js	0.2.x[12]
Threads.js	0.7.2

Table 1. Versions of Evaluated Runtime Systems

resolve imports and functions. Functions and imports where the resolution requires data flow analysis is therefore not supported, e.g. it is unsupported to call a function that has been passed as a function argument. For the same reason, only ES6 modules and imports are supported.

6 Evaluation

The evaluation focuses on computations that are expected to profit from parallelization, e.g. compute-intensive tasks or tasks over a large set of data. However, using the presented runtime system may also be beneficial for long-running, but not parallelized computations that otherwise would block the main thread and result in a degraded user experience. The evaluation compares the presented work with the alternatives introduced in section 2 conserning performance and applicability to the following set of problems:

- Knight Tour: Computes the number of open tours from a given start field. Low memory usage but very high computational needs.
- Mandelbrot 10'000x10'000: Computation of a Mandelbrot for a given image size. This problem requires a relatively large amount of memory compared to the computational time needed.
- Risk profiling: Simulates the development of customer assets over a period of 15 years by using a Monte Carlo simulation. The simulation is used to determine the probability that the amount needed by an investment — e.g. buying a house after ten years — is available at this point in time. This problem is an instance of a real-world problem [14].

The versions used by the evaluated runtime systems are shown in table 1.

6.1 Applicability

The applicability is mainly evaluated by comparing the Mandelbrot implementations. The synchronous and Parallel.es based implementations of the Mandelbrot have been introduced in section 3 and are shown in listings 3.2 and 3.3. The implementation of the per line computation is omitted for brevity since it is almost identical for all libraries. The

---

[11]The version used is based on 3.9.0 but contains a fix for input data that is not a typed array (https://github.com/austinksmith/Hamsters.js/issues/16).

[12]Latest version from master as mentioned in section 2.

preliminary focus of the evaluation is on readability and type safety. However, some of the results might be subjective and represent the opinion of the author. The source code for all examples is available on GitHub [15].

Parallel.js The Mandelbrot implementation using the reactive API of Parallel.js is shown in listing 6.1. It differs only slightly from the synchronous implementation. A parallel task is created using the `Parallel` constructor (line 10). The first constructor argument is the data to process, the second — optional — is an options object affecting the task execution where the value of the `env` property is exposed as `global.env` in the background thread (line 13). The task function passed to the `map` operation (line 12) is called for every element in the input array and produces the elements in the output array.

```
1  function computeMandelbrotLine(y, options) {
2    const arraySize = options.imageWidth * 4;
3    const line = new Uint8ClampedArray(arraySize);
4    // compute...
5    return line;
6  }
7
8  const lines = _.range(options.imageHeight);
9
10 new Parallel(lines, {env: options})
11   .require(computeMandelbrotLine)
12   .map(function (line) {
13     return computeMandelbrotLine(line, global.env));
14   })
15   .then(result => console.log(result));
```

Listing 6.1. Mandelbrot Implementation using Parallel.js

The use of the global, undeclared variable `global` (line 13) to expose additional data in the task function is problematic since it breaks static scoping and requires additional care if typed languages are used. One way to work around the problem is to declare the `global` variable in every module it is used or globally in a declaration file. In both cases, no specific type can be annotated for the environment property since its type depends upon the actual problem. This only leaves the option to disable type checking for the environment property at all by annotating a special opt-out type like `any` in TypeScript. Furthermore, declaring the variable on a module basis or globally has the undesired side effect that the type checker no longer complains if the variable is used outside of a task function and is undeclared. The global variable also hinders that the same function can be called from the main thread. In this case, the `global` variable is undeclared and the function call fails.

A powerful feature of the Parallel.js is the ability to include additional functions in a task without the need for static code transpilation — as it is by Parallel.es — by using the `require` function (line 11). The included functions can then be called like normal functions from inside a task function.

To sum up, the API has the disadvantage not to be type safe. I further believe that the API is slightly less expressive than the one of the proposed work.

Threads.js The Mandelbrot implementation using Threads.js is shown in listing 6.2. Threads.js can be used with or without thread pools. A thread pool needs to be created manually if one is desired (line 8). The task function is defined using the `pool.run` method (line 9). A new task for this function is created every time the `send` method (line 15) is called where the passed in value is used as the argument for the task function. The result of a single task can be retrieved by registering the `done` handler (line 16). The sub-results of the tasks have to be joined manually. The thread pool offers the event `finished` (line 19) that is called when all tasks of this pool have been completed[13].

```
1   function computeMandelbrotLine({ y, options}, done) {
2     const arraySize = options.imageWidth * 4;
3     const line = new Uint8ClampedArray(arraySize);
4     // compute...
5     done.transfer(line, [line.buffer]);
6   }
7
8   const pool = new Pool();
9   pool.run(computeMandelbrotLine);
10
11  const lines = _.range(options.imageHeight);
12  const result = new Array(options.imageHeight);
13  for (const y of lines) {
14    pool
15      .send({ y, options })
16      .on("done", line => result[y] = line);
17  }
18
19  pool.on("finished", () => console.log(result));
```

Listing 6.2. Mandelbrot Implementation using threads.js

The Threads.js API offers a clean, flexible, messaging based API to run single tasks in a background thread but does not offer a higher-level abstraction for common operations. This lack of higher-level API complicates the migration of existing code since the programmer needs to partition the work into different tasks and join the sub-results. Furthermore, the programmer is responsible for managing the lifetime of the thread pool. However, the API makes it impossible for a type checker to guarantee type safety. The problem is that the parameters of the function passed into `run` have to match the arguments passed to `send`. Therefore, whenever `run` is called, the type of the pool instance has to be changed to reflect the type of the expected `send` arguments. However, the `run` method does not return a new instance; it instead changes the existing one. Therefore, the change can not be reflected into the type signature of the pool instance.

Hamsters.js The implementation of the Mandelbrot is shown in listing 6.3. A task is started using the `hamsters.run` method (line 14). The passed arguments have the following semantic:

1. An object that is passed to the task function. The special property `array` defines the input data. The object is exposed as the `params` variable (line 2) in the task function.

2. The task function to execute.
3. Callback function that is invoked when the operation has completed.
4. The number of tasks to create at most — into how many tasks should the input data be partitioned.
5. Defines if the sub-results of the tasks are automatically joined (**true**) into the end-result.

Hamsters.js automatically splits the input data into sub-arrays that are to be processed by a single task. However, iterating over the elements in the sub-array is left to the task function (line 6). The result of the task function must be written into the `rtn.data` array (line 10) that is provided by Hamsters.js.

```
1   function computeMandelbrotLine () {
2     const options = params.options;
3     const input = params.array;
4     const arraySize = options.imageWidth * 4;
5
6     for (let i = 0; i < input; ++i) {
7       const y = input[i];
8       const line = new Uint8ClampedArray(arraySize);
9       // compute...
10      rtn.data.push(line);
11    }
12  }
13
14  hamsters.run(
15    params: {
16      array: _.range(options.imageHeight),
17      options
18    },
19    computeMandelbrotLine,
20    result => console.log(result),
21    hamsters.maxThreads,
22    true);
```

Listing 6.3. Mandelbrot Implementation using Hamsters.js

The API of Hamsters.js is a mixture of a low- and high-level API: On one hand, it offers only a single `run` method, on the other advanced features like work partitioning, result joining, and even result caching. I believe that exposing all these features in a single method makes the API hard to use. It is difficult to remember the correct ordering and semantic of the arguments. Even though a high-level API is offered, still most of the work is left to the programmer like iterating over the input array elements. The API further has the disadvantage not to offer type safety because of the global, undeclared `params` and `rtn` variables in the task function. These global variables also hinder code reuse because they are undeclared if the function is not used as a task function.

6.2 Performance Comparison

The benchmark results from fig. 2 show the absolute time needed by each implementation to compute the solution and as well a percentage indicating the time relative to the synchronous implementation. The test setup uses Firefox 50 on a Windows 10 computer with a 4-Core, 2.5 GHz Xeon E5-2609v2 processor[14]. The results are explained in the rest

---

[13]The finished event cannot be used if other tasks are executed on the same thread pool.

[14]The results of the benchmarks only differ slightly when using Chrome or Safari. This is mostly rooted in the cause that the optimization performed

of this section.

Knight Tour   The time needed to solve the knight tour problem is mainly determined by the available computational resources. The knight tour calculation is parallelized by computing the number of tours starting from a specific start-field sequence and summarizing the number of found tours at the end.

Parallel.js creates new tasks that are executed on fresh worker instances for accumulating the sub-results computed by two tasks. This results in a significant overhead for the smaller 5x5 board. However, the impact is no longer visible for the larger board where the computation of the tours takes a multitude of the accumulation overhead caused by accumulating in separate tasks.

The usage of a thread pool that avoids spawning new background threads for every task does not bring the expected performance gain. The overhead for creating background threads in Firefox seems to be very inexpensive so that the effect is not visible at all. However, the benchmarking results of Google Chrome shown in fig. 3 give evidence that a thread pool might be beneficial for very short running tasks. Thus, Hamsters.js and Parallel.es achieve slightly better results than Parallel.js, that is not using a thread pool at all, and Threads.js, where a new thread pool is created manually for each execution[15].

The test case of the 6x6 knight tour only shows significant differences for the Hamsters.js runtime system. The difference is rooted in the used distribution strategy of the start-field sequences onto the tasks. Hamsters.js creates as many tasks as background threads are available by evenly distributing the start-field sequences across the background threads. However, some start-field sequences require more time to compute than others, resulting in unused computation resources when other tasks complete early. Parallel.js and Threads.js always use a task size of 1 to avoid this misfortune situation. Parallel.es also uses an even distribution but by default creates four-times as many tasks as background threads are available. This has shown to be a good balance between having a large enough set of items to process by each task reducing the overhead for the task processing while still leaving some room to compensate for nonlinear problems.

Mandelbrot   The Mandelbrot problem is parallelized by computing a subset of the lines per task. The computation time needed to compute a single line depends upon the position of the line in the image — it is a nonlinear problem. This is the reason why the Hamsters.js based implementation takes significantly longer. Its even distribution strategy of the work onto the background threads results in tasks computing the center of the Mandelbrot taking longer than the ones at the top or bottom of the field.

The performance gain of Threads.js compared to the other runtime system is rooted in the fact that Threads.js supports transferables [17, Section 2.7.4]. Transferables allow moving a memory range between threads instead of copying it. Hamsters.js also support transferables, however, only if the input and output are objects supported by the transferable standard what is not the case for the Mandelbrot implementation.

Risk profiling   The risk profiling implementation uses sim.js [18] in the Monte Carlo simulation as random number generator that supports seed numbers. This is needed to achieve reproducible forecasts. Hamsters.js lacks support for importing functions from external files and is therefore not part of this evaluation. The problem is parallelized by computing the outcome for a subset of investments in each task. However, this requires that each background thread has to perform the Monte Carlo simulation to get the data needed to calculate the outcome for an investment. Therefore, only a smaller speedup can be achieved by parallelizing this problem since the simulation requires significantly more time to compute than for calculating the outcome of a single investment.

Parallel.es requires more time for the computation because of the work splitting strategy used. Parallel.es distributes the investments evenly onto the background threads. However, computing the result of an investment is nonlinear. It depends on the year in which the investment takes place, the later this is the case, the more values have to be computed. This nonlinear computation time results in some tasks completing earlier than others leaving computation resources unused. Specifying a task-size of one is not a solution for this problem as this result in recomputing Monte Carlo simulation for each investment reducing the performance even more. Parallel.js and Threads.js can only use a task-size of one as the thread pool is not reused and therefore, side effects in the background-threads can be used to temporary store the simulation outcome in a global variable. This is not desired if a shared background threads from a thread pool are used as it results in having potential memory leaks.

Recursive Tasks   TODO oder Weglassen? Knight Tours ist eigentlich ein Recursive Task. Benötigt Momentan eine Aufteilung, damit überhaupt lösbar. Anderes Beispiel wäre Quick Sort. Ggf. noch Beispiel von Github suchen? Ebenso nicht lösbar bei allen Implementierungen sind Tasks welche gegenseitige Wartebedinungen haben (ok, mit viel würgen geht das evtl mit Threads.js)

---

by the JIT compiler differs between browsers. However, the performance of parallel computations drops significantly in Microsoft Edge if the runtime system uses **new** `Function` or `eval` to create dynamic function instances — that is the case for Hamsters.js and Threads.js. This observation has been reported and is confirmed by Microsoft [16].

[15]This is not strictly necessary for the knight tour problem. However, it is needed to store the simulation result of the risk profiling problem.

7   Conclusion

A typical JavaScript application runs only inside of the main thread and therefore, does not make use of all the available computation resources. However, with the increasing complexity of the problems implemented in JavaScript, the immersed rise of mobile and IoT devices in the world-wide
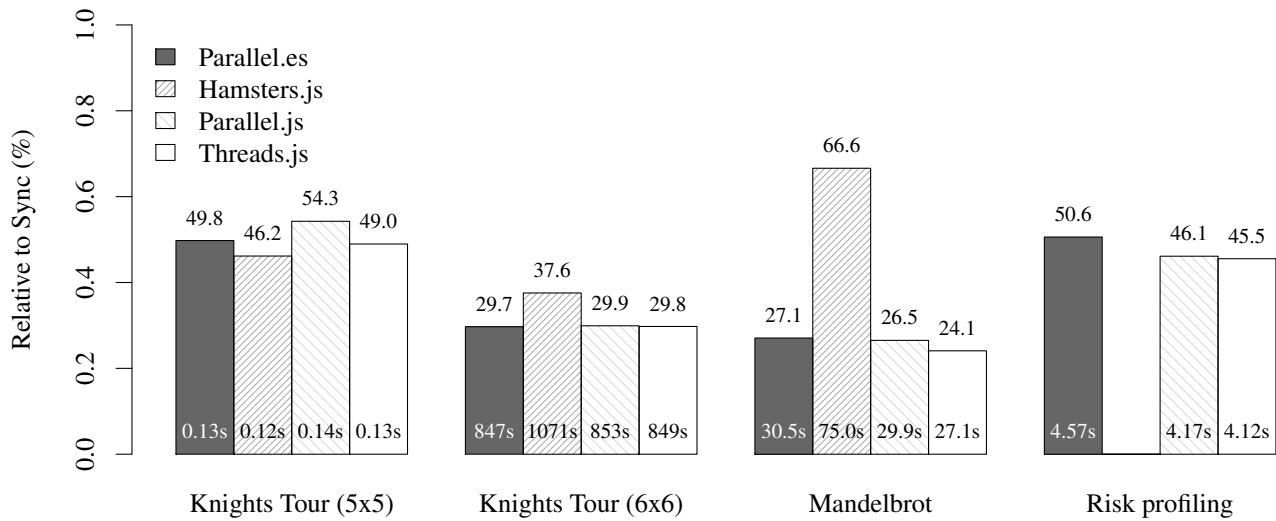
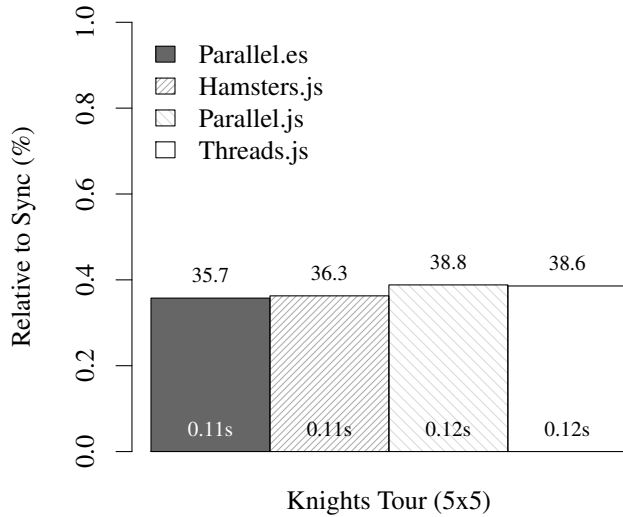Figure 2. Runtime Performance of Parallelization Problems Relative to Synchronously Execution

Figure 3. Knight Tour 5x5 Runtime Performance using Chrome

with a relatively low CPU clock but multiple cores, and stagnating CPU frequencies, the adoption of applications to multicore architecture is more essential than ever. All the prominent JavaScript environments answered to this trend with an environment-specific standard for spawning background threads where most use a messaging based communication model between threads. However, this jungle of standards and the programming model gap hardens writing application taking profit of the multicore hardware available.

This paper presented Parallel.es, a runtime environment independent, type safe API and runtime system that aims to provide a seamless integration of parallel tasks into the existing programming model to address this need. It addresses the different needs of programmers by providing two APIs,

a low-level API that allows running a single function in a background thread, but otherwise, does not offer any assistance and a reactive based API that allows parallelizing data flow based operations with ease. Furthermore, a transpiler has been presented that hides most of the limitations rooted in the programming model gap by rewriting the relevant code and allowing an even "seam-lesser" co-existence of parallel tasks and synchronous code.

The evaluation showed that most existing runtime systems perform similarly concerning execution time when applied to the given set of problems. Parallel.ese is neither the fastest runtime-system nor does is it significantly slower than others in any problem instance. Besides performance, a key factor when choosing a library is its API and integration into existing code and tools. The evaluation showed that none of the related work offers a true type safe API as it is by Parallel.es. Therefore, using Parallel.es together with typed languages like Flow or TypeScript is clearly to be preferred over the other runtime systems. Besides type safety, the API has been evaluated, but clearly, states the opinion of the — biased – author. Hamsters.js API is the least trivial to use because all features are exposed by a single function, resulting in a very lengthy argument-list. The API of Threads.js is clean and simple in use, but its programming model also uses messaging that hardens transitioning existing code and code reuse. The API offered by Parallel.js and Parallel.es are almost equivalent and allow easy parallelization of operations applied to data streams. A unique feature of Parallel.es achieved by using the transpilation is to offer same debugging functionality as developers are used to from synchronous code.

References

[1] "Web Worker", W3, Tech. Rep., Sep. 2015. [Online]. Available: https://www.w3.org/TR/workers/.

[2] Node.js Foundation. (2016). Child process, [Online]. Available: https://nodejs.org/api/child_process.html (visited on 11/13/2016).

[3] Ringo. (2016). Ringojs, [Online]. Available: http://ringojs.org/ (visited on 11/21/2016).

[4] DocumentCloud. (2016). Underscore.js, [Online]. Available: http://underscorejs.org/ (visited on 11/11/2016).

[5] JS Foundation. (2016). Lodash, [Online]. Available: https://github.com/lodash/lodash (visited on 11/11/2016).

[6] A. Smith. (2016). Hamsters.js, [Online]. Available: http://www.hamsters.io/ (visited on 11/12/2016).

[7] A. Savitzky and S. Mayr. (2016). Parallel.js, [Online]. Available: https://parallel.js.org/ (visited on 11/11/2016).

[8] A. Wermke. (2016). Threads.js, [Online]. Available: https://github.com/andywer/threads.js (visited on 11/11/2016).

[9] "ECMAScript® 2015 Language Specification", Ecma International, Tech. Rep., Jun. 2015. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/ (visited on 11/14/2016).

[10] "ECMAScript Shared Memory and Atomics", Ecma International, Tech. Rep., Nov. 2016. [Online]. Available: http://tc39.github.io/ecmascript_sharedmem/shmem.html (visited on 11/13/2016).

[11] "HTML", WHATWG, Tech. Rep., Nov. 2016. [Online]. Available: https://html.spec.whatwg.org/multipage/index.html (visited on 11/13/2016).

[12] JS Foundation. (2016). Webpack, [Online]. Available: https://webpack.js.org (visited on 11/11/2016).

[13] Babel. (2016). Babel, [Online]. Available: http://babeljs.io/ (visited on 11/25/2016).

[14] J. Zaugg. (Jun. 1, 2015). Risikoprofilierung des Anlagekunden, Vom Papierfragebogen zur digitalen Visualisierung, ZHAW, [Online]. Available: http://pd.zhaw.ch/publikation/upload/208631.pdf (visited on 11/20/2016).

[15] M. Reiser. (2016). Parallel.es examples project, [Online]. Available: https://github.com/MichaReiser/parallel-es-example (visited on 11/11/2016).

[16] ——, (Nov. 2016). New function in web workers significantly slower, [Online]. Available: https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/9779930/ (visited on 11/20/2016).

[17] "HTML5", W3, Tech. Rep., Oct. 2014. [Online]. Available: https://www.w3.org/TR/html5/ (visited on 11/11/2016).

[18] M. Varshney. (2016). SIM.JS - Discrete Event Simulation in JavaScript, [Online]. Available: http://www.simjs.com/ (visited on 11/20/2016).