Parallelize JavaScript Computations with Ease

Micha Reiser

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

Abstract

JavaScript evolves to a general purpose language. Simultaneously, the complexity of its applications is rising, demanding for even more computational resources that can no longer be satisfied by a single-threaded runtime-system. However, the JavaScript community has not widely employed multithreading. Simply because the available standards are platform-dependent and enforce a messaging-based programming model that does not integrate seamlessly into existing applications. This paper presents *Parallel.es*, a platform-independent, type-safe API and runtime system simplifying the creation of multithreaded applications in JavaScript by abstracting the messaging-based programming model. Parallel tasks are defined by normal JavaScript functions and are executed concurrently in background threads. The runtime system further offers a reactive API simplifying the parallelization of data-stream-based operations by facilitating automated work partitioning and result joining. The evaluation shows that the runtime system performs well compared to related work. Nevertheless, the proposed system mainly shines because of its seamless integration into existing code bases and the type-safety of its API. Moreover, the same debugging experience as for synchronous code can be provided if the presented transpiler is used that creates source maps for parallel tasks pointing back to the original code.

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 runtime system without negative affecting the user experience. Only using a single thread 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 low clock rate but a higher number of CPU-cores. These circumstances require 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 multithreaded 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 memory model of JavaScript is not defined for concurrent access. The absence of a shared memory 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. Moreover, code running in background threads must be located in designated files. This division of the code 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 multithreaded 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 multithreaded JavaScript — including shared memory and all problems shared state brings with it. This jungle of standards requires the specific adoption of applications to the standards provided by the used runtime environments.

The author believes 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 multithreaded applications in JavaScript. This paper presents *Parallel.es*, a platform-independent type-safe API and runtime system for creating multithreaded applications in JavaScript. The API abstracts the messaging-based programming model used by some of the underlying standards, allowing a seamless integration into existing code. The runtime system consists of two parts: Firstly, a low-level API allowing to run single functions on a background thread, and secondly 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 provides more flexibility for the cases where the reactive programming model does not fit well with the problem to solve.

This work further presents a transpiler that hides most 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 is optional.

The first section describes the related work and compares it with Parallel.es. The following section describes the programming model and is followed by the section 4 that explains the functionality of the runtime system. Section 5 describes the functioning and benefits of the transpiler. The section 6 evaluates the presented runtime system 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 of 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 and uses a global thread pool to manage the created background threads. It supports transferable objects and provides various helper functionalities like array sorting, aggregating, or caching.

The main difference to Hamsters.js is that the API of the proposed work is type-safe and also offers a high-level API that simplifies parallelization. Hamsters.js further has the limitation that no functionalities of external libraries can be exposed in the backgorund threads.

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 automatic work partitioning and result joining.

Parallel.js differences in two points from the presented work. First, it does not use a thread pool and therefore can not reuse background threads across operations (e.g. map or filter). Second, it 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 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 new background thread for every element in the input array exhausting the thread limit of the browser. The most recent version on GitHub has corrected this behavior by limiting the number of spawned workers. 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 simple to use and powerful multithreading library for NodeJS and the browser. It uses child processes when running in NodeJS and web workers when running in the browser.

The main difference of Threads.js is its messaging-based programming model that is nearer to the programming model used by the underlying technologies. Therefore, bridging the programming model gap is left to the programmer.

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. These callbacks are executed on the main thread and allow retrieval of the result or error.

[1]NPM is a package manager for JavaScript. The latest published version to date of Parallel.js is 0.2.1.

[2]Commit 2e4b36bf16e330abaaff213e772fcf4074fd866b

```
function fib(num) {
  if (num <= 2) {
      return 1;
  }

  return fib(num - 1) + fib(num - 2);
}

parallel.run(fib, 100)
  .catch(error => console.error(error))
  .then(result => console.log(result));
```

Listing 3.1. Fibonacci Implementation

Not every JavaScript function is a sound task function. This limitation is rooted insofar 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]. Illegal references to symbols from outer scope result in runtime errors thrown by the runtime environment. Therefore, all data needed by a task function has to be explicitly passed when the task is started and is serialized using the structured clone algorithm [11, Section 2.9.4]. The structured cloning enforces a pass-by-value semantic and requires that the passed data be serializable, e.g. Functions, DOM-Elements and Errors are not cloneable. The same applies to the result returned by the task.

The global context in which a task function is defined differs from the global context of the background threads executing the task function. Changes made to the global context of one thread are not visible to the other threads since each thread has its own global context and the runtime system does not reflect the changes among these. Therefore, the global context cannot be used to store shared state. Furthermore, the APIs accessible in background threads may vary from the one offered in the main thread, e.g. the DOM API is not accessible to web workers. These are no significant limitations for task functions since they perform in general compute intensive, but side effect free, operations only depending on local data.

Parallel tasks are isolated from one another since tasks share no variables and every thread executes one task at a time. However, the tasks executed in a specific background thread share the same global state. It is, therefore, possible that a task affects another task if they access and modify the same global state. Modification to the global state can not be prevented by the runtime system but are 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 [4] and lodash [5] 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. 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. Adopting a well-known and commonly used API facilitates a fast learning curve and 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 using lodash [5] 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. This callback is invoked in the main thread if the computation succeeds — an optional error callback can be defined as well.

```
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

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. The runtime system, 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

[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 restriction also implies that an arrow function used as task function is semantically equal to a function expression.

```
function computeMandelbrotLine(y, options) {
  const arraySize = options.imageWidth * 4;
  const line = new Uint8ClampedArray(arraySize);
  // compute...
  return line;
}

const result = _.chain()
  .range(options.imageHeight)
  .map(y => computeMandelbrotLine(y, options))
  .value();

console.log(result);
```

Listing 3.3. Synchronous Mandelbrot Implementation using Lodash [5]

slaves is limited to the number of logical processors provided by the hardware[5]. 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 has the advantage that performed JIT-optimizations are not thrown away if a task has completed. The caching can be especially useful for frequent but short running tasks for which the serialization and JIT-optimization overhead weight heavier.

4.2 Limitations

The current implementation supports the essential features. However, it misses support for asynchronous task operations and environments other than the browser. 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.

[5]The number of logical processors can be determined using `navigator.hardwareConcurrency`. The runtime system assumes the hardware has four logical processors if the used browser does not support this API.

5 Transpiler

The absence of a shared, 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 task functions are not allowed to refer to symbols from the outer scope. This restraint impedes code reuse and hinders a seamless integration into existing code. The presented transpiler hides these 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.

Listing 5.1 shows the Mandelbrot implementation referencing the `imageWidth` variable (line 12, 14) and the function `computePixel` (line 4) from the outer scope. The transpiler rewrites the code and makes the `imageWidth` variable and the `computePixel` function available in the task function. Listing 5.2 shows the transpiled Mandelbrot implementation. The transpiler creates the new function `_environmentExtractor` (line 9) that returns an object containing the values of all referenced variables and inserts it above the definition of the task function. This function is used to extract the current values of the referenced variables in the master thread. The object returned by the `_environmentExtractor` function is made available to the task function 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

[6]A JavaScript module bundler

[7]A framework for Transforming JavaScript code

[8]The transpiler does not remove the task function from the code run in the main thread since it might be used elsewhere. Webpack 2 can eliminate the function using tree shaking if it is not used elsewhere in the main thread.

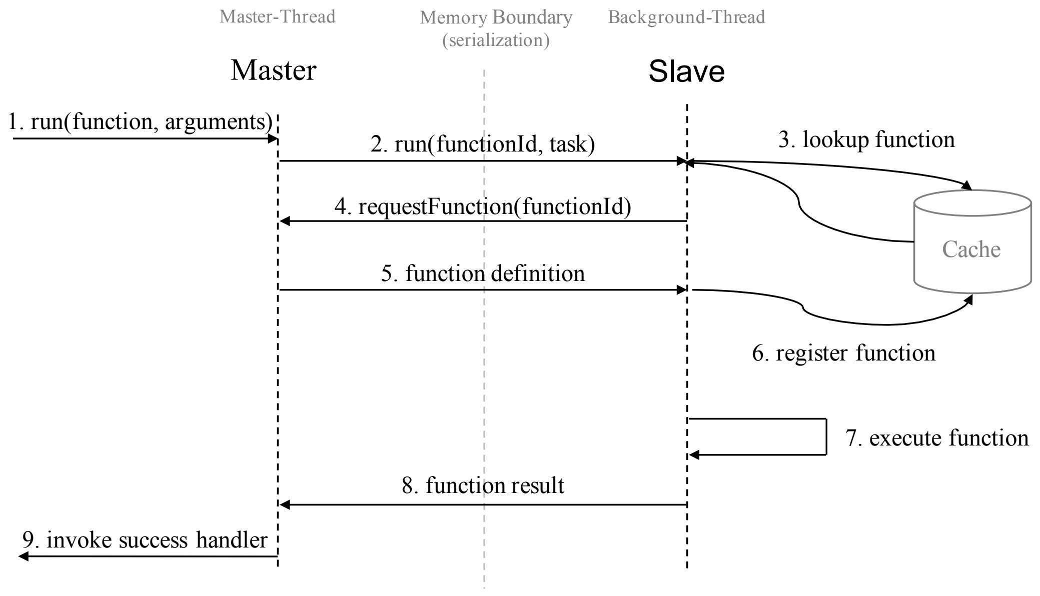

Figure 1. Parallel.es Runtime System

```
const imageWidth = 10000;
const imageHeight = 10000;

function computePixel(x, y) {
  //...
  return n;
}

function _environmentExtractor() {
  return {
    imageWidth: imageWidth
  };
}

parallel.range(imageHeight)
  .inEnvironment(_environmentExtractor())
  .map({
    identifier: "static:file.js/_entry_anonymous",
    _______isFunctionId: true
  });
```

Listing 5.2. Transpiled Mandelbrot Implementation

Listing 5.3 shows the code inserted by the transpiler into the script run by the slaves. 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 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 utilized in the master thread. This pre-registration 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. The source maps enable 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. The source map support is a distinct feature not offered by any of the related work.

```
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 the Transpiled Mandelbrot Implementation

[9]The transpiler wraps the functions of each module in the code run on the slaves with an immediately invoked function expression to isolate the functions of one module from the others and avoid naming clashes.

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

Runtime System	Version
Parallel.es	0.1.17
Hamsters.js	3.9.0[11]
Parallel.js	0.2.x[12]
Threads.js	0.7.2

Table 1. Versions of Evaluated Runtime Systems

5.1 imitations

The current implementation of the transpiler only supports the reactive API even though no technical reason therefore exists. By design, the transpiler prohibits references to non-constant variables — a variable is assumed to be constant if the value is assigned in the declaration and is never changed afterward. This restriction prevents visibility issues since a programmer might assume that the runtime system reflects changes to variables across threads.

The current implementation only uses static scoping to resolve imports and functions. Functions and imports where the resolution requires data flow analysis are therefore not supported, e.g. it is unsupported to call functions passed as function arguments. The transpiler only supports ES6 modules and imports for the same reason.

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 concerning 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 $10'000 \times 10'000$: Computation of a Mandelbrot for a given image size. This problem requires a relatively large amount of memory compared to the computational time needed.
- The risk profiling uses a Monte Carlo simulation to create forecasts for the customer's asset development over a period of 15 years for various investment strategies and assuming different states of the economic. The forecast is used to explain to the customer how the chosen investment strategy and the state of the economic effects the development of his assets and therefore, planned investments — e.g. buying a house after ten years. 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 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.

Parallel.es Listing 5.1 from section 5 shows the Mandelbrot implementation using Parallel.es. The implementation is almost identical to the lodash [5] based implementation shown in listing 3.3 facilitating a fast learning curve and simplifying the adoption of existing code. The API of Parallel.es is type-safe for a solid integration into projects developed with typed languages. Using the transpiler gives the programmer the liberty to structure the code as he is used to and not according to the requirements of the runtime systems. This freedom allows a seamless integration of parallel task that none of the other evaluated runtime systems achieves but comes at the cost that the source code needs to be transpiled potentially requiring an additional build step. However, the author believes that the benefits of a seamless integration outweigh the additional complexity in the build process. Especially because transpiling of source code — mostly using Babel [13] — is very common in the JavaScript community.

Parallel.js Listing 6.1 shows the Mandelbrot implementation using the reactive API of Parallel.js. 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.

The use of the undeclared variable `global` (line 13) to expose additional data in the task function is problematic since it breaks static scoping and requires additional care in typed languages. Typed languages require that the variable `global` is declared. The variable can either be declared 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. Therefore, type checking needs to be disabled for the environment property by annotating a special opt-out type like `any` in TypeScript. However, declaring the variable has the undesired side effect that the type checker no longer complains if the variable is used outside of a task function in which case the variable is truly undeclared. The `global` variable also

[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.

```
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

```
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

hinders code reuse because the global variable is undeclared if a task function is called from the main thread.

The implementation of the risk profiling problem in Parallel.js requires some tricks to be performant. The issue is that Parallel.js provides no mean to store the Monte Carlo simulation results across the invocations of the task function in the background thread other than saving it in the global context so that the simulation does not need to be recomputed for each investment. Storing the simulation result in the global context is unesthetic but at least can not introduce memory leaks since Parallel.js terminates the background threads when the operation has completed. However, an explicit API from Parallel.js would be desired that also remains functional if Parallel.js is using thread pools in the future.

A powerful feature of 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 and does not provide an API to store data across task function invocations. The author further believes that the API is slightly less expressive than the one of the proposed work.

Threads.js Listing 6.2 shows the Mandelbrot implementation using Threads.js. 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 by invoking the `send` method (line 15). The arguments passed are used to invoke the task function in the background thread. The result of a single task can be retrieved by registering the `done` handler (line 16) that is used in this example to manually join the sub-results of the tasks. The thread pool offers the `finished` event (line 19) that is triggered when all tasks of this pool have been completed[13].

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

The implementation of the risk profiling problem in Threads.js requires storing the Monte Carlo simulation results in the global context to be performant for the same reason as for Parallel.js. However, the misuse of the global context to store the results can introduce memory leaks in Threads.js if a shared thread pool is used. It is, therefore, desired to have an explicit API provided by Threads.js to store data across task function invocations that are only computed once.

To sum up. The API of Threads.js is simple in use but does not provide commonly like features like joining the sub-results. The API is neither type-safe making it a nonideal choice for projects using typed language.

Hamsters.js Listing 6.3 shows the Mandelbrot implementation using Hamsters.js. 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 in a background thread.
3. A 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 where a single task processes one sub-array. However, iterating over the elements of the sub-array is left to the task

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

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 are provided. The author believes that exposing all these features in a single method makes the API hard to use. It is hard to remember the correct ordering and semantic of the arguments. Even though Hamsters.js offers a high-level API, 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 be type-safe because of the undeclared `params` (line 2) and `rtn` (line code:hamsterjs-result) variables in the task function. These variables also hinder code reuse because they are undeclared if the function invoked not as 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 a Windows 10 computer with a 4-Core, 2.5 GHz Xeon E5-2609v2 processor. The benchmark has been performed using different browsers. Some of the benchmarks differ significantly depending upon the used browser. These discrepancies are caused by the different JIT-optimizations performed by a browser. The optimization supported, and strategies used are very particular to a browser. Microsoft Edge shows the most notable discrepancies since the performance of parallel computations drops significantly 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 [15]. The following section describes the benchmark results measured using Firefox v.50.

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 for accumulating the subresults of start-field sequences computed by two tasks and executes them on designated background threads. That results in a significant overhead for the smaller 5×5 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 results of Firefox do not indicate that using a thread pool to avoid spawning new background threads for every task is beneficial. It seems that creating background threads in Firefox is very inexpensive. However, the benchmarking results of Google Chrome shown in fig. 3 give evidence that a thread pool might be advantageous 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[14].

The test case of the 6×6 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. The number of background threads to use has been defined manually to the number of logical processors offered by the hardware because Hamsters.js does not determine the optimal number of background threads automatically. The start-field sequences are then distributed evenly onto the available 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 strategy but creates four-times as many tasks as background threads are available for better balancing the workload if the problems are nonlinear. This strategy has shown to be a beneficial balance between having a large enough set of items to process by each task to reduce the overhead for starting the tasks while still leaving room to compensate for nonlinear problems.

Mandelbrot The Mandelbrot problem is parallelized by computing a subset of the lines per task. The time needed to compute a single line depends upon the position of the line in the image — it is a nonlinear problem. This nonlinearity 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 insofar that Threads.js sup-

[14]A new thread pool for each run is not strictly necessary for the knight tour problem. However, it is needed to store the simulation result of the risk profiling problem.

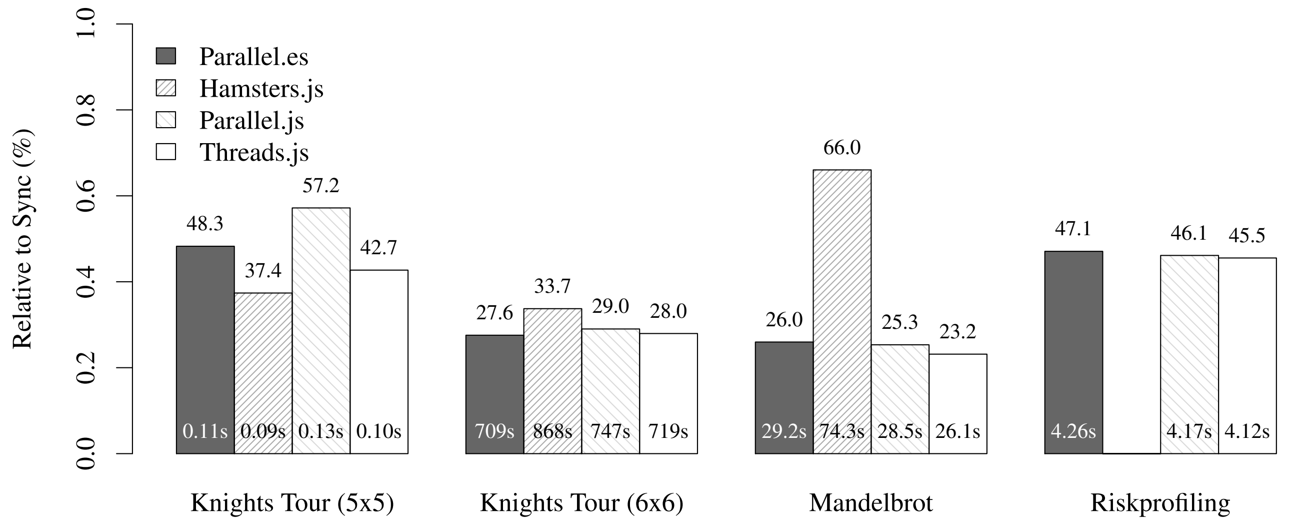

Figure 2. Runtime Performance of Parallelization Problems Relative to Synchronous Execution

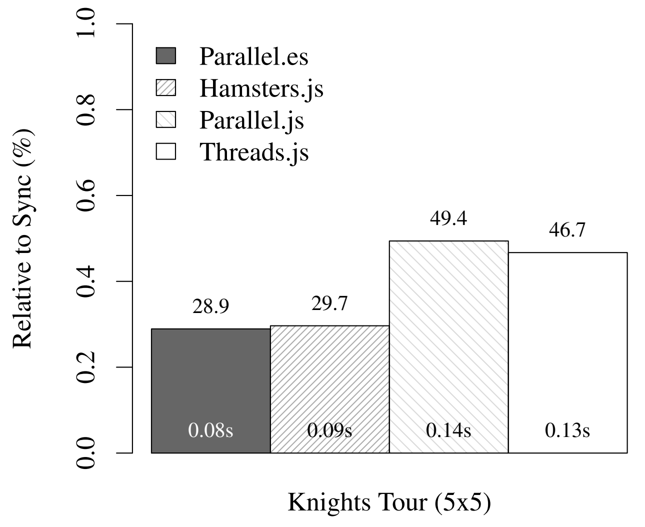

Figure 3. Knight Tour 5x5 Runtime Performance using Chrome

ports transferables [16, 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 transferable objects what is not the case for the Mandelbrot implementation.

Risk Profiling The risk profiling implementation uses sim.js [17] in the Monte Carlo simulation as random number generator that supports seed numbers. A random number generator supporting seed numbers is needed to achieve reproducible forecasts. Hamsters.js is not part of this evaluation since it lacks support for importing functions from external files and can therefore not use the sim.js library.

The problem is parallelized by computing the outcome for a subset of investments in each task. However, this requires that each background thread performs the Monte Carlo simulation to get the data needed to calculate the outcome of the planned investment. Therefore, parallelizing this problem can only achieve a smaller speedup since the Monte Carlo simulation is computationally significantly more intensive than 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, the more values have to be computed. This nonlinear computation time results in some tasks completing earlier than others leaving computation resources unused. Enforcing a task-size of one is not a solution for this problem as this lead to recomputing the 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. Using side effects is not desired if shared background threads from a thread pool are used as it results in potential memory leaks.

Recursive Tasks None of the evaluated libraries allow modeling recursive problems like the Knight-Tour or Quicksort naturally. Recursive problems have the characteristic that the input data for the subproblems is computed in the same step as the problem is solved. The backtracking based Knight Tour algorithm starts with a field and creates branches for every possible move by recursively descending for each distinct subpath allowing to parallelize the problem by computing each path in a separate task. This strategy requires a runtime system allowing to start subtasks from inside a task. These created subtasks can be executed on any background thread to achieve a better work balance. The current implementation does not support this scenario and therefore, the main thread precomputes start-field sequences that are solved synchronously inside of a single task without further dividing into subtasks.

An efficient implementation to support recursive tasks requires a communication channel between all web workers that allows starting a task from one 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. Shared Workers allow a worker to have multiple channels between various workers. However, older browsers often lack support for Shared Workers [1, section 4.6.4]. Furthermore, the "run to completion" model of JavaScript can be problematic because a busy background thread does not process received answers — and therefore, does not replay to requests — until the current work has completed.

Further research is required to determine how recursive tasks can be supported in an environment without shared memory and the "run to completion" model.

6.3 Summary of Evaluation Results

The evaluation of the performance shows that the result is mainly determined by the used work splitting strategy. Surprisingly, the overhead needed to run a task on a background thread is almost negligible. Therefore, a task size of one —- as used by Parallel.js and Threads.js — seems generally to be a better choice than processing too many items in a single task. The latter is preliminary problematic if the problem itself is nonlinear in which case a smaller task size would help to balance the workload. The approach of creating one task for each logical processor used by the Hamsters.js based implementation is the cause why it performs significantly worse than the other runtime systems for some of the problem instances. The approach used by Parallel.es — creating four times as many tasks as the hardware provides logical processors — showed to be a good choice.

The evaluated systems differ more significantly in their APIs. Some of the evaluated systems only offer a low-level API while other also provide a high-level API simplifying their handling. If one or the other is to be preferred is very subjective and principally depends on the specific problem and the programmer's preferences. However, other properties are more objective. Hamsters.js impedes code reuse and is unsuitable if libraries are needed in task functions since it does not permit to expose additional functions, e.g. functions imported from libraries, to task functions. For projects using typed-languages Parallel.es is the only valid option because none of the related works APIs are type-safe.

All the runtime systems have in common that they specify how a task function needs to be structured. This lack of freedom creates a clearly visible seam between parallel tasks and the rest of the application. Parallel.es can reduce this seam to a minimum if the presented transpiler is used that hides most of the limitations allowing a far more complete set of JavaScript functions to be used as task functions.

7 Conclusion

A typical JavaScript application runs only inside of the main thread and therefore, only sparsely uses the available computation resources of nowadays computers. However, with the increasing complexity of the applications running in JavaScript, the immersed rise of mobile and IoT devices with a relatively low CPU clock, and stagnating CPU frequencies, the adoption of applications to multicore architecture is indispensable. Even though, writing multithreaded JavaScript applications is non-trivial because the standards for creating background threads are platform-dependent and often enforce a messaging-based communication model that introduces non-inherent complexity to applications.

This paper presented a runtime environment independent, type-safe API and runtime system that provides a seamless integration of parallel tasks into existing applications. 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 and a reactive API allowing parallelizing data flow based operations with ease.

The evaluation showed that most existing runtime systems perform similarly concerning execution time when applied to the given set of problems. Parallel.es is not always the fastest runtime-system, but never performs significantly slower than the others for any problem instance. However, the main advantage of the library over the related work is its API and integration into existing code. The proposed API is close to APIs commonly used by programmers to write synchronous code facilitating fast learning and straightforward transitioning of existing code. Moreover, Parallel.es is to be preferred for projects using typed language like Flow or TypeScript since none of the related work offers a type-safe API. Combining the runtime system with the transpiler has further the benefit of a seamless integration of parallel tasks into existing code that cannot be achieved by any related work. Without the transpiler, the programmer is forced to structure the code in the way supported by the runtime system resulting in a clear break in the code style. The transpiler further generates source maps pointing back to the original code helping identifying errors from production and enable a pleasant debugging experience, a feature not offered by any related work.

However, the evaluation also showed that the proposed system does not fit well with recursive problems like Quicksort or Knight-Tour that require a system supporting recursive tasks. Adding support for recursive tasks is non-trivial and subject of further research. Nevertheless, the proposed work eases creating multithreaded applications enabling them to use the computation power provided by the underlying hardware and hopefully contributes to the growth of multithreaded applications written in JavaScript.

8 Availability

The source code of the runtime system, the transpiler, and the implemented problem instances are published on GitHub together with the benchmark results [18]–[21]. The libraries are released under the MIT license [22]. The latest versions of the packages are as well published on NPM [23].

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. (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).

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

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

[18] M. Reiser. (2016). Parallel.es, [Online]. Available: https://github.com/MichaReiser/parallel.es (visited on 11/11/2016).

[19] ——, (2016). Parallel.es babel plugin, [Online]. Available: https://github.com/MichaReiser/babel-plugin-parallel-es (visited on 11/11/2016).

[20] ——, (2016). Parallel.es webpack plugin, [Online]. Available: https://github.com/MichaReiser/parallel-es-webpack-plugin (visited on 11/11/2016).

[21] ——, (2016). Parallel.es examples project, [Online]. Available: https://github.com/MichaReiser/parallel-es-example (visited on 11/11/2016).

[22] (2016). MIT License, [Online]. Available: https://opensource.org/licenses/mit-license.php.

[23] npm Inc. (2016). NPM, [Online]. Available: https://www.npmjs.com/.