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 because the available standards are platform-dependent and enforce a messaging-based programming model. This paper presents *Parallel.es*, a platform-independent type-safe API and runtime system allowing to create multithreaded applications in JavaScript with ease. The runtime system abstracts the messaging-based programming model for a seamless integration into existing program code. Background tasks are defined by standard 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 is mainly superior because of its seamless integration into existing code and the type-safety of its API.

1 Introduction

The role of JavaScript drastically changed in recent years. It emerged from a browser-only to a general-purpose language used to write web-, desktop-, mobile-, and server-applications. This shift comes along with more sophisticated requirements demanding for more computational resources that can no longer be provided by a single-threaded runtime system without negatively affecting the user experience. Moreover, the need for multithreaded runtime systems and applications has intensified in recent years because of the stagnating CPU clock rates and the spreading use of mobile devices with low clocked, multicore CPUs.

The W3C responded to these new demands with the web worker draft in 2009, laying the foundation for creating multithreaded applications [1] in the browser. The draft defines the web worker API that allows a script to run in a background thread without interfering with the user interface. The communication between threads is messaging-based, and each thread has its distinct memory area. However, the messaging-based programming model does not fit well into existing applications and results in a clear seam between existing code and code running in background threads. Moreover, code running in background threads has to be located in designated files. This unnatural division of the code separates coherent logic and makes 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 background threads.

Unfortunately, implementing multithreaded applications targeting different runtime environments in JavaScript is non-trivial since no uniform standard for creating background threads exists. The web worker standard, defined by the W3C consortium, is only implemented in browsers, NodeJS provides the child-process API [2], and JavaScript applications running on the JVM can use RingoJS [3], a JVM-based multithreaded runtime environment. This jungle of standards requires the explicit adoption of applications to the standards provided by the used runtime environments.

The platform-dependent standards together with the inherent complexity caused by the programming model gap are most likely 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 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 designed to achieve simplicity in use by covering the aspects of work partitioning and result joining while providing a well-known and familiar API allowing 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. The work further presents a transpiler that reduces the programming model gap to a minimum by rewriting the program code before execution

allowing a seamless integration into existing code.

The remainder of this paper is structured as follow: Section 2 describes the related work and compares it with Parallel.es. Section 3 defines the programming model of background tasks. The execution of a background task and the design of the runtime system is described in section 4. Section 5 explains the functioning of the transpiler. The related work and the presented runtime system are evaluated in section 6 that 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. The presented work has two characteristics that none of the related work has: Firstly, it offers the same debugging functionalities as developers are used to when working with synchronous code and secondly, it allows code run in background threads to access functions and read-only variables from in the main thread.

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 integrates seamlessly into existing program code. Hamsters.js further has the limitation that functionalities of external libraries can not be used in a background thread.

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 and platform-independent API for multi-core processing in JavaScript. 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 and the presented work differ in three points: Firstly, Parallel.js does not use a thread pool and therefore, can not reuse background threads across operations, e.g. map or filter. Secondly, Parallel.js awaits the sub-results of the proceeding operation before continuing with the next operation if 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. Thirdly, the sub-results of an operation are transmitted back to the main thread before starting the next operation on new background threads resulting in the unneeded — and potentially very expensive — copying of intermediate results from and to background threads.

Remark 1. *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 restricting 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. The main difference of Threads.js is its messaging-based programming model that is closer to the programming model used by the underlying standards. 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 concurrently in background tasks without blocking the main thread.

3.1 Background Task

A *background task* — further referred to as task — represents an operation that is asynchronously started in the main thread and is executed on a background thread. Listing 3.1 shows an example of a task computing the Fibonacci number of 100 in a background thread. The `run` method creates a new task implemented by the passed function (line 9) that is invoked in a background thread with the provided arguments. Since the returned object implements the promise interface [9, Section 18.3.18] a `then` and an optional /catch/ callback can be registered. The `then` callback is invoked with the result if the computation is successful and the `catch` callback otherwise. These callbacks are executed on the main thread and allow the 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 JavaScript functions used to implement a task have to comply with the following discussed restrictions. References to non-read-only variables from the outer scope of a task

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

[2] Commit 2e4b36bf16e330abaaff213e772fcf4074fd866b

function are prohibited[3] as well as references to functions not resolvable by static scoping, e.g. a function passed as an argument. The transpiler detects and reports illegal references to non-read-only variables from a task function. The values of referenced variables, the arguments passed to the task function, and the value returned by the task function are passed "by value" and have to be structured cloneable [10, Section 2.9.4], e.g. this is not the case for DOM-Elements and Errors. The structured cloning is enforced by the underlying, messaging-based standards and can not be abstracted by the runtime system. The passing of not cloneable values results in runtime errors.

In contrast to standard JavaScript functions, the global context of the background thread executing the task function differs from the global context in which the function is defined. Therefore, 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. Furthermore, changes made to the global context of one thread are not visible to the other threads since each thread has its distinct global context. Therefore, the global context cannot be used to store shared state. These are no significant limitations for task functions since they, in general, perform compute-intensive, but side effect free, operations only depending on local data.

Tasks are isolated from one another since threads 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 context. It is, therefore, possible that two tasks affect each other if they access and modify the same global context. Modification to the global context are not prevented but are strongly discouraged as changes are only thread-local and may introduce memory leaks. The next section describes the reactive API that simplifies the parallelization of data-stream-based operations.

3.2 Reactive API

The runtime system further offers a reactive API [11]. The goal of it is to provide a well-known and understood API that uses the available computation resources without any further doing of the programmer. The API is, therefore, 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 task-results into the end-result. 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. It differs only slightly from the sequential, lodash [5] based implementation shown in listing 3.3. The differences between the implementations are highlighted in gray. This alikeness of the APIs facilitates a fast learning curve and simplifies transitioning of existing code.

The `range` method (line 18) defines the data stream to process. It creates a data stream containing the values from 0 up to the image height (exclusive). The input stream is transformed by mapping (`map` on line 19) each element from the input stream to an output element that is computed by the `computeMandelbrotLine` function (line 9). The `computeMandelbrotLine` function — that is executed in a background thread — has access to the current array element and the read-only variables from its outer scope. It can further make use of the `computePixel` (line 4) function defined in the outer scope or functions imported from other modules. The computation is started using the `then` method (line 21) that registers a callback. The *then* callback is executed in the main thread and is invoked with a single array containing the joined lines of the Mandelbrot if the computation succeeds. An optional error callback can be defined that is invoked in case the execution fails. The API further allows retrieving sub-results by registering the *next* callback using the `subscribe` method (line 20). The *next* callback is invoked whenever a task has completed and is passed the lines computed by this task, the index of this task, and the number of lines computed by each task. The sub-results can be used to show a progress update, e.g. drawing already computed lines of the Mandelbrot instead of waiting until all lines have been computed. The *next* callback is invoked in the main thread and the order of task completion. The next section describes the functioning of the runtime system orchestrating the background tasks.

```
1  const imageWidth = 10000;
2  const imageHeight = 10000;
3
4  function computePixel(x, y) {
5    // ...
6    return n;
7  }
8
9  function computeMandelbrotLine(y) {
10   const line = new Uint8ClampedArray(imageWidth * 4);
11   for (let x = 0; x < imageWidth; ++x) {
12     line[x * 4] = computePixel(x, y);
13   }
14   return line;
15 }
16
17 parallel
18   .range(imageHeight)
19   .map(computeMandelbrotLine)
20   .subscribe((subResult, index, batchSize) => ...)
21   .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: Firstly, the slaves running in background threads executing the tasks and secondly, the public API in the main thread forming the facade and acting as the master for the slaves. Applications are using the facade provided by the master to run a function in a background thread. The master is

[3] The special identifiers **`this`** and **`super`** are treated as references to non-read-only-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.

```
1  const imageWidth = 10000;
2  const imageHeight = 10000;
3
4  function computePixel(x, y) {
5    // ...
6    return n;
7  }
8
9  function computeMandelbrotLine(y) {
10   const line = new Uint8ClampedArray(imageWidth * 4);
11   for (let x = 0; x < imageWidth; ++x) {
12     line[x * 4] = computePixel(x, y);
13   }
14   return line;
15 }
16
17 const result = _.chain()
18   .range(imageHeight)
19   .map(computeMandelbrotLine)
20   .value();
21
22 console.log(result);
```

Listing 3.3. Sequentiall, Lodash [5] based Mandelbrot Implementation

responsible for spawning the slaves and distributing the work onto these. Hence, a thread pool is used to manage the created slave instances and queue the 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[4]. The next section describes how the runtime system executes a single task.

4.1 Task Execution

The steps needed to process a single task are shown in fig. 1. The application passes the task function together with the arguments for its invocation to the facade that acts as the master (1). The thread pool, residing in the master, selects an idle slave to execute the task or queues the task until a slave gets available. The master then transmits the structured cloned representation of the function call — consisting of a unique id identifying the task function and the arguments for the task function invocation — to this slave (2). The slave performs a lookup in the function cache to obtain the function with the given id (3). If the task function is executed for the first time on this slave instance, then it is not present in the function cache and, therefore, the slave requests the definition from the master (4). The master transmits the function definition to the slave (5) which deserializes and registers the function in the function cache (6). The slave calls the deserialized task 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 hand the result over 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]The number of logical processors can be determined using `navigator.hardwareConcurrency`. The runtime system assumes that the hardware has four logical processors if the used browser does not support this API.

4.2 Limitations

The current runtime system supports the essential features. However, it misses support for asynchronous task operations and runtime environments other than the browser. There are no conceptual or technical reasons for not supporting either of these features.

5 Transpiler

The absence of a shared memory that is accessible by all threads[5] and allows to store shared variables requires an explicit passing of the variables referenced by the task function to the background thread executing the task. The transpiler covers this explicit passing of the variables by rewriting the program code. It extracts and rewrites the task functions from the code run in the main thread and adds it to the file loaded by the slaves. The transpiler further adds the imports referenced by the task function and pre-registers it in the slave's function cache.

The functioning of the transpiler is following explained by using the Mandelbrot implementation shown in listing 3.2. The transpiler rewrites the program code to make the variable `imageWidth` (line 10) and the function `computePixel` (line 4) available to the task function. Listing 5.1 shows the transpiled Mandelbrot implementation in which the changes made by the transpiler are highlighted in gray. The transpiler creates the new function `_environmentExtractor` (line 9) that returns an object containing the values of the variables referenced by the task function and inserts it above the task function declaration. This function is used to extract the value of the referenced `imageWidth` variable in the master thread. The object returned by the `_environmentExtractor` function is set as the task environment by using the `inEnvironment` method (line 25). The runtime system passes the task environment as the last argument when invoking the task function in a background thread. The transpiler further replaces the reference to the task function with a unique function-id (lines 26-29)[6].

Listing 5.2 shows the code inserted by the transpiler into the script run by the slaves. The transpiler injects the code of the task function (lines 8-14), and the referenced `computePixel` function (lines 3-6)[7]. Furthermore, an *entry*-function (lines 16-24) is generated that initializes the `imageWidth` variable (line 1) with the value stored in the task environment (line 19) — that contains the values of the variables from the main thread — and calls the actual task function (line 20). The entry function is registered in the function cache (lines 26-29) using the same function-id as utilized in the master thread. This pre-registration allows the

[5]This might change with the SharedMemory [12] standard that is currently a draft in stage 2. However, the standard only allows storing variables of a very limited set of types.

[6]The transpiler does not remove the task function from the code run in the main thread since it might be referenced elsewhere. Removing unreferenced functions is left to minifiers as proofing a function to be truly unused is non-trivial.

[7]The transpiler wraps the functions of each module with an immediately invoked function expression to isolate the functions of different modules.

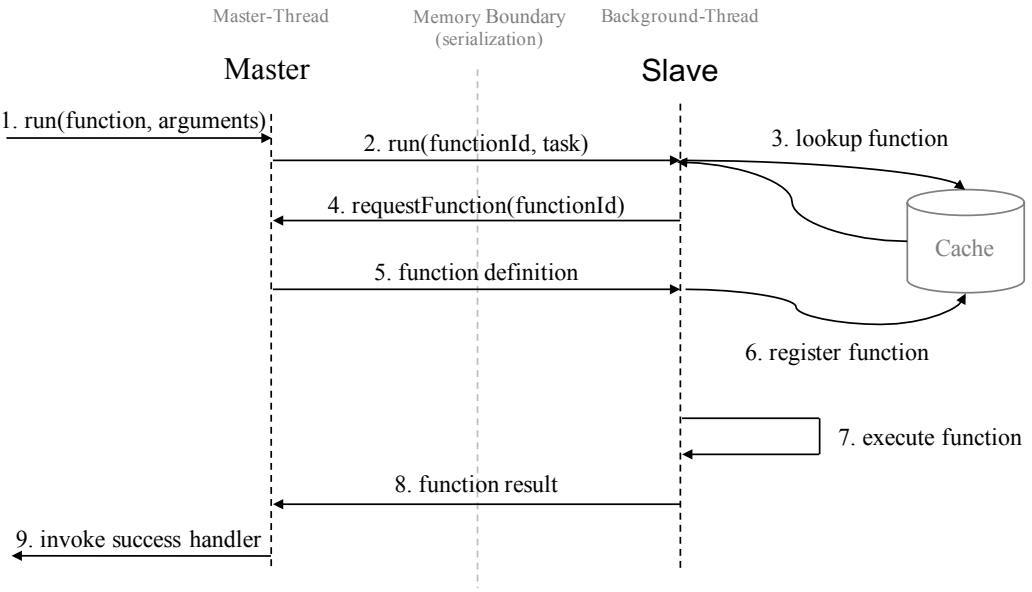

Figure 1. Parallel.es Runtime System

```
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 function computeMandelbrotLine(y) {
16   const line = new Uint8ClampedArray(imageWidth * 4);
17   for (let x = 0; x < imageWidth; ++x) {
18     line[x * 4] = computePixel(x, y);
19   }
20   return line;
21 }
22
23 parallel
24   .range(imageHeight)
25   .inEnvironment(_environmentExtractor())
26   .map({
27     identifier: "static:_entrycomputeMandelbrotLine",
28     ________isFunctionId: true
29   })
30   .then(result => console.log(result));
```

Listing 5.1. Transpiled Mandelbrot Implementation

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

Listing 5.2. Code Generated by the Transpiler that is Executed on the Slaves

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.

Additionally, the transpiler generates source maps that point back to the original location of the extracted task function and as well, transitively referenced functions. The source maps enable a true debugging experience allowing to set breakpoints inside of the browser developer tools[8]. 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. This translation of the error messages helps to locate the source of an error from production more easily. The source map support is a distinct feature not offered by any of the related work.

The transpiler is based on top of webpack[9] [13] and Babel[10] [14]. The use of the transpiler is optional if task functions are not referencing symbols from their outer scope. The limitations of the transpiler are discussed in the next

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

[9] A JavaScript module bundler.

[10] A framework for Transforming JavaScript code.

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

section.

5.1 Implementation Restrictions

The current transpiler implementation only supports the reactive API even though no technical reason therefore exists. Moreover, it only supports import statements according to the ECMAScript 6 module specification [9, Section 15.2].

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 applicability and performance using the following set of problems:

- Knight Tour: Computes the number of open tours from a given start field. This problem has low memory but very high computational needs.
- Mandelbrot $10'000 \times 10'000$: Computation of a Mandelbrot of the given image size. This problem requires a large amount of memory compared to the computational time needed.
- Risk Profiling: 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. The forecast is used to illustrate how a chosen investment strategy affects the development of the customer's assets and therefore, planned investments — e.g. buying a house after ten years. This problem is an instance of a real-world problem [15].

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

6.1 Applicability

The applicability is assessed by comparing the Mandelbrot implementations. The reference implementation is the sequential, lodash [5] based implementation of the Mandelbrot that has been introduced in section 3 and is shown in listing 3.3. The implementations for the specific runtime systems highlight the differences to the reference implementations in gray. The implementations omit the computation of a single pixel for brevity. The preliminary focus of the evaluation is the type-safety of the APIs and how seamless background tasks integrate into existing code. Some of the results might be subjective and represent the opinion of the author.

Parallel.es The Parallel.es based implementation of the Mandelbrot from section 3 and shown in listing 3.2 is almost identical to the sequential implementation. The implementation does not require any adjustment to the `computePixel` and `computeMandelbrotLine` functions because the task function can reference read-only variables and functions from its outer scope allowing a seamless integration of background tasks into existing applications. However, this liberty comes at the cost of requiring a potentially additional build step to transpile the program code. It is reasonable to assume that the benefits of a seamless integration outweigh the additional complexity in the build process. Especially because transpiling of source code — mostly using Babel [14] — 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 sequential implementation. A background task is created using the `Parallel` constructor (line 22). The first constructor argument is the data to process, the second — optional — argument is an options object affecting the task execution. The value of the options object's `env` property is exposed as `global.env` in the background thread (line 26). The task function passed to the `map` operation (line 25) is called for every element in the input array and produces the elements in the output array.

Functions called from inside of a task function have to be made explicitly available in the background thread by using the `require` method (line 23). Additionally, the variable `imageWidth` can not be referenced by the task function (and the `computeMandelbrot` function). Instead, the value needs to be explicitly passed to the task by storing it in the `env` property of the options object (line 22) and reading it in the task function from the `global.env` object (line 26).

The use of the undeclared variable `global` (line 26) 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 the variable `global` to be declared. It 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 [16]. However, declaring the `global` variable has the undesired side effect that the type checker no longer detects illegal usages of it outside of a task function in which case the variable

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

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

```
1  const imageWidth = 10000;
2  const imageHeight = 10000;
3
4  function computePixel(x, y) {
5    // ...
6    return n;
7  }
8
9  function computeMandelbrotLine(y, imageWidth) {
10   const arraySize = imageWidth * 4;
11   const line = new Uint8ClampedArray(arraySize);
12
13   for (let x = 0; x < imageWidth; ++x) {
14     line[x * 4] = computePixel(x, y);
15   }
16
17   return line;
18 }
19
20 const lines = _.range(imageHeight);
21
22 new Parallel(lines, { env: { imageWidth } })
23   .require(computeMandelbrotLine)
24   .require(computePixel)
25   .map(function (line) {
26     const width = global.env.imageWidth;
27     return computeMandelbrotLine(line, width));
28   })
29   .then(result => console.log(result));
```

Listing 6.1. Mandelbrot Implementation using Parallel.js

is truly undeclared. The `global` variable also hinders code reuse because it 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 other than saving it in the global context of the background thread. Storing the simulation result in the global context is unaesthetic but 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 favored that also remains functional if Parallel.js is using thread pools in the future.

To sum up, the API has the disadvantage not to be type-safe and does not allow to store data across task function invocations. Furthermore, variables and functions used in a task function need to be explicitly made available to the task, resulting in a clear break in the programming style. If this break can be accepted and type-safety is not a concern, then Parallel.js is a good choice as it allows to include additional functions in a task without the need for static code transpilation by using the `require` function (line 23).

Threads.js Listing 6.2 shows the Threads.js based Mandelbrot implementation. Threads.js can be used with or without thread pools. A thread pool needs to be created manually if one is desired (line 20). The `pool.run` method (line 21) specifies the task function. A new task for this function is created by invoking the `send` method (line 27) whereby the passed arguments are used to invoke the task function in the background thread. The result of a single task can be retrieved by registering a `done` handler (line 28) that, in this example, is used to join the sub-results of the tasks. The thread pool offers the `finished` event (line 31) that is triggered when all tasks of this pool have completed[13].

```
1  const imageWidth = 10000;
2  const imageHeight = 10000;
3
4  function computeMandelbrotLine({ y, width}, done) {
5    function computePixel(x, y) {
6      // ...
7      return n;
8    }
9
10   const arraySize = width * 4;
11   const line = new Uint8ClampedArray(arraySize);
12
13   for (let x = 0; x < width; ++x) {
14     line[x * 4] = computePixel(x, y);
15   }
16
17   done.transfer(line, [line.buffer]);
18 }
19
20 const pool = new Pool();
21 pool.run(computeMandelbrotLine);
22
23 const lines = _.range(imageHeight);
24 const result = new Array(imageHeight);
25 for (const y of lines) {
26   pool
27     .send({ y, imageWidth })
28     .on("done", line => result[y] = line);
29 }
30
31 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 background threads 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. Neither provides the API a mechanism to expose a function from the same module to a task function. Therefore, the `computePixel` function (line 5) has to be nested inside of the `computeMandelbrotLine` function. This missing mechanism restricts the programmer in his options to structure the code. Alike Parallel.js, the API of Threads.js is not type-safe since `run` does not return a new thread pool instance; it instead changes the task function of the existing instance. A new instance is required to reflect the relation between the task function's parameters, specified using the `run` method, and the arguments passed to `send`, that are used to invoke the task function, in the thread pools type signature.

The implementation of the risk profiling problem in Threads.js requires storing the Monte Carlo simulation results in the global context of the background thread to be performant for the same reason as for Parallel.js. However, this misuse of the global context introduces memory leaks if a shared thread pool is used. Therefore, an explicit API provided by Threads.js to store data across task function invocations would be favored.

[13] The finished event cannot be used to determine the completion of an operation consisting of multiple tasks if tasks of other operations are executed on the same thread pool.

To sum up. The API of Threads.js is simple in use but commonly used features like the joining of the sub-results are missing. The messaging-based programming model results in a clear break in the programming style. Moreover, Threads.js is a non-ideal choice for projects using typed language since the API is not type-safe.

Hamsters.js Listing 6.3 shows the Mandelbrot implementation using Hamsters.js. A task is started using the `hamsters.run` method (line 26). 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 by the `params` variable (line 10 and 11) in the task function.
2. The task function to execute in a background thread.
3. The 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 array 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 each sub-array is processed by a single task. However, iterating over the elements of the sub-array is left to the task function (line 15). The result of the task function has to be written into the `rtn.data` array (line 22) that is provided by Hamsters.js.

```
1  const imageWidth = 10000;
2  const imageHeight = 10000;
3
4  function computeMandelbrotLine () {
5    function computePixel(x, y) {
6      // ...
7      return n;
8    }
9
10   const options = params.options;
11   const input = params.array;
12
13   const arraySize = options.imageWidth * 4;
14
15   for (let i = 0; i < input.length; ++i) {
16     const y = input[i];
17     const line = new Uint8ClampedArray(arraySize);
18
19     for (let x = 0; x < width; ++x) {
20       line[x * 4] = computePixel(x, y);
21     }
22     rtn.data.push(line);
23   }
24 }
25
26 hamsters.run(
27   params: {
28     array: _.range(options.imageHeight),
29     options
30   },
31   computeMandelbrotLine,
32   result => console.log(result),
33   hamsters.maxThreads,
34   true);
```

Listing 6.3. Mandelbrot Implementation using Hamsters.js

The `run` method is a mixture of a low- and high-level API. It allows to run a single function on a background thread but also exposes advanced features like work partitioning, result joining, and result caching. This overloading of the function makes it difficult to use because remembering the right order and semantics of the arguments is challenging. Such methods, which accept many arguments and expose multiple features, are considered as a code smell. Even though the `run` method offers many advanced features, still much manual work is needed to parallelize a problem, e.g. iterating over the input array elements. Alike Threads.js, the `computePixel` function (line 5) needs to be nested inside of the `computeMandelbrotLine` function because Hamsters.js does not provide an API to expose additional functions inside of the task function. This restriction limits the programmer's possibilities to structure the code and results in a clear break in the programming style. The API further has the disadvantage not to be type-safe because of the undeclared `params` (line 10) and `rtn` (line 22) variables in the task function. These variables also hinder code reuse because they are undeclared if the function is not invoked as a task function.

To sum up. Alike Parallel.js and Threads.js, Hamsters.js does not provide a type-safe API. A more significant limitation is that functions from external modules can not be used inside of background tasks. It is, therefore, impossible to use any libraries in background tasks.

The next section evaluates the different runtime systems in concern of runtime performance.

6.2 Performance Comparison

The benchmark results from fig. 2 show the absolute time needed by each implementation and a percentage indicating the fraction of the sequential runtime. The computer used for benchmarking has a quad core, 2.5 GHz Xeon E5-2609v2 processor and runs Windows 10. The benchmark has been performed on all major browsers whereby some of the results differ significantly from one another dependent upon the used browser. These discrepancies are caused by the browser specific JIT-optimizations. 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 — which is the case for Hamsters.js and Threads.js. This observation has been reported and is confirmed by Microsoft [17]. The following section describes the benchmark results measured using Firefox v.50. These benchmark results are used because they do not contain outliers caused by the browser's JIT-Optimizations as it is the case for other browsers.

Knight Tour The time needed to solve the Knight Tour problem is mainly determined by the available computational resources. The calculation is parallelized by computing different start-field sequences in each task and then summarizing the number of found tours by each task.

Parallel.js creates new tasks for accumulating the sub-

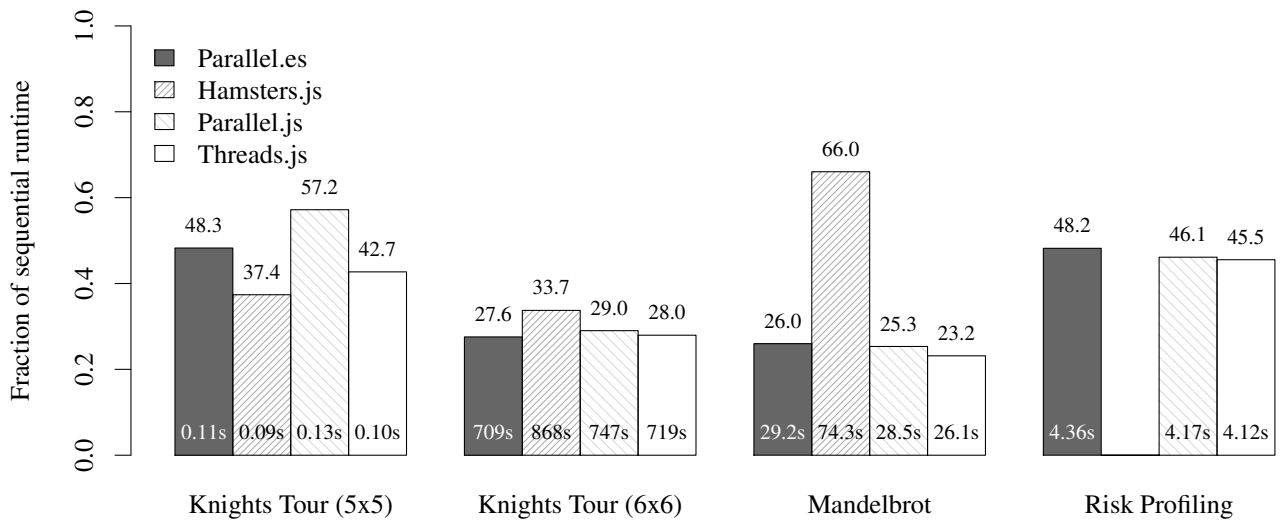

Figure 2. Runtime Performance of Parallelization Problems Relative to Sequential Execution

results of start-field sequences computed by two tasks and executes them on designated background threads. This spawning of new background threads for accumulating the sub-results cause a significant overhead for the smaller 5×5 board. However, the impact is negligible for the larger board.

The test case of the 6×6 Knight Tour only shows significant differences for the Hamsters.js runtime system. This difference is rooted in the strategy used to split the start-field sequences into tasks. Hamsters.js splits the start-field sequences evenly onto the available background threads[14]. 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 one to avoid this misfortune situation. Parallel.es also uses an even work splitting strategy but creates four-times as many tasks as background threads are available for a better workload balance in case of nonlinear problems. This strategy has shown to be a beneficial trade-off 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.

The results of Firefox do not indicate any advantage of using a thread pool over spawning new background threads for every task. It seems that creating background threads in Firefox is very inexpensive. However, the benchmarking results of Google Chrome v. 54 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, which is not using a thread pool at all, and Threads.js, whereby each benchmark run creates a new thread pool[15].

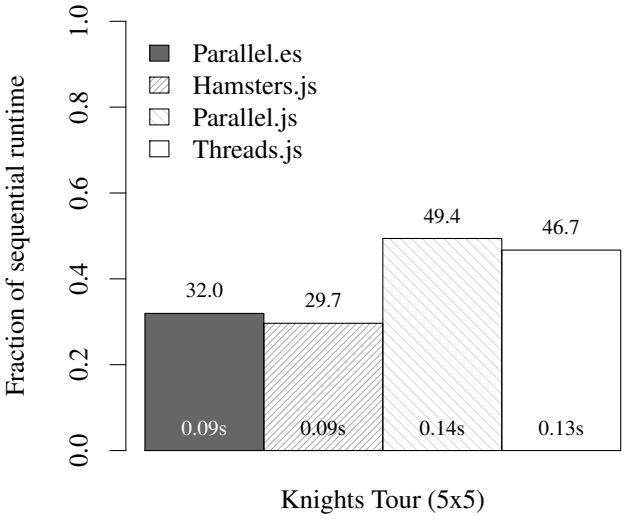

Figure 3. Knight Tour 5x5 Runtime Performance using Chrome

Mandelbrot The Mandelbrot problem is parallelized by computing a subset of the lines per task. The runtime of computing a single line depends upon the position of the line in the image. This nonlinearity is the reason why the Hamsters.js based implementation takes significantly longer. The even distribution 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 better performance of Threads.js is rooted insofar that Threads.js supports transferables [18, Section 2.7.4]. Transferables allow moving a memory range between threads instead of copying it. Hamsters.js also support transferables,

[14] The number of background threads used by Hamsters.js is manually set to the number of logical processors offered by the hardware because it does not determine the optimal number of background threads automatically.

[15] 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.

however, only if the input and output are transferable objects what is not the case for the Mandelbrot implementation.

Risk Profiling The Risk Profiling problem is parallelized by computing the outcome for a subset of investments in each task. However, this requires that each background thread run the Monte Carlo simulation to calculate the outcome of the planned investment. The overhead of computing the Monte Carlo simulation in each background thread is the reason why the achieved speedup is smaller than for the other problem instances.

It is important for the Risk Profiling implementation that the forecasts produced with the Monte Carlo simulation are reproducible. Therefore, sim.js [19] is used as random number generator, instead of `Math.random`, as it can be initialized with a seed. Hamsters.js is not part of this evaluation since it lacks support for importing functions from other modules and can therefore not use the sim.js library.

Parallel.es requires more time for the computation because of its work splitting strategy used that 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 nonlinearity results in some tasks completing earlier than others leaving computation resources unused. Enforcing a smaller task-size is not a solution for this problem as it leads to recomputing the Monte Carlo simulation for each investment reducing the performance even more. Therefore, Parallel.es has been configured to create as many tasks as background threads are available. Parallel.js and Threads.js can only use a task-size of one as the thread pool is not reused and therefore, the global context of the background thread can be used to store the simulation outcome. Manipulating the global context is not desired if shared background threads from a thread pool are used as it creates 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 sub-problems 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 sub-path 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 then be executed on any background thread to achieve a better work balance. The current implementation does not support this scenario and therefore, a limited set of start-field sequences are precomputed in the main thread and started as separate tasks. However, each task computes the number of tours sequentially without further dividing into subtasks.

An efficient implementation to support recursive tasks requires a communication channel between all background threads to allow to start a subtask on another, idle background thread without an additional roundtrip over the main thread. However, web workers only have a single communication channel between the thread that has started the web worker and the spawned web worker. Shared Web Workers [1, section 4.6.4] allow a worker to have multiple channels between various workers but are not even supported by the latest versions of all major browsers. Furthermore, the "run to completion" model of JavaScript can be problematic because a busy background thread does not process received messages until the current work has completed. Therefore, the background thread does not respond to received messages in a guaranteed timeframe.

Further research is required to determine how recursive tasks can be efficiently 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 elements in a single task. The results of Hamsters.js show that the latter is preliminary problematic if the problem is nonlinear, in which case a smaller task size helps to balance the workload. The approach used by Parallel.es is somewhat in between by creating four times as many tasks as the hardware provides logical processors. This approach showed to be a good default configuration but might require adjustment for specific problem instances, e.g. as it is the case for the Risk Profiling problem.

The evaluated systems differ more significantly in their APIs. Some of the evaluated systems only offer a low-level API while other provide a high-level API as well. If one or the other is to be preferred is very subjective and principally dependent on the specific problem and the programmer's preferences. However, other properties are more objective: Hamsters.js impedes code reuse and is unsuitable if the task function depends upon libraries since it does not permit to expose additional functions, e.g. functions imported from libraries, to task functions. Parallel.es is the only valid option for projects using typed languages because it is the only with a type-safe API. Threads.js supports transferables that allow moving the result of a computation instead of copying it, resulting in a better runtime for computations over large amounts of data as it is the case for Mandelbrot problem.

All the runtime systems have in common that they specify restrictions for task functions affecting the structuring of the code. This lack of freedom creates a clearly visible seam between background tasks and the rest of the application. Parallel.es reduces this seam to a minimum by transpiling the program code prior to execution allowing a far more complete set of JavaScript functions to be used as task functions.

7 Conclusion

Multithreading is only sparsely used in nowadays JavaScript applications because the standards for creating

background threads are platform-dependent and often enforce a messaging-based communication model. This paper presented a platform-independent type-safe API and runtime system that provides a seamless integration of background 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-stream-based operations with ease.

The evaluation shows that most existing runtime systems perform similarly concerning execution time when applied to the given set of problems. Even though Parallel.es is one of the faster runtime systems, its main advantage is the API and the seamless integration into existing code. The proposed API is close to APIs widely used by the JavaScript community facilitating fast learning and straightforward transitioning of existing code. Moreover, the type-safety of the API is clearly beneficial for projects using typed languages like TypeScript [16] or Flow [20]. The additional transpilation step allows a seamless integration of background tasks into existing code since a task function can reference read-only variables and functions from its outer scope without any additional doing of the programmer. The related work, on the contrary, defines more restrictions on task functions resulting in a clear break between background tasks and the rest of the program. The additional transpilation step further has the advantage that it generates source maps helping to identify errors from production and enabling a pleasant debugging experience, a feature not offered by any related work.

However, the evaluation also shows that the proposed system does not fit naturally 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 writing multithreaded applications in JavaScript enabling them to use the device's hardware efficiently.

8 Availability

The source code of the runtime system, the transpiler, the implemented problem instances, and the benchmark results are published on GitHub [21]–[24]. The libraries are released under the MIT license [25] and are published on NPM [26].

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

[11] E. Meijer, "Your mouse is a database", *Queue*, vol. 10, no. 3, 20:20–20:33, Mar. 2012, ISSN: 1542-7730. DOI: 10.1145/2168796.2169076. [Online]. Available: http://doi.acm.org/10.1145/2168796.2169076.

[12] "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).

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

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

[15] 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).

[16] Microsoft. (2016). Typescript, [Online]. Available: https://www.typescriptlang.org/.

[17] 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).

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

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

[20] Facebook Inc. (2016). Flow, [Online]. Available: https://flowtype.org/.
[21] M. Reiser. (2016). Parallel.es, [Online]. Available: https://github.com/MichaReiser/parallel.es (visited on 11/11/2016).
[22] ——, (2016). Parallel.es babel plugin, [Online]. Available: https://github.com/MichaReiser/babel-plugin-parallel-es (visited on 11/11/2016).
[23] ——, (2016). Parallel.es webpack plugin, [Online]. Available: https://github.com/MichaReiser/parallel-es-webpack-plugin (visited on 11/11/2016).
[24] ——, (2016). Parallel.es examples project, [Online]. Available: https://github.com/MichaReiser/parallel-es-example (visited on 11/11/2016).
[25] (2016). MIT License, [Online]. Available: https://opensource.org/licenses/mit-license.php.
[26] npm Inc. (2016). NPM, [Online]. Available: https://www.npmjs.com/.