Parallelize JavaScript-Computations with Ease

Micha Reiser

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

Abstract

TODO

1 Introduction

The role of JavaScript drastically changed in recent years. From an unpopular language used to add dynamic effects to web pages to one widely used with a strong and growing community. It emerged from a browser-only language to a general-purpose language used to write web-, desktop-, mobile-, and server-applications. The new use cases come along with new requirements demanding for more computational resources that can no longer be provided by a single threaded environment without negative effecting the user experience. The W3C responded to this new requirements with the Web Worker api draft in 2009 providing the infrastructure for creating multithreaded applications [1]. However, the web worker api requires that the code for web workers is located in a separate file resulting in separating code that belongs together and an increased complexity to the build procedure. Furthermore, the Communication between the main thread and web workers — and between web workers — is only messaging based. The messaging based programing model often does not fit well into the existing application design.

Besides that the programing model from Web Workers differs from the normal JavaScript programing model it is also not universal across platforms. The Web Worker draft is specified as Web api standard that is only implemented by browsers and is therefore not available in non-browser environments. NodeJS provides the child-process api that allows to spawn new child processes allowing to simulate a multithreaded application. However, this requires that messaging is used since the processes have no shared memory as it is the case for Web Workers. I believe that platform dependence together with the inherent complexity of the low level apis are the main reasons for the slow adaption and low exploitation of the available computation resources.

Mention negative impact on usability?

This paper presents parallel.es, a platform independent, type safe api and a runtime system for performing parallel computations using JavaScript. The main goals of the proposed api are a seamless integration into the existing code base and simplicity. The runtime system consists of two parts. At one hand the system provides a low level api allowing to run single functions on a background thread. At the other hand a reactive api is provided that is based upon the low level api and is inspired by the commonly used underscore [2] and lodash [3] 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. The low level api is designated for those requiring more flexibility or where the reactive programing model does not fit well to the problem to solve.

In addition to the runtime system a transpiler that automatically extracts functions executed in background threads into a designated file and hides limitations inherent from the programing model gap by rewriting the code. This includes features like debugging, access to variables from the outer scope or importing functionalities from other files — e.g. using functions from libraries.

The section 2 describes and compares existing work in the same area with the proposed solution. The section 3 describes the programing model of parallel tasks, followed by section 4 that explains the design and implementation of the runtime system and section 5 describing which limitations of the runtime system can be hidden by using the transpiler. The section 6 compares the presented runtime system with the existing work and is followed by the conclusion.

2 Related Work

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

2.1 Parallel.js

Parallel.js [4] has been initiated in 2013 and is the oldest of the evaluated libraries. The main goal is to provide a simple

api for multiprocessing in JavaScript. It provides an uniform api for the browser and NodeJS — Web Workers are used in the browser, child process [5] on NodeJS. Parallel.js provides a low level api for running a function in a background thread and a reactive api providing an automatic task scheduling for the provided data.

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

The latest published version on npm[1] spawns a separate worker for every element in the input array exhausting the thread limits of the browser resulting in a crashed tab. The latest version on GitHub has adjusted the behavior to only spawn as many workers as max concurrency supported by the hardware. Therefore, when Parallel.js is referenced, the latest version[2] from GitHub is meant.

2.2 Threads.js

Thread.js [6] aims to be a powerful but simple to use multi-threading library for NodeJS and the browser as well. Like Parallel.js, child processes are used in NodeJS and Web Workers in the browser.

The main difference of Threads.js is that it is messaging based. It therefore fits better to the programing model of Web Workers and child processes but requiring the programer to bridge the gap between the programing models.

2.3 Hamsters.js

Hamsters.js [7] is the library with the highest attention measured by the number of GitHub stars. It provides a low level api for running functions in a background thread. The background threads are managed in a global thread pool. In addition, various helper functionalities like array sorting, aggregating or caching are provided. The library has support for transferable objects and claims to support scheduling threads from a background-thread.

The main differences to Hamsters.js is that the proposed work offers a higher level api that simplifies parallelization.

```
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));
```

Fig. 1. Fibonacci Implementation

3 Programing Model

The programing model of Parallel.es motivates the programer to perform parallel computations asynchrony in background-tasks.

3.1 Background Task

A *background task* — further referred to as task — represents a single asynchronous operation executed on a designated thread and is implemented by a JavaScript function. Figure 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 [8, Section 18.3.18] allowing to register a `then` callback that is invoked with the result if the computation was successful and an error handler (**`catch`**) that is triggered otherwise. The callbacks are always executed on the ui-thread.

Some restrictions apply to the function executed in a task that are inherent from the fact that Web Workers have no shared memory[3]. These restrictions are following discussed.

3.2 Function Restrictions

The function may only refer to data that is explicitly passed to it by function arguments. Any references to variables from its closure — including other functions or imports — are prohibited. Data passed to a task is serialized using the structured clone algorithm [10, Section 2.9.4]. This implies that passed object are passed by value and therefore changes made in a task are not reflected in the ui-thread. The structured clone also restricts the objects that can be passed, e.g. DOM elements and functions are not cloneable.

The function context of a task function (**`this`**) is always set to **`undefined`**. This also applies to `ArrowFunctionExpressions` that are semantically treaded equally to `FunctionExpressions` — and therefore loose their additional semantics — if used as a task implementation.

The global context of a task differs from the one in the ui-thread. The browser api exposed in Web Workers is far more limited, e.g. the DOM cannot be accessed from a worker.

[1] NPM is a JavaScript Package Manager. The latest published version to date of Parallel.js is 0.2.1.

[2] Commit 2e4b36bf16e330abaaff213e772fcf4074fd866b

[3] This might change with the SharedMemory [9] standard that is currently a draft in stage 2.

```
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);
```

Fig. 2. Mandelbrot Implementation in parallel.es

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

Fig. 3. Synchronous Mandelbrot Implementation

However, this should not be a restriction as tasks are preliminary used to perform computing intensive operations.

A task differs from a normal function invocation as the computation is performed on a designated thread instead of in the main/ui thread where the computation blocks the event-loop resulting in delayed ui-updates and reduced frame rates. Tasks may be executed concurrently if the hardware has more than one logical processor. By design, all tasks are isolated from each other as each task uses its own local variables.

3.3 Reactive API

The proposed work further offers a reactive api that provides automatic work partitioning and result joining. The api is inspired by the commonly used underscore and lodash libraries. The goal of this api is to provide a well known and understood api that uses the available computation resources without any further actions by the programer. This simplifies it for the programmer to transform synchronous implementations into parallelized ones. The api is based on top of the low level api and therefore the same restrictions apply.

The api motivates the programer to define the computations as operations on a data flow. The defined operations on the data flow are chained together and are executed concurrently as multiple tasks. An implementation of the Mandelbrot computation using Paralle.es is shown in fig. 2. The implementation differs only slightly from the synchronous implementation shown in fig. 3. The task is defined from line 8 to 11 using the descriptive api. The `range` method defines the input as a range from 0 up to the image height. The object passed to `inEnvironment` is available to the function in the background task through an additional function parameter. The operation to apply to every element from the data stream is `map` with the passed mapping function `computeMandelbrotLine`. The computation is started using the `then` operation (line 12) that registers a callback to invoke in case the computation succeeds — an optional error callback can be defined as well. The then callback is executed in the ui-thread. The `computeMandelbrotLine` function (line 1) — that is executed on the background-thread — has access to the current array element and as well to the `environment` object that was passed in the `inEnvironment` call. No additional build step is required.

4 Runtime System

The runtime system of Parlallel.es consists of two parts: The public api in the ui-thread that forms the facade and acts as the master for the slaves and the slaves running in background-threads that execute the tasks. 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 uses by default a fair FIFO thread pool with a constant limit of background-threads initialized with the number of logical processors provided by the hardware[4]. The thread pool implementation is interchangeable.

4.1 Task Roundtrip

The steps needed to process a single task are shown in fig. 4. The application passes the function to execute in a background-thread together with the arguments to pass to the function to the facade that acts as master (1). The master creates a serialized representation of the function call to perform since messaging must be used between the master and slave as implied by the Web Worker programing model. The serialized representation consists of a unique id that identifies the function to call and the arguments to pass to the function. These information are passed to the slave (2). The slave performs a lookup in the function cache to retrieve the function with the given id (3). The function definition is requested from the master (4, 5), instantiated as a function and registered in the function cache (6) if the function is executed the first time on this slave — and therefore no registration exists in the function cache. 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 ui-thread to pass the result to the application (9).

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

[4]The api `navigator.hardwareConcurrency` provides the logical processor counts on newer browser. A default of 4 is assumed when the api is not available in the used browser

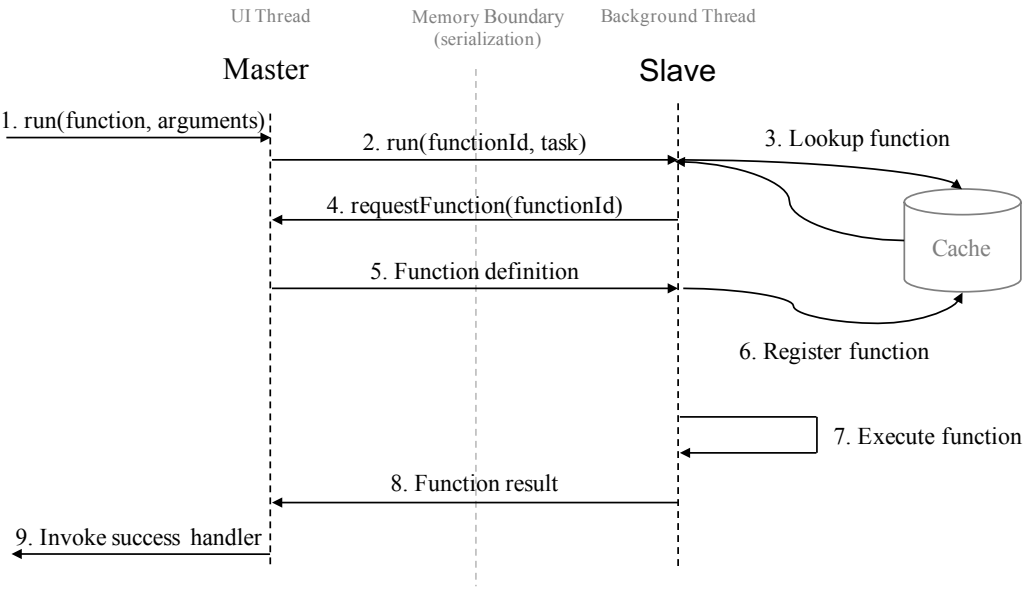

Fig. 4. Parallel.es Runtime System

4.2 Limitations

The current implementation supports the most essential features. However, the implementation currently is missing the support for asynchronous task operations and the NodeJS environment. There are no technical reasons for not supporting either of these features. Adding support for NodeJS requires using a structured clone polyfill as this is not supported by NodeJS.

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

5 Transpiler

The project further presents a JavaScript transpiler that hides some of the limitations for task-functions. It is based on top of WebPack[5] and Babel[6]. The transpiler extracts the task-functions into the file loaded by the slaves and registers the function in the slaves function-cache, potentially resulting in a better runtime performance. The main advantage of using the transpiler over the bare runtime system is that task-functions are allowed to refer to variables — including functions and imports — from the outer scope and therefore reducing the programing model gap and allowing a far larger set of functions.

Furthermore, the transpiler adds debugging support for task-functions — a distinct feature not offered by any related work. It generates source maps pointing back to the original source locations from where the task-functions have been extracted. Without these source maps, the only way to break inside of a task-function is by using the inflexible `debugger` statement. With the source maps pointing back to the original source break points can be set inside of the browsers developer tools giving a far better debugging experience to the developer[7].

5.1 Limitations

The transpiler only uses static lexical scoping to resolve the binding of a variable and determining if a function is referenced. Therefore, the transpiler only supports references to functions where lexical scoping is sufficient to identify that the referenced value is a function. Furthermore, only ES2015 imports are supported.

6 Evaluation

The evaluation focuses on computation intensive tasks or tasks over a large set of data that are expected to profit from parallelization. However, using the presented runtime system may also be beneficial for computations that cannot be parallelized by avoiding blocking the ui-thread and therefore, enhance the user experience. The evaluation compares the presented work with the solutions introduced in section 2 in respect to the performance results and simplicity of application.

[5] A JavaScript module bundler

[6] Framework for transforming JavaScriptCode

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

6.1 Performance Comparison

To compare the performance of the presented and related work the following set of problems is used:

1. Mandelbrot 10'000x10'000: Computation of a Mandelbrot for a given image size. Requires a relatively large amount of memory compared to the computational time needed.
2. Knight Tour: Computes the number of open tours from a given start field. Low memory usage but very high computational needs.
3. Riskprofiling: Simulates the development of the customers asset over a time period of 15 years by using a monte carlo simulation. The result of the simulation are used to determine the probability that the amount needed by an investment — e.g. buying a house after 10 years — is available at that point in time. Representation of a real world example [11].

The results of the performance measurement are shown in fig. 5. It shows the time needed by each implementation to compute the solution relative to the synchronous pendant. The code of the Parallel.es uses the transpiler to make imports available. The benchmark was performed using Firefox v50 on a Windows 10 computer with a 4-Core, 2.5 GHz Xeon E5-2609v2 processor[8]. The results show that the speedup achieved is approximately identical for all tested runtime systems — with some exceptions.

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

Not as expected, the time needed to spawn the background-tasks seems to have no noticeable impact on the test results. Hamsters.js and Parallel.es use a single thread pool instance with pre spawned background-tasks. Parallel.js and Threads.js create new background-tasks for each test run.

The test case of the 6x6 knight tour only shows significant differences for the Hamsters.js runtime system. The difference is rooted in the used distribution strategy of the start fields onto the tasks. Hamsters.js creates as many task as background-threads are available and evenly distributes the start-field-sequences across the tasks. However, some start-field-sequences require more time to compute then others, resulting in unused computation resources when other tasks complete early. Parallel.js and Threads.js always use a task size of 1. Parallel.es has been configured to use task size of 1 for the knight tour problem to avoid this situation.

Mandelbrot The computation of the Mandelbrot is parallelized by distributing the computation of single lines onto the background-tasks. However, the computation time needed to compute a single line depends upon the position of the line in the image — its a non-linear problem. This is the reason why the Hamsters.js based computations takes significantly longer, because of its even distribution of the work onto the background-threads. Tasks computing the center of the Mandelbrot take longer than the ones at the top and bottom shorter. Parallel.es also uses an even distribution but by default creates four-times as many tasks as background-threads are available. This has shown to be a good balance between having a large enough set of items to process by each task reducing the overhead for the task processing while still leaving some room to compensate for non-linear problems.

The performance gain of Threads.js compared to the other runtime system is rooted in the fact that Threads.js supports transferables [13, Section 2.7.4]. Transferables allow to move a memory range between threads instead of copying it. Hamsters.js also support transferables, however, only if the input and output are objects supported by the transferable standard what is not the case for the Mandelbrot implementation.

Riskprofiling The risk profiling uses sim.js [14] in the monte carlo simulation as random number generator that supports seed numbers. This is needed to achieve reproducible forecasts. Hamsters.js lacks support for importing functions from external files and is therefore not part of this evaluation. The problem is parallelized by computing the outcome for a sub set of investments in each task. However, this requires that each background-thread simulates the development of the customer assets over the years once to get the data needed to calculate the outcome for an investment. Only a smaller speedup can be achieved by parallelizing this problem since the simulation requires significant more time to compute than for calculating the outcome of the investments, only a lower speedup can be achieved.

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 for an investment is non linear. It depends on the year in which the investment takes place, the later this is the case, the more values have to be computed. This non linear computation time results in some tasks completing earlier than others leaving computation resources unused. Reducing the task-size to 1 is no solution in this case as this result in recomputing the asset development for each investment reducing the performance even more. Parallel.js and Threads.js can only use a task size of 1 as the thread pool is not reused and therefore side effects in the background-

[8] The results of the benchmarks only differ slightly when using Chrome or Safari. This is mostly rooted in the case that the optimization performed by the JIT compiler differs between browsers. However, the performance of parallel computations drops significantly in Microsoft Edge if the runtime system uses `new Function` or `eval` to create dynamic function instances — that is the case for Hamsters.js and Threads.js. This observation has been reported and is confirmed by Microsoft [12].

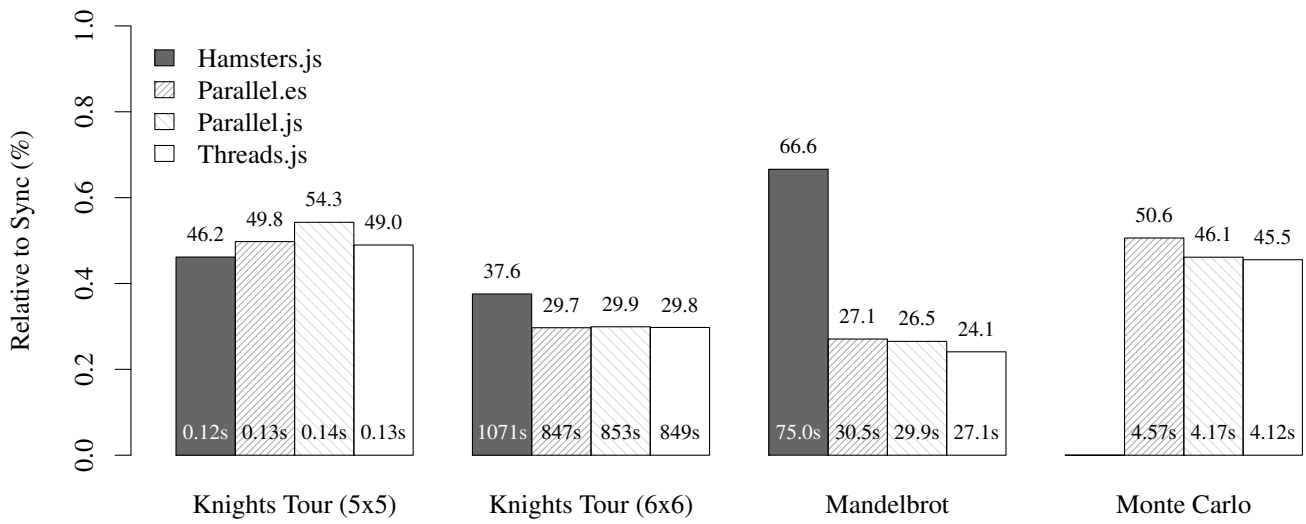

Fig. 5. Runtime Performance of Parallelization Problems Relative to Synchronously Execution

tasks can be used to temporary store the simulation outcome in a global variable. This is not desired in a shared thread pool as it results in memory leaks.

6.2 Usability

The usability is evaluated by comparing the implementations for the Mandelbrot problem. The synchronous implementation is shown in fig. 3. The actual implementation of the per line computation is omitted for brevity as it is almost identical for all libraries. The preliminary focus is on type safety, readability, compactness and build system integration. However, some of the result might be subjective and represent the opinion of the author. The source code for all examples is available in a designated GitHub project [15].

6.2.1 Parallel.js

The Mandelbrot implementation in Parallel.js is shown in fig. 6. The first argument of the `Parallel` constructor (line 11) is the data to process. The second — optional — argument is an options-object affecting the behavior of Parallel.js. The value of the `env` property is exposed as `global.env` in the background thread (line 14). This use of the global, undeclared variable `global` to expose the data is problematic as it breaks static scoping and requires additional care if typed languages are used. One way to work around the problem is to manually declare the `global` variable every time it is used or globally in a declaration file and annotating its type. The global declaration is problematic as the type may differ depending on the problem leaving only the option to disable type checking for the variable at all by annotating a special opt-out type like `any` in TypeScript. Even with a per use declaration, type safety is no longer guaranteed as the type checker does not have the ability to check if the passed object in the environment property matches the structure of the declared `global` variable. The global variable also hinders

```
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  const parallelOptions = { env: options };
10
11 new Parallel(lines, parallelOptions)
12   .require(computeMandelbrotLine)
13   .map(
14     line => computeMandelbrotLine(line, global.env)
15   )
16   .then(result => console.log(result));
```

Fig. 6. Mandelbrot Implementation using Parallel.js

that the same function can be called from the ui-thread as the variable in this case is not initialized.

Parallel.js provides a runtime feature that allows to include external files and other functions inside the started background-thread using the `require` function (line 12) without the need for static code transpiling as it is needed by Parallel.es.

Parallel.js also offers a reactive api resulting in an implementation that differs only slightly from the synchronous implementation. The function passed to the `map` operation (line 13) is called for every element in the input array and produces the elements in the output array.

6.2.2 Threads.js

The Mandelbrot implementation using threads.js is shown in fig. 7. Threads.js does not use thread pools by default. A thread pool needs to be created manually if desired (line 9). The function to execute in the background worker is defined using the `run` method on the pool (line 10). A new task is created every time `send` is invoked to process the passed in element. The result can be retrieved by waiting

```
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 taskPromises = [];
9  const pool = new Pool();
10 pool.run(computeMandelbrotLine);
11
12 const lines = _.range(options);
13 for (const y of lines) {
14   const taskPromise = pool.send({ y, options})
15                           .promise());
16   taskPromises.push(taskPromise);
17 }
18
19 Promise.all(taskPromises)
20   .then(result => console.log(result));
```

Fig. 7. Mandelbrot Implementation using threads.js

until the promise of the task resolves.

The final result over all tasks can be retrieved by waiting until all promises of the tasks are resolved.

6.2.3 Hamsters.js

The implementation of the Mandelbrot is shown fig. 8. A background-task is started using `hamsters.run(...)` (line 20). The first argument are the parameters that are passed to the task-function. The task-function is defined by the second argument. The third argument is the callback that is invoked when the operation has completed. The passed parameters are available on the global object in the `params` property(line 2) in the background task. This is less problematic for type safety than it is the case for Parallel.js since some languages allow to define the type of the this **`this`** object.

Hamsters.js passes a sub array to each task leaving it to the background-operation to iterate over the sub-array elements. The data returned by the background task need to be written into the `rtn.data` (line 10) array data is also defined globally. This requires additional care if typed languages are used and reduces code sharing between background-task logic and ui-thread logic.

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

Fig. 8. Mandelbrot Implementation using Hamsters.js

7 Conclusion

References

[1] *Web Worker*, W3C Std., sep 2015. [Online]. Available: https://www.w3.org/TR/workers/

[2] DocumentCloud. (2016) underscore.js. [Online]. Available: http://underscorejs.org/

[3] JS Foundation. (2016) lodash. [Online]. Available: https://github.com/lodash/lodash

[4] A. Savitzky and S. Mayr. (2016) Parallel.js. [Online]. Available: https://parallel.js.org/

[5] Node.js Foundation. Child process.

[6] A. Wermke. (2016) threads.js. [Online]. Available: https://github.com/andywer/threads.js

[7] A. Smith. (2016) Hamsters.js. [Online]. Available: http://www.hamsters.io/

[8] *ECMAScript® 2015 Language Specification*, Ecma International Std., Rev. 6th Edition, Jun. 2015. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/

[9] *ECMAScript Shared Memory and Atomics*, Ecma TC39 Std., Nov. 2016. [Online]. Available: http://tc39.github.io/ecmascript_sharedmem/shmem.html

[10] *HTML*, WHATWG Std., Nov. 2016. [Online]. Available: https://html.spec.whatwg.org/multipage/index.html

[11] K&W Software AG, "Risikoprofilierung von anlagekunden," Apr. 2016.

[12] M. Reiser, "new function in web workers significantly slower," Nov. 2016. [Online]. Available: https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/9779930/

[13] *HTML5*, W3C Std., oct 2014. [Online]. Available: https://www.w3.org/TR/html5/

[14] M. Varshney. (2011) Sim.js – discrete event simulation in javascript. [Online]. Available: http://www.simjs.com/

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

Todo list

Mention negative impact on usability? 1