

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 negatively affecting the user experience. This becomes even more severe with the spreading use of mobile devices in the world wide web. The CPUs of nowadays mobile devices have — compared to desktop computers — a lower clock rate that is recompensed by the higher number of CPU-cores. However, this requires that applications make use of the available computation resources.

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

Unfortunately, implementing multi-threaded applications targeting different environments in JavaScript is non trivial

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 to spawn sub process using the child-process API [2]. JavaScript applications running on the JVM can use RingoJS [3] that enables multi-threaded JavaScript — including shared memory and all problems shared state brings with it. Therefore, an application needs to explicitly be adopted to the standard provided by its supported environments.

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

Some differences between the programming model cannot be hidden just by the runtime system. The gap is further reduced by the presented transpiler that rewrites the code to reduce the programming model gap to a minimum. The transpiler adds debugging support, allows function to import functionalities from libraries, calling other functions or accessing read only variables from the outer closure. However, the use of the transpiler remains optional.

The first section of this paper describes the related work and compares it to the presented solution. The following section 3 describes the programming model of parallel tasks and is followed by the section 4 that explains the functionality of

the runtime system. The section 5 explains which programming model differences can be hidden when 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 main difference of the the presented system to the related work is that it offers the same debugging functionalities as developers are used to when working with synchronous code.

2.1 Hamsters.js

Hamsters.js [6] is the library with the highest attention measured by the number of GitHub stars. It provides a low level api for running functions in a background thread. The background threads are managed in a global thread pool. 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.

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 an uniform api for the browser and NodeJS — web workers are used in the browser, child processes [2] on NodeJS. Parallel.js provides a low level api for running a function in a background thread and a reactive api providing an automatic task scheduling for the provided data.

This work differs from Parallel.js in the fact that a thread pool is used that reuses already created background-threads and performs queuing if needed. Parallel.js has kind of a thread pool approach but only on a per-operation (e.g. map or filter) basis. The second important difference is that Parallel.js awaits the sub results of the proceeding operation before continuing with the next operation if multiple operations are chained together, e.g. the reduce step 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 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 npm¹ spawns a separate worker for every element in the input array exhausting the thread limits of the browser. The latest version on GitHub has adjusted this behavior to only spawn as many workers as

logical processors are available by the hardware. Therefore, when Parallel.js is referenced, the latest version² from GitHub is meant.

2.3 Threads.js

Thread.js [8] aims to be a powerful but simple to use multi-threading library for NodeJS and the browser 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 programming model used by web workers and child processes. However, this requires that the programmer bridges the gap between the programming model used inside the application and the messaging based programming model of Threads.js

3 Programming Model

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

3.1 Background Task

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

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

Listing 3.1. Fibonacci Implementation

The valid task-functions are only a subset of all JavaScript functions since not all JavaScript features are allowed to be used. This is rooted in the fact that neither web workers nor child processes have a shared memory³ that can

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

²Commit 2e4b36bf16e330abaaff213e772fcf4074fd866b

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

be used to store data accessed by different threads. It is therefore disallowed that a task-function references any variables or functions — including imports from external files — from the outer scope⁴. Data needed by a task function needs to be explicitly passed when the task is started and is serialized using the structured clone algorithm [11, Section 2.9.4]. This enforces a pass-by-value semantic and requires that the passed data is serializable according to the structured clone standard, e.g. Functions, DOM-Elements and errors are not cloneable. The same applies for the result returned by the task.

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

Parallel tasks are isolated from one another since each task operates on its own copy of the variables and every thread only executes on task at the time. However, the global state of a thread is shared among all tasks executed on this thread instances. Tasks can access and modify the global state. This is strongly discouraged as changes are only thread local — and therefore not replicated between threads — and may introduce memory leaks.

3.2 Reactive API

The runtime system further offers a reactive api. This api is inspired by the commonly used underscore and lodash libraries and motivates the programmer to define the computations as operations on data streams. The runtime system takes care of splitting the work into several sub tasks and aggregating the sub results to the end result. The created tasks perform all operations on a subset of the input values and are potentially executed concurrently if enough resources are available. The goal of this api is to provide a well known and understood api that uses the available computation resources without any further actions needed by the programmer. This simplifies transforming synchronous code into parallelized. 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 3.3. The implementation differs only slightly from the synchronous implementation shown in listing 3.2. 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 was passed in the `inEnvironment` call. The computation is started using the `then` method (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 master-thread. No additional build step is required.

```
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);
```

Listing 3.2. Synchronous Mandelbrot Implementation

```
1 function computeMandelbrotLine(y, environment) {
2   const arraySize = 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 3.3. 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 uses a thread pool that queues task in the case no idle background-thread is available. The default thread pool uses a FIFO queue and the number of background-threads is limited to the number of logical processors provided by the hardware⁵. The next section describes how the runtime system processes a single task.

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

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

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 master (1). The created task is enqueued in the thread pool and executed on the first slave that get 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 retrieve the function with the given id (3). If the function is executed the first time on this slave, then the function definition is requested from the master (4, 5), instantiated as a function and registered in the function cache (6). Otherwise, the deserialized function is returned by 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 main-thread to pass the result to the application (9).

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

4.2 Limitations

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

A further limitation is that a task can not start other tasks. An efficient implementation to support recursive tasks requires a communication channel between all web workers that allows to start a task from a background-thread on another, idle background-thread without an additional roundtrip over the main-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 absence of a shared memory that is accessible by all threads results in various limitations applying to task-functions. These limitations are described in the previous section and have in common, that no symbols from the outer scope are allowed to be referenced from inside of a task-function. This impedes code reuse and hinders a seamless integration into existing code. The presented transpiler hides the limitations of the runtime system from the programmer by rewriting the task-start and task-functions.

The transpiler is based on top of webpack⁶ [12] and Babel⁷ [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 slaves function-cache.

The Mandelbrot implementation referencing the `imageWidth` variable (line 12, 14) and the function `computePixel` (line 4) from the outer scope is shown in listing 5.1. The transpiler rewrites the code and makes the `imageWidth` variable and the `computePixel` function available in the task-function. The transpiled code is shown in listing 5.2. The transpiler creates a new function that returns an object containing the values of all referenced variables and inserts it above the definition of the task-function (line 9). This function is used to extract the current values of the referenced variables in the master-thread. The object returned by the function is made available inside the task by passing it to the `inEnvironment` (line 16). The transpiler further replaces the task-definition by a unique function-id (line 17) that the slave uses to resolve the function from the function-cache⁸.

```
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 Outer Scope

The code inserted by the transpiler into the script run by the slave is shown in listing 5.3. The transpiler injects the code of the task-function (line 7) and the referenced `computePixel` function (line 2)⁹. It further generates an *entry-function* (line 15) that initializes the `imageWidth` variable (line 1) with the value stored in the environment variable — that contains the latest value from the master-thread — and calls the actual task-function. The entry function is registered in the function cache (line 25) using the same function id as used in the master-thread. This allows the slave

⁶A JavaScript module bundler

⁷Framework for transforming JavaScriptCode

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

⁹The transpiler creates an immediately invoked function expression to isolate the task-functions of different files and avoiding naming clashes

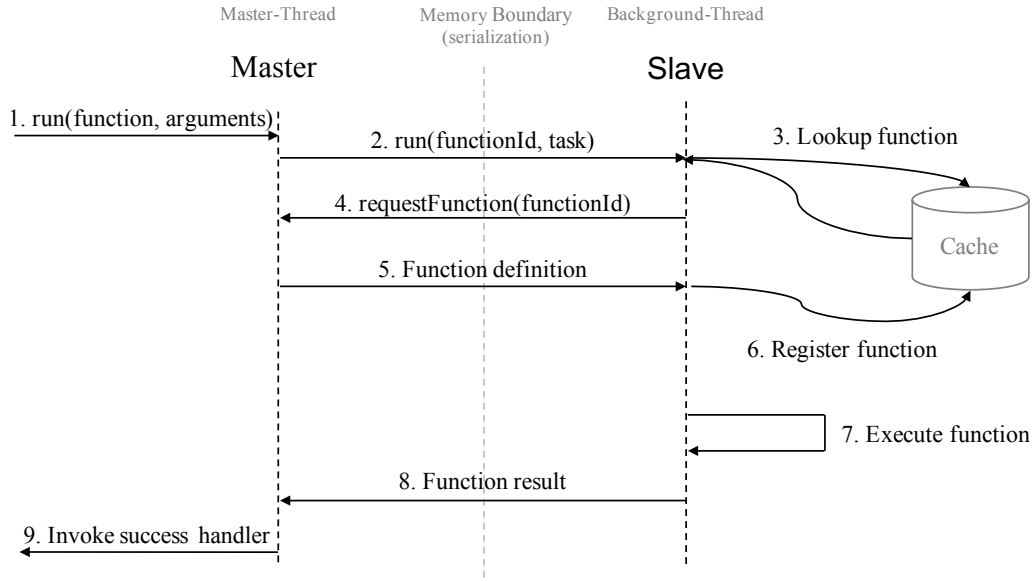


Figure 1. Parallel.es Runtime System

```

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

```

Listing 5.2. Transpiled Mandelbrot Implementation

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-functions — and transitive functions. This enables a true debugging experience that allows to set break points inside of the browser-developer tools¹⁰. Without the 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 provided by any of the related work.

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

```

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

```

Listing 5.3. Generated Slave-Code for Transpiled Mandelbrot Implementation

5.1 Limitations

The current implementation of the transpiler supports the reactive api, however, there are no technical limitations that prevent supporting the low-level api. The transpiler is limited to only allow references to constant-variables — either the variable is declared as **const** or the value does not change after the initialization. Furthermore, symbols referring to functions or imports need to be resolvable using lexical scoping, e.g. its not supported to call a function that has been passed as function-argument, hence the transpiler does not use any data flow analysis for symbol resolution. Finally, the transpiler only supports ES6 modules and import statements.

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 in respect to performance and applicability to the following set of problems:

- 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.
- Knight Tour: Computes the number of open tours from a given start field. Low memory usage but very high computational needs.
- Riskprofiling: Simulates the development of customer assets over a time period of 15 years by using a Monte Carlo simulation. The simulation used to determine the probability that the amount needed by an investment — e.g. buying a house after 10 years — is available at this point in time. Representation of a real world problem [14].

6.1 Applicability

The applicability is mainly evaluated by comparing the Mandelbrot implementations. The synchronous implementation of the Mandelbrot has been introduced in section 3 (listing 3.2). 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 result might be subjective and represent the opinion of the author. The source code for all examples is available on GitHub [15].

Parallel.js The Mandelbrot implementation using the reactive api of Parallel.js is shown in listing 6.1. It differs only slightly from the synchronous implementation. A parallel task is created using the `Parallel` constructor (line 10). The first constructor-argument is the data to process, the second — optional — is an options-object affecting the task execution where the value of the `env` property is exposed as `global.env` in the background thread (line 13). The 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 global, undeclared variable `global` to expose additional data in the task is problematic since it breaks static scoping and requires additional care if typed languages are used. One way to work around the problem is to declare the `global` variable in every module it is used or globally in a declaration file. In both cases, no specific type can be annotated for the environment property since its type depends upon the concrete problem. This only leaves the option to disable type checking for the environment property at all by annotating a special opt-out type like `any` in TypeScript.

```
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

Declaring the variable on a module basis or globally has the undesired side effect that the type checker no longer complains if the variable is actually undeclared if used outside of a task function. The global variable also hinders that the same function can be called from the main-thread. In this case, the `global` variable is undeclared and the function call fails.

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

Overall, the api of Parallel.js and Parallel.es are very similar. However, the api of the latter is type safe.

Threads.js The Mandelbrot implementation using Threads.js is shown in listing 6.2. Threads.js can be used with or without thread pools. A thread pool needs to be created manually if one is desired (line 8). The function to run in the background-thread is defined using the `pool.run` method (line 9). A new task for this function is created every time `send` (line 15) is called where the passed in value is used as argument to call the function. The result of a single task can be retrieved by registering the `done` handler (line 16). The sub results of the tasks have to be joined manually. The thread pool offers the event `finished` (line 19) that is called when all tasks of this pool have been completed¹¹.

The Threads.js api offers a clean, flexible, messaging-based api to run single tasks in a background-thread but does not offer a higher level abstraction for common operations. This lack of higher level api complicates the migration of existing code since the programmer needs to partition the work into different tasks and joining the sub-results. Furthermore, the programmer is responsible to manage the lifetime of the thread pool.

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

1. Object that is passed to the task-function. The special property `array` defines the input data. The object is

¹¹The finished event cannot be used if other tasks are executed on the thread pool.

```

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

exposed as params in the task-function.

2. The task-function that is executed in the background-thread.
3. Callback function that is invoked when the operation has completed.
4. Into how many tasks should the data be partitioned.
5. Defines if the sub-results of the tasks are automatically joined (**true**) into the end result.

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

```

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

```

Listing 6.3. Mandelbrot Implementation using Hamsters.js

The api of Hamsters.js is a mixture of a low- and high-level api: On one hand it offers only a single `run` method, on the other advanced features like work partitioning, result joining, and even features like result caching. I believe that exposing all these features in a single method makes the api hard to use. It is difficult to remember the correct ordering or

the semantics of the arguments. Even though the offered 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 to not offer type safety because of the global, undeclared params and `rtn` variables in the task-function. These global variables also hinder code reuse because they are undeclared if the function is not used as a task-function.

6.2 Performance Comparison

The benchmark results are shown in fig. 2. It shows the time needed by each implementation to compute the solution relative to the synchronous pendant. The test setup uses Firefox 50 on a Windows 10 computer with a 4-Core, 2.5 GHz Xeon E5-2609v2 processor¹². The results are explained in the rest of this section.

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

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

The usage of a thread pool does not have the expected significance. In the results using Firefox the effect is not visible at all. However, the benchmarking results of Google Chrome shown in fig. 3 give evidence that a thread pool might actually be beneficial for very short running tasks. Thus, Hamsters.js and Parallel.es achieve slightly better results than Parallel.js, that is not using a thread pool at all, and Threads.js where manually a new thread pool is created for each execution¹³.

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

¹²The results of the benchmarks only differ slightly when using Chrome or Safari. This is mostly rooted in the cause 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 [16].

¹³This is not strictly necessary. However, it is needed to store the simulation result for the risk profiling problem.

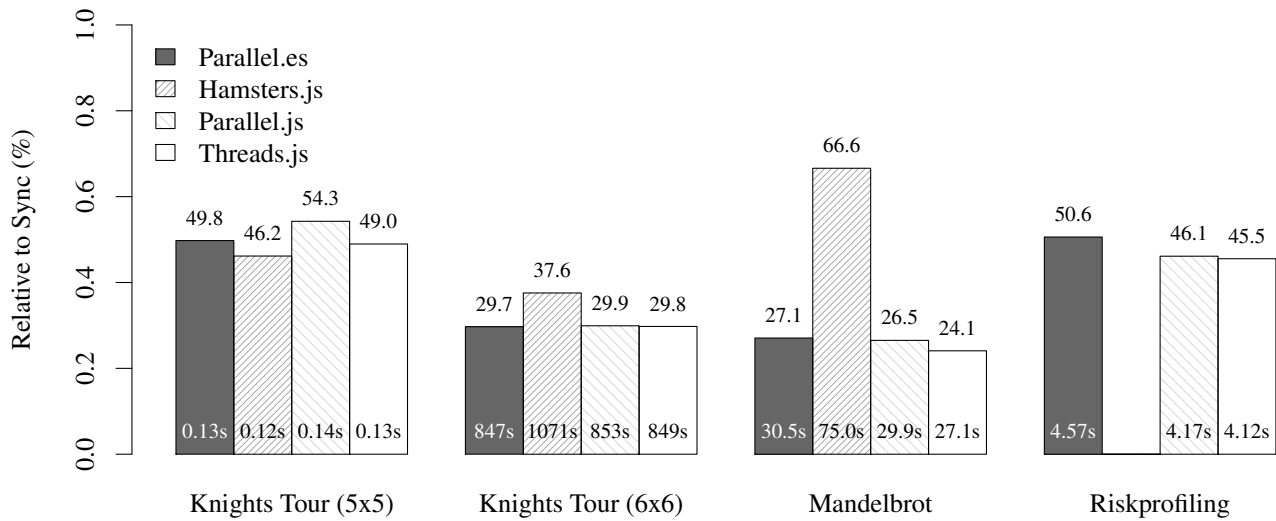


Figure 2. Runtime Performance of Parallelization Problems Relative to Synchronously Execution

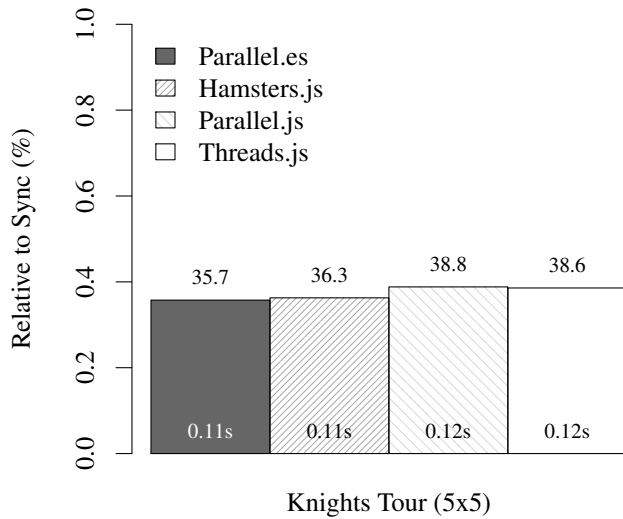


Figure 3. Knight Tour 5x5 Runtime Performance using Chrome

Mandelbrot The Mandelbrot problem is parallelized by computing a subset of the lines per task. However, the computation time needed to compute a single line depends upon the position of the line in the image — it is a non-linear problem. This is the reason why the Hamsters.js based implementation 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 or bottom of the field. 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 [17, 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 implementation uses sim.js [18] in the monte carlo simulation as random number generator that supports seed numbers. This is needed to achieve reproducible forecasts. Hamsters.js lacks support for importing functions from external files and is therefore not part of this evaluation. The problem is parallelized by computing the outcome for a sub set of investments in each task. However, this requires that each background-thread has to perform the monte carlo simulation to get the data needed to calculate the outcome for an investment. Therefore, only a smaller speedup can be achieved by parallelizing this problem since the simulation requires significant more time to compute than for calculating the outcome of the investments.

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. Specifying a task-size of one is not a solution for this problem 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 one as the thread pool is not reused and therefore side effects

in the background-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.

7 Conclusion

References

- [1] “Web Worker”, W3, Tech. Rep., Sep. 2015. [Online]. Available: <https://www.w3.org/TR/workers/>.
- [2] Node.js Foundation. (2016). Child process, [Online]. Available: https://nodejs.org/api/child_process.html (visited on 11/13/2016).
- [3] Ringo. (2016). Ringojs, [Online]. Available: <http://ringojs.org/> (visited on 11/21/2016).
- [4] DocumentCloud. (2016). Underscore.js, [Online]. Available: <http://underscorejs.org/> (visited on 11/11/2016).
- [5] JS Foundation. (2016). Lodash, [Online]. Available: <https://github.com/lodash/lodash> (visited on 11/11/2016).
- [6] A. Smith. (2016). Hamsters.js, [Online]. Available: <http://www.hamsters.io/> (visited on 11/12/2016).
- [7] A. Savitzky and S. Mayr. (2016). Parallel.js, [Online]. Available: <https://parallel.js.org/> (visited on 11/11/2016).
- [8] A. Wermke. (2016). Threads.js, [Online]. Available: <https://github.com/andywer/threads.js> (visited on 11/11/2016).
- [9] “ECMAScript® 2015 Language Specification”, Ecma International, Tech. Rep., Jun. 2015. [Online]. Available: <http://www.ecma-international.org/ecma-262/6.0/> (visited on 11/14/2016).
- [10] “ECMAScript Shared Memory and Atomics”, Ecma International, Tech. Rep., Nov. 2016. [Online]. Available: http://tc39.github.io/ecmascript_sharedmem/shmem.html (visited on 11/13/2016).
- [11] “HTML”, WHATWG, Tech. Rep., Nov. 2016. [Online]. Available: <https://html.spec.whatwg.org/multipage/index.html> (visited on 11/13/2016).
- [12] JS Foundation. (2016). Webpack, [Online]. Available: <https://webpack.js.org> (visited on 11/11/2016).
- [13] Babel. (2016). Babel, [Online]. Available: <http://babeljs.io/> (visited on 11/25/2016).
- [14] J. Zaugg. (Jun. 1, 2015). Risikoprofilierung des Anlagekunden, Vom Papierfragebogen zur digitalen Visualisierung, ZHAW, [Online]. Available: <http://pd.zhaw.ch/publikation/upload/208631.pdf> (visited on 11/20/2016).
- [15] M. Reiser. (2016). Parallel.es examples project, [Online]. Available: <https://github.com/MichaReiser/parallel-es-example> (visited on 11/11/2016).
- [16] —, (Nov. 2016). New function in web workers significantly slower, [Online]. Available: <https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/9779930/> (visited on 11/20/2016).
- [17] “HTML5”, W3, Tech. Rep., Oct. 2014. [Online]. Available: <https://www.w3.org/TR/html5/> (visited on 11/11/2016).
- [18] M. Varshney. (2016). SIM.JS - Discrete Event Simulation in JavaScript, [Online]. Available: <http://www.simjs.com/> (visited on 11/20/2016).

Todo list