# 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. This becomes even more sever with the spreading use of mobile devices in the world wide web. The cpus of nowadays mobil devices have — compared to desktop computers — a lower clock rate that is recompensed by the higher number of cpu-cores. However, this requires that application 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 programing 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 programing 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 programing 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 programing model does not fit well to the problem to solve.

Some differences between the programing 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 programing 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 secton of this paper describes the related work and compares it to the presented solution. The following section 3 describes the programing model of parallel tasks and is followed by the section 4 that explains the functionality of the runtime system. The section 5 explains which programing 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 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. 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 version[2] from GitHub is meant.

```
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

### 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 programing model used by web workers and child processes. However, this requires that the programer bridges the gap between the programing model used inside the application and the messaging based programing model of Threads.js

## 3 Programing Model

The programing model of Parallel.es motivates the programer 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. 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 [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.

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 memory[3] that can 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

[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 [10] standard that is currently a draft in stage 2.

the outer scope[4]. Data needed by a task function needs to be explicitly passed when the task is started and is serialized using the structured clone algorithm [11, Section 2.9.4]. This enforces a pass-by-value semantic and requires that the passed data 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 it. 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 tasks 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.

```
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

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

const range = _.range(options.imageHeight);
const result = range.map(
  y => computeMandelbrotLine(y, options)
);
console.log(result);
```

Fig. 3. Synchronous Mandelbrot Implementation

## 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 programer 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 programer. This simplifies transforming synchronous code into parallelized. The reactive api uses the infrastructure provided by the low level api. Therefore, the same programing model applies.

An implementation of the Mandelbrot computation using the reactive api of Paralle.es is shown in fig. 2. The implementation differs only slightly from the synchronous implementation shown in fig. 3. The `range` method (line 8) defines the data stream to process. It creates a data stream containing the values from 0 up to the image height. The input stream is transformed by mapping (`map` on line 11) each element from the input stream to an output element that is computed by the `computeMandelbrotLine` function (line 1). The `computeMandelbrotLine` function — that is executed in a background-thread — has access to the current array element and as well to the `environment` object that 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.

## 4 Runtime System

The runtime system of Parlallel.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 hardware[5]. The next section describes how the runtime system processes a single task.

## 4.1 Task Roundtrip

The steps needed to process a single task are shown in fig. 4. 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 identifiying the function to call and the arguments to pass to the function — to the slave (2). The slave performs a lookup

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

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

in the function cache to retrieve the function with the given id (3). If the function is executed the first time on this slave, than 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 it is 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 project further presents a JavaScript transpiler that hides some of the limitations for task-functions. It is based on top of WebPack[6] and Babel[7]. 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[8].

[6]A JavaScript module bundler

[7]Framework for transforming JavaScriptCode

### 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.

### 6.1 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 [12].

**Parallel.js** The Mandelbrot implementation in Parallel.js is shown in fig. 5. The first argument of the `Parallel` constructor (line 10) 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 13). 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

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

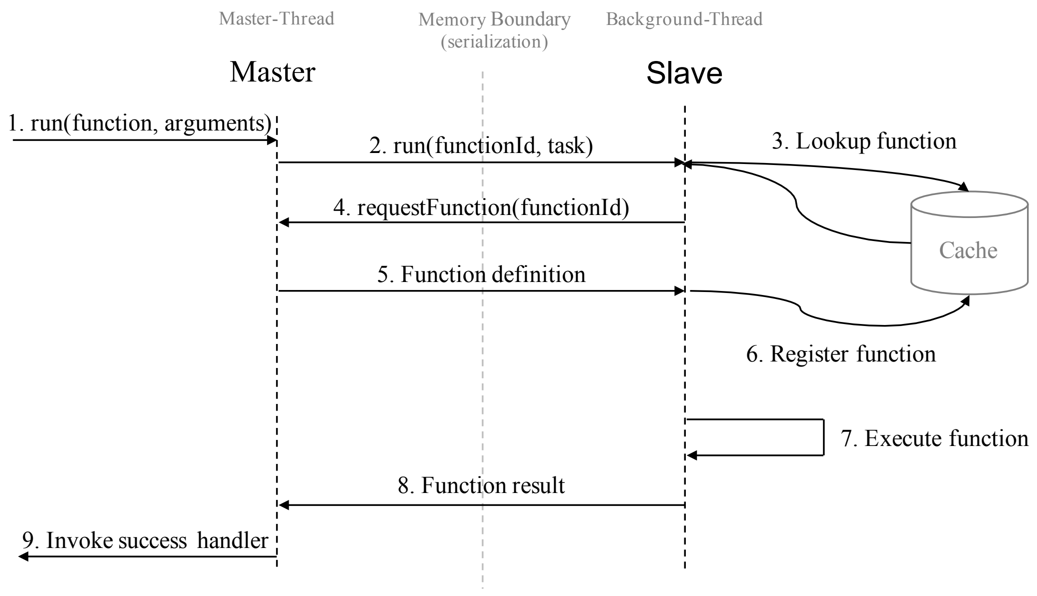

Fig. 4. Parallel.es Runtime System

```
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, parallelOptions)
11   .require(computeMandelbrotLine)
12   .map(
13     line => computeMandelbrotLine(line, {env: options})
14   )
15   .then(result => console.log(result));
```

Fig. 5. Mandelbrot Implementation using Parallel.js

```
1  function computeMandelbrotLine({ y, options}, done) {
2    const arraySize = options.imageWidth * 4;
3      const line = new Uint8ClampedArray(arraySize);
4      // compute...
5      done.transfer(line, [line.buffer]);
6  }
7
8  const 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. 6. Mandelbrot Implementation using threads.js

the declared `global` variable. The global variable also hinders 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 11) 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 12) is called for every element in the input array and produces the elements in the output array.

**Threads.js** The Mandelbrot implementation using threads.js is shown in fig. 6. 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 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.

**Hamsters.js** The implementation of the Mandelbrot is shown fig. 7. 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

```
function computeMandelbrotLine () {
  const options = params.options;
  const width = options.imageWidth;
  const input = params.array;

  for (let i = 0; i < input; ++i) {
    const y = input[i];
    const line = new Uint8ClampedArray(width * 4);
    // compute...
    rtn.data.push(line);
  }
}

const range = _.range(options.imageHeight);
const params = {
  array: range,
  options
};

hamsters.run(
  params,
  computeMandelbrotLine,
  result => console.log(result),
  hamsters.maxThreads,
  true);
```

Fig. 7. Mandelbrot Implementation using Hamsters.js

used and reduces code sharing between background-task logic and ui-thread logic.

### 6.2 Performance Comparison

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

- 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 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 [13].

The results of the performance measurement are shown in fig. 8. 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[9]. The results show that the speedup achieved is approximately identical for all tested runtime systems — with some exceptions.

[9]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 [14].

**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 [15, 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 [16] 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.

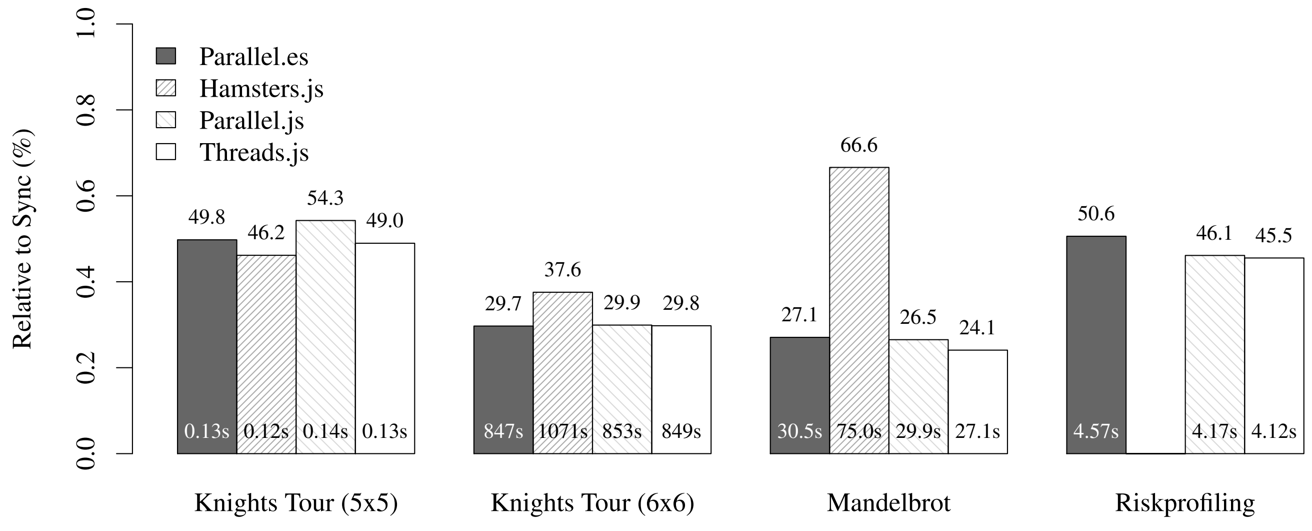

Fig. 8. Runtime Performance of Parallelization Problems Relative to Synchronously Execution

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

[2] Node.js Foundation. Child process.

[3] Ringo. ringojs. [Online]. Available: http://ringojs.org/

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

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

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

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

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

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

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

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

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

[13] K&W Software AG, “Risikoprofilierung von anlagekunden,” Apr. 2016.

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

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

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

**Todo list**