Parallel.es

Parallelize your JavaScript Applications with Ease

Micha Reiser

February 17, 2017

HSR

Overview

Motivation

Initial Position

Parallel.es

- Overview
- Implementation

Use Case

Conclusion

Motivation

Motivation for Parallelization

- Performance

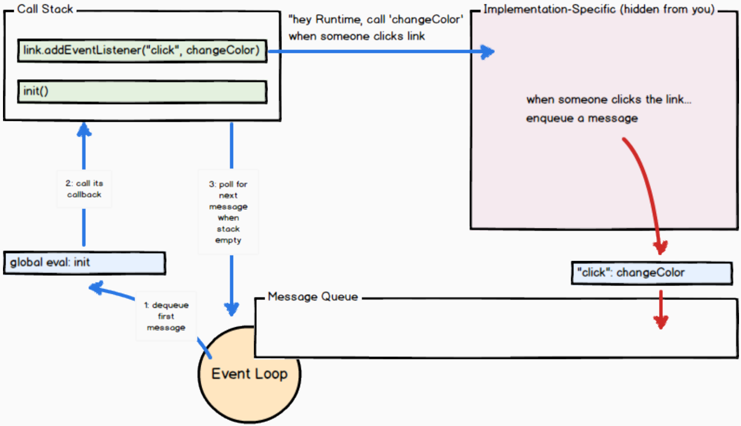
Motivation for Parallelization

- Performance
- . . . but moreover, a better user experience

A better User Experience?

- Because JavaScript is single threaded
- Long running tasks are blocking the UI-thread
- and therefore, the UI is not responsive

The JavaScript Event Loop is the Reason therefore


Call Stack
link.addEventListener("click", changeColor)
init()
"hey Runtime, call 'changeColor' when someone clicks link
Implementation-Specific (hidden from you)
when someone clicks the link... enqueue a message
2: call its callback
3: poll for next message when stack empty
global eval: init
"click": changeColor
1: dequeue first message
Message Queue
Event Loop

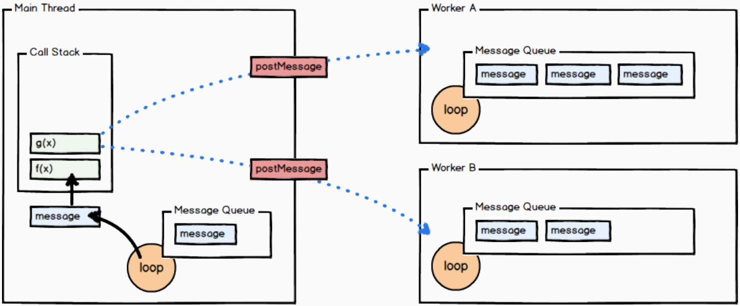
[1]

Initial Position

Technologies offered by the Runtime Environment

- Browser: Web Worker Standard [2]
- Node: Child Process [3]
- JVM: RingoJS [4]

Web Worker Architecture



[1]

Memory

Each Web Worker uses a distinct runtime environment (e.g. V8), therefore, the memory of each web worker is distinct too.

“Simple” Web Worker Example

main.js

```
const worker = new Worker("./worker.js");
worker.postMessage(40);

worker.addEventListener(
  "message",
  result => console.log(result.data)
);
```

“Simple” Web Worker Example

main.js

```javascript
const worker = new Worker("./worker.js");
worker.postMessage(40);

worker.addEventListener(
  "message",
  result => console.log(result.data)
);
```

worker.js

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

onmessage = function (event) {
  const num = event.data;
  const result = fib(num);
  postMessage({
    number: num,
    fib: result
  });
};
```

But I also have to. . .

- Handle Errors
- Return a Promise in the UI-Thread
- Perform the Computation for multiple Items
- Besides, it should run on Node.JS too

And I don’t like that. . .

- code splitting is enforced by technology instead of by semantics
- the messaging model results in a clear seam
- integration adds non inherent complexity
- the build gets far more complicated

Parallel.es

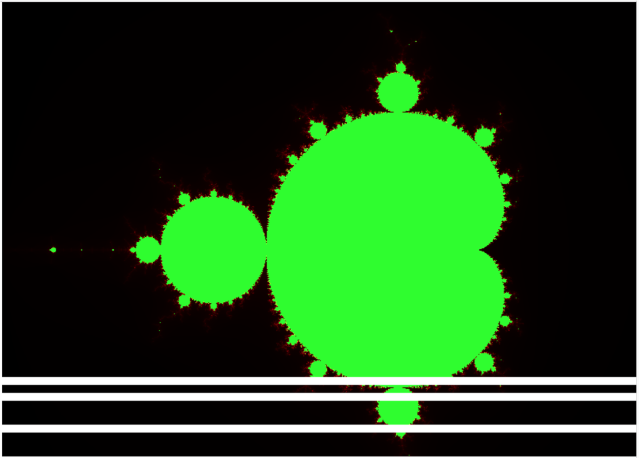
Parallel.es eases parallelizing JavaScript applications

- The API is type-safe
- . . . and runtime environment independent, write once, run everywhere
- Uses a thread pool
- Detects the optimum of threads to create

“Simple” Example

```
import parallel from "parallel-es";

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

parallel.run(fib, 40)
  .catch(error => console.error(error))
  .then(result => console.log(result));
```

The Mandelbrot Showcase



`https://michareiser.github.io/parallel-es-example/`

Code for Single Threaded Execution

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

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

function computeMandelbrotLine(y) {
  const line = new Uint8ClampedArray(imageWidth * 4);
  for (let x = 0; x < imageWidth; ++x) {
    line[x * 4] = computePixel(x, y);
  }
  return line;
}

const result = _.chain()
  .range(imageHeight)
  .map(computeMandelbrotLine)
  .value();

// draw to canvas
```

Idea

- Compute Lines in Background Threads
- Preferred, create as many Background Threads as CPU's are available

Poor Mans Solution

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

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

function computeMandelbrotLine(y) {
  // ...
  return line;
}

for (int i = 0; i < imageHeight; ++i) {
  parallel.run(computeMandelbrotLine, i).then(line => {
    // draw to canvas
  });
}
```

It works[1]! But...

- I don't want to be responsible to split the work on multiple threads
- Flow of logic is hard to catch

[1] Actually, it depends, details follow

Therefore, Parallel.es offers a descriptive API

- Inspired by lodash / underscore
- Handles Work Partitioning
- Allows subscribing to sub results
- . . . or the joined overall result

Descriptive Implementation

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

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

function computeMandelbrotLine(y) {
  const line = new Uint8ClampedArray(imageWidth * 4);
  for (let x = 0; x < imageWidth; ++x) {
    line[x * 4] = computePixel(x, y);
  }
  return line;
}

parallel
  .range(imageHeight)
  .map(computeMandelbrotLine)
  .subscribe((subResult, index, batchSize) => /* draw line */)
  .catch(error => /* handle error in computation of any line */)
  .then(result => /* handle overall result */);
```

Difficulties

Each Worker has its distinct memory and therefore,

- data needs to be transferred between workers
- . . . as well as all functions executed in background threads

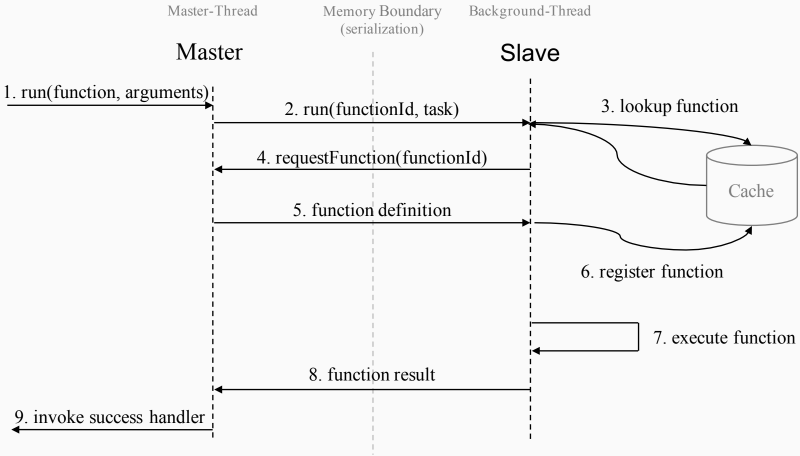
Comparsion to other Languages

Other languages, e.g. Java, have a shared memory that is accessible by all threads. However, parallelization in JavaScript is more like multiprocess programming using inter process communication.

Two Possibilities

- Runtime Serialization of Functions
- Transpilation of Source Code

Runtime Serialization of Functions - Overview



Runtime Serialization of Functions - Serialization

```javascript
const s = func.toString();
const name = getFunctionName(func);
const args = s.substring(s.indexOf("(") + 1, s.indexOf(")")).split(",");
const body = s.substring(s.indexOf("{") + 1, s.lastIndexOf("}")).trim();

const definition = {
  argumentNames: args.map(arg => arg.trim()),
  body,
  id,
  name: name ? name : undefined
};
```

Runtime Serialization of Functions - Deserialization

```
if (definition.name) {
  const args = definition.argumentNames.join(", ");
  const source = `return function ${definition.name} (${args}) {
    ${definition.body}
  };`;
  const wrapper = Function.apply(undefined, [ source ]);
  return wrapper();
}

return Function.apply(undefined, [
  ...definition.argumentNames,
  definition.body
]);
```

But what about a Function's Closure?

- Transitive Functions
- . . . or Variables referenced from the Function's outer Scope

are not supported by Runtime Serialization

Why Not?

It's possible to analyze a function at runtime, e.g. by using Babel. However, there is no way to get access to the values of a function's closure as it is the case in C#. This is a security feature of JS.

The Issue with Function Closures and Runtime Serialization

UI-Thread

```
const width = 10000;
const height = 10000;

function computePixel(x, y) {
  // ...
}

function computeMandelbrotLine(y) {
  const l = new Uint8ClampedArray(width * 4);
  for (let x = 0; x < width; ++x) {
    l[x * 4] = computePixel(x, y);
  }
  return l;
}

parallel
  .range(height)
  .map(computeMandelbrotLine)
  .then(result => /* handle overall result */);
```

The Issue with Function Closures and Runtime Serialization

UI-Thread

```
const width = 10000;
const height = 10000;

function computePixel(x, y) {
  // ...
}

function computeMandelbrotLine(y) {
  const l = new Uint8ClampedArray(width * 4);
  for (let x = 0; x < width; ++x) {
    l[x * 4] = computePixel(x, y);
  }
  return l;
}

parallel
  .range(height)
  .map(computeMandelbrotLine)
  .then(result => /* handle overall result */);
```

Worker-Thread

```
function computeMandelbrotLine(y) {
  const l = new Uint8ClampedArray(width * 4);
  for (let x = 0; x < width; ++x) {
    l[x * 4] = computePixel(x, y);
  }
  return l;
}
```

Issue

`height` variable and `computePixel` function are not defined in worker-thread.

Source Code Transpilation

- Analyses all calls to `parallel`
- Extracts passed functions
- . . . and as well transitive functions
- Registers these functions in the background-thread source file
- Split into a Babel (extraction) and Webpack-Plugin (registration)

Transpiled UI-Thread Code of Mandelbrot

UI-Thread

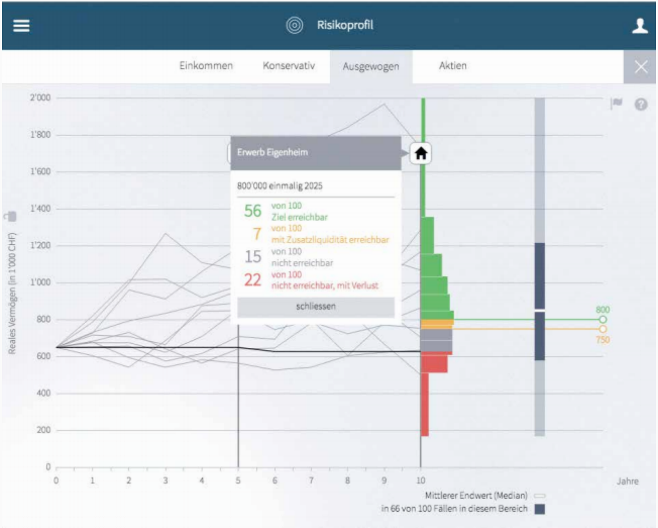
```
const width = 10000;
const height = 10000;

function computePixel(x, y) {
  // ...
}

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

function computeMandelbrotLine(y) {
  const l = new Uint8ClampedArray(width * 4);
  // ...
  return l;
}

parallel
  .range(height)
  .inEnvironment(_environmentExtractor())
  .map({
    identifier: "static:_entrycomputeMandelbrotLine",
    _______isFunctionId: true
  })
  .then(result => console.log(result));
```

Worker-Thread

```
var width;
function computePixel(x, y) {
  // ...
}

function computeMandelbrotLine(y) {
  var l = new Uint8ClampedArray(width * 4);
  // ..
  return l;
}

function _entrycomputeMandelbrotLine() {
  try {
    var _environment = arguments[arguments.length - 1];
    width = _environment.width;
    return computeMandelbrotLine.apply(this, arguments);
  } finally {
    width = undefined;
  }
}

slaveFunctionLookupTable.registerStaticFunction({
    identifier: 'static:_entrycomputeMandelbrotLine',
    _______isFunctionId: true
  }, _entrycomputeMandelbrotLine);
```

Use Case

Riskprofiling



Erwerb Eigenheim

800'000 einmalig 2025

56 von 100 Ziel erreichbar

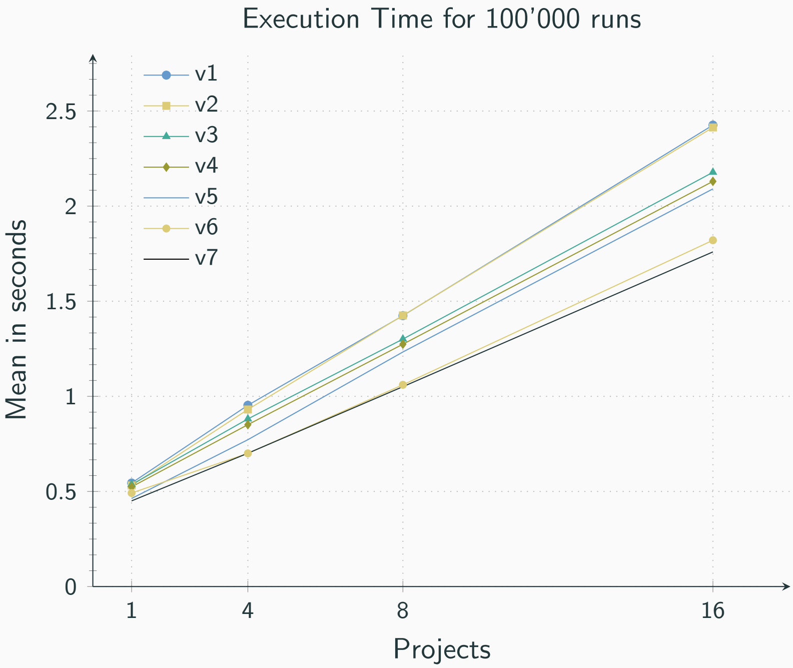
7 von 100 mit Zusatzliquidität erreichbar

15 von 100 nicht erreichbar

22 von 100 nicht erreichbar, mit Verlust

Performance Improving prior Parallelization

- A problem needs to be parallelizable
- Parallelization always adds complexity
- Maybe it's enough to improve "synchronous" code


Execution Time for 100'000 runs
v1
v2
v3
v4
v5
v6
v7
Mean in seconds
0
0.5
1
1.5
2
2.5
1
4
8
16
Projects

V1, authored by Micha Reiser

- Computes the simulation result over 15 years
- For each project
  - Creates the four groups “red, gray, yellow, green”
  - Creates for each group 10 buckets and assigns the simulation values
- Median, min, max are computed in the chart

Main Concern

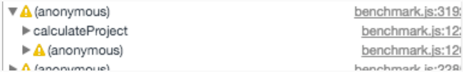
Simulation returns all values instead of the needed information median, min and max. Therefore, absolutely unsuited for parallelization because of large data amount to be transferred.

V2, only returns Information needed by Chart

- Marginal Faster (no array resizes, $\approx$ 20ms)
- Chart needs to perform less computations
- Easier to parallelize, less data needs to be transferred

V3, Address V8 deoptimizations

If you see this...



```
▼ ⚠ (anonymous)                  benchmark.js:319
    ▶ calculateProject            benchmark.js:12
    ▶ ⚠ (anonymous)               benchmark.js:12
  ▶ ⚠ (anonymous)                benchmark.js:228
```

... you are doomed ;)

...or at least, there is much Room for Improvements

V8 Optimization Killers [5]

- Unsupported Syntax (e.g. `debugger`, `eval`, `with`, generator functions...)
- Manipulating `arguments`
- Very large Switch-case Statements (+128 cases)
- `for in`
- Infinite Loops with deep logic or unclear exit
- Others

...or at least, there is much Room for Improvements

V8 Optimization Killers [5]

- Unsupported Syntax (e.g. `debugger`, `eval`, `with`, generator functions...)
- Manipulating `arguments`
- Very large Switch-case Statements (+128 cases)
- `for in`
- Infinite Loops with deep logic or unclear exit
- Others

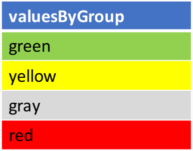
Workaround

- Clean Code
- Small Functions

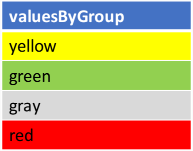
In this Case, the Reason is a Dynamic Object Structure

```
const valuesByGroup: { [groupName: string]: number } = {};
const bucketSize = Math.round(simulatedValuesThisYear.length / NUMBER_OF_BUCKETS);
const buckets: IBucket[] = [];

for (let i = 0; i < simulatedValuesThisYear.length; i += bucketSize) {
  const bucket: IBucket = {
    max: Number.MIN_VALUE,
    min: Number.MAX_VALUE,
    subBuckets: {}
  };

  for (let j = i; j < i + bucketSize; ++j) {
    const value = simulatedValuesThisYear[j];
    bucket.min = Math.min(bucket.min, value);
    bucket.max = Math.max(bucket.max, value);

    const group = groupForValue(simulatedValuesThisYear[j], groups);
    valuesByGroup[group.name] = (valuesByGroup[group.name] || 0) + 1;
    const subBucket = bucket.subBuckets[group.name] = bucket.subBuckets[group.name] ||
            { group: group.name, max: Number.MIN_VALUE, min: Number.MAX_VALUE };
    subBucket.min = Math.min(subBucket.min, value);
    subBucket.max = Math.max(subBucket.max, value);
  }

  buckets.push(bucket);
}
```

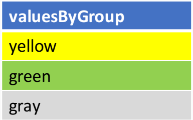
…that makes it Impossible to Optimize the Property Access



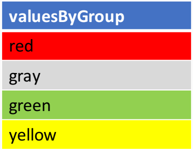
valuesByGroup
green
yellow
gray
red



valuesByGroup
yellow
green
gray
red



valuesByGroup
yellow
green
gray

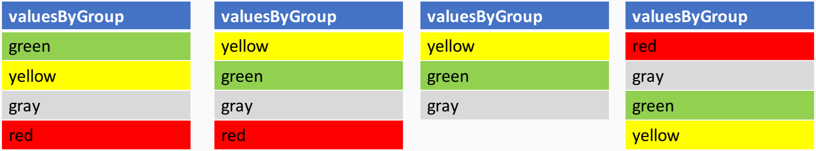


valuesByGroup
red
gray
green
yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for "green":

1. $+0$

...that makes it Impossible to Optimize the Property Access



valuesByGroup
green
yellow
gray
red

valuesByGroup
yellow
green
gray
red

valuesByGroup
yellow
green
gray

valuesByGroup
red
gray
green
yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for "green":

1. +0
2. +4 (32bit pointer)

…that makes it Impossible to Optimize the Property Access

valuesByGroup	valuesByGroup	valuesByGroup	valuesByGroup
green	yellow	yellow	red
yellow	green	green	gray
gray	gray	gray	green
red	red		yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for “green”:

1. +0
2. +4 (32bit pointer)
3. +4

...that makes it Impossible to Optimize the Property Access

valuesByGroup
green
yellow
gray
red

valuesByGroup
yellow
green
gray
red

valuesByGroup
yellow
green
gray

valuesByGroup
red
gray
green
yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for “green”:

1. +0
2. +4 (32bit pointer)
3. +4
4. +8

…that makes it Impossible to Optimize the Property Access

valuesByGroup
green
yellow
gray
red

valuesByGroup
yellow
green
gray
red

valuesByGroup
yellow
green
gray

valuesByGroup
red
gray
green
yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for "green":

1. +0
2. +4 (32bit pointer)
3. +4
4. +8
5. …

…that makes it Impossible to Optimize the Property Access

valuesByGroup
green
yellow
gray
red

valuesByGroup
yellow
green
gray
red

valuesByGroup
yellow
green
gray

valuesByGroup
red
gray
green
yellow

Depending on the order of the simulated values the offset for the properties differ, e.g. for “green”:

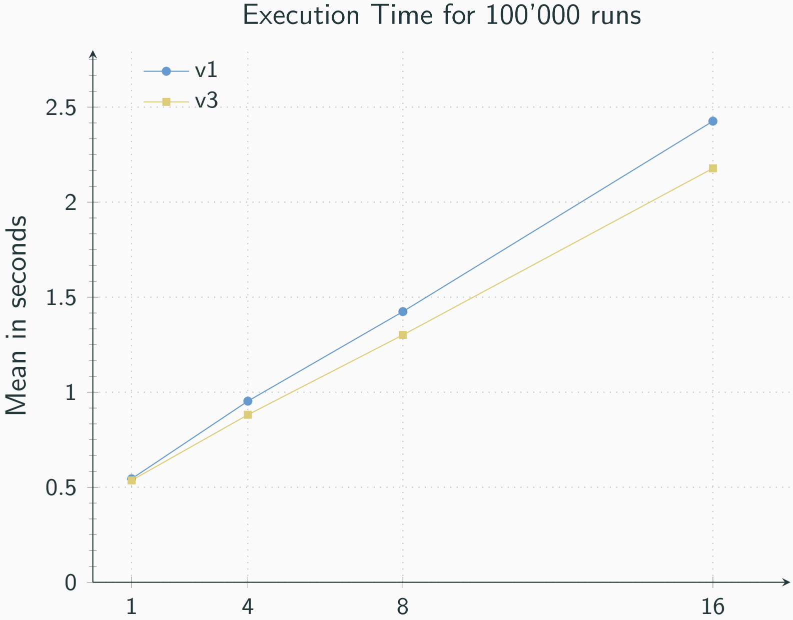
1. +0
2. +4 (32bit pointer)
3. +4
4. +8
5. …

Therefore, V8 deoptimizes the **whole** function

Solution: Ensure that the Object is always Initialized in the same Order

```
const bucket: IBucket = {
  max: Number.MIN_SAFE_INTEGER,
  min: Number.MAX_SAFE_INTEGER,
  // Needed to avoid deoptimization because of changed attribute orders in subBuckets. Ini
  subBuckets: {
    green: {
      group: "green",
      max: Number.MIN_SAFE_INTEGER,
      min: Number.MAX_SAFE_INTEGER,
      empty: true
    },
    yellow: {
      group: "yellow",
      max: Number.MIN_SAFE_INTEGER,
      min: Number.MAX_SAFE_INTEGER,
      empty: true
    },
    //...
  }
};
```

This small Change Improves Performance by up to 235ms



Further Improvements

- v4: Reduce Nesting of Functions (-47ms)
- v5: Create Arrays with expected Size (-60ms)
- v6: Only simulate number of Years needed (-270ms)
- v7: Avoid Object Destructuring (-61ms)

Overall

Improvement by 666ms in best case (1.76 instead of 2.426s) or by 28%

Further Improving the Performance by Parallelizing the Computation

- Computation consists of

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
    - Simulation

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
  - Simulation
  - Creating Buckets for each Projects

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
  - Simulation
  - Creating Buckets for each Projects
- Parallelization by computing buckets for multiple projects simultaneous

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
  - Simulation
  - Creating Buckets for each Projects
- Parallelization by computing buckets for multiple projects simultaneous
- However, simulation is most expensive

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
  - Simulation
  - Creating Buckets for each Projects
- Parallelization by computing buckets for multiple projects simultaneous
- However, simulation is most expensive
- But sharing the data of the simulation between worker is as expensive as performing the simulation in each worker

Further Improving the Performance by Parallelizing the Computation

- Computation consists of
    - Simulation
    - Creating Buckets for each Projects
- Parallelization by computing buckets for multiple projects simultaneous
- However, simulation is most expensive
- But sharing the data of the simulation between worker is as expensive as performing the simulation in each worker
- So it only makes sense to parallelize if there are 2+ projects

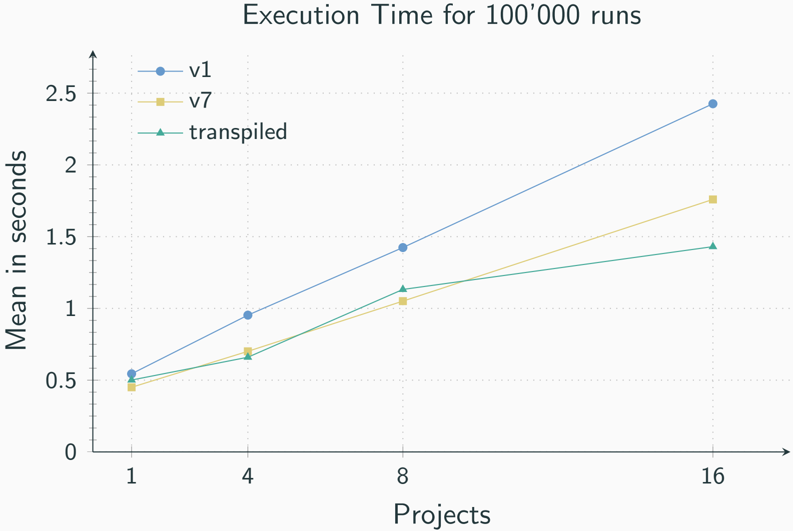
Further Improving the Performance by Parallelizing the Computation

- Computation consists of
  - Simulation
  - Creating Buckets for each Projects
- Parallelization by computing buckets for multiple projects simultaneous
- However, simulation is most expensive
- But sharing the data of the simulation between worker is as expensive as performing the simulation in each worker
- So it only makes sense to parallelize if there are 2+ projects

But Performance is not everyting

The UI is no longer blocked for ≈ 2s

Performance Improvement of up to 300ms (2 Cores, HT)



But sometimes it is even slower.

Conclusion

Conclusion

- Parallel.es eases writing multithreaded JS applications
- It is platform independent
- Compared to alternatives ([6]–[8]),
  - The API is type-safe
  - Performs well in most cases
  - Transpilation allows a seamless integration
- Source Map support is helpful
- However, no support for
  - Low Level synchronization primitives
  - Recursive Tasks

However

- The problem needs to fit to be easy parallelizable

However

- The problem needs to fit to be easy parallelizable
- . . . otherwise, a lot of prior work might be needed

However

- The problem needs to fit to be easy parallelizable
- . . . otherwise, a lot of prior work might be needed
- . . . maybe, these improvements are already sufficient

However

- The problem needs to fit to be easy parallelizable
- . . . otherwise, a lot of prior work might be needed
- . . . maybe, these improvements are already sufficient
- . . . and parallelization always adds non-inherent complexity

Further Work

- WebAssembly: A new standard of the Web

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine
- Allows to run C++ code in the browser

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine
- Allows to run C++ code in the browser
- However, who wants to write C++ ;)

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine
- Allows to run C++ code in the browser
- However, who wants to write C++ ;)
- Controversial Idea: compile a subset of JS to Web Assembly

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine
- Allows to run C++ code in the browser
- However, who wants to write C++ ;)
- Controversial Idea: compile a subset of JS to Web Assembly
- Hopefully profit from better performance

Further Work

- WebAssembly: A new standard of the Web
- Provides a stack machine
- Allows to run C++ code in the browser
- However, who wants to write C++ ;)
- Controversial Idea: compile a subset of JS to Web Assembly
- Hopefully profit from better performance
- And if time allows, take benefit of the pthreads API

Further Resources

- Project Page [9]
- Project Thesis Paper [10]
- Optimization Killers [5]

References

E. Swenson-Healey. (2013), The javascript event loop: Explained, [Online]. Available: http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/.

"Web Worker", W3, Tech. Rep., Sep. 2015. [Online]. Available: https://www.w3.org/TR/workers/.

Node.js Foundation. (2016), Child process, [Online]. Available: https://nodejs.org/api/child_process.html (visited on 11/13/2016).

- Ringo. (2016), Ringojs, [Online]. Available: http://ringojs.org/ (visited on 11/21/2016).
- bluebird. (2017), Optimization killers, [Online]. Available: https://github.com/petkaantonov/bluebird/wiki/Optimization-killers.
- A. Smith. (2016), Hamsters.js, [Online]. Available: http://www.hamsters.io/ (visited on 11/12/2016).
- A. Savitzky and S. Mayr. (2016), Parallel.js, [Online]. Available: https://parallel.js.org/ (visited on 11/11/2016).
- A. Wermke. (2016), Threads.js, [Online]. Available: https://github.com/andywer/threads.js (visited on 11/11/2016).

M. Reiser. (2017), Parallel.es, [Online]. Available: https://michareiser.github.io/parallel.es/.

——, “Parallelize javascript computations with ease”, , 2016. [Online]. Available: https://github.com/MichaReiser/parallel-es-report/blob/master/parallel-es.pdf.

Questions and Discussion