Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новосибирский государственный технический университет»

Кафедра вычислительной техники

Расчетно-графическая работа

по дисциплине «Методы и средства высокопродуктивных вычислений»

*«Параллельные вычисления в JavaScript»*

Факультет: АВТ

Группа: АММ-15

Студент: Моргунов А.С.

Преподаватель: Малявко А.А.

Новосибирск, 2017

# Оглавление

Оглавление 2

Введение 3

1 Web workers 4

2 Библиотека Parallel.js 6

3 Тестирование 8

Заключение 11

Список литературы 12

Приложение А 13

Приложение Б 15

# Введение

В первое время, после появления языка *JavaScript*, никто не беспокоился о его производительности. *JavaScript* изначально был создан как простое средство для исполнения небольших сценариев в веб-странице. Но спустя 22 года он стал одним из самых популярных языков программирования, имеющий в себе большое количество отличных парадигм и гибких решений, но и содержащий много недостатков.

Один из недостатков языка *JavaScript* — это его подход к выполнению большого объема вычислений. Обычно задачи, выполняющиеся продолжительное время, работают незаметно в фоновом режиме, позволяя пользователю продолжать работать над другими аспектами задачи. Но *JavaScript* по своей природе язык однопоточный, поэтому любые трудоемкие вычисления прерывают выполнение всех других задач на странице, вынуждая пользователя ожидать их завершения. Он не позволяет создавать дочерние потоки, взаимодействовать с ними, что в свою очередь является проблемой для разработчиков современных приложений.

Есть несколько решений данной проблемы. Эти решения основаны на разбиении долговременных задач на несколько меньших частей и исполнении этих частей по одной с помощью методов *setInternval()* или *setTimeout().* Для некоторых типов задач это решение работает хорошо (например, это практичный способ для анимирования содержимого холста). Но если нужно выполнить одну очень долгую операцию безостановочно от начала до конца, этот метод порождает большую сложность.

Целью этой работы является рассмотреть технологию *Web workers*, который позволяет создавать новые потоки и библиотеку *Parallel.js*, которая предоставляет удобное *API* для работы с *Web workers*.

# Web workers

Спецификация *HTML5* предлагает лучшее решение в виде специализированного объекта, называющегося потоком (*web worker*), предназначенного для выполнения фоновых вычислений. Для выполнения долговременной задачи нужно создать новый поток, даем ему необходимый код и запустить его выполнять поставленную задачу. В процессе выполнения потоком своей задачи с ним можно безопасно поддерживать ограниченное взаимодействие посредством сообщений.

Суть технологии проста — в отдельные файлы выносятся функции, обеспечивающие функционирование *AJAX*, либо функции обрабатывающие большие массивы информации, которые во время работы уменьшают скорость построения страницы. При выполнении скрипта на стороне браузера создается специальный объект *Worker*, который и отвечает за вызов необходимых функций. Поток позволяет исполнять код *JavaScript* в фоновом режиме, в то время как на переднем плане исполняются другие задачи. *Web Workers* поддерживаются всеми современными веб-браузерами.

Перед началам работы с *Web Workers* необходимо проверить, поддерживаются они браузером:

if (!!window.Worker) {

// поддерживается

}

Для создания объекта потока необходимо вызвать функцию-конструктор *Worker*, и первым параметром передать имя файла с исполняемой функцией.

var worker = new Worker("worker.js");

Созданный объект обладает следующими основными методами:

* postMessage(); ключевой метод, инициализирующий обмен данными, отправляет данные потоку
* onmessage(); метод, исполняемый при поступлении ответа от вызванного "работника".
* onerror(); метод, вызываемый при возникновении ошибок

Все браузеры (которые поддерживают данную спецификацию) позволяют обмениваться строковыми и JSON-объектными сообщениями. Сообщение как может быть отослано *Worker*, так и *Worker* может ответить родительской странице таким сообщением. Ниже приведен пример обмена сообщениями.

Обмен сообщениями производится при помощи *API postMessage* следующим образом:

// index.js (Главный файл)

var worker = new Worker("worker.js");

// Ждем сообщений от worker

worker.onmessage = function(e){

// Сообщение от клиента:

console.log(e.data);

};

// запускаем (Worker)

worker.postMessage("start");

// worker.js Клиент

self.onmessage = function(e){

if ( e.data === "start" ) {

// Выполняем какие-нибудь вычисления

done();

}

};

function done(){

// Отправляем полученный результат на главную страницу

postMessage("done");

}

Однако, есть несколько серьезных ограничений:

1. *Worker* не имеет доступа к *DOM*. Никакого доступа к *document*, *getElementById* и т.д. (Наиболее важными исключениями из этого правила будут методы *setTimeout*, *setInterval* и *XMLHttpRequest*.)
2. У worker нет прямого доступа к родительской странице.
3. Максимальное количество *workerов* – 256, после чего стек переполняется и пишет ошибку: "*Maximum number of Web Worker instances(256) exceeded for this window*." [1].

Эти ограничения на передачу информации при помощи сообщений имеет под собой веские основания: это позволяет безопасно запускать дочерний скрипт (потому что он не может повлиять на родительский) и защищает родительский поток (обеспечение безопасности для *DOM* из других потоков).

Есть много примеров использования данной технологии, например, расчет освещения (*RayTracing*) [2]. Этот пример использует *Canvas* для отрисовки рассчитанной сцены. Если включить *Web Workers* (есть возможность включить через веб-страницу), то заметно, как картинка отрисовывается по частям. Это происходит благодаря разбиению всей работы на части и поручение каждого набора пикселей отдельному *Worker*. Этот *Worker* затем падает массив цветов для отрисовки на Canvas, а родительская страница их применяет.

Другой пример: эмуляция огня [3]. Этот пример пытается ограничить несколько случайных точек, используя алгоритм эмуляции огня (*simulated annealing*) [4]. Также здесь приведено анимированное *PNG*-изображение, которое вращается, пока идет вычисление в фоновом режиме.

# 2 Библиотека Parallel.js

В настоящее время логику веб-приложений довольно часто переносят на сторону клиента и довольно часто приходится обрабатывать большое количество данных. Как правило, делать длительную обработки в основном потоке пользовательского интерфейса неверный подход, так как пользователь заметит «зависания» (либо во время обработки интерфейс не будет доступен вообще). *Web Workers* являются отличным решением проблемы, но использовать их в чистом виде сложно и не совсем удобно.

Библиотека *Parallel.js* - это небольшая библиотека, «обёртка» над *JavaScript Web Workers*, которая позволяет создавать интересные приложения с параллельной обработкой с помощью *JavaScript* [6]. Используя *Web Workers* необходимо создавать отдельный скрипт, который будет загружен в отдельный рабочий поток, что не удобно, когда нужно разделить код между основными и рабочими потоками. *Parallel.js* решает эту проблему и обеспечивает хороший *API* для выполнения общих задач.

*Parallel.js* позволяет порождать (англ. *spawn*) рабочих (*worker*), содержащий одну или несколько функций, определенный в родительском потоке, а не в виде отдельного файла. Достаточно просто передать имена функций и аргументы для вызова функции и *Parallel.js* породит рабочий поток, запустит функцию с переданными аргументами и вернет результат обратно в основной поток асинхронно с использованием *API* на основе *Promise*.

В библиотеки также присутствует *API* для обработки больших массивов данных, которые позволяет разделить данные на части, которые будут обрабатываться параллельно с произвольным числом рабочих потоков. Когда вся обработка будет завершена, результат объединяется вместе и возвращается в основной поток. По умолчанию количество потоков будет равно количество ядер *CPU* (используется свойство браузера *NavigatorConcurrentHardware.hardwareConcurrency*). Так же, если в браузере веб-воркеры недоступны, то библиотека предоставляет синхронный *фоллбэк* для обработки данных с помощью таймеров.

Для параллельной обработки необходимо создать экземпляр функции-конструктора *Parallel*, в который первым параметром передавать массив с данными (или любой *JSON*-сериализуемый объект), а вторым - объект с опциями (например, количество потоков).

var data = [1, 2, 3, 4];

var p = new Parallel(data, { maxWorkers: 4 });

Далее все операции можно организовать с помощью цепочек из доступных функций:

- *spawn(fn)* - функция порождает новый процесс в рабочем потоке. Функция *fn* принимает на вход текущие данные, и возвращает и функции обновленные текущие данные.

var p = new Parallel('forwards');

// порождаем новый поток для обработки данных

p.spawn(data => {

data = data.reverse();

return data;

})

.then(data => {

console.log(data) // logs sdrawrof

});

- *map(fn)* - данная функция применит предоставленную функцию к каждому элементу в данных. Библиотека будет порождать один поток для каждого элемента массива в данных или заданного аргумента *maxWorkers*. Возвращенные значения будут сохранены для дальнейшей обработки.

var p = new Parallel([0, 1, 2, 3, 4, 5, 6]);

var log = function () { console.log(arguments); };

function fib(n) {

return n < 2 ? 1 : fib(n - 1) + fib(n - 2);

};

p.map(fib).then(log)

- *reduce(fn)* - функция применяет операцию к каждому элементу данных и возвращает скалярное значение, созданное этой операцией. Эту функцию можно использовать для комбинирования результатов, например, суммируя числа. Это принимает функцию сокращения, которая получает аргумент, данные, массив сохраненного значения и текущий элемент.

Ниже представлен пример, в котором подсчитывается сумма положительных чисел:

const p = new Parallel([0, 7, -2, 3, -1, 15, -6, -7, 8]);

p

.map(number => number > 0 : number : 0)

.reduce(acc => acc[0] + acc[1])

.then(result => console.log(result));

# 3 Тестирование

Для тестирования работы *Web Workers* и библиотеки *Parallel.js* была поставлена задача: подсчитать количество чисел от N до M, которые являются простыми числами и палиндромами одновременно.

Программа для *Web Workers* находится в приложении А. Первый файл (*workers.js*) содержит цикл, который проходится от M до N и проверяет число на каждой итерации на то, что оно является простым и палиндромом. Как цикл заканчивается, количество таких чисел передается в главный поток. В главном потоке промежуток чисел делится на K (количество веб-воркеров) частей и создает K потоков, каждый их которых получает соответствующий промежуток.

По завершению работы в консоль браузера выводится время, затраченное на выполнение скрипта.

var startTime = Date.now();

run((result) => {

console.log((Date.now() - startTime)/1000 + 'sec');

});

Этот же код был выполнен в основном потоке браузера. В приложении Б представлена реализация алгоритма с помощью библиотеки *Parallel.js*. Количество кода вышло в несколько раз меньше, например, вся логика уместилась в семи строках.

p

.map(n => isPrime(n) && isPalindrome(n) ? 1 : 0)

.reduce(d => d[0] + d[1])

.then(function(result) {

console.log((Date.now() - startTime)/1000 + 'sec');

console.log("Count: " + result);

});

Результаты представлены на рисунке 1.

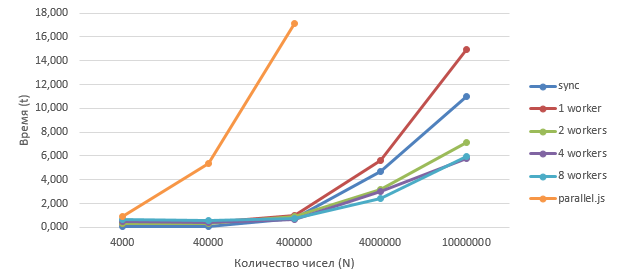


Рис. 1 – Результаты тестирования

По графикам можно сделать сразу несколько выводов:

* Из-за того, что веб-воркеры должны подгрузить через *AJAX* дочерний скрипт (сетевые соединения), а также общаться с основным потоком, на небольших данных их использование не является приоритетными;
* При ресурсоемких вычислений (обработки больших данных) c использованием *Web Workers* удалось добиться уменьшения времени выполнения программы в два раза;
* Результаты *parallel.js* показывают очень медленные результаты работы, т.к. для каждой функции создаются свои воркеры, а также, потому что при каждом прогоне тестов создается массив (плюс библиотеки - работа на уровне массивов) и приходится между функциями передавать (копировать) этот массив с данными.

# Заключение

При использовании технологии воркеров для выполнения простых действий выигрыш в скорости будет минимален. Однако при сложных вычислениях можно существенно ускорить работу системы.

*Web Workers* позволяют эффективно использовать пользовательские машины для параллельных вычислений. Если необходимо выполнять задачи с ресурсоёмкими алгоритмами и большими данными, которые полностью «убивают» пользовательский процессор и «замораживают» интерфейс браузера, использование *Web Workers* является отличным решением.

*ParallelJS* позволяет обойти многие проблемы, которые могут возникнуть при использовании *Web Workers*. Кроме того, библиотека обладает хороший API, который содержит несколько полезных абстракций. Но, в ходе тестирования результаты библиотеки оказались намного хуже, чем использование *Web Workers* напрямую.

На данный момент при необходимости вынести вычисления в отдельный поток и если обработка будет производиться не над массивом данных, лучшим решением будет использовать нативную работу с потоками. Но если проблема только в "замораживании" интерфейса и скорость вычислений не является основной проблемой, можно использовать библиотеку *Parallel.js*, которая в несколько раз сократит объем кода и улучшит его читаемость.

# Список литературы

1. Computing with JavaScript Web Workers [Электронный ресурс] https://johnresig.com/blog/web-workers/
2. RayTracing [Электронный ресурс] http://nerget.com/rayjs-mt/rayjs.html
3. Simulated annealing [Электронный ресурс] https://people-mozilla.org/~prouget/demos/worker\_and\_simulatedannealing/index.xhtml
4. Simulated annealing [Электронный ресурс] https://en.wikipedia.org/wiki/Simulated\_annealing
5. Многопоточность в JavaScript [Электронный ресурс] http://stepansuvorov.com/blog/2013/12/webworkers/
6. ParallelJS [Электронный ресурс] https://parallel.js.org/

# Приложение А

**// Главный файл**

var start = 0;

var end = 10000000;

var workers = 1;

// var workers = navigator.hardwareConcurrency;

function run(callback) {

var result = 0;

var endedWorkers = 0;

var subrangeLength = (end - start)/workers;

var currentStart = start;

Array.from('x'.repeat(workers)).map(function () {

var worker = new Worker('worker.js');

worker.addEventListener('message', function(e) {

console.log('Worker said: ' + e.data);

result += e.data;

endedWorkers++;

if (endedWorkers === workers) {

callback(result);

}

}, false);

worker.postMessage({

from: currentStart,

to: currentStart + subrangeLength

});

currentStart += subrangeLength;

});

}

var startTime = Date.now();

run(function(result) {

console.log((Date.now() - startTime)/1000 + 'sec');

console.log(result);

});

**// Файл воркера**

function isPrime(num) {

for(var i = 2, s = Math.sqrt(num); i <= s; i++) {

if(num % i === 0) {

return false;

}

}

return num !== 1;

}

function isPalindrome(str) {

var reversedString = str

.toString()

.split('')

.reverse()

.join('');

return reversedString === str.toString();

}

function process(start, end) {

var count = 0;

for(var i = start; i < end; i++) {

if (isPalindrome(i) && isPrime(i)) {

count++;

}

}

return count;

}

self.addEventListener('message', function(e) {

var data = e.data;

console.log(JSON.stringify(data));

var count = process(data.from, data.to);

self.postMessage(count);

self.close();

}, false);

# Приложение Б

var start = 0;

var end = 100000;

var data = [];

for(var i = start; i < end; i++) {

data.push(i);

}

var p = new Parallel(data, { maxWorkers: 4 });

p.require(isPrime);

p.require(isPalindrome);

var startTime = Date.now();

p

.map(n => isPrime(n) && isPalindrome(n) ? 1 : 0)

.reduce(d => d[0] + d[1])

.then((result) => {

console.log((Date.now() - startTime)/1000 + 'sec');

console.log(result);

});