Poznan University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Master's thesis

LARGE SCALE COMPUTATION USING VOLUNTARY COMPUTING POWER

Tomasz Fabisiak

Supervisor
Arkadiusz Danilecki, Ph.D

Poznań, 2017

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Contents

Chapter 1

Introduction

1.1 Motivation

The Internet has become so popular and powerful that humans are mostly unable to live without it. Every year larger share of the population is using the Internet with more than one device. The devices connected to the Internet are mostly notebooks and mobile phones which are often equipped with multiple CPU's. It is common for the most popular smartphones to have 4 or more CPU's. Notebooks for 250 dollars are likely to have 4 physical cores and as many more logical cores. This power is rarely used in 100% or not even in 50% during basic usage such as browsing websites, watching movies or listening to music. One of the solutions to harness this unused computing power is known from decades and is called 'voluntary computing'. Most of the known forms of voluntary computing require the user to download and install computation software and then use it in the background - this will be called hearafter a 'classic' case. What clearly shows the power of voluntary computing is a case of the search for the Mersenne numbers. Such number has a form of

$$M_n = 2^n - 1$$

If such digit is a prime number, than p is also a prime number, but not the other way. The total number of Mersenne numbers is 47 and the top 12 highest ones were found in a voluntary computing project called GIMPS. [BJ12]
However there are also other forms of voluntary computing - one of them is based on Internet browsers. Such solutions are called Browser Based Voluntary Computing

(BBVC) and solve some problems that a classic form has. Some of these problems are explained below.

1.2 Problem statement

1.2.1 Enhancing standard solutions

The biggest disadvantage of classic voluntary computing is a need for downloading software in order to start computing. This is followed by convincing the user that the software is safe, as well as hoping that the user gets interested in helping as a volunteer. Most of the people that are willing to contribute are young, technical skilled men and it is difficult to encourage other types of Internet users. Users are rather cautious and know the threats of installing any software. Most of them also wouldn't believe that their computer could help with any computations. The solution to that problems is the browser based approach - where the user enters the website, optionally gets asked if they want to help with computations - and they instantly become one of the computation nodes. The only downside of this approach is that the code to be computed must be a JavaScript code. Despite being faster every year - JavaScript is still slower than compiled languages like Java, C or C++. It's speed is connected to being a script language and also being one-threaded (but could be enhanced to be multi-threaded with the help of a mechanism called Web-Workers. WebWorkers work on separate threads and communicate with the main thread via messages).

1.2.2 Lack of trust

There are many solutions providing logic in browsers - JavaScript, Flash, Java Applets and Silverlight plugin but it was difficult for browser distributors to keep track of their security so some of them are already deprecated and some will be soon. The only technologies that are not planned to disappear from browsers are Flash and JavaScript.

If the webpage has any logic inside - it usually as to provide a .js file. These files are JavaScript script codes that browser executes after fetching. Since they are scripts - everyone can look inside and see the code. Also, browsers provide debugging tools

for developers to attach to certain lines of code and e.g. view or change something. In desktop applications that are usually compiled - it is difficult to debug their code and manipulating the requests requires some software that analyzes Ethernet packets. In browser applications it is much easier to manipulate anything. This make frontend of web applications particularly unreliable - and requires two step validation - on the server and client side. This affects also browser based voluntary computing - any user can replace the results of computation with the fake data. Lack of trust is one of the main characteristics of Byzantine Fault tolerance - a common problem for distributed systems. For almost all of the algorithms - detecting byzantine processes is crucial. There are several solutions for this problem and some of them will be explained in further chapters.

1.2.3 Connection issues

There is a problem that a user can lose connection with the server in any moment. What is more - user could leave a website in any moment. User can is also able to open many tabs of the same website in a browser - and if there was no detection of such event - then the user could outreach the number of available CPU's. Let's assume that the user received some work to do, but left the website before finishing the computation - the data assigned to that user will no longer be computed. Thanks to the TCP protocol - server can detect the disconnection and handle this event. The answer to how to handle such situation will be answered in further chapters.

1.2.4 Comfort with writing algorithms

The popular distributed algorithms are usually very different from each other. Some of them require input data, some of them require some kind of communication, other require nodes to be dependent from each other. Above statements make it complicated to build a generic platform in such way, that it will be possible to implement most of the algorithms. Such platform needs also to be intuitive and have an easy way to debug code and handle the messages. It also needs to be configurable and provide an interface to write algorithms in a generic way.

1.3 Aim of the thesis

The aim of the thesis is to extend the possibilities of browser based voluntary computing. Such extension is an application called POVoCoP (Poznan Open Voluntary Computing Platform), that was built by author particularly for this thesis. Several features are provided that are new to browser based voluntary computing such as
- redundancy,
- ability to continue the computation without the need to visit the website,
- making easier to develop new distributed applications for the platform,
- adaptive scheduling,
- basic support for the platform security,
- fault tolerance. It will be shown how it differs from other solutions, how simple it is to write algorithms and most of all - how exactly the common algorithms are implemented in the platform - from the code and database perspective. The platform was meant to allow people to adapt their websites easily to volunteer platforms. Its goal was also to give website owners possibility to write algorithms in a very simple way and use the computing power of their viewers to compute personal algorithms.

1.4 Overview of the thesis

There are six more chapters in this thesis.

The second chapter consists of the summary of author's engineer's thesis. The engineering thesis contains the description of my initial work in the domain of BBVC. There is also the brief history of BBVC based on the survey that was published before the thesis - created by both the author and the supervisor of this thesis.

Next chapter gives detailed information about the system's architecture, the main entities and concepts of the system and their responsibilities. In addition it contains an overview of how the components of the system are connected with each other.

There are also details and examples of the Jobs division between nodes.

In another chapter one can find information about the technologies used to build the platform - both from the frontend and backend side as well as the system and database technologies. It is shown there how are the messages transported and how is the system secured. Chapter ends with the description of the solutions that were proposed to enhance the computation performance as well as the safety and reliability.

Chapter five is about the usage of the system - how the code is submitted and the ways to provide input data into the platform. There is also an explanation of how is the user rewarded for the computation and how they can use gained points across different applications. The chapter all together gives the reader information about the flow of the data from fetching input data, through sending results, to results approval or rejection.

The chapter six presents the example implementations of several algorithms for the PovoCop platform. The exact input data structure, code and methods for results browsing are also described in this chapter. One can also see when there is a need to verify the data and when to use different parameters for job configuration that the platform provides.

In the last chapter there are conclusions and tests results. There is also information about the survey that was made to get feedback from the users about participating in computations. One can find the predictions about the future of browser based voluntary computing.

Chapter 2

Related work

The thesis is based on the earlier works which were authored or co-authored by the author of the thesis. First of this work was engineer's thesis 'Web browser based distributed computing' on Poznan University of Technology, which was defended on February 2016.
Half year later the survey was published on February 2017 in the journal Foundations of Computing and Decision Sciences that is published by Poznan University of Technology.
In [2] the current state of art in browser-based VC was described.
Preliminary description of the PovoCop was also published in 'Job Description Language for a Browser-Based Computing Platform - A Preliminary Report' which was published on ACIIDS 2017 conference in Kaga.

2.1 Author's engineer's thesis

2.1.1 The aim of the engineer's thesis

The aim of engineer's thesis was to create an application for browser based computing as a proof of concept, produce algorithms for it and compare them with native solutions written in fast, low level language C.

2.1.2 The capabilities of created system

System required Job Owners to write code in both frontend and backend side, which both required only JavaScript knowledge. Since it was only proof of concept, the

results were not verified in any way, the loss of input data wasn't handled and the results were only written to a file. System had capability of getting information about the number of user's CPU cores. It used WebWorkers mechanism in order to run code in multiple threads. The created application was working only for algorithms described below and was not reusable for any other. It had small reusable blocks such as a test to find CPU cores or a mechanism to create threads.

2.1.3 Performance

There were 3 algorithms implemented in engineer's thesis
- the Monte Carlo PI computation,
- matrix multiplication
- and array sorting.

System

The system performance was limited by the network speed. The algorithms that required massive input data, such as array sorting and matrix multiplication didn't perform well due to this limitation. The time of computation was usually lower than the time of getting input data from server.

Algorithms

The PI computation had the best performance results from all algorithms due to two conditions. Firstly, it didn't have any input data that had to be sent to users. Secondly, every user had different random number's generator so many results were independent from each other, which produced very precise results of computed PI number.

2.2 FCDS survey on BBVC

This section is based on a survey co-authored by author of this thesis. I summarize here the most important and exemplary solutions; the interested reader may find more information in [2].

2.2.1 The aim of the survey

Our intention for the survey was to list the most popular BBVC platforms over last two decades, compare them and find their similarities and unique solutions to common problems. With this knowledge we produced an architecture that is reliable, lacks the single point of failures and provides generic ways to implement different algorithms. Such architecture was named PovoCop and only had a simple prototype. This master thesis is - on the other hand - a full implementation of this architecture.

2.2.2 Three generations of BBVC

First generation - Java applets

The first generation of browser based voluntary computing application was based on Java applets. The main applications from that period were Javelin, Bayanihan and POPCORN. The first one - Javelin was the most often mentioned in scientific papers. It is an implementation of language independent architecture called Super-Web. The architecture consists of a Client that produces a Java applet and estimates required resources to run it, Hosts that are the machines that run the computation via applets and Brokers that are in the middle between Hosts and Clients - giving credits to users. There was an interesting approach for the overloaded Brokers:

The primary broker was created by user with the use of a configuration file. When the primary broker was overloaded, it chose one of the hosts that were connected, preempted all applications running on that hosts and sent it its own broker code which caused the host to become a new secondary broker. [OFND97]
The second one - Bayanihan was on the second place of the approaches that appeared the most often. This application is managed by HTTP server and Java application that creates problem objects that are mapped with program objects. The pool of data objects was used to manage input data. Using advocate objects - the Java application was able to invoke doWork method on the client side.
The client, after the computation, invoke sendDone method to send results to the server side. Another interesting approach was here - the tasks were assigned in

round-robin algorithm, so whenever there were more clients then jobs - there was a redundancy that was a natural result verifier. [L.F98]
POPCORN on the other hand was a different approach - the client (the Seller) first computed a brief computation that generated a report about their resources. Then the client put their computation time on sale - with the price depending on their resources power described in the report. The price was in a currency called 'Popcoin' which in theory could be transformed into real currency. The Buyer payed the Sellers and provided a code to execute. There was also suggested a possibility for Sellers to put an animated logo to their website in order to receive Popcoins for sharing visitors computing power. [NSON98]

Second generation - single-threaded JavaScript

The problem with the applets was that they required a user to click on pop-up window, download the applet and then allow applet to be run. Due to problems above - the applets never gained popularity. JavaScript however got a lot faster every year with browsers companies (Mozilla, Google) wars over the having the quickest JavaScript runtime. Its performance grew a lot since first implementations in 90's, when it was 10-100 times slower than Java. [FSA+07]
It's big disadvantage was the use of only one thread, but still it could run in a background with no pop-ups or additional activity of user. Above statements were the main reasons why the next generation of BBVC was born. The starting date is assumed to be year 2007, where three approaches appeared, which used Ajax requests to communicate with the server, using XML based messages. [J.J05]

According to best of my knowledge - this was also the year where first genetic algorithms appeared in BBVC. Some projects even implemented special JavaScript implementation of Push3 - a language for genetic programming.
A project AGAGAJ (Asynchronous Genetic Algorithm with JavaScript and JSON) on the other hand used JSON based messages and computed the algorithm on client side before sending the best local results to server. Clients could re-run the algorithm in order to achieve better results.

Third generation - multi-threaded JavaScript

The division between Second and third generation is believed to be a year 2010. Google chrome started to use just-in-time(JIT) compilers in their new runtime called V8, which made JavaScript much faster than before. In addition to that - V8 was used to write HTTP servers in JavaScript - using the technology called Node.js .
The ability of writing the code in the same language on frontend and backend made it easier to implement a mechanism to allow real-time communication between browser and server - such technology is similar to TCP sockets and is called WebSockets. WebSockets mechanism made it finally able for server to send data to client at any moment, which was not possible until then. What is more - a protocol HTML5 was created, which introduces WebWorkers to web development. WebWorkers are JavaScript scripts that run on separate threads and communicate with the main thread only with messages. These technologies made significant changes in BBVC. Multi-threaded computations with real time message passing were clearly missing in previous generations.

2.2.3 Inspiration

2.3 PJDL - Job description language

PovoCop Job Description Language - is a language designed to build PovoCop Jobs in declarative way. It allows Job Owners to separate repetitive tasks from the logic of application. Such tasks could be results verification, input data distribution or exception handling. Tasks and Job configuration can be added to the system by writing them in a human-friendly way using PJDL language. PJDL language could then be used to automatically generate both client's and servers's side codes.
The idea is to submit the Job as a zip archive - containing a main.pjdl file in which there will be job description and two directories - resources with input data and scripts with computation scripts. The descriptions consist of basic job description, resource description and activities.
Basic job description does have a results visibility field which tells the system whether to show the results to: the public (which is a default option), to a group or only to Job Owner. Other two available options are an instantiation (when should

the Job start) and an URL for the final results to be posted.

Resource description gives information about the type of resource, which can be: a **vector**, an **array**, a **cube**, **raw** or a **folder**. First two are one and two dimensional arrays, the cube is three dimensional. The raw means that the scripts are going to handle the resource. Finally a folder means that there will be many files included inside a directory. There are several options for fetching the resources - it could either be embedded in the Job configuration or accessible remotely. The scripts can also be accessed remotely or be embedded in the job configuration.

Activities on the other hand are the procedures which are invoked when particular event happen. There are events such as **ongoal**, **oninit** or simply **oncondition** - which is specified by Job Owner using: and, or, xor or not operators. There are three actions that are possible to be invoked in any activity: http, run and distribute. The first one - http will perform a http request with any of common types (put, get, post or delete) to a specified URL. The run method will simply run a JavaScript piece of code. The distribute action is the most important and its goal is to describe the tasks to be send to the volunteers. The Job Owner can specify whether the data is sent through WebSockets or is going to be available in any URL or possibly handled by the script.

System will automatically divide some traditional data structures like arrays or vectors and send pieces to each browser. If the Job finishes - the result can be automatically merged depending on specified option (**.min**, **.max**, **.avg**, **.sum**, **.nomerge**, **.vector**, **.array** and **.nomerge** - if there is no need to merge the results).

The verification of results of the computations is also configurable and the following options are available: **.none**, **.majority voting**, **.weighted voting**, **.spot checking**, **.self** - where self means that the additional script is going to handle the verification.

The first parser implementation of PJDL was in a language called Elixir which made it difficult to combine it with node.js. The current version of PovoCop is not yet adapted to PJDL, however already the initial work was done to make it easier to adapt PJDL to the platform. Many features were already implemented, and in addition a GUI was created to describe the jobs [ATM17]

2.4 Modern solutions

2.4.1 Polish team project

Two books about Grid and Volunteer computing had been released in Poland by the researchers from Gdansk. [BJ12]
First of the books is written in Polish and consists of three parts. Part one is about distributed computing systems, including their architectures, common properties and approaches for common problems. Part two gives information about the system Comcute that was designed for that book - how are the tasks scheduled, how the distribution and synchronization looks, as well as how the system looks from the security perspective. In the last part the reader can find the use cases of the Comcute application. The examples of system usage are
- image recognition in security cameras
- data distribution in case of database failure
- techniques for computing the probability of random graph to be connected
- hash cracking of algorithms with short key length

The book has a chapter dedicated to BOINC project - one of the most popular volunteer computing platforms - and describes how exactly that system is designed. In BOINC, the client is connected to the Scheduler, which monitors the users resources and does not allow the platform to exceed the CPU, RAM or GPU limit. It also tracks the performance of the nodes, so that stronger machines are more willingly used and weaker machines do not get overloaded. To verify the results BOINC uses the redundancy - it sends the same data to different nodes and checks their consistency.
The Comcute system is also a browser-based voluntary computing platform. The architecture consists of Z - customer user, W - core nodes, S - proxy servers, I - internet users computers. The Z layer is serving the interface to users and validates the customer requests. The proxy layer S is used to separate the computers of I layer from the system core. The W layer is where data is stored. It requires to provide a code to divide the data into chunks which are later sent to the users. It also requires to provide a code to validate the results and set the level of trust

- to send the same input data from 1 and up to 10 additional nodes. The code can be written in 4 technologies that Comcute system support: JavaScript, Java applet, Flash or .Net.
The second book is an English description of the comcute system.

2.4.2 Nimiq blockchain

Nimiq is cryptocurrency that uses a browser-based blockchain implementation. To become a part of the blockchain, it only requires a person to enter a website. Firstly, the browser will fetch the blocks in order to synchronize with the system and after few minutes it will start mining blocks. For every mined block a user gets a reward of 50 Nimiq - which is the system currency. The system also shows the global hash rate and computer's hash rate. There is also an expectation when will the block be mined and is estimated at from few hours to few days depending on computer's resource power. What is interesting - there is a possibility to mine also with node.js servers. If there are low demanding periods e.g. in the night - such server could start computing. This is still work in progress with a working demo version. This project shows how powerful cloud computing is and that even complex computations are possible in JavaScript. [nim]

Chapter 3

Architecture of system

The tentative architecture of the system was proposed in FCDS paper and it did not change much since then. The biggest change affects the directory service which is now basically embedded in the scheduling server. The reason for such approach was the time pressure of creating the platform and the fact that directory server is strictly connected with Scheduler. The initial thought was to have one directory service and several Schedulers connected to it, but it would prevent Schedulers from putting input data in cache. Treating both Directory Service and Scheduler as one application made it possible to update the cache with no additional message passing and synchronization.

The programmers (Job owners) produce the code which is then saved in the repository - such process creates a Job in a platform. In any time the Job owners can see the results of computations. There are many configuration options available to add to the job - for instance - Job owner can decide how many users must verify the result to make it approved. The configuration of the Job, the configuration data for the algorithm and the code are possible to be created or updated all at once with the same API endpoint.

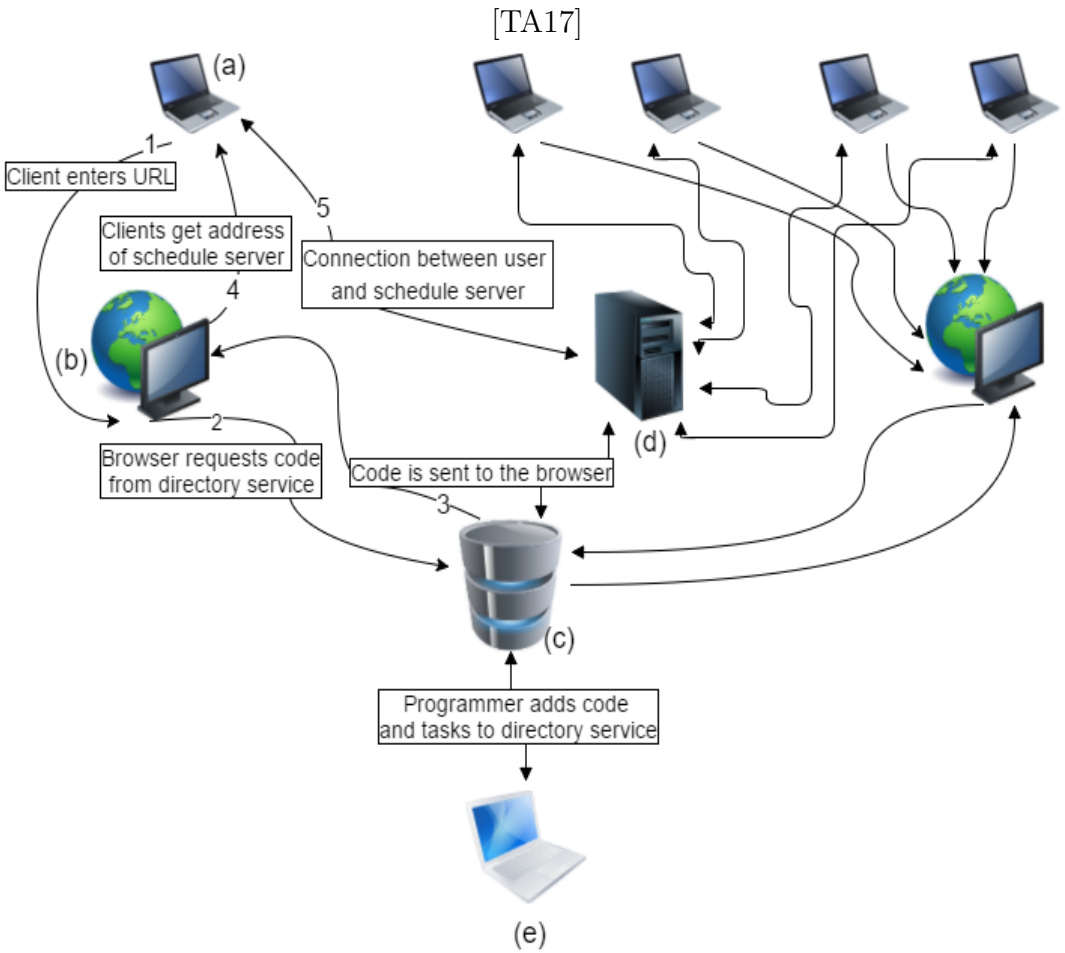
In order for a website to become a part of PovoCop platform, the script must be first added to a website. If user connects to the website - the script makes a connection with their browser to a server. When connection is established and user have never been a volunteer on that website - they receive a brief test in order for server to get the information about their resources. The test takes less than 15 seconds and gives back the information of how many CPU threads does the user have and saves this

information in their cookies. Every new visit to the website from that moment will not require doing any tests as long as the user does not clear their cookies or use another browser distribution.

Server's main job is to schedule the tasks to clients and to assign them to their particular CPU threads. The server sends the code to be computed to the client's browser, which uses one of its threads to compute it and produce a result. A result is then sent to the server which adds it to the repository as unproved (if there is any verification needed) and sends it to other clients in order to verify it.

The idea is to have multi-threaded clients, each of them being connected through one message channel with the server.

FIGURE 3.1: Architecture of the system proposed in the FCDS article [TA17]



The names of all the entities in the system are described below:

3.1 Actors of system

3.1.1 Job owner

A job owner is a person that owns the code and created it or outsourced its creation to any programmer with JavaScript knowledge. A job owner has to have an intuitive interface for creating algorithms. They may need good specification, examples of algorithms that were already implemented on the platform. They also need to have ability to easily add input data for algorithms. Job owner may need to change a configuration data for algorithms in any moment. There should also be a possibility to restart all workers that are running particular algorithm if the configuration changes. A platform has solutions for all of these problems, which are explained in chapter 4.

3.1.2 Executor

An Executor is a sandboxed thread that is running a computation script (algorithm's code). Executor's job is only to compute the code (optionally using input data) or to verify someone else's results. After computation or verification Executor sends the message with a result and does not worry about how it is transmitted and if it is eventually saved into the database. Executor can receive 3 types of messages:
-about any change to algorithm's configuration
-about data to be computed
-about data to be verified
For each client there should be maximum of n Executors, where n is a total number of logical CPU cores. Otherwise the computations will be slower due to the need of switching the CPU context.

3.1.3 Plantation

Plantation is simply client's browser. All running threads (Executors) of one client are one Plantation. The server should see client as a Plantation, but should be able to send message directly to a particular Executor. Plantation keeps connection with the server and sends the results received from Executors. It also receives input data, configuration and data to be verified and assigns them to Executors.

Any configuration change is sent by a Plantation to all of the Executors. The data that needs to be verified is sent to an random Executor of random Plantation (that is not the same Plantation that produced the result). The Executor of input data on the other hand is specified by the server and is only passed forward by the Plantation. If a Plantation disconnects from the server, all data assigned to it should be reassigned to any of other Plantations. This entity is assigned an unique ID and recognized by IP address.

3.1.4 User

User is an actual volunteer in the system. Sometimes they enter the website with a purpose of doing computations, but mostly they will use the website as normal and just do the computations in background. It is up to Recruiter if they want to ask users to take part in computations - it is configurable in parameters of the script tag in HTML. User may want to receive something valuable from taking part in computing. The system needs to have a way to reward its users for their resource usage. The rewarding is explained in chapter 5.

3.1.5 Recruiter

A Recruiter's role is to find the users that can do computations and allow them to connect to the platform. The details of recruiting is further explained in 'Servers' section.

3.1.6 Job

The whole configuration of the algorithm together with the code are named a Job. The Job consists of: a code for the algorithm, a static input data for the algorithm, dynamic input data (optional) and the configuration which includes: should the last result be included in the onConfig message, should the Workers be restarted after any change in the Job, is there any dynamic (additional) input data and how many plantations must verify the result to approve it (redundancyFactor).
Lets take into consideration the Monte Carlo PI computation - such Job requires only the radius length and number of total iterations. This data is be the same for every Executor, so it has to be provided in configuration data. There is no

need to provide last result in the configuration and there is no need to provide any additional data that would be divided between Executors. It is also difficult to verify such result because it is nondeterministic. The only verification that could be used is to check if the proposed result (points that were inside the circle and the total number of points) is giving approximate result of 3.14. Lets take another algorithm into consideration - a brute force sequence generating. In such algorithm every Executor should start from different number to generate sequence, so the starting number would be the additional input data (dynamic input data), and has to be properly divided by Job Owner. Such algorithm may require last approved result in the configuration in order to compare the local result with the global one. Also it rather requires the verification of the result, so if any Executor proposes a value, the verification should check if the proposed value indeed gives the best global result.
The Scheduler can schedule many Jobs at once. It has to be taken into consideration that too many Jobs can slow down the Schedulers speed or exceed its memory limit (because of caching the input data). It also possible to easily create one Scheduler per Job in order to increase performance.

3.1.7 Task

The Job is only the specification of algorithm, what is actually executed is a Task. A Task is a part of the Job with particular input data. If there is only static data in the Job, then every user receives the same task to compute. The total number of Tasks depends on the number of input data rows in the database. For instance, let's assume that there is a Job to find a string that produced a particular known hash, assuming that the string has a length of 5. The algorithm would check every possible sequence of the word with 5 letters, hash it, and compare with the hash that was in the static input data, it can be done as following: The input data that Executor receives is only the first letter of the sequence (so one Executor starts from AAAAA and ends with AZZZZ, in parallel another starts with BAAAA and so on, until there will be all sequences search up to ZZZZZ). That means that there will be as many tasks as the letters - twenty six. Creating a task with 6 letters to check would be too long for a single website visit, but it could be possible to divide such Job and provide not one, but two first letters as input. That would mean that

one executor would start from AAAAAA and another would start from ABAAAA. Such input data in the database would consists of 676 rows (26 letters * 26 letters) and the Job would be done without the need for the users to stay longer on the website.

3.2 Servers

3.2.1 Directory service

The code and the results are stored in a repository which in the platform is called Directory service. The directory service is also used to keep input data and configuration. Natural approach for directory service is to be a database. The input data is taken from directory service by a server, then sent do the Plantation which passes it to Executors. The results are sent exactly the other way round - they go to the Directory service immediately - but their status of approval can be set depending on application configuration. If the verification is not needed - the result is automatically approved. If the verification (redundancy factor) is set, then the approval status is set after the result will become approved or rejected by as many nodes as the Job owner decides. The code can be changed at any moment by updating the application configuration in directory service.

3.2.2 Scheduler

The Scheduler does the main logic in the platform. Its main job is to schedule the work between Executors. It manages how the data flows, keeps the configuration synchronized with both Plantations and directory service. It sends verification requests if needed. For Job Owners - the Scheduler provides endpoints for adding input data or adding/changing configuration for their jobs. Scheduler also provides endpoints for getting the results, which can be public accessible depending on job configuration. This server is stateless, nevertheless it caches the input data to be sent for performance improvement. It is able to schedule more than one job. Mostly it connects Plantation to run a job specified by a Recruiter, but if the job is not specified - it connects to a random job that it runs.

3.2.3 Recruiter

A recruiter is a website that has a special script included - for connecting the browser with the Scheduler. Any website in the Internet can be a Recruiter and it only requires it's owner to add a <script src='...'>to the HTML. The script is configurable and allows Recruiter to pick a domain or IP address of the particular Scheduler. There may be more than one Recruiter connected to the same Scheduler. The Recruiter can optionally specify which job does it want the visitors to be computing. The more number of Recruiters the Scheduler has - the bigger is its computing power.

3.3 Solutions supporting computation

3.3.1 Browser plugin

In an initial version of PovoCop (described in Bachelor's enginering thesis), the computations were performed only when a user entered the website that was connected to the scheduling server. He or she can be on any other tab as well - and browser will still be computing). Sometimes it will be more convenient for users to have a way to compute without the need of logging into the website. The solution for this kind of need is a browser plugin. Such plugin was created and works as specified above - it only requires user to once install it and then whenever the user opens the browser - the computations are performed. This approach is similar to classic voluntary computing - it requires from the user to download something. But sometimes this is not a problem - if we imagine a big corporation with hundreds of employees - their IT specialist could install a plugin in their browsers and the company would gain a significant computation power to use at a daily basis. The same thing could happen to the university - if the computers in laboratories had the plugin installed - hundreds of executors would be available for this university's researchers.

3.3.2 Help from Scheduler server

The scheduling server may not have much work to do in some periods e.g. at the middle of the night. Also, the scheduling server runs only on a single thread and sometimes a machine has more idle threads which are not used. The Scheduler can act like one of the executors and create child processes that will help with computations. There are two ways to fulfill this concept. Scheduler can use Node.js technology (JavaScript on the server side) and run the same algorithms that were produced for the browser. It will directly pass the data to the workers. There is even a simpler way - it will connect to one of the Recruiters (websites) with a headless browser (headless means with no GUI - because in most cases applications run on the servers with no GUI). It is up to the Scheduler administrator to set up this approach.

3.3.3 Adaptive scheduling

Some of the plantations may be much more efficient despite having same number of executors than other. The average time of clients are spending on the web pages varies and is very important from the Scheduler's point of view - the longer they stay - the more computation they will do. System keeps track of users average time of connection to Scheduler - and may be more likely to send computation data to the ones that usually stay longer. System also checks how does the time of particular computations compare to the the average time of computation for the particular job - and detects which users have better resources. They may be better for doing the computations rather than doing verification (which are usually much quicker). Tracking user's performance was not found in any of other BBVC solutions, these solutions were constantly sending equal amount of data.

Chapter 4

Technologies

The system was built with several different technologies on front-end, back-end and on the system layer. Some of the technologies were created less than 6 years before the creation of this thesis which shows that 10 or 15 years ago such system would look much different, would certainly be not so efficient and it's creation would take much more time that it took in the year 2017.

4.1 Front-end technologies

4.1.1 Browser

Browser in the beginning of the web was meant to be a tool to browse content. The JavaScript was later added due to a demand for some kind of logic or animations. Nowadays - browsers are powerful and can stream live videos, render 3D graphics, store huge amount of data or communicate in real time with other computers. Browsers are now often replacing desktop application - you are able to run Skype, write emails or browse your own files that you don't even need to have on your hard drive.

Browser in our architecture is a computing platform. All of the modern browsers are able to run the computations. Older browsers may have problems with HTML5 features such as WebSockets or WebWorkers, which are crucial in this platform. Browsers are pre-installed applications in almost all of the operating systems - that fulfills all the software requirements to create a plantation. What makes browser even a better platform is the safety of its usage. There are several mechanisms

created by browser creators that make it almost impossible for websites to hack the user or get any files from their hard drive without their permission. There are also mechanisms in browsers to detect that a script which is computing is not ending. The tab throws an error and doesn't freeze the whole browser or system.

4.1.2 JavaScript

JavaScript is a scripting computing language. Its syntax is similar to C or Java so understanding the code or even writing own code in JavaScript would be easier for the developers that already created code in above mentioned languages. JavaScript is dynamically typed and has many features of functional languages. This is the only language to use the platform - the basic knowledge of it is required to write algorithms or easily create input data.

4.1.3 WebWorkers

WebWorkers is a technology that allows creating independent threads in the browser. Their biggest advantage is being absolutely independent from the main script and not stopping its event listeners and computations. Creating more WebWorkers than number of CPU cores of the computer is possible, but in case of computations it is not efficient. Every WebWorker has a JavaScript code attached, which is passed to them in the moment of creation. They communicate with the main thread with messages and can receive data in any of JavaScript formats.

4.2 Back-end technologies

4.2.1 Node.js server

Job scheduler is written in JavaScript using a technology called Node.js. The technology allowed JavaScript developers to write HTTPS servers in the same language that is used to build front-end. This is the most popular technology to work with WebSockets on the server side. It also allows to write APIs very fast and exactly control the route of the request from the receiving it - to responding. Such control was crucial in creating this platform, because the data should be cached in a custom

way, the results should be verified immediately after receiving them. There should be data reassignment as well. Node.js seemed to be a perfect choice for such needs. If somebody was writing algorithm for the platform and would need something that the platform does not provide - thanks to the same language of the algorithms and Scheduler - they can change the Scheduler's code by themselves and adapt it to their needs.

4.2.2 Database as a repository

The repository that was explained in previous chapters was decided to be a database. The data is strictly related (e.g. input data is related with the results) so the database needed to be SQL database. The choice was PostgreSQL because of its popularity, performance and the fact that there is no need to pay for it. Node.js is connecting to PostgreSQL with Sequelize library as an Object Relational Mapping (ORM) approach. The other important reason why the author chose PostgreSQL is that it has a JSON data type. JSON data type not only validates the data to be a valid JSON object, but also allows access to nested objects inside the JSON in database queries. The results in some algorithms can be complex - it may not be a single integer or string. It can be an array or many arrays. This make it impossible to store as a particular field type - it would not be generic that way. An alternative could be a String-based type like 'VARCHAR', but it would require parsing the string to JSON every time to work with the variables in algorithms. It would also not be possible to query the contents of the results as they were variables.
An example of the query of JSON type field is below:

```
select sum((result->'results'->>'points_inside')::int) from results;
```

with the following structure of results field:

```
{
   "results" : {
      points_inside: 286800,
      points_total: 365411,
```

```
    }
}
```

In some algorithms this may be an easy way to aggregate the results. One can also directly insert some math to the queries and get the ultimate result. The example of such query is shown in chapter 6.

4.2.3 WebSockets

The WebSockets is a technology that is available in browser since the year 2011. It was a dramatic change in Web applications because it allowed a constant communication with server and the client. Servers started to be able to send a message to the connected browser in any time. Before this technoogy - to achieve something similar web developers had to use hacks like e.g. long pooling - and periodically send a request to server to allow it to respond. There is one more thing useful in WebSockets technology. It sends the message without unnecessary HTTP headers. It's more a TCP socket connection than a HTTP request. The most important feature of this technology is a detection when user had stopped the connection due to tab/browser closing or Internet connection/power loss. This is performed by a library called socket.io that is an enhancement of WebSockets capabilities.

4.2.4 REST interface

The Scheduler provides an API for bulk inserting the input data for jobs. Job owner can also browse results with the API. The API also has a possibility to create and update the configuration of the jobs. The API is based on REST protocol. The HTTP request is perform with either GET, POST, PUT or DELETE methods, containing a JSON object inside the request body. The request is send to a particular URL, if the method concern management of data, then the URL starts with
/manage/
and then - depending on management of config or input data it may continue as:
/manage/data/
or

/manage/config
After the last backslash there should be the id or name of the resource - in the case of the platform it is the job name.

4.2.5 JSON Web Token

A platform has to be secured in order to allow many Job owners to create or update the configuration or input data, without having ability to access each other's data. One of the most popular technologies to achieve security are JSON Web Tokens (JWT). If the user provides correct credentials - server generates a token for that user and sends a response to them with the token. The token is signed with private key by the server and has an expiration date. User now can now access restricted URL of the application. JWT can store small piece of information inside it.
Its structure consist of three encoded objects. First of them is the name of the algorithm to encode the token. The second one is the data, usually the information about the user that is used on the server side. It is also accessible without the need of having the public key. The public key however can be used to verify if the token was signed by the server and that the token has not been changed the user.
The JWT is also used in PovoCop as a proof of work. When client computes the results and is constantly connected to the server - the server gives points to the user and sends a token to them. With every new points the Scheduler updates the token. With the token - user has a proof how much they were computing - and may use such token in other applications in order to get some extra bonuses. JWT is also used in order to provide basic security to the platform.

4.3 System technologies

4.3.1 Docker technology

A modern approach for deploying web applications is to create a container image of them. The most popular container technology is Docker, and a process of putting an application into Docker container is in other words - 'dockerizing'. By downloading a Docker image it is easy for system operators to set up an application on their own environments. This works similar to creating a virtual machine with

certain operating system and pre installed dependencies - such as system libraries, programming language runtime (both installed in particular version) as well as the pre-created users with required capabilities. This not only make environment setup faster, but also ensures that if the images works in one operating system, it will also work in any other distribution - no matter if it's Windows, system based on RedHat or system based on Debian.

PovoCop is complex platform consisting of distributed system of 2 application working together. These application include Postgres database and scheduling server. Assuming that we have two Docker images created for these applications, the containers could be built and connected together by a docker-based service called docker-compose.

Programmer can set the configuration of the Scheduler while starting the platform. This can be done by setting environment variables in the 'docker run' command. The configuration and the names of variables will be explained further in chapter 6.

4.4 Solutions supporting safety and reliability

4.4.1 Database replication

If the Directory Service fails or loses connection with the Scheduler - all computations would be done in vain. To avoid such situation there must be a mechanism of the database replication. Other problem which may occur is the loss of data on the Directory Service due to e.g. disk damage. The database replication would also fix this problem and save the data in two databases at once. This approach means that database is no longer a single point of failure. The replication is provided by the technology called pgPool. It creates a cluster from multiple postgreSQL and is seen from the outside as one database. This solution however does not come as default with the platform due to the need of cluster configuration. It is a job of scheduler administrator to provide database replication using one of the pgPool (or similar solutions) tutorials. The databases should be set up on different machines to provide safety and reliability.

4.4.2 Failover

If the databases are replicated, the Scheduler becomes a single point of failure of the system. The computations could be done but there would be no server to send the results. To avoid such situation there could be more than one Scheduler available. If the plantation is unable to connect to the Scheduler, it could try connecting to other one which IP is provided in the script's attributes as an additional one. This may take effects if the connection suddenly stops or if the client enters website but cannot establish the connection. The list of additional schedulers is provided in src tag in the html of Recruiter. The best approach is for additional schedulers to be on different machines (perhaps in different data centers).

4.4.3 Redundancy

In order to discard byzantine failures or malicious users the work can be sent to more than one Executor. The platform uses a slightly different way to achieve safety - it sends the verification request to few Executors from different plantations and they decide whether the result is valid or not (a separate function has to be provided in the algorithms code). By doing the redundancy in such way one can avoid doing all the computation from the scratch by another Executor. Redundancy is also used as a solution to provide fault tolerance, because even if the process computes an invalid value, it won't be eventually accepted. In most cases it will only verify if the proposed data indeed leads to the proper algorithms outcome. The data given to the clients to computed will be rescheduled whatever happens to the client - so any possible faults connected to electricity, internet connection loss or leaving the website in any moment are tolerated by the system.

Chapter 5

System usage

This chapter shows the usage of the PovoCop platform from the Job owner perspective, as well as from the User perspective. Reader can also find information about the data flow in the platform and the details of data scheduling algorithm.

5.1 Code writing

5.1.1 Setting up the scheduler and database

Following script has to be executed in bash, in order to set up a scheduler and the database. There is an option to change the listening port replacing the left value of -p option (9000 in Scheduler and 5432 in postgres).

```
1   #!/bin/bash
2   docker run -d --name postgres -p 5432:5432 --restart=always -e
    POSTGRES_DB=povocop_1 -e POSTGRES_PASSWORD=povocop -v /data/
    postgresql/data:/var/lib/postgresql/data postgres:9.4
3
4   docker run -dti --name povocop -v ~/logs:/var/logs -p 9000:9000
    \
5  -e "dbName=povocop_1" \
6  -e "dbUser=postgres" \
7  -e "dbPassword=povocop" \
8  -e "dbHost=172.17.0.2" \
9  -e "dbLoggings=true" \
10 -e "secretToSignJWT=abXcdEF96412" \
11 pjesek/povocop:scheduler
```

5.1.2 Job configuration

The Job has some configuration included. Such configuration tells the Scheduler how to behave and tells the front-end script how to handle messages. This is how configuration data looks like:

```
{
   "configurationData": JSON,
   "restartAllWorkersOnConfigChange": Boolean,
   "redundancyFactor": Integer,
   "provideLastResultInConfig": Boolean,
   "code": String,
   "includesInputData": Boolean
}
```

The **configurationData** is a JSON object that includes the static data connected to the algorithm. This data can change in the life cycle of the computations. For algorithms that do not have different input data scheduled for each Executor - this could be the only input data. For instance - it could consist of iteration count and radius length for PI computation with Monte Carlo method.
If the computation data changes - it could stop the current Executors doing what they do - and replace them with new ones using the new configuration. Other way is to let the Executors finish their tasks and then send them the 'onConfig' message with new configuration. Depending on the approach - it is configurable with Boolean type **restartAllWorkersOnConfigChange**. The important thing is, that due to JavaScript being single threaded - Executor will handle the onConfig operation only after it stops computing the currently executed code and everything that is on its stack. If there is a need to immediately handle this event - then the restartAllWorkersOnConfigChange should be set. This option is also good for Job Owners while creating the algorithms, because it will restart the workers with every change in the code.

The **redundancyFactor** field is important for almost all algorithms. It allows the results to be automatically verified by other Plantations. This field is an Integer type and tells the system how many plantations must verify the result to make its status 'approved'. If there is no redundancy factor - then every result is inserted into the database with already approved status. The higher redundancy factor is - the more trustful the Job Owner can be about the approved result. Although setting the redundancy factor to high integer can slow down the computation speed because the Executors will spend too much time on verifying and too little time on computing. On the other hand, redundancy factor higher than 2 is crucial for algorithms where the computation depends on last approved result. However the redundancy factor of 1 is not recommended because anyone could lead to reject it even if it is legit. Approving an nonlegit result in such way would be less possible because it would require two different plantations - first to create invalid result and than second to approve this invalid result on purpose. It is suggested for redundancyFactor to be from 2 to 4.
It may be useful in some algorithms to provide the last approved result (by setting **provideLastResultInConfig**). For instance - it would be useful if there is a need to compare the local result with the global result and post it only if the local result is better. This may be used in e.g. Traveling Salesman algorithm - to send the local best combination of travel sequence only if it is better than any other Executor's result. It is important to depend only on reliable results, therefore it is recommended that if this field is active, the redundancy factor should be provided as well - having at least value of 2 (if the results are rarely inserted to DB it could be even 4).
The **code** is a JavaScript code of the algorithm for the Executor to compute. The code is saved as string type and then the Executor will create the 'in memory' JavaScript file from the code and use it in the constructor's first parameter of WebWorker. Code should have the onConfig, onData functions implemented. If there is a redundancy factor, there must also be a validate function provided in the code that returns either true or false statements. If the result is computed, it only requires to send it to the main thread as following:

self.postMessage(result);
the actual data in result variable will be saved as JSON in the database. The result

will also be sent in the same structure to verifying Executors, together with the input data that produced this result.

The last field is **includesInputData**, which tells the Scheduler if there is any input data in the database to schedule. If this field is set, then Scheduler will send data as onData event to Executors. If this field is not set, then the algorithm must handle onConfig method which will be invoked in two situations: just after establishing the connection and after sending the results. In the meantime, the Executor can receive verify request, so in this way - it will be able to verify results between computations. The task has to end eventually, because the browsers do not allow the JavaScript functions to run to long - in order to overcome never ending scripts that would freeze the browser.

5.1.3 Code submission

Every job requires a code to be executed. Platform was meant in such way, that the programmer only needs to focus on the algorithm and input data structure, not about any message passing or threads creation. The outcome of this approach is that the threads will be created automatically depending on user's resources (number of CPU cores). The threads will execute the code and produce results. Sending the results to the main thread will eventually insert them into the DB. That means that it only requires algorithm to end with sending a message in order to save the result in DB - everything that happens afterwards is done by the platform. There are 3 types of messages that can be handled - onConfig, onData and verify. These 3 events are believed to be sufficient for most of algorithms.
The 'onConfig' message is sent whenever the configuration of the application changes - or if new approved result appears. It can be either a change in the code, or in the configuration data, or when the last approved result changes. The onData message only applies to algorithms that have input data provided - the data that is sent from Scheduler appears in Executors scope with the onData event. Verify event on the other hand is sent in algorithms where there is a redundancy factor provided. These algorithms must have the verify function implemented. It is believed that for many algorithms the verification would be much quicker than computation.

For instance - in Traveling salesman problem, the verify function would check if for the given path - the cost of traveling is the same as was provided in the result. That means that if the solution is found, the verify function usually checks only if for the computed solution - the output is valid.
In less common cases - it would require to compute the whole algorithms for given input data in order to validate the result, which would mean that the computation power would fall *redundancyFactor+1* times - for redundancy factor equal to 2 instead of having 100% total speed - the cluster would have 33% total speed - in comparison to the approach with no verification. Nevertheless, in all of produced algorithms that are explained in chapter 6 - such demanding verification was not needed.
If any library helping with the operation on arrays or strings is needed, the WebWorker can import any library from any URL as following:
importScripts('http://cdn.example.com/script.js');
It could, for instance, be popular 'Underscore' library that has many helpful methods as removing duplicate elements from array, finding the same elements in two arrays or deep copying the object.

5.1.4 Code validation

Depending on the use cases of Scheduler - its administrator could allow users to write their own codes. Despite that a code runs in the sandbox (and does not allow WebWorker to change the URL, HTML or access the cookies) - such approach could lead to some dangerous situations. For example - someone could create a botnet instead of computing algorithms - and just send a big amount of requests to different domains. Other bad approach would be if someone created a Job that sends a lot of requests and simply overloads the database causing the lack of disk space. It would be safe to first verify the submitted code before sending it to users - but it has to be done manually by the owner of Scheduler. In the current implementation the submitted code is immediately available to the users without any verification.

5.1.5 Input data submission

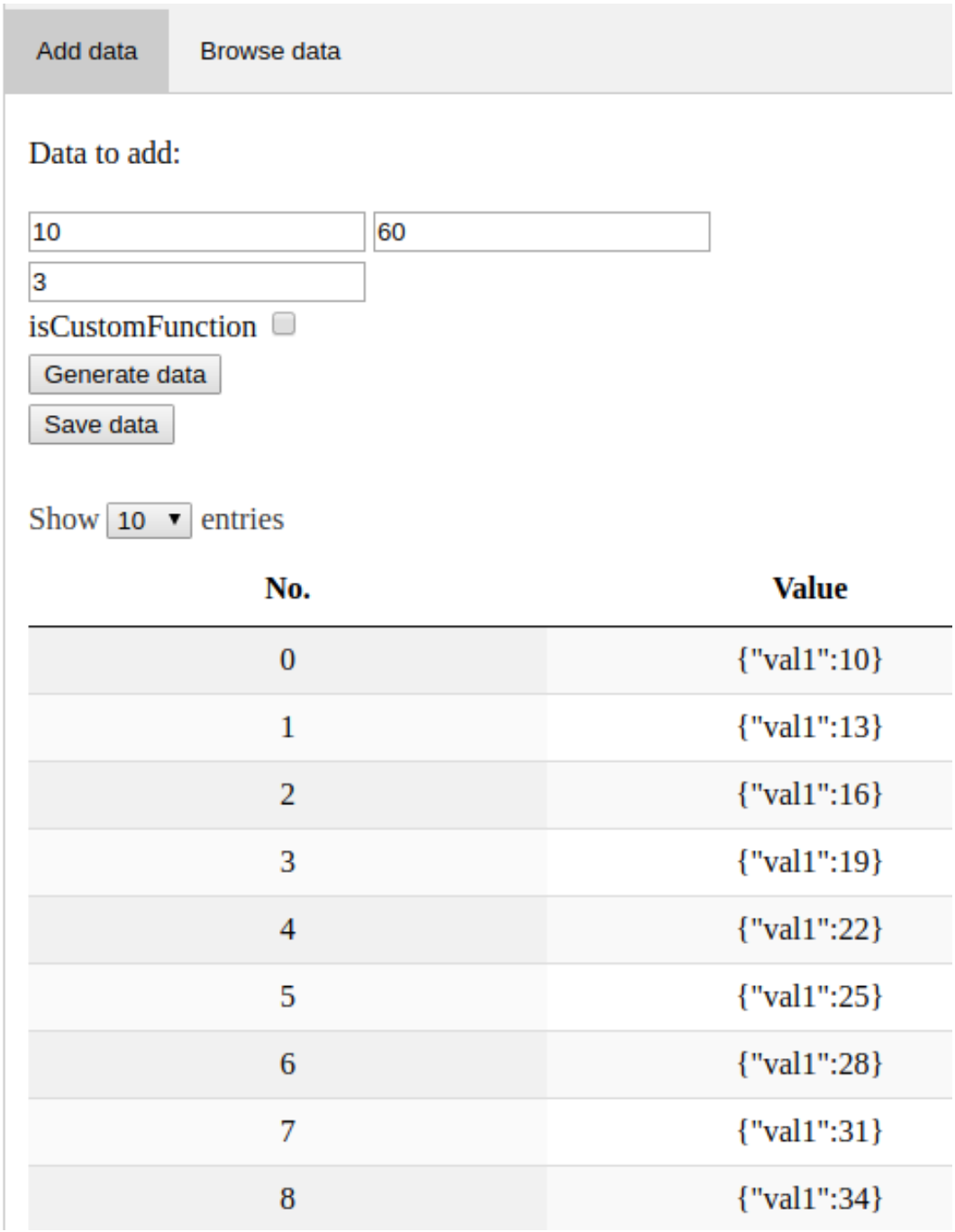
The platform assumes that the input data is the simple data e.g. sequential integers sent to different Executors. The data can be more complex because it is stored in the database a JSON field, so it can have many variables as input for the algorithms. It is recommended that the single input row is less than 1MB - to not overload the network. If the platform is used in LAN network and only the internal computers are computing - then input data can be much larger and the performance should still be satisfying. The total data should be split into chunks for each Executer to compute. Such chunks should be properly divided, so that the Executor can handle it and produce the result before leaving the website. If the Executor leaves the website, the data will of course be reassigned to the Executor of some other plantation, but the Job Owner should also be aware of the browser limits of computation time of single function.

The input data generator can also use a custom function to generate the data. This function takes 3 parameters: start number, end number and a factor. It only requires to do slight changes in the base function to generate wanted input data sequence. The base function is following:

```
1    function (start,end,factor){
2        var iterator=0;
3        while(start+iterator < end){
4            dataSet.push({val1: (start+iterator)*factor})
5            iterator+=1;
6        }
7    }
8
```

The platform provides a GUI for adding the input data. Such GUI looks like following:

FIGURE 5.1: User interface for creating input data in the platform



5.2 Voluntary computation

5.2.1 Test of resources

5.2.2 Code fetching

The user enters the website and establishes a connection between the browser and the Scheduler. Depending on the parameters in <script src="">on the website - if the Job name is specified - then the WebSockets connect to particular namespace and will only receive input data for that Job. If there is no specification of Job - the browser will connect to a random namespace.
The first message received by the Plantation is the configuration. The code of the algorithm and static data for the algorithm are provided in this configuration.
When the plantation receives this message, it creates as many WebWorkers (Executors) as the computer total number of logic CPU cores is (which was calculated by the test explained in the previous subsection). The WebWorkers are initialized with the code as their constructor's first parameter. Depending on if there is also additional input data required - the Executors will wait for Scheduler to send it to them. If there is no input data, the algorithm should be invoked in Executors 'onConfig' function. If the Job Owner will do any changes in the code and if they had set up the option to reset the Executors, then the Plantation will receive another onConfig message and terminate all existing workers. It will create new ones as a replacement and pass them the data that the terminated ones were computing.

5.2.3 Code computing

The event handlers in Executor are: onData (if algorithm relies on the input data) and onConfig (if there is no additional input data) - should invoke a main function with the actual algorithm. Before ending the function, in order to get next input data, the Executor must send a message with a result. If no significant result was found, the Executor must at least send an empty message. Such result will also be verified. The message needs to be send in order to get more input data to compute. In most cases though, there will be some result of the algorithm. The code must also have the verify function implemented, if the Job Owner is willing to have the results

approved by setting redundancy factor. The platform provides some mechanisms to synchronize the data between Executors. If there is a need - the last approved result will be included in the configuration. This option was designed for algorithms that are searching for the input that is producing the best output. Because of the access to last result, the Executors do not have to send local-only best results, but they will be sure that the results are the best from the global perspective.

5.2.4 Result verification

Verify function

The results will only be verified if the redundancy factor had been set. To verify results, there must be an input data provided, together with the result that was proposed for this data. To do a verification, the only thing is to implement a function called 'verify' that takes two arguments: input data and result. Such function should return a boolean value. This function has to determine if for the particular input data the output is the same as was proposed. If everything fits together - the function should return true and if there is inconsistency - return false. It is believed that for many algorithms the verify function will not require much computation.

Verify example

For instance, the Executor was brute forcing or producing random sequences (using genetic algorithms) and found that the his local best sequence is 5, 1, 4, 2, 3 and produces the value of 150. They send such result to the Scheduler and it inserts the result to the database and sends it to other plantations to verify. The verification would just use a function to get a value for such sequence and if it will result in the same number (150), it will give the approval for such result.

Approval and rejecting

If redundancy factor is higher than 0, then it would require sending the verification requests to as many plantation, as the value of this factor is. If received verifications are not equal - then the request is sent to another plantation, whose outcome will decide whether to approve a result or not. If the result is not approved - the reject

process is invoked. Such process runs as following: it deletes a result from the database and remove the assignment from the database for the connected input data and returns it back to the cache. It can also blacklist the IP of the plantation that produced such result and automatically disconnect if such computer wants to connect to the Scheduler every other time. Such blacklist option if not used cautiously could lead to blacklist innocent executors (if the validate function is wrongly implemented and has bugs). The blacklist method is not implemented in the current version, but will certainly be implemented in next releases (being configurable for each Job). As it was previously mentioned - the redundancy factor should not be equal to 1. Any process could then force the disapproval and that will cause the potentially legit result to be removed from the DB. Having more than 50 users connected to the website make it unlikely for a legit result to get two disapprovals, or the other way round, so in most cases the redundancy factor of 2 will be sufficient. Verification is only used to determine if the result was not sent by Executor with byzantine failure. If the plantations are local computers and the Job Owner trusts users - there is no point of using it and setting the redundancy factor.

5.2.5 Result persistence

Every result is inserted into the database with the following structure:
username: STRING,
appName: STRING,
ip: STRING,
result: JSON,
uuid: UUID,
approved: BOOLEAN

The **username** is the name of the person that created the result. It could be used for rewarding people for finding the best results.

Other field is **appName** which is basically a Job name that this result is connected to (the results for all the Jobs are in the same table, grouped by this field).

The **IP** is the IP address of the plantation that produced the result, it is used in order to send verification to plantations with different IP than the one that proposed the result.

The **result** is a JSON field, so it (the same like the input data) can be generic for the algorithms - can consist of different data types and multiple variables.

The **uuid** is a unique id for every result. It is used to connect the result that was sent to the Executors for verification with the result in the database. That way, the executor will not sent the approval status for any result that was not sent to him - because it will be unable to guess the unique id.

The result **approved** status is either false (not yet approved) or true (approved). If the result is rejected it is deleted from DB, so there are no rejected objects in the results table.

5.2.6 Result browsing

The results can be easily browsed with the GUI provided on the platform under the endpoint '/manager/browse/:jobName'. They can also be fetched as JSON on the endpoint GET '/api/browse/:jobName' The browsing of the results is configurable in Job configuration and is set to public by default (to encourage users to take part in computations). The IP address is not provided in the fetched results due to security reasons, but the username is available.

5.2.7 Getting rewarded for computation

How to get results

The users are rewarded for their contribution by getting the points. With every 2 minutes of connection - if the user sent at least one result - they get the points. The more Executors they are using, the bigger amount of points they get. If they have e.g. 4 CPU cores, then maximum of 4 points will be added whenever 2 minute pass - if all the Executors send the result in the meanwhile.

A token including the proof of work

The points are sent to users in a form of a signed token (using the JSON Web Token technology). The token is encrypted with the RS256 algorithm, which consists of private and public key. The private key is used to sign it and a public key used to verify the signature - but the public key is not required to see the contents of the token. The content of the token is the total sum of points earned, the username of the user, the unique id and the website's domain name of the Recruiter that allowed the user be a volunteer.

Where to access the token

The token is accessible in the cookies of the website where the computations were performed. The name of the cookie that includes the token is 'povocoptoken'. The Recruiter can access it during every request to their server or from the level of JavaScript. That means that the token can be displayed somewhere in the HTML of the website if needed.

What to do with the points

The points are believed to be later exchanged for something more material or to be used when building leaderboards (the username could be used there as well). If such leaderboards would be available in public, then it would awaken interest of the users for taking part in computations. The uuid is used to invalidate the token if the points are used (the application that exchanges the points should have a list of uuids of invalid tokens).

Setting the username

In case of willing to provide the username in the token, the Recruiter has to put user's username in the browser cookies. Such cookie has to be named povocopusername and should be available on the level of JavaScript (without flag http only). If the user already performed some computations without having the username set - in any

moment when the povocopusername cookie appears - the username will be added to the token, without resetting the points or generating new uuid.

Chapter 6

Algorithms implementation and performance tests

There are three algorithms created on the platform in order to proof its capabilities. These algorithms are popular examples of distributed or parallel computing. First of them is PI computing using the Monte Carlo method. The second one is finding a word that produces particular hash. The last one is the genetic algorithms for the traveling salesman problem.

6.1 Monte Carlo PI simulation

6.1.1 Algorithm description

The Monte Carlo method is a technique of simulations in mostly physical and mathematical problems. It is used to estimate certain values using repeated random sampling. One of the most popular problems that uses the Monte Carlo method is estimating the value of PI. This algorithm is a classic approach to test Browser based Voluntary Computing platforms. Implementing it in PovoCop platform allows comparison between the Author's master bachelor's thesis. The fact that Monte Carlo Pi computing is possible to be implemented shows how easy it is to write algorithms on the platform.

Knowing, that the area of square is a^2 and area of the circle inside such square would be is $\pi(a/2)^2$, then we can count the proportion of the areas. $\frac{\pi}{4}$ would be equal to the proportion of the circle to the square. So $pi = 4 * \frac{CircleArea}{SquareArea}$ The only thing left

is to randomly generate a point within a square and check if it is inside the circle or not. The equation for circle boundaries is useful for this: $x^2 + y^2 <= r^2$. The more number of points is computed, the more exact is the result.

6.1.2 Configuration

There are two input variables for this algorithm. First one is the maximum iteration count, after which the Task will end. The second one is the radius of the circle. Both of them are the same for every Task, so they are static input data, which means that they should be stored in configuration and no additional input data is needed. There is also no need to provide the last approved result in configuration. The validation would be difficult to do, because the result is non deterministic, so the redundancy factor equals 0 in this case.

6.1.3 Additional Input Data

There is no input data for this algorithm other than static data.

6.1.4 Algorithm's code

```
1  var configuration;
2  function main(){
3   var itEnd = configuration.iterationCount;
4   var radius=configuration.radius;
5   var points_total = 0;
6   var points_inside = 0;
7
8   while (1) {
9     points_total++;x
10    var x = Math.random() * r * 2 - r;
11    var y = Math.random() * r * 2 - r;
12    if (parseFloat((y * y + x * x))  < parseFloat(r * r))
13    {
14          points_inside++;
15    }
16    if (points_total == itEnd){break;}
17  }
```

```
18
19  self.postMessage({
20    results:{
21      'points_inside': points_inside,
22      'points_total' : points_total
23    }
24  });
25 }
26 function onConfig(config,lastResult){
27   configuration = config;
28   main();
29 }
30
```

6.1.5 Validation function

There is no need to provide a validation function because the redundancy factor equals 0. However there could be a validation to check if the proposed result is e.g. between 3.140 and 3.142 (because exact result is know to be 3.141592) - then any faulty results won't be considered while calculating ultimate result. Such validation function would look as following:

```
1     function validate(res){
2         var val = 4*result.points_inside/result.points_total
3         return  val > 3.140 && val  < 3.142
4     }
5
```

6.1.6 Performance

During the survey that was sent to Internet users, they computed a PI value of 3.141541, which is a close result, because the rounded known value of PI is 3.141592. The median of time of filling the survey was 93 seconds and they were 50 answers.

The tests in Bachelor's thesis involved machines set up especially for the test, while real users were the testers in this thesis.

6.1.7 Results cumulation

In order to concatanate the results, one can send a following query to PostgreSQL database:

```
1  select sum((result->'results'->>'points_inside')::int) from
   results;
2  select round(4*sum((result->'results'->>'points_inside')::
   decimal)/sum((result->'results'->>'points_total')::decimal),10)
   from results;
3
```

6.2 Brute force search

6.2.1 Algorithm description

The brute force search is a method for searching a particular value by using all permutations of possible input. This method is easy to distribute between nodes - each of them can compute the subset of permutations. The algorithm created as an example of brute force search is MD5 hash finding. The goal is to find a text that produces a hash given as an input for algorithm.

6.2.2 Configuration

The hash is static between nodes so it will be provided in configuration data. Also the subsets of the permutations for every Executor should be scheduled. The subsets are different for every Task so they are provided as additional input data. The probability that the result would be found by byzantine process (who will hide the fact of finding it) is rather low. But the validation could prevent the processes from claiming of finding the result which in fact does not encode to the provided hash. This means that the redundancy factor must be provided and, for

instance, may be equal to 2. The last approved result may be useful in this case in order to end the algorithm if the result is found.

6.2.3 Additional Input Data

The input data for this algorithms are the three first letters of the sequences to find. The rest 4 letters can be search in full range, because its creation will took only few seconds. Platform provides an easy way to generate such sequence and add it into the database, it only requires to provide a custom function based on the one that provides sequential data. A following function must be pasted to the input data view:

```
function customFunctionCodeTemplate(start,end,factor){
var letters = new Array(26)
.join(',')
.split(',')
.map(function(value,index){
return String.fromCharCode(97+index)
})
.concat(['']);

var iterator=0;
while(start+iterator < end){
 letters.forEach(function(val,idx){
   letters.forEach(function(val2,idx2){
     dataSet.push({
       letter1: letters[iterator],
       letter2: letters[idx],
       letter3: letters[idx2]
     });
   });
 });
 iterator+=1;
}
}

```

Such function will generate a sequences as following [a,a,a],[a,a,b].. [z,z,z]. It will also generate blank values in order to also search through words that are 6 and 5 letters long.

6.2.4 Algorithm's code

```
importScripts('https://cdnjs.cloudflare.com/.../core.js')
importScripts('https://cdnjs.cloudflare.com/.../md5.js')
var configuration;
function main(data){
  var lastIdx=-1;
  var found = false;
  configuration.letters.forEach(function(letter4){
    configuration.letters.forEach(function(letter5,idx){
      configuration.letters.forEach(function(letter6){
        configuration.letters.forEach(function(letter7){
            var word = [data.letter1,data.letter2,data.letter3,
            letter4,letter5,letter6,letter7].join('');
          var hashedWord=CryptoJS.MD5(word).toString()
          if(hashedWord==configuration.hashedValue)
          {
            self.postMessage({
              results: word
            });
            found=true
          }
          });
        });
      });
    });
  if(!found){
    self.postMessage({
      results: 'none'
    });
  }
}

```

6.2.5 Validation function

A function to validate if the result really gives a given hash:

```
1 function validate(result){
2     if(result.results=='none'){return true};
3     return configuration.hashedValue===CryptoJS.MD5(result.results)
      .toString()
4 }
5
```

6.2.6 Performance

One Executor was able to compute one Task in 8 seconds. That means that searching for the 7 letters word (26*26*26 Tasks, where 26 is the length of the alphabet) would take 39 hours for one Executor to do. Assuming that there will be average of 40 plantations with 2 Executors each - such process would take 60 minutes to be done.

6.3 Genetic algorithm

6.3.1 Algorithm description

A genetic algorithms are based on biology. The approach is to treat data structures as chromosomes and constantly build new populations by mutating and crossing them over.

Such method is one of the heuristic approaches for a traveling salesman problem. The problem is to find the closest way between number of cities, knowing the distance between all the pairs. Firstly, the algorithm produces random sequences of possible ways. Then the sequences with the best ways are crossed over and mutated. Each crossover produces two more 'children' which have the genes from both crossed 'parents'. After that - the process is repeated again with the new population.

The algorithm on the platform will work as described above, so that every Executor will have its own population. The best found way will be send by an Executor and then validated by other plantations. After the proposed way is approved - it will be sent to all the Executors, so they will only search only for better way than already

proposed. Having a last approved result also has one more advantage - it can be added to Executor's population and be crossed over with their best chromosomes. Such crossing may produces good results.

6.3.2 Configuration

The input data for this algorithm is a matrix of distances between cities. There is no dynamic data to be distributed between nodes. The validation is required and can be done by multiple plantations, because it will be a rare situation to find better way then existing one. So the redundancy factor can be set to 3. Also the last approved result is very important to have in the configuration, so it should be set in the configuration of the algorithm.

6.3.3 Additional Input Data

There is no additional input data in this algorithm.

6.3.4 Algorithm's code

```
1  var population;
2  function main(){
3   var matrix = configuration.matrix;
4   var populationCount = configuration.populationCount;
5   population = population || [];
6   population = generateRandomEntities();
7   population = findBestEntities(population)
8   population = mutate(bestGenes);
9   population = crossover(population,matrix)
10
11  setTimeout(main,1);
12 }
13 function onConfig(config,lastResult){
14  configuration = config;
15  main();
16 }
17
```

Please note that when the main function ends, the setTimeout function is invoked. The reason of putting this function in the end of the algorithm is that browsers do not allow the functions to run for too long. Because of setTimeout - the function will finish and 1 millisecond later another iteration of the function will start.
The implementation of particular functions such as mutate or crossover are not a part of this thesis.

6.3.5 Validation function

The validation function is following:

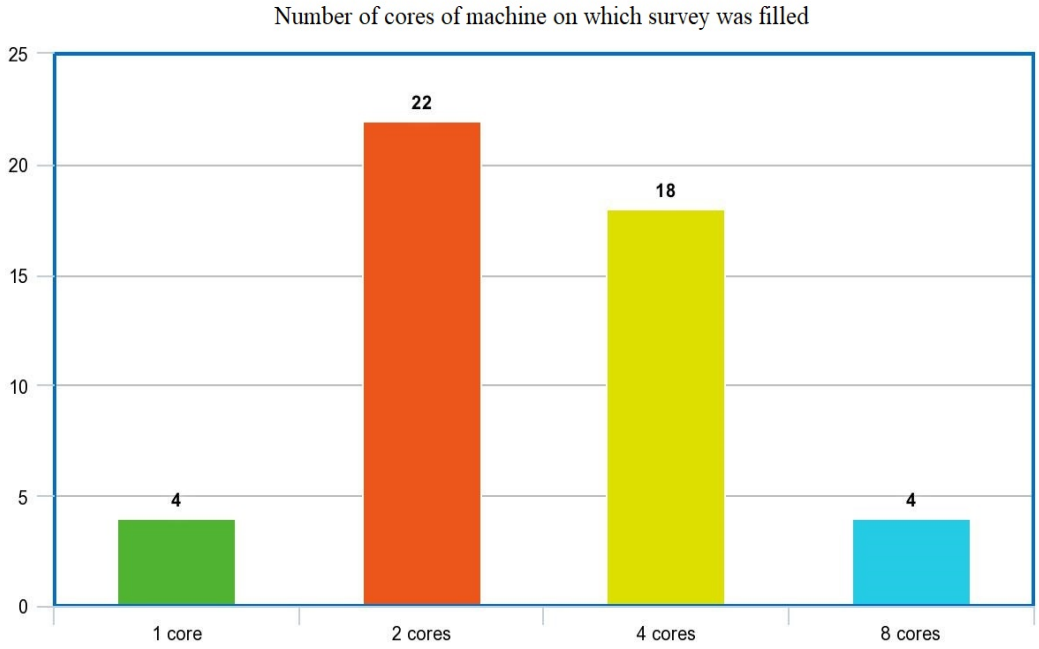
```
1 function verify(result){
2 var cost = getJourneyCost(result.cities)
3     if(cost<bestScore){
4         bestScore = cost
5         return true;
6     }
7     return false;
8 }
9
```

6.4 Survey

The big question of voluntary computing is if the user would like to participate in it. To get to know the thoughts of people there was a survey created for this thesis. We asked 50 people (26 men and 24 women) what they think of voluntary computing.

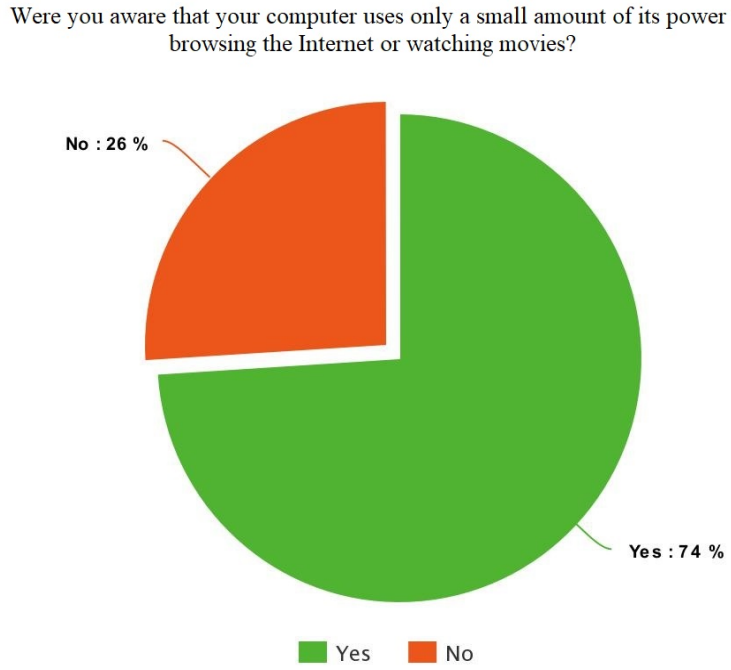
Also - during the survey - there was a script computing PI in the background - so the survey website was connected to the PovoCop platform. Only 4% people that filled the survay had problems with connecting to the platform and did not send any results.
Graph below shows that most users have multiple cores available on their devices. They could use even half of them for browsing Internet and leave the rest for computations.

FIGURE 6.1: Bar chart of the number of cores of the users devices that were participating in survey



6.4.1 Awareness of using only small amount of the power

FIGURE 6.2: Pie chart of awareness of not using whole power of computer

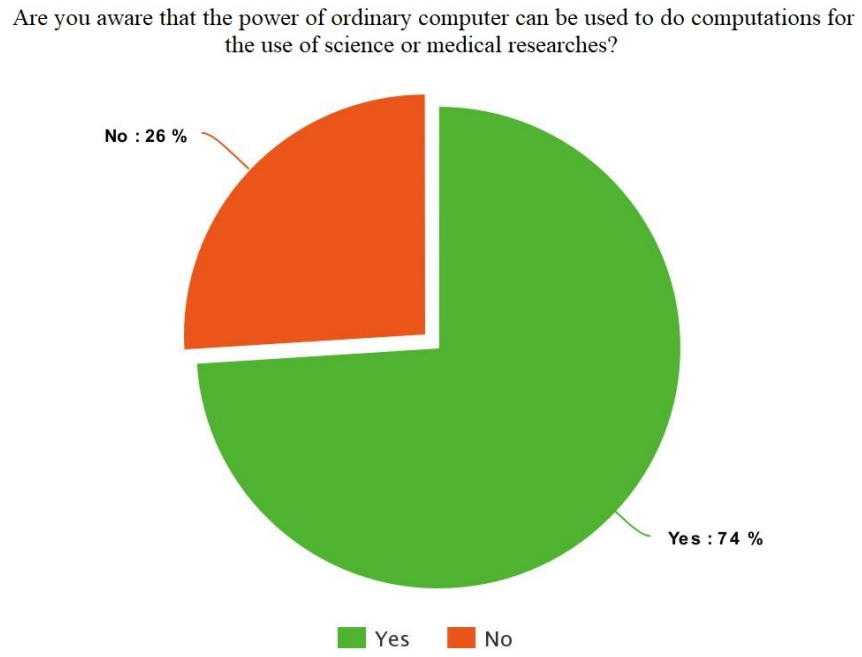


The results show that 3 out of 4 people are aware that their computer's resources are not used in 100%. Still - 24% of surveyed people didn't know the power that their computers have or thought that it is used in 100% for browsing or watching

movies.

6.4.2 Awareness that the power can be used to help science

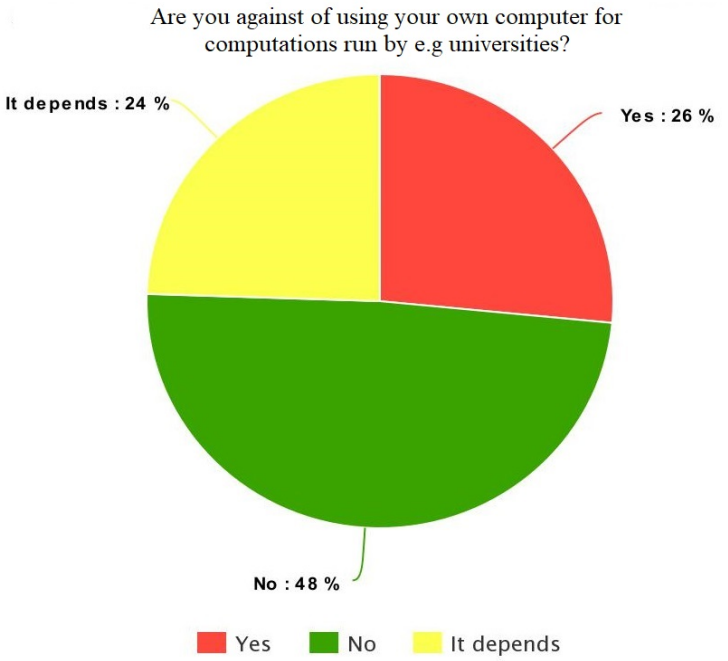
FIGURE 6.3: Pie chart of awareness of being able to help sciences



Most people are aware that even ordinary computers are able to do computations that might help the science.

6.4.3 Own computer used for voluntary computing

FIGURE 6.4: Pie chart of being against of sharing own computer's resources



Half of the surveyed people do not mind if universities used their computing power for their needs. The rest of them is divided in two groups - 26 people do not want their resources to be used in any way. 24% say that it depends and they add that:

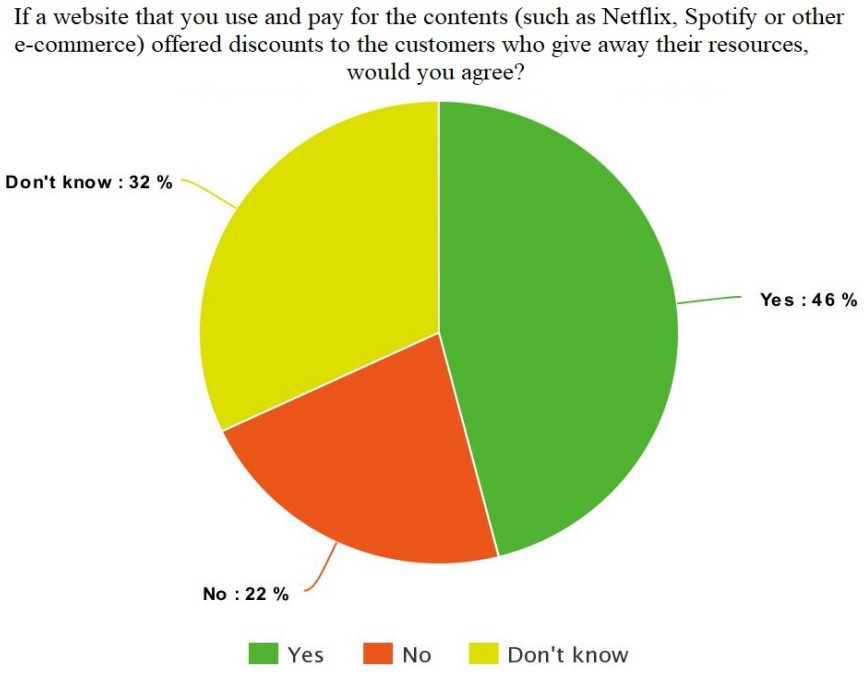
- it depends on what kind of computations that would be
- for what needs they will be used, who will benefit from the computations
- if it will affects their computer speed or
- if the script would take any data from their computer disk

Some people say that they will only allow computations if they knew what Job exactly is their computer doing or if they could somehow manage how much resources they give and be able to stop processing in any moment. There was also a concern about the costs of used electricity and if it will be more than 90% of resources used, then a person would participate only if there was some kind of reward like in Bitcoin.

6.4.4 Discounts from big companies

Only 22% of people wouldn't for sure share their resources if they would have a discount in applications that require payments. Half of the answers were positive,

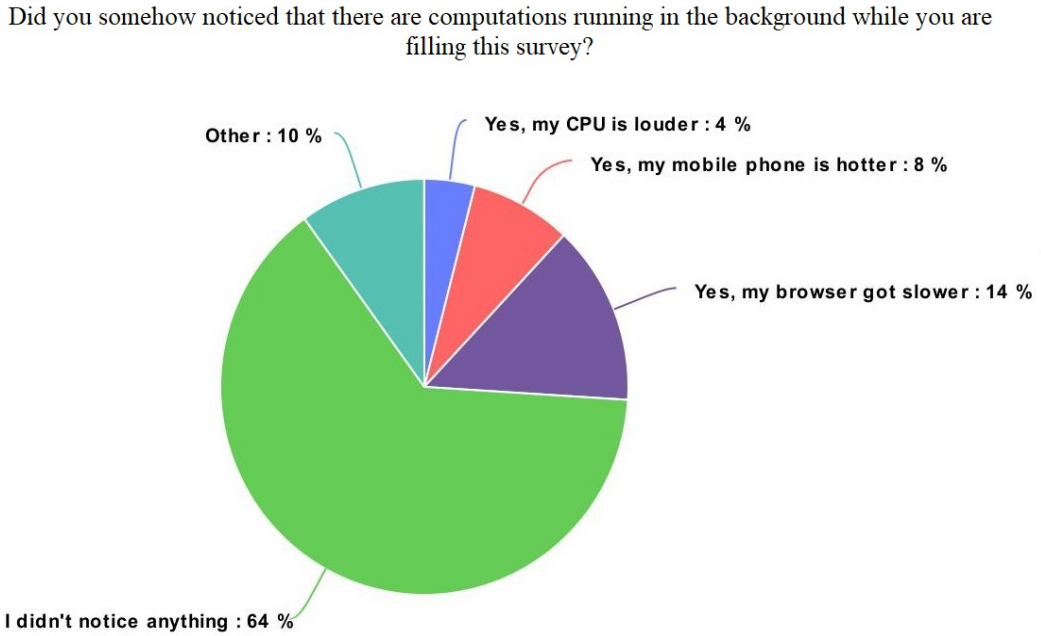
FIGURE 6.5: Pie chart of willing to participate in exchange of discounts



and 32% weren't sure if they would agree on such terms. This is interesting, because big companies spend much money on servers that perform big data analysis which could be outsourced to their client's computers to compute for only a small discount.

6.4.5 Noticing that there are computations

FIGURE 6.6: Pie chart of noticing the computations in the background



During the survey there were some computations performed in the background. People were asked if they noticed the computations by their device behaving differently in any way. Most of them, 64% did not noticed anything strange (78% of them were using mobile phones).

Only 14% of people said that the browser got slower and 8% said that the mobile phone become hotter. This shows that in most cases users will not bother giving away their resources - because they would not even feel the difference. The only problem may be a case of ethics, where solution would be for Recruiters to warn users what kind of computations are performed at the time by their computers.

Chapter 7

Conclusions

7.1 Summary

7.1.1 Contribution of created platform

The platform is a convenient way for programmers to add computations to their website. The creation of the platform required both using the modern technologies and constructing new theoretical solutions. Such concepts make this platform different from known solutions. There are several new features compared to the other solutions.

First feature is an ease to set up the platform. Setting up the backend (the Scheduler and PostgreSQL database) for computations requires only Docker technology installed in order to run it. To set the application up one has to only execute a script provided in this thesis (or only a part of the script, if the database was created manually). The script will start the node.js scheduling server and the database that it connects to.

Another unique approach is an adaptive scheduling. Scheduling server keeps track of user's avarage computing speed and compares it with the current speed whenever user sends the results - and may send less work to this user if its speed has decreased.

There is also a lack of single-point of failure due to failover of both databases and scheduling servers. Databases are replicated so any disk failure will not destroy the results of computation or the input data assignment.

The platform provides an interface to create, browse and change Jobs. There are two main web pages that are available for Job Owners: configuration page and input

data page. Configuration page consists of fields to fill, in order to easily create a Job providing the code of the algorithm. Input data page provides input forms that create sequential input data for algorithms from the range of numbers. There are also two options available: to change the function that generates the data in order to create custom input data and to paste the array of input data in JSON format. Results can be easily browsed and verified by simply providing a number of expected verifications and creating a result validation function. Such result verification provides a redundancy which is a solution for fault-tolerance and byzantine failures. In most cases a website owners would not have their own computations - they can still attach their website to a any other platform and hire the volunteers to it. Any website on the internet can be a part of the platform only by putting a simple 'script' tag into its HTML structure. The last thing that enhances the browser based voluntary computing is a browser plugin. Such plugin allows users to connect to the platform immediately when they open their browser. Using this plugin - there is no need to enter any website and stay there for long. It only requires to have a browser running. This thesis also provides a survey results about what people think of voluntary computing. The outcome of this survey is that people would like to know what exactly is computed on their computers, for whom the results are sent - and if they could decide how many resources are used. If these statements will be fulfilled - people are rather willing to share their resources, especially if they get any rewards for it.

The platform is believed to be used by entities that require a lot of computing power and may not have enough funds or want to lower their expenses. Such entities may consist of researchers at universities, IT industry (especially big data industry) or Bioinformatics industry.

7.1.2 Future of browser computations

Web assembly

Web assembly is a technology that is an enhancement of JavaScript. It allows JavaScript code to be compiled into binary files and then used in the scripts as modules. This approach speeds up a performance of JavaScript since the modules perform as they were written in low level languages like C. The technology is already

available in browsers and at the date of August 2017 is supported by 57% of modern browsers. [was] It is likely to be a boost for BBVC capabilities.

WebRTC

WebRTC is a technology that allows browser to connect to other browser without server being a proxy. This gives many opportunities described below.
Firstly - it allows fast communication between browsers of different machines in the same network. This provides new possibilities for BBVC as well - the server farms could use browser as a computation platform and with high speed of message passing - many network-dependent algorithms could run faster.
Secondly - it decrease the server load, especially when thousands of clients are connected to it at once. Server is only needed to perform a handshake.
WebRTC protocol however has complex API and requires an ICE server for STUN (Session Traversal Utilities for NAT) in order for clients with private IP's to connect to each other.

GPU support

Many modern computing platforms use GPU as computing hardware, which in many cases is much faster than CPU. GPU consists of hundreds of cores, which can handle thousands of threads in parallel (these cores are able to perform only simple math operations). That means that any operations on matrices (addition or multiplication) are very fast. This is not only used by graphic power demanding application (such as games or high quality videos) but also by algorithms (crypto currencies are mined with graphic cards for instance).
The browsers support GPU, but the basic implementation has rather difficult syntax which made it difficult to write algorithms on it. This was the reason why GPU.js project was born - it allows to write GPU accelerated algorithms with JavaScript syntax. On the projects website there is a test that take few seconds and gives back information of how the speed of multiplying two 512 x 512 matrices compare from doing it on CPU and GPU. On the computer with processor intel i7 4710HQ the CPU speed was 0.414s. The computer has two graphic cards, and the speed for the integrated one (Intel HD graphic 4600) was 0.134 (3 times faster than CPU), but for

the second one (NVIDIA GeForce GTX850M) the speed was 0.019s (21 times faster than CPU). The main disadvantage is that the browser freezes for some moment during GPU computations - which did not occur when the JavaScript code had been run in WebWorkers. Perhaps this problem may disappear in the future versions of the browsers.

Bibliography

[ATM17] Danilecki A., Fabisiak T., and Kaszubowski M. Job description language for a browser-based computing platform - a preliminary report. Studies in Computational Intelligence, Advanced Topics in Intelligent Information and Database Systems, vol 710, 2017.

[BJ12] Kuchta J. Balicki J. *Obliczenia rozproszone w systemach komputerowych o architekturze klasy grid*. Wyd. Politechnika Gdanska, 2012.

[FSA+07] Konishi F., Ohki S., Konagaya A., Umestu R., and Ishii M. Rabc: A conceptual design of pervasive infrastructure for browser computing based on ajax technologies. In Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007.

[J.J05] Gareth J.J. Ajax: A new approach to web applications. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, 2005.

[L.F98] Sarmenta L.F.G. Bayanihan: Web-based volunteer computing using java. Worldwide Computing and Its Applications—WWCÁ98 Springer Berlin Heidelberg, 1998.

[nim] Nimiq project. `https://nimiq.com/faq/`. Accessed: 2017-09-09.

[NSON98] Nisan N., London S., Regev O., and Camiel N. Globally distributed computation over the internet - the popcorn project. Proceedings of 18th International Conference on Distributed Computing Systems Amsterdam, 1998.

[OFND97] Christiansen B. O., Cappello P.and Ionescu M. F., M. O. Neary, and Schauser K. E. Wu D. Javelin: Internet-based parallel computing using java. concurrency: Pract. exper., 9, 1997.

[TA17] Fabisiak T. and Danilecki A. Browser-based harnessing of voluntary computing power. Foundations of Computing and Decision Science vol. 42, Poznań, Poland, 2017.

[was] Webassembly support table. `http://caniuse.com/#feat=wasm`. Accessed: 2017-09-09.

List of Figures

Poznan University of Technology

Faculty of Computer Science and Management

Institute of Computer Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
    author = "Tomasz Fabisiak",
    title = "{Large scale computation using voluntary computing power}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2017",
}
```