Capataz: a framework for distributing algorithms via the World Wide Web

Gonzalo J. Martínez and Leonardo Val
Facultad de Ingeniería y Tecnologías, Universidad Católica del Uruguay,
Montevideo, Uruguay, 11600.
gonmarti@correo.ucu.edu.uy
lval@ucu.edu.uy

Abstract

In recent years, some scientists have embraced the distributed computing paradigm. As experiments and simulations demand ever more computing power, coordinating the efforts of many different processors is often the only reasonable resort. We developed an open-source distributed computing framework based on web technologies, and named it *Capataz*. Acting as an HTTP server, web browsers running on many different devices can connect to it to contribute in the execution of distributed algorithms written in Javascript. Capataz takes advantage of architectures with many cores using web workers.
This paper presents an improvement in Capataz' usability and why it was needed. In previous experiments the total time of distributed algorithms proved to be susceptible to changes in the execution time of the jobs. The system now adapts by bundling jobs together if they are too simple. The computational experiment to test the solution is a brute force estimation of pi. The benchmark results show that by bundling jobs, the overall perfomance is greatly increased.

Keywords: distributed algorithms, distributed computing, world wide web, cross-platform, Javascript.

1 Introduction

In 2012 our research at the Catholic University of Uruguay (UCUDAL) demanded to run some really CPU intensive simulations. The computing power requirements exceeded what our budget could afford. Hence we shifted our focus from buying cloud time to using the computers that were already available to us.

In our university there is an important number of computers used by students and employees. Many spend a significant amount of time idle. Harnessing their CPU time has three main issues. How could we make use of the computing power of many different devices and platforms? How could we achieve this without installing any kind of software in those computers? How could we assure the users their computers would not be compromised in any way while running our software?

Almost all end user devices with network connectivity and a CPU powerful enough have at least one brand of web browser available and up to date. Also, modern web browsers have efficient JavaScript interpreters (or compilers) to provide its owner with a decent web experience. JavaScript is a powerful programming language. Finally, running the jobs in the browser does not represent a security threat. Javascript code executing in the browser's environment is greatly restricted. It cannot access the file system, the operating system, and can only establish network connections with the server it was downloaded from.

Based on the concept of volunteer computing, we developed an open source[1] framework that allows a developer to split a given workload into jobs programmed in JavaScript. We called it *Capataz*, after the Spanish word for *foreman*. Jobs are sent through HTTP to the worker computers running a web browser. An HTML page with embedded JavaScript controls all the process, handling the web workers and the AJAX calls. Hence no special software is required to be installed, and an good portability is achieved.

A first version Capataz was presented in [1]. The first tests performed with the system showed that the performance of the distributed process was susceptible to the amount of time each job spent executing in the browsers. In this paper we present a new functionality aimed to mitigate this problem called *job bundling*. We explain why it is necessary, the experiment designed to verify it and the results produced by it.

[1] Sources can be obtained at `http://github.com/LeonardoVal/capataz.js`.

The rest of this paper is structured as follows. The second section introduces the state of the art, going into volunteer computing, and comparing Capataz with similar projects. The third section describes the features that make Javascript in particular and the web in general, suitable for the project. The fourth section describes first tests and the benchmark problem used to validate the design. The fifth section introduces the job bundling feature, the experiments performed to test it and the obtained results. Finally, the sixth section contains our conclusions and final remarks.

2 State of the art

In this section we first present BOINC as the best exemplar of volunteer computing. Afterwards we describe the first attempts to set up distributed algorithms using web technologies. Finally we introduce current projects in this area, among which CrowdProcess is perhaps the most similar to Capataz.

2.1 Volunteer computing

Distributed systems are groups of networked computers working together towards the same goal. Volunteer computing (or public distributed computing) is a type of distributed systems built with computing resources donated by individuals, rather than bought by a single organization. The Berkeley Open Infrastructure for Network Computing's (a.k.a. BOINC) [2] is perhaps the most popular platform to set up this kind of systems. Taken from their own website [2]:

> "Volunteer computing is an arrangement in which people (volunteers) provide computing resources to projects, which use the resources to do distributed computing and/or storage.
>
> - Volunteers are typically members of the general public who own Internet-connected personal computers. Organizations such as schools and businesses may also volunteer the use of their computers.
> - Projects are typically academic (university-based) and do scientific research."

BOINC started in 2002, as an improvement of the SETI@Home project. As of 2014 it hosts multiple projects like the World Community Grid [3], ClimatePrediction.net [4] and SETI@Home [5]. BOINC API is written in C++ and is preferably used with C++ or C. The low level programming allows for better performance tuning, as well as the availability of GPU computing. BOINC does run in many platforms (as of 2014 it includes most personal computer platforms, Android devices and Sony PlayStation). Still, the distributed application must provide a version specifically tailored for the platform. It usually requires recompilation to any specific platform.

From our perspective, BOINC has two major disadvantages. First, in order to contribute to a BOINC project the installation of both the BOINC client and the software for a specific project is required. The user must create an account, and in some cases manually configure the client. Second, writing a performing application in C or C++ is usually more difficult than writing it in a scripting language like Python or Javascript. Such languages provide automatic memory management, among other facilities.

A web based solution will not allow the programmer to handle the client's hardware at such a low level as BOINC does. Despite that, reasonable levels of efficiency can be achieved with Javascript. The enhanced portability and not requiring users to install any specific software are advantages with which BOINC cannot yet compete.

2.2 Early experiments

Running distributed algorithms using the computing power of web browsers is not a new idea. The first attempts can be traced back to 2007, when Boldrin et al. [6] proposed a new approach for distributed computing. Embedding AJAX in web pages, it was possible to distribute work to clients (web browsers) while the users navigate the website. The server was implemented as a Java Servlet and managed problem partitioning, subproblems distribution and results reassembling, communicating with client through the http web protocol.

Although the authors' proposed to exploit web browsers as clients, the idea on how to do it differs from Capataz in a few ways. First, they propose using the client without the user noticing. Our approach is not to embed code in a website in parasitic way. With Capataz the user would have to volunteer their computing power knowingly. Another difference is that Web browsers in 2007 had some shortcomings as a platform

[2] `http://boinc.berkeley.edu/trac/wiki/VolunteerComputing`

for distributed computing. Some of the issues detected back then, were related to Javascript not having multithreading capabilities, something that clearly changed since 2007.

Also, the paper proposes a new architecture, but it does not present or provide a general framework to use that architecture with any kind of problem, which is in the scope of Capataz. Finally, server side Javascript was not an option in 2007, so Capataz differs in that it's possible to use the same code in the server and in the clients, giving the developer the possibility properly tune the solution by deciding at later stages which code runs on the server and which one does on the client.

Capataz started as a project to run optimization algorithms in parallel efficiently [1], as did other experiments before [7] [8] that used web browsers as a distributed computing platform. Both approaches also lacked Javascript multithreading capabilities and the possibility to use modern technologies such as server side Javascript. Even though they ran the algorithms showing promising results, they were specific for the problem they were trying to solve, whereas Capataz evolved to be a general framework to run distributed algorithms of any sort.

Other approaches were tested in [9] based on Java applets or plugins. These are not entirely portable, since not every web browser can run Java applets out of the box.

2.3 Contemporary work

With the advent of new technologies (namely Google's V8, NodeJS, HTML5, etc) new possibilities arose in the development of volunteer computing. Not only the web browsers became a more feasible way to exploit computing power. They became also a very efficient one. Some benchmarks [10] have a worst case scenario where Javascript code is only 6 times slower than native C code.

Now it's not only possible to use the CPU for calculations (like Capataz does), but it's also possible to use GPU. Duda and Dlubacz work with evolutionary computing in web browsers with GPU acceleration [11] shows remarkable results. The calculation of the fitness function is accelerated by up to 50%, and the local search process can be accelerated tenfold. Again, it is a custom development rather than a general framework. Yet they demonstrate a novel approach to accelerate distributed computing by using the GPU.

There are many ways to use the GPU in Javascript but they all require the use of plugins which undermines portability. The most supported solution is WebGL, which is included natively in the latest versions of many web browsers. Although the use of WebGL is going to reduce the portability of the solution, it does mantain the principle of not asking the user to install specific software.

There's been at least one attempt to provide a general framework for volunteer computing on the GPU using WebCL [12]. In 2013, MacWilliam and Cecka presented the CrowdCL framework [13]. CrowdCL not only allows for the distribution of jobs to run WebCL tasks, but also provides a level of abstraction on top of WebCL to provide an easier programming model for the developer. Although the results are not compared to parallel javascript in order to compare it with other distributed computing solutions, and is reliant on a browser plugin, it's a very interesting solution and integration with the WebCL abstraction may provide Capataz with an easy way to develop WebCL solutions.

Using the GPU was not considered for Capataz due to the context in which it was going to be executed in first place: machines without a capable GPU. There are theoretically no limits imposed by the framework about the use of WebCL, WebGL or any other plugin based technology, since Capataz will use web workers as default, but it also provides DOM access if required.

2.3.1 CrowdProcess

Volunteer computing is an area that is growing, not only in the academic environment as shown before, but also on the market. CrowdProcess [14] is a company founded in 2013 by Pedro Gabriel Fonseca, João Pinto Jerónimo and João Abiul Menano. They provide a namesake cloud based solution that is the sole web only commercial volunteer computing development as of 2014.

The company recruits webmasters to embed a specific component into their websites. The company states that this component will take a little of the website users' computational power to run distributed algorithms. Developers wishing to use the distributed platform are provided with a web API, but the actual implementation is kept private.

3 Javascript's suitability

JavaScript is the *de-facto* standard programming language of the world wide web. It is evident considering the changes in the W3C HTML specification. Dating back to 1999, version 4.01 [15] mentions three different options for scripting languages embedded in web pages: JavaScript [16], TCL and Visual Basic Script. As

of July 2014, the W3C HTML 5 Specification [17] working draft only mentions JavaScript as the default language.

Besides its current popularity, Javascript has many advantages as a platform for volunteer computing nowadays. In terms of efficiency, companies like Google, Microsoft, Apple and Opera are interested in having efficient scripting implementation running inside their web browsers. The performance of JavaScript executing in any modern web browser easily competes with any other scripting language like Python or Ruby [18]. Compared with C, Javascript code can be at most 10 times slower [10].

Nevertheless, until recently it was impossible for Javascript programs to spawn parallel processes or threads, in order to take advantage of architectures with many cores [6]. A change came in 2010 with the addition of the web workers API of the WHATWG HTML Living Standard [19] (chapter 9 "Web workers"). Web workers are an explicit, standard and safe way to run many threads of Javascript code in parallel. Hence web browsers can exploit architecture with multiple processors.

JavaScript also makes transporting code through a network quite easy. Any user defined function's code can be accessed by converting it to a string (see "15.3.4.2 `Function.prototype.toString()`" in [16]). After retrieving this code in text form, the client uses the `eval` function to get an executable function object, calls it and send the result back to the server. Still appending all necessary definitions to the serialized code is both clumsy and inefficient. JavaScript does not have a module system built in yet, but some libraries can be used instead [20].

Another issue with volunteer computing is its safety. Every volunteer is downloading code from a network to execute it in their computers. If the servers get compromised, a malicious hacker could very well severely damage all connected computers. Here there is another advantage of using web technologies.

Web users are constantly under attack from malicious web sites that try to exploit every single feature of the browser. Web browsers have to be secure as well as functional. Every script runs very isolated from the host computer. There is a very restricted access to persistent storage [21], and none to the local file system. Access to remote sites through networking (AJAX calls, for instance) is restricted by the same origin policy [22]. Executing any program installed in the local computer is difficult, even more without the user's explicit permission.

All these limitations may appear as problems, but they are actually a relief. The constraints and policies the browsers apply guarantee the user a level of security that other distributed computing solutions do not.

Lastly, Javascript can be used both on clients and servers of a distributed system. JavaScript running in the server became a reality when Joyent Inc. released NodeJS in 2009 [23]. This software is a version of the Google Chrome's V8 engine [24] reworked to run outside of the browser. There are other options for running JavaScript in the server (like Mozilla's Rhino), but NodeJS remains the most popular [25].

Implementing the Capataz server in JavaScript has an important advantage. During a distributed algorithm development, many times it is unclear which parts of the workload will be sent to the clients to run, and which parts will be executed in the server. For example, a genetic algorithm can be distributed in several ways. The fitness evaluation for each member can be sent to a separate worker, or the whole genetic algorithm is better executed in parallel, e.g. with an island scheme [26]. If all logic (at both server and clients) is expressed in the same programming language (JavaScript), changing what executes on either side is far easier and quicker.

4 First tests

Capataz was implemented as a library and not as an application itself. It is not meant to be used as a separate server to which to submit jobs (as CrowdProcess), but rather to be embedded in another software. Distributed algorithms run in the server using Capataz to instantiate an HTTP server. Web browsers connect to this server, which will provide them with a web page and all the necessary code to become a contributing part of the distributed algorithm. Crowdprocess has the advantage that it is already running on the web with around 2.000 contributors most of the time. Yet, contrary to CrowdProcess' solution, Capataz is open source and it is possible to control how, when or where the distributed algorithm is executed.

Jobs are scheduled by the host application, to be split among all the connected clients. Promises are used to handle the asynchronism as comfortably as possible. Capataz handles lost connections, reassigning jobs if necessary. The browsers are also instructed by the server to retry lost connections. If the server gets shutdown and restarted, browsers reload all code that could have changed.

All of these features allow the programmers of distributed algorithms to ignore a lot of the details of the workload distribution itself, and better concentrate on its actual goal. Please refer to [1] for more details on the implementation.

There were two major concerns about the implementation. The first one was portability, i.e. if the system could actually bring many different platforms to work together. Portability must be checked, because is the

main justification to use web technologies. If a good portability is not achived, Capataz would be just a slower clone of BOINC.

The second one was performance, particularly how much the full execution time is reduced by parallelizing and distributing the workload. Performance must be verified because of two reasons: Javascript and HTTP. Javascript code is fast when compared with other scripting languages like Python, but it is slow compared to languages like C or C++. Using HTTP generates more traffic than UDP or TCP for the same amount of information being transfered.

4.1 Benchmark problem

The first tests of Capataz were designed to assess if the system was truly portable and met the performance expectations. The test case used for this purpose was an old, inefficient and inaccurate brute force method to estimate *pi*. This constant can be obtained using the formula for a circle's area.

$$A = \pi \times r^2 \therefore \pi = \frac{A}{r^2} \tag{1}$$

Consider a circle with radius r and centered at the origin (0,0). Any point which distance from the origin is less than or equal to r will be inside the circle. Therefore it is possible to estimate the area of the circle by adding up the ordinates of the points in the circumference at each abscissa.

$$if \quad r^2 = x^2 + y^2 \quad then \quad y = \sqrt{r^2 + x^2} \tag{2}$$

$$\therefore \frac{A}{4} \approx \sum_{x=0}^{r} \sqrt{r^2 - x^2} \tag{3}$$

$$\therefore \pi \approx \sum_{x=0}^{r} \sqrt{r^2 - x^2} = \sum_{x=0}^{r} \sqrt{\frac{4}{r^2} r^2 - x^2} \tag{4}$$

This process can be trivially parallelized in any number of concurrent calculations. The workload is distributed by making the clients return a slice of the sum for all values of x within a certain range. After all values of $x \in [0, r]$ have been dealt with, the server sums up all the intermediate results. After that it applies the formula and outputs the estimation, compared with the available (and quite accurate) value in `Math.PI`.

This technique is by no means a serious attempt at estimating *pi*. There are many far faster and more precise methods. In spite of that, it is a very good test case for Capataz. It is simple to program, yet it can easily demand huge amounts of computing power. Varying the radius of the circle the total amount of CPU time required for the whole process can be varied. Also, the process can be parallelized in any number of concurrent calculations. Varying the range, the amount of time spent by the client to finish one job can be altered as well. Since the accuracy is never greater than the other methods for estimating *pi*, the result of the process can always be properly verified.

4.2 Portability

Our goal was to be able to run code in all the devices usually available in university classrooms, computer labs and offices. The list includes:

- PCs and laptops with Windows XP, Windows Vista, Windows 7 or Windows 8.
- PCs and laptops with Ubuntu Linux.
- Macbooks and iOS devices (iPhones and iPads).
- Tablets and smartphones based on Android 3 or more.

So far the web browsers tested successfully to work with Capataz are:

- Internet Explorer 10 in PCs with Windows XP, Windows Vista, and Windows 7.
- Mozilla Firefox 21 or greater in PCs with Windows XP, Windows 7 and Ubuntu Linux.
- Google Chrome 27 or greater in PCs with Windows XP, Windows 7 and Ubuntu Linux.
- Opera 11 or greater in PCs with Windows 7.

- Safari 6 in iPhone 4S with iOS 6.
- Google Chrome 29 or greater in Android 4, with smart phones Nexus 4 and Samsung Galaxy S3.

Hence we conclude that the portability of Capataz is very good. The only real option missing is Internet Explorer version 9 or previous, which as of 2014 is still quite common. Despite that, PCs running this rather old version of Microsoft's web browser have other compatible options like Mozilla Firefox or Google Chrome.

4.3 Performance

The performance of Capataz was measured as the total time taken by distributed computations to complete. We believed that the network traffic would have a more considerable impact than the execution of the Javascript code. Specially if the jobs assigned to the browsers were too simple, i.e. took very little time to execute. HTTP is not a particularly efficient protocol, yet it is the only protocol available in standard browsers.

Our hypothesis was that the total runtime would result as the combination of two behaviours. First, there is the effect of parallelization. Suppose the whole workload takes T seconds to run executing in one processor. It is expected that if the workload is split in two jobs, executing in two different processors in parallel, it would take $T/2$ seconds to run the whole workload. In general, if the workload is split in P jobs (each run by a different processor) the time to complete would be T/P. Nevertheless this estimation is ignoring the cost of splitting the workload and joining the results after all processors have finished. We assumed the overhead of parallelization of a task to be a linear function of the number of jobs.

The total time of the distributed workload would be the sum of both functions. Figure 1 shows a simple example. The time gained by further splitting the workload decreases as the number of jobs increases. Also, the cost of splitting the workload, distributing the jobs and joining the results increases as the number of jobs increases. Hence, there is an optimal number of jobs in which the total time of the whole distributed workload is minimal.

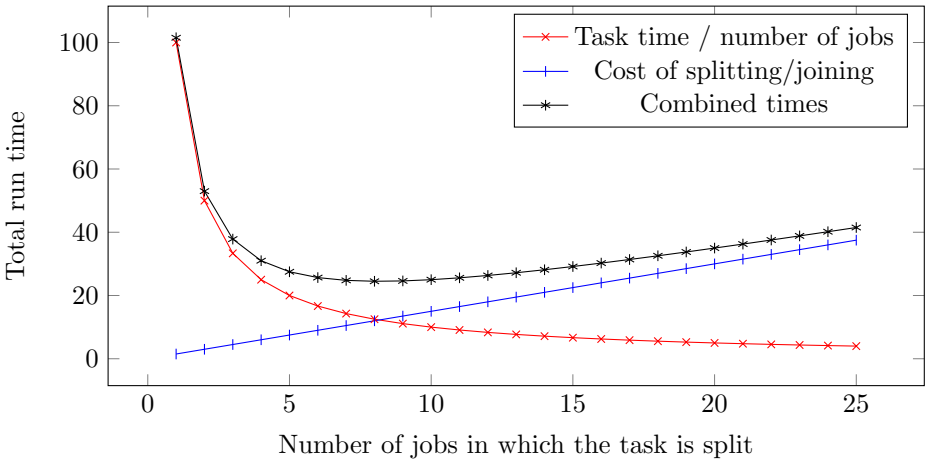

Figure 1: Hypothesized behaviour of distributed algorithms' total time

The cost of splitting includes the communication between the different processors. Furthermore, unless splitting the workload or joining the result were extremely complicated, most of this time is spent in network tranfers. Ergo, the network time greatly affects the optimal number of jobs. If this optimum were too low, using Capataz could be pointless, since it would be counterproductive to use a large number of devices.

The experiment performed with the *pi* estimation benchmark is described in [1] in full detail. Figure 2 shows how the total run time changed as the number of jobs increased. The test was run in a dedicated LAN of 12 computers (with 2 cores each). The full workload running in a single processor took approximately 85 seconds. The best number of jobs was 128, yet the task could be split up to around 8000 parts before the performance became worse that running the task in only one processor. At this extreme, job execution took around 50 milliseconds in average.

After these first experiments, Capataz's development focused on increasing the system's performance. The results showed that the full execution time of a distributed algorithm degraded severely if the jobs where evaluated too quickly by the clients. The amount of time spent in communication between the clients and the server should be reduced. Most importantly, the ratio between time spent in communication and the execution time in the browsers should be reduced. In the next section, we introduce the new feature added to the system to remedy this problem.

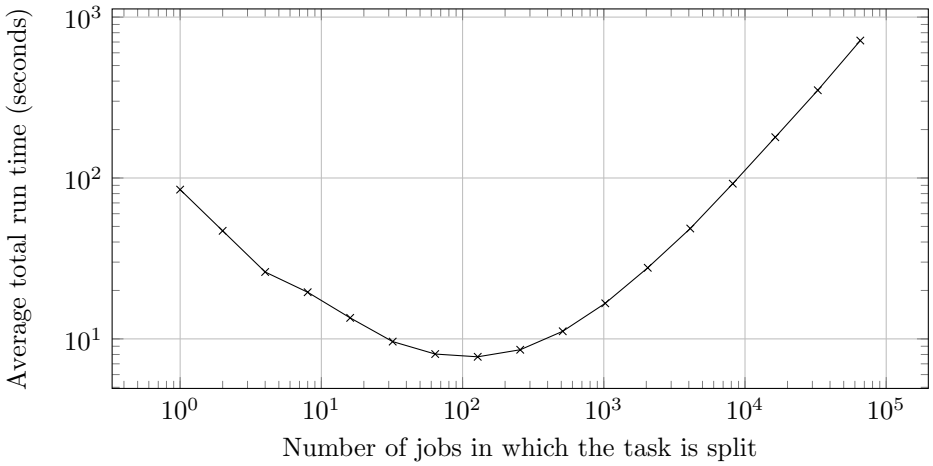

Figure 2: Experimental results of the first experiments.

5 Job bundling

It was speculated that bundling jobs could decrease the ratio of communication time over execution time. As the jobs get executed and results are reported, the server can monitor the execution time in the clients. The server can detect if execution time at the clients drops below a given threshold. The server can try to mitigate this by bundling jobs together, sending them all together to the workers. The client will execute them all in sequence and post all the results also in one bundle. In this way communication time is reduced, and the performance of the whole run can be improved. The bundling should be done transparently, so that the developers using Capataz would not have to change their code to use it.

The second test suite was carried out with an updated version of the server. The server no longer sends single jobs to the clients. It packs several of these jobs into what we call *tasks*. The server now monitors the average execution time of the jobs executed recently by the browsers. Two parameters were added to the server's configuration: the desired execution time for the clients (5 seconds by default) and the maximum amount of jobs that can be included in a task (50 by default). The amount of jobs to put into a task is calculated as follows.

$$taskSize = \max\left(1, \min\left(maxJobsPerTask, \left\lfloor \frac{desiredExecutionTime}{averageExecutionTime} \right\rfloor\right)\right) \tag{5}$$

5.1 Experiments

The performance improvement of this modification needed to be verified. The network time added by the work fragmentation is due in part to connection negotiation, which is reduced by bundling the jobs. Still, the amount of data sent back and forth between server and client in both cases is similar in size. Other factors might come into play here, like server side compression, Javascript compilation optimization, etc.

For the second test suite, Capataz would distribute the benchmark *pi* estimation in a LAN with 16 computers, using 2 web workers on each. The computer were Lenovo laptops with an Intel(R) Core(TM) i3 processor (3120M with 2 cores running at 2.5 GHz) and 4 GB of RAM, running Google Chrome 36 browsers on Windows 7 Professional SP1 (64 bits). These machines are more powerful that the ones used before. Also differently from the first tests, the server was not installed on the same network as the clients. All connections would have to go through a WAN link. Again all computers were exclusively dedicated to run these tests.

The server executed the *pi* estimation benchmark with 17 different configurations, 39 times each. All configurations used a radius of $2^{32} - 1$. The server performed each configuration with two task sizes: 1 (no bundling) and 50. Executing the whole workload by only one client took in average 25.3 seconds. When the workload was split in 2^{16} pieces and jobs were not bundled, each took around 11.4 milliseconds in average. Surprisingly, in the same scenario but with job bundling, each job took around 9.6 milliseconds in average.

In order to properly assess if the average execution time with job bundling is actually less than the average without bundling, we performed Welsh t-tests [27] on each configuration. The null hypothesis is that the average execution time without bundling is less than or equal to the average time with bundling. Hence, the alternative hypothesis is that job bundling has a lower average execution time. The Welsh variant

of the t-test is preferred over the simpler Student t-test, because we do not believe it can be assumed that the standard deviations of the distributions are the same.

Table 1: Execution time

Job count	Task size=1	Task size=50	Welsh-t	Welsh-df	p-value
1	24686.2	24724.9	-0.46	67.99	100%
2	12467.6	12371.8	2.24	131.83	1%
4	6282.4	6268.5	0.57	297.43	28%
8	3217.3	3129.5	5.44	392.62	0%
16	1606.7	1555.6	12.55	775.07	0%
32	819.5	778.1	16.40	1360.67	0%
64	424.1	400.3	18.73	2874.41	0%
128	226.9	203.8	43.46	5628.22	0%
256	117.9	107.7	52.80	12775.94	0%
512	68.4	60.1	81.84	29355.27	0%
1024	42.5	36.2	112.18	71634.24	0%
2048	28.3	22.6	209.85	132573.84	0%
4096	20.4	15.9	324.98	257690.90	0%
8192	16.0	12.6	418.04	538484.86	0%
16384	13.5	11.0	490.07	1123995.24	0%
32768	12.1	9.9	599.83	2243580.37	0%
65536	11.4	9.6	696.73	4475557.50	0%

Table 1 shows the measurements for the average job execution time at the clients with both task sizes, together with the values of the Welsh t-test. The null hypothesis cannot be refuted when executing with the whole workload, or when splitting it into 4 pieces. With 8 or more pieces it can be refuted with a significance level of 1%. It is clear that the average execution time decreases with job bundling when there is a significant amount of jobs.

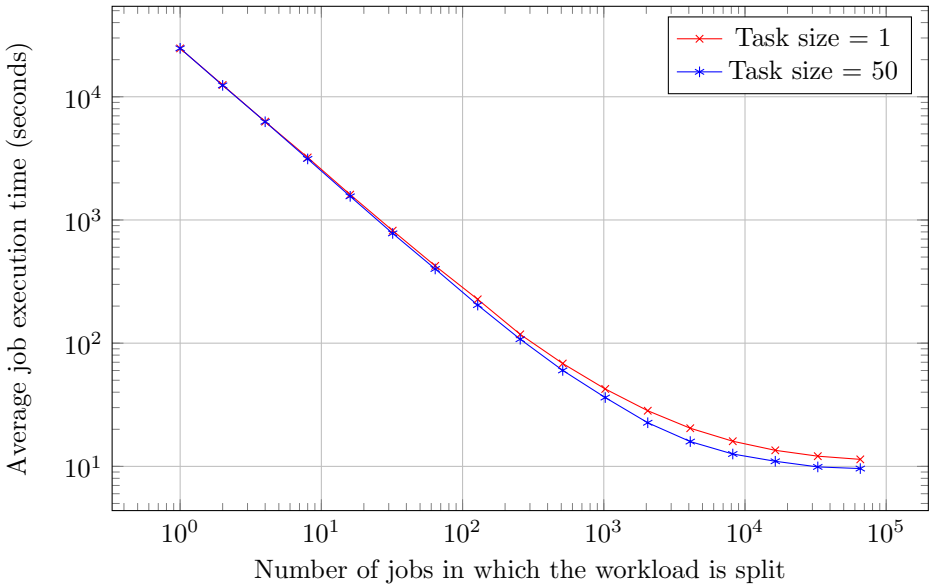

Figure 3: Job bundling comparison on execution time at the browsers as a function of the number of jobs

The only explanation we can provide for this behaviour is that Javascript engines in the clients are making optimizations. The scheduled jobs use the same code with different arguments. Maybe the JIT compiler is reusing the repeated code when the jobs are put together, but not when the code is retrieved from the server every time. Yet, this is highly speculative. In figure 3 the average execution times for both task sizes are plotted together, showing the small but clear difference between them. Besides the comparison of both scenarios, it is worth noting that the shape of the curves matches the results of the previous experiments. The execution time appears to converge to a lower value, since we are using faster hardware.

The main intent of these tests was to verify whether the full execution time degrades less with job

bundling or not. The answer is yes, although there is still an increase as the workload is split into more jobs. Figure 4 plots the average total times for both task sizes.

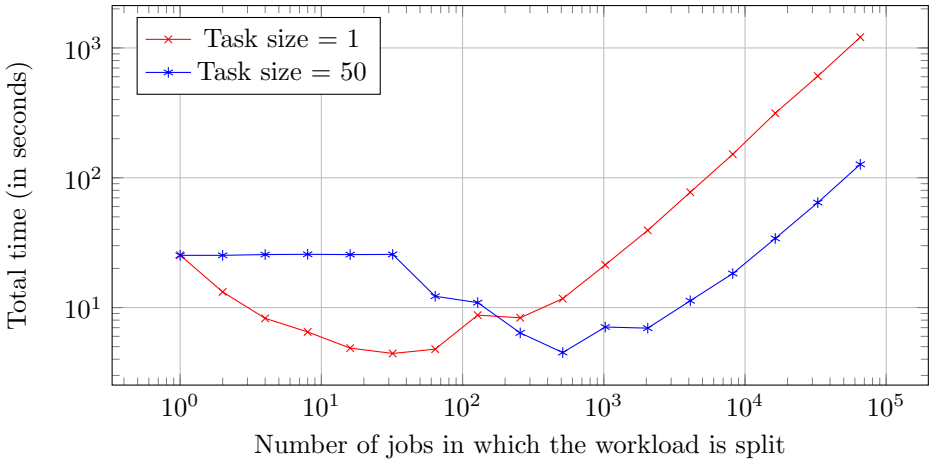

Figure 4: Job bundling comparison on total run time as a function of the number of jobs

With bundling the total time remains almost the same until the number of jobs is larger than the task size. As the amount of jobs increases, the time of the entire run is improved by job bundling almost by a factor of 10, compared to dispatching single jobs. Performance still degrades, as it is to be expected. Again, Welsh t-tests are performed to check if the average full execution time is really different with and without job bundling. The null hypothesis asserts both averages are equal, and the alternative hypothesis asserts the opposite. Table 2 shows the data plotted in figure 4, together with the calculations of the Welsh t-tests.

Table 2: Total run time without job bundling (task size 1) versus with job bundling (task size 50).

Job count	Task size=1	Task size=50	Welsh-t	Welsh-df	p-value
1	25.31	25.25	0.45	73.78	65%
2	13.20	25.25	140.00	73.89	0%
4	8.26	25.60	76.69	49.20	0%
8	6.49	25.72	99.21	55.76	0%
16	4.87	25.58	214.03	68.59	0%
32	4.43	25.67	241.93	58.57	0%
64	4.78	12.22	93.92	63.68	0%
128	8.74	10.91	22.40	46.10	0%
256	8.34	6.38	18.45	47.25	0%
512	11.72	4.50	60.62	46.07	0%
1024	21.32	7.08	6.73	38.61	0%
2048	39.34	6.94	124.75	48.89	0%
4096	77.34	11.28	113.68	41.91	0%
8192	151.70	18.29	86.79	39.34	0%
16384	314.12	34.10	24.06	38.13	0%
32768	607.08	64.37	108.76	38.89	0%
65536	1209.39	126.86	73.65	38.75	0%

The average total execution time can be considered different in all cases except when the workload is not split. This is reasonable. Capataz reassigns jobs, to cope with clients becoming unavailable. However jobs are not repeated in the same task, since it would be counterproductive. Hence, if there is only one job there can be only one task with only this job. That is also the reason why before 64 jobs, the full time with job bundling is the same as with only 1 job.

All configuration from 1 to 32 jobs are below the 50 jobs per task cap. In all these scenarios all jobs will be put into one task at the first request. Figure 5 plots the minimum, maximum and average task sizes for the run with task size 50. It shows the three values are the same until the job counts gets greater than the task size. The average execution time Capataz uses to calculate the task size is not available at the start, when results have not been posted yet. In this case, the server will use the maximum size. This logic may need to be revised.

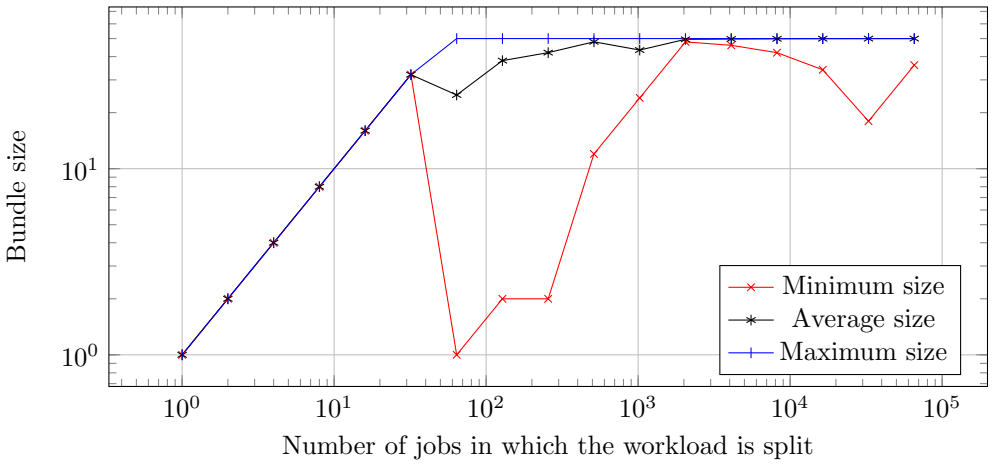

Figure 5: Task size as a function of the number of jobs

In spite of a worse performance with low job counts, as the amount increases the job packing results in better times. The cross point is between the 128 and 256 jobs, with execution times around 200 and 100 milliseconds respectively.

Again it is worth noting that the full execution time without job bundling matches the results of previous experiments. In this case the lower time is found where the job count is equal to the amount of parallel workers being used: 32.

6 Conclusion

In this paper we have shown Capataz to be an adequate solution for distributed computing. After the first tests our biggest concern was the possible performance degradation if jobs got executed too quickly. Job bundling is very useful to mitigate this problem. Experiments show that the total execution time can be reduced by up to 10 times, when compared to the previous version. It allows the programmers of the code to be distributed to split it as they see fit, regardless of any possible problems resulting from these parts being too simple.

6.1 Future work

Nonetheless there is still room for enhancements. For example, the server trusts the clients to run the task exactly as it was delivered, and to reply with the actual results properly. Yet a browser may return a wrong result, either by accident or intentionally. Volunteer computing schemes usually apply some sort of control over the connected clients and the results they report. User authentication can be enforced, but it may hinder the participation of some devices in the distributed algorithm.

This problem can be tackled with a (configurable) degree of redundancy of the scheduled jobs. Basically some or all jobs are assigned many times to different clients. Only the results in which more than one client concur are taken into account. By doing so, one or a few rogue workers cannot seriously damage the whole distributed run.

Capataz is quite robust from the browsers' perspective. Every client is set up to retry failed communications with the server, resuming as soon as it is possible. Still, a server failure always implies a restart of the whole process. Server restart is monitored by the clients, and when detected it forces the reloading of all Javascript modules, to cope with possible changes. But former scheduled jobs and their results are not stored, so they cannot be restored and have to be executed again.

This is a result of how the solution was designed. Capataz manages the task scheduled by the process that uses it. This abstraction allows for a very flexible framework, but makes the system recovery very difficult. So far the recovery of an interrupted run must be handled by the program using Capataz.

Lastly, a new development is being discussed in the Web Hypertext Application Technology Working Group (WHATWG), as of October 2014. Allegedly, Javascript code will be able to inspect the number of threads the browser can spawn safely. This property is given the provisional name `hardwareConcurrency`. When standardized, this datum will allow Capataz clients to adapt to the host platform. The current version spawns a fixed number of web workers, part of the server's configuration.

References

[1] G. Martinez and L. Val, "Implementing crossplatform distributed algorithms using standard web technologies," in *Computing Conference (CLEI), 2014 XL Latin American*, Sept 2014, pp. 1–8.

[2] BOINC Project, "Volunteer Computing," Oct. 2012 (accessed Sep. 2014). [Online]. Available: http://boinc.berkeley.edu/trac/wiki/VolunteerComputing

[3] IBM Corporation, "World Community Grid," 2015 (accessed May. 2015). [Online]. Available: http://www.worldcommunitygrid.org/

[4] Oxford University, "ClimatePrediction.net," 2015 (accessed May. 2015). [Online]. Available: http://www.climateprediction.net/

[5] University of California, Berkeley, "SETI@home," 2015 (accessed May. 2015). [Online]. Available: http://setiathome.berkeley.edu/

[6] F. Boldrin, C. Taddia, and G. Mazzini, "Distributed computing through web browser," in *Vehicular Technology Conference, 2007. VTC-2007 Fall. 2007 IEEE 66th*, Sept 2007, pp. 2020–2024.

[7] J. J. Merelo, A. M. García, J. L. J. Laredo, J. Lupión, and F. Tricas, "Browser-based distributed evolutionary computation: Performance and scaling behavior," in *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 2851–2858. [Online]. Available: http://doi.acm.org/10.1145/1274000.1274083

[8] J. Merelo-Guervos, P. Castillo, J. Laredo, A. Mora Garcia, and A. Prieto, "Asynchronous distributed genetic algorithms with javascript and json," in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, June 2008, pp. 1372–1379.

[9] M. Grivas and D. Kehagias, "A multi-platform framework for distributed computing," in *Informatics, 2008. PCI '08. Panhellenic Conference on*, Aug 2008, pp. 163–167.

[10] R. Cushing, G. Putra, S. Koulouzis, A. Belloum, M. Bubak, and C. De Laat, "Distributed computing on an ensemble of browsers," *Internet Computing, IEEE*, vol. 17, no. 5, pp. 54–61, Sept 2013.

[11] J. Duda and W. Dlubacz, "Gpu acceleration for the web browser based evolutionary computing system," in *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference*, Oct 2013, pp. 751–756.

[12] Nokia Research, "Webcl," 2014 (accessed May, 2015). [Online]. Available: http://webcl.nokiaresearch.com/

[13] T. MacWilliam and C. Cecka, "Crowdcl: Web-based volunteer computing with webcl," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, Sept 2013, pp. 1–6.

[14] J. P. Jerónimo, P. G. Fonseca, and J. A. Menano, "CrowdProcess / the HTML5 supercomputer," 2013 (accessed October 2014). [Online]. Available: https://crowdprocess.com

[15] A. L. Hors, D. Raggett, and I. Jacobs, "HTML 4.01 specification," W3C, W3C Recommendation, Dec. 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-html401-19991224

[16] ECMA, *ECMA-262: ECMAScript Language Specification*, 5th ed., Jun. 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[17] R. Berjon, S. Faulkner, T. Leithead, S. Pfeiffer, E. O'Connor, and E. D. Navara, "HTML5," W3C, Candidate Recommendation, Jul. 2014. [Online]. Available: http://www.w3.org/TR/2014/CR-html5-20140731/

[18] B. Fulgham and I. Gouy, "The Computer Language Benchmarks Game," 2014 (accessed Sep. 2014). [Online]. Available: http://benchmarksgame.alioth.debian.org

[19] Web Hypertext Application Technology Working Group (WHATWG), "HTML living standard," Tech. Rep., Jun. 2013 (accessed Sep. 2014). [Online]. Available: https://html.spec.whatwg.org/

[20] J. R. Burke, "RequireJS: A Javascript module loader," Sep. 2009 (accessed Oct. 2014). [Online]. Available: http://requirejs.org

[21] I. Hickson, "Web storage," W3C, W3C Recommendation, May 2014. [Online]. Available: http://dev.w3.org/html5/webstorage/

[22] A. Barth, "The Web Origin Concept," Internet Requests for Comments, RFC 6454, Dec. 2011. [Online]. Available: http://tools.ietf.org/html/rfc6454

[23] C. Metz, "The node ahead: JavaScript leaps from browser into future." *The Register*, Mar. 2011.

[24] Google Inc., "V8 JavaScript Engine," 2008 (accessed Oct. 2014). [Online]. Available: https://code.google.com/p/v8/

[25] M. Baxter-Reynold, "Here's why you should be happy that microsoft is embracing node.js," *The Guardian*, Nov. 2011. [Online]. Available: http://www.guardian.co.uk/technology/blog/2011/nov/09/programming-microsoft

[26] F. Lardeux and A. Goëffon, "A dynamic island-based genetic algorithms framework," in *Proceedings of the 8th International Conference on Simulated Evolution and Learning*, ser. SEAL'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 156–165. [Online]. Available: http://dl.acm.org/citation.cfm?id=1947457.1947476

[27] B. L. Welch, "The generalization of 'student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. pp. 28–35, 1947.