IMPROVING EFFICIENCY OF A WEB-BASED DISTRIBUTED EVOLUTIONARY COMPUTING SYSTEM

Jerzy Duda and Wojciech Dłubacz

*AGH University of Science and Technology, Faculty of Management
ul. Gramatyka 10, 30-067 Kraków, Poland*

ABSTRACT

The paper presents a study on performance improvement of an evolutionary computing system built on the basis of JavaScript technology. The system consists of web browsers performing computational tasks and a web server based on Go programming language (Golang technology) that controls the computing environment. The newest web technologies such as HTML5 Web Workres and JavaScript WebAssembly are used to maximize the utilization of the resources provided by web clients. Computational experiments are performed on large instances of popular NP-hard problems: travelling salesman problem (TSP), flowshop scheduling problem (FSP) and recourse constrained project scheduling problem (RCPSP).

KEYWORDS

Distributed Computing, Evolutionary Computation, Social Computing, Grid Computing, Internet

1. INTRODUCTION

Since the launch of SETI@home project in 1999 and its successor BOINC (Berkeley Open Infrastructure for Network Computing (Anderson, 2004)), grid computing basing on personal computers is becoming more and more popular and in many applications may be considered as viable alternative for very expensive super computers. On the other hand, growing interest in optimization of complex problems, as well as processing huge amount of data required for decision making in contemporary business and management steadily increase the demand for computing power not only from the side of research centers, government agencies or military organizations, but also from the side of large and medium-sized companies.

Despite their advantages, grid computer systems based on BOINC or similar platforms (e.g. Grid MP, distributed.net) have two major drawbacks. First of all a user participating in a computing project must download a special client application (agent) that communicates with the central server. Some users may not be willing to download software that regularly sends data from their computers. Moreover, such application is dedicated to a particular operating system and is device dependent (different versions must be installed for a computer, a mobile device, etc.), what further limits the number of potential participants in a computing grid. Another problem with such type of distributed systems can be caused by firewalls blocking communication between a client and the server as they usually use some non-standard TCP ports.

Introduction of Google's V8 JavaScript engine based on just-in-time compilation in 2008 started a new era for web browsers allowing a significant reduction of the execution speed gap between a JavaScript code and a machine code generated by such compiled languages as C or C++. This contributed to a growing interest in grid computing based entirely on web-browser clients.

Along with the introduction of HTML5 new APIs such as Web Workers have become available for JavaScript programmers. In the recent years web browser developers presented some new techniques which allow for further acceleration of the code written in JavaScript. The goal of this paper is the analysis of how those techniques may contribute to performance improvement of a web-based distributed computing.

2. WEB-BASED DISTRIBUTED COMPUTING SYSTEMS

First web browser based distributed computing systems emerged with the beginning of intensive development of JavaScript technology. Merelo et. al. presented a series of papers describing distributed evolutionary computing system. Their first system (Merelo et al., 2007) was based on a master-slave architecture. The server running on Ruby on Rails (RoR) technology maintained the population of solutions and sent the tasks consisting of fitness function calculation to the distributed clients (web browsers). The tasks were sent in the JavaScript Object Notation (JSON) format as an AJAX response to the request of awaiting clients. The system was difficult for scaling since only server performed the main routine of the evolutionary algorithm and due to poor distributing performance of RoR even small amount of clients have to wait for an answer from the server. An improved version of the system was described in (Merelo et al., 2008). It followed an island model of computations in which populations were run independently on clients' machines. The tasks for the server were only collecting the best individuals and sending them back in a response to clients' requests. An improved version of the system scaled much better and was able to serve more than 40 clients. In one of their latest papers the authors developed a simple Node.js server (Merelo et al., 2014) in order to show that a Node.js server with a master-slave architecture of parallel computing is more effective than peer to peer (P2P) computation.

Another group of web browser based computing systems consists of general purpose systems that may perform any kind of tasks provided in the form of a JS function that is run parallelly on clients' machines. For example a system called WeevilScout was described in (Cushing et al., 2013). It is a master-slave distributed computing system in which the server uses REpresentational State Transfer (REST) service to queue awaiting computations. Web browsers pull packaged jobs from the server described in XML messages and send back the results to the REST service.

A separate group of systems are the systems in which the users' machines execute some unwilling computations. The system performing a parasitic JavaScript computations was proposed by Jenkin (2008). It utilized JSON and AJAX technologies and was able to limit the CPU load on the clients' machines by the application of timer based programming. The system was applied not only to solve an optimization problem (N-Queens) by a genetic algorithm, but also for ray tracing tasks and for perceptron learning in an artificial neural network. Another system that performed unwilling distributed computations was presented by Klein and Spector (2007). The system used PHP and Push technology on the server's side, whereas JavaScript and AJAX technologies were used on the clients' side to solve simple genetic programming problems.

Finally, MacWilliam and Cecka (2013) described a web-based volunteer computing system that used exclusively WebCL to solve Thomson nonlinear optimization problem.

3. WEB-BASED DISTRIBUTED COMPUTING SYSTEMS

Computational efficiency of an evolutionary distributed computing system is conditioned by two main factors:

1. Technological solutions used to code a server side, as well as the ones used to code a client's side.
2. Architecture of an evolutionary system itself.

The proposed evolutionary computing system uses Go programming language (Golang) on the server side, while Web Workers and WebAssembly technologies are used on the client's side. The architecture of the system is shown in Figure 1.

3.1 Server Implementation

In contrast to (Merelo et al., 2007) and our earlier system presented in (Duda, Dlubacz, 2013), the system supervisor - a WebSockets service that all clients connect to is implemented in Golang technology (Newmarch, 2017).

Research made at the initial stage of system development showed that the implementation of message exchange using Node.js have an additional overhead for handling each message reaching 40 ms, what after just-in-time-compilation (JIT) turns into an overhead of 15 ms, compared to the same implementation in Golang. JIT technique, which is built inside JavaScript, compiles frequently used functions preventing from

their interpretation at each call (Martinsen et al., 2011). The Golang was developed by Google and it is a compiled language, with the ease of writing a scripting language, and in compare to naturally one-threaded JavaScript Node.js environment it has built-in multithreading support. Thanks to the use of this language, very high scalability of the system has been achieved, providing support for many simultaneous connections and dynamic communication between them.

Figure 1. Platform workflow diagram

A non-relational database MongoDB (Plugge, 2010), which is also easily scalable and enables support of multiple simultaneous connections, was used to collect data obtained from the computing units.

Genetic algorithms can be easily run parallelly because by definition computations are made on the set of solutions (population of individuals). In general, three basic optimization models can be applied to a distributed system of genetic algorithms, as enumerated in (Talbi, 2006):

- global optimization model - calculations of the objective function for members of the population maintained on the server are executed in parallel on clients' computers; such approach was applied in (Merelo, 2017),
- master-slave model (synchronous island model) - the main population on the server communicates with subpopulations to collect the best solutions to obtain final solution,
- asynchronous island model - the central server is responsible only for collecting the best solutions from the population and sending them to clients.

In the presented platform the last of the above listed models has been implemented, as subpopulations can be potentially calculated on different heterogeneous machines with different computing power, including mobile devices. Additionally different versions of the genetic algorithm or some other algorithms (e.g. used for local search) may be delegated in the future to some clients, depending on their computing capabilities.

3.2 Client's Side Implementation

To make the whole evolutionary computing system flexible, its client's logic is built of interchangeable blocks of the algorithm codes:

- chromosome generators - create initial population, thus determine the representation of the solution (chromosome structure)
- fitness functions (FX) - represent objective function for a given optimization problem
- crossovers - create new solutions on the basis of two existing solutions
- mutations - alter some small portion of the solution (e.g. changes a gene value)

Those blocks are combined to form a calculation scheme. Parameters that can be additionally modified are:

- the probability of crossover,
- the probability of mutation,
- size of the population (number of members of a given population),
- calculation time limit for a single client.

The way in which a user can define the components of an evolutionary algorithm is shown on the example of the flowshop scheduling problem, which will be discussed in the following section.

Initially in the chromosome generation stage, a random permutation of jobs is generated. After a solution is created or altered by the means of genetic operators, its fitness value has to be evaluated. Listing 1 shows fitness evaluation for the flowshop scheduling problem.

```
return function fitness(c, p) {
  var m = 0, i = 0, t = {}, job = 0, n = p[0].length, mC = p.length;
  (function() {
    for (let j = 0; j < mC; j++) {
      t[j] = {};
      for (let k = 0; k < n; k++) {
        t[j][k] = 0;
      }
    }
  })();
  for (i in c) {
    i = parseInt(i, 10);
    job = c[i];
    if (typeof(job) !== 'number') { continue; }
    for (let m = 0; m < mC; m++) {
      if (m === 0) {
        t[m][i] = (p[m][job] + t[m][i - 1]) || p[m][job];
      } else if (typeof(t[m][i - 1]) === 'number') {
        t[m][i] = ((t[m][i - 1] > t[m - 1][i]) ? t[m][i - 1] : t[m - 1][i])
          + p[m][job];
      } else {
        t[m][i] = t[m - 1][i] + p[m][job];
      }
    }
  }
  return t[mC - 1][n - 1];
};
```

Listing 1. Fitness function for the flowshop scheduling problem written in JS code

Next, the best fitted members in the population are selected to form a new generation basing on their fitness values. Selection algorithm (binary tournament) is currently embedded into the flow of the evolutionary algorithm and cannot be changed yet by the user. After selection stage, crossover (e.g. PBX) and mutation (e.g. swapping two jobs) operators are applied with the probability that can be set manually before running computations. The codes of the standard operators are already predefined in the system, but the user can define new operators.

4. EFFICIENCY IMPROVEMENTS

4.1 Test Problems

In order to test the performance of the system three different popular combinatorial optimization problems have been used:

- Symmetric Traveling Salesman Problem (TSP)
- Permutation Flowshop Scheduling Problem (FSP)

- Resource Constrained Project Scheduling Problem (RCPSP)

TSP is a classical problem in combinatorial optimization formulated first in 1930 and since then studied by many researchers. It can be formulated as follows: Given a list of cities and their pairwise distances, the goal is to find the shortest possible tour in which all cities are visited exactly once. Usually the position of the cities is provided as points' coordinates in 2D Euclidean space. Thus the length of the tour (fitness function) is a simple sum of distances between cities in the sequence (the last city becomes also the first city, as the salesman must return to the origin):

$$f(x) = \sum_{i=1}^n \sqrt{(q_{i,1} - q_{i+1,1})^2 + (q_{i,2} - q_{i+1,2})^2}$$

Such function is easy to compute regardless the language that is used.

Flowshop scheduling problem denotes the problem of determining the best sequence of jobs that have to be processed on each machine in the flow shop in exactly same order. The most frequently criterion of optimality for FSP is minimization of the total time necessary to complete all jobs on all machines (makespan). JavaScript code of such fitness function was presented in Listing 1. Solutions for FSP can have the same representation as for TSP, but its objective function is more complicated to compute.

The last problem that was tested - Resource Constrained Scheduling Problem - has even more complicated goal function. RCPSP considers resources of limited availability and activities of known durations, resource requests and precedence constraints. The goal is to find a schedule that minimizes total duration of activities by assigning a start time to each activity in the way that the precedence constraints and the resource availabilities are respected. Thus each time the fitness is calculated all the constraints have to be evaluated.

It is worth noting that for the computational experiments the test problems have been chosen in order to have different computational complexity of their objective functions.

4.2 Multithreading a JavaScript GA Algorithm

Due to the fact that nowadays even mobile phones have multi-core processors, one of the basic solution to speed up calculations is to use all CPU cores simultaneously. As it was mentioned earlier, in the case of GA, splitting calculations into more threads is relatively easy. In this approach the simplest solution is to let every computational thread manages its own population. These populations, after finding another better solution, communicates with the main thread of the computing procedure performed on the client's side and then the best one is sent to the server.

In JavaScript, currently the only API allowing to use more than one CPU core is called Web Worker. There is another JavaScript API named SharedWorkers but it also bases on the same mechanism of spawning new threads. JavaScript can spawn as many Web Workers as only are allowed by an operating system. To check for the amount of CPU cores JavaScript provides property called *navigator.hardwareConcurrency*. The main problem with Web Workers is that they communicate with the main thread by copying data rather than sharing them. To verify the efficiency of Web Workers technology, the tests shown in Table 1 were performed. For the code execution, a *setTimeout* method technique was used to prevent clients' web browsers from slowing down, analogously as in the system presented in (Duda, Dlubacz, 2013).

Amount of generated populations for Flowshop problem with 20 jobs and 10 machines for different Web Workers performed on Intel i5 4 core CPU processor was presented in Table 1. Data are presented as number of generations performed per second.

Table 1. Generations per sec. for different WebWorkers amount

Main	1	2	3	4	5	6
88.62	99.93	199.76	297.37	398.84	472.43	542.43

The results show almost linear increase of the system performance until the number of used Web Workers equals to the number of CPU physical cores. After exceeding this limit, due to the use of the processor unloading technique, the increase in the number of computed generations still occurs, but no longer in a linear way. This experiment shows also that calculations run on a separate Web Worker are more efficient than these run in the main thread which is also responsible for rendering web page.

4.3 Analysis of the Algorithm's Components

To further increase GA performance on the client's side some studies have been performed to identify its bottlenecks. For this purpose, code profiling for different sizes of the FSP and RCPSP problem was performed.

The results showing the impact of the particular stage of the evolutionary algorithm performed by a client are gathered in Table 2 and Table 3.

Table 2. Running time for JavaScript implemented objective function, crossover and mutation for FSP problem

EA stages	20x10	50x10	100x10	200x10
Fitness calculation	87.91%	85.90%	80.70%	71.45%
Crossover calculation	11.96%	14.06%	19.27%	28.53%
Mutation calculation	0.12%	0.05%	0.03%	0.02%

Table 3. Running time for JavaScript implemented objective function, crossover and mutation for RCPSP problem

EA stages	30	60	120
Fitness calculation	94.52%	96.51%	97.90%
Crossover calculation	5.44%	3.48%	2.10%
Mutation calculation	0.05%	0.01%	0.01%

Analyzing the above results it can be clearly seen that the most time consuming stage of the evolutionary algorithm is the calculation of the fitness function. To compare the time spending on the particular stages of GA both crossover and mutation probability were set to 1.0 (100%). In real cases probability of crossover is usually set between 0.5 and 0.9, while mutation probability usually does not exceed 0.05. This means that in real cases the fitness calculation will be using even more computational power relatively to the calculation of the crossover stage, while the computing power spent on the mutation stage is virtually meaningless.

4.4 Using of WebAssembly

In February 2017, four major browser manufacturers: Chrome, Edge, Firefox, and WebKit, came to an consensus on the technology for linking compiled high-level language code into JavaScript. That consensus included a JavaScript API and binary code format. This technology is currently named WebAssembly (WASM). It is a binary instruction format for a virtual machine and is designed as a portable target for compilation of high-level languages.

Developers of the WebAssembly believe that this technology is extremely fast – application can run almost with the speed of a native binary code execution. It should be emphasized that rewriting JavaScript fitness function into WebAssembly code was not obvious due to frequent changes in its specification and small community. Listing 2 presents C++ code for counting flowshop fitness function that was compiled with Emscripten - recommended tool to compile C/C++ code into WebAssembly binary.

```
int EMSCRIPTEN_KEEPALIVE fitness(int n, int m, int *t, int *o) {
    int w;
    int *C = new int[m + 1];
    C[0] = t[o[0] * m + 0];
    for (int j = 1; j < m; j++)
        C[j] = C[j - 1] + t[o[0] * m + j];
    for (inti = 1; i < n; i++) {
        C[0] += t[o[i] * m + 0];
        for (intj = 1; j < m; j++)
            C[j] = (C[j - 1] > C[j] ? C[j - 1] +
                t[o[i] * m + j] : C[j] + t[o[i] * m + j]);
    }
    w = C[m - 1];
    delete[] C;
    return w;
}
```

Listing 2. WebAssembly fitness function written in C++

The compiled binary file with the .wasm extension was then added as a dependency for the evolutionary algorithm. The results presenting calculation time measured for the fitness function for different problems and different size are presented in Table 4.

Table 4. Running time comparison for JS, WASM and C/C++

Problem/size	JavaScript	WebAssembly	Visual C++
TSP 100	0.100	0.099	0.000
TSP 1000	0.799	0.099	0.001
TSP 10000	7.100	0.100	0.006
TSP 100000	18.399	0.900	0.062
FSP 50x5	0.100	0.099	0.000
FSP 200x20	0.800	0.099	0.001
FSP 800x60	11.000	0.299	0.019
FSP 2000x120	15.000	1.399	0.081
RCPSP 30	1.499	0.099	0.000
RCPSP 60	3.099	0.099	0.000
RCPSP 120	28.699	0.200	0.009
RCPSP 240	55.299	2.699	0.086

The results of experiments discloses that the usage of WebAssembly technology can enable to perform complex calculations in a significantly shorter time than standard JavaScript execution. It is also visible that WebAssembly is still not as fast as native binary code due to necessary data transfer to WASM module form JavaScript script.

In Table 5 the savings in execution time for the fitness evaluation function and for the whole genetic algorithm procedure are shown for different optimization problems and their sizes.

Table 5. Time reduction for fitness function and the whole GA execution resulting from applying WASM module

Problem/size	Fitness time reduction	GA time reduction
TSP 1000	87.63%	71.19%
TSP 10000	98.59%	80.10%
TSP 100000	95.11%	77.27%
FSP 200x20	87.63%	71.19%
FSP 800x60	97.28%	79.03%
FSP 2000x120	90.67%	73.66%
RCPSP 30	93.40%	88.27%
RCPSP 60	96.81%	93.43%
RCPSP 120	99.30%	96.87%

Since objective function in the resource constrained project scheduling problem (RCPSP) is more complicated and time consuming than makespan calculation for the flowshop scheduling problem, applying WASM module in a distributed system brings significant reduction in computational time for the whole genetic algorithm even for small size of problem instances. Time reduction varies from 88.3% to almost 97%, which means that the distributed system can process from 9 to 33 times more tasks (e.g. generations) than the system without the WASM module. For the FSP problem a significant gain from applying WASM can be observed for the instances with 100 jobs and more. In such cases time reduction varies from 71.2% to 79.0%, which means that the distributed system with WASM module can perform about 3 times more tasks.

5. CONCLUSIONS

The presented distributed evolutionary computing system thanks to its independent component structure pointed the weaknesses of a typical way in which genetic algorithm is implemented. The conducted research showed that the main factor influencing the number of evolutionary algorithm generations that can be computed in a single second on a client's machine was the calculation of the fitness function.

The technologies examined in this article, such as Web Workers and WebAssembly allowed for a significant performance increase of the evolutionary distributed computing system.

Web Workers technology seems to linearly increase system speed to the limit of the amount of physical CPU cores installed on the client's machine. This technology have its main bottleneck in the data transfer mechanism that is making a serialized copy between host thread and Web Worker instead of sharing a reference.

The obtained results indicate that in many cases WebAssembly technology can significantly speed up the function execution. However, due to constantly changing specification and difficulties related to transferring complex JavaScript structures to the function module, it can be assumed that this technology should be only used to implement functions that have the greatest impact on the performance of the evolutionary algorithm.

In the future the application of GPGPU will be considered, however, contrary to WebAssembly, the WebCL standard allowing for easy integration of OpenCL expressions with a JavaScript code is currently not implemented in any web browser.

ACKNOWLEDGEMENT

This work is financed by grants for the maintenance and development of research capacities.

REFERENCES

- Anderson, D. P., 2004. Boinc: A system for public-resource computing and storage. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing GRID'04*, IEEE Computer Society, pp. 4–10.
- Cushing, R. et al., 2013. Distributed computing on an ensemble of browsers. *Internet Computing*. IEEE, Vol. 17, No. 5, pp. 54–61.
- Duda, J., Dlubacz, W., 2013. Distributed evolutionary computing system based on web browsers with Javascript. *11th International Conference, Applied Parallel and Scientific Computing PARA 2012*, Springer Berlin Heidelberg, pp. 183–191.
- Jenkin, N., 2008, COMP520-08 Report: Parasitic JavaScript.
- Klein, J., Spector, L., 2007. Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation GECCO'07*, ACM, pp. 1628–1635.
- MacWilliam, T., Cecka, C., 2013. CrowdCL: Web-based volunteer computing with WebCL. *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2013, pp. 1–6.
- Martinsen, J. K., Grahm, H., Isberg A., 2011. A Comparative Evaluation of JavaScript Execution Behavior. *Web Engineering. ICWE 2011*. Lecture Notes in Computer Science, vol 6757. Springer, Berlin, Heidelberg.
- Merelo, J. J. et al., 2007. Browser-based Distributed Evolutionary Computation: Performance and Scaling Behavior. *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation GECCO'07*, ACM, pp. 2851–2858.
- Merelo, J. J. et al., 2014, NodeEO, a Multi-paradigm Distributed Evolutionary Algorithm Platform in JavaScript. *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*. GECCO Comp'14, ACM, pp. 1155–1162.
- Merelo-Guervos, J. J. et al., 2008. Asynchronous distributed genetic algorithms with Javascript and JSON. *Evolutionary Computation*, pp. 1372–1379.
- Newmarch, J., 2017, *Network Programming with Go: Essential Skills for Using and Securing Networks*. Apress, Berkely, CA, USA.
- Plugge, E., 2010. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, Apress, Berkely, CA, USA.
- Syswerda, G., 1991. *Schedule optimization using genetic algorithms*, Handbook of Genetic Algorithms, New York, USA.
- Talbi, E.-G., 2006. *Parallel Combinatorial Optimization (Wiley Series on Parallel and Distributed Computing)*, Wiley-Interscience, New York, USA.