Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML

Jon Klein
Cognitive Science
Hampshire College, Amherst, MA, USA, *and*
Physical Resource Theory
Chalmers U. & Goteborg U., Goteborg, Sweden
jk@artificial.com

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA, USA
lspector@hampshire.edu

ABSTRACT

The success of a genetic programming system in solving a problem is often a function of the available computational resources. For many problems, the larger the population size and the longer the genetic programming run the more likely the system is to find a solution. In order to increase the probability of success on difficult problems, designers and users of genetic programming systems often desire access to distributed computation, either locally or across the internet, to evaluate fitness cases more quickly. Most systems for internet-scale distributed computation require a user's explicit participation and the installation of client side software. We present a proof-of-concept system for distributed computation of genetic programming via asynchronous javascript and XML (AJAX) techniques which requires no explicit user interaction and no installation of client side software. Clients automatically and possibly even unknowingly participate in a distributed genetic programming system simply by visiting a webpage, thereby allowing for the solution of genetic programming problems without running a single local fitness evaluation. The system can be easily introduced into existing webpages to exploit unused client-side computation for the solution of genetic programming and other problems.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Performance

Keywords

JavaScript, XML, AJAX, Push, PushGP, stack-based genetic programming, networking

1. INTRODUCTION

Genetic programming is a computation-hungry technique. Provided that a genetic programming system has been properly equipped (through system configuration and program representation) to solve a particular problem, the probability of success often increases with population size and the number of generations run. The limiting factor in applying genetic programming to a problem is thus frequently the amount of computation available to the process.

Fortunately, traditional genetic programming techniques can be described as "embarrassingly parallel," meaning that parallelization is a relatively trivial process. Individual fitness tests are typically independent of one another and can be run on different machines with little concern for synchronization between them. Even in situations of low network bandwidth in which distributing individual fitness tests is impractical, simulations may be parallelized though the simultaneous evolution of independent subpopulations (informal versions of demes) with occasional migration of small numbers of individuals between subpopulations. An overview of the many ways in which evolutionary algorithms can be parallelized can be found in [12].

The challenge in exploiting additional computation for genetic programming problems is thus not in finding a way to parallelize the underlying process, but rather in procuring access to additional computation to be exploited. In recent years, a number of projects (mostly unrelated to evolutionary computation) have begun to exploit unused computation across the internet. Among the most popular are SETI@home, a search for signs of extra-terrestrial intelligence[1]; Folding@Home, simulations of protein-folding and other molecular dynamics simulations[2]; and distributed.net, which is equipped to solve a number of different problems, mostly related to cryptography.[3]

In spite of the success many of these projects have had in reaping idle computation, few would argue that there is not still an enormous untapped potential for distributed computation. Participation in existing distributed computation projects requires awareness of the projects, willingness to participate, computer savvy, and a time investment to download and install client-side computational software. It is possible that a large number of users would consent to a small donation of computational resources, but that it is only a relatively small portion of these users who have the

[1]`http://setiathome.berkeley.edu/`
[2]`http://folding.stanford.edu/`
[3]`http://distributed.net/`

interest, knowledge, time, and motivation to seek out and install one of the systems described above. So although the requirements for participation are modest we suspect that they substantially limit the pool of available computation nonetheless.

In this paper we present a technique, well suited to evolutionary computation, for exploiting unused client-side computation without requiring client-side software or special user action and even without a user's awareness of the process (although we do advocate disclosure—see the section on Ethics and Security below). The computational resources can be automatically harvested when a user visits a website, with no user consent required. Simply by opening a webpage, a user's machine can be used to evaluate fitness tests and aid in the solution of genetic programming problems. Using this technique, we have been able to solve simple genetic programming problems without providing any of our own computational resources to the evaluation of fitness tests and without the awareness or consent of the owners of participating client machines.

2. DISTRIBUTED GENETIC PROGRAMMING SOFTWARE PACKAGES

A number of general distributed genetic programming systems have been developed. Compared to the technique we present in this paper, most of these systems offer a more full-featured genetic programming environment but also require greater client-side overhead to deploy.

One of the systems that is actively being developed is the Distributed Genetic Programming Framework, or DGPF [11]. DGPF is an open-source Java environment for distributing genetic programming runs over large networks. DGPF supports several different types of distributed computational models including peer-to-peer and client/server techniques. The DGPF framework can also be used to distribute other types of search algorithms aside from genetic programming, including simulated annealing or genetic algorithms more generally. Deploying the DGPF for problem solving requires the download and installation of a large software package on each client machine.

The system we present in this paper, which we call "unwitting distributed genetic programming," is not intended to supersede existing distributed GP frameworks. The main innovation of our system is a lightweight distributed fitness testing framework which differs from existing systems in a number of important ways:

- Client participation, including client-side software setup, occurs automatically and lasts only for the duration of a pageview — as little as a fraction of a second — during which time useful computation can be performed.
- Fitness testing can be performed without a user taking any special action, and even without the user's knowledge.
- No client-side software is required aside from a standard web browser.
- The fitness tests (and thus the problems to be solved) are served dynamically by the web server and may be changed at any time as needed.

The unwitting distributed genetic programming system we present could in fact be used to augment existing distributed GP systems by providing additional computational resources for performing fitness tests.

3. PREVIOUS USES OF JAVASCRIPT FOR EVOLUTIONARY COMPUTATION

As insightfully noted by user "foobar5892" in a JavaScript discussion on the website digg.com, "JavaScript is the Rodney Dangerfield of programming languages," in that it don't get no respect.[4] In spite of its ubiquity in web pages, JavaScript is rarely considered for serious programming projects. The authors of the present paper admit that they too gave JavaScript no respect before embarking on this project. They would also like to report that they now do have more respect for JavaScript, although they are still unlikely to consider it for large-scale projects, except when the ability to execute code in a web-browser is critical.

JavaScript has been used previously in evolutionary computation applications in situations where the ability to run interactively in a web-broswer was central to the design of the process [1, 3]. In these systems, JavaScript is used to enable interactive browser-based evolution by presenting an evolved images of snowflakes [3] or recordings of a synthesized voice [1] to a user who serves as a fitness evaluator the evolved media.

4. METHODS

4.1 Asynchronous JavaScript and XML (AJAX)

Once consisting of mostly static content, web pages have taken on new levels of interactivity thanks to a number of disparate but intertwined technologies and concepts such as JavaScript, Cascading Style Sheets (CSS), the Document Object Model (DOM) and the Extensible Markup Language (XML). Recently, the aggregate of these technologies, collectively known as Asynchronous JavaScript and XML, or AJAX, has gained particular traction in creating dynamic web applications. Although the component technologies of AJAX have been available for many years, AJAX techniques have recently gained popularity as a driving force behind so-called "Web 2.0" applications, in which sophisticated, interactive applications can be run inside of a web browser. Google's interactive mapping application, Google Maps, and in-browser word processor, Writely, both use AJAX to attain unprecedented interactivity for web pages.

The popularity of JavaScript and the dependance of many popular sites upon JavaScript presents a unique opportunity for individuals wishing to make use of low development overhead distributed computation without requiring client computers to install any software. If a user opts to allow JavaScript (in general this is more accurately stated, "if the user does not opt to disable JavaScript"), then JavaScript code can be executed automatically when a user loads a page, thus performing arbitrary (but secure — see the section on Ethics and Security below) computation on the client machine.

In addition to AJAX, a number of other browser-based technologies, such as Java and Flash, can also be used to

[4]`http://digg.com/programming/Should_Javascript_Get_More_Respect_2`

varying degrees to perform automatic client based computation. We chose JavaScript because it is a light-weight open standard, is fast-loading, and has no visual impact on the resulting web-page. Though the execution speed of JavaScript is a concern, we felt that it was the most appropriate choice for a prototype implementation of unwitting computation. Aside from JavaScript and other browser-based technologies we know of no other widely available, legal and consensual technique for performing client computations without requiring the client user to explicitly download and install client software.

4.2 Push

4.2.1 The Push3 Language

Push is a programming language intended primarily for use in evolutionary computation systems (such as genetic programming systems), as the language in which evolving programs are expressed [10, 7]. Push has an unusually simple syntax, which facilitates the development (or evolution) of mutation and recombination operators that generate and manipulate programs. Despite this simple syntax, Push provides more expressive power than most other program representations that are used for program evolution. Push programs can process multiple data types (without the syntax restrictions that usually accompany this capability), and they can express and make use of arbitrary control structures (e.g. recursive subroutines and macros) through the explicit manipulation of their own code (via "CODE" and "EXEC" data types). This allows Push to support the automatic evolution of modular program architectures in a particularly simple way, even when Push is employed in an otherwise ordinary genetic programming system (such as PushGP, which is a "generic" GP system except that it evolves Push programs rather than Lisp-style program trees). Push can also support entirely new evolutionary computation paradigms such as *autoconstructive evolution*, in which genetic operators and other components of the evolutionary system themselves evolve (as in the Pushpop and SwarmEvolve2 systems [4, 8]), although we use it only for standard genetic programming, and only using a minimal instruction set, in the work described here.

4.2.2 PushScript

PushScript is a JavaScript implementation of most of the Push3 specification [9] that was designed to run in web-broswers.[5] The entire implementation is contained in a 30k JavaScript file and can be dynamically loaded in a web-broswer with no noticeable difference in page load times.

Unlike most existing implementations of Push3, PushScript is not accompanied by a standalone genetic programming system. PushScript does not include functions for genetic operators or population management, and it does not provide any user interface or API for launching genetic programming runs. Instead, PushScript is designed only to run fitness tests provided by a network server.

PushScript does include support for all of the basic Push types, including CODE, and it can be extended via callbacks to other JavaScript functions. In conjunction with a server that provides programs to be evaluated, PushScript can be applied to any problem to which a regular Push implementation can be applied.

An interactive demo of the PushScript system is available online.[6]

4.2.3 Why Use Push for Unwitting Computation?

Compared to distributed computation systems that involve the installation of software on client machines, our web-based distributed computation is more limited in the choice of native GP program representation. Though any GP program representation is possible, the fitness tests must ultimately be evaluated in the system's native browser language, JavaScript in our case. Taking this into account, some of the requirements for our GP programming language representation are:

- Should be well-suited to evolutionary computation. For example, it should support simple mutation and crossover genetic operators that are guaranteed to produce syntactically valid programs. This requirement rules out most "human" programming languages, which typically combine infix and prefix notations and a variety of syntactic special cases for control structures. In particular, it rules out the use of JavaScript as the evolved language (which would otherwise be desirable due to the fact that it is the native language of our browser-based GP system).

- The programs to be fitness-tested can be efficiently represented as text. Text is the native format for interactions between the server and the clients in our framework. As a consequence, representing the programs as text facilitates both their transfer and their execution. Using binary data to represent the program would likely require additional data encoding and decoding steps for both the server and the client.

- The programs to be fitness-tested must be easy to parse and interpret in real-time by the browser. Unless the program is represented in a browser native language like JavaScript, the program will need to be parsed and executed in real-time.

- The choice of program representation should not place a high load on the web server. Because the system is designed to exploit high-traffic web pages, server load is an important consideration. This makes the idea of server-side translation from some genetic programming language to browser-runnable native JavaScript far less attractive.

Push was chosen for this project because its unusual combination of syntactic simplicity, representational power, and suitability for evolutionary computation better satisfied the above requirements than the available alternatives.

4.3 Client-Side Code

The client side code consists of the PushScript implementation described above, along with another JavaScript library of utility functions, pushfitnesstest.js, which handles server communication and execution of fitness tests. We refer to the entire process of running multiple fitness tests, including server communication of the programs and results,

[5]PushScript is currently missing a few of the less frequently used Push instructions.

[6]`http://www.spiderland.org/PushScript`

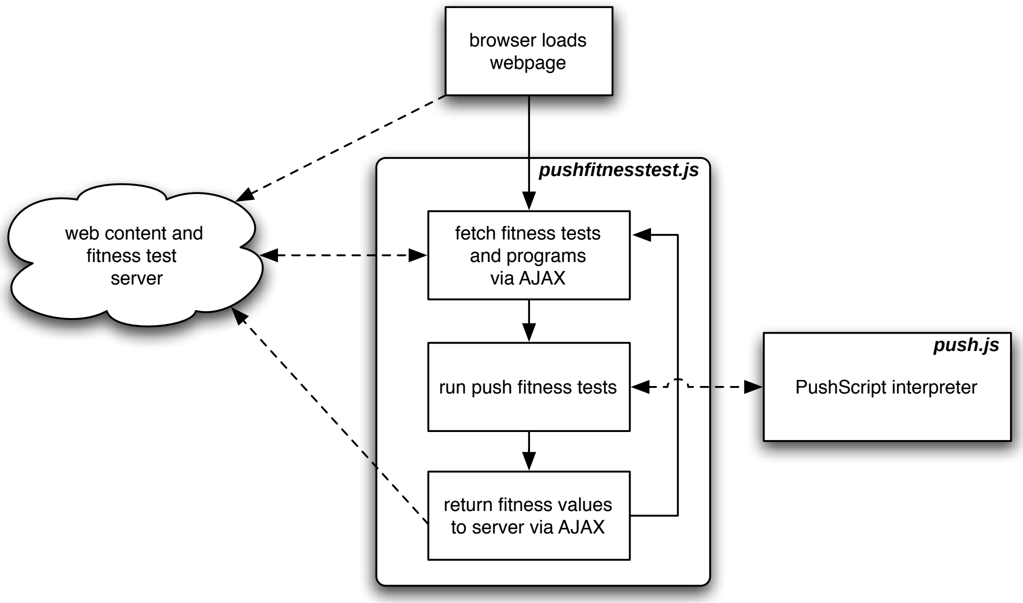
browser loads webpage
pushfitnesstest.js
web content and fitness test server
fetch fitness tests and programs via AJAX
run push fitness tests
return fitness values to server via AJAX
push.js
PushScript interpreter

Figure 1: Client-side process for genetic programming via unwitting distributed computation.

a Push fitness session. Each Push fitness session executes fitness tests for multiple programs, typically about 20 depending on server-side configuration. We adjust the number of fitness tests per session so that there is no noticable impact on page-load times.

The entire client-side process is described in the following steps:

1. User visits a website and a page is served from a content server.
2. Via the JavaScript "onload" handler, the page automatically initiates a Push fitness session upon page load.
3. The Push fitness session issues a fitness test request to the content server.
4. The webserver selects fitness cases and multiple programs for testing, and returns them to the client as XML.
5. The Push fitness session uses the PushScript interpreter to evaluate the fitness of the programs.
6. The Push fitness session returns fitness data, encoded as CGI parameters, to the content server.
7. The Push fitness session schedules a future session after a small delay (typically 5 seconds), effectively returning to step 3.

This process is illustrated in Figure 1.

In order to gracefully handle infamous incompatibilities between different web browsers, we use the open-source Ajax-Request library as a high-level API to access AJAX functionality on all browsers.[7] Our system is cross-platform and has been tested in popular browsers including Internet Explorer, Firefox and Safari.

[7]`http://www.ajaxtoolbox.com/request/`

4.4 Server-Side Code

Because the PushScript language handles only the distributed fitness test aspect of the genetic programming system, the remainder of the system is implemented using server-side code on a web server. In our implementation, the server-side is implemented as a series of PHP scripts that deal with web client interaction and interface with the C++ Push3 implementation that is integrated into the breve Simulation Environment [2][8] to handle population management and reproduction. Client interaction is handled via two PHP scripts, getprogram.php and reportfitness.php, which run on the server side to provide Push programs to clients and record fitness values, respectively. getprogram.php returns to the client XML data encoding a list of programs to be fitness-tested, as well as a series of input and desired output values. The number of programs sent for testing can be configured on the server side and can be adjusted for each problem to exploit as much client computation as possible without any perceivable change in page-load times. Fitness values are returned via an asynchronous HTTP page request to reportfitness.php, which reports an arbitrary number of fitness values encoded as CGI parameters.

Once an entire generation of individual programs has been evaluated and fitness values have been returned to the server, getprogram.php calls a breve simulation script, GPGenerateOffspring.tz, to generate offspring and prepare evaluation of the next generation.

4.5 Problems

We used symbolic regression problems to test our proof-of-concept implementation of unwitting distributed genetic programming via AJAX. We chose symbolic regression problems because they could be easily implemented in the current PushScript implementation and because they do not require any additional domain-specific Push instructions.

[8]`http://www.spiderland.org/breve`

Table 1: Problems and parameters used to demonstrate genetic programming with unwitting distributed computation.

Problems	1. $8*x*x*x+3*x*x+x$ 2. $x*x*x+x*x+x$ 3. $x*x*x-2*x*x-x$ 4. $x*x*x*x+x*x*x+x*x+x-8$ 5. $x*x*x*x*x*x-2*x*x*x*x+x*x-2$
Input (x) values	1-8
Fitness	sum of absolute value of errors
Crossover rate	40%
Fair mutation rate	40%
Deletion mutation rate	5%
Duplication rate	15%
Population size	2000
Maximum program size	50
Tournament size	7
Ephemeral random constants	integers from -10 to 10
Instruction set (Dec. 10 problems 1, 2 and 3)	`FLOAT.+, FLOAT.-, FLOAT.*, FLOAT./, FLOAT.POP, FLOAT.DUP FLOAT.SWAP, INPUT`
Instruction set (Dec. 10 problems 4 and 5, Jan. 15 all)	`INTEGER.+, INTEGER.-, INTEGER.*, INTEGER./, INTEGER.POP, INTEGER.DUP, INTEGER.SWAP, INPUT`

Table 2: Solutions to the test problems evolved with unwitting distributed computation.

Problem	Generation	Solution
1	32	`(2 INTEGER.* INTEGER.DUP INTEGER.DUP INTEGER.DUP INTEGER.* INTEGER.+ INTEGER.DUP INPUT INTEGER.- INTEGER.+ INTEGER.SWAP INTEGER.SWAP INTEGER.SWAP INPUT INTEGER.- INTEGER.* INPUT INTEGER.+)`
2	14	`(INPUT INTEGER.DUP INTEGER.* INTEGER.DUP INPUT INTEGER.* INTEGER.+ INTEGER.+)`
3	7	`(INPUT INPUT INTEGER.+ INTEGER.- INTEGER.DUP INTEGER.* INPUT INTEGER.- INPUT INTEGER.- 1 INTEGER.- INPUT INTEGER.*)`
4	49	`(-8 INPUT INPUT INTEGER.* INTEGER.DUP 9 INTEGER.+ INTEGER.POP INTEGER.DUP INPUT INTEGER.* INPUT INTEGER.DUP INTEGER.* INTEGER.DUP INTEGER.* INTEGER.+ INTEGER.+ INTEGER.+ INTEGER.+)`
5	91	`(INPUT INPUT INPUT INTEGER.* INTEGER.* INTEGER.- INTEGER.DUP INTEGER.* -1 INTEGER.+ -1 INTEGER.+)`

We used 5 simple symbolic regression problems which had been studied previously using another PushGP implementation [6], and which were therefore known to be readily solved in a fully functional PushGP system. We chose the 5 problems in [6] requiring the lowest computational effort. These problems had success rates of between 58% and 100% in the previously reported work. The problems and parameters we chose are described in Table 1.

We ran two sets of experiments on the website for the breve simulation environment itself, `spiderland.org`, with a single run of each symbolic regression problem in each set. The initial set of runs was conducted during the week of December 10th, 2006, coinciding with the release of version 2.5.1 of the breve simulation software package (which led to a large, temporary increase in traffic to the website). During these initial runs we tried both floating point and integer based instruction sets. The second set of runs was performed on `spiderland.org` during the week of January 15th, 2007 using the same integer instruction set for all problems.

5. RESULTS

The system was able to solve all of the problems listed in Table 1. In the first set of runs, problems 1 and 2 were solved on the first attempt, in generations 71 and 27 respectively. The remaining three problems all failed on the first attempt, but succeeded on the second attempt. The runs were manually restarted after 946, 196 and 431 generations respectively. The large variation in generation counts of the failed runs was due the runs being restarted manually after overnight (or longer) runs with varying levels of web traffic.

The duration of individual generations varied greatly with the level of traffic on the website. The shortest generation observed during actual problem solving was 1 minute and 17 seconds. In test runs during heavy traffic, we observed even shorter generation times of 30 seconds or lower. Table 3 shows the average and best generation times for each problem.

During the second run, the same integer-based instruction set was used for all problems. All problems were solved on the first attempt. Table 2 shows the problem solutions obtained during the second set of runs, after an automatic simplification process that removes unused code.[9]

6. DISCUSSION

6.1 Ethics and Security

The use of AJAX techniques has become so widespread that, from a practical and technical perspective, the unwitting distributed computation technique outlined here does not degrade the performance of client computers or disrupt the normal web-browsing experience in any unusual fashion. In fact, popular websites including Google, Amazon and Digg may load several hundreds or thousands of lines of JavaScript with each page load, depending on which pages the user is viewing. The use of AJAX has become an integral part of "Web 2.0," and one can argue that almost all of this computation is "unwitting" from the user's perspective, as most non-technical users are unlikely to be aware of its execution. Even by looking at the JavaScript code (which is often intentionally obfuscated in order to avoid comprehension), one often cannot determine the nature of the JavaScript computation.

[9]The syntactic simplicity of Push permits a particularly simple automatic simplification strategy; we iteratively remove random pieces of code, re-test for fitness, and retain the smaller program if it performs as well as the original.

Table 3: Generation times (minutes:seconds) observed during solution of the test problems by genetic programming with unwitting distributed computation.

	Problem				
	1	2	3	4	5
Best	1:17	1:53	2:07	2:43	3:35
Average	2:56	7:46	12:26	6:34	7:56

The difference between the uses mentioned above and the unwitting distributed computation technique outlined here is that, ostensibly, most sites use AJAX to enhance the user's experience on the site and not to perform unrelated computations. This distinction is not always clear, however, as JavaScript is often used for purposes of tracking user activity without any actual enhancement of the user experience. In the context of the current popular use of AJAX, unwitting distributed computation does not appear to be unusually exploitative. But it does raise ethical questions. We cannot explore such questions fully in this paper, but pending any definitive answers to these questions we suggest that developers of unwitting distributed computation systems disclose their use of the technique on their sites.

There are no specific security concerns for clients from the use of unwitting distributed computation. While the technique we present here does allow us to "steal" computation from client computers, there are no security implications to the technique beyond the normal security issues surrounding the general use of JavaScript. While JavaScript allows for arbitrary computation, it does so in a restricted environment, quarantined from the operating system and from the computer's resources. In particular, JavaScript does not allow filesystem access and greatly restricts network access.

On the other hand, our current implementation does have an obvious security vulnerability on the server side, since a malicious agent could easily spoof reportfitness.php, report bogus fitness values, and thereby lead the GP run astray. A variety of countermeasures might be deployed against such attacks, including lightweight forms of encryption, periodic verification, and banning of offending clients.

6.2 Execution Speed

The most significant factor affecting execution speed in the current proof of concept is the use of the JavaScript language, which is interpreted and does not have a strong reputation for speed. In an informal benchmark of arithmatic operations similar to those required for the execution of the PushScript language, JavaScript execution was far slower than C and showed enormous variation betwen browsers. On a MacBook Pro laptop, the benchmark executed in 0.044 seconds in C; and in JavaScript in 0.91 seconds in Firefox, 3.25 seconds in Camino, and 8.97 seconds in Safari. This is a slowdown of a factor of between 20.7 and 203.9 between C and the various JavaScript implementations.

Our tests were run on a relatively low traffic private webserver. In our implementation, and using our low-traffic web site, solving a simple symbolic regression problem requires several hundreds or thousands of page requests, typically over several hours.

The simple problems we chose can typically be solved in under 2 minutes (and often less than 30 seconds) each on a top-of-the-line desktop machine. The execution speed of our system, at least when deployed on a low-traffic website, is therefore nothing short of embarrassing as a symbolic regression problem solver, but it could be quite practical for other problems or in other circumstances. Some of the circumstances under which this approach may be more practical include:

- On web-sites with high numbers of page requests per minute. The benefits of the distributed approach increase with the number of pages served per minute. A popular web-site such as Google could employ unwitting distributed computation to perform incredible amounts of computation and solve real problems.

- On problems in which the required computation per fitness test is relatively large, on the order of at least a second per fitness test evaluation. For these problems, farming out fitness tests provides a clear advantage over local computation, although one must be careful not to use fitness tests that degrade user page load times unacceptably. On the highest traffic day, we recorded 151178 page requests to the reportfitness.php script, which is called to report results when a batch of fitness test completes. The local computation approach can perform the same number of fitness computations per day when the duration of a single fitness test is less than or equal to .57 seconds ($60 * 60 * 24$ seconds per day, divided by 151178 completed tests). Assuming a single fitness test per page request (in practice, we use a larger number of tests per request, typically 20), the computation of the distributed approach at this rate can surpass the local computation approach when the fitness test exceeds 0.57 seconds per test. Note that other variables, such as the number of tests per page request and the time duration between requests, can be adjusted to attempt to make the technique practical even with shorter fitness tests.

- On open-ended problems which will benefit from enormous populations and longer runs. The toy problems we solved in this project were selected specifically because we knew they could be solved with relatively small populations and in a relatively small number of generations. These problems have known answers and it is typically only a matter of time before they are found. Real problems which require larger overall numbers of fitness runs are more likely to benefit from distributed computation where additional fitness evaluations may directly improve the chances of finding a solution.

Although these conditions clearly do not apply to the proof-of-concept problems we described in this paper, we believe that these conditions actually do apply to many "real-world" genetic programming problems. One example of a problem which would benefit from this approach is our recent work on the evolution of programs for quantum computers [5]. For these problems, we ran large populations of individuals for several days on a local cluster of 23 CPUs. Finding a solution to these problems was not guaranteed: most of our runs resulted in failure. Work on problems of this scale and difficulty is more likely to benefit from additional fitness evaluations that could be provided by unwitting distributed computation.

6.3 Network Latency and Reliability

Provided that there is a legitimate benefit of the distributed approach to a specific problem as described above, the effects of network latency are negligible because fitness tests do not depend on the results of each other. The server can thus continuously farm out new fitness tests even when the previous tests have not finished running. In our current implementation when a client fails to return fitness values to the server the programs are simply assigned the worst possible fitness values, which practically speaking results in the programs being discarded and effectively a small loss of population size. Alternatively one could re-farm out the missing tests to additional clients.

6.4 Client-Side Overhead and "Free" Lunch

We feel that the most important benefit of this technique is the fact that there is no client-side software acquisition or configuration overhead. The required "software" (a JavaScript file) is automatically downloaded and maintained only for the duration of a pageview, after which it is effectively "uninstalled" automatically. This means that client users can opt to participate in a distributed process simply by visiting a website, and can opt-out of the process by closing the webpage.

The technique we present could be adapted for a larger audience through the creation of a personalized web portal (similar to Google or Yahoo personalized pages) where users could voluntarily participate in meaningful projects. Simply by clicking on provided links, users could enable or disable the client side computation, or could opt to switch from one distributed project to another. Because both the problems and the fitness test code are defined dynamically by the webserver, the problems or fitness testing framework may be changed dynamically on the server side without requiring any client side changes.

Although the technique presented here offers a relatively low level of computation compared to a dedicated client approach, the process is fully automatic with no client-side software installation or configuration overhead whatsoever. The low level of computation provided by each individual client can thus be offset by the simplicity of recruiting additional clients (either with or without their awareness of the process). Although the system here does not promise a "free lunch," it does allow for the consumption of small pieces of other peoples' "lunches" (whether with or without their consent) if they're not going to eat them.

7. FUTURE WORK

We envision three primary avenues for future work on this project:

- Expansion and generalization of PushScript and application to new problems. The most important future addition to this system is the addition of new frameworks for solving real-world problems. One of the main challenges in expanding the system for use with other problems is interfacing with domain-specific software frameworks, which are typically not written in JavaScript. By expanding the PushScript implementation and creating additional domain-specific frameworks in JavaScript, we will allow the system to be applied to a wider variety of real-world problems.

- Applications beyond genetic programming. The system we present here, including the PushScript implementation, provides a simple way to serve arbitrary computational tasks to client computers without the need to install client-side software, and (optionally) without the approval of the client user. Although the short-lived nature of the computational relationship makes this technique especially well suited to solving genetic programming problems, we believe that the computation could be harnessed for a number of other applications including Folding@home- and SETI@home-style distributed projects. Although the amount of computation that can be taken from a single host during a page-load is clearly miniscule in comparison to a dedicated computational client (such as a screensaver), a far greater number of hosts — all visitors to a website — can be used with the unwitting distributed computation approach.

- Further investigation of unwitting computation using different programming languages. The use of Java (and potentially other in-browser programming languages) should allow for far better performance than can be achieved through JavaScript, potentially rivaling the performance of native C code. Potential downsides include slower download and startup times for compiled code. We suspect that JavaScript may prove more effective for sites with very rapid page reloads, while Java may be more appropriate when the browser is left on a single page.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. National Science Foundation under Grant No. 0308540. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] J. Carlsson, C. Paiz, K. Wolff, and P. Nordin. Interactive evolution of speech using voiceXML speaking to you GP system. In N. Callaos, A. Pisarchik, and M. Ueda, editors, *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, volume VI, pages 58–62. IIIS, 2002.

[2] J. Klein. BREVE: a 3d environment for the simulation of decentralized systems and artificial life. In R. K. Standish, M. A. Bedau, and H. A. Abbass, editors, *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 329–334. The MIT Press, 2002. `http://www.spiderland.org/breve/breve-klein-alife2002.pdf`.

[3] W. B. Langdon. Global distributed evolution of L-systems fractals. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming, Proceedings of EuroGP'2004*, volume 3003 of *LNCS*, pages 349–358, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[4] L. Spector. Adaptive populations of endogenously diversifying Pushpop organisms are reliably diverse. In R. K. Standish, M. A. Bedau, and H. A. Abbass, editors, *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 142–145, University of New South Wales, Sydney, NSW, Australia, 9th-13th Dec. 2002. The MIT Press.

[5] L. Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach.* Kluwer Academic Publishers, Boston, 2004 (paperback pub. by Springer, 2007).

[6] L. Spector and J. Klein. Trivial geography in genetic programming. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 8, pages 109–123. Springer, Ann Arbor, 12-14 May 2005.

[7] L. Spector, J. Klein, and M. Keijzer. The Push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[8] L. Spector, J. Klein, C. Perry, and M. Feinstein. Emergence of collective behavior in evolving populations of flying agents. *Genetic Programming and Evolvable Machines*, 6(1):111–125, Mar. 2005.

[9] L. Spector, C. Perry, J. Klein, and M. Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA, 10 Sept. 2004.

[10] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[11] K. G. Thomas Weise. DGPF - an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In B. Filipič and J. Šilc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, International Conference on Bioinspired Optimization Methods and their Application (BIOMA), pages 157–166. Jožef Stefan Institute, Ljubljana, Slovenia, Oct 2006.

[12] M. Tomassini. Parallel and distributed evolutionary algorithms. In K. Miettinen, editor, *Evolutionary Algorithms in Engineering and Computer Science*, pages 113–133. John Wiley & Sons, Inc., New York, NY, USA, 1999.